> Geographic Data Science with PySAL and the pydata stack

Sergio J. Rey

Dani Arribas-Bel

Table of Contents

Install guide

The materials for the workshop and all software packages have been tested on Python 2 and 3 on the following three platforms:

- Linux (Ubuntu-Mate x64)
- Windows 10 (x64)
- Mac OS X (10.15.7 x64).

The workshop depends on the following libraries/versions:

- `numpy>=1.19.5`
- `pandas>=1.1.5`
- `matplotlib>=3.3.4`
- `jupyter>=1.0`
- `seaborn>=0.11.1`
- `pip>=21.0.1`
- `geopandas>=0.8.2`
- `pysal>=2.4.0`
- `libpysal>=4.4.0`
- `cartopy>=0.18.0`
- `pyproj>=2.6.1`
- `shapely>=1.7.1`
- `geopy>=2.1.0`
- `scikit-learn>=0.17.1`
- `bokeh>2.3.0`
- `mplleaflet>=0.0.5`
- `datashader>=0.12.0`
- `geojson>=2.5.0`
- `folium>=0.12.1`
- `statsmodels>=0.12.2`
- `xlrd>=2.0.1`
- `xlsxwriter>=1.3.7`

Linux/Mac OS X

1. Install Anaconda
2. Get the most up to date version:

```
> conda update conda
```

1. Add the `conda-forge` channel:

```
> conda config --add channels conda-forge
```

1. Create an environment named `gds` :

```
> conda create --name gds python=3 pandas numpy matplotlib bokeh seaborn
scikit-learn jupyter statsmodels xlrd xlsxwriter
```

1. Install additional dependencies:

`> conda install --name gds geojson geopandas mplleaflet datashader cartopy folium`

1. To activate and launch the notebook:

```
> source activate gds

> jupyter notebook
```

Windows

1. Install Anaconda3-4.0.0-Windows-x86-64
2. open a cmd window
3. Get the most up to date version:

`> conda update conda`

1. Add the `conda-forge` channel:

`> conda config --add channels conda-forge`

1. Create an environment named `gds` :

`> conda create --name gds pandas numpy matplotlib bokeh seaborn statsmodels scikit-learn jupyter xlrd xlsxwriter geopandas mplleaflet datashader geojson cartopy folium`

1. To activate and launch the notebook:

```
> activate gds

> jupyter notebook
```

Testing

Once installed, you can run the notebook `test.ipynb` placed under `content/infrastructure/test.ipynb` to make sure everything is correctly installed. Follow the instructions in the notebook and, if you do not get any error, you are good to go.

Distribution

> `[URL]` `[PDF]` `[EPUB]` `[MOBI]` `[IPYNB]`

NOTE: for downloads, if a single click does not work, try right click and save file.

License

About the authors

Acknowledgements

This document has also received contributions from the following people:

- Sergio Ray
- Dani Arribas
- Levi John Wolf.
- Wei Kan.

Install guide

The materials for the workshop and all software packages have been tested on Python 2 and 3 on the following three platforms:

- Linux (Ubuntu-Mate x64)
- Windows 10 (x64)
- Mac OS X (10.15.7 x64).

The workshop depends on the following libraries/versions:

- `numpy>=1.19.5`
- `pandas>=1.1.5`
- `matplotlib>=3.3.4`
- `jupyter>=1.0`
- `seaborn>=0.11.1`
- `pip>=21.0.1`
- `geopandas>=0.8.2`
- `pysal>=2.4.0`
- `libpysal>=4.4.0`
- `cartopy>=0.18.0`
- `pyproj>=2.6.1`
- `shapely>=1.7.1`
- `geopy>=2.1.0`
- `scikit-learn>=0.17.1`
- `bokeh>2.3.0`
- `mplleaflet>=0.0.5`
- `datashader>=0.12.0`
- `geojson>=2.5.0`
- `folium>=0.12.1`
- `statsmodels>=0.12.2`
- `xlrd>=2.0.1`
- `xlsxwriter>=1.3.7`

Linux/Mac OS X

1. Install Anaconda
2. Get the most up to date version:

```
> conda update conda
```

1. Add the `conda-forge` channel:

```
> conda config --add channels conda-forge
```

1. Create an environment named `gds` :

```
> conda create --name gds python=3 pandas numpy matplotlib bokeh seaborn
scikit-learn jupyter statsmodels xlrd xlsxwriter
```

1. Install additional dependencies:

`> conda install --name gds geojson geopandas mplleaflet datashader cartopy folium`

1. To activate and launch the notebook:

```
> source activate gds

> jupyter notebook
```

Windows

1. Install Anaconda3-4.0.0-Windows-x86-64
2. open a cmd window
3. Get the most up to date version:

`> conda update conda`

1. Add the `conda-forge` channel:

`> conda config --add channels conda-forge`

1. Create an environment named `gds` :

`> conda create --name gds pandas numpy matplotlib bokeh seaborn statsmodels scikit-learn jupyter xlrd xlsxwriter geopandas mplleaflet datashader geojson cartopy folium`

1. To activate and launch the notebook:

```
> activate gds

> jupyter notebook
```

Testing

Once installed, you can run the notebook `test.ipynb` placed under `content/infrastructure/test.ipynb` to make sure everything is correctly installed. Follow the instructions in the notebook and, if you do not get any error, you are good to go.

Outline

Part I

1. Software and Tools Installation (10 min)
2. Spatial data processing with PySAL (45 min)
    a. Input-output
    b. Visualization and Mapping
    c. Spatial weights
3. Exercise (10 min)
4. ESDA with PySAL (45 min)
    a. Global Autocorrelation
    b. Local Autocorrelation
    c. Space-Time exploratory analysis
5. Exercise (10 min)

Part II

1. Point Patterns (30 min)
    a. Point visualization
    b. Kernel Density Estimation
2. Exercise (10 min)
3. Spatial clustering (30 min)
    a. Geodemographic analysis
    b. Regionalization
4. Exercise (30 min)
5. Spatial Regression (30 min)
    a. Baseline (nonspatial) regression
    b. Exogenous and endogenous spatially lagged regressors
    c. Prediction performance of spatial models
6. Exercise (10 min)

Data

This tutorial makes use of a variety of data sources. Below is a brief description of each dataset as well as the links to the original source where the data was downloaded from. For convenience, we have repackaged the data and included them in the compressed file with the notebooks. You can download it here.

Texas counties

This includes Texas counties from the Census Bureau and a list of attached socio-economic variables. This is an extract of the national cover dataset `NAT` that is part of the example datasets shipped with `PySAL`.

AirBnb listing for Austin (TX)

This dataset contains information for AirBnb properties for the area of Austin (TX). It is originally provided by Inside AirBnb. Same as the source, the dataset is released under a CC0 1.0 Universal License. You can see a summary of the dataset here.

Source: Inside AirBnb's extract of AirBnb locations in Austin (TX).

Path: `data/listings.csv.gz`

Austin Zipcodes

Boundaries for Zipcodes in Austin. The original source is provided by the City of Austin GIS Division.

Source: open data from the city of Austin [url]

Path: `data/Zipcodes.geojson`

Part I

Spatial Data Processing with PySAL & Pandas

IPYNB

```
#by convention, we use these shorter two-letter names
import pysal as ps
import libpysal as lps
import pandas as pd
import numpy as np
```

PySAL has two simple ways to read in data. But, first, you need to get the path from where your notebook is running on your computer to the place the data is. For example, to find where the notebook is running:

```
!pwd # on windows !cd
```

```
/Users/admin/code/gds/content/part1
```

PySAL has a command that it uses to get the paths of its example datasets. Let's work with a commonly-used dataset first.

```
lps.examples.available()
```

```
['10740',
 'arcgis',
 'baltim',
 'book',
 'burkitt',
 'calemp',
 'chicago',
 'columbus',
 'desmith',
 'geodanet',
 'juvenile',
 'Line',
 'mexico',
 'nat',
 'networks',
 'newHaven',
 'Point',
 'Polygon',
 'sacramento2',
 'sids2',
 'snow_maps',
 'south',
 'stl',
 'street_net_pts',
 'taz',
 'us_income',
 'virginia',
 'wmat']
```

```
lps.examples.explain('us_income')
```

```
{'description': 'Per-capita income for the lower 47 US states 1929-2010',
 'explanation': [' * us48.shp: shapefile ',
  ' * us48.dbf: dbf for shapefile',
  ' * us48.shx: index for shapefile',
  ' * usjoin.csv: attribute data (comma delimited file)'],
 'name': 'us_income'}
```

```
csv_path = lps.examples.get_path('usjoin.csv')
```

```
f = lps.io.open(csv_path)
f.header[0:10]
```

```
['Name',
 'STATE_FIPS',
 '1929',
 '1930',
 '1931',
 '1932',
 '1933',
 '1934',
 '1935',
 '1936']
```

```
y2009 = f.by_col('2009')
```

```
y2009[0:10]
```

```
[32274, 32077, 31493, 40902, 40093, 52736, 40135, 36565, 33086, 30987]
```

Working with shapefiles

We can also work with local files outside the built-in examples.

To read in a shapefile, we will need the path to the file.

```
shp_path = 'data/texas.shp'
print(shp_path)
```

```
data/texas.shp
```

Then, we open the file using the `lps.io.open` command:

```
f = lps.io.open(shp_path)
```

`f` is what we call a "file handle." That means that it only *points* to the data and provides ways to work with it. By itself, it does not read the whole dataset into memory. To see basic information about the file, we can use a few different methods.

For instance, the header of the file, which contains most of the metadata about the file:

```
f.header
```

```
{'BBOX Mmax': 0.0,
 'BBOX Mmin': 0.0,
 'BBOX Xmax': -93.50721740722656,
 'BBOX Xmin': -106.6495132446289,
 'BBOX Ymax': 36.49387741088867,
 'BBOX Ymin': 25.845197677612305,
 'BBOX Zmax': 0.0,
 'BBOX Zmin': 0.0,
 'File Code': 9994,
 'File Length': 49902,
 'Shape Type': 5,
 'Unused0': 0,
 'Unused1': 0,
 'Unused2': 0,
 'Unused3': 0,
 'Unused4': 0,
 'Version': 1000}
```

To actually read in the shapes from memory, you can use the following commands:

```
f.by_row(14) #gets the 14th shape from the file
```

```
<pysal.cg.shapes.Polygon at 0x10d8baa20>
```

```
all_polygons = f.read() #reads in all polygons from memory
```

```
len(all_polygons)
```

```
254
```

So, all 254 polygons have been read in from file. These are stored in PySAL shape objects, which can be used by PySAL and can be converted to other Python shape objects.

They typically have a few methods. So, since we've read in polygonal data, we can get some properties about the polygons. Let's just have a look at the first polygon:

```
all_polygons[0:5]
```

```
[<pysal.cg.shapes.Polygon at 0x10d8baba8>,
 <pysal.cg.shapes.Polygon at 0x10d8ba908>,
 <pysal.cg.shapes.Polygon at 0x10d8ba860>,
 <pysal.cg.shapes.Polygon at 0x10d8ba8d0>,
 <pysal.cg.shapes.Polygon at 0x10d8baa90>]
```

```
all_polygons[0].centroid #the centroid of the first polygon
```

```
(-100.27156110567945, 36.27508640938005)
```

```
all_polygons[0].area
```

```
0.23682222998468205
```

```
all_polygons[0].perimeter
```

```
1.9582821721538344
```

While in the Jupyter Notebook, you can examine what properties an object has by using the tab key.

```
polygon = all_polygons[0]
```

```
polygon. #press tab when the cursor is right after the dot
```

```
  File "<ipython-input-20-aa03438a2fa8>", line 1
    polygon. #press tab when the cursor is right after the dot
                                                              ^
SyntaxError: invalid syntax
```

Working with Data Tables

```
dbf_path = "data/texas.dbf"
print(dbf_path)
```

```
data/texas.dbf
```

When you're working with tables of data, like a `csv` or `dbf`, you can extract your data in the following way. Let's open the dbf file we got the path for above.

```
f = lps.io.open(dbf_path)
```

Just like with the shapefile, we can examine the header of the dbf file.

```
f.header
```

```
['NAME',
 'STATE_NAME',
 'STATE_FIPS',
 'CNTY_FIPS',
 'FIPS',
 'STFIPS',
 'COFIPS',
 'FIPSNO',
 'SOUTH',
 'HR60',
 'HR70',
 'HR80',
 'HR90',
 'HC60',
 'HC70',
 'HC80',
 'HC90',
 'PO60',
 'PO70',
 'PO80',
 'PO90',
 'RD60',
 'RD70',
 'RD80',
 'RD90',
 'PS60',
 'PS70',
 'PS80',
 'PS90',
 'UE60',
 'UE70',
 'UE80',
 'UE90',
 'DV60',
 'DV70',
 'DV80',
 'DV90',
 'MA60',
 'MA70',
 'MA80',
 'MA90',
 'POL60',
 'POL70',
 'POL80',
 'POL90',
 'DNL60',
 'DNL70',
 'DNL80',
 'DNL90',
 'MFIL59',
 'MFIL69',
 'MFIL79',
 'MFIL89',
 'FP59',
 'FP69',
 'FP79',
 'FP89',
 'BLK60',
 'BLK70',
 'BLK80',
 'BLK90',
 'GI59',
 'GI69',
 'GI79',
 'GI89',
 'FH60',
 'FH70',
 'FH80',
 'FH90']
```

So, the header is a list containing the names of all of the fields we can read. If we just wanted to grab the data of interest, `HR90`, we can use either `by_col` or `by_col_array`, depending on the format we want the resulting data in:

```
HR90 = f.by_col('HR90')
print(type(HR90).__name__, HR90[0:5])
HR90 = f.by_col_array('HR90')
print(type(HR90).__name__, HR90[0:5])
```

```
list [0.0, 0.0, 18.31166453, 0.0, 3.6517674554]
ndarray [[  0.        ]
 [  0.        ]
 [ 18.31166453]
 [  0.        ]
 [  3.65176746]]
```

As you can see, the `by_col` function returns a list of data, with no shape. It can only return one column at a time:

```
HRs = f.by_col('HR90', 'HR80')
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-25-1fef6a3c3a50> in <module>()
----> 1 HRs = f.by_col('HR90', 'HR80')


TypeError: __call__() takes 2 positional arguments but 3 were given
```

This error message is called a "traceback," as you see in the top right, and it usually provides feedback on why the previous command did not execute correctly. Here, you see that one-too-many arguments was provided to `__call__`, which tells us we cannot pass as many arguments as we did to `by_col`.

If you want to read in many columns at once and store them to an array, use `by_col_array`:

```
HRs = f.by_col_array('HR90', 'HR80')
```

```
HRs[0:10]
```

```
array([[  0.        ,   0.        ],
       [  0.        ,  10.50199538],
       [ 18.31166453,   5.10386362],
       [  0.        ,   0.        ],
       [  3.65176746,  10.4297038 ],
       [  0.        ,   0.        ],
       [  0.        ,  18.85369532],
       [  2.59514448,   6.33617194],
       [  0.        ,   0.        ],
       [  5.59753708,   6.0331825 ]])
```

It is best to use `by_col_array` on data of a single type. That is, if you read in a lot of columns, some of them numbers and some of them strings, all columns will get converted to the same datatype:

```
allcolumns = f.by_col_array(['NAME', 'STATE_NAME', 'HR90', 'HR80'])
```

```
allcolumns
```

```
array([['Lipscomb', 'Texas', '0.0', '0.0'],
       ['Sherman', 'Texas', '0.0', '10.501995379'],
       ['Dallam', 'Texas', '18.31166453', '5.1038636248'],
       ...,
       ['Hidalgo', 'Texas', '7.3003167816', '8.2383277607'],
       ['Willacy', 'Texas', '5.6481219994', '7.6212251119'],
       ['Cameron', 'Texas', '12.302014455', '11.761321464']],
      dtype='<U13')
```

Note that the numerical columns, `HR90` & `HR80` are now considered strings, since they show up with the single tickmarks around them, like `'0.0'`.

These methods work similarly for `.csv` files as well.

Using Pandas with PySAL

A new functionality added to PySAL recently allows you to work with shapefile/dbf pairs using Pandas. This *optional* extension is only turned on if you have Pandas installed. The extension is the `lps.io` module:

```
lps.io
```

```
<module 'pysal.contrib.pdutilities' from '/Users/dani/anaconda/envs/gds-scipy1
```

To use it, you can read in shapefile/dbf pairs using the `ps.pdio.read_files` command.

```
shp_path = lps.examples.get_path('virginia.shp')
data_table = lps.io.shp_file(shp_path)
```

This reads in *the entire database table* and adds a column to the end, called `geometry`, that stores the geometries read in from the shapefile.

Now, you can work with it like a standard pandas dataframe.

```
data_table.header
```

{'File Code': 9994, 'Unused0': 0, 'Unused1': 0, 'Unused2': 0, 'Unused3': 0, 'Unused4': 0, 'File Length': 35708, 'Version': 1000, 'Shape Type': 5, 'BBOX Xmin': -83.67526245117188, 'BBOX Ymin': 36.541481018066406, 'BBOX Xmax': -75.24258422851562, 'BBOX Ymax': 39.45690155029297, 'BBOX Zmin': 0.0, 'BBOX Zmax': 0.0, 'BBOX Mmin': 0.0, 'BBOX Mmax': 0.0}

The `read_files` function only works on shapefile/dbf pairs. If you need to read in data using CSVs, use pandas directly:

```
usjoin = pd.read_csv(csv_path)
#usjoin = ps.pdio.read_files(csv_path) #will not work, not a shp/dbf pair
```

```
usjoin.head()
```

	Name	**STATE_FIPS**	**1929**	**1930**	**1931**	**1932**	**19**
0	Alabama	1	323	267	224	162	16
1	Arizona	4	600	520	429	321	30
2	Arkansas	5	310	228	215	157	15
3	California	6	991	887	749	580	54
4	Colorado	8	634	578	471	354	35

5 rows × 83 columns

The nice thing about working with pandas dataframes is that they have very powerful baked-in support for relational-style queries. By this, I mean that it is very easy to find things like:

The number of counties in each state:

```
data_table.groupby("STATE_NAME").size()
```

```
STATE_NAME
Alabama                  67
Arizona                  14
Arkansas                 75
California               58
Colorado                 63
Connecticut               8
Delaware                  3
District of Columbia      1
Florida                  67
Georgia                 159
Idaho                    44
Illinois                102
Indiana                  92
Iowa                     99
Kansas                  105
Kentucky                120
Louisiana                64
Maine                    16
Maryland                 24
Massachusetts            12
Michigan                 83
Minnesota                87
Mississippi              82
Missouri                115
Montana                  55
Nebraska                 93
Nevada                   17
New Hampshire            10
New Jersey               21
New Mexico               32
New York                 58
North Carolina          100
North Dakota             53
Ohio                     88
Oklahoma                 77
Oregon                   36
Pennsylvania             67
Rhode Island              5
South Carolina           46
South Dakota             66
Tennessee                95
Texas                   254
Utah                     29
Vermont                  14
Virginia                123
Washington               38
West Virginia            55
Wisconsin                70
Wyoming                  23
dtype: int64
```

Or, to get the rows of the table that are in Arizona, we can use the `query` function of the dataframe:

```
data_table.query('STATE_NAME == "Arizona"')
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
1707	Navajo	Arizona	04	017
1708	Coconino	Arizona	04	005
1722	Mohave	Arizona	04	015
1726	Apache	Arizona	04	001
2002	Yavapai	Arizona	04	025
2182	Gila	Arizona	04	007
2262	Maricopa	Arizona	04	013
2311	Greenlee	Arizona	04	011
2326	Graham	Arizona	04	009
2353	Pinal	Arizona	04	021
2499	Pima	Arizona	04	019
2514	Cochise	Arizona	04	003
2615	Santa Cruz	Arizona	04	023
3080	La Paz	Arizona	04	012

14 rows × 70 columns

Behind the scenes, this uses a fast vectorized library, `numexpr`, to essentially do the following.

First, compare each row's `STATE_NAME` column to `'Arizona'` and return `True` if the row matches:

```
data_table.STATE_NAME == 'Arizona'
```

```
0       False
1       False
2       False
3       False
4       False
5       False
6       False
7       False
8       False
9       False
10      False
11      False
12      False
13      False
14      False
15      False
16      False
17      False
18      False
19      False
20      False
21      False
22      False
23      False
24      False
25      False
26      False
27      False
28      False
29      False
        ...
3055    False
3056    False
3057    False
3058    False
3059    False
3060    False
3061    False
3062    False
3063    False
3064    False
3065    False
3066    False
3067    False
3068    False
3069    False
3070    False
3071    False
3072    False
3073    False
3074    False
3075    False
3076    False
3077    False
3078    False
3079    False
3080     True
3081    False
3082    False
3083    False
3084    False
Name: STATE_NAME, dtype: bool
```

Then, use that to filter out rows where the condition is true:

```
data_table[data_table.STATE_NAME == 'Arizona']
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
1707	Navajo	Arizona	04	017
1708	Coconino	Arizona	04	005
1722	Mohave	Arizona	04	015
1726	Apache	Arizona	04	001
2002	Yavapai	Arizona	04	025
2182	Gila	Arizona	04	007
2262	Maricopa	Arizona	04	013
2311	Greenlee	Arizona	04	011
2326	Graham	Arizona	04	009
2353	Pinal	Arizona	04	021
2499	Pima	Arizona	04	019
2514	Cochise	Arizona	04	003
2615	Santa Cruz	Arizona	04	023
3080	La Paz	Arizona	04	012

14 rows × 70 columns

We might need this behind the scenes knowledge when we want to chain together conditions, or when we need to do spatial queries.

This is because spatial queries are somewhat more complex. Let's say, for example, we want all of the counties in the US to the West of `-121` longitude. We need a way to express that question. Ideally, we want something like:

```
SELECT
        *
FROM
        data_table
WHERE
        x_centroid < -121
```

So, let's refer to an arbitrary polygon in the the dataframe's geometry column as `poly`. The centroid of a PySAL polygon is stored as an `(X,Y)` pair, so the longitude is the first element of the pair, `poly.centroid[0]`.

Then, applying this condition to each geometry, we get the same kind of filter we used above to grab only counties in Arizona:

```
data_table.geometry.apply(lambda x: x.centroid[0] < -121)\
                   .head()
```

```
0    False
1    False
2    False
3    False
4    False
Name: geometry, dtype: bool
```

If we use this as a filter on the table, we can get only the rows that match that condition, just like we did for the `STATE_NAME` query:

```
data_table[data_table.geometry.apply(lambda x: x.centroid[0] < -119)].head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	I
3	Okanogan	Washington	53	047	5
27	Whatcom	Washington	53	073	5
31	Skagit	Washington	53	057	5
42	Chelan	Washington	53	007	5
44	Clallam	Washington	53	009	5

5 rows × 70 columns

```
len(data_table[data_table.geometry.apply(lambda x: x.centroid[0] < -119)]) #ho
```

```
109
```

Other types of spatial queries

Everybody knows the following statements are true:

1. If you head directly west from Reno, Nevada, you will shortly enter California.
2. San Diego is in California.

But what does this tell us about the location of San Diego relative to Reno?

Or for that matter, how many counties in California are to the east of Reno?

