gps2space

Mar 20, 2022

Getting Started

Many data are not readily in spatial format. For example, data from wearable devices, surveys, and social media platforms such as Facebook and Twitter have GPS location information, but usually in raw Lat/Long format. For social scientists who do not have strong background in Geographic Information System (GIS), compiling and analyzing spatial data from the aforementioned sources can be tedious and error-prone. `GPS2space` is an open source solution to this issue and can ease the processes of compiling and calculating activity spaces based on raw Lat/Long coordinate pairs.

CHAPTER 1

Description

The primary goals of `GPS2space` are: 1) to build spatial data from raw Lat/Long coordinate pairs and make the process less painful for social scientists with little GIS background; 2) to build minimum bounding geometry from Points using buffer, convex hull methods, and use activity space as building box to calculate shared space at different scales; 3) to calculate the nearest distance from user-defined landmarks.

1.1 Installation

1.1.1 Installing with pip

`GPS2space` can be installed from PyPI using `pip install`:

```
pip install gps2space
```

1.1.2 Installing from source

You can install by cloning the GitHub repository, then use pip to install from where you store the cloned files:

```
git clone https://github.com/shuai-zhou/gps2space.git
cd gps2space
pip install .
```

You can also install directly from the GitHub repository:

```
pip install git+git://github.com/shuai-zhou/gps2space.git
```

1.2 Overview

Note: An accurate and appropriate GIS database depends on the geographic coordinate system (sometimes is used interchangeably with datum) your are using. For example, in the North America, commonly used datums are NAD37, NAD83, and WGS84. Throughout our package, we are using WGS84 datum because WGS84 is commonly used all over the world and it is the default geographic coordinate system for the Global Positioning System (GPS).

Although the differences between those datums are usually not discernible, we recommend you to check what datum you are using with your data vendor for accurate spatial measures. If your datum is not WGS84, please transform it to WGS84 datum before you using this package.

This package is released under the MIT License, which exempts the authors and copyright holders from any claim, damages or other liability. We will do our best to guarantee the reliability and validity of our package, but users are responsible for their own work. See *Tips* for some of our suggestions in using this package to conduct reliable and replicable research.

The following shows the available functions of the package:

- `geodf.df_to_gdf`: This function builds unprojected GeoDataFrame from DataFrame with Lat/Long coordinate pairs
- `space.buffer_space`: This function calculates buffer-based activity space with user-defined level of aggregation, buffer distance, and projection
- `space.convex_space`: This function calculates convex hull-based activity space with user-defined level of aggregation and projection
- `dist.dist_to_point`: This function calculates nearest Point-Point distance with user-defined projection
- `dist.dist_to_poly`: This function calculates nearest Point-Polygon distance with user-defined projection and search radius

1.3 Building Spatial Data

Compiling spatial data from raw Latitude/Longitude coordinate pairs sometimes is tedious and error-prone, especially for social scientists without much background in GIS. This example will guide you on how to convert raw Latitude/Longitude coordinate pairs data to spatial data using our function `df_to_gdf`.

The data we are using is making-up Latitude/Longitude coordinate pairs of Person 1 (P1) and Person 2 (P2) in Pennsylvania, USA from 2020-01-01 to 2020-08-01. See here for information about how we compile the data. You can download all the data we will be using from this GitHub Repository. Make sure the directory is your own working directory when implementing the following steps.

To begin with, we need to import libraries we will be using to read and manipulate data.

[1]:
```
%matplotlib inline

import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
```

Now, we load our data example.

[2]:
```
df = pd.read_csv('../data/example.csv')
df.head()
```

[2]:
```
  pid                      timestamp   latitude  longitude
0  P2  2020-04-27 10:42:22.162176000  40.993799 -76.669419
```

(continues on next page)

(continued from previous page)

```
1  P2  2020-06-02 01:12:45.308505600  39.946904 -78.926234
2  P2  2020-05-08 23:47:33.718185600  41.237403 -79.252317
3  P2  2020-04-26 14:31:12.100310400  41.991390 -77.467769
4  P2  2020-03-31 15:53:27.777897600  41.492674 -76.542921
```

We import our `geodf` module. The `geodf` module has a function `df_to_gdf` which takes three parameters:

- df: This is the name of your DataFrame
- x: This is the column name of your Longitude
- y: This is the column name of your Latitude

Make sure that you pass your Longitude and Latitude columns to x and y, respectively.

