Journal of Statistical Software

July 2022, Volume 103, Issue 4. doi: 10.18637/jss.v103.i04

scikit-mobility: A Python Library for the Analysis, Generation, and Risk Assessment of Mobility Data

Luca Pappalardo
ISTI-CNR

Filippo Simini
Argonne National Laboratory

Gianni Barlacchi
Amazon, Alexa AI

Roberto Pellungrini
University of Pisa

Abstract

The last decade has witnessed the emergence of massive mobility datasets, such as tracks generated by GPS devices, call detail records, and geo-tagged posts from social media platforms. These datasets have fostered a vast scientific production on various applications of 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. 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. In this paper, 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 **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 statistical patterns to assessing the privacy risk related to the analysis of mobility datasets.

Keywords: data science, human mobility, mobility analysis, spatio-temporal analysis, big data, network science, data mining, Python, mathematical modeling, migration models, privacy.

1. Introduction

The last decade has witnessed the emergence of massive datasets of digital traces that portray human movements at an unprecedented scale and detail. Examples include tracks generated by GPS devices embedded in personal smartphones (Zheng, Wang, Zhang, Xie, and Ma 2008), private vehicles (Pappalardo, Rinzivillo, Qu, Pedreschi, and Giannotti 2013) or boats (Fernandez Arguedas, Pallotta, and Vespe 2018); call detail records produced as a by-product of the communication between cellular phones and the mobile phone network (González, Hidalgo, and Barabási 2008; Barlacchi *et al.* 2015); geotagged posts from the most disparate social media platforms (Noulas, Scellato, Lambiotte, Pontil, and Mascolo 2012); even traces describing the sports activity of amateurs or professional athletes (Rossi, Pappalardo, Cintia, Iaia, Fernández, and Medina 2018). The availability of digital mobility data has attracted enormous interests from scientists of diverse disciplines, fueling advances in several applications, from computational health (Tizzoni *et al.* 2012; Barlacchi, Perentis, Mehrotra, Musolesi, and Lepri 2017) to the estimation of air pollution (Nyhan, Kloog, Britter, Ratti, and Koutrakis 2018; Bohm, Nanni, and Pappalardo 2021), from the design of recommender systems (Wang, Pedreschi, Song, Giannotti, and Barabasi 2011) to the optimization of mobile and wireless networks (Karamshuk, Boldrini, Conti, and Passarella 2011; Tomasini, Mahmood, Zambonelli, Brayner, and Menezes 2017), from transportation engineering and urban planning (Zhao, Tarkoma, Liu, and Vo 2016) to the estimation of migratory flows (Simini, González, Maritan, and Barabási 2012; Ahmed *et al.* 2016) and people's place of residence (Pappalardo, Ferres, Sacasa, Cattuto, and Bravo 2021), from the well-being status of municipalities, regions, and countries (Pappalardo, Vanhoof, Gabrielli, Smoreda, Pedreschi, and Giannotti 2016b; Voukelatou *et al.* 2020) to the prediction of traffic and future displacements (Zhang, Zheng, and Qi 2017; Rossi, Barlacchi, Bianchini, and Lepri 2019).

It is hence not surprising that the last decade has also witnessed a vast scientific production on various aspects of human mobility (Luca, Barlacchi, Lepri, and Pappalardo 2021; Wang, Kong, Xia, and Sun 2019; Blondel, Decuyper, and Krings 2015; Barbosa *et al.* 2018). The first strand of literature addresses data pre-processing 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 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, Qu, Blumm, and Barabási 2010b); the presence of the returners and explorers dichotomy (Pappalardo, Simini, Rinzivillo, Pedreschi, Giannotti, and Barabási 2015); a conservative quantity in the number of locations actively visited by individuals (Alessandretti, Sapiezynski, Sekara, Lehmann, and Baronchelli 2018), and more (Barbosa *et al.* 2018; Luca *et al.* 2021). 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., models that can generate synthetic trajectories able to reproduce, realistically, the laws of human mobility. A class of algorithms aim to reproduce spatial properties of mobility (Song, Koren, Wang,

and Barabási 2010a; Pappalardo, Rinzivillo, and Simini 2016a); another one focuses on the accurate representation of the time-varying behavior of individuals (Barbosa, de Lima-Neto, Evsukoff, and Menezes 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, Yang, Gupta, Veneziano, Athavale, and González 2016; Luca *et al.* 2021). 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, Hidalgo, Verleysen, and Blondel 2013; Fiore *et al.* 2020). 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 digital data in general (Monreale, Rinzivillo, Pratesi, Giannotti, and Pedreschi 2014) and mobility records in particular (Pellungrini, Pappalardo, Pratesi, and Monreale 2017; Pellungrini, Pappalardo, Simini, and Monreale 2022; de Montjoye *et al.* 2013, 2018).

Despite the increasing importance of mobility analysis for many scientific and industrial domains, there is no statistical software that can support scientists and practitioners with all the aspects of mobility analysis mentioned above (Section 9).

To fill this gap, we propose **scikit-mobility** (Pappalardo, Simini, Barlacchi, and Pellungrini 2022), a Python (Van Rossum *et al.* 2011) library that has the ambition of providing scientists and practitioners with an environment to reproduce existing research and perform analysis of 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 ('`TrajDataFrame`' and '`FlowDataFrame`') based on the standard Python libraries **numpy** (Oliphant 2006), **pandas** (McKinney 2010) and **geopandas** (Jordahl *et al.* 2019) (Section 2), as well as to visualize trajectories and flows on interactive maps based on the Python libraries **folium** (Fernandes *et al.* 2019) and **matplotlib** (Hunter 2007) (Section 4).

2. Clean and pre-process 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 mobility patterns both at the individual and at the collective level (Section 5), such as the computation of travel and characteristic distances, object and location entropies, location frequencies, waiting times, origin-destination matrices, and more.

4. Run the most popular mechanistic generative models to simulate individual mobility, such as the exploration and preferential return model (EPR) 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 dataset through the simulation of the re-identification risk associated with a vast repertoire of privacy attacks (Section 8).

Next-location prediction, i.e., predicting the next location(s) an individual will visit given their mobility history (Luca *et al.* 2021; Wu, Luo, Shao, Tian, and Peng 2018), is a relevant mobility-related task not covered in the current version of **scikit-mobility**. We plan to include location prediction algorithms in future versions of the library.

Note that, while **scikit-mobility** has been conceived for human movement analysis and the **privacy** module makes sense for human mobility data only, most features can be applied to other types of mobility (e.g., boats, animal movements, boat trips). **scikit-mobility** is designed to deal with spatiotemporal trajectories and mobility flows and functions to deal with other types of mobility-related data, such as accelerometer data from wearable devices are not currently covered in this library.

The methods currently implemented have been chosen by the authors based mostly on their expertise and are not meant to be exhaustive. In future library releases, we plan to expand the range of methods and models.

scikit-mobility is publicly available on GitHub at: `https://scikit-mobility.github.io/scikit-mobility/`. Tutorials on how to use the library for mobility analysis are available at: `https://github.com/scikit-mobility/tutorials`. 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** (McKinney 2010). Thus, both '`TrajDataFrame`' and '`FlowDataFrame`' inherit all the functionalities provided by the '`DataFrame`' and all the efficient optimizations for reading and writing tabular data (e.g., mobility datasets). This choice allows broad compatibility of **scikit-mobility** with other Python libraries and machine learning tools, such as **scikit-learn** (Pedregosa *et al.* 2011).

The current version of the library is designed to work with the latitude and longitude system (`epsg:4326`), the most used in practical scenarios of mobility analysis. Therefore, the Haversine formula is the default when the library's functions compute distances. We plan to extend the library to deal with other reference systems, even user-defined ones. This extension would imply associating a custom distance function to a reference system.

2.1. Trajectory

Mobility data describe the movements of a set of moving objects during a period of observation. The objects may represent individuals (González *et al.* 2008), animals (Ramos-Fernández, Mateos, Miramontes, Cocho, Larralde, and Ayala-Orozco 2004), private vehicles (Pappalardo *et al.* 2015), boats (Fernandez Arguedas *et al.* 2018), or 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, so-

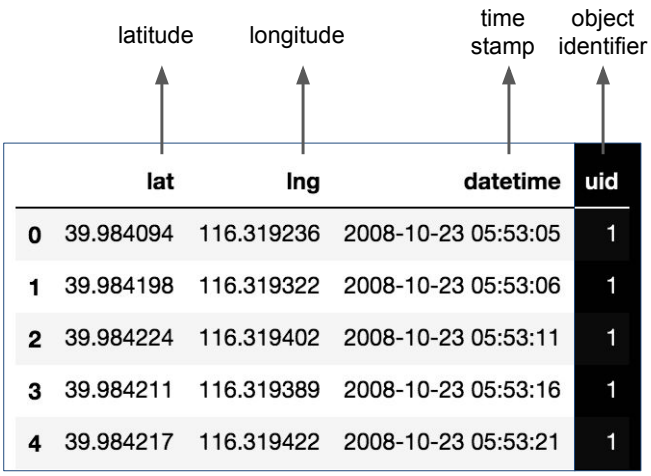

Figure 1: Representation of a '`TrajDataFrame`' object. Each row represents a point of a moving object's trajectory, described by three mandatory columns (`lat`, `lng`, `datetime`) and by optional columns `uid` and `tid`, indicating the identifier of the moving object associated with the point and the trajectory identifier, respectively.

cial networking platforms, video cameras) and stored as trajectories, a temporally ordered sequence of spatiotemporal points where an object stopped in or went through. In the literature of mobility analytics, a trajectory is often formally defined as follows (Luca *et al.* 2021; Zheng, Capra, Wolfson, and Yang 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 is `uid`: it identifies the object associated with the trajectory point and can be of any type (string, int, or float). If `uid` is not present, **scikit-mobility** assumes that the '`TrajDataFrame`' contains trajectories associated with a single moving object. The second one is `tid` (any type) and specifies the trajectory's identifier to which the point belongs. If `tid` is not present, **scikit-mobility** assumes that all the rows in the '`TrajDataFrame`' associated with a `uid` belong to the same trajectory. Note that, besides the mandatory columns, the user can add to a '`TrajDataFrame`' as many columns as they want since the data structures in **scikit-mobility** inherit all the **pandas** '`DataFrame`' functionalities.