```
geom = data_table.query('(NAME == "Washoe") & (STATE_NAME == "Nevada")').geome
```

```
lon,lat = geom.values[0].centroid
```

```
lon
```

```
-119.6555030699793
```

```
cal_counties = data_table.query('(STATE_NAME=="California")')
```

```
cal_counties[cal_counties.geometry.apply(lambda x: x.centroid[0] > lon)]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
1312	Mono	California	06	051
1591	Fresno	California	06	019
1620	Inyo	California	06	027
1765	Tulare	California	06	107
1956	Kern	California	06	029
1957	San Bernardino	California	06	071
2117	Ventura	California	06	111
2255	Riverside	California	06	065
2279	Orange	California	06	059
2344	San Diego	California	06	073
2351	Los Angeles	California	06	037
2358	Imperial	California	06	025

12 rows × 70 columns

```
len(cal_counties)
```

```
58
```

This works on any type of spatial query.

For instance, if we wanted to find all of the counties that are within a threshold distance from an observation's centroid, we can do it in the following way.

But first, we need to handle distance calculations on the earth's surface.

```
from math import radians, sin, cos, sqrt, asin

def gcd(loc1, loc2, R=3961):
    """Great circle distance via Haversine formula

    Parameters
    ----------

    loc1: tuple (long, lat in decimal degrees)

    loc2: tuple (long, lat in decimal degrees)

    R: Radius of the earth (3961 miles, 6367 km)

    Returns
    -------
    great circle distance between loc1 and loc2 in units of R


    Notes
    ------
    Does not take into account non-spheroidal shape of the Earth



    >>> san_diego = -117.1611, 32.7157
    >>> austin = -97.7431, 30.2672
    >>> gcd(san_diego, austin)
    1155.474644164695


    """
    lon1, lat1 = loc1
    lon2, lat2 = loc2
    dLat = radians(lat2 - lat1)
    dLon = radians(lon2 - lon1)
    lat1 = radians(lat1)
    lat2 = radians(lat2)

    a = sin(dLat/2)**2 + cos(lat1)*cos(lat2)*sin(dLon/2)**2
    c = 2*asin(sqrt(a))

    return R * c

def gcdm(loc1, loc2):
    return gcd(loc1, loc2)

def gcdk(loc1, loc2):
    return gcd(loc1, loc2, 6367 )
```

```
san_diego = -117.1611, 32.7157
austin = -97.7431, 30.2672
gcd(san_diego, austin)
```

```
1155.474644164695
```

```
gcdk(san_diego, austin)
```

```
1857.3357887898544
```

```
loc1 = (-117.1611, 0.0)
loc2 = (-118.1611, 0.0)
gcd(loc1, loc2)
```

```
69.13249167149539
```

```
loc1 = (-117.1611, 45.0)
loc2 = (-118.1611, 45.0)
gcd(loc1, loc2)
```

```
48.88374342930467
```

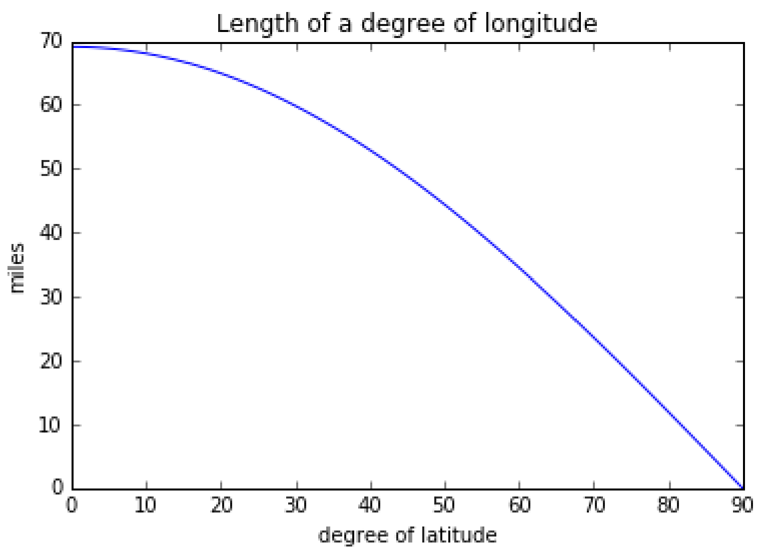
```
loc1 = (-117.1611, 89.0)
loc2 = (-118.1611, 89.0)
gcd(loc1, loc2)
```

```
1.2065130336642724
```

```
lats = range(0, 91)
onedeglon = [ gcd((-117.1611,lat),(-118.1611,lat)) for lat in lats]
```

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(lats, onedeglon)
plt.ylabel('miles')
plt.xlabel('degree of latitude')
plt.title('Length of a degree of longitude')
```

```
<matplotlib.text.Text at 0x114174470>
```



```
san_diego = -117.1611, 32.7157
austin = -97.7431, 30.2672
gcd(san_diego, austin)
```

```
1155.474644164695
```

Now we can use our distance function to pose distance-related queries on our data table.

```
# Find all the counties with centroids within 50 miles of Austin
def near_target_point(polygon, target=austin, threshold=50):
    return gcd(polygon.centroid, target) < threshold

data_table[data_table.geometry.apply(near_target_point)]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
2698	Burnet	Texas	48	053
2716	Williamson	Texas	48	491
2742	Travis	Texas	48	453
2751	Lee	Texas	48	287
2754	Blanco	Texas	48	031
2762	Bastrop	Texas	48	021
2769	Hays	Texas	48	209
2795	Caldwell	Texas	48	055
2798	Comal	Texas	48	091
2808	Guadalupe	Texas	48	187

10 rows × 70 columns

Moving in and out of the dataframe

Most things in PySAL will be explicit about what type their input should be. Most of the time, PySAL functions require either lists or arrays. This is why the file-handler methods are the default IO method in PySAL: the rest of the computational tools are built around their datatypes.

However, it is very easy to get the correct datatype from Pandas using the `values` and `tolist` commands.

`tolist()` will convert its entries to a list. But, it can only be called on individual columns (called `Series` in `pandas` documentation).

So, to turn the `NAME` column into a list:

```
data_table.NAME.tolist()[0:10]
```

```
['Lake of the Woods',
 'Ferry',
 'Stevens',
 'Okanogan',
 'Pend Oreille',
 'Boundary',
 'Lincoln',
 'Flathead',
 'Glacier',
 'Toole']
```

To extract many columns, you must select the columns you want and call their `.values` attribute.

If we were interested in grabbing all of the `HR` variables in the dataframe, we could first select those column names:

```
HRs = [col for col in data_table.columns if col.startswith('HR')]
HRs
```

```
['HR60', 'HR70', 'HR80', 'HR90']
```

We can use this to focus only on the columns we want:

```
data_table[HRs].head()
```

	HR60	HR70	HR80	HR90
0	0.000000	0.000000	8.855827	0.000000
1	0.000000	0.000000	17.208742	15.885624
2	1.863863	1.915158	3.450775	6.462453
3	2.612330	1.288643	3.263814	6.996502
4	0.000000	0.000000	7.770008	7.478033

With this, calling `.values` gives an array containing all of the entries in this subset of the table:

```
data_table[['HR90', 'HR80']].values
```

```
array([[  0.        ,   8.85582713],
       [ 15.88562351,  17.20874204],
       [  6.46245315,   3.4507747 ],
       ...,
       [  4.36732988,   5.2803488 ],
       [  3.72771194,   3.00003    ],
       [  2.04885495,   1.19474313]])
```

Using the PySAL pdio tools means that if you're comfortable with working in Pandas, you can continue to do so.

If you're more comfortable using Numpy or raw Python to do your data processing, PySAL's IO tools naturally support this.

Exercises

1. Find the county with the western most centroid that is within 1000 miles of Austin.

2. **Find the distance between Austin and that centroid.**

Spatial Data Processing with PySAL & Pandas

IPYNB

```
#by convention, we use these shorter two-letter names
import pysal as ps
import libpysal as lps
import pandas as pd
import numpy as np
```

PySAL has two simple ways to read in data. But, first, you need to get the path from where your notebook is running on your computer to the place the data is. For example, to find where the notebook is running:

```
!pwd # on windows !cd
```

```
/Users/dani/code/gds_scipy16/content/part1
```

PySAL has a command that it uses to get the paths of its example datasets. Let's work with a commonly-used dataset first.

```
lps.examples.available()
```

```
['10740',
 'arcgis',
 'baltim',
 'book',
 'burkitt',
 'calemp',
 'chicago',
 'columbus',
 'desmith',
 'geodanet',
 'juvenile',
 'Line',
 'mexico',
 'nat',
 'networks',
 'newHaven',
 'Point',
 'Polygon',
 'sacramento2',
 'sids2',
 'snow_maps',
 'south',
 'stl',
 'street_net_pts',
 'taz',
 'us_income',
 'virginia',
 'wmat']
```

```
lps.examples.explain('us_income')
```

```
{'description': 'Per-capita income for the lower 47 US states 1929-2010',
 'explanation': [' * us48.shp: shapefile ',
  ' * us48.dbf: dbf for shapefile',
  ' * us48.shx: index for shapefile',
  ' * usjoin.csv: attribute data (comma delimited file)'],
 'name': 'us_income'}
```

```
csv_path = lps.examples.get_path('usjoin.csv')
```

```
f = lps.io.open(csv_path)
f.header[0:10]
```

```
['Name',
 'STATE_FIPS',
 '1929',
 '1930',
 '1931',
 '1932',
 '1933',
 '1934',
 '1935',
 '1936']
```

```
y2009 = f.by_col('2009')
```

```
y2009[0:10]
```

```
[32274, 32077, 31493, 40902, 40093, 52736, 40135, 36565, 33086, 30987]
```

Working with shapefiles

We can also work with local files outside the built-in examples.

To read in a shapefile, we will need the path to the file.

```
shp_path = '../data/texas.shp'
print(shp_path)
```

```
../data/texas.shp
```

Then, we open the file using the `ps.open` command:

```
f = lps.io.open(shp_path)
```

`f` is what we call a "file handle." That means that it only *points* to the data and provides ways to work with it. By itself, it does not read the whole dataset into memory. To see basic information about the file, we can use a few different methods.

For instance, the header of the file, which contains most of the metadata about the file:

```
f.header
```

```
{'BBOX Mmax': 0.0,
 'BBOX Mmin': 0.0,
 'BBOX Xmax': -93.50721740722656,
 'BBOX Xmin': -106.6495132446289,
 'BBOX Ymax': 36.49387741088867,
 'BBOX Ymin': 25.845197677612305,
 'BBOX Zmax': 0.0,
 'BBOX Zmin': 0.0,
 'File Code': 9994,
 'File Length': 49902,
 'Shape Type': 5,
 'Unused0': 0,
 'Unused1': 0,
 'Unused2': 0,
 'Unused3': 0,
 'Unused4': 0,
 'Version': 1000}
```

To actually read in the shapes from memory, you can use the following commands:

```
f.by_row(14) #gets the 14th shape from the file
```

```
<pysal.cg.shapes.Polygon at 0x10d8baa20>
```

```
all_polygons = f.read() #reads in all polygons from memory
```

```
len(all_polygons)
```

```
254
```

So, all 254 polygons have been read in from file. These are stored in PySAL shape objects, which can be used by PySAL and can be converted to other Python shape objects.

They typically have a few methods. So, since we've read in polygonal data, we can get some properties about the polygons. Let's just have a look at the first polygon:

```
all_polygons[0:5]
```

```
[<pysal.cg.shapes.Polygon at 0x10d8baba8>,
 <pysal.cg.shapes.Polygon at 0x10d8ba908>,
 <pysal.cg.shapes.Polygon at 0x10d8ba860>,
 <pysal.cg.shapes.Polygon at 0x10d8ba8d0>,
 <pysal.cg.shapes.Polygon at 0x10d8baa90>]
```

```
all_polygons[0].centroid #the centroid of the first polygon
```

```
(-100.27156110567945, 36.27508640938005)
```

```
all_polygons[0].area
```

```
0.23682222998468205
```

```
all_polygons[0].perimeter
```

```
1.9582821721538344
```

While in the Jupyter Notebook, you can examine what properties an object has by using the tab key.

```
polygon = all_polygons[0]
```

```
polygon. #press tab when the cursor is right after the dot
```

```
  File "<ipython-input-20-aa03438a2fa8>", line 1
    polygon. #press tab when the cursor is right after the dot
                                                              ^
SyntaxError: invalid syntax
```

Working with Data Tables

```
dbf_path = "../data/texas.dbf"
print(dbf_path)
```

```
../data/texas.dbf
```

When you're working with tables of data, like a `csv` or `dbf` , you can extract your data in the following way. Let's open the dbf file we got the path for above.

```
f = ps.open(dbf_path)
```

Just like with the shapefile, we can examine the header of the dbf file.

```
f.header
```

```
['NAME',
 'STATE_NAME',
 'STATE_FIPS',
 'CNTY_FIPS',
 'FIPS',
 'STFIPS',
 'COFIPS',
 'FIPSNO',
 'SOUTH',
 'HR60',
 'HR70',
 'HR80',
 'HR90',
 'HC60',
 'HC70',
 'HC80',
 'HC90',
 'PO60',
 'PO70',
 'PO80',
 'PO90',
 'RD60',
 'RD70',
 'RD80',
 'RD90',
 'PS60',
 'PS70',
 'PS80',
 'PS90',
 'UE60',
 'UE70',
 'UE80',
 'UE90',
 'DV60',
 'DV70',
 'DV80',
 'DV90',
 'MA60',
 'MA70',
 'MA80',
 'MA90',
 'POL60',
 'POL70',
 'POL80',
 'POL90',
 'DNL60',
 'DNL70',
 'DNL80',
 'DNL90',
 'MFIL59',
 'MFIL69',
 'MFIL79',
 'MFIL89',
 'FP59',
 'FP69',
 'FP79',
 'FP89',
 'BLK60',
 'BLK70',
 'BLK80',
 'BLK90',
 'GI59',
 'GI69',
 'GI79',
 'GI89',
 'FH60',
 'FH70',
 'FH80',
 'FH90']
```

So, the header is a list containing the names of all of the fields we can read. If we just wanted to grab the data of interest, `HR90`, we can use either `by_col` or `by_col_array`, depending on the format we want the resulting data in:

```
HR90 = f.by_col('HR90')
print(type(HR90).__name__, HR90[0:5])
HR90 = f.by_col_array('HR90')
print(type(HR90).__name__, HR90[0:5])
```

```
list [0.0, 0.0, 18.31166453, 0.0, 3.6517674554]
ndarray [[  0.        ]
 [  0.        ]
 [ 18.31166453]
 [  0.        ]
 [  3.65176746]]
```

As you can see, the `by_col` function returns a list of data, with no shape. It can only return one column at a time:

```
HRs = f.by_col('HR90', 'HR80')
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-25-1fef6a3c3a50> in <module>()
----> 1 HRs = f.by_col('HR90', 'HR80')


TypeError: __call__() takes 2 positional arguments but 3 were given
```

This error message is called a "traceback," as you see in the top right, and it usually provides feedback on why the previous command did not execute correctly. Here, you see that one-too-many arguments was provided to `__call__`, which tells us we cannot pass as many arguments as we did to `by_col`.

If you want to read in many columns at once and store them to an array, use `by_col_array`:

```
HRs = f.by_col_array('HR90', 'HR80')
```

```
HRs[0:10]
```

```
array([[  0.        ,   0.        ],
       [  0.        ,  10.50199538],
       [ 18.31166453,   5.10386362],
       [  0.        ,   0.        ],
       [  3.65176746,  10.4297038 ],
       [  0.        ,   0.        ],
       [  0.        ,  18.85369532],
       [  2.59514448,   6.33617194],
       [  0.        ,   0.        ],
       [  5.59753708,   6.0331825 ]])
```

It is best to use `by_col_array` on data of a single type. That is, if you read in a lot of columns, some of them numbers and some of them strings, all columns will get converted to the same datatype:

```
allcolumns = f.by_col_array(['NAME', 'STATE_NAME', 'HR90', 'HR80'])
```

```
allcolumns
```

```
array([['Lipscomb', 'Texas', '0.0', '0.0'],
       ['Sherman', 'Texas', '0.0', '10.501995379'],
       ['Dallam', 'Texas', '18.31166453', '5.1038636248'],
       ...,
       ['Hidalgo', 'Texas', '7.3003167816', '8.2383277607'],
       ['Willacy', 'Texas', '5.6481219994', '7.6212251119'],
       ['Cameron', 'Texas', '12.302014455', '11.761321464']],
      dtype='<U13')
```

Note that the numerical columns, `HR90` & `HR80` are now considered strings, since they show up with the single tickmarks around them, like `'0.0'`.

These methods work similarly for `.csv` files as well.

Using Pandas with PySAL

A new functionality added to PySAL recently allows you to work with shapefile/dbf pairs using Pandas. This *optional* extension is only turned on if you have Pandas installed. The extension is the `ps.pdio` module:

```
ps.pdio
```

```
<module 'pysal.contrib.pdutilities' from '/Users/dani/anaconda/envs/gds-scipy1
```

To use it, you can read in shapefile/dbf pairs using the `ps.pdio.read_files` command.

```
shp_path = ps.examples.get_path('NAT.shp')
data_table = ps.pdio.read_files(shp_path)
```

This reads in *the entire database table* and adds a column to the end, called `geometry`, that stores the geometries read in from the shapefile.

Now, you can work with it like a standard pandas dataframe.

```
data_table.head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	F
0	Lake of the Woods	Minnesota	27	077	27
1	Ferry	Washington	53	019	53
2	Stevens	Washington	53	065	53
3	Okanogan	Washington	53	047	53
4	Pend Oreille	Washington	53	051	53

5 rows × 70 columns

The `read_files` function only works on shapefile/dbf pairs. If you need to read in data using CSVs, use pandas directly:

```
usjoin = pd.read_csv(csv_path)
#usjoin = ps.pdio.read_files(csv_path) #will not work, not a shp/dbf pair
```

```
usjoin.head()
```

	Name	STATE_FIPS	1929	1930	1931	1932	19
0	Alabama	1	323	267	224	162	16
1	Arizona	4	600	520	429	321	30
2	Arkansas	5	310	228	215	157	15
3	California	6	991	887	749	580	54
4	Colorado	8	634	578	471	354	35

5 rows × 83 columns

The nice thing about working with pandas dataframes is that they have very powerful baked-in support for relational-style queries. By this, I mean that it is very easy to find things like:

The number of counties in each state:

```
data_table.groupby("STATE_NAME").size()
```

```
STATE_NAME
Alabama                  67
Arizona                  14
Arkansas                 75
California               58
Colorado                 63
Connecticut               8
Delaware                  3
District of Columbia      1
Florida                  67
Georgia                 159
Idaho                    44
Illinois                102
Indiana                  92
Iowa                     99
Kansas                  105
Kentucky                120
Louisiana                64
Maine                    16
Maryland                 24
Massachusetts            12
Michigan                 83
Minnesota                87
Mississippi              82
Missouri                115
Montana                  55
Nebraska                 93
Nevada                   17
New Hampshire            10
New Jersey               21
New Mexico               32
New York                 58
North Carolina          100
North Dakota             53
Ohio                     88
Oklahoma                 77
Oregon                   36
Pennsylvania             67
Rhode Island              5
South Carolina           46
South Dakota             66
Tennessee                95
Texas                   254
Utah                     29
Vermont                  14
Virginia                123
Washington               38
West Virginia            55
Wisconsin                70
Wyoming                  23
dtype: int64
```

Or, to get the rows of the table that are in Arizona, we can use the `query` function of the dataframe:

```
data_table.query('STATE_NAME == "Arizona"')
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
1707	Navajo	Arizona	04	017
1708	Coconino	Arizona	04	005
1722	Mohave	Arizona	04	015
1726	Apache	Arizona	04	001
2002	Yavapai	Arizona	04	025
2182	Gila	Arizona	04	007
2262	Maricopa	Arizona	04	013
2311	Greenlee	Arizona	04	011
2326	Graham	Arizona	04	009
2353	Pinal	Arizona	04	021
2499	Pima	Arizona	04	019
2514	Cochise	Arizona	04	003
2615	Santa Cruz	Arizona	04	023
3080	La Paz	Arizona	04	012

14 rows × 70 columns

Behind the scenes, this uses a fast vectorized library, `numexpr`, to essentially do the following.

First, compare each row's `STATE_NAME` column to `'Arizona'` and return `True` if the row matches:

```
data_table.STATE_NAME == 'Arizona'
```

```
0       False
1       False
2       False
3       False
4       False
5       False
6       False
7       False
8       False
9       False
10      False
11      False
12      False
13      False
14      False
15      False
16      False
17      False
18      False
19      False
20      False
21      False
22      False
23      False
24      False
25      False
26      False
27      False
28      False
29      False
        ...
3055    False
3056    False
3057    False
3058    False
3059    False
3060    False
3061    False
3062    False
3063    False
3064    False
3065    False
3066    False
3067    False
3068    False
3069    False
3070    False
3071    False
3072    False
3073    False
3074    False
3075    False
3076    False
3077    False
3078    False
3079    False
3080     True
3081    False
3082    False
3083    False
3084    False
Name: STATE_NAME, dtype: bool
```

Then, use that to filter out rows where the condition is true:

```
data_table[data_table.STATE_NAME == 'Arizona']
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
1707	Navajo	Arizona	04	017
1708	Coconino	Arizona	04	005
1722	Mohave	Arizona	04	015
1726	Apache	Arizona	04	001
2002	Yavapai	Arizona	04	025
2182	Gila	Arizona	04	007
2262	Maricopa	Arizona	04	013
2311	Greenlee	Arizona	04	011
2326	Graham	Arizona	04	009
2353	Pinal	Arizona	04	021
2499	Pima	Arizona	04	019
2514	Cochise	Arizona	04	003
2615	Santa Cruz	Arizona	04	023
3080	La Paz	Arizona	04	012

14 rows × 70 columns

We might need this behind the scenes knowledge when we want to chain together conditions, or when we need to do spatial queries.

This is because spatial queries are somewhat more complex. Let's say, for example, we want all of the counties in the US to the West of `-121` longitude. We need a way to express that question. Ideally, we want something like:

```
SELECT
        *
FROM
        data_table
WHERE
        x_centroid < -121
```

So, let's refer to an arbitrary polygon in the the dataframe's geometry column as `poly`. The centroid of a PySAL polygon is stored as an `(X,Y)` pair, so the longitude is the first element of the pair, `poly.centroid[0]`.

Then, applying this condition to each geometry, we get the same kind of filter we used above to grab only counties in Arizona:

```
data_table.geometry.apply(lambda x: x.centroid[0] < -121)\
                   .head()
```

```
0    False
1    False
2    False
3    False
4    False
Name: geometry, dtype: bool
```

If we use this as a filter on the table, we can get only the rows that match that condition, just like we did for the `STATE_NAME` query:

```
data_table[data_table.geometry.apply(lambda x: x.centroid[0] < -119)].head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	I
3	Okanogan	Washington	53	047	5
27	Whatcom	Washington	53	073	5
31	Skagit	Washington	53	057	5
42	Chelan	Washington	53	007	5
44	Clallam	Washington	53	009	5

5 rows × 70 columns

```
len(data_table[data_table.geometry.apply(lambda x: x.centroid[0] < -119)]) #ho
```

```
109
```

Other types of spatial queries

Everybody knows the following statements are true:

1. If you head directly west from Reno, Nevada, you will shortly enter California.
2. San Diego is in California.

But what does this tell us about the location of San Diego relative to Reno?