[3]:
```
from gps2space import geodf
```

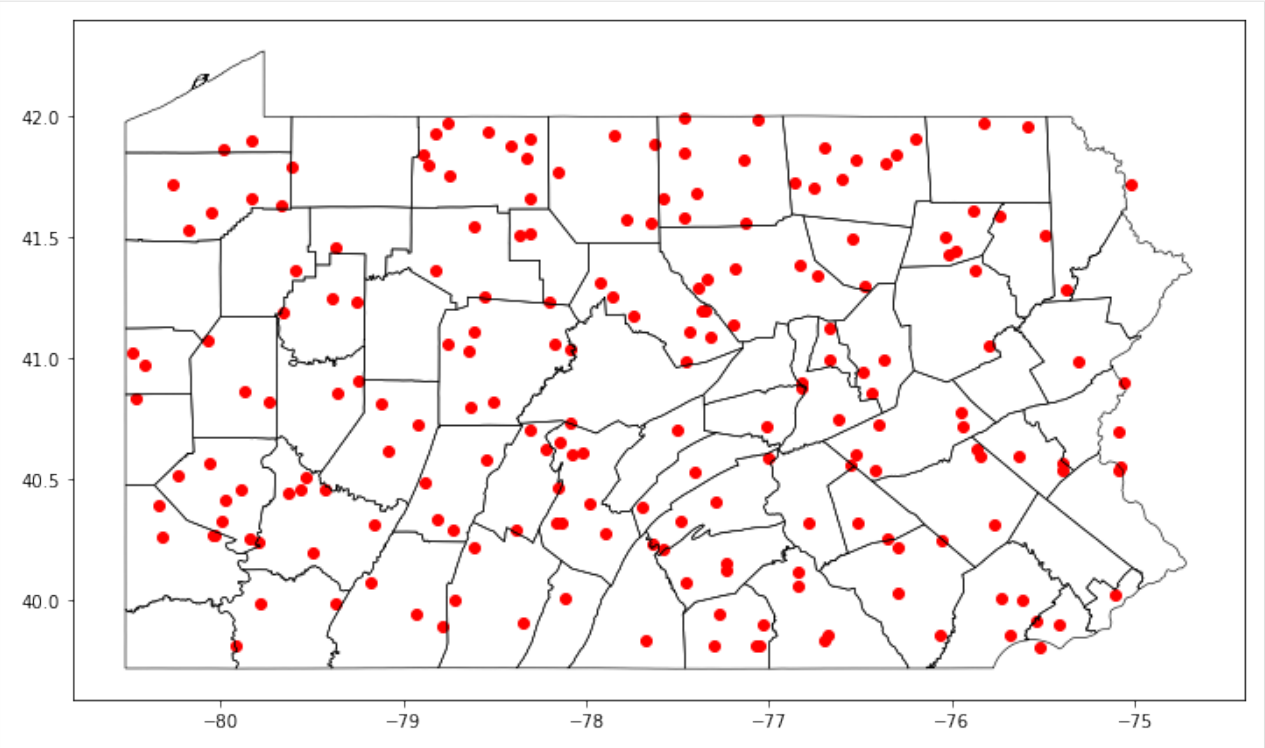
[4]:
```
gdf = geodf.df_to_gdf(df, x='longitude', y='latitude')
gdf.head()
```

[4]:
```
  pid                     timestamp   latitude  longitude  \
0  P2  2020-04-27 10:42:22.162176000  40.993799 -76.669419
1  P2  2020-06-02 01:12:45.308505600  39.946904 -78.926234
2  P2  2020-05-08 23:47:33.718185600  41.237403 -79.252317
3  P2  2020-04-26 14:31:12.100310400  41.991390 -77.467769
4  P2  2020-03-31 15:53:27.777897600  41.492674 -76.542921

                     geometry
0  POINT (-76.66942 40.99380)
1  POINT (-78.92623 39.94690)
2  POINT (-79.25232 41.23740)
3  POINT (-77.46777 41.99139)
4  POINT (-76.54292 41.49267)
```

Now the data are in spatial format with a geometry column that stores the geographical location information, we can plot the Point GeoDataFrame on the map of Pennsylvania, USA. Please note the spatial data is in WGS84 datum and is not projected.

[5]:
```
pa = gpd.read_file('../data/pacounty.shp')
```

[6]:
```
ax = pa.boundary.plot(figsize=(12, 12), edgecolor='black', linewidth=0.6)
gdf.plot(ax=ax, color='r')

plt.show();
```



We can then save the spatial data to `shp` file.

[7]:
```
gdf.to_file('../data/example.shp')
```

1.4 Activity Space

With the Point GeoDataFrame, we can measure activity space by building geometric shapes using minimum bounding box methods. There are several ways to build minimum bounding box, including buffer, convex hull, circle, envelope, etc., each with pros and cons depending on the geographic distribution of the Point GeoDataFrame. Currently, we support buffer and convex hull methods in building minimum bounding box and calculating corresponding activity space. The following examples demonstrate how to implement buffer- and convex hull-based activity space. We will use the example data we used in the last section as an example to illustrate how to calculate buffer- and convex hull-based activity space. You can refer to here for how we compile the data.

We first need to import libraries we will be using for the examples.

[1]:
```
%matplotlib inline

import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
```

We then load the data and create spatial data using the `df_to_gdf` method as we did in the last section.

[2]:
```
df = pd.read_csv('../data/example.csv')
df.head()
```

[2]:
```
  pid                      timestamp   latitude  longitude
0  P2  2020-04-27 10:42:22.162176000  40.993799 -76.669419
```

(continues on next page)

(continued from previous page)

```
1  P2  2020-06-02 01:12:45.308505600  39.946904 -78.926234
2  P2  2020-05-08 23:47:33.718185600  41.237403 -79.252317
3  P2  2020-04-26 14:31:12.100310400  41.991390 -77.467769
4  P2  2020-03-31 15:53:27.777897600  41.492674 -76.542921
```

There are two persons, P1 and P2, and their locations along with timestamp.

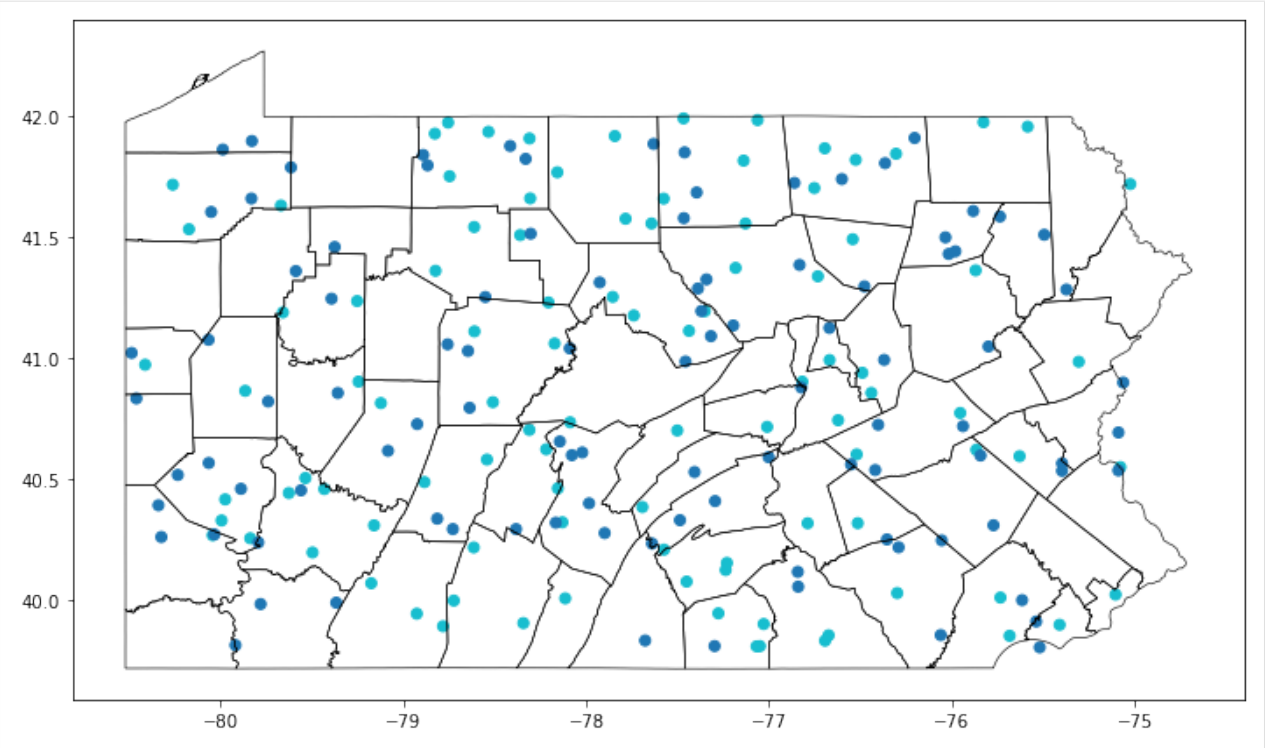
[3]:
```
from gps2space import geodf
```