Each '`TrajDataFrame`' object 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 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** '`DataFrame`'s). To load a '`TrajDataFrame`' from a file, we first import the library:

```
>>> import skmob
```

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

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

Note that the values corresponding to the `lat`, `lng`, and `datetime` columns must be necessarily float, float and datetime, respectively, otherwise the library raises an exception.[1]

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 file's path.

```
>>> print(tdf.crs)
```

```
{'init': 'epsg:4326'}
```

```
>>> print(tdf.parameters)
```

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

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

```
>>> 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
```

[1] The `TrajDataFrame` constructor forces the conversion of the three mandatory columns' values to the preset types. If the conversion fails, it raises an exception. For example, the constructor can successfully convert string `"39.1432"` to float 39.1432, but it cannot convert (hence raising an exception) string `"39.2ui2"`.

2.2. Flows

Origin-destination matrices, aka flows, are another common representation of mobility data. While trajectories refer to single objects' movements, 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`' with 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` (any type), `destination` (any type) and `flow` (type: integer). The user can add to a '`FlowDataFrame`' as many columns as they want.

In mobility tasks, the territory is often discretized by mapping the coordinates to a spatial tessellation, i.e., a covering of the bi-dimensional space using a countable number of geometric shapes (e.g., squares, hexagons) called tiles, with no overlaps and no gaps. For instance, for the analysis or prediction of mobility flows, a spatial tessellation is used to aggregate flows of people moving among locations (the tiles of the tessellation). For this reason, each '`FlowDataFrame`' is associated with a spatial tessellation, a **geopandas** '`GeoDataFrame`' that contains two mandatory columns: `tile_ID` (any type) indicates the identifier of a location; `geometry` indicates the geometric shape that describes the location on a territory (e.g., a square, a hexagon, the shape of a neighborhood).[2] Note that each location identifier in the `origin` and `destination` columns of a '`FlowDataFrame`' must be present in the associated spatial tessellation. Otherwise, the library raises an exception. Similarly, **scikit-mobility** raises an exception if the type of the `origin` and `destination` columns in the '`FlowDataFrame`' and the type of the `tile_ID` column in the associated tessellation are different.

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

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

Then, we load the spatial tessellation and the '`FlowDataFrame`' using the `from_file` method of the classes '`GeoDataFrame`' and '`TrajDataFrame`', respectively. Note that `from_file` requires to specify the associated spatial tessellation through the `tessellation` argument.

```
>>> tessellation = gpd.GeoDataFrame.from_file('NY_counties_2011.geojson')
>>> fdf = skmob.FlowDataFrame.from_file('NY_commuting_flows_2011.csv', \
...   tessellation = tessellation, tile_id = 'tile_id')
```

The spatial tessellation and '`FlowDataFrame`' have the structure shown below.

[2]Since a tessellation is a **geopandas** '`GeoDataFrame`', it supports any geometry (e.g., '`Polygon`', '`Point`'). However, the point geometry should be avoided because it does not correctly represent a tessellation tile. In general, '`Polygon`' and '`MultiPolygon`' shapes should be preferred to describe the tiles.

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

```
tile_id    population                                          geometry
0   36019       81716  POLYGON ((-74.00667 44.88602, -74.02739 44.995...
1   36101       99145  POLYGON ((-77.09975 42.27421, -77.09966 42.272...
2   36107       50872  POLYGON ((-76.25015 42.29668, -76.24914 42.302...
3   36059     1346176  POLYGON ((-73.70766 40.72783, -73.70027 40.739...
4   36011       79693  POLYGON ((-76.27907 42.78587, -76.27535 42.780...
```

```
>>> 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 pre-processing

As any analytical process, mobility data analysis requires data cleaning and pre-processing steps (Zheng 2015). The **preprocessing** module allows the user to perform noise filtering, stop detection, and trajectory compression. Note that if a '`TrajDataFrame`' contains multiple trajectories from multiple objects, the pre-processing methods automatically apply to the single trajectory and, when necessary, to the single object. Table 1 lists the available methods for trajectory pre-processing.

3.1. Noise filtering

Trajectory data are generally noisy, usually because of recording errors like poor signal reception. When the error associated with the coordinates of points is large, the best solution is to filter out these points. In **scikit-mobility**, the 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:

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

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

```
>>> tdf = skmob.TrajDataFrame.from_file('geolife_sample.txt.gz')
>>> print('Number of points in tdf: %d\n' %len(tdf))
>>> print(tdf.head())
```

Method	Description	Arguments
`clustering.cluster`	Cluster the stay locations of each object in a 'TrajDataFrame' using DBSCAN (Hariharan and Toyama 2004).	'TrajDataFrame' object, `cluster_radius_km = 0.1`, `min_samples = 1`
`compression.compress`	Reduce the number of points of each object in a 'TrajDataFrame' with median coordinates within a radius (Zheng 2015).	'TrajDataFrame' object, `spatial_radius_km = 0.2`
`detection.stay_locations`	Detect the stay locations (stops) for each object in a 'TrajDataFrame' with a time threshold (Hariharan and Toyama 2004; Zheng 2015).	'TrajDataFrame' object, `stop_radius_factor = 0.5`, `minutes_for_a_stop = 20.0`, `spatial_radius_km = 0.2`, `leaving_time = True`, `no_data_for_minutes = 1e12`, `min_speed_kmh = None`
`filtering.filter`	For each object, filter out the noise or outlier points (Zheng 2015).	'TrajDataFrame' object, `max_speed_kmh = 500.0`, `include_loops = False`, `speed_kmh = 5.0`, `max_loop = 6`, `ratio_max = 0.25`

Table 1: Trajectory pre-processing methods currently implemented in **scikit-mobility**. For each method, we provide a description, its arguments and their default values (if any).

```
Number of points: 217653

         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

>>> ftdf = preprocessing.filtering.filter(tdf, max_speed_kmh = 10.)
>>> print('Number of points in ftdf: %d' %len(ftdf))
>>> print('Number of filtered points: %d\n' %(len(tdf) - len(ftdf)))
>>> print(ftdf.head())

Number of points in ftdf: 108779
Number of filtered points: 108874

         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, 108,874 points out of 217,653 are filtered out. The intensity of the filter is controlled by the `max_speed` parameter. The lower the value, the more intense the filter is.

3.2. Stop detection

Some points in a trajectory can represent points-of-interest (POIs) such as schools, restaurants, and bars or represent individual-specific places such as home and work locations. These points are usually called stay locations 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 `stay_locations` function, contained in the **detection** module, finds the stay points visited by an object. For instance, to identify the stay locations where the object spent at least `minutes_for_a_stop` minutes within a distance `spatial_radius_km` × `stop_radius_factor`, from a given point, we can use the following code:

```
>>> from skmob.preprocessing import detection
>>> stdf = detection.stay_locations(tdf, stop_radius_factor = 0.5, \
...   minutes_for_a_stop = 20.0, spatial_radius_km = 0.2)
>>> print(stdf.head())
```

```
         lat         lng            datetime  uid    leaving_datetime
0  39.978253  116.327275 2008-10-23 06:01:05    1 2008-10-23 10:32:53
1  40.013819  116.306532 2008-10-23 11:10:09    1 2008-10-23 23:46:02
2  39.978950  116.326439 2008-10-24 00:12:30    1 2008-10-24 01:48:57
3  39.981316  116.310181 2008-10-24 01:56:47    1 2008-10-24 02:28:19
4  39.981451  116.309505 2008-10-24 02:28:19    1 2008-10-24 03:18:23
```

As shown in the code snippet, a new column `leaving_datetime` is added to the '`TrajDataFrame`' object to indicate the time when the moving object left the stop location.