Or for that matter, how many counties in California are to the east of Reno?

```
geom = data_table.query('(NAME == "Washoe") & (STATE_NAME == "Nevada")').geome
```

```
lon,lat = geom.values[0].centroid
```

```
lon
```

```
-119.6555030699793
```

```
cal_counties = data_table.query('(STATE_NAME=="California")')
```

```
cal_counties[cal_counties.geometry.apply(lambda x: x.centroid[0] > lon)]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
1312	Mono	California	06	051
1591	Fresno	California	06	019
1620	Inyo	California	06	027
1765	Tulare	California	06	107
1956	Kern	California	06	029
1957	San Bernardino	California	06	071
2117	Ventura	California	06	111
2255	Riverside	California	06	065
2279	Orange	California	06	059
2344	San Diego	California	06	073
2351	Los Angeles	California	06	037
2358	Imperial	California	06	025

12 rows × 70 columns

```
len(cal_counties)
```

```
58
```

This works on any type of spatial query.

For instance, if we wanted to find all of the counties that are within a threshold distance from an observation's centroid, we can do it in the following way.

But first, we need to handle distance calculations on the earth's surface.

```
from math import radians, sin, cos, sqrt, asin

def gcd(loc1, loc2, R=3961):
    """Great circle distance via Haversine formula

    Parameters
    ----------

    loc1: tuple (long, lat in decimal degrees)

    loc2: tuple (long, lat in decimal degrees)

    R: Radius of the earth (3961 miles, 6367 km)

    Returns
    -------
    great circle distance between loc1 and loc2 in units of R


    Notes
    ------
    Does not take into account non-spheroidal shape of the Earth



    >>> san_diego = -117.1611, 32.7157
    >>> austin = -97.7431, 30.2672
    >>> gcd(san_diego, austin)
    1155.474644164695


    """
    lon1, lat1 = loc1
    lon2, lat2 = loc2
    dLat = radians(lat2 - lat1)
    dLon = radians(lon2 - lon1)
    lat1 = radians(lat1)
    lat2 = radians(lat2)

    a = sin(dLat/2)**2 + cos(lat1)*cos(lat2)*sin(dLon/2)**2
    c = 2*asin(sqrt(a))

    return R * c

def gcdm(loc1, loc2):
    return gcd(loc1, loc2)

def gcdk(loc1, loc2):
    return gcd(loc1, loc2, 6367 )
```

```
san_diego = -117.1611, 32.7157
austin = -97.7431, 30.2672
gcd(san_diego, austin)
```

```
1155.474644164695
```

```
gcdk(san_diego, austin)
```

```
1857.3357887898544
```

```
loc1 = (-117.1611, 0.0)
loc2 = (-118.1611, 0.0)
gcd(loc1, loc2)
```

```
69.13249167149539
```

```
loc1 = (-117.1611, 45.0)
loc2 = (-118.1611, 45.0)
gcd(loc1, loc2)
```

```
48.88374342930467
```

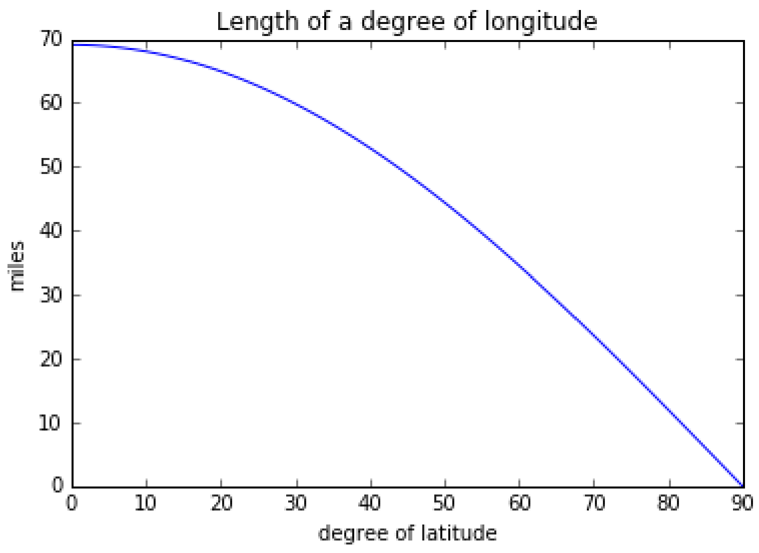
```
loc1 = (-117.1611, 89.0)
loc2 = (-118.1611, 89.0)
gcd(loc1, loc2)
```

```
1.2065130336642724
```

```
lats = range(0, 91)
onedeglon = [ gcd((-117.1611,lat),(-118.1611,lat)) for lat in lats]
```

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(lats, onedeglon)
plt.ylabel('miles')
plt.xlabel('degree of latitude')
plt.title('Length of a degree of longitude')
```

```
<matplotlib.text.Text at 0x114174470>
```



```
san_diego = -117.1611, 32.7157
austin = -97.7431, 30.2672
gcd(san_diego, austin)
```

```
1155.474644164695
```

Now we can use our distance function to pose distance-related queries on our data table.

```
# Find all the counties with centroids within 50 miles of Austin
def near_target_point(polygon, target=austin, threshold=50):
    return gcd(polygon.centroid, target) < threshold

data_table[data_table.geometry.apply(near_target_point)]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
2698	Burnet	Texas	48	053
2716	Williamson	Texas	48	491
2742	Travis	Texas	48	453
2751	Lee	Texas	48	287
2754	Blanco	Texas	48	031
2762	Bastrop	Texas	48	021
2769	Hays	Texas	48	209
2795	Caldwell	Texas	48	055
2798	Comal	Texas	48	091
2808	Guadalupe	Texas	48	187

10 rows × 70 columns

Moving in and out of the dataframe

Most things in PySAL will be explicit about what type their input should be. Most of the time, PySAL functions require either lists or arrays. This is why the file-handler methods are the default IO method in PySAL: the rest of the computational tools are built around their datatypes.

However, it is very easy to get the correct datatype from Pandas using the `values` and `tolist` commands.

`tolist()` will convert its entries to a list. But, it can only be called on individual columns (called `Series` in `pandas` documentation).

So, to turn the `NAME` column into a list:

```
data_table.NAME.tolist()[0:10]
```

```
['Lake of the Woods',
 'Ferry',
 'Stevens',
 'Okanogan',
 'Pend Oreille',
 'Boundary',
 'Lincoln',
 'Flathead',
 'Glacier',
 'Toole']
```

To extract many columns, you must select the columns you want and call their `.values` attribute.

If we were interested in grabbing all of the `HR` variables in the dataframe, we could first select those column names:

```
HRs = [col for col in data_table.columns if col.startswith('HR')]
HRs
```

```
['HR60', 'HR70', 'HR80', 'HR90']
```

We can use this to focus only on the columns we want:

```
data_table[HRs].head()
```

	HR60	HR70	HR80	HR90
0	0.000000	0.000000	8.855827	0.000000
1	0.000000	0.000000	17.208742	15.885624
2	1.863863	1.915158	3.450775	6.462453
3	2.612330	1.288643	3.263814	6.996502
4	0.000000	0.000000	7.770008	7.478033

With this, calling `.values` gives an array containing all of the entries in this subset of the table:

```
data_table[['HR90', 'HR80']].values
```

```
array([[  0.        ,   8.85582713],
       [ 15.88562351,  17.20874204],
       [  6.46245315,   3.4507747 ],
       ...,
       [  4.36732988,   5.2803488 ],
       [  3.72771194,   3.00003   ],
       [  2.04885495,   1.19474313]])
```

Using the PySAL pdio tools means that if you're comfortable with working in Pandas, you can continue to do so.

If you're more comfortable using Numpy or raw Python to do your data processing, PySAL's IO tools naturally support this.

Exercises

1. Find the county with the western most centroid that is within 1000 miles of Austin.
2. Find the distance between Austin and that centroid.

636f30a097e66b464102c7a639d7f5
1915ce5edd

Choropleth Mapping

> `IPYNB`

Introduction

When PySAL was originally planned, the intention was to focus on the computational aspects of exploratory spatial data analysis and spatial econometric methods, while relying on existing GIS packages and visualization libraries for visualization of computations. Indeed, we have partnered with esri and QGIS towards this end.

However, over time we have received many requests for supporting basic geovisualization within PySAL so that the step of having to interoperate with an exertnal package can be avoided, thereby increasing the efficiency of the spatial analytical workflow.

In this notebook, we demonstrate several approaches towards a particular subset of geovisualization methods, namely **choropleth maps**. We start with a self-contained exploratory workflow where no other dependencies beyond PySAL are required. The idea here is to support quick generation of different views of your data to complement the statistical and econometric work in PySAL. Once your work has progressed to the publication stage, we point you to resources that can be used for publication quality output.

We then move on to consider three other packages that can be used in conjunction with PySAL for choropleth mapping:

- geopandas
- folium
- cartopy
- bokeh

PySAL Viz Module

The mapping module in PySAL is organized around three main layers:

- A lower-level layer that reads polygon, line and point shapefiles and returns a Matplotlib collection.
- A medium-level layer that performs some usual transformations on a Matplotlib object (e.g. color code polygons according to a vector of values).
- A higher-level layer intended for end-users for particularly useful cases and style preferences pre-defined (e.g. Create a choropleth).

```
%matplotlib inline
import numpy as np
import pysal as ps
import libpysal as lp
import random as rdm
from splot import mapping as maps
from pylab import *
```

Lower-level component

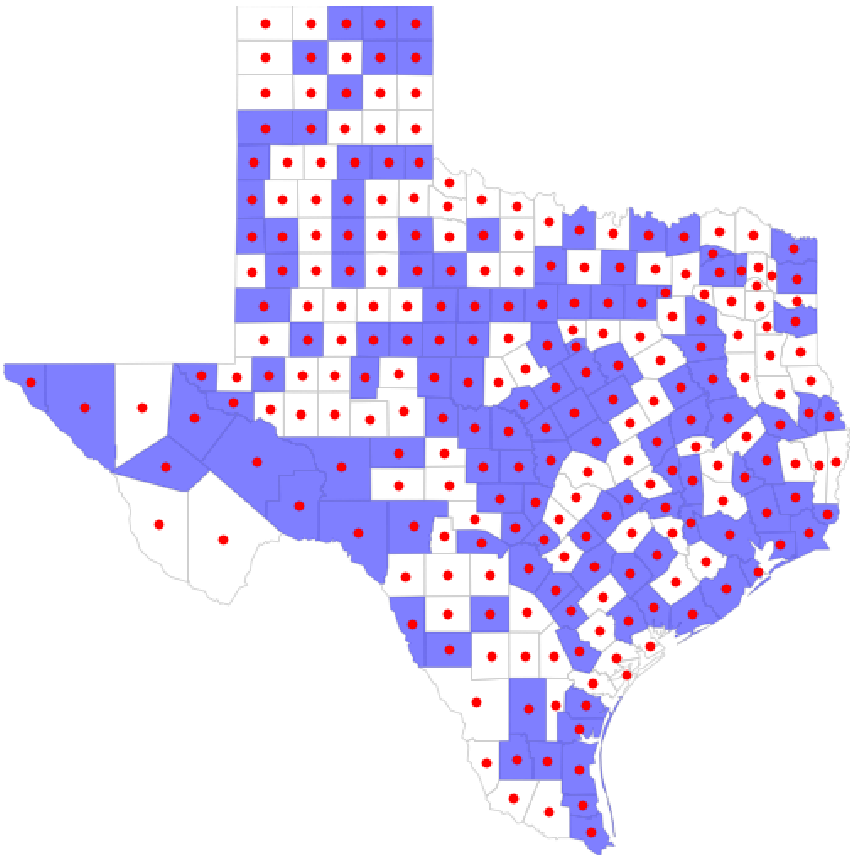
This includes basic functionality to read spatial data from a file (currently only shapefiles supported) and produce rudimentary Matplotlib objects. The main methods are:

- map_poly_shape: to read in polygon shapefiles
- map_line_shape: to read in line shapefiles
- map_point_shape: to read in point shapefiles

These methods all support an option to subset the observations to be plotted (very useful when missing values are present). They can also be overlaid and combined by using the `setup_ax` function. the resulting object is very basic but also very flexible so, for minds used to matplotlib this should be good news as it allows to modify pretty much any property and attribute.

Example

```
shp_link = 'data/texas.shp'
shp = lp.io.open(shp_link)
some = [bool(rdm.getrandbits(1)) for i in lp.io.open(shp_link)]

fig = figure(figsize=(9,9))

base = maps.map_poly_shp(shp)
base.set_facecolor('none')
base.set_linewidth(0.75)
base.set_edgecolor('0.8')
some = maps.map_poly_shp(shp, which=some)
some.set_alpha(0.5)
some.set_linewidth(0.)
cents = np.array([poly.centroid for poly in ps.open(shp_link)])
pts = scatter(cents[:, 0], cents[:, 1])
pts.set_color('red')

ax = maps.setup_ax([base, some, pts], [shp.bbox, shp.bbox, shp.bbox])
fig.add_axes(ax)
show()
```



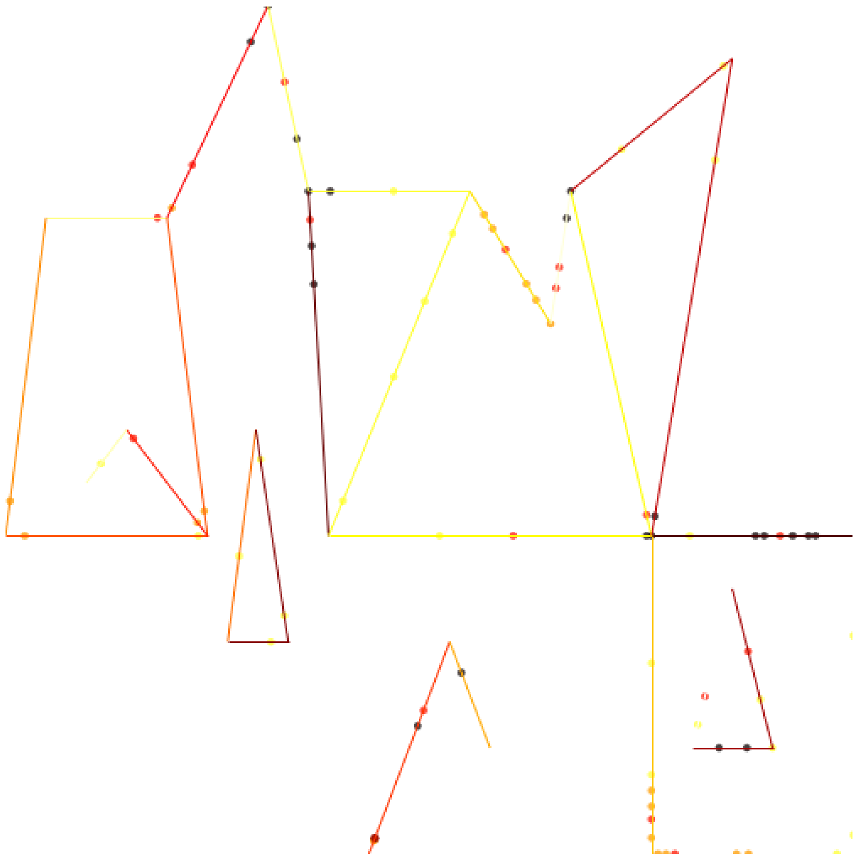
Medium-level component

This layer comprises functions that perform usual transformations on matplotlib objects, such as color coding objects (points, polygons, etc.) according to a series of values. This includes the following methods:

- `base_choropleth_classless`
- `base_choropleth_unique`
- `base_choropleth_classif`

Example

```
net_link = lp.io.examples.get_path('eberly_net.shp')
net = lp.io.open(net_link)
values = np.array(ps.open(net_link.replace('.shp', '.dbf')).by_col('TNODE'))

pts_link = ps.examples.get_path('eberly_net_pts_onnetwork.shp')
pts = lp.io.open(pts_link)

fig = figure(figsize=(9,9))

netm = maps.map_line_shp(net)
netc = maps.base_choropleth_unique(netm, values)

ptsm = maps.map_point_shp(pts)
ptsm = maps.base_choropleth_classif(ptsm, values)
ptsm.set_alpha(0.5)
ptsm.set_linewidth(0.)

ax = maps.setup_ax([netc, ptsm], [net.bbox, net.bbox])
fig.add_axes(ax)
show()
```



```
maps.plot_poly_lines('data/texas.shp')
```

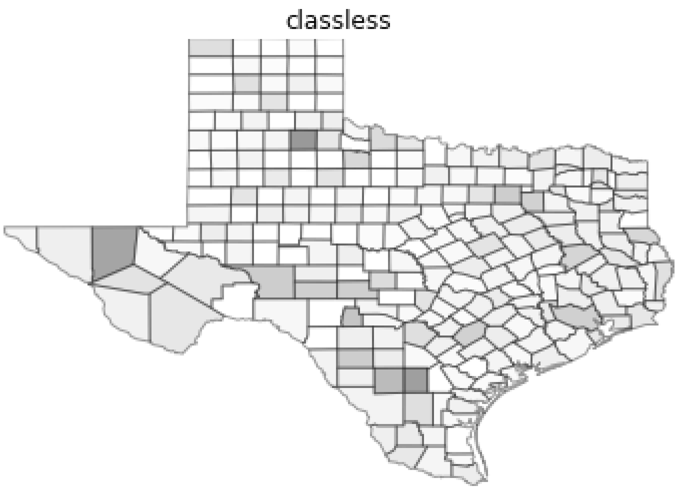
```
callng plt.show()
```

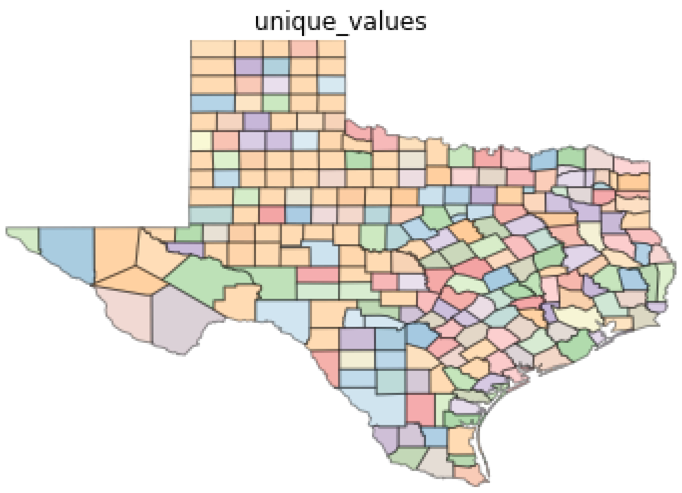
Higher-level component

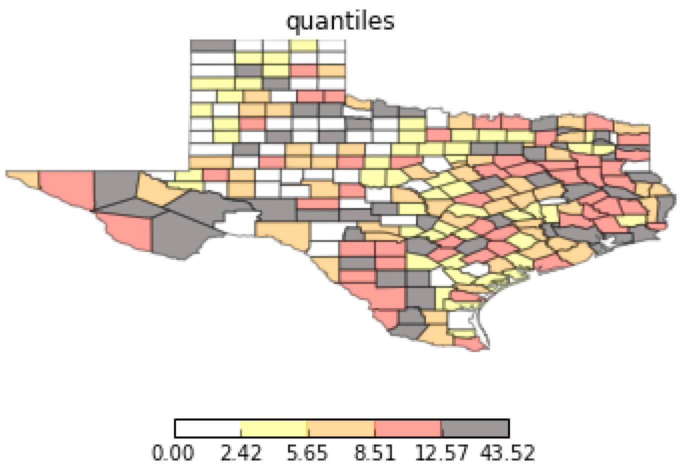
This currently includes the following end-user functions:

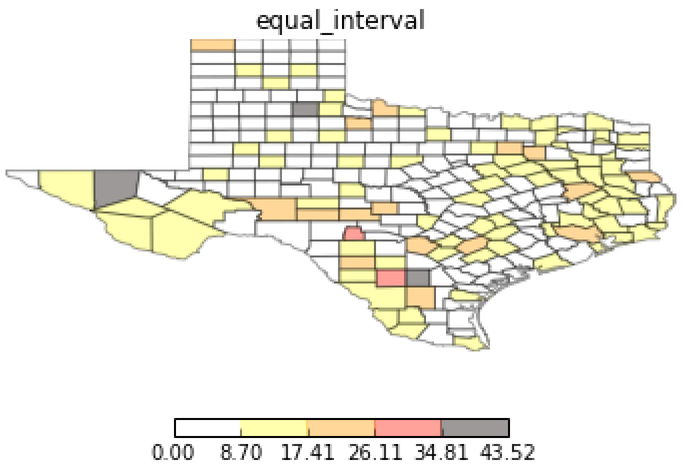
- `plot_poly_lines` : very quick shapfile plotting

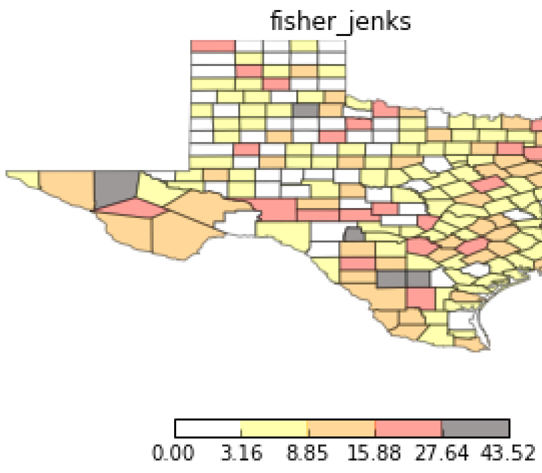
```
shp_link = 'data/texas.shp'
values = np.array(ps.open('../data/texas.dbf').by_col('HR90'))

types = ['classless', 'unique_values', 'quantiles', 'equal_interval', 'fisher_
for typ in types:
    maps.plot_choropleth(shp_link, values, typ, title=typ)
```




unique_values


quantiles
0.00
2.42
5.65
8.51
12.57
43.52


equal_interval
0.00
8.70
17.41
26.11
34.81
43.52



PySAL Map Classifiers