[4]:
```
gdf = geodf.df_to_gdf(df, x='longitude', y='latitude')
gdf.head()
```

[4]:
```
  pid                     timestamp   latitude  longitude  \
0  P2  2020-04-27 10:42:22.162176000  40.993799 -76.669419
1  P2  2020-06-02 01:12:45.308505600  39.946904 -78.926234
2  P2  2020-05-08 23:47:33.718185600  41.237403 -79.252317
3  P2  2020-04-26 14:31:12.100310400  41.991390 -77.467769
4  P2  2020-03-31 15:53:27.777897600  41.492674 -76.542921

                    geometry
0  POINT (-76.66942 40.99380)
1  POINT (-78.92623 39.94690)
2  POINT (-79.25232 41.23740)
3  POINT (-77.46777 41.99139)
4  POINT (-76.54292 41.49267)
```

[5]:
```
pa = gpd.read_file('../data/pacounty.shp')
```

[6]:
```
ax = pa.boundary.plot(figsize=(12, 12), edgecolor='black', linewidth=0.6)
gdf.plot(ax=ax, column='pid')

plt.show();
```



The figure shows the distribution of our data with two different colors representing P1 and P2, respectively.

1.4.1 Buffer-based activity space

We import the `space` module. The `space` module has a function `buffer_space` which takes four parameters:

- gdf: This is your GeoDataFrame
- dist: This is the buffer distance, the default value is 0 meter
- dissolve: This is the level of aggregating from which you aggregate points to form polygon, the default value is week
- proj: This is the EPSG identifier you want to use to project your spatial data, the default value is 2163

Please note: Buffer distance and your projection are related. For raw Lat/Long coordinate pairs (often called unprojected data), the unit is degree. It is not usual to buffer geometry in degrees. You have to decide which projection system is most appropriate for your own data based on the geographical location. For example, **EPSG:2163** is commonly used in the United States, and the unit of distance is meter. See here for more information about EPSG identifier.

In this example, we will calculate activity space on a weekly basis. Before that, we need to create a column represents `week` from the timestamp. We will also create `year` and `month` just to show how you can obtain those information from timestamp. It is better to include `infer_datetime_format=True` because this will make `datetime` function much faster, especially when dealing with big data.

[7]:
```
gdf['timestamp'] = pd.to_datetime(gdf['timestamp'], infer_datetime_format=True)
gdf['year'] = gdf['timestamp'].dt.year
gdf['month'] = gdf['timestamp'].dt.month
gdf['week'] = gdf['timestamp'].dt.week
gdf.head()
```

```
<ipython-input-7-0eabe24f516a>:4: FutureWarning: Series.dt.weekofyear and Series.dt.
↪week have been deprecated.  Please use Series.dt.isocalendar().week instead.
  gdf['week'] = gdf['timestamp'].dt.week
```

[7]:
```
  pid                     timestamp   latitude  longitude  \
0  P2 2020-04-27 10:42:22.162176000  40.993799 -76.669419
1  P2 2020-06-02 01:12:45.308505600  39.946904 -78.926234
2  P2 2020-05-08 23:47:33.718185600  41.237403 -79.252317
3  P2 2020-04-26 14:31:12.100310400  41.991390 -77.467769
4  P2 2020-03-31 15:53:27.777897600  41.492674 -76.542921

                      geometry  year  month  week
0  POINT (-76.66942 40.99380)  2020      4    18
1  POINT (-78.92623 39.94690)  2020      6    23
2  POINT (-79.25232 41.23740)  2020      5    19
3  POINT (-77.46777 41.99139)  2020      4    17
4  POINT (-76.54292 41.49267)  2020      3    14
```

Now that we have the week column, we can calculate buffer-based activity space on a weekly basis using the `buffer_space` function and pass your choice to the four parameters we mentioned before. In this example, we will use 100 meters as buffer distance and project our data in **EPSG:2163**.