3.3. Trajectory compression

Trajectory compression aims to reduce the number of trajectory points while preserving the trajectory structure. This step is generally applied right after the stop detection step, significantly reducing the number of trajectory points. In **scikit-mobility**, we can use the method `compression.compress` under the **preprocessing** module. For instance, to merge all the points closer than 0.2km from each other, we can use the following code:

```
>>> from skmob.preprocessing import compression
>>> 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
```

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

         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
4  40.016006  116.306444 2008-10-23 11:08:07    1
```

Once compressed, the trajectory presents a smaller number of points, allowing easy plotting 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 datasets, 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 library **folium** (Fernandes *et al.* 2019). The choice of **folium** is motivated by the fact that, given the complexity of mobility data, the user may need to zoom in/out and interact with the components of trajectories, flows, and tessellations. This interaction would be impossible with static plotting libraries, such as **matplotlib** (Hunter 2007). The user can save an interactive plot in an HTML file or take a screenshot to save it as a PNG file. Table 2 lists the plotting functions available in the library.

4.1. Visualizing trajectories

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

Plot trajectories

The `plot_trajectory` method for '`TrajDataFrame`' objects 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 '`TrajDataFrame`'s 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 and the type of map tiles to use. The user can also plot markers denoting the trajectory's starting and ending points.

The `plot_trajectory` method, as well as all the other plotting methods, return an object of class '`folium.Map`', which can be used by other **folium** and **scikit-mobility** functions to visualize additional data on the same map. A '`folium.Map`' object can be passed to a plotting

Method	Description	Arguments
`plot_diary`	Plot a single moving object's mobility diary.	`uid`, `start_datetime = None`, `end_datetime = None`, `ax = None`, `legend = False`
`plot_stops`	Plot the stops (stay locations) in the '`TrajDataFrame`' on a **folium** map.	`map_f = None`, `max_users = 10`, `tiles = 'cartodbpositron'`, `zoom = 12`, `hex_color = -1`, `opacity = 0.3`, `radius = 12`, `number_of_sides = 4`, `popup = True`
`plot_trajectory`	Plot the trajectories on a **folium** map.	`map_f = None`, `max_users = 10`, `max_points = 1000`, `style_function = <function <lambda>>`, `tiles = 'cartodbpositron'`, `zoom = 12`, `hex_color = -1`, `weight = 2`, `opacity = 0.75`, `dashArray = '0, 0'`, `start_end_markers = True`
`plot_tessellation`	Plot the spatial tessellation on a **folium** map.	`map_f = None`, `maxitems = -1`, `style_func_args = ,` `popup_features = ['tile_ID']`, `tiles = 'Stamen Toner'`, `zoom = 6`, `geom_col = 'geometry'`
`plot_flows`	Plot the flows of a '`FlowDataFrame`' on a **folium** map.	`map_f = None`, `min_flow = 0`, `tiles = 'Stamen Toner'`, `zoom = 6`, `flow_color = 'red'`, `opacity = 0.5`, `flow_weight = 5`, `flow_exp = 0.5`, `style_function = <function <lambda>>`, `flow_popup = False`, `num_od_popup = 5`, `tile_popup = True`, `radius_origin_point = 5`, `color_origin_point = '#3186cc'`

Table 2: Plotting methods currently implemented in **scikit-mobility**. For each method, we provide a description, its arguments and their default values (if any).

method via the argument `map_f` (default: `None`, i.e., the mobility data are plotted on a new map).

The following code generates a plot by the `plot_trajectory` method (see Figure 2):

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

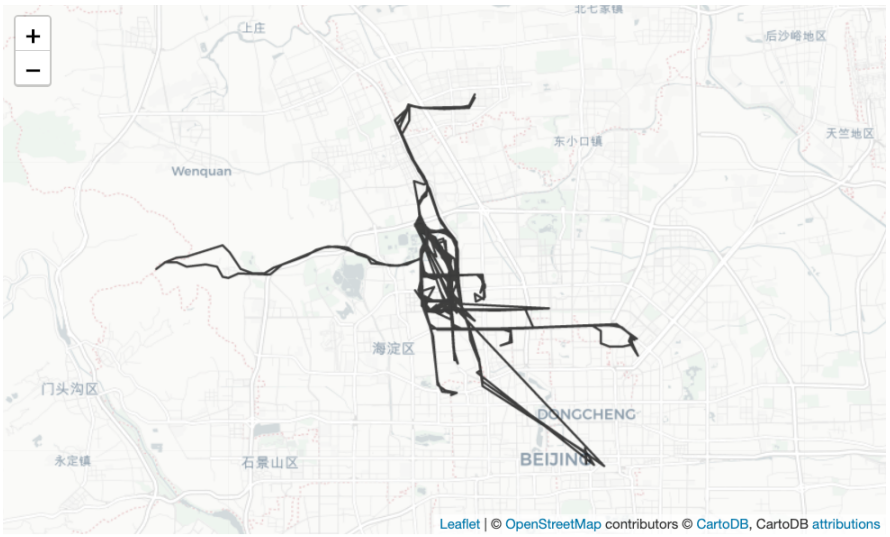

Figure 2: Example of a plot generated by the `plot_trajectory` method for '`TrajDataFrame`' objects.

Note that if trajectories represent abstract mobility, such as movements extracted from social media posts or mobile phone calls, straight lines may appear that do not consider walls, buildings, and similar structures on the road network.

By default, a '`TrajDataFrame`' object represents the full mobility of a set of individuals, i.e., covering the entire observation period (e.g., one month). The user can split the trajectory of an individual using pre-processing functions, such as the `detection.stay_locations` function (Section 3), and then split the whole trajectory into sub-trajectories, adding a proper column to identify them (i.e., the `tid` column). At this point, the user may visualize the portion of the '`TrajDataFrame`' selecting for values of the created column.

Plot stops

The `plot_stops` method for '`TrajDataFrame`' objects plots the stay locations as markers on a map. This method requires a '`TrajDataFrame`' object with a `constants.LEAVING_DATETIME` column, which is created by the **scikit-mobility** functions to detect stay locations (see Section 3). The argument `max_users` (type: int, default: 10) limits the number of objects whose stay locations should be plotted. The plot style can be customized via arguments to specify the color, radius, opacity of the markers, and the type of 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 stay location's popup window includes coordinates, `uid`, arrival, and leaving times.

We show below the code for generating a plot with the `plot_stops` method (see Figure 3). Note that if the `cluster` column is present in the '`TrajDataFrame`' object, as it happens when the `cluster` method is applied, the stay locations are automatically colored according to the value of that column (to identify different clusters of stay locations).

```
>>> from skmob.preprocessing import detection, clustering
>>> tdf = skmob.TrajDataFrame.from_file('geolife_sample.txt.gz')
```

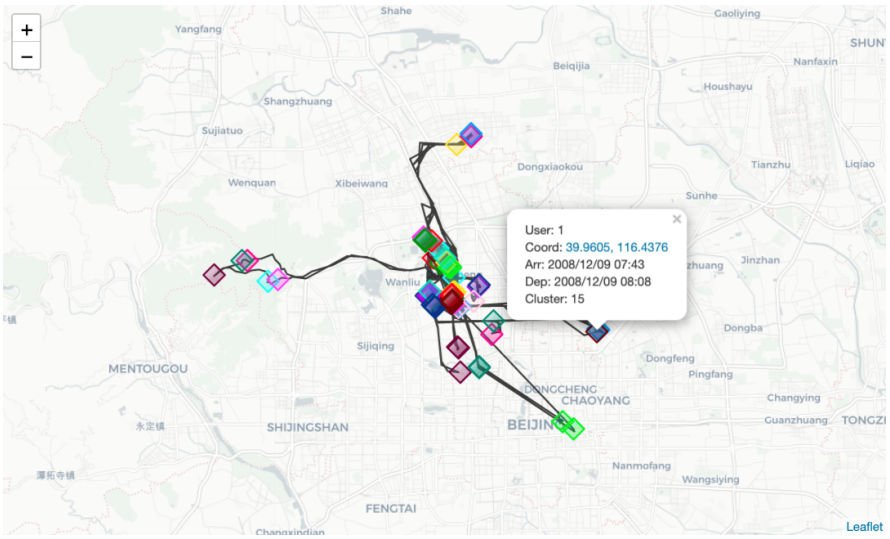

Figure 3: Example of a plot generated by the `plot_stops` method for '`TrajDataFrame`' objects.

```
>>> stdf = detection.stay_locations(tdf)
>>> cstdf = clustering.cluster(stdf)
>>> cstdf.plot_stops(max_users = 1, map_f = map_f)
```

Plot diary

The `plot_diary` method for '`TrajDataFrame`' objects plots the time series of the locations visited by an object. If the column `uid` is present, one object identifier 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 stay locations (see Section 3).

The plot displays time on the `x` axis and shows a series of rectangles of different colors representing the object's visits to the various stay locations. The length of a rectangle denotes the visit's duration: 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 stay locations 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 the code to generate a plot by the `plot_diary` method (see Figure 4):

```
>>> cstdf.plot_diary(user = 1, legend = True)
```

The user can compare multiple moving objects by plotting their diaries next to each other (see Figure 5):

```
>>> ax = cstdf.plot_diary(1)
>>> ax = cstdf.plot_diary(5, legend = True)
```

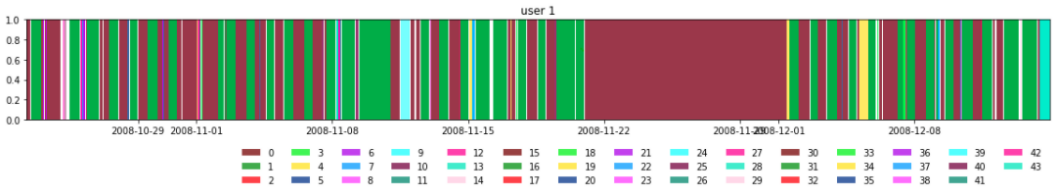

Figure 4: Example of a plot generated by the `plot_diary` method for '`TrajDataFrame`' objects.

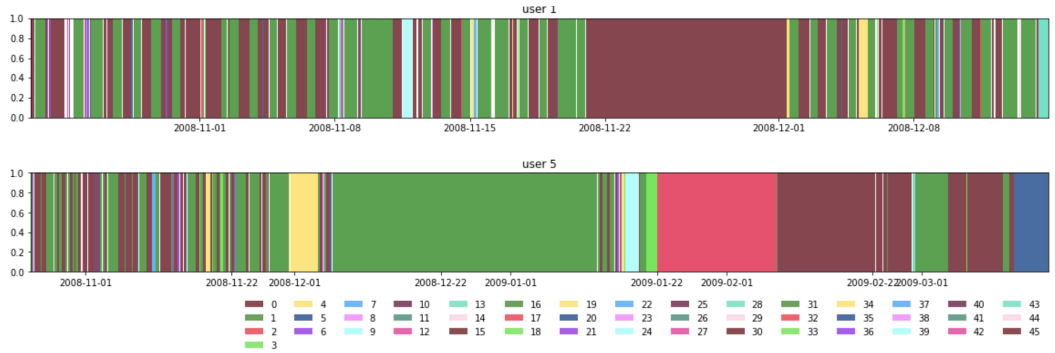

Figure 5: Example of two plots next to each other generated by the `plot_diary` method for '`TrajDataFrame`' objects.

4.2. Visualizing flows

A '`FlowDataFrame`' object 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.

Plot tessellation

The `plot_tessellation` method for '`FlowDataFrame`' objects 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, opacity of the tiles, and the type of map tiles to use. The argument `popup_features` (type: list, default: `[constants.TILE_ID]`) allows enhancing 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.

We show below the code to generate a plot by the `plot_tessellation` method (see Figure 6):

