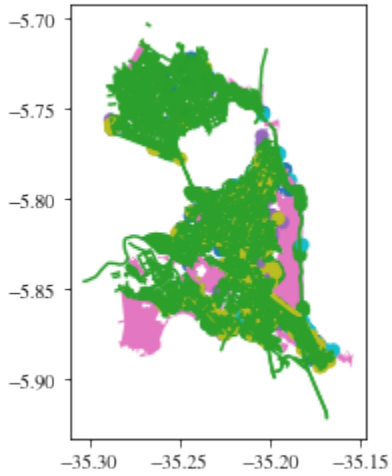
Geohunter

May 02, 2021

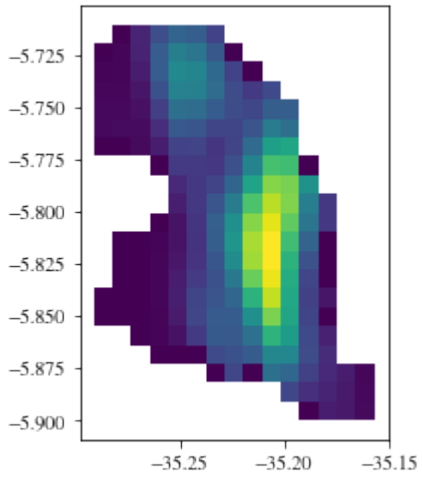
Contents

Geohunter is a python package for parsing and analyzing data from the Overpass API in a pandas-like programming framework. There are similar packages to ours, but our purpose is to create a bridge between data from the OpenStreetMap platform to the geopandas data structures, which offer lots of useful tools for geospatial data analysis.

This package was originally created with the purpose of providing volunteered geographic information to machine learning pipelines. Data from points-of-interest of the city may represent an important source for feature extraction. There are cases where the density of geographic information may help predict a particular variable. For example, consider a set of various points-of-interest.



In our package, we provide some geographic feature extraction procedures with the function *util.kde_interpolation* (see result in figure below), based on the Scipy implementation of KDE. See the plot below with an example for schools density.



If you find any opportunity to do so, feel free to open an issue, fork the repository and contribute to this project.

CHAPTER 1

Package modules

1.1 Geohunter.osm

geohunter.osm

This module wraps requests to OpenStreetMap's Overpass API with an interface for the GeoPandas data structures. The OpenStreetMap has a data model based on nodes, ways and relations. The geometric data structures available in geopandas are points, lines and polygons (but also multipoints, multilines and multipolygons).

For a complete list of data categories available ("map features"), please look the OpenStreetMap.

class geohunter.osm.**Eagle**

Bases: object

Eagle is the facade for requesting data given the map keys available with the *request_overpass()* method. This class also implements a *get()* method which return the data required in a single pandas DataFrame that has a geometric attribute that makes it a geopandas GeoDataFrame (please consult geopandas documentation for more details).

close()

debug__find_geom_not_being_successfully_parsed(*bbox*, *key*, *item*)

get(**args*, ***kwargs*)

request_overpass(*bbox*, *map_feature_key*, *map_feature_item*)

Return the json resulted from *a single* request on Overpass API.

It generates the Overpass QL query from the map features defined, including nodes, ways and relations, and request data from the API. Please consult OpenStreetMap documentation (https://wiki.openstreetmap.org/wiki/Map_Features) for a full list of map features available.

Parameters

- **bbox** (*str or geopandas.GeoDataFrame*) – If str, follow the structure (south_lat,west_lon,north_lat,east_lon), but if you prefer to pass a geopandas.GeoDataFrame, the bbox will be defined as the maximum and minimum values delimited by the geometry.

- **map_feature_key** (*str*) – Map key item from OpenStreetMap, such as "amenity", "highway" etc.
- **map_feature_item** (*str*) –

Returns Data requested in output format of Overpass API.