[8]:
```
from gps2space import space
```

[9]:
```
buff_space = space.buffer_space(gdf, dist=100, dissolve='week', proj=2163)
```

[10]:
```
buff_space.head()
```

[10]:
```
   week                                           geometry pid  \
0     1  MULTIPOLYGON (((1720865.011 -318218.702, 17208...  P2
1     2  MULTIPOLYGON (((1825885.203 -276088.593, 18258...  P2
2     3  MULTIPOLYGON (((1813177.513 -244398.992, 18131...  P2
3     4  MULTIPOLYGON (((1745369.818 -298945.761, 17453...  P2
4     5  MULTIPOLYGON (((1940966.624 -274243.069, 19409...  P2

                     timestamp   latitude  longitude  year  month  \
0 2020-01-05 07:50:30.081292799  40.199583 -79.496295  2020      1
1 2020-01-09 09:16:06.603628800  40.156435 -77.230625  2020      1
2 2020-01-16 04:50:28.301625600  40.705452 -78.310757  2020      1
3 2020-01-26 07:27:26.824550400  41.820474 -76.526561  2020      1
4 2020-01-31 16:15:44.942918400  40.820036 -78.510162  2020      1

       buff_area
0  219558.394338
1   94096.454716
2  188192.909433
3  282289.364149
4  219558.394338
```

We can double-check what is the unit in **EPSG:2163** projection system:

[11]:
```
buff_space.crs.axis_info[0].unit_name
```

[11]:
```
'metre'
```

The result is "metre". Accordingly, the `buff_area` column represents the buffer-based activity space measured in square meters on a weekly basis. You probably noticed that this example did not separate P1 and P2 in calculating activity space. Currently, the `dissolve` parameter only accept one string, not a list of strings. To get activity space

for each person on a weekly basis is easy, all you need to do is to concatenate `pid` and `week`, then dissolve by the newly created column.

[12]:
```
gdf['person_week'] = gdf['pid'].astype(str) + '_' + gdf['week'].astype(str)
buff_space_person_week = space.buffer_space(gdf, dist=100, dissolve='person_week',␣
↪proj=2163)
buff_space_person_week[['person_week','geometry','year','month','week','buff_area']].
↪head()
```

[12]:
```
  person_week                                           geometry  year  month  \
0        P1_1  MULTIPOLYGON (((1743323.684 -263848.055, 17433...  2020      1
1       P1_10  MULTIPOLYGON (((1910983.134 -311490.071, 19109...  2020      3
2       P1_11  POLYGON ((1639281.583 -176601.784, 1639281.101...  2020      3
3       P1_12  MULTIPOLYGON (((2042446.492 -187461.495, 20424...  2020      3
4       P1_13  MULTIPOLYGON (((1736945.903 -338011.606, 17369...  2020      3

   week      buff_area
0     1   94096.454716
1    10  156827.424527
2    11   31365.484905
3    12   62730.969811
4    13   94096.454716
```

Now you get the activity space for each person on a weekly basis. Likewise, you can easily get each person's activity space on a yearly basis by simply concatenating `pid` and `year`, or activity space on a monthly basis by simply concatenating `pid` and `month`

You can select the columns you are interested and save the GeoDataFrame to a spatial dataset or non-spatial dataset. Here, we save the GeoDataFrame to a `shp` file and a `csv` file.

[13]:
```
buff_space_person_week[['person_week','buff_area','geometry']].to_file('../data/
↪buffer_space.shp')
buff_space_person_week[['person_week','buff_area']].to_csv('../data/buffer_space.csv')
```

```
<ipython-input-13-69566ebaa6bd>:1: UserWarning: Column names longer than 10␣
↪characters will be truncated when saved to ESRI Shapefile.
  buff_space_person_week[['person_week','buff_area','geometry']].to_file('../data/
↪buffer_space.shp')
```

1.4.2 Convex hull-based space

We can also calculate the convex hull-based activity space using the `convex_space` function. The `convex_space` takes three parameters:

- gdf: This is your GeoDataFrame
- group: This is the level of aggregating from which you group points to form polygon, the default value is week
- proj: This is the EPSG identifier you want to use to project your spatial data, the default value is 2163

In this example, we will dissolve the points by `person_week`

[14]:
```
convex_space = space.convex_space(gdf, group='person_week', proj=2163)
```

[15]:
```
convex_space.head()
```

[15]:
```
  person_week                                           geometry  \
0        P1_1  POLYGON ((1743224.165 -263838.253, 1767302.859...
```

(continues on next page)

(continued from previous page)

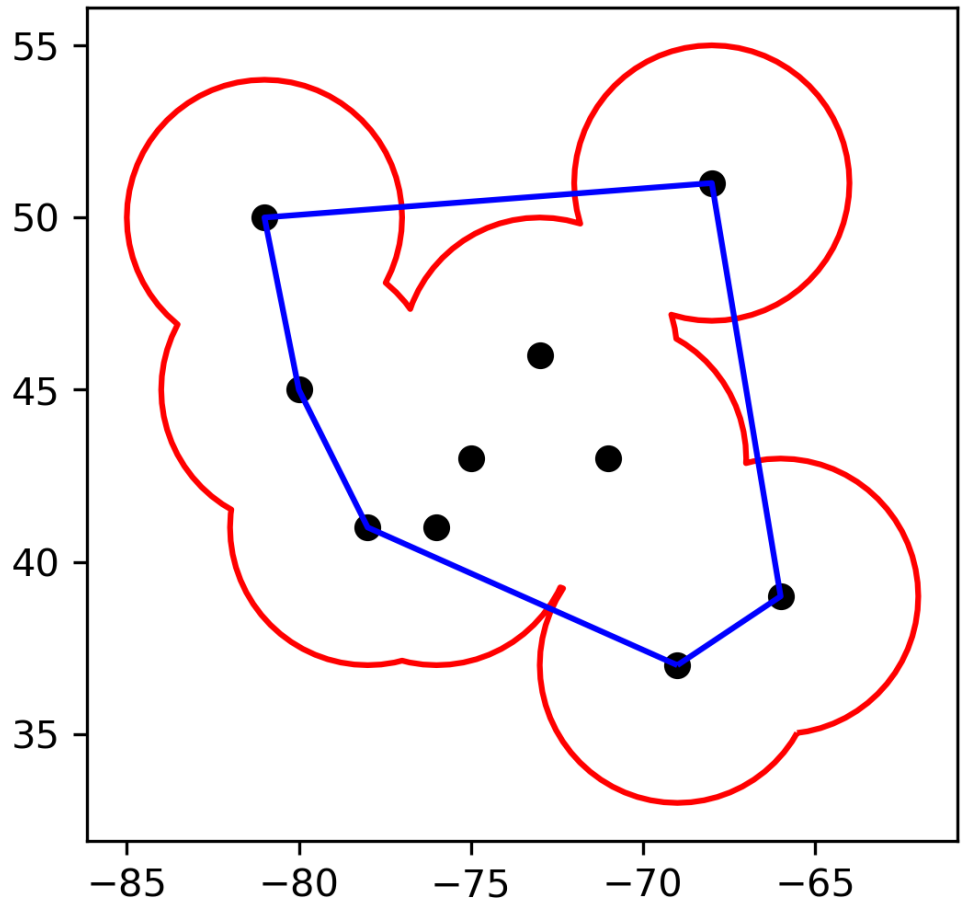
```
1       P1_10  POLYGON ((1910885.055 -311470.562, 1770277.130...
2       P1_11                    POINT (1639181.583 -176601.784)
3       P1_12  LINESTRING (1728441.354 -131257.473, 2042346.9...
4       P1_13  POLYGON ((1736846.385 -338001.805, 1838266.575...

    convex_area pid                       timestamp   latitude  longitude  year  \
0  4.150956e+09  P1 2020-01-05 02:34:34.559443200  40.619272 -79.083519  2020
1  2.779601e+10  P1 2020-03-04 15:38:10.246531200  40.720979 -75.939881  2020
2  0.000000e+00  P1 2020-03-14 21:43:34.435401600  41.605263 -80.050210  2020
3  0.000000e+00  P1 2020-03-19 15:47:51.952416000  41.797624 -78.867266  2020
4  1.331579e+10  P1 2020-03-25 20:27:12.545942400  41.851171 -77.461965  2020

   month  week
0      1     1
1      3    10
2      3    11
3      3    12
4      3    13
```