```
>>> import geopandas as gpd
>>> from skmob import FlowDataFrame
>>> tessellation = gpd.GeoDataFrame.from_file('NY_counties_2011.geojson')
>>> fdf = FlowDataFrame.from_file('NY_commuting_flows_2011.csv', \
...   tessellation = tessellation, tile_id = 'tile_id')
```

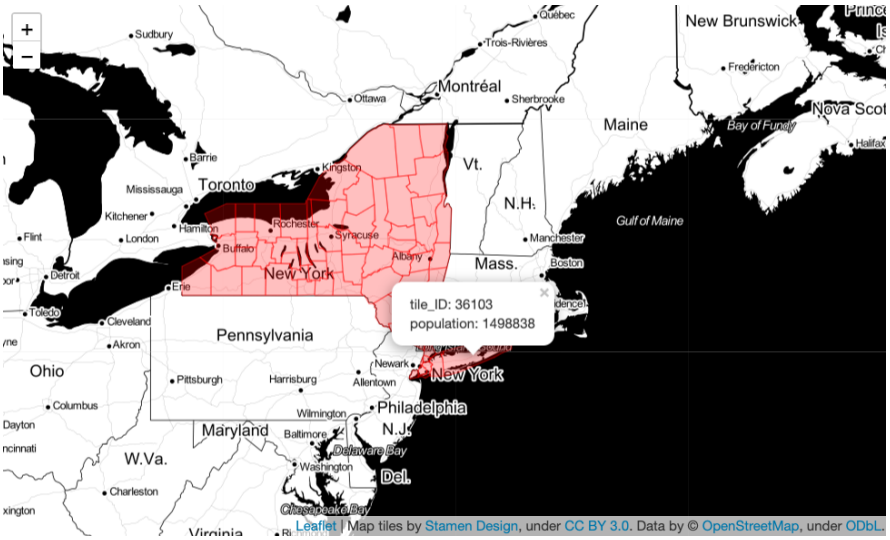

Figure 6: Example of a plot generated by the `plot_tessalation` method for '`FlowDataFrame`' objects.

```
>>> fdf.plot_tessellation(popup_features = ['tile_id', 'population'], \
...   style_func_args = {'fillColor': 'red', 'color':'red'})
```

Plot flows

The `plot_flows` method for '`FlowDataFrame`' objects plots the flows on a geographic map as lines between the centroids of the tiles in the tessellation of the '`FlowDataFrame`' object. Large '`FlowDataFrame`'s 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, opacity of the flow lines, and the type of map tiles to use. The arguments `flow_popup` and `tile_popup` allow enhancing 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.

We show below the code to generate a plot with the `plot_flows` method (see Figure 7):

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

The user can also visualize the tessellation and the flows in the same plot (see Figure 8):

```
>>> map_f = fdf.plot_tessellation(popup_features = ['tile_id', \
...   'population'], style_func_args = {'color':'red', 'fillColor': 'red'})
>>> fdf.plot_flows(map_f = map_f, min_flow = 50)
```

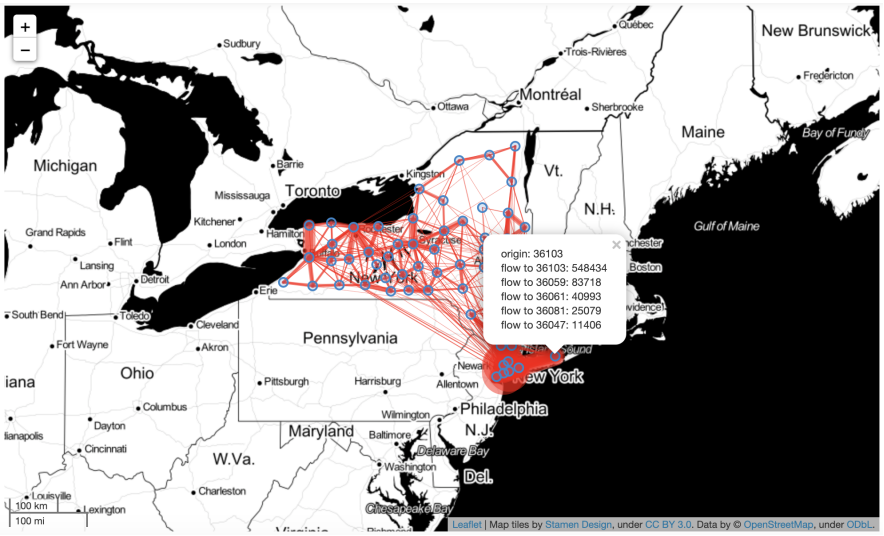

Figure 7: Example of a plot generated by the `plot_flows` method for '`FlowDataFrame`' objects.

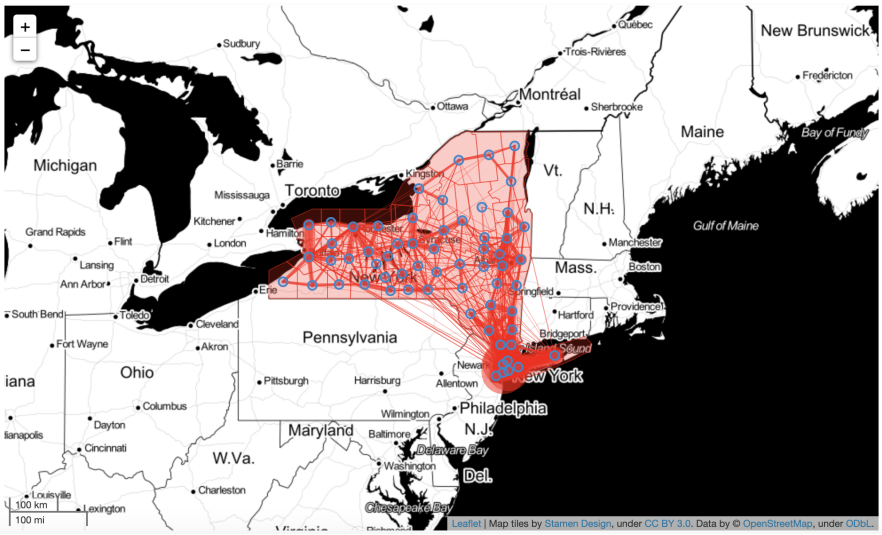

Figure 8: Example of a plot combining the output of the `plot_tessellation` and the `plot_flows` method for '`FlowDataFrame`' objects.

5. Mobility measures

In the last decade, several measures have been proposed to capture human mobility patterns, both at the individual and collective levels (Barbosa *et al.* 2018; Luca *et al.* 2021). Individual measures summarize the mobility patterns of a single moving object, while collective measures summarize the 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. Several measures inspired by the Shannon entropy have been proposed to quantify the predictability of an individual's movements (Song *et al.* 2010b).

Individual measure	Description	Arguments
`radius_of_gyration`	Characteristic distance (in km) traveled by an individual (González *et al.* 2008).	'`TrajDataFrame`' object, `show_progress = True`
`k_radius_of_gyration`	Characteristic distance (in km) traveled by an individual between their k most frequent locations (Pappalardo *et al.* 2015).	'`TrajDataFrame`' object, `k = 2`, `show_progress = True`
`random_entropy`	Degree of predictability of an individual's whereabouts if each location is visited with equal probability (Song *et al.* 2010b).	'`TrajDataFrame`' object, `show_progress = True`
`uncorrelated_entropy`	Historical probability that a location was visited by an individual (Song *et al.* 2010b).	'`TrajDataFrame`' object, `normalize = False`, `show_progress = True`
`real_entropy`	Mobility entropy of an individual considering also the order in which locations were visited (Song *et al.* 2010b).	'`TrajDataFrame`' object, `show_progress = True`
`jump_length`	Distances (in km) traveled by an individual (Brockmann, Hufnagel, and Geisel 2006).	'`TrajDataFrame`' object, `show_progress = True`, `merge = False`
`maximum_distance`	Maximum distance (in km) traveled by an individual (Williams, Thomas, Dunbar, Eagle, and Dobra 2015).	'`TrajDataFrame`' object, `show_progress = True`
`distance_straight_line`	Sum of the distances (in km) traveled by an individual (Williams *et al.* 2015).	'`TrajDataFrame`' object, `show_progress = True`
`waiting_times`	Inter-times (in s) between the movements of an individual (Song *et al.* 2010a).	'`TrajDataFrame`' object, `show_progress = True`, `merge = False`

Table 3: Individual measures currently implemented in **scikit-mobility** (part 1). For each method, we provide a description, its arguments and their default values (if any).

scikit-mobility provides a wide set of mobility measures, each implemented as a function that takes as input a '`TrajDataFrame`' and outputs a **pandas** '`DataFrame`'. In the modules **skmob.measure.individual** and **skmob.measures.collective**, individual and collective measures are implemented, respectively. Tables 3 and 4 list the available individual measures; Table 5 lists the available collective measures.

The code below computes the distances traveled by the objects and their radius of gyration.

Individual measure	Description	Arguments
`number_of_locations`	Number of distinct locations visited by an individual.	'TrajDataFrame' object, `show_progress = True`
`home_location`	Location (lat, long pair) most visited by an individual during nighttime (Phithakkitnukoon, Smoreda, and Olivier 2012).	'TrajDataFrame' object, `start_night = '22:00'`, `end_night = '07:00'`, `show_progress = True`
`max_distance_from_home`	Maximum distance (in km) from home traveled by an individual (Canzian and Musolesi 2015).	'TrajDataFrame' object, `start_night = '22:00'`, `end_night = '07:00'`, `show_progress = True`
`number_of_visits`	Number of visits to any location by an individual.	'TrajDataFrame' object, `show_progress = True`
`location_frequency`	Visitation frequency of each location of an individual (Song *et al.* 2010a).	'TrajDataFrame' object, `normalize = True`, `as_ranks = False`, `show_progress = True`, `location_columns = ['lat', 'lng']`
`individual_mobility_network`	Mobility network of an individual (Bagrow and Lin 2012; Rinzivillo, Gabrielli, Nanni, Pappalardo, Pedreschi, and Giannotti 2014).	'TrajDataFrame' object, `self_loops = False`, `show_progress = True`
`recency_rank`	Recency rank of the locations of an individual (Barbosa *et al.* 2015).	'TrajDataFrame' object, `show_progress = True`
`frequency_rank`	Frequency rank of the locations of an individual (Barbosa *et al.* 2015).	'TrajDataFrame' object, `show_progress = True`

Table 4: Individual measures currently implemented in **scikit-mobility** (part 2). For each method, we provide a description, its arguments and their default values (if any).

First, we import the two functions from the library.

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

We then download a 'TrajDataFrame' from the web, describing all checkins made through the social network platform Brightkite:

```
>>> url = \
...   'https://snap.stanford.edu/data/loc-brightkite_totalCheckins.txt.gz'
>>> import pandas as pd
```

Collective measure	Description	Arguments
`random_location_entropy`	Random entropy of locations with respect to individual visits.	'TrajDataFrame' object, `show_progress = True`
`uncorrelated_location_entropy`	Uncorrelated location entropy (Cho, Myers, and Leskovec 2011).	'TrajDataFrame' object, `normalize = False`, `show_progress = True`
`mean_square_displacement`	Mean square displacement traveled by the individuals after a time (Brockmann *et al.* 2006).	'TrajDataFrame' object, `days = 0`, `hours = 1`, `minutes = 0`, `show_progress = True`
`visits_per_location`	Number of visits per location (Pappalardo and Simini 2018).	'TrajDataFrame' object
`homes_per_location`	Number of homes per location (Pappalardo and Simini 2018).	'TrajDataFrame' object, `start_night = '22:00'`, `end_night = '07:00'`
`visits_per_time_unit`	Number of visits to any location per time unit (Pappalardo and Simini 2018).	'TrajDataFrame' object, `time_unit = '1h'`
`origin_destination_matrix`	Origin-destination matrix inferred from the trajectories (Calabrese, Di Lorenzo, Liu, and Ratti 2011).	'TrajDataFrame' object, `self_loops = False`, `show_progress = True`

Table 5: Collective measures currently implemented in **scikit-mobility**. For each method, we provide a description and the set of arguments with their default values (if any).

```
>>> df = pd.read_csv(url, sep = '\t', header = 0, nrows = 100000, \
...   names = ['user', 'check-in_time', 'latitude', 'longitude', \
...     'location id'])
>>> tdf = skmob.TrajDataFrame(df, latitude = 'latitude', \
...   longitude = 'longitude', datetime = 'check-in_time', \
...     user_id = 'user').sort_values(by = 'datetime')
>>> print(tdf.head())

       uid                  datetime        lat         lng  \
15410   12 2008-03-22 05:01:29+00:00  39.768753 -105.006395
15409   12 2008-03-22 05:50:55+00:00  39.768057 -105.002983
15408   12 2008-03-22 06:17:35+00:00  39.768057 -105.002983
15407   12 2008-03-22 06:34:37+00:00  39.768057 -105.002983
15406   12 2008-03-22 19:45:30+00:00  39.785486 -104.895457
```

```
                                    location id
15410  7b52009b64fd0a2a49e6d8a939753077792b0554
15409  fa35e192121eabf3dabf9f5ea6abdbcbc107ac3b
15408  fa35e192121eabf3dabf9f5ea6abdbcbc107ac3b
15407  fa35e192121eabf3dabf9f5ea6abdbcbc107ac3b
15406  f1abd670358e036c31296e66b3b66c382ac00812
```

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

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

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

```
>>> 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 `rg_df` the column `radius_of_gyration` contains the moving object's radius of gyration.

```
>>> 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 similarly. The code below computes a collective measure – the number of visits per location (by any object). First, we import the function.

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

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

```
>>> 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 indicates the number of visits to that location in the 'TrajDataFrame'.

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

```
         lat         lng  n_visits
0  39.739154 -104.984703      3392
1  37.580304 -122.343679      2248
2  39.099275  -76.848306      1715
3  39.762146 -104.982480      1442
4  40.014986 -105.270546      1310
```

6. Individual generative algorithms

Generative mobility algorithms aim to create a population of agents whose mobility patterns are statistically indistinguishable from those of real individuals (Luca *et al.* 2021). A generative algorithm typically generates a synthetic trajectory corresponding to a single moving object, generally assuming that an object is independent of the others.

scikit-mobility implements the most common mechanistic individual generative algorithms, such as the exploration and preferential return model or EPR (Song *et al.* 2010a) and its variants (Pappalardo *et al.* 2016a), and DITRAS (Diary-based trajectory simulator modeling framework) (Pappalardo and Simini 2018). Generative individual mobility models based on artificial intelligence (Luca *et al.* 2021) are not included in the library's current version and will be included in later versions. Table 6 lists the individual generative models currently implemented in **scikit-mobility**.

Each generative algorithm is a Python class with a generate method to start the generation of synthetic trajectories. Below we show the code to generate a 'TrajDataFrame' describing the synthetic trajectory of 1000 agents that move on a spatial tessellation for a period specified in the input. First, we import the generative algorithm class ('DensityEPR') from the library.

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

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

```
>>> tessellation = gpd.GeoDataFrame.from_file('NY_counties_2011.geojson')
>>> start_time = pd.to_datetime('2019/01/01 08:00:00')
>>> 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 as input the start and end times, the spatial tessellation, the number of agents, and other model-specific parameters. The output of the simulation is a 'TrajDataFrame' object containing the trajectory of the 1000 agents.

```
>>> depr = DensityEPR()
>>> tdf = depr.generate(start_time, end_time, tessellation, \
...   n_agents = 1000, relevance_column = 'population', random_state = 42)
>>> print(tdf.head())
```

```
   uid                 datetime        lat        lng
0    1 2019-01-01 08:00:00.000000  42.393730 -76.875204
1    1 2019-01-01 08:36:25.019263  42.452018 -76.473618
2    1 2019-01-01 09:10:52.828149  42.393730 -76.875204
3    1 2019-01-02 03:14:59.370208  42.702464 -78.224637
4    1 2019-01-02 03:40:17.509278  44.592993 -74.303615
```

The argument random_state in method generate allows the user to set a seed that guarantees the results of the generation process are reproducible. Since random_state is None by default, if the user does not specify a value for random_state, every time they run the generate method the synthetic trajectories would be different each time.

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 as input a spatial tessellation, which should be a 'GeoDataFrame' with two columns geometry and relevance. These columns 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 spatial 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 gravity model, inspired by an analogy with Newton's law of gravitation (Zipf 1946), is based on the assumption that the number of travelers between two locations (flow) increases with the locations' populations while the number decreases with the distance between them. Formally, the gravity model can be formalized as $T_{i,j} = K m_i m_j f(r_{i,j})$ with $T_{i,j}$ denoting the flow between locations i and j, K representing a constant, m_i and m_j the populations of locations i and j respectively and $f(r_{i,j})$ is a deterrence function that decreases as the distance $r_{i,j}$ between i and j increases and it is usually a power law or an exponential. For more details on the gravity models, see Barbosa *et al.* (2018).

Generative model	Description	Arguments
`DensityEPR`	Density exploration and preferential return model (Pappalardo *et al.* 2016a; Barbosa *et al.* 2018).	`name = 'Density EPR model' rho = 0.6, gamma = 0.21, beta = 0.8 tau = 17, min_wait_time_minutes = 20`
`SpatialEPR`	Spatial exploration and preferential return model (Song *et al.* 2010b; Barbosa *et al.* 2015; Pappalardo *et al.* 2016a).	`name = 'Spatial EPR model', rho = 0.6, gamma = 0.21, beta = 0.8, tau = 17, min_wait_time_minutes = 20`
`Ditras`	Diary-based trajectory simulator modeling framework (Pappalardo and Simini 2018).	`diary_generator, name = 'Ditras model', rho = 0.3, gamma = 0.21`
`MarkovDiaryGenerator`	Markov diary learner (Pappalardo and Simini 2018).	`name = 'Markov diary'`
`Gravity`	Gravity model of human mobility (Zipf 1946; Barbosa *et al.* 2018).	`deterrence_func_type = 'power_law', deterrence_func_args = [-2.0], origin_exp = 1.0, destination_exp = 1.0, gravity_type = 'singly constrained', name = 'Gravity model'`
`Radiation`	Radiation model for human mobility (Simini *et al.* 2012).	`name = 'Radiation model'`

Table 6: Generative models currently implemented in **scikit-mobility**. For each method, we provide a description, its arguments and their default values (if any).

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