```
hr90 = values
hr90q5 = ps.Quantiles(hr90, k=5)
hr90q5
```

```
               Quantiles

Lower              Upper              Count
=========================================
         x[i] <=  2.421                  51
 2.421 < x[i] <=  5.652                  51
 5.652 < x[i] <=  8.510                  50
 8.510 < x[i] <= 12.571                  51
12.571 < x[i] <= 43.516                  51
```

```
hr90q4 = ps.Quantiles(hr90, k=4)
hr90q4
```

```
               Quantiles

Lower              Upper              Count
=========================================
         x[i] <=  3.918                  64
 3.918 < x[i] <=  7.232                  63
 7.232 < x[i] <= 11.414                  63
11.414 < x[i] <= 43.516                  64
```

```
hr90e5 = ps.Equal_Interval(hr90, k=5)
hr90e5
```

```
              Equal Interval

Lower              Upper               Count
=========================================
         x[i] <=  8.703                 157
 8.703 < x[i] <= 17.406                  76
17.406 < x[i] <= 26.110                  16
26.110 < x[i] <= 34.813                   2
34.813 < x[i] <= 43.516                   3
```

```
hr90fj5 = ps.Fisher_Jenks(hr90, k=5)
hr90fj5
```

```
               Fisher_Jenks

Lower              Upper               Count
=========================================
         x[i] <=  3.156                  55
 3.156 < x[i] <=  8.846                 104
 8.846 < x[i] <= 15.881                  64
15.881 < x[i] <= 27.640                  27
27.640 < x[i] <= 43.516                   4
```

```
hr90fj5.adcm # measure of fit: Absolute deviation around class means
```

```
352.10763138100003
```

```
hr90q5.adcm
```

```
361.5413784392
```

```
hr90e5.adcm
```

```
614.51093704210064
```

```
hr90fj5.yb[0:10] # what bin each value is placed in
```

```
array([0, 0, 3, 0, 1, 0, 0, 0, 0, 1])
```

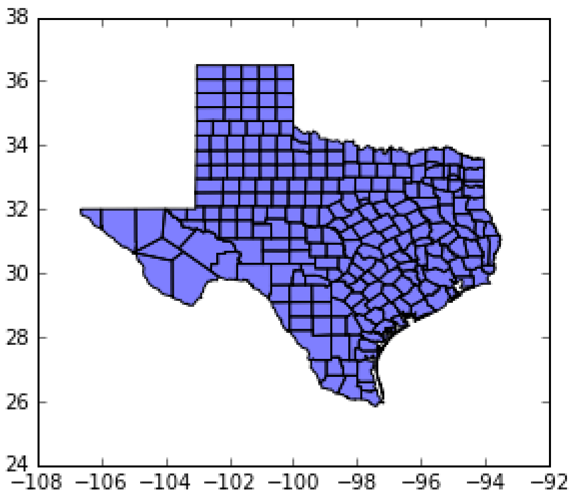
```
hr90fj5.bins # upper bounds of each bin
```

```
array([  3.15613527,   8.84642604,  15.88088069,  27.63957988,  43.51610096])
```

GeoPandas

```
import geopandas as gpd
shp_link = "data/texas.shp"
tx = gpd.read_file(shp_link)
tx.plot(color='blue')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x11ceab6a0>
```

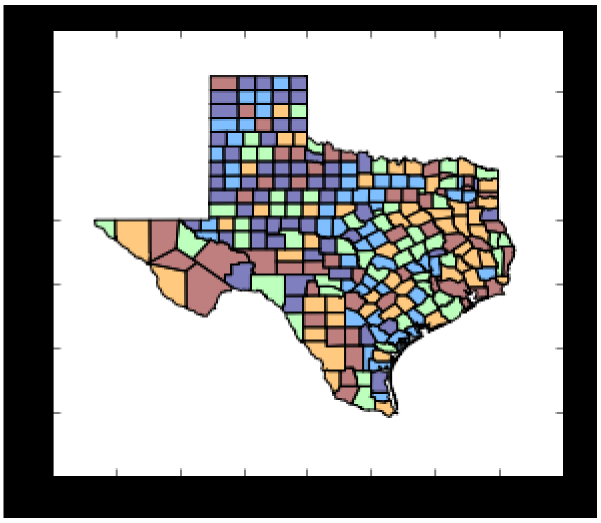


```
type(tx)
```

```
geopandas.geodataframe.GeoDataFrame
```

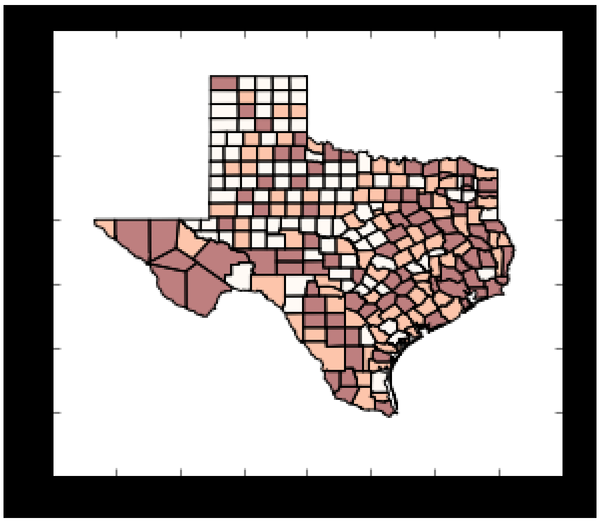
```
tx.plot(column='HR90', scheme='QUANTILES') # uses pysal classifier under the h
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x11dcd85f8>
```



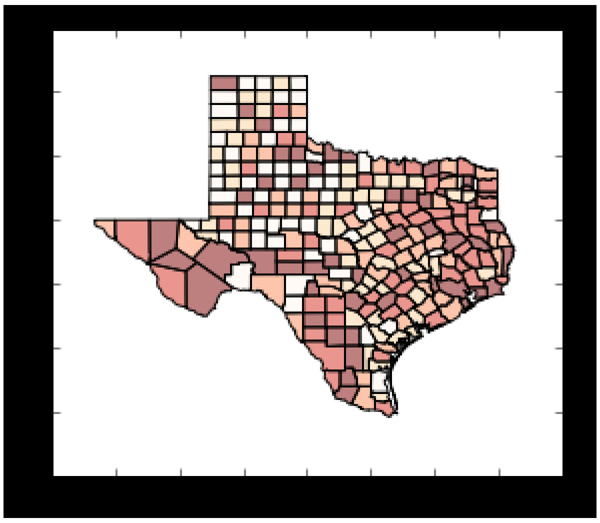
```
tx.plot(column='HR90', scheme='QUANTILES', k=3, cmap='OrRd') # we need a conti
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x11e676c18>
```



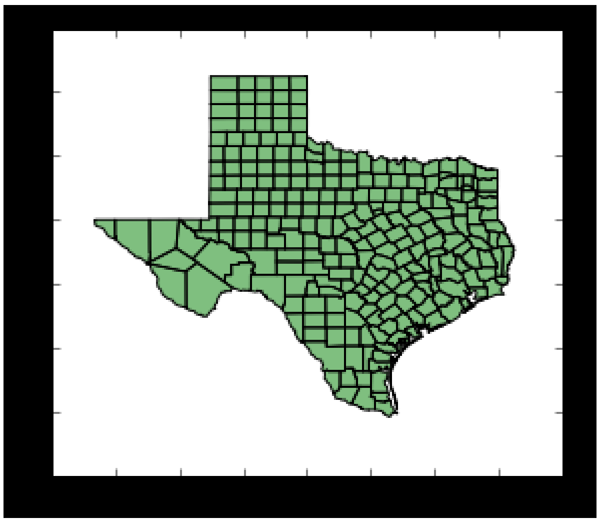
```
tx.plot(column='HR90', scheme='QUANTILES', k=5, cmap='OrRd') # bump up to quin
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x11f24fac8>
```



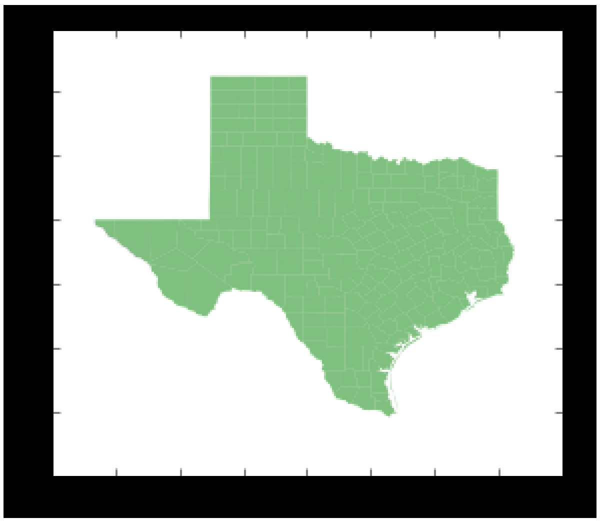
```
tx.plot(color='green') # explore options, polygon fills
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x11fc2d630>
```



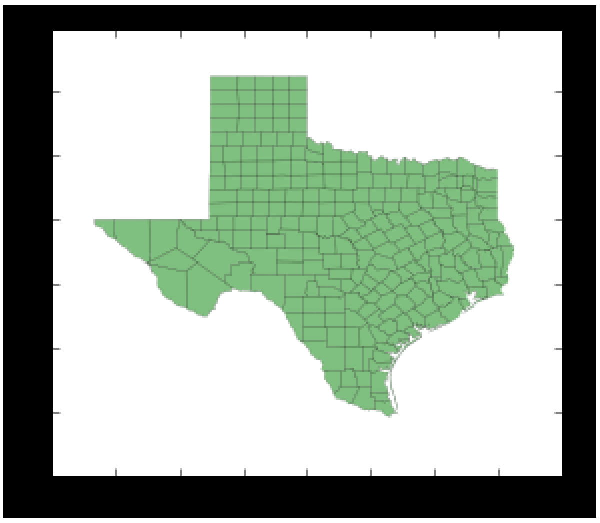
```
tx.plot(color='green',linewidth=0) # border
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x12080a710>
```



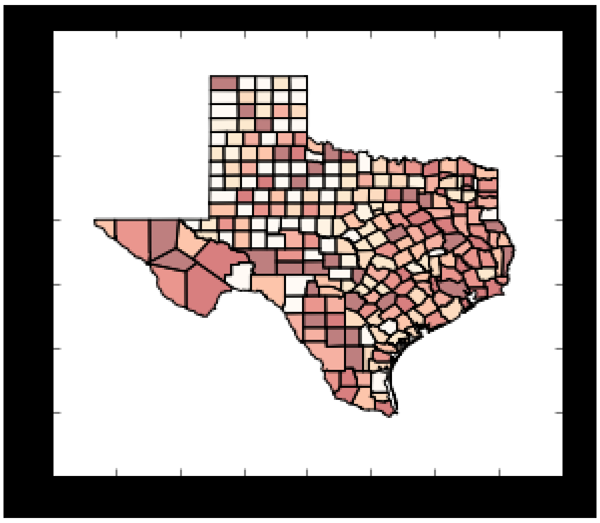
```
tx.plot(color='green',linewidth=0.1) # border
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1210dfe10>
```



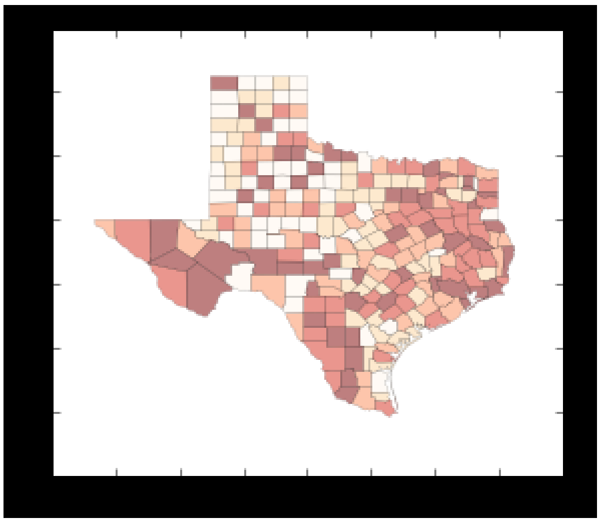
```
tx.plot(column='HR90', scheme='QUANTILES', k=9, cmap='OrRd') # now with qunati
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x121cae6a0>
```

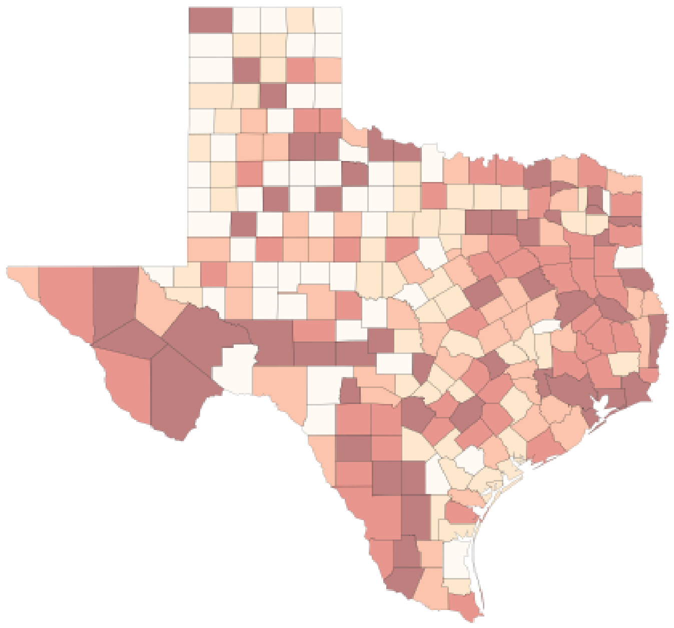


```
tx.plot(column='HR90', scheme='QUANTILES', k=5, cmap='OrRd', linewidth=0.1)
```

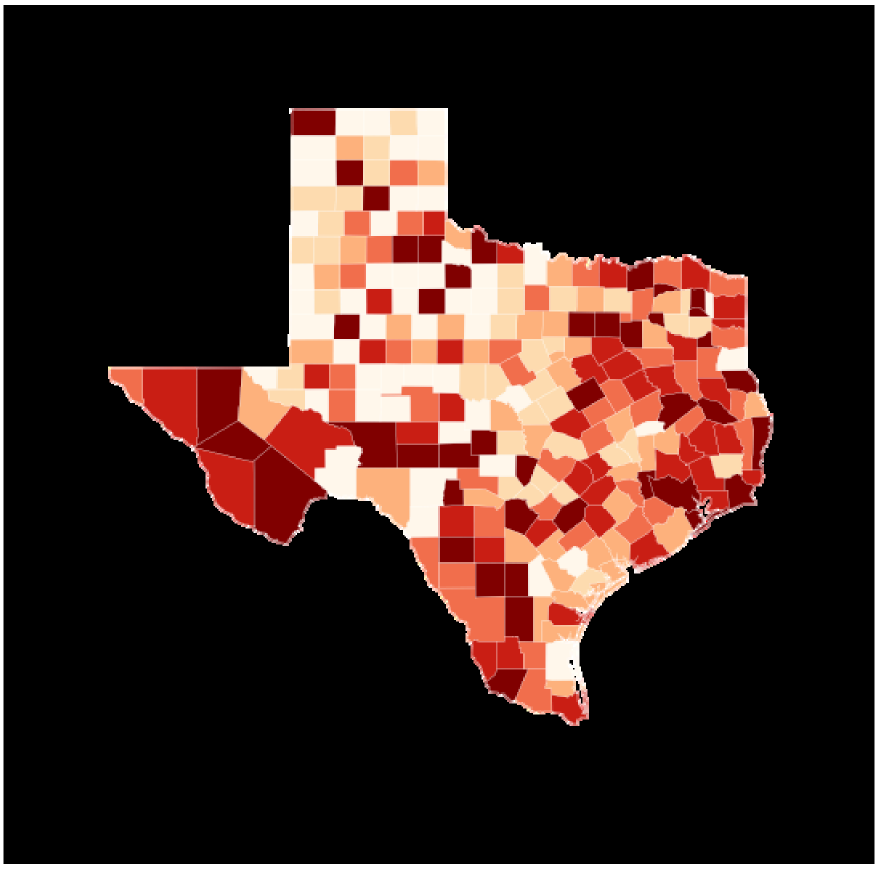
```
<matplotlib.axes._subplots.AxesSubplot at 0x1227929e8>
```



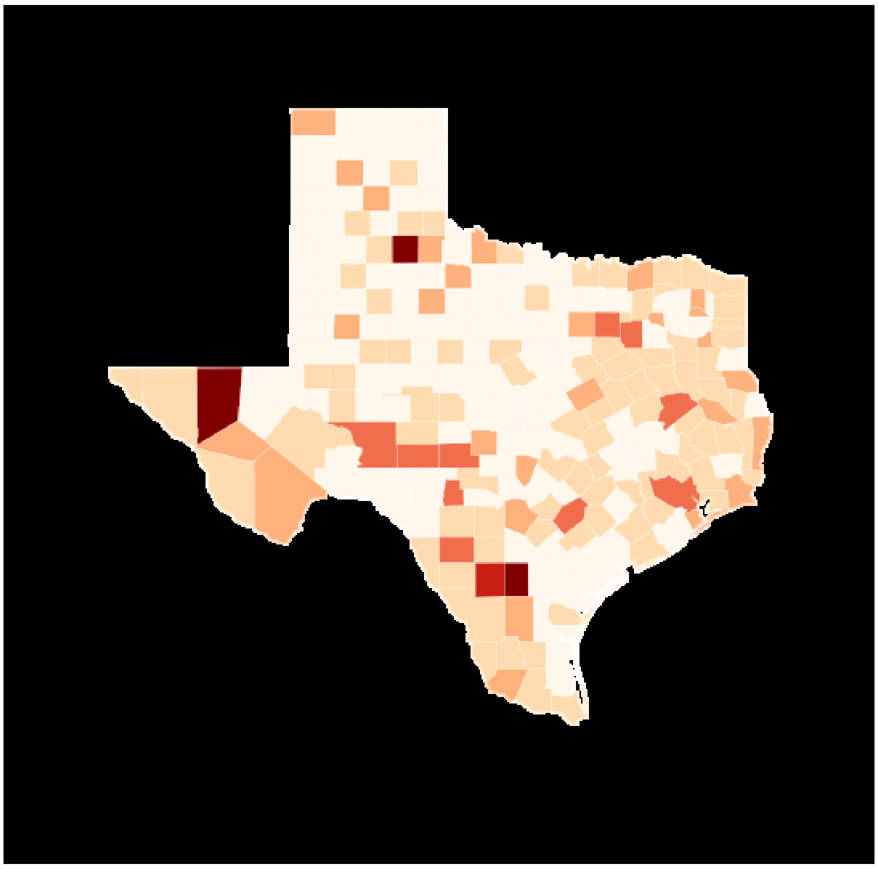
```
import matplotlib.pyplot as plt # make plot larger

f, ax = plt.subplots(1, figsize=(9, 9))
tx.plot(column='HR90', scheme='QUANTILES', k=5, cmap='OrRd', linewidth=0.1, ax
ax.set_axis_off()
plt.show()
```



```
f, ax = plt.subplots(1, figsize=(9, 9))
tx.plot(column='HR90', scheme='QUANTILES', \
        k=6, cmap='OrRd', linewidth=0.1, ax=ax, \
        edgecolor='white')
ax.set_axis_off()
plt.show()
```

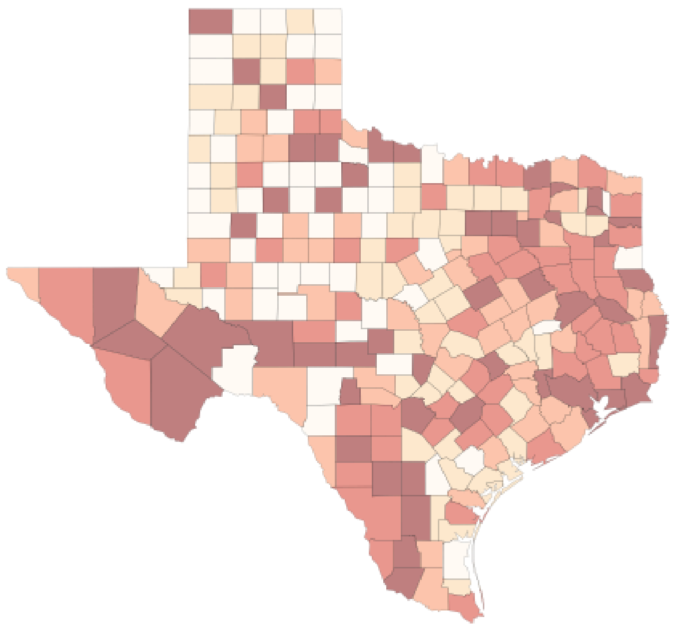


```
f, ax = plt.subplots(1, figsize=(9, 9))
tx.plot(column='HR90', scheme='equal_interval', \
        k=6, cmap='OrRd', linewidth=0.1, ax=ax, \
        edgecolor='white')
ax.set_axis_off()
plt.show()
```



```
# try deciles
f, ax = plt.subplots(1, figsize=(9, 9))

tx.plot(column='HR90', scheme='QUANTILES', k=10, cmap='OrRd', linewidth=0.1, a
ax.set_axis_off()
plt.show()
```

```
/Users/dani/anaconda/envs/gds-scipy16/lib/python3.5/site-packages/geopandas/ge
  return plot_dataframe(self, *args, **kwargs)
```



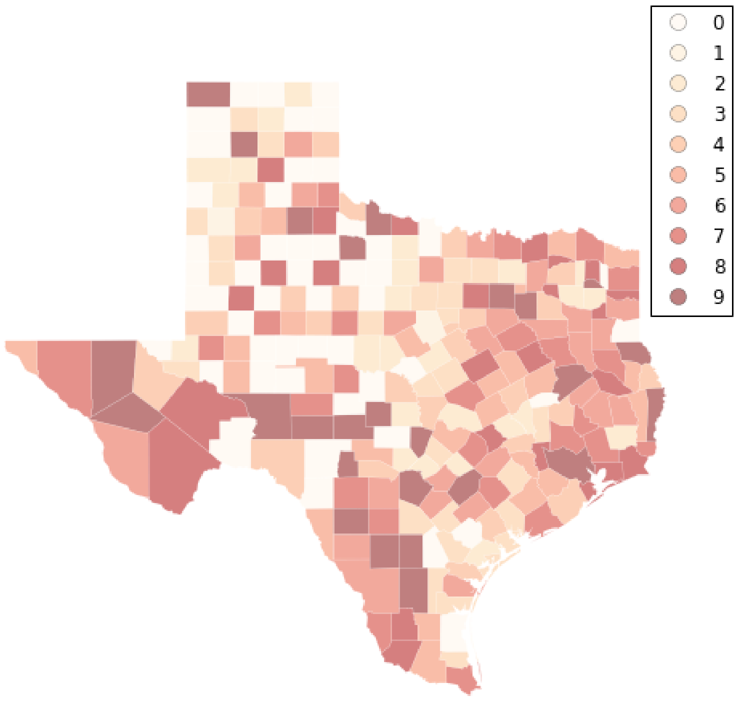
```
# ok, let's work around to get deciles
q10 = ps.Quantiles(tx.HR90,k=10)
q10.bins
```