The `convx_area` column represents the convex hull-based activity space measured in square meters in **EPSG:2163**. This dataset is not perfect, we see that in Week 11 for P1, there is only one point, therefore a Point shape is constructed and 0 is returned for its area. Similarly, in Week 12 for P1, a line shape is constructed rather than a Polygon and 0 is returned for its area. You can also save this GeoDataFrame to a spatial dataset or a non-spatial dataset just like what we have done for the buffer-based activity space measure.

1.4.3 Which method to choose

There are pros and cons of the buffer- and convex hull-based measure of activity space. Knowing how they work will help you understand the process and choose the appropriate method.

The following figure shows the buffer-based activity space (in red color) and convex hull-based activity space (in blue color) from a set of Lat/Long coordinate pairs (in black color). In essence, what buffer-based activity space does is: first, draw a circle around every point using specified buffer distance, then dissolve all the buffers into a single feature to form a Polygon. What convex hull-based activity space does is to line up the outermost points and form the polygon.



The pro of buffer-based activity space is that it works with even only one point where the activity space is eventually the area of the circle. However, you have to specify the buffer distance which sometimes is arbitrary and varies across disciplines.

The convex hull-based activity space does not need any arbitrary parameter. However, if there are two or three points that can line up, it is impossible to form an enclosing shape, and the returned activity space will be 0. In addition, if there are extreme points that are beyond the point cluster, the convex hull-based activity space will be inflated.

The choice of methods depends on which one makes more sense for you research questions and which one is widely acceptable in your field.

Other than the buffer- and convex hull-based minimum bounding box, there are other methods, for example, circle, envelope, concave, etc. We may (or may not) include those methods in our package later on.

1.5 Shared space

Activity space can act as the building box for compiling shared space to indicate interactions by overlaying the activity space at different scales. In the following example, we provide a workflow for building shared space from activity space between P1 and P2 on a weekly basis. We use the buffer-based activity space and redo it with buffer distance of 1000 meter to get more overlapping areas for P1 and P2.

[16]:
```
gdf.head()
```

[16]:
```
  pid                     timestamp   latitude  longitude  \
0  P2 2020-04-27 10:42:22.162176000  40.993799 -76.669419
1  P2 2020-06-02 01:12:45.308505600  39.946904 -78.926234
2  P2 2020-05-08 23:47:33.718185600  41.237403 -79.252317
3  P2 2020-04-26 14:31:12.100310400  41.991390 -77.467769
4  P2 2020-03-31 15:53:27.777897600  41.492674 -76.542921

                      geometry  year  month  week person_week
0  POINT (-76.66942 40.99380)  2020      4    18       P2_18
1  POINT (-78.92623 39.94690)  2020      6    23       P2_23
2  POINT (-79.25232 41.23740)  2020      5    19       P2_19
3  POINT (-77.46777 41.99139)  2020      4    17       P2_17
4  POINT (-76.54292 41.49267)  2020      3    14       P2_14
```

[17]:
```
buff1000m = space.buffer_space(gdf, dist=1000, dissolve='person_week', proj=2163)
```

[18]:
```
buff1000m.head()
```

[18]:
```
  person_week                                           geometry pid  \
0        P1_1  MULTIPOLYGON (((1744219.350 -263936.270, 17442...  P1
1       P1_10  MULTIPOLYGON (((1911865.841 -311665.653, 19118...  P1
2       P1_11  POLYGON ((1640181.583 -176601.784, 1640176.767...  P1
3       P1_12  MULTIPOLYGON (((2043342.159 -187549.710, 20433...  P1
4       P1_13  MULTIPOLYGON (((1737841.570 -338099.822, 17378...  P1

                    timestamp   latitude  longitude  year  month  week  \
0 2020-01-05 02:34:34.559443200  40.619272 -79.083519  2020      1     1
1 2020-03-04 15:38:10.246531200  40.720979 -75.939881  2020      3    10
2 2020-03-14 21:43:34.435401600  41.605263 -80.050210  2020      3    11
3 2020-03-19 15:47:51.952416000  41.797624 -78.867266  2020      3    12
4 2020-03-25 20:27:12.545942400  41.851171 -77.461965  2020      3    13

      buff_area
0  9.409645e+06
1  1.568274e+07
2  3.136548e+06
3  6.273097e+06
4  9.409645e+06
```