```
>>> import skmob
>>> import geopandas as gpd
>>> tessellation = gpd.GeoDataFrame.from_file('NY_counties_2011.geojson')
>>> 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 a file:

```
>>> fdf = skmob.FlowDataFrame.from_file('NY_commuting_flows_2011.csv',
...   tessellation = tessellation, tile_id = 'tile_id')
>>> 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. First, we instantiate the model:

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

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

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

```
  origin destination      flow
0  36019       36101  0.004387
1  36019       36107  0.003702
2  36019       36059  0.019679
3  36019       36011  0.006894
4  36019       36123  0.002292
```

8. Privacy risk assessment

Mobility data are sensitive since individuals' movements can reveal confidential personal information or allow the re-identification of individuals in a database, creating serious privacy risks (de Montjoye *et al.* 2013, 2018). The General Data Protection Regulation (GDPR) explicitly imposes 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 dataset. 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, 2022). A common framework for privacy risk assessment (Pratesi, Monreale, Trasarti, Giannotti, Pedreschi, and Yanagihara 2018) assumes that during the attack, a malicious adversary acquires, in some way, access to an anonymized mobility dataset, i.e., a mobility dataset in which the moving object associated with a trajectory is not known. Moreover, it assumes that the malicious adversary acquires information about (a portion of) a trajectory of an individual represented in the dataset. 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 dataset (Pellungrini *et al.* 2017).

scikit-mobility provides several attack models, each implemented as a Python class. In Table 7, we list the privacy attacks currently available in the library. 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). The following code instantiates a '`LocationAttack`' object:

```
>>> import skmob
>>> from skmob.privacy import attacks
>>> 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 '`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).

```
>>> tdf = skmob.TrajDataFrame.from_file('privacy_toy.csv')
>>> tdf_risk = at.assess_risk(tdf)
>>> 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
```

Attack model	Background knowledge	Arguments
`LocationAttack`	Locations visited by an object.	`knowledge_length`
`LocationSequenceAttack`	Temporal sequence of locations visited by an object.	`knowledge_length`
`LocationTimeAttack`	Locations visited by an object and the time of visit.	`knowledge_length` `time_precision = 'Hour'`
`UniqueLocationAttack`	Unique locations visited by an object, disregarding repeated visits to the same location.	`knowledge_length`
`LocationFrequencyAttack`	Unique locations visited by an object and frequency of visitation.	`knowledge_length` `tolerance = 0.0`
`LocationProbabilityAttack`	Unique locations visited by an object and probability of visiting each location.	`knowledge_length` `tolerance = 0.0`
`LocationProportionAttack`	Unique locations visited by an object and relative proportion of the frequencies of visit.	`knowledge_length` `tolerance = 0.0`
`HomeWorkAttack`	The two most visited locations in the trajectory of an object.	

Table 7: List of privacy attacks currently implemented in **scikit-mobility**. For each method, we provide a description, its arguments and their default values (if any).

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 objects with `uid` 1 and 2 only:

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

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

```
          lat        lng            datetime  uid  instance  instance_elem  \
0   43.843014  10.507994 2011-02-03 08:34:04    2         1              1
1   43.708530  10.403600 2011-02-03 09:34:04    2         1              2
2   43.843014  10.507994 2011-02-03 08:34:04    2         2              1
3   43.843014  10.507994 2011-02-04 10:34:04    2         2              2
4   43.843014  10.507994 2011-02-03 08:34:04    2         3              1
5   43.544270  10.326150 2011-02-04 11:34:04    2         3              2
6   43.708530  10.403600 2011-02-03 09:34:04    2         4              1
7   43.843014  10.507994 2011-02-04 10:34:04    2         4              2
8   43.708530  10.403600 2011-02-03 09:34:04    2         5              1
9   43.544270  10.326150 2011-02-04 11:34:04    2         5              2
10  43.843014  10.507994 2011-02-04 10:34:04    2         6              1
11  43.544270  10.326150 2011-02-04 11:34:04    2         6              2

    prob
0   0.25
1   0.25
2   0.50
3   0.50
4   0.25
5   0.25
6   0.25
7   0.25
8   0.25
9   0.25
10  0.25
11  0.25
```

The result is a **pandas** 'DataFrame' that contains 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. Existing tools

This section briefly describes some of the existing libraries and tools that provide functionalities for movement data management. Overall, none of the other packages are tailored explicitly for human mobility, and none of them includes functions for privacy risk assessment. Table 8 gives a summary of the packages and their functionalities.

9.1. R

A review of state of the art (Joo, Boone, Clay, Patrick, Clusella-Trullas, and Basille 2020) reveals that several packages (more than 50) deal with trajectory data in the R environment

Software	Data type	Processing	Plotting	Measures	Models	Privacy
scikit-mobility	Many	✓	✓	✓	✓	✓
bandicoot	Mobile phone	✓	–	✓	–	–
movingpandas	Many	✓	✓	–	–	–
spacetime	Many	✓	✓	–	–	-
trajectories	Many	✓	✓	✓	–	–
adehabitatLT	Many	✓	✓	–	–	–
TrajDataMining	Many	✓	✓	–	–	–

Table 8: Comparison of **scikit-mobility** with other libraries that cover similar aspects of spatio-temporal and mobility data.

for statistical computing and graphics (R Core Team 2022). In the following, we give a brief overview of the packages that are the closest in the scope to **scikit-mobility**.

spacetime

The **spacetime** package (Pebesma 2012) provides methods and functionalities from two other R packages, **sp** (Pebesma and Bivand 2005) and **xts** (Ryan and Ulrich 2020). Package **sp** deals with different spatial data such as polygons, shapes, lines, or points, while package **xts** handles time and dates. **spacetime** provides several functionalities for the handling of spatio-temporal data, such as interpolation and calculation of empirical orthogonal functions. For visualizing data, **spacetime** relies on other R packages, for example **maps** (Becker, Wilks, Brownrigg, Minka, and Deckmyn 2021) is used to draw geographical maps.

trajectories

trajectories (Pebesma, Klus, and Moradi 2021) builds on the foundation of **spacetime**, providing a wider set of tools for managing non-domain specific trajectory data. It allows for handling single tracks of movement for each agent, plotting, and simulating trajectories of different nature. It also provides model fitting for studying the behavior of individual tracks.

adehabitat

The packages under the **adehabitat** family (Calenge 2006) cover methods and functions to manage animal movement and habitat selection. The original package has been split into smaller packages: **adehabitatHR** deals with home-range analysis, **adehabitatHM** deals with habitat selection analysis, **adehabitatLT** deals with animal trajectory analysis, and **adehabitatMA** deals with maps. Many of the functions presented in these packages are specific to animal movement. **adehabitatLT** (Calenge 2020) is the most similar library to **scikit-mobility**. While **adehabitatLT** can handle both regular and irregular trajectories, its design has been optimized for handling regular ones, as specified in the package's documentation. Moreover, the main trajectory class of **adehabitatLT** has been designed to handle bursts of movements to better model animals' behavior, which tends to be one of alternating rests and movements. **scikit-mobility**, while focusing on human trajectories, is agnostic to the sampling of the points in the trajectories or their periodicity.

TrajDataMining

TrajDataMining (Monteiro 2018) provides methods for trajectory data preparation, such as filtering, compression, and clustering. It also provides some pattern recognition tools to extract recurrent movement behaviors from the trajectories. However, it does not implement generative models nor advanced plotting functionalities.

9.2. Python

As for Python, some libraries have been proposed to manage and manipulate mobility data. In this section, we revise the most similar libraries in their purpose to what we propose in this paper, highlighting the differences between them and **scikit-mobility**.

bandicoot

bandicoot (de Montjoye, Rocher, and Pentland 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 analyzing 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 provides to the use a complete set of standard mobility measures ready to be used. 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 absent in **bandicoot**.

movingpandas

movingpandas (Graser 2019) is an extension of Python library **pandas** (McKinney 2010) and its spatial extension **geopandas** (Jordahl *et al.* 2019) 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. Since **movingpandas** is 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 focuses on the concept of trajectory, **movingpandas** does not implement any features specific to mobility analysis, such as statistical laws of mobility, generative models, standard pre-processing functions, and methods to assess privacy risk in mobility data.

10. Conclusion and future developments

In this paper, we presented **scikit-mobility**, a Python library for the analysis, generation, and privacy risk assessment of mobility data. **scikit-mobility** allows the user to manage two types

of mobility data – trajectories and flows – and it provides several modules, each dedicated to a specific aspect of mobility data analysis.

scikit-mobility has the advantage of providing, in a single environment, functions to deal with various aspects of mobility analysis, such as data pre-processing, cleaning, and visualization, computation of mobility metrics, generation of synthetic trajectories and flows, and the assessment of privacy risk. The current version of the library has some limitations, too. For example, since **pandas** '`DataFrame`'s must be fully loaded in memory, the size of the mobility dataset that can be analyzed is limited by the capacity of the memory of the user's machine. Moreover, although the library could be easily adapted to work with any geographic reference system, it is currently designed to work with the latitude and longitude reference system only.

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 (Luca *et al.* 2021; Wu *et al.* 2018). We will also consider including a module for performing map matching, i.e., assigning the points of a trajectory to the street network, and a module to compute the similarity between trajectories.

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 efficient on mobility datasets in the order of gigabytes, it is not scalable to massive mobility data in the order of terabytes or more. Since new Python libraries similar to **pandas** but more computationally efficient are developed every year such as **dask** (Rocklin 2015), we plan to re-implement crucial functions in **scikit-mobility** so that they can exploit the computational efficiency of these libraries. This aspect will become crucial when the scientific community largely adopts the library.