```
array([  0.        ,   2.42057708,   4.59760916,   5.6524773 ,
         7.23234613,   8.50963716,  10.30447074,  12.57143011,
        16.6916767 ,  43.51610096])
```

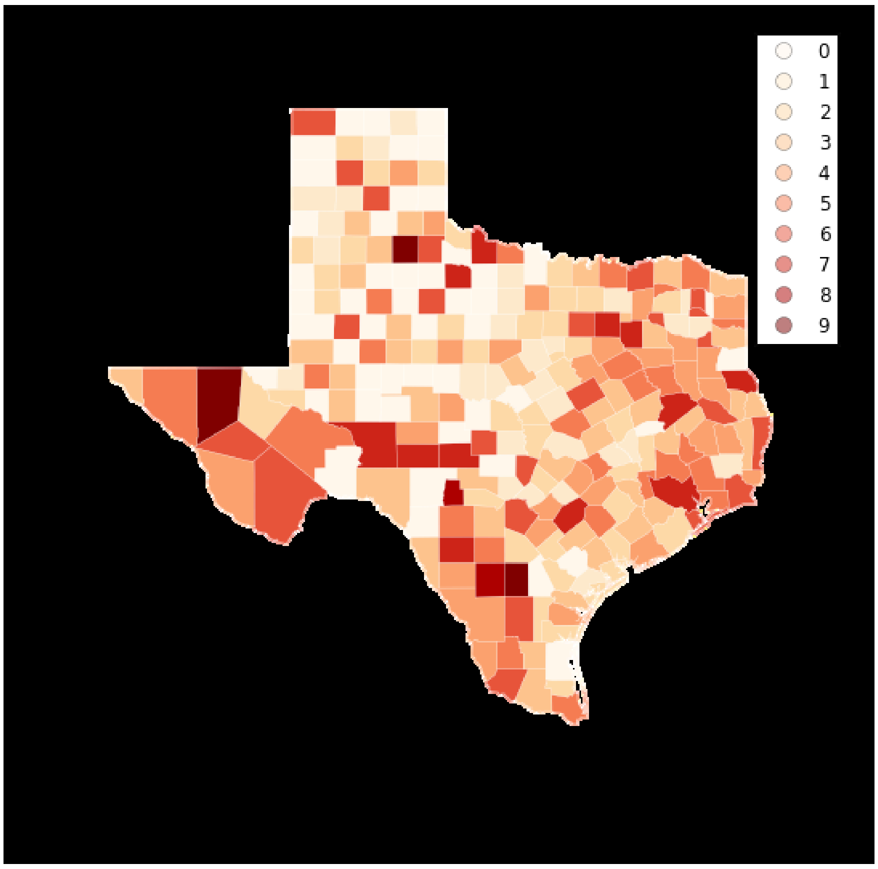
```
q10.yb
```

```
array([0, 0, 9, 0, 2, 0, 0, 2, 0, 3, 9, 3, 6, 4, 0, 2, 8, 0, 0, 2, 0, 2, 5,
       0, 7, 6, 4, 9, 9, 8, 5, 4, 1, 3, 0, 8, 0, 4, 7, 7, 6, 5, 8, 0, 0, 0,
       6, 2, 3, 9, 0, 0, 5, 8, 6, 3, 3, 6, 2, 8, 0, 0, 2, 0, 8, 2, 8, 0, 3,
       0, 4, 0, 7, 9, 2, 3, 3, 8, 9, 5, 8, 0, 4, 0, 4, 0, 8, 2, 0, 2, 8, 9,
       4, 6, 6, 8, 4, 3, 6, 7, 7, 5, 6, 3, 0, 4, 4, 1, 6, 0, 6, 7, 4, 6, 5,
       4, 6, 0, 0, 5, 0, 2, 7, 0, 2, 2, 7, 2, 8, 9, 4, 0, 7, 5, 9, 8, 7, 5,
       0, 3, 5, 3, 5, 0, 5, 0, 5, 4, 9, 7, 0, 8, 5, 0, 4, 3, 6, 8, 4, 7, 9,
       5, 6, 5, 9, 0, 7, 0, 9, 6, 4, 4, 2, 9, 2, 2, 7, 3, 2, 9, 9, 8, 0, 6,
       5, 7, 8, 2, 0, 9, 7, 7, 4, 3, 0, 4, 5, 8, 7, 8, 6, 9, 2, 5, 9, 2, 2,
       3, 4, 8, 6, 5, 9, 9, 6, 7, 5, 7, 0, 4, 8, 6, 6, 3, 3, 7, 3, 4, 9, 7,
       5, 0, 0, 3, 9, 9, 6, 2, 3, 6, 4, 3, 9, 3, 6, 3, 8, 7, 5, 0, 8, 5, 3,
       7])
```

```
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=q10.yb).plot(column='cl', categorical=True, \
        k=10, cmap='OrRd', linewidth=0.1, ax=ax, \
        edgecolor='white', legend=True)
ax.set_axis_off()
plt.show()
```



```
fj10 = ps.Fisher_Jenks(tx.HR90,k=10)
fj10.bins
#labels = ["%0.1f"%l for l in fj10.bins]
#labels
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=fj10.yb).plot(column='cl', categorical=True, \
        k=10, cmap='OrRd', linewidth=0.1, ax=ax, \
        edgecolor='white', legend=True)
ax.set_axis_off()
plt.show()
```



```
fj10.adcm
```

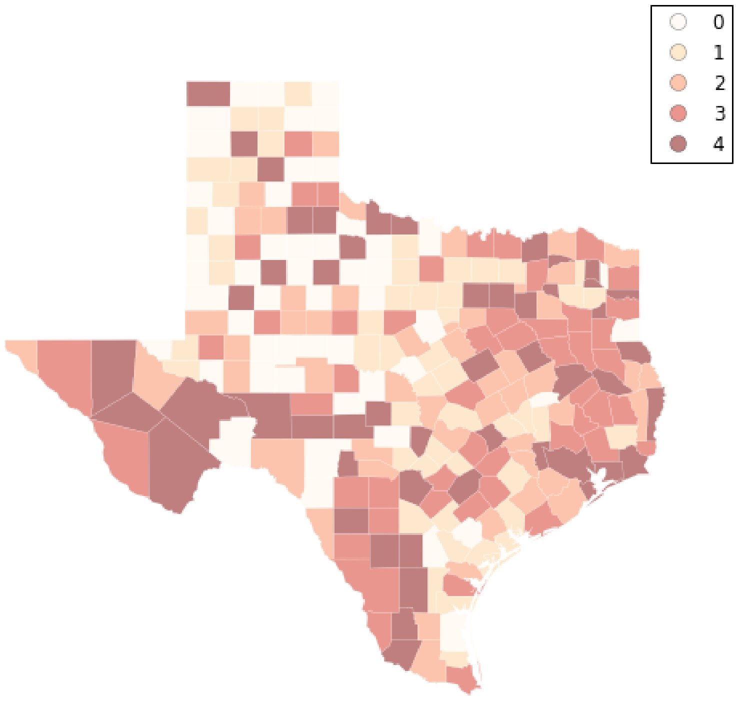
```
133.99950285589998
```

```
q10.adcm
```

```
220.80434598560004
```

```
q5 = ps.Quantiles(tx.HR90,k=5)
```

```
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=q5.yb).plot(column='cl', categorical=True, \
        k=10, cmap='OrRd', linewidth=0.1, ax=ax, \
        edgecolor='white', legend=True)
ax.set_axis_off()
plt.show()
```



Folium

In addition to using matplotlib, the viz module includes components that interface with the folium library which provides a Pythonic way to generate Leaflet maps.

```
import pysal as ps
import geojson as gj
from pysal.contrib.viz import folium_mapping as fm
```

First, we need to convert the data into a JSON format. JSON, short for "Javascript Serialized Object Notation," is a simple and effective way to represent objects in a digital environment. For geographic information, the GeoJSON standard defines how to represent geographic information in JSON format. Python programmers may be more comfortable thinking of JSON data as something akin to a standard Python dictionary.

```
filepath = '../data/texas.shp'[:-4]
shp = ps.open(filepath + '.shp')
dbf = ps.open(filepath + '.dbf')
```

```
js = fm.build_features(shp, dbf)
```

Just to show, this constructs a dictionary with the following keys:

```
js.keys()
```

```
dict_keys(['bbox', 'type', 'features'])
```

```
js.type
```

```
'FeatureCollection'
```

```
js.bbox
```

```
[-106.6495132446289, 25.845197677612305, -93.50721740722656, 36.49387741088867
```

```
js.features[0]
```

```
{"bbox": [-100.5494155883789, 36.05754852294922, -99.99715423583984, 36.493877
```

Then, we write the json to a file:

```
with open('./example.json', 'w') as out:
    gj.dump(js, out)
```

Mapping

Let's look at the columns that we are going to map.

```
list(js.features[0].properties.keys())[:5]
```

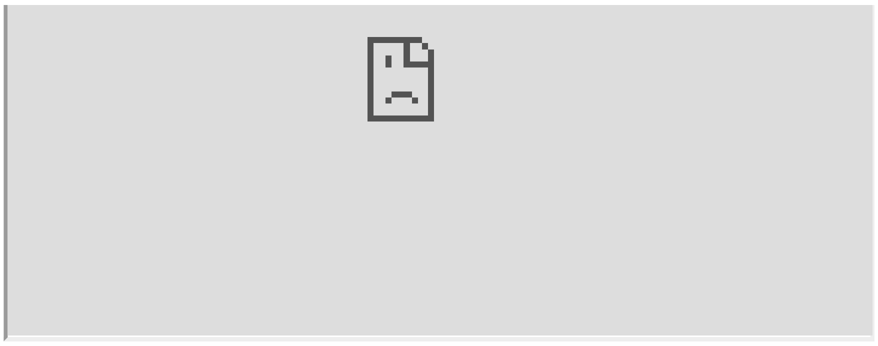
```
['DNL90', 'RD90', 'HR90', 'FH80', 'DNL70']
```

We can map these attributes by calling them as arguments to the choropleth mapping function:

```
fm.choropleth_map?
```

```
# folium maps have been turned off for creating gitbook.
# to run them, uncomment.
fm.choropleth_map('./example.json', 'FIPS', 'HR90',zoom_start=6)
```

```
/Users/dani/anaconda/envs/gds-scipy16/lib/python3.5/site-packages/folium/foliu
  warnings.warn('This method is deprecated. '
```

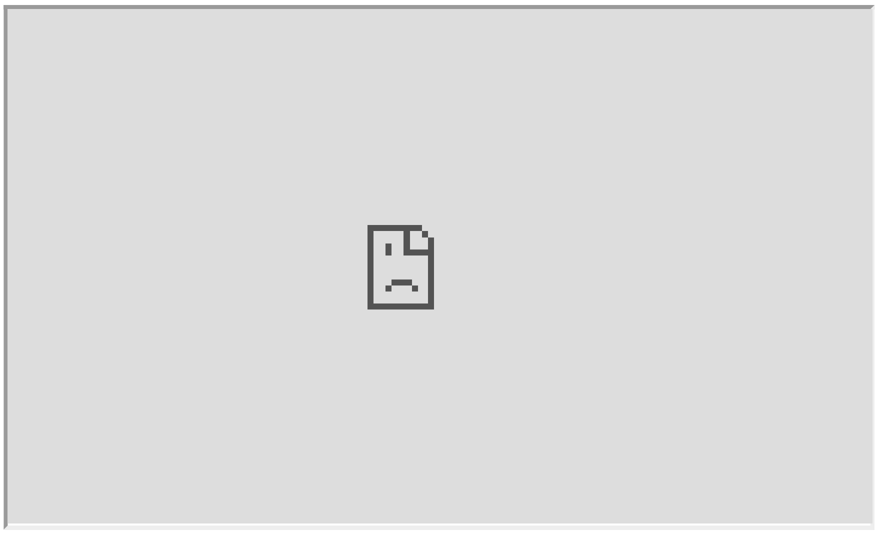


This produces a map using default classifications and color schemes and saves it to an html file. We set the function to have sane defaults. However, if the user wants to have more control, we have many options available.

There are arguments to change the classification scheme:

```
# folium maps have been turned off for creating gitbook.
# to run them, uncomment.
fm.choropleth_map('./example.json', 'FIPS', 'HR90', classification = 'Quantile
```

```
/Users/dani/anaconda/envs/gds-scipy16/lib/python3.5/site-packages/folium/foliu
  warnings.warn('This method is deprecated. '
```

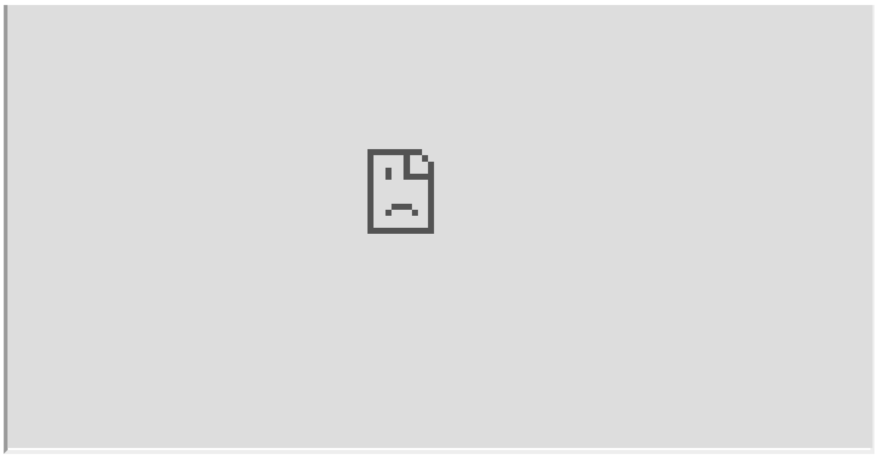


Most PySAL classifiers are supported.

Base Map Type

```
# folium maps have been turned off for creating gitbook.
# to run them, uncomment.
fm.choropleth_map('./example.json', 'FIPS', 'HR90', classification = 'Jenks Ca
                  tiles='Stamen Toner',zoom_start=6, save=True)
```

```
/Users/dani/anaconda/envs/gds-scipy16/lib/python3.5/site-packages/folium/foliu
  warnings.warn('This method is deprecated. '
```

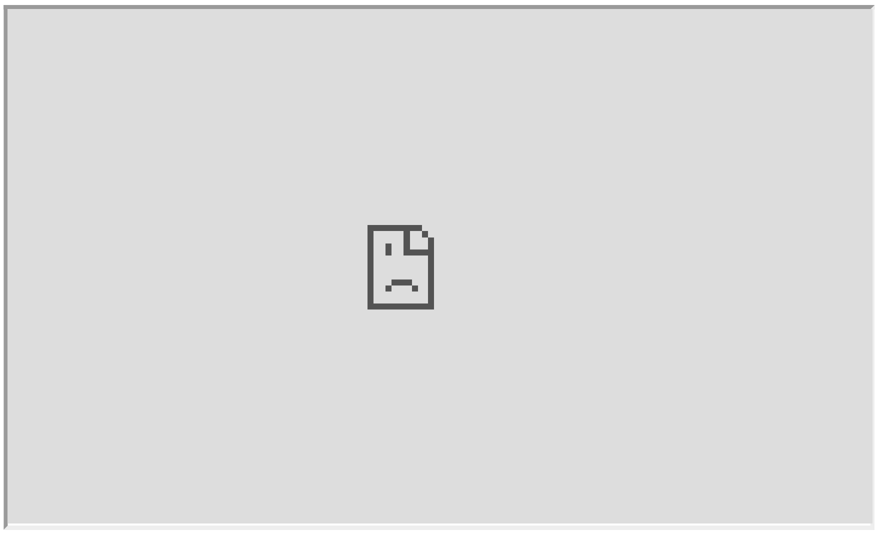


We support the entire range of builtin basemap types in Folium, but custom tilesets from MapBox are not supported (yet).

Color Scheme

```
# folium maps have been turned off for creating gitbook.
# to run them, uncomment.

fm.choropleth_map('./example.json', 'FIPS', 'HR80', classification = 'Jenks Ca
                  tiles='Stamen Toner', fill_color = 'PuBuGn', save=True)
```

```
/Users/dani/anaconda/envs/gds-scipy16/lib/python3.5/site-packages/folium/foliu
  warnings.warn('This method is deprecated. '
```



All color schemes are Color Brewer and simply pass through to `Folium` on execution.

Folium supports up to 6 classes.

Cartopy

Next we turn to cartopy.

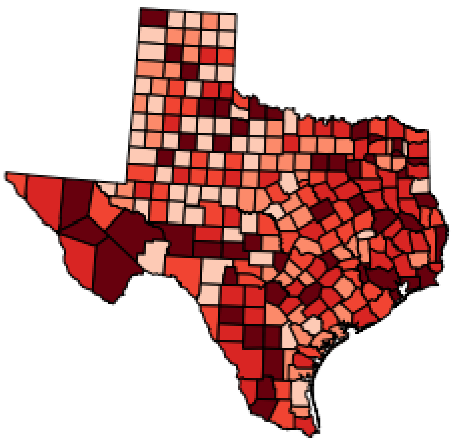
```
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import cartopy.io.shapereader as shpreader

reader = shpreader.Reader("../data/texas.shp")
```

```
def choropleth(classes, colors, reader, legend=None, title=None, fileName=None
    ax = plt.axes([0,0,1,1], projection=ccrs.LambertConformal())
    ax.set_extent([-108, -93, 38, 24], ccrs.Geodetic())
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)
    if title:
        plt.title(title)
    ax.set_extent([-108, -93, 38, 24], ccrs.Geodetic())
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)

    for i,state in enumerate(reader.geometries()):
        facecolor = colors[classes[i]]
        #facecolor = 'red'
        edgecolor = 'black'
        ax.add_geometries([state], ccrs.PlateCarree(),
                          facecolor=facecolor, edgecolor=edgecolor)

    leg = [ mpatches.Rectangle((0,0),1,1, facecolor=color) for color in colors
    if legend:
        plt.legend(leg, legend, loc='lower left', bbox_to_anchor=(0.025, -0.1)
    if fileName:
        plt.savefig(fileName, dpi=dpi)
    plt.show()
```

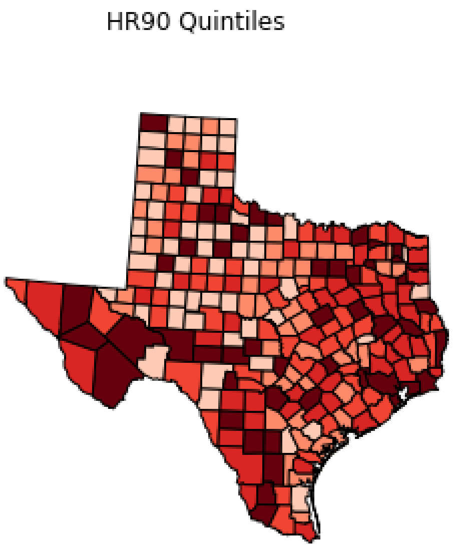
```
HR90 = values
```