[19]:
```
share_space_list = []

for idx, row in buff1000m.iterrows():
    if idx<len(buff1000m.index)-1:
        main_poly = buff1000m.iloc[[idx]]
        other_poly = buff1000m.iloc[(idx+1):]
        share_space = gpd.overlay(main_poly, other_poly, how='intersection')
        share_space['share_space'] = share_space.geometry.area
        share_space_list.append(share_space)
```

[20]:
```
df = pd.concat(share_space_list)
df.shape
```

[20]:
```
(4, 29)
```

[21]:
```
df[['person_week_1','person_week_2','week_1','week_2','share_space']]
```

[21]:
```
  person_week_1 person_week_2  week_1  week_2   share_space
0         P1_18          P2_5    18.0     5.0  9.145960e+04
0         P1_27         P2_12    27.0    12.0  1.439406e+06
0         P1_28          P2_4    28.0     4.0  3.537871e+05
0         P2_12         P2_31    12.0    31.0  3.973167e+05
```

The `share_space` column represents the shared space between P1 and P2 on a weekly basis. Because we compiled the example data randomly, there are not many interactions measured by overlapping activity space on a weekly basis even we use buffer distance of 1000 meters to calculate their activity space. For example, there is a shared space of 9.145960e+04 (~ 91459.6) square meters between P1 at the 18th week of 2020 and P2 at the 5th week of 2020.

Please be aware that the above scripts overlay each row on every another row in the data, so there are duplicates, you can select the rows you want to keep. For example, if you are interested in who have shared space with P1, you can just keep those start with P1 in the `person_week_1` (or `person_week_2`) column and export to whatever data formats you want to work with.

1.6 Measuring Distance

1.6.1 Point to Point distance

In spatial analysis, we often want to know the shortest distance between two features. For example, we may want to know the distance from residence to pharmacy store to see if the distance affects people's health. Or, we may be interested in whether the distance to airport or highway affects population growth. In this example, we will measure the nearest distance to airport in Pennsylvania, USA.

As usual, we need to import libraries we will be using.

[1]:
```
%matplotlib inline

import numpy as np
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
```

Then, we load the example data and airport data and explore the data by plotting them together.

[2]:
```
df = pd.read_csv('../data/example.csv')
```

[3]:
```
from gps2space import geodf
gdf = geodf.df_to_gdf(df, x='longitude', y='latitude')
gdf.head()
```

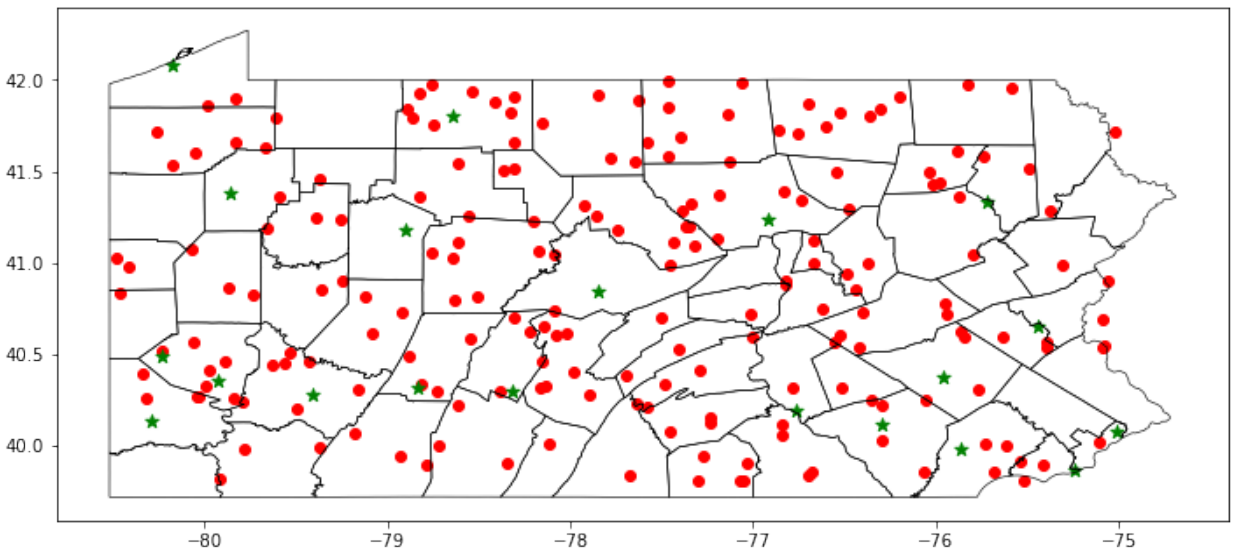
[3]:
```
  pid                       timestamp   latitude  longitude  \
0  P2  2020-04-27 10:42:22.162176000  40.993799 -76.669419
1  P2  2020-06-02 01:12:45.308505600  39.946904 -78.926234
2  P2  2020-05-08 23:47:33.718185600  41.237403 -79.252317
3  P2  2020-04-26 14:31:12.100310400  41.991390 -77.467769
4  P2  2020-03-31 15:53:27.777897600  41.492674 -76.542921

                     geometry
0  POINT (-76.66942 40.99380)
1  POINT (-78.92623 39.94690)
2  POINT (-79.25232 41.23740)
3  POINT (-77.46777 41.99139)
4  POINT (-76.54292 41.49267)
```

[4]:
```
airport = gpd.read_file('../data/paairport.shp')
airport.head()
```

[4]:
```
          STATE                                    NAME  \
0  Pennsylvania                    Erie International
1  Pennsylvania                     Bradford Regional
2  Pennsylvania                      Venango Regional
3  Pennsylvania  Wilkes-Barre/Scranton International
4  Pennsylvania                Williamsport Regional

                      geometry
0  POINT (-80.17600 42.08208)
1  POINT (-78.63987 41.80313)
2  POINT (-79.86014 41.37793)
3  POINT (-75.72390 41.33823)
4  POINT (-76.92144 41.24205)
```

[5]:
```
pacounty = gpd.read_file('../data/pacounty.shp')
```

[6]:
```
ax = pacounty.boundary.plot(figsize=(12, 12), edgecolor='black', linewidth=0.6)
gdf.plot(ax=ax, color='r')
airport.plot(ax=ax, color='g', marker='*', markersize=60)

plt.show();
```



The red dots are the footprints of Person 1 (P1) and Person 2 (P2) while the green stars are the airports in Pennsylvania, USA.