Return type dict

geohunter.osm.**overpass_result_to_geodf**(*result*, *as_points=False*)
Transforms the result from Overpass request to GeoDataFrame.

geohunter.osm.**parse_geometry**(*x_elements_df*)
Transforms coordinates into shapely objects.

geohunter.osm.**parse_relation**(*x_members*)
Transforms coordinates of 'relation' objects into shapely objects.

geohunter.osm.**requests_retry_session**(*retries=3*, *backoff_factor=0.5*, *session=None*, *status_forcelist=(500*, *503*, *502*, *504)*)

geohunter.osm.**timelog**(*func*)

1.2 Geohunter.util

geohunter.util

This module implements further operations for analyzing OpenStreetMap data.

geohunter.util.**contour_geojson**(*y*, *lonv*, *latv*, *cmin*, *cmax*)
Supports plotting the result of *kde_interpolation*.

geohunter.util.**kde_interpolation**(*poi*, *bw='scott'*, *grid=None*, *resolution=1*, *area=None*, *return_contour_geojson=False*)
Applies kernel density estimation to a set points-of-interest measuring the density estimation on a grid of places (arbitrary points regularly spaced).

Parameters

- **poi** (*GeoDataFrame.*) – Corresponds to input data.
- **bw** (*'scott', 'silverman' or float.*) – The bandwidth for kernel density estimation. Check scipy docs about their bw parameter of gaussian_kde.
- **grid** (*GeoDataFrame or None, default is None.*) – If a grid is not given, then it is provided according to the area parameter and resolution.
- **resolution** (*float, default is 1.*) – Space in kilometers between the arbitrary points of resulting grid.
- **area** (*GeoDataFrame or None, default is None.*) – If area is given, grid will be bounded accordingly with the GeoDataFrame passed.
- **return_contour_geojson** (*bool, default is False.*) – If True, it returns the result of the kde as a contourplot in the geojson format.

Returns

- *GeoDataFrame with a grid of points regularly spaced with the respective*
- *density values for the input points-of-interest given.*

Example

```
>>> import geohunter as gh
>>> poi = gh.osm.Eagle().get(bbox='(-5.91,-35.29,-5.70,-35.15)',
            amenity=['hospital' , 'police'], natural='*')
>>> neighborhood = gh.osm.Eagle().get(bbox='(-5.91,-35.29,-5.70,-35.15)',
            largest_geom=True,
            name='Ponta Negra')
>>> result = kde_interpolation(poi, bw='scott', area=neighborhood, resolution=0.5)
>>> ax = area.plot(edgecolor='black', color='white')
>>> result.plot(column='density', ax=ax)
```

geohunter.util.**make_gridpoints**(*bbox*, *resolution=1*, *return_coords=False*)

It constructs a grid of points regularly spaced.

Parameters

- **bbox** (*str, GeoDataFrame or dict.*) – Corresponds to the boundary box in which the grid will be formed. If a str is provided, it should be in '(S,W,N,E)' format. With a GeoDataFrame, we will use the coordinates of the extremities. Also one can provide a dict with 'south', 'north', 'east', 'west'.
- **resolution** (*float, default is 1.*) – Space between the arbitrary points of resulting grid.
- **return_coords** (*bool*) – If it is wanted to return the coordinate sequences.

geohunter.util.**make_gridsquares**(*city_shape*, *resolution=1*)

It constructs a grid of square cells.

Parameters

- **city_shape** (*GeoDataFrame.*) – Corresponds to the boundary geometry in which the grid will be formed.
- **resolution** (*float, default is 1.*) – Space between the square cells.

geohunter.util.**moran_i_ongrid**(*data*, *coords*, *d_threshold*)

geohunter.util.**parse_bbox**(*bbox*)

Organizes bbox to the standard format used in other places in the package and also in Overpass API.

Parameters **bbox** (*str, GeoDataFrame or dict.*) – Corresponds to the boundary box wanted to be formatted.

Returns

Return type str containing '(S,W,N,E)' coordinates of the bounding box.

geohunter.util.**q_ongrid**(*data*, *grid*, *strata_col*)

CHAPTER 2

Installation

Clone the package, go to the repository directory (where setup.py is) and simply install it with pip.

```
pip install .
```