```
bins_q5 = ps.Quantiles(HR90, k=5)
```

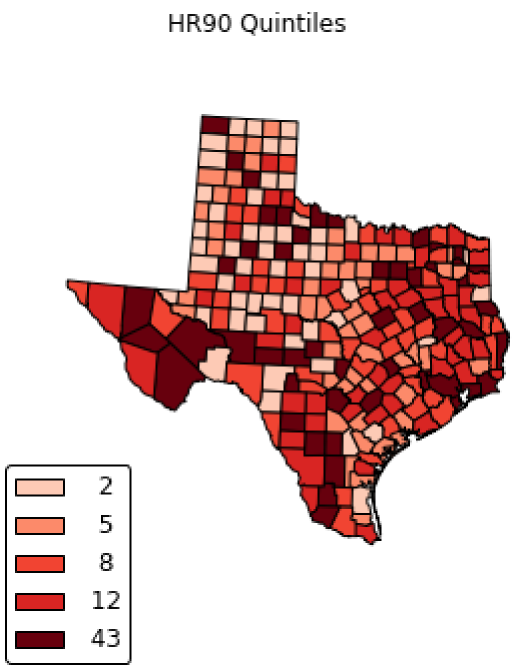
```
bwr = plt.cm.get_cmap('Reds')
bwr(.76)
c5 = [bwr(c) for c in [0.2, 0.4, 0.6, 0.7, 1.0]]
classes = bins_q5.yb
choropleth(classes, c5, reader)
```



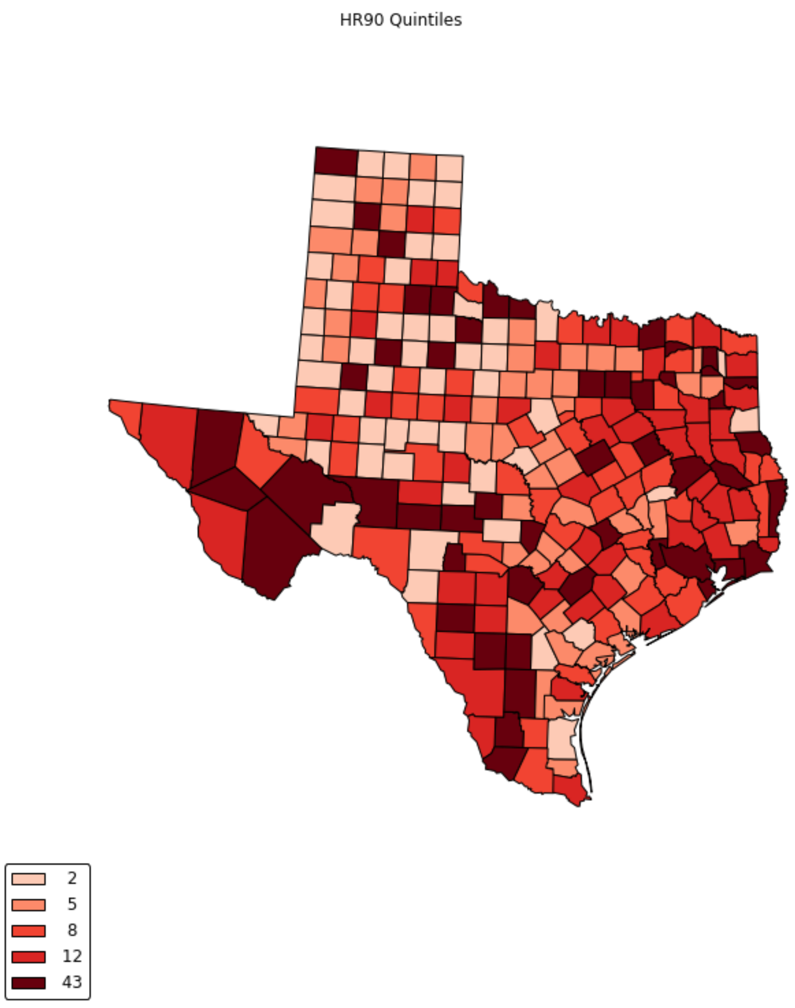
```
choropleth(classes, c5, reader, title="HR90 Quintiles")
```



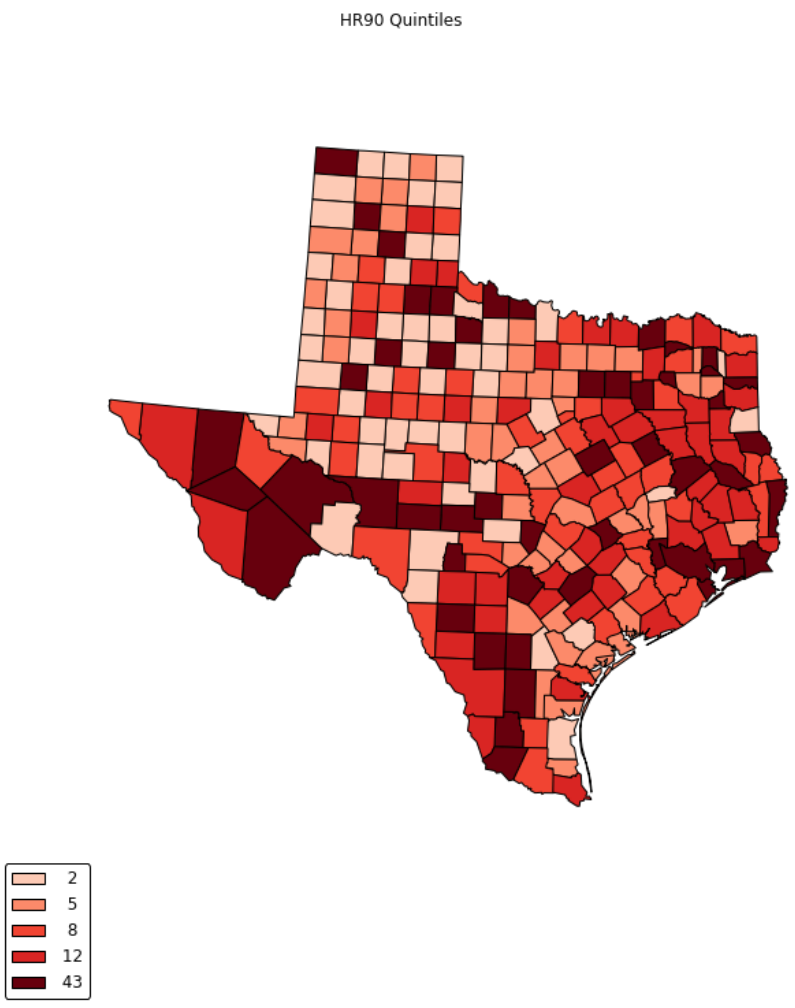
```
legend =[ "%3d"%ub for ub in bins_q5.bins]
choropleth(classes, c5, reader, legend, title="HR90 Quintiles")
```



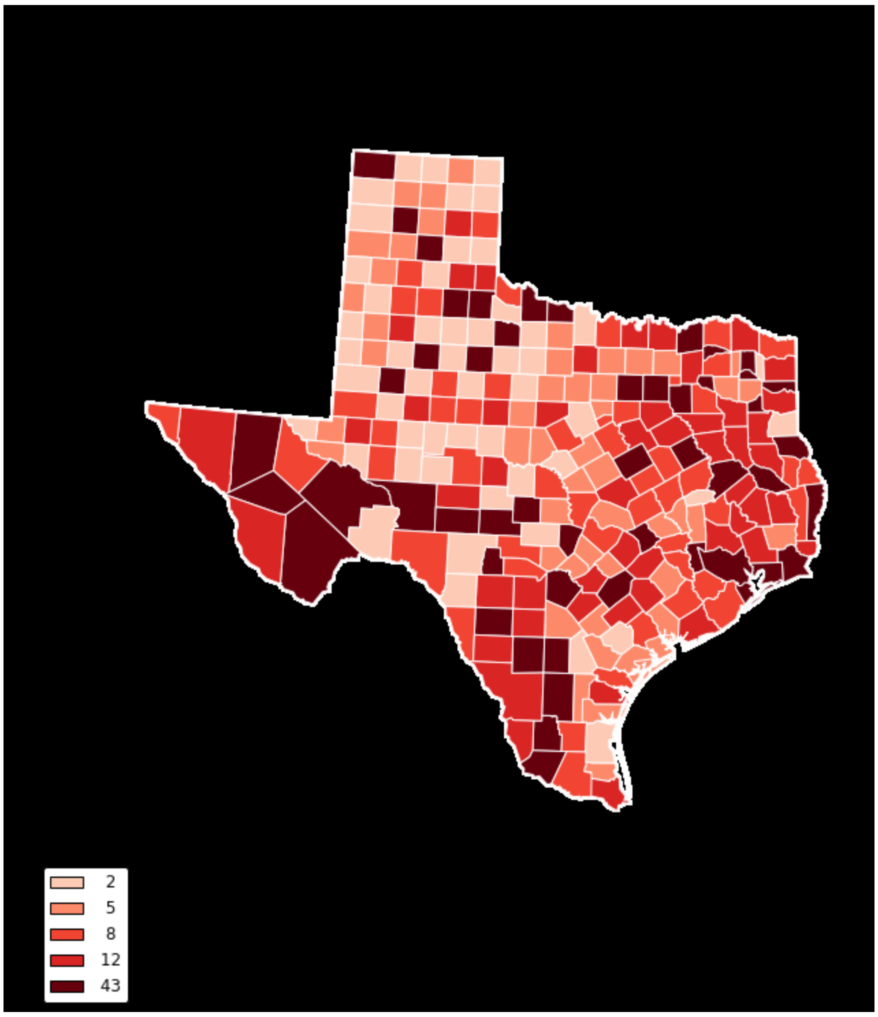
```
def choropleth(classes, colors, reader, legend=None, title=None, fileName=None
    f, ax = plt.subplots(1, figsize=(9,9))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.axison=False

    ax = plt.axes([0,0,1,1], projection=ccrs.LambertConformal())
    ax.set_extent([-108, -93, 38, 24], ccrs.Geodetic())
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)
    if title:
        plt.title(title)
    ax.set_extent([-108, -93, 38, 24], ccrs.Geodetic())
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)

    for i,state in enumerate(reader.geometries()):
        facecolor = colors[classes[i]]
        #facecolor = 'red'
        edgecolor = 'black'
        ax.add_geometries([state], ccrs.PlateCarree(),
                          facecolor=facecolor, edgecolor=edgecolor)


    leg = [ mpatches.Rectangle((0,0),1,1, facecolor=color) for color in colors
    if legend:
        plt.legend(leg, legend, loc='lower left', bbox_to_anchor=(0.025, -0.1)
    if fileName:
        plt.savefig(fileName, dpi=dpi)
    #ax.set_axis_off()
    plt.show()

legend =[ "%3d"%ub for ub in bins_q5.bins]
choropleth(classes, c5, reader, legend, title="HR90 Quintiles")
```



```
legend =[ "%3d"%ub for ub in bins_q5.bins]
choropleth(classes, c5, reader, legend, title="HR90 Quintiles")
```


HR90 Quintiles
2
5
8
12
43

```
def choropleth(classes, colors, reader, legend=None, title=None, fileName=None
    f, ax = plt.subplots(1, figsize=(9,9), frameon=False)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.axison=False


    ax = plt.axes([0,0,1,1], projection=ccrs.LambertConformal())
    ax.set_extent([-108, -93, 38, 24], ccrs.Geodetic())
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)

    if title:
        plt.title(title)

    for i,state in enumerate(reader.geometries()):
        facecolor = colors[classes[i]]
        edgecolor = 'white'
        ax.add_geometries([state], ccrs.PlateCarree(),
                          facecolor=facecolor, edgecolor=edgecolor)


    leg = [ mpatches.Rectangle((0,0),1,1, facecolor=color) for color in colors
    if legend:
        plt.legend(leg, legend, loc='lower left', bbox_to_anchor=(0.025, -0.1)
    if fileName:
        plt.savefig(fileName, dpi=dpi)
    plt.show()

legend =[ "%3d"%ub for ub in bins_q5.bins]
choropleth(classes, c5, reader, legend, title="HR90 Quintiles")
```



For an example publication and code where Cartopy was used for the mapping see: Rey (2016).

Bokeh

website

```
from collections import OrderedDict

#from bokeh.sampledata import us_counties, unemployment
from bokeh.plotting import figure, show, output_notebook, ColumnDataSource
from bokeh.models import HoverTool

from bokeh.charts import Scatter, output_file, show
```

```
def gpd_bokeh(df):
    """Convert geometries from geopandas to bokeh format"""
    nan = float('nan')
    lons = []
    lats = []
    for i,shape in enumerate(df.geometry.values):
        if shape.geom_type == 'MultiPolygon':
            gx = []
            gy = []
            ng = len(shape.geoms) - 1
            for j,member in enumerate(shape.geoms):
                xy = np.array(list(member.exterior.coords))
                xs = xy[:,0].tolist()
                ys = xy[:,1].tolist()
                gx.extend(xs)
                gy.extend(ys)
                if j < ng:
                    gx.append(nan)
                    gy.append(nan)
            lons.append(gx)
            lats.append(gy)

        else:
            xy = np.array(list(shape.exterior.coords))
            xs = xy[:,0].tolist()
            ys = xy[:,1].tolist()
            lons.append(xs)
            lats.append(ys)

    return lons,lats
```

```
lons, lats = gpd_bokeh(tx)
```

```
p = figure(title="Texas", toolbar_location='left',
          plot_width=1100, plot_height=700)
p.patches(lons, lats, fill_alpha=0.7, #fill_color=state_colors,
         line_color="#884444", line_width=2, line_alpha=0.3)
output_file('choropleth.html', title="choropleth.py example")
show(p)
```

```
bwr = plt.cm.get_cmap('Reds')
bwr(.76)
c5 = [bwr(c) for c in [0.2, 0.4, 0.6, 0.7, 1.0]]
classes = bins_q5.yb
colors = [c5[i] for i in classes]
```

```
colors5 = ["#F1EEF6", "#D4B9DA", "#C994C7", "#DF65B0", "#DD1C77"]
colors = [colors5[i] for i in classes]

p = figure(title="Texas HR90 Quintiles", toolbar_location='left',
          plot_width=1100, plot_height=700)
p.patches(lons, lats, fill_alpha=0.7, fill_color=colors,
         line_color="#884444", line_width=2, line_alpha=0.3)
output_file('choropleth.html', title="choropleth.py example")
show(p)
```

Hover

```
from bokeh.models import HoverTool
from bokeh.plotting import figure, show, output_file, ColumnDataSource
```

```
source = ColumnDataSource(data=dict(
        x=lons,
        y=lats,
        color=colors,
        name=tx.NAME,
        rate=HR90
    ))

TOOLS = "pan, wheel_zoom, box_zoom, reset, hover, save"
p = figure(title="Texas Homicide 1990 (Quintiles)", tools=TOOLS,
          plot_width=900, plot_height=900)

p.patches('x', 'y', source=source,
         fill_color='color', fill_alpha=0.7,
         line_color='white', line_width=0.5)

hover = p.select_one(HoverTool)
hover.point_policy = 'follow_mouse'
hover.tooltips = [
    ("Name", "@name"),
    ("Homicide rate", "@rate"),
    ("(Long, Lat)", "($x, $y)"),
]


output_file("hr90.html", title="hr90.py example")
show(p)
```

Exercises

1. Using Bokeh, use PySALs Fisher Jenks classifier with k=10 to generate a choropleth map of the homicide rates in 1990 for Texas counties. Modify the hover tooltips so that in addition to showing the Homicide rate, the rank of that rate is also shown.
2. Explore `ps.esda.mapclassify.` (hint: use tab completion) to select a new classifier (different from the ones in this notebook). Using the same data as in exercise 1, apply this classifier and create a choropleth using Bokeh.

Spatial Weights

> `IPYNB`

Spatial weights are mathematical structures used to represent spatial relationships. Many spatial analytics, such as spatial autocorrelation statistics and regionalization algorithms rely on spatial weights. Generally speaking, a spatial weight $w_{i,j}$ expresses the notion of a geographical relationship between locations $i$ and $j$. These relationships can be based on a number of criteria including contiguity, geospatial distance and general distances.

PySAL offers functionality for the construction, manipulation, analysis, and conversion of a wide array of spatial weights.

We begin with construction of weights from common spatial data formats.

```
import pysal as ps
import numpy as np
```

There are functions to construct weights directly from a file path.

```
shp_path = "../data/texas.shp"
```

Weight Types

Contiguity:

Queen Weights

A commonly-used type of weight is a queen contigutiy weight, which reflects adjacency relationships as a binary indicator variable denoting whether or not a polygon shares an edge or a vertex with another polygon. These weights are symmetric, in that when polygon $A$ neighbors polygon $B$, both $w*{AB} = 1$ and $w*{BA} = 1$.

To construct queen weights from a shapefile, use the `queen_from_shapefile` function:

```
qW = ps.queen_from_shapefile(shp_path)
dataframe = ps.pdio.read_files(shp_path)
```

```
qW
```

```
<pysal.weights.weights.W at 0x104142860>
```

All weights objects have a few traits that you can use to work with the weights object, as well as to get information about the weights object.

To get the neighbors & weights around an observation, use the observation's index on the weights object, like a dictionary:

```
qW[4] #neighbors & weights of the 5th observation (0-index remember)
```

```
{0: 1.0, 3: 1.0, 5: 1.0, 6: 1.0, 7: 1.0}
```

By default, the weights and the pandas dataframe will use the same index. So, we can view the observation and its neighbors in the dataframe by putting the observation's index and its neighbors' indexes together in one list:

```
self_and_neighbors = [4]
self_and_neighbors.extend(qW.neighbors[4])
print(self_and_neighbors)
```

```
[4, 0, 3, 5, 6, 7]
```

and grabbing those elements from the dataframe:

```
dataframe.loc[self_and_neighbors]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	I
4	Ochiltree	Texas	48	357	4
0	Lipscomb	Texas	48	295	4
3	Hansford	Texas	48	195	4
5	Roberts	Texas	48	393	4
6	Hemphill	Texas	48	211	4
7	Hutchinson	Texas	48	233	4

6 rows × 70 columns

A full, dense matrix describing all of the pairwise relationships is constructed using the `.full` method, or when `pysal.full` is called on a weights object:

```
Wmatrix, ids = qW.full()
#Wmatrix, ids = ps.full(qW)
```

```
Wmatrix
```

```
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  1., ...,  0.,  0.,  0.],
       [ 0.,  1.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  1.,  1.],
       [ 0.,  0.,  0., ...,  1.,  0.,  1.],
       [ 0.,  0.,  0., ...,  1.,  1.,  0.]])
```

```
n_neighbors = Wmatrix.sum(axis=1) # how many neighbors each region has
```

```
n_neighbors[4]
```

```
5.0
```

```
qW.cardinalities[4]
```

```
5
```

Note that this matrix is binary, in that its elements are either zero or one, since an observation is either a neighbor or it is not a neighbor.

However, many common use cases of spatial weights require that the matrix is row-standardized. This is done simply in PySAL using the `.transform` attribute

```
qW.transform = 'r'
```

Now, if we build a new full matrix, its rows should sum to one:

```
Wmatrix, ids = qW.full()
```

```
Wmatrix.sum(axis=1) #numpy axes are 0:column, 1:row, 2:facet, into higher dime
```

```
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Since weight matrices are typically very sparse, there is also a sparse weights matrix constructor:

```
qW.sparse
```

```
<254x254 sparse matrix of type '<class 'numpy.float64'>'
    with 1460 stored elements in Compressed Sparse Row format>
```

```
qW.pct_nonzero #Percentage of nonzero neighbor counts
```

```
2.263004526009052
```

By default, PySAL assigns each observation an index according to the order in which the observation was read in. This means that, by default, all of the observations in the weights object are indexed by table order. If you have an alternative ID variable, you can pass that into the weights constructor.

For example, the `texas.shp` dataset has a possible alternative ID Variable, a `FIPS` code.

```
dataframe.head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FI
0	Lipscomb	Texas	48	295	482
1	Sherman	Texas	48	421	484
2	Dallam	Texas	48	111	481
3	Hansford	Texas	48	195	481
4	Ochiltree	Texas	48	357	483

5 rows × 70 columns

The observation we were discussing above is in the fifth row: Ochiltree county, Texas. Note that its FIPS code is 48357.

Then, instead of indexing the weights and the dataframe just based on read-order, use the `FIPS` code as an index:

```
qW = ps.queen_from_shapefile(shp_path, idVariable='FIPS')
```

```
qW[4] #fails, since no FIPS is 4.
```

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-21-1d8a3009bc1e> in <module>()
----> 1 qW[4] #fails, since no FIPS is 4.


/Users/dani/anaconda/envs/gds-scipy16/lib/python3.5/site-packages/pysal/weight
    504         {1: 1.0, 4: 1.0, 101: 1.0, 85: 1.0, 5: 1.0}
    505         """
--> 506         return dict(list(zip(self.neighbors[key], self.weights[key])))
    507 
    508     def __iter__(self):


KeyError: 4
```

Note that a `KeyError` in Python usually means that some index, here `4`, was not found in the collection being searched, the IDs in the queen weights object. This makes sense, since we explicitly passed an `idVariable` argument, and nothing has a `FIPS` code of 4.

Instead, if we use the observation's `FIPS` code:

```
qW['48357']
```

```
{'48195': 1.0, '48211': 1.0, '48233': 1.0, '48295': 1.0, '48393': 1.0}
```

We get what we need.

In addition, we have to now query the dataframe using the `FIPS` code to find our neighbors. But, this is relatively easy to do, since pandas will parse the query by looking into python objects, if told to.

First, let us store the neighbors of our target county:

```
self_and_neighbors = ['48357']
self_and_neighbors.extend(qW.neighbors['48357'])
```

Then, we can use this list in `.query` :

```
dataframe.query('FIPS in @self_and_neighbors')
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	I
0	Lipscomb	Texas	48	295	4
3	Hansford	Texas	48	195	4
4	Ochiltree	Texas	48	357	4
5	Roberts	Texas	48	393	4
6	Hemphill	Texas	48	211	4
7	Hutchinson	Texas	48	233	4

6 rows × 70 columns

Note that we have to use `@` before the name in order to show that we're referring to a python object and not a column in the dataframe.

```
#dataframe.query('FIPS in self_and_neighbors') will fail because there is no c
```

Of course, we could also reindex the dataframe to use the same index as our weights:

```
fips_frame = dataframe.set_index(dataframe.FIPS)
fips_frame.head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
FIPS				
48295	Lipscomb	Texas	48	295
48421	Sherman	Texas	48	421
48111	Dallam	Texas	48	111
48195	Hansford	Texas	48	195
48357	Ochiltree	Texas	48	357

5 rows × 70 columns

Now that both are using the same weights, we can use the `.loc` indexer again:

```
fips_frame.loc[self_and_neighbors]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
FIPS				
48357	Ochiltree	Texas	48	357
48295	Lipscomb	Texas	48	295
48195	Hansford	Texas	48	195
48393	Roberts	Texas	48	393
48211	Hemphill	Texas	48	211
48233	Hutchinson	Texas	48	233

6 rows × 70 columns

Rook Weights

Rook weights are another type of contiguity weight, but consider observations as neighboring only when they share an edge. The rook neighbors of an observation may be different than its queen neighbors, depending on how the observation and its nearby polygons are configured.

We can construct this in the same way as the queen weights, using the special `rook_from_shapefile` function:

```
rW = ps.rook_from_shapefile(shp_path, idVariable='FIPS')
```

```
rW['48357']
```

```
{'48195': 1.0, '48295': 1.0, '48393': 1.0}
```

These weights function exactly like the Queen weights, and are only distinguished by what they consider "neighbors."

```
self_and_neighbors = ['48357']
self_and_neighbors.extend(rW.neighbors['48357'])
fips_frame.loc[self_and_neighbors]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
FIPS				
48357	Ochiltree	Texas	48	357
48295	Lipscomb	Texas	48	295
48195	Hansford	Texas	48	195
48393	Roberts	Texas	48	393

4 rows × 70 columns

Bishop Weights

In theory, a "Bishop" weighting scheme is one that arises when only polygons that share vertexes are considered to be neighboring. But, since Queen contiguigy requires either an edge or a vertex and Rook contiguity requires only shared edges, the following relationship is true:

$\mathcal{Q} = \mathcal{R} \cup \mathcal{B}$

where $\mathcal{Q}$ is the set of neighbor pairs *via* queen contiguity, $\mathcal{R}$ is the set of neighbor pairs *via* Rook contiguity, and $\mathcal{B}$ *via* Bishop contiguity. Thus:

$\mathcal{Q} \setminus \mathcal{R} = \mathcal{B}$

Bishop weights entail all Queen neighbor pairs that are not also Rook neighbors.

PySAL does not have a dedicated bishop weights constructor, but you can construct very easily using the `w_difference` function. This function is one of a family of tools to work with weights, all defined in `ps.weights`, that conduct these types of set operations between weight objects.

```
bW = ps.w_difference(qW, rW, constrained=False, silent_island_warning=True) #s
```

```
bW.histogram
```

```
[(0, 161), (1, 48), (2, 33), (3, 8), (4, 4)]
```

Thus, the vast majority of counties have no bishop neighbors. But, a few do. A simple way to see these observations in the dataframe is to find all elements of the dataframe that are not "islands," the term for an observation with no neighbors:

```
islands = bW.islands
```

```
# Using `.head()` to limit the number of rows printed
dataframe.query('FIPS not in @islands').head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FI
0	Lipscomb	Texas	48	295	48
1	Sherman	Texas	48	421	48
2	Dallam	Texas	48	111	48
3	Hansford	Texas	48	195	48
4	Ochiltree	Texas	48	357	48

5 rows × 70 columns

Distance

There are many other kinds of weighting functions in PySAL. Another separate type use a continuous measure of distance to define neighborhoods.

```
radius = ps.cg.sphere.RADIUS_EARTH_MILES
radius
```

```
3958.755865744055
```

```
#ps.min_threshold_dist_from_shapefile?
```

```
threshold = ps.min_threshold_dist_from_shapefile('../data/texas.shp',radius) #
```

```
threshold
```

```
60.47758554135752
```

knn defined weights

```
knn4_bad = ps.knnW_from_shapefile('../data/texas.shp', k=4) # ignore curvature
```

```
knn4_bad.histogram
```

```
[(4, 254)]
```

```
knn4 = ps.knnW_from_shapefile('../data/texas.shp', k=4, radius=radius)
```

```
knn4.histogram
```

```
[(4, 254)]
```

```
knn4[0]
```

```
{3: 1.0, 4: 1.0, 5: 1.0, 6: 1.0}
```

```
knn4_bad[0]
```

```
{4: 1.0, 5: 1.0, 6: 1.0, 13: 1.0}
```

Kernel W

Kernel Weights are continuous distance-based weights that use kernel densities to define the neighbor relationship. Typically, they estimate a `bandwidth`, which is a parameter governing how far out observations should be considered neighboring. Then, using this bandwidth, they evaluate a continuous kernel function to provide a weight between 0 and 1.