We can calculate the distance from each point of P1 and P2 to the nearest airport using the `dist_to_point` function in the `dist` module. The `dist_to_point` function takes three parameters:

- gdf_a: This is the GeoDataFrame of P1 and P2's footprints
- gdf_b: This is the landmark from where you want to measure the distance
- proj: This is the EPSG identifier you want to use to project your spatial data and will be applied to gdf_a and gdf_b

Because the airport data come from other source, we do not know if it has been projected or what is the projection

system. So we want to check the projection system for airport data.

[7]:
```
airport.crs
```

It returns nothing, which means this data do not have projection. We will give it an initial projection of **EPSG:4326**.

[8]:
```
airport.crs = ("epsg:4326")
```

Now, we can import the `dist` function to calculate the distance from each point of P1 and P2 to the nearest airport.

[9]:
```
from gps2space import dist
```

[10]:
```
dist_to_airport = dist.dist_to_point(gdf, airport, proj=2163)
```

[11]:
```
dist_to_airport.head()
```

[11]:
```
  pid                       timestamp   latitude  longitude  \
0  P2  2020-04-27 10:42:22.162176000  40.993799 -76.669419
1  P2  2020-06-02 01:12:45.308505600  39.946904 -78.926234
2  P2  2020-05-08 23:47:33.718185600  41.237403 -79.252317
3  P2  2020-04-26 14:31:12.100310400  41.991390 -77.467769
4  P2  2020-03-31 15:53:27.777897600  41.492674 -76.542921

                          geometry         STATE                       NAME  \
0  POINT (1926745.083 -169042.499)  Pennsylvania     Williamsport Regional
1  POINT (1774126.223 -333525.438)  Pennsylvania  Johnstown-Cambria County
2  POINT (1712951.727 -200231.269)  Pennsylvania  Du Bois-Jefferson County
3   POINT (1833671.313 -79623.054)  Pennsylvania     Williamsport Regional
4  POINT (1921659.985 -112174.444)  Pennsylvania     Williamsport Regional

     dist2point
0  34579.711173
1  42187.331826
2  30051.080354
3  94804.346362
4  42388.435141
```

The `dist2point` column represents the distance from each point to the nearest airport measured in meters. Likewise, you can then save the GeoDataFrame to a spatial dataset or non-spatial dataset as we did in the last section.

1.6.2 Point to Polygon distance

In this example, we want to calculate the nearest distance to parks represented in polygons using `dist_to_poly` function. The `dist_to_poly` function incorporates R-tree and spatial indexing technologies to boost the nearest neighbor query. The `dist_to_poly` function takes four parameters:

- gdf_source: This is the source GeoPandas dataframe
- gdf_target: This is the target GeoPandas dataframe
- proj: This is the EPSG identifier you want to use to project your spatial data and will be applied to gdf_source and gdf_target
- search_radius: This is the search radius in meters with a default value of None

Please note that:

1. If `search_radius` is specified, points with no neighbors within the search radius, then the `dist_to_poly` function returns a `NaN` value

2. If `search_radius` is not specified, the `dist_to_poly` function employs brute-force search to find the nearest distance, and it may take longer time to calculate the nearest distance, especially for data in larger volumes

As usual, we read the park data as a GeoPandas dataframe. Then we illustrate how `dist_to_poly` works using two examples: an example specifying the `search_radius` and another example without specifying the `search_radius`

[12]:
```
park = gpd.read_file('../data/papark.shp')
park.head()
```

[12]:
```
   park_id                 park_name  park_acres  \
0        1    11th Avenue Playground        1.48
1       22            Alpine Parklet        0.12
2     6117               Negley Park       18.46
3     8202  Deer Lake Community Park       32.76
4     8215         Delano Playground        1.37

                                            geometry
0  POLYGON ((-79.89948 40.40552, -79.89946 40.406...
1  POLYGON ((-80.01282 40.45765, -80.01303 40.457...
2  POLYGON ((-76.89575 40.25092, -76.89178 40.249...
3  MULTIPOLYGON (((-76.05700 40.62615, -76.05696 ...
4  POLYGON ((-75.97098 40.80281, -75.97062 40.801...
```

Specifying `search_radius`

[13]:
```
%%time
dist_with_search_radius = dist.dist_to_poly(gdf, park, proj=2163, search_radius=10000)
```

```
A search_radius of 10000 meters is specified. Points with no neighbors intersected␣
↪with thte search radius will return NaN.
Wall time: 4.57 s
```

[14]:
```
dist_with_search_radius
```

[14]:
```
    pid                       timestamp   latitude  longitude  \
0    P2  2020-04-27 10:42:22.162176000  40.993799 -76.669419
1    P2  2020-06-02 01:12:45.308505600  39.946904 -78.926234
2    P2  2020-05-08 23:47:33.718185600  41.237403 -79.252317
3    P2  2020-04-26 14:31:12.100310400  41.991390 -77.467769
4    P2  2020-03-31 15:53:27.777897600  41.492674 -76.542921
..   ..                            ...        ...        ...
195  P1  2020-04-14 22:59:47.187801600  40.592932 -77.002548
196  P1  2020-02-18 16:00:05.505350400  40.263436 -80.322911
197  P1  2020-02-24 10:22:29.605353600  40.726640 -76.403706
198  P1  2020-01-13 10:02:15.962697600  40.279678 -77.898978
199  P1  2020-04-02 23:09:49.639881600  41.660656 -79.830351

                           geometry    dist2poly
0    POINT (1926745.083 -169042.499)  3143.431951
1    POINT (1774126.223 -333525.438)  4007.426442
2    POINT (1712951.727 -200231.269)  7198.553223
3     POINT (1833671.313 -79623.054)  4418.972172
4    POINT (1921659.985 -112174.444)  4997.223163
..                               ...          ...
195  POINT (1912029.573 -220204.526)  2789.935358
```