CHAPTER 3

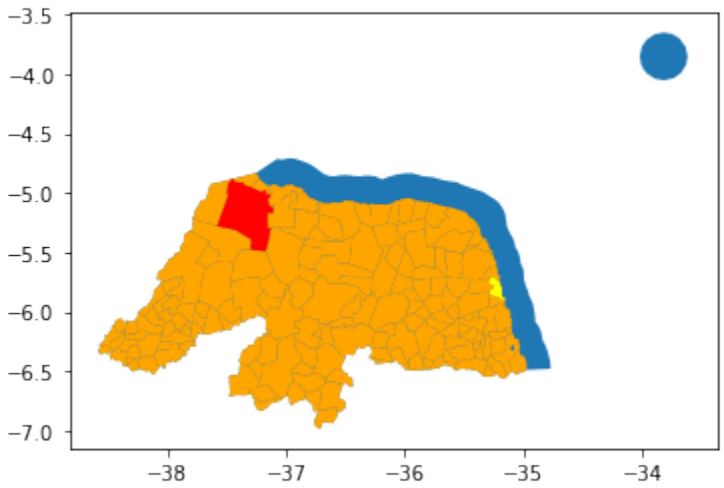
Example

[1]:
```
import geohunter as gh

api = gh.osm.Eagle()


state = api.get('(-8.02, -41.01, -3.0, -33.0)',
             largest_geom=True,
             name='Rio Grande do Norte')

city = api.get('(-8.02, -41.01, -3.0, -33.0)',
             largest_geom=True,
             name='Natal')

poi = api.get(city,
            amenity=['school', 'hospital'],
            highway='primary',
            natural='*')

all_cities = api.get(state, sjoin_op='within',
                   admin_level='8')

biggest_city = api.get(state, sjoin_op='within', largest_geom=True,
                   admin_level='8')
```

```
Geohunter: [TIMELOG] get -- {'largest_geom': True, 'name': 'Rio Grande do Norte'} --␣
↪Completed in 2.8099s
Geohunter: [TIMELOG] get -- {'largest_geom': True, 'name': 'Natal'} -- Completed in 1.
↪2079s
Geohunter: [TIMELOG] get -- {'amenity': ['school', 'hospital'], 'highway': 'primary',
↪'natural': '*'} -- Completed in 5.8873s
Geohunter: [TIMELOG] get -- {'sjoin_op': 'within', 'admin_level': '8'} -- Completed␣
↪in 3.1931s
Geohunter: [TIMELOG] get -- {'sjoin_op': 'within', 'largest_geom': True, 'admin_level
↪': '8'} -- Completed in 2.7193s
```

[2]:
```
ax = state.plot()
all_cities.plot(ax=ax, color='orange')
biggest_city.plot(ax=ax, color='red')
city.plot(ax=ax, color='yellow')
```

[2]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a292cff10>
```



[3]:
```
poi.head()
```

[3]:

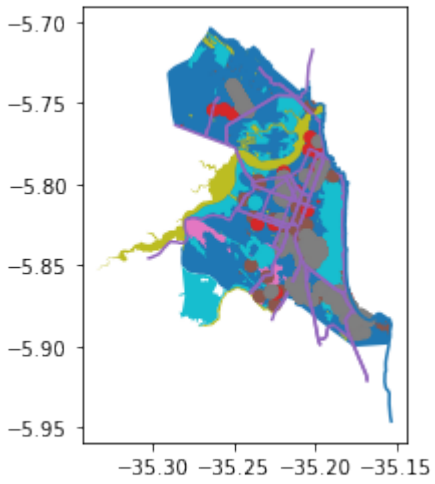
	type	id	tags	\
0	node	501170977	{'amenity': 'school'}	
1	node	501170997	{'amenity': 'school', 'name': 'Centro de Atenç...	
2	node	501784871	{'amenity': 'school'}	
3	node	501784918	{'amenity': 'school', 'name': 'Escola Boa Ideia'}	
4	node	502442042	{'amenity': 'school'}	

	geometry	key	item	\
0	POINT (-35.23427 -5.86902)	amenity	school	
1	POINT (-35.22261 -5.82361)	amenity	school	
2	POINT (-35.21073 -5.81294)	amenity	school	
3	POINT (-35.22495 -5.87417)	amenity	school	
4	POINT (-35.23957 -5.84999)	amenity	school	

	name
0	NaN
1	Centro de Atenção Integral a Criança e ao Adol...
2	NaN
3	Escola Boa Ideia
4	NaN

[5]:
```
ax = city.plot()
poi.plot(ax=ax, column='item')
```

[5]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a2d02a810>
```


−5.70
−5.75
−5.80
−5.85
−5.90
−5.95
−35.30
−35.25
−35.20
−35.15

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index