Acknowledgments

Gianni Barlacchi has done the work prior to joining Amazon. Luca Pappalardo and Roberto Pellungrini have been supported by EU project SoBigData++ RI grant #871042 and by EU project Track&Know H2020 grant #780754. Filippo Simini has been partially supported by EPSRC First Grant EP/P012906/1. We thank Anita Graser for the useful discussions, and Michele Ferretti, Giuliano Cornacchia, and Massimiliano Luca for their support in the development of the library.

L.P. developed modules, performed experiments, developed code examples, made the documentation, 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 MN, Barlacchi G, Braghin S, Calabrese F, Ferretti M, Lonij V, Nair R, Novack R, Paraszczak J, Toor AS (2016). "A Multi-Scale Approach to Data-Driven Mass Migration Analysis." In *SoGood@ ECML-PKDD*.

Alessandretti L, Sapiezynski P, Sekara V, Lehmann S, Baronchelli A (2018). "Evidence for a Conserved Quantity in Human Mobility." *Nature Human Behaviour*, **2**(7), 485–491. doi:10.1038/s41562-018-0364-x.

Bagrow YR, Lin JP (2012). "Mesoscopic Structure and Social Aspects of Human Mobility." *PLOS ONE*, **7**(5), 1–6. doi:10.1371/journal.pone.0037676.

Barbosa H, Barthelemy M, Ghoshal G, James CR, Lenormand M, Louail T, Menezes R, Ramasco JJ, Simini F, Tomasini M (2018). "Human Mobility: Models and Applications." *Physics Reports*, **734**, 1–74. doi:10.1016/j.physrep.2018.01.001.

Barbosa H, de Lima-Neto FB, Evsukoff A, Menezes R (2015). "The Effect of Recency to Human Mobility." *EPJ Data Science*, **4**(1), 21. doi:10.1140/epjds/s13688-015-0059-8.

Barlacchi G, De Nadai M, Larcher R, Casella A, Chitic C, Torrisi G, Antonelli F, Vespignani A, Pentland A, Lepri B (2015). "A Multi-Source Dataset of Urban Life in the City of Milan and the Province of Trentino." *Scientific Data*, **2**, 150055. doi:10.1038/sdata.2015.55.

Barlacchi G, Perentis C, Mehrotra A, Musolesi M, Lepri B (2017). "Are You Getting Sick? Predicting Influenza-Like Symptoms Using Human Mobility Behaviors." *EPJ Data Science*, **6**(1), 27. doi:10.1140/epjds/s13688-017-0124-6.

Becker RA, Wilks AR, Brownrigg R, Minka TP, Deckmyn A (2021). ***maps**: Draw Geographical Maps*. R package version 3.4.0, URL https://CRAN.R-project.org/package=maps.

Blondel VD, Decuyper A, Krings G (2015). "A Survey of Results on Mobile Phone Datasets Analysis." *EPJ Data Science*, **4**, 1–55. doi:10.1140/epjds/s13688-015-0046-0.

Bohm M, Nanni M, Pappalardo L (2021). "Quantifying the Presence of Air Pollutants over a Road Network in High Spatio-Temporal Resolution." In *Climate Change AI, NeurIPS Workshop*.

Brockmann D, Hufnagel L, Geisel T (2006). "The Scaling Laws of Human Travel." *Nature*, **439**(7075), 462–465. doi:10.1038/nature04292.

Calabrese F, Di Lorenzo G, Liu L, Ratti C (2011). "Estimating Origin-Destination Flows Using Mobile Phone Location Data." *IEEE Pervasive Computing*, **10**(4), 36–44. doi:10.1109/mprv.2011.41.

Calenge C (2006). "The Package **adehabitat** for the R Software: Tool for the Analysis of Space and Habitat Use by Animals." *Ecological Modelling*, **197**(3–4), 1035. doi:10.1016/j.ecolmodel.2006.03.017.

Calenge C (2020). ***adehabitatLT**: Analysis of Animal Movements*. R package version 0.3.25, URL https://CRAN.R-project.org/package=adehabitatLT.

Canzian L, Musolesi M (2015). "Trajectories of Depression: Unobtrusive Monitoring of Depressive States by Means of Smartphone Mobility Traces Analysis." In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pp. 1293–1304. ACM, New York. doi:10.1145/2750858.2805845.

Cho E, Myers SA, Leskovec J (2011). "Friendship and Mobility: User Movement in Location-Based Social Networks." In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pp. 1082–1090. ACM, New York. doi:10.1145/2020408.2020579.

de Montjoye YA, Gambs S, Blondel V, Canright G, de Cordes N, Deletaille S, Engø-Monsen K, Garcia-Herranz M, Kendall J, Kerry C, Krings G, Letouzé E, Luengo-Oroz M, Oliver N, Rocher L, Rutherford A, Smoreda Z, Steele J, Wetter E, Pentland AS, Bengtsson L (2018). "On the Privacy-Conscientious Use of Mobile Phone Data." *Scientific Data*, **5**(1), 180286. doi:10.1038/sdata.2018.286.

de Montjoye YA, Hidalgo CA, Verleysen M, Blondel VD (2013). "Unique in the Crowd: The Privacy Bounds of Human Mobility." *Scientific Reports*, **3**, 1376. doi:10.1038/srep01376.

de Montjoye YA, Rocher L, Pentland AS (2016). "**bandicoot**: A Python Toolbox for Mobile Phone Metadata." *Journal of Machine Learning Research*, **17**(175), 1–5.

Fernandes F, *et al.* (2019). ***Folium**: Phyton Visualization v0.5.1*. doi:10.5281/zenodo.3559751.

Fernandez Arguedas V, Pallotta G, 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. doi:10.1109/tits.2017.2699635.

Fiore M, Katsikouli P, Zavou E, Cunche M, Fessant F, Hello DL, Aïvodji UM, Olivier B, Quertier T, Stanica R (2020). "Privacy in Trajectory Micro-Data Publishing: A Survey." *Transactions on Data Privacy*, **13**(2), 91–149.

González MC, Hidalgo CA, Barabási AL (2008). "Understanding Individual Human Mobility Patterns." *Nature*, **453**(7196), 779–782. doi:10.1038/nature06958.

Graser A (2019). "**movingpandas**: Efficient Structures for Movement Data in Python." *Journal of Geographic Information Science*, **1**, 54–68. doi:10.1553/giscience2019_01_s54.

Hariharan R, Toyama K (2004). "Project Lachesis: Parsing and Modeling Location Histories." In *International Conference on Geographic Information Science*, pp. 106–124. Springer-Verlag.

Hunter JD (2007). "**matplotlib**: A 2D Graphics Environment." *Computing in Science & Engineering*, **9**(3), 90–95. doi:10.1109/mcse.2007.55.

Jiang S, Yang Y, Gupta S, Veneziano D, Athavale S, González MC (2016). "The TimeGeo Modeling Framework for Urban Mobility without Travel Surveys." *Proceedings of the National Academy of Sciences of the United States of America*, **113**(37), E5370. doi:10.1073/pnas.1524261113.

Joo R, Boone ME, Clay TA, Patrick SC, Clusella-Trullas S, Basille M (2020). "Navigating through the R Packages for Movement." *Journal of Animal Ecology*, **89**(1), 248–267. doi:10.1111/1365-2656.13116.

Jordahl K, Van den Bossche J, Wasserman J, McBride J, Gerard J, Tratner J, Perry M, Farmer C, Gillies S, Cochran M, *et al.* (2019). ***geopandas** v0.5.1*. doi:10.5281/zenodo.3333010.

Karamshuk D, Boldrini C, Conti M, Passarella A (2011). "Human Mobility Models for Opportunistic Networks." *IEEE Communications Magazine*, **49**(12), 157–165. doi:10.1109/mcom.2011.6094021.

Luca M, Barlacchi G, Lepri B, Pappalardo L (2021). "A Survey on Deep Learning for Human Mobility." *ACM Computing Surveys*, **55**(1), 7. doi:10.1145/3485125.

McKinney W (2010). "Data Structures for Statistical Computing in Python." In S Van der Walt, J Millman (eds.), *Proceedings of the 9th Python in Science Conference*, pp. 51–56.

Monreale A, Rinzivillo S, Pratesi F, Giannotti F, Pedreschi D (2014). "Privacy-by-Design in Big Data Analytics and Social Mining." *EPJ Data Science*, **3**(1), 10. doi:10.1140/epjds/s13688-014-0010-4.

Monteiro D (2018). ***TrajDataMining**: Trajectories Data Mining*. R package version 0.1.6, URL https://CRAN.R-project.org/package=TrajDataMining.

Noulas A, Scellato S, Lambiotte R, Pontil M, Mascolo C (2012). "A Tale of Many Cities: Universal Patterns in Human Urban Mobility." *PLOS ONE*, **7**(5), 1–10. doi:10.1371/journal.pone.0037027.

Nyhan MM, Kloog I, Britter R, Ratti C, 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. doi:10.1038/s41370-018-0038-9.

Oliphant TE (2006). *A Guide to **numpy***, volume 1. Trelgol Publishing.

Pappalardo L, Ferres L, Sacasa M, Cattuto C, Bravo L (2021). "Evaluation of Home Detection Algorithms on Mobile Phone Data Using Individual-Level Ground Truth." *EPJ Data Science*, **10**(1), 29. doi:10.1140/epjds/s13688-021-00284-9.