(continues on next page)

(continued from previous page)

```
196  POINT (1651469.678 -328218.968)  3433.440261
197  POINT (1956064.504 -191577.975)  6104.559787
198  POINT (1848682.909 -274721.379)  2696.384797
199  POINT (1655332.627 -166134.557)  5926.050348

[200 rows x 6 columns]
```

Without specifying `search_radius`

[15]:
```
%%time
dist_no_search_radius = dist.dist_to_poly(gdf, park, proj=2163)
```

```
No search_radius is specified, the calculation may take longer time for datasets in␣
↪large volumes.
Wall time: 17.3 s
```

[16]:
```
dist_no_search_radius
```

[16]:
```
    pid                       timestamp   latitude  longitude  \
0    P2  2020-04-27 10:42:22.162176000  40.993799 -76.669419
1    P2  2020-06-02 01:12:45.308505600  39.946904 -78.926234
2    P2  2020-05-08 23:47:33.718185600  41.237403 -79.252317
3    P2  2020-04-26 14:31:12.100310400  41.991390 -77.467769
4    P2  2020-03-31 15:53:27.777897600  41.492674 -76.542921
..   ..                           ...        ...        ...
195  P1  2020-04-14 22:59:47.187801600  40.592932 -77.002548
196  P1  2020-02-18 16:00:05.505350400  40.263436 -80.322911
197  P1  2020-02-24 10:22:29.605353600  40.726640 -76.403706
198  P1  2020-01-13 10:02:15.962697600  40.279678 -77.898978
199  P1  2020-04-02 23:09:49.639881600  41.660656 -79.830351

                           geometry    dist2poly
0    POINT (1926745.083 -169042.499)  3143.431951
1    POINT (1774126.223 -333525.438)  4007.426442
2    POINT (1712951.727 -200231.269)  7198.553223
3     POINT (1833671.313 -79623.054)  4418.972172
4    POINT (1921659.985 -112174.444)  4997.223163
..                               ...          ...
195  POINT (1912029.573 -220204.526)  2789.935358
196  POINT (1651469.678 -328218.968)  3433.440261
197  POINT (1956064.504 -191577.975)  6104.559787
198  POINT (1848682.909 -274721.379)  2696.384797
199  POINT (1655332.627 -166134.557)  5926.050348

[200 rows x 6 columns]
```

[18]:
```
dist_no_search_radius[dist_no_search_radius['dist2poly'] == 'NaN']
```

[18]:
```
Empty GeoDataFrame
Columns: [pid, timestamp, latitude, longitude, geometry, dist2poly]
Index: []
```

[23]:
```
dist_with_search_radius.describe().T
```

[23]:
```
            count         mean          std        min          25%  \
latitude    200.0    40.878765     0.649778  39.807771    40.321969
longitude   200.0   -77.732011     1.465171 -80.485216   -78.824666
dist2poly   185.0  3919.033243  2802.089284   0.000000  1550.077077

                    50%          75%           max
latitude      40.821446    41.468584     41.991390
longitude    -77.635756   -76.549980    -75.025528
dist2poly   3623.525457  5316.719387  13855.516463
```

[20]:
```
dist_no_search_radius.describe().T
```

[20]:
```
            count         mean          std        min          25%  \
latitude    200.0    40.878765     0.649778  39.807771    40.321969
longitude   200.0   -77.732011     1.465171 -80.485216   -78.824666
dist2poly   200.0  4726.981887  4027.497246   0.000000  1849.390860

                    50%          75%           max
latitude      40.821446    41.468584     41.991390
longitude    -77.635756   -76.549980    -75.025528
dist2poly   3877.189734  5970.677707  21244.390382
```

The above results show that specifying a search radius decreases the time needed for the nearest distance calculation, and most of the points have neighbors within the search radius, the final results are similar.

1.7 Tips

- Make sure you treat latitude and longitude columns correctly when building spatial data from raw Lat/Long coordinate pairs
- Make sure to double-check your projection Coordinate Reference System (CRS) and make sure that that projection CRS is appropriate for your data
- Most spatial operations require the spatial data are in the same CRS, make sure you double check the CRS before you conduct spatial operations such as overlay, spatial join, and distance query
- Make sure you know what is the unit measure of your projection. See here for more information about projection
- Think about what the results will look like beforehand. If the results are different from what you have expected, then you know there are something wrong either in you mind or in the program
- Most importantly, backup your scripts and data frequently or use version control

1.8 Useful Links

- Pandas
- GeoPandas
- Shapely
- StackOverflow

1.9 Contributing

Issues, suggestions, and contributions are welcome. You can submit issues, suggestions, and your thoughts on the GitHub Repository. You can also contact the author through Email.

1.10 Authors

- Shuai Zhou (Department of Agricultural Economics, Sociology, and Education, Penn State University) is the key developer and corresponding author of the package.
- Yanling Li (Department of Human Development and Family Studies, Penn State University) helped test the package with simulated and usage examples.
- Guangqing Chi (Department of Agricultural Economics, Sociology, and Education, Penn State University), Sy-Miin Chow (Department of Human Development and Family Studies, Penn State University) helped conceptualize the scope and functionality of the package, and provided feedback for testing and improving the package.
- Yosef Bodovski (Population Research Institute, Penn State University) provided usage examples and feedback to improve and refine the functions in this package.

Development and sharing of this publicly available package was supported by National Institutes of Health grants U24AA027684, U01DA046413 (SV/NF) and P2C HD041025, National Science Foundation grants BCS-1052736, IGE-1806874, and SES-1823633, and the Pennsylvania State University Quantitative Social Sciences Initiative and UL TR000127 from the National Center for Advancing Translational Sciences.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search