Many different choices of kernel functions are supported, and bandwidths can either be fixed (constant over all units) or adaptive in function of unit density.

For example, if we want to use adaptive bandwidths for the map and weight according to a gaussian kernel:

```
kernelWa = ps.adaptive_kernelW_from_shapefile('../data/texas.shp', radius=radi
kernelWa
```

```
<pysal.weights.Distance.Kernel at 0x7f8fe4cfe080>
```

```
dataframe.loc[kernelWa.neighbors[4] + [4]]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FI
4	Ochiltree	Texas	48	357	483
5	Roberts	Texas	48	393	483
3	Hansford	Texas	48	195	481
4	Ochiltree	Texas	48	357	483

4 rows × 70 columns

```
kernelWa.bandwidth[0:7]
```

```
array([[ 30.30546757],
       [ 30.05684855],
       [ 39.14876899],
       [ 29.96302462],
       [ 29.96302462],
       [ 30.21084447],
       [ 30.23619029]])
```

```
kernelWa[4]
```

```
{3: 9.99999900663795e-08, 4: 1.0, 5: 0.002299013803371608}
```

```
kernelWa[2]
```

```
{1: 9.99999900663795e-08, 2: 1.0, 8: 0.23409571720488287}
```

Distance Thresholds

```
#ps.min_threshold_dist_from_shapefile?
```

```
# find the largest nearest neighbor distance between centroids
threshold = ps.min_threshold_dist_from_shapefile('../data/texas.shp', radius=r
Wmind0 = ps.threshold_binaryW_from_shapefile('../data/texas.shp', radius=radiu
```

```
WARNING: there are 2 disconnected observations
Island ids:  [133, 181]
```

```
Wmind0.histogram
```

```
[(0, 2),
 (1, 3),
 (2, 5),
 (3, 4),
 (4, 10),
 (5, 26),
 (6, 16),
 (7, 31),
 (8, 70),
 (9, 32),
 (10, 29),
 (11, 12),
 (12, 5),
 (13, 2),
 (14, 5),
 (15, 2)]
```

```
Wmind = ps.threshold_binaryW_from_shapefile('../data/texas.shp', radius=radius
```

```
Wmind.histogram
```

```
[(1, 2),
 (2, 3),
 (3, 4),
 (4, 8),
 (5, 5),
 (6, 20),
 (7, 26),
 (8, 9),
 (9, 32),
 (10, 31),
 (11, 37),
 (12, 33),
 (13, 23),
 (14, 6),
 (15, 7),
 (16, 2),
 (17, 4),
 (18, 2)]
```

```
centroids = np.array([list(poly.centroid) for poly in dataframe.geometry])
```

```
centroids[0:10]
```

```
array([[-100.27156111,   36.27508641],
       [-101.8930971 ,   36.27325425],
       [-102.59590795,   36.27354996],
       [-101.35351324,   36.27230422],
       [-100.81561379,   36.27317803],
       [-100.81482387,   35.8405153 ],
       [-100.2694824 ,   35.83996075],
       [-101.35420366,   35.8408377 ],
       [-102.59375964,   35.83958662],
       [-101.89248229,   35.84058246]])
```

```
Wmind[0]
```

```
{3: 1, 4: 1, 5: 1, 6: 1, 13: 1}
```

```
knn4[0]
```

```
{3: 1.0, 4: 1.0, 5: 1.0, 6: 1.0}
```

Visualization

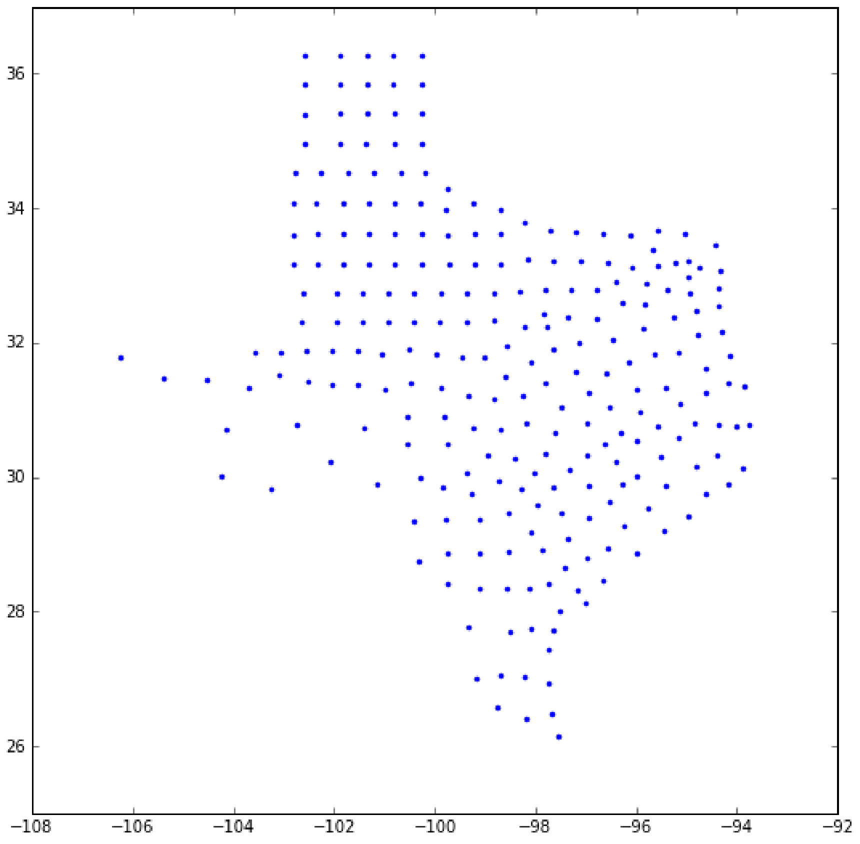
```
%matplotlib inline
import matplotlib.pyplot as plt
from pylab import figure, scatter, show
```

```
wq = ps.queen_from_shapefile('../data/texas.shp')
```

```
wq[0]
```

```
{4: 1.0, 5: 1.0, 6: 1.0}
```

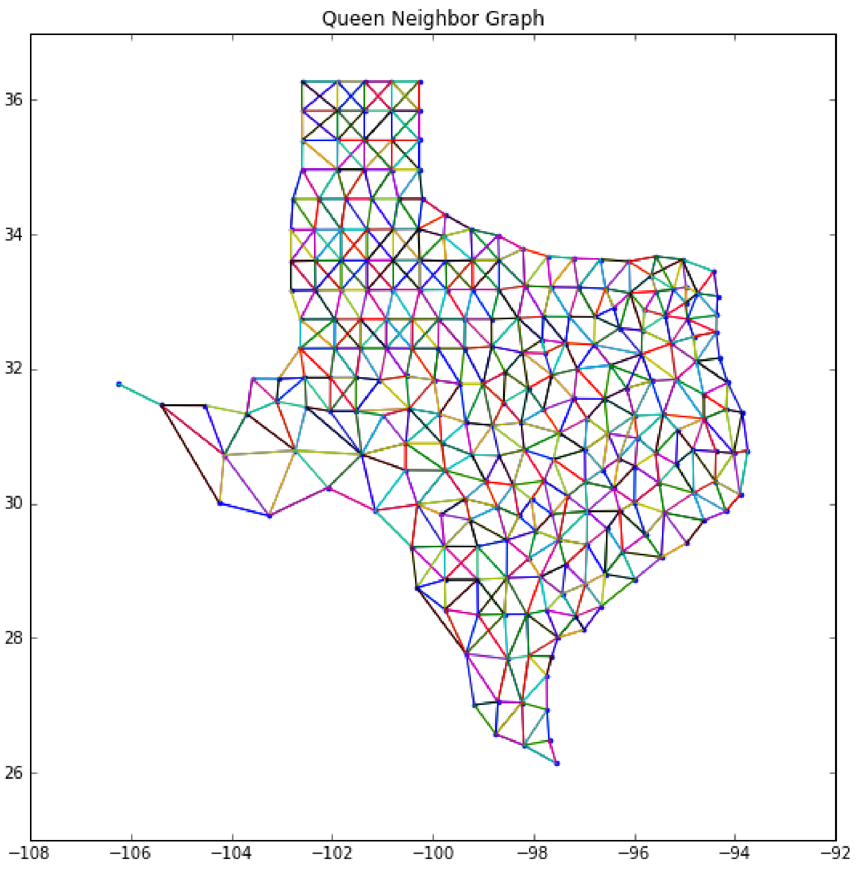
```
fig = figure(figsize=(9,9))
plt.plot(centroids[:,0], centroids[:,1],'.')
plt.ylim([25,37])
show()
```



```
wq.neighbors[0]
```

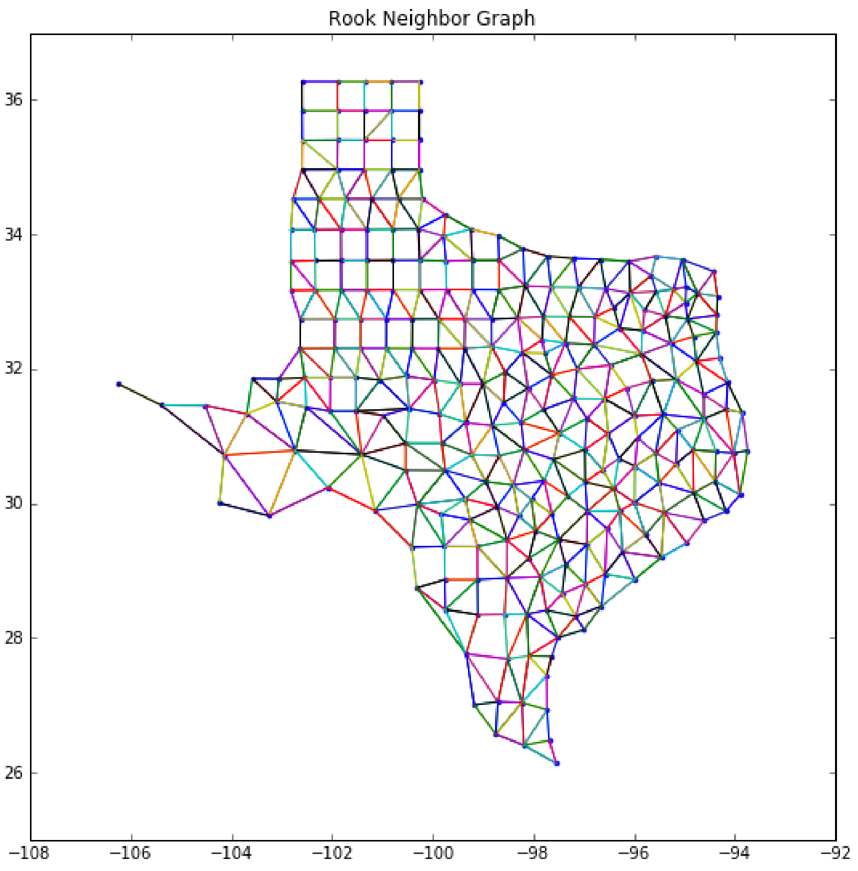
```
[4, 5, 6]
```

```
from pylab import figure, scatter, show
fig = figure(figsize=(9,9))

plt.plot(centroids[:,0], centroids[:,1],'.')
#plt.plot(s04[:,0], s04[:,1], '-')
plt.ylim([25,37])
for k,neighs in wq.neighbors.items():
    #print(k,neighs)
    origin = centroids[k]
    for neigh in neighs:
        segment = centroids[[k,neigh]]
        plt.plot(segment[:,0], segment[:,1], '-')
plt.title('Queen Neighbor Graph')
show()
```

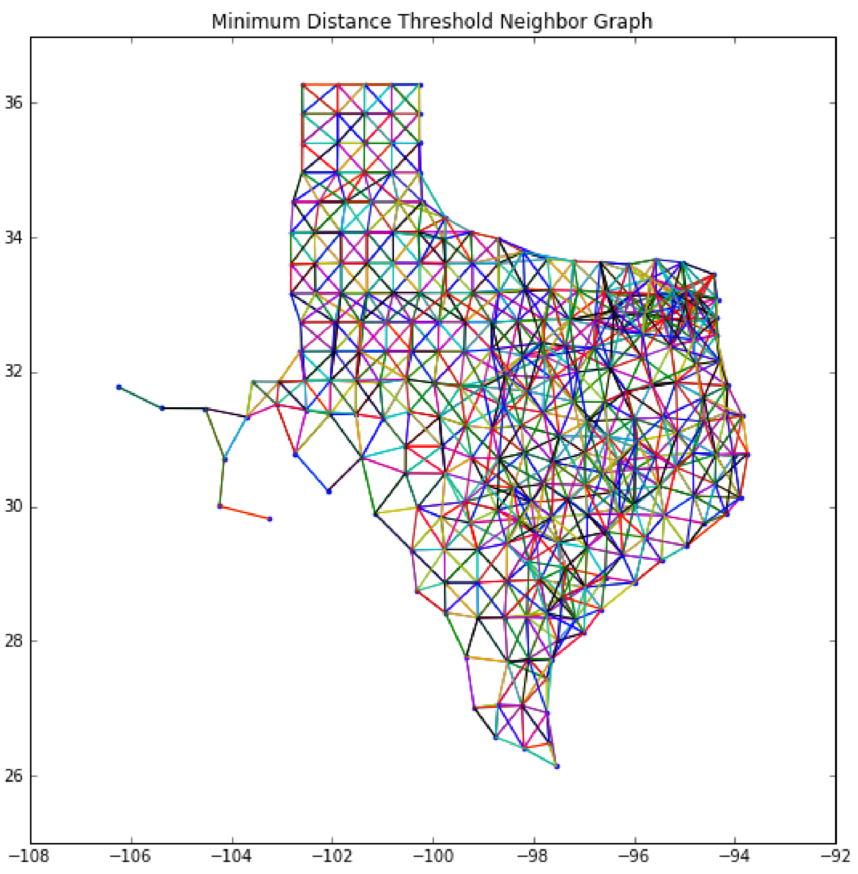


```
wr = ps.rook_from_shapefile('../data/texas.shp')
```

```
fig = figure(figsize=(9,9))

plt.plot(centroids[:,0], centroids[:,1],'.')
#plt.plot(s04[:,0], s04[:,1], '-')
plt.ylim([25,37])
for k,neighs in wr.neighbors.items():
    #print(k,neighs)
    origin = centroids[k]
    for neigh in neighs:
        segment = centroids[[k,neigh]]
        plt.plot(segment[:,0], segment[:,1], '-')
plt.title('Rook Neighbor Graph')
show()
```



```
fig = figure(figsize=(9,9))
plt.plot(centroids[:,0], centroids[:,1],'.')
#plt.plot(s04[:,0], s04[:,1], '-')
plt.ylim([25,37])
for k,neighs in Wmind.neighbors.items():
    origin = centroids[k]
    for neigh in neighs:
        segment = centroids[[k,neigh]]
        plt.plot(segment[:,0], segment[:,1], '-')
plt.title('Minimum Distance Threshold Neighbor Graph')
show()
```



```
Wmind.pct_nonzero
```

```
3.8378076756153514
```

```
wr.pct_nonzero
```

```
2.0243040486080974
```

```
wq.pct_nonzero
```

```
2.263004526009052
```

Exercise

1. Answer this question before writing any code: What spatial weights structure would be more dense, Texas counties based on rook contiguity or Texas counties based on knn with k=4?
2. Why?
3. Write code to see if you are correct.

Exploratory Spatial Data Analysis (ESDA)

IPYNB

```
%matplotlib inline
import pysal as ps
import pandas as pd
import numpy as np
from pysal.contrib.viz import mapping as maps
```

A well-used functionality in PySAL is the use of PySAL to conduct exploratory spatial data analysis. This notebook will provide an overview of ways to conduct exploratory spatial analysis in Python.

First, let's read in some data:

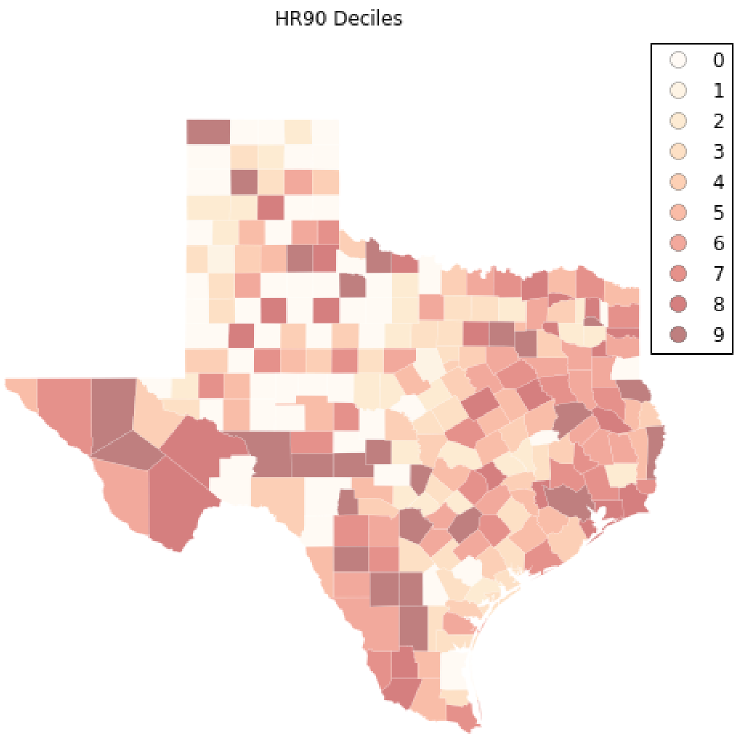
```
data = ps.pdio.read_files("../data/texas.shp")
```

```
data.head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FI
0	Lipscomb	Texas	48	295	482
1	Sherman	Texas	48	421	484
2	Dallam	Texas	48	111	481
3	Hansford	Texas	48	195	481
4	Ochiltree	Texas	48	357	483

5 rows × 70 columns

```
import matplotlib.pyplot as plt

import geopandas as gpd
shp_link = "../data/texas.shp"
tx = gpd.read_file(shp_link)
hr10 = ps.Quantiles(data.HR90, k=10)
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=hr10.yb).plot(column='cl', categorical=True, \
        k=10, cmap='OrRd', linewidth=0.1, ax=ax, \
        edgecolor='white', legend=True)
ax.set_axis_off()
plt.title("HR90 Deciles")
plt.show()
```



Spatial Autocorrelation

Visual inspection of the map pattern for HR90 deciles allows us to search for spatial structure. If the spatial distribution of the rates was random, then we should not see any clustering of similar values on the map. However, our visual system is drawn to the darker clusters in the south west as well as the east, and a concentration of the lighter hues (lower homicide rates) moving north to the pan handle.

Our brains are very powerful pattern recognition machines. However, sometimes they can be too powerful and lead us to detect false positives, or patterns where there are no statistical patterns. This is a particular concern when dealing with visualization of irregular polygons of differning sizes and shapes.

The concept of *spatial autocorrelation* relates to the combination of two types of similarity: spatial similarity and attribute similarity. Although there are many different measures of spatial autocorrelation, they all combine these two types of simmilarity into a summary measure.

Let's use PySAL to generate these two types of similarity measures.

Spatial Similarity

We have already encountered spatial weights in a previous notebook. In spatial autocorrelation analysis, the spatial weights are used to formalize the notion of spatial similarity. As we have seen there are many ways to define spatial weights, here we will use queen contiguity:

```
data = ps.pdio.read_files("../data/texas.shp")
W = ps.queen_from_shapefile("../data/texas.shp")
W.transform = 'r'
```

Attribute Similarity

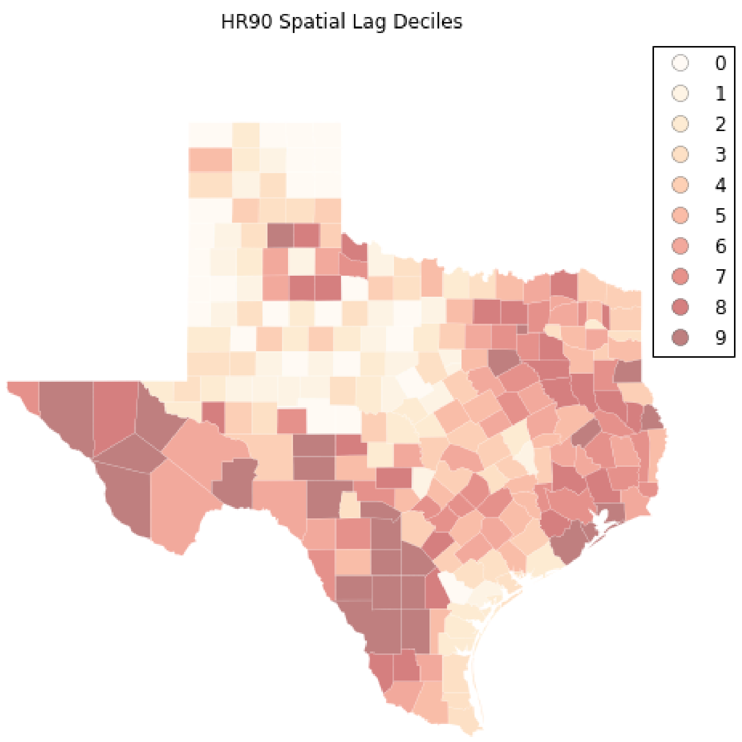
So the spatial weight between counties i and j indicates if the two counties are neighbors (i.e., geographically similar). What we also need is a measure of attribute similarity to pair up with this concept of spatial similarity. The **spatial lag** is a derived variable that accomplishes this for us. For county i the spatial lag is defined as:

$HR90Lag_i = \sum_j w_{i,j} HR90_j$

```
HR90Lag = ps.lag_spatial(W, data.HR90)
```

```
HR90LagQ10 = ps.Quantiles(HR90Lag, k=10)
```

```
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=HR90LagQ10.yb).plot(column='cl', categorical=True, \
        k=10, cmap='OrRd', linewidth=0.1, ax=ax, \
        edgecolor='white', legend=True)
ax.set_axis_off()
plt.title("HR90 Spatial Lag Deciles")

plt.show()
```

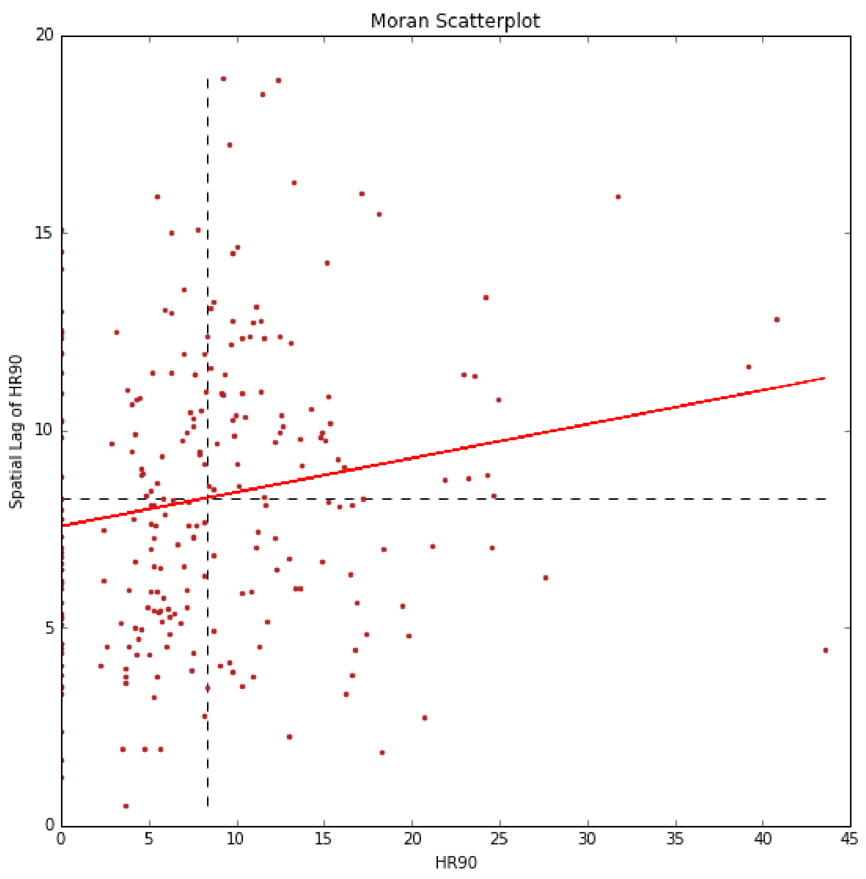


The decile map for the spatial lag tends to enhance the impression of value similarity in space. However, we still have the challenge of visually associating the value of the homicide rate in a county with the value of the spatial lag of rates for the county. The latter is a weighted average of homicide rates in the focal county's neighborhood.