Pappalardo L, Rinzivillo S, Qu Z, Pedreschi D, Giannotti F (2013). "Understanding the Patterns of Car Travel." *The European Physical Journal Special Topics*, **215**(1), 61–73. doi:10.1140/epjst/e2013-01715-5.

Pappalardo L, Rinzivillo S, Simini F (2016a). "Human Mobility Modelling: Exploration and Preferential Return Meet the Gravity Model." *Procedia Computer Science*, **83**, 934–939. doi:10.1016/j.procs.2016.04.188. 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 L, Simini F (2018). "Data-Driven Generation of Spatio-Temporal Routines in Human Mobility." *Data Mining and Knowledge Discovery*, **32**(3), 787–829. doi:10.1007/s10618-017-0548-4.

Pappalardo L, Simini F, Barlacchi G, Pellungrini R (2022). ***scikit-mobility**: Mobility Analysis in Python*. Phyton library version 1.7, URL https://github.com/scikit-mobility/scikit-mobility.

Pappalardo L, Simini F, Rinzivillo S, Pedreschi D, Giannotti F, Barabási AL (2015). "Returners and Explorers Dichotomy in Human Mobility." *Nature Communications*, **6**, 8166. doi:10.1038/ncomms9166.

Pappalardo L, Vanhoof M, Gabrielli L, Smoreda Z, Pedreschi D, 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. doi:10.1007/s41060-016-0013-2.

Pebesma E (2012). "**spacetime**: Spatio-Temporal Data in R." *Journal of Statistical Software*, **51**(7), 1–30. doi:10.18637/jss.v051.i07.

Pebesma E, Klus B, Moradi M (2021). ***trajectories**: Classes and Methods for Trajectory Data*. R package version 0.2-3, URL https://CRAN.R-project.org/package=trajectories.

Pebesma EJ, Bivand RS (2005). "Classes and Methods for Spatial Data in R." *R News*, **5**(2), 9–13. URL https://CRAN.R-project.org/doc/Rnews/.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011). "**scikit-learn**: Machine Learning in Python." *Journal of Machine Learning Research*, **12**, 2825–2830.

Pellungrini R, Pappalardo L, Pratesi F, Monreale A (2017). "A Data Mining Approach to Assess Privacy Risk in Human Mobility Data." *ACM Transactions on Intelligent Systems and Technology*, **9**(3), 1–27. doi:10.1145/3106774.

Pellungrini R, Pappalardo L, Simini F, Monreale A (2022). "Modeling Adversarial Behavior Against Mobility Data Privacy." *IEEE Transactions on Intelligent Transportation Systems*, **23**(2), 1145–1158. doi:10.1109/tits.2020.3021911.

Phithakkitnukoon S, Smoreda Z, Olivier P (2012). "Socio-Geography of Human Mobility: A Study Using Longitudinal Mobile Phone Data." *PLOS ONE*, **7**(6), 1–9. doi:10.1371/journal.pone.0039253.

Pratesi F, Monreale A, Trasarti R, Giannotti F, Pedreschi D, 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 G, Mateos JL, Miramontes O, Cocho G, Larralde H, Ayala-Orozco B (2004). "Lévy Walk Patterns in the Foraging Movements of Spider Monkeys (*Ateles Geoffroyi*)." *Behavioral Ecology and Sociobiology*, **55**(3), 223–230. doi:10.1007/s00265-003-0700-6.

R Core Team (2022). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rinzivillo S, Gabrielli L, Nanni M, Pappalardo L, Pedreschi D, Giannotti F (2014). "The Purpose of Motion: Learning Activities from Individual Mobility Networks." In *2014 International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 312–318. doi:10.1109/dsaa.2014.7058090.

Rocklin M (2015). "Dask: Parallel Computation with Blocked Algorithms and Task Scheduling." In K Huff, J Bergstra (eds.), *Proceedings of the 14th Python in Science Conference*, pp. 126–132.

Rossi A, Barlacchi G, Bianchini M, Lepri B (2019). "Modelling Taxi Drivers' Behaviour for the Next Destination Prediction." *IEEE Transactions on Intelligent Transportation Systems*, **21**(7), 2980–2989. doi:10.1109/tits.2019.2922002.

Rossi A, Pappalardo L, Cintia P, Iaia MF, Fernández J, Medina D (2018). "Effective Injury Prediction in Professional Soccer with GPS Data and Machine Learning." *PLOS ONE*, **13**(7), 1–15. doi:10.1371/journal.pone.0201264.

Ryan JA, Ulrich JM (2020). ***xts**: eXtensible Time Series*. R package version 0.12.1, URL https://CRAN.R-project.org/package=xts.

Simini F, González MC, Maritan A, Barabási AL (2012). "A Universal Model for Mobility and Migration Patterns." *Nature*, **484**(7392), 96–100. doi:10.1038/nature10856.

Song C, Koren T, Wang P, Barabási AL (2010a). "Modelling the Scaling Properties of Human Mobility." *Nature Physics*, **6**(10), 818–823. doi:10.1038/nphys1760.

Song C, Qu Z, Blumm N, Barabási AL (2010b). "Limits of Predictability in Human Mobility." *Science*, **327**(5968), 1018–1021. doi:10.1126/science.1177170.

Tizzoni M, Bajardi P, Poletto C, Ramasco JJ, Balcan D, Gonçalves B, Perra N, Colizza V, Vespignani A (2012). "Real-Time Numerical Forecast of Global Epidemic Spreading: Case Study of 2009 A/H1N1pdm." *BMC Medicine*, **10**(1), 165. doi:10.1186/1741-7015-10-165.

Tomasini M, Mahmood B, Zambonelli F, Brayner A, Menezes R (2017). "On the Effect of Human Mobility to the Design of Metropolitan Mobile Opportunistic Networks of Sensors." *Pervasive and Mobile Computing*, **38**(1), 215–232. doi:10.1016/j.pmcj.2016.12.007.

Van Rossum G, *et al.* (2011). *Python Programming Language*. URL https://www.python.org.

Voukelatou V, Gabrielli L, Miliou I, Cresci S, Sharma R, Tesconi M, Pappalardo L (2020). "Measuring Objective and Subjective Well-Being: Dimensions and Data Sources." *International Journal of Data Science and Analytics*, **11**(4), 279–309. doi:10.1007/s41060-020-00224-2.

Wang D, Pedreschi D, Song C, Giannotti F, Barabasi AL (2011). "Human Mobility, Social Ties, and Link Prediction." In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, volume 21 of *KDD '11*, pp. 1100–1108. ACM, New York. doi:10.1145/2020408.2020581.

Wang J, Kong X, Xia F, Sun L (2019). "Urban Human Mobility: Data-Driven Modeling and Prediction." *ACM SIGKDD Explorations Newsletter*, pp. 1–19. doi:10.1145/3331651.3331653.

Williams NE, Thomas TA, Dunbar M, Eagle N, Dobra A (2015). "Measures of Human Mobility Using Mobile Phone Records Enhanced with GIS Data." *PLOS ONE*, **10**(7), 1–16. doi:10.1371/journal.pone.0133630.

Wilson AG (1971). "A Family of Spatial Interaction Models, and Associated Developments." *Environment and Planning A*, **3**(1), 1–32. doi:10.1068/a030001.

Wu R, Luo G, Shao J, Tian L, Peng C (2018). "Location Prediction on Trajectory Data: A Review." *Big Data Mining and Analytics*, **1**(2), 108–127. doi:10.26599/bdma.2018.9020010.

Zhang J, Zheng Y, Qi D (2017). "Deep Spatio-Temporal Residual Networks for Citywide Crowd Flows Prediction." In *Thirty-First AAAI Conference on Artificial Intelligence*.

Zhao K, Tarkoma S, Liu S, Vo H (2016). "Urban Human Mobility Data Mining: An Overview." In *2016 IEEE International Conference on Big Data*, pp. 1911–1920. doi:10.1109/bigdata.2016.7840811.

Zheng Y (2015). "Trajectory Data Mining: An Overview." *ACM Transactions on Intelligent Systems and Technology*, **6**(3), 29:1–29:41. doi:10.1145/2743025.

Zheng Y, Capra L, Wolfson O, Yang H (2014). "Urban Computing: Concepts, Methodologies, and Applications." *ACM Transactions on Intelligent System Technologies*, **5**(3), 38:1–38:55. doi:10.1145/2629592.

Zheng Y, Wang L, Zhang R, Xie X, Ma WY (2008). "GeoLife: Managing and Understanding Your Past Life over Maps." In *Proceedings of the The Ninth International Conference on Mobile Data Management*, MDM '08, pp. 211–212. IEEE Computer Society, Washington, DC. doi:10.1109/mdm.2008.20.

Zipf GK (1946). "The P_1P_2/D Hypothesis: On the Intercity Movement of Persons." *American Sociological Review*, **11**(6), 677–686. doi:10.2307/2087063.

Affiliation:

Luca Pappalardo
ISTI-CNR
Via G. Moruzzi 1
56124 Pisa, Italy
E-mail: luca.pappalardo@isti.cnr.it
Twitter: @lucpappalard

Filippo Simini
Argonne Leadership Computing Facility
Argonne National Laboratory
Lemont, IL, United States of America
and
The Alan Turing Institute
London, United Kingdom
E-mail: fsimini@anl.gov

Gianni Barlacchi
Amazon, Alexa AI
Berlin, Germany
E-mail: gbarlac@amazon.com

Roberto Pellungrini
University of Pisa
Department of Computer Science
Pisa, Italy
E-mail: roberto.pellungrini@di.unipi.it

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
July 2022, Volume 103, Issue 4 *Submitted:* 2019-10-28
doi:10.18637/jss.v103.i04 *Accepted:* 2021-06-10