To complement the geovisualization of these associations we can turn to formal statistical measures of spatial autocorrelation.

```
HR90 = data.HR90
b,a = np.polyfit(HR90, HR90Lag, 1)
```

```
f, ax = plt.subplots(1, figsize=(9, 9))

plt.plot(HR90, HR90Lag, '.', color='firebrick')

 # dashed vert at mean of the last year's PCI
plt.vlines(HR90.mean(), HR90Lag.min(), HR90Lag.max(), linestyle='--')
 # dashed horizontal at mean of lagged PCI
plt.hlines(HR90Lag.mean(), HR90.min(), HR90.max(), linestyle='--')

# red line of best fit using global I as slope
plt.plot(HR90, a + b*HR90, 'r')
plt.title('Moran Scatterplot')
plt.ylabel('Spatial Lag of HR90')
plt.xlabel('HR90')
plt.show()
```



Global Spatial Autocorrelation

In PySAL, commonly-used analysis methods are very easy to access. For example, if we were interested in examining the spatial dependence in `HR90` we could quickly compute a Moran's I statistic:

```
I_HR90 = ps.Moran(data.HR90.values, W)
```

```
I_HR90.I, I_HR90.p_sim
```

```
(0.085976640313889768, 0.012999999999999999)
```

Thus, the I statistic is 0.859 for this data, and has a very small p value.

```
b # note I is same as the slope of the line in the scatterplot
```

```
0.085976640313889505
```

We can visualize the distribution of simulated I statistics using the stored collection of simulated statistics:

```
I_HR90.sim[0:5]
```

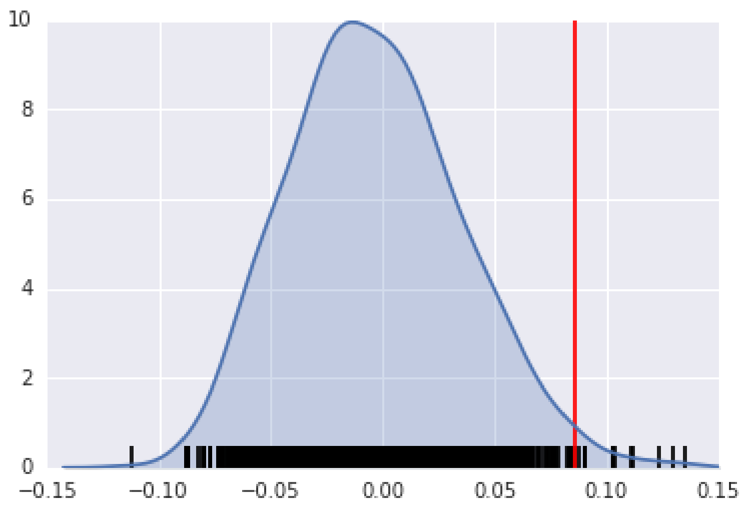
```
array([-0.05640543, -0.03158917,  0.0277026 ,  0.03998822, -0.01140814])
```

A simple way to visualize this distribution is to make a KDEplot (like we've done before), and add a rug showing all of the simulated points, and a vertical line denoting the observed value of the statistic:

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

```
sns.kdeplot(I_HR90.sim, shade=True)
plt.vlines(I_HR90.sim, 0, 0.5)
plt.vlines(I_HR90.I, 0, 10, 'r')
plt.xlim([-0.15, 0.15])
```

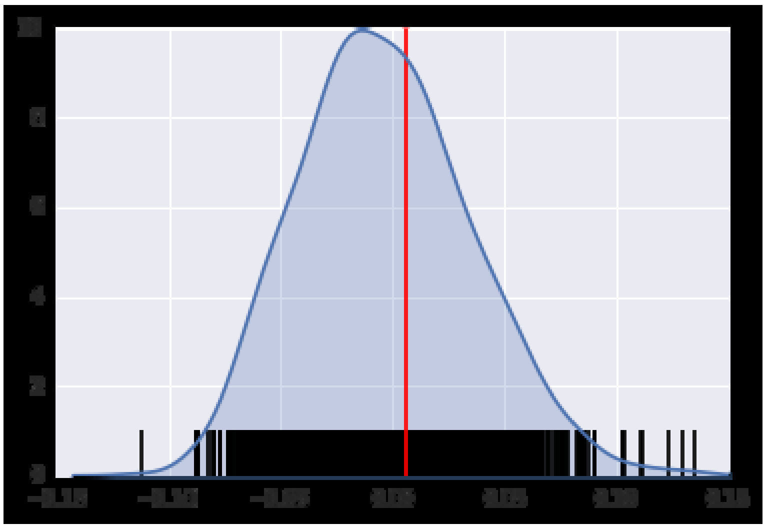
```
/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j




(-0.15, 0.15)
```



Instead, if our I statistic were close to our expected value, `I_HR90.EI`, our plot might look like this:

```
sns.kdeplot(I_HR90.sim, shade=True)
plt.vlines(I_HR90.sim, 0, 1)
plt.vlines(I_HR90.EI+.01, 0, 10, 'r')
plt.xlim([-0.15, 0.15])
```

```
/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j




(-0.15, 0.15)
```



The result of applying Moran's I is that we conclude the map pattern is not spatially random, but instead there is a signficant spatial association in homicide rates in Texas counties in 1990.

This result applies to the map as a whole, and is sometimes referred to as "global spatial autocorrelation". Next we turn to a local analysis where the attention shifts to detection of hot spots, cold spots and spatial outliers.

Local Autocorrelation Statistics

In addition to the Global autocorrelation statistics, PySAL has many local autocorrelation statistics. Let's compute a local Moran statistic for the same data shown above:

```
LMo_HR90 = ps.Moran_Local(data.HR90.values, W)
```

Now, instead of a single I statistic, we have an *array* of local I_i statistics, stored in the `.Is` attribute, and p-values from the simulation are in `p_sim`.

```
LMo_HR90.Is[0:10], LMo_HR90.p_sim[0:10]
```

```
(array([ 1.12087323,  0.47485223, -1.22758423,  0.93868661,  0.68974296,
         0.78503173,  0.71047515,  0.41060686,  0.00740368,  0.14866352]),
 array([ 0.013,  0.169,  0.037,  0.015,  0.002,  0.009,  0.053,  0.063,
         0.489,  0.119]))
```

We can adjust the number of permutations used to derive every *pseudo*-$p$ value by passing a different `permutations` argument:

```
LMo_HR90 = ps.Moran_Local(data.HR90.values, W, permutations=9999)
```

In addition to the typical clustermap, a helpful visualization for LISA statistics is a Moran scatterplot with statistically significant LISA values highlighted.

This is very simple, if we use the same strategy we used before:

First, construct the spatial lag of the covariate:

```
Lag_HR90 = ps.lag_spatial(W, data.HR90.values)
HR90 = data.HR90.values
```

Then, we want to plot the statistically-significant LISA values in a different color than the others. To do this, first find all of the statistically significant LISAs. Since the $p$-values are in the same order as the $I_i$ statistics, we can do this in the following way

```
sigs = HR90[LMo_HR90.p_sim <= .001]
W_sigs = Lag_HR90[LMo_HR90.p_sim <= .001]
insigs = HR90[LMo_HR90.p_sim > .001]
W_insigs = Lag_HR90[LMo_HR90.p_sim > .001]
```

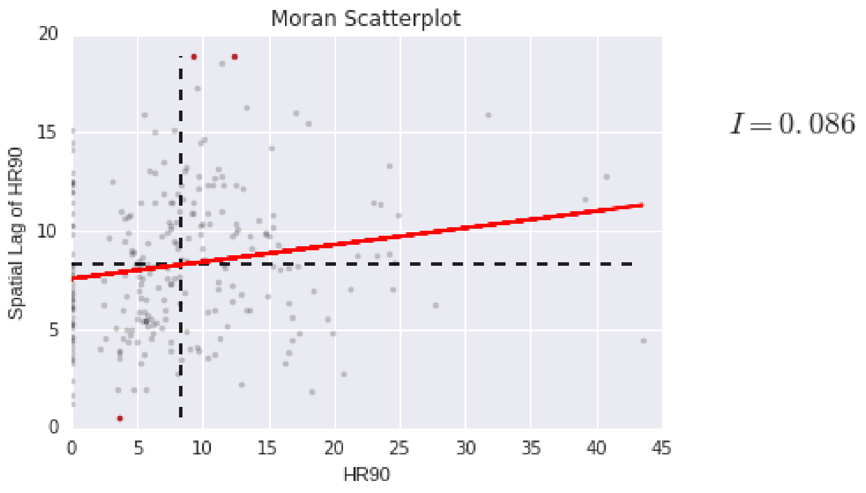
Then, since we have a lot of points, we can plot the points with a statistically insignficant LISA value lighter using the `alpha` keyword. In addition, we would like to plot the statistically significant points in a dark red color.

```
b,a = np.polyfit(HR90, Lag_HR90, 1)
```

Matplotlib has a list of named colors and will interpret colors that are provided in hexadecimal strings:

```
plt.plot(sigs, W_sigs, '.', color='firebrick')
plt.plot(insigs, W_insigs, '.k', alpha=.2)
 # dashed vert at mean of the last year's PCI
plt.vlines(HR90.mean(), Lag_HR90.min(), Lag_HR90.max(), linestyle='--')
 # dashed horizontal at mean of lagged PCI
plt.hlines(Lag_HR90.mean(), HR90.min(), HR90.max(), linestyle='--')

# red line of best fit using global I as slope
plt.plot(HR90, a + b*HR90, 'r')
plt.text(s='$I = %.3f$' % I_HR90.I, x=50, y=15, fontsize=18)
plt.title('Moran Scatterplot')
plt.ylabel('Spatial Lag of HR90')
plt.xlabel('HR90')
```

```
<matplotlib.text.Text at 0x7fd6cf324d30>
```



We can also make a LISA map of the data.

```
sig = LMo_HR90.p_sim < 0.05
```

```
sig.sum()
```

```
44
```

```
hotspots = LMo_HR90.q==1 * sig
```

```
hotspots.sum()
```

```
10
```

```
coldspots = LMo_HR90.q==3 * sig
```

```
coldspots.sum()
```

```
17
```

```
data.HR90[hotspots]
```

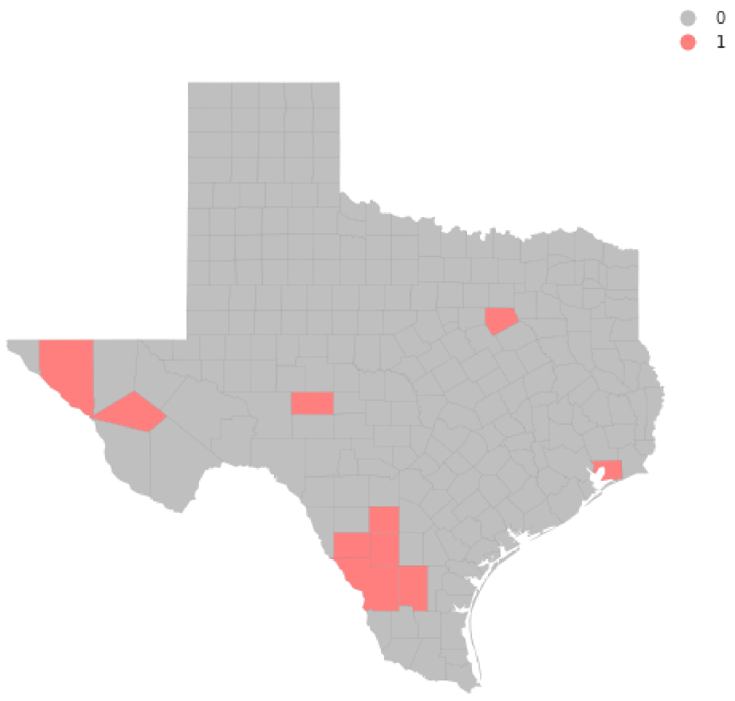
```
98      9.784698
132    11.435106
164    17.129154
166    11.148272
209    13.274924
229    12.371338
234    31.721863
236     9.584971
239     9.256549
242    18.062652
Name: HR90, dtype: float64
```

```
data[hotspots]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS
98	Ellis	Texas	48	139
132	Hudspeth	Texas	48	229
164	Jeff Davis	Texas	48	243
166	Schleicher	Texas	48	413
209	Chambers	Texas	48	071
229	Frio	Texas	48	163
234	La Salle	Texas	48	283
236	Dimmit	Texas	48	127
239	Webb	Texas	48	479
242	Duval	Texas	48	131

10 rows × 70 columns

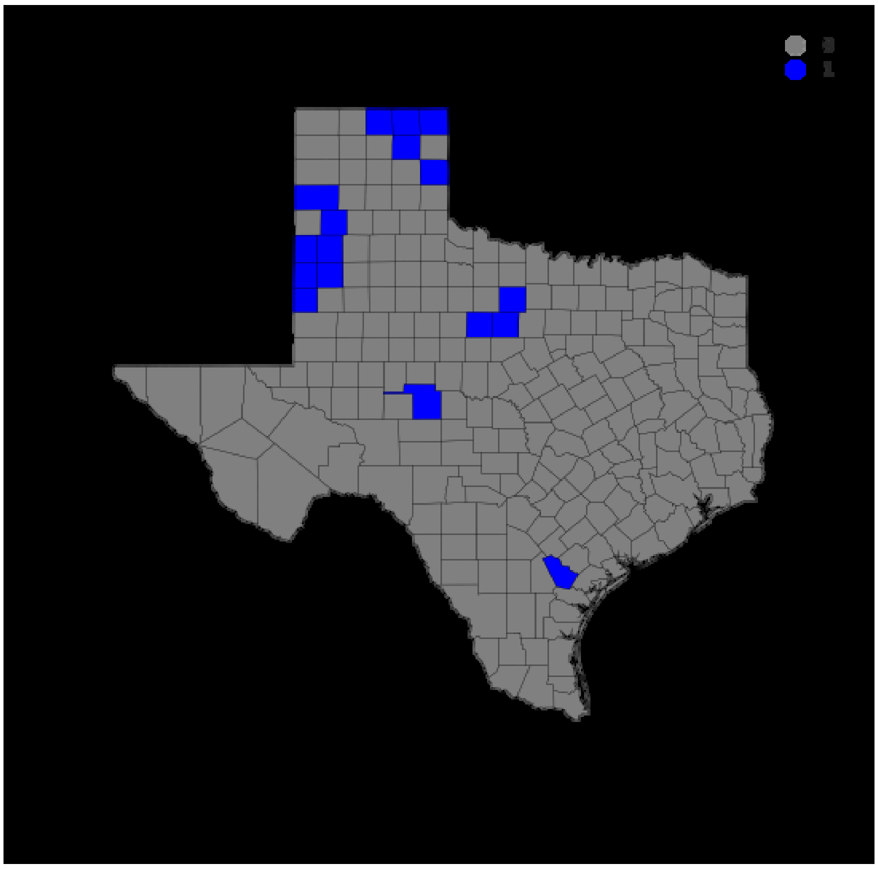
```
from matplotlib import colors
hmap = colors.ListedColormap(['grey', 'red'])
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=hotspots*1).plot(column='cl', categorical=True, \
        k=2, cmap=hmap, linewidth=0.1, ax=ax, \
        edgecolor='grey', legend=True)
ax.set_axis_off()
plt.show()
```



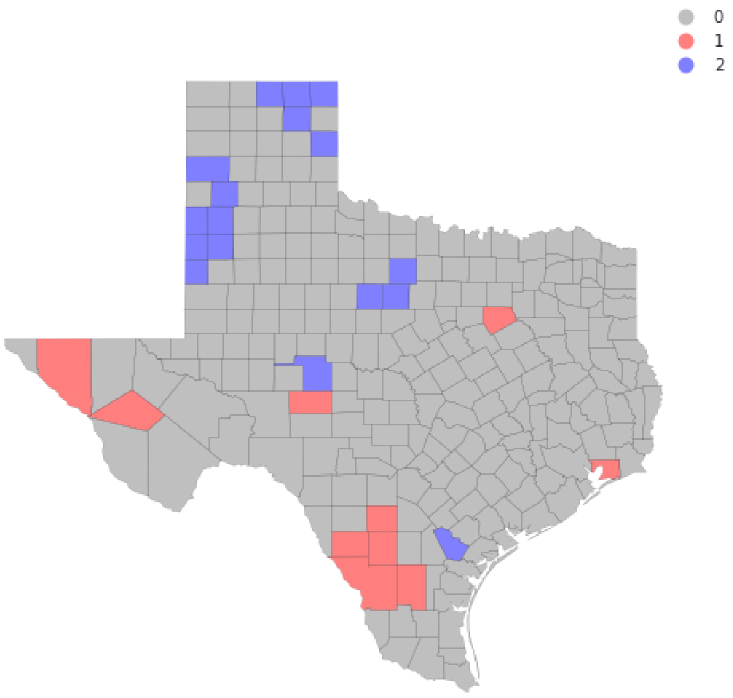
```
data.HR90[coldspots]
```

```
0      0.000000
3      0.000000
4      3.651767
5      0.000000
13     5.669899
19     3.480743
21     3.675119
32     2.211607
33     4.718762
48     5.509870
51     0.000000
62     3.677958
69     0.000000
81     0.000000
87     3.699593
140    8.125292
233    5.304688
Name: HR90, dtype: float64
```

```
cmap = colors.ListedColormap(['grey', 'blue'])
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=coldspots*1).plot(column='cl', categorical=True, \
        k=2, cmap=cmap, linewidth=0.1, ax=ax, \
        edgecolor='black', legend=True)
ax.set_axis_off()
plt.show()
```

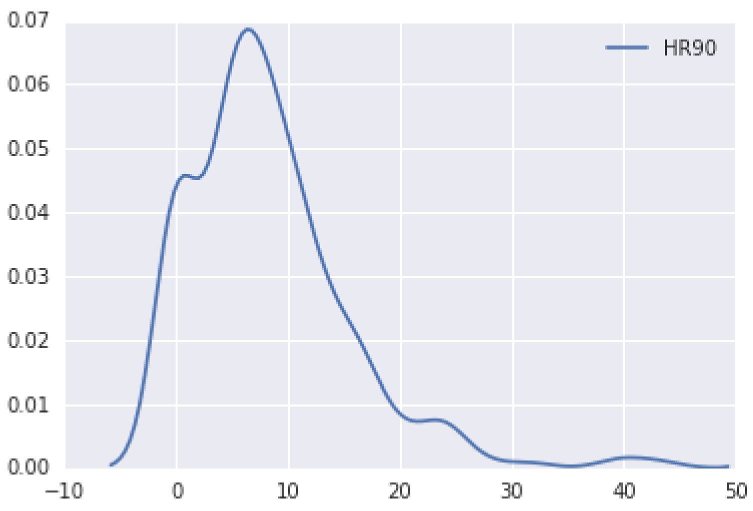


```python
from matplotlib import colors
hcmap = colors.ListedColormap(['grey', 'red','blue'])
hotcold = hotspots*1 + coldspots*2
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=hotcold).plot(column='cl', categorical=True, \
        k=2, cmap=hcmap,linewidth=0.1, ax=ax, \
        edgecolor='black', legend=True)
ax.set_axis_off()
plt.show()
```



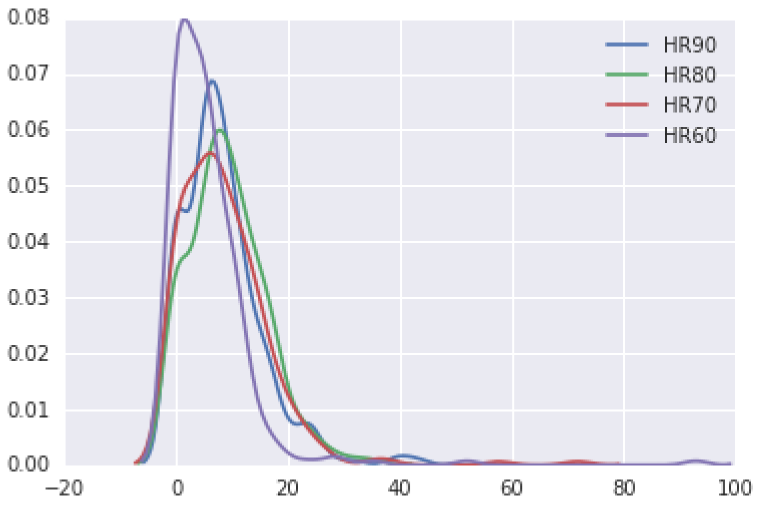
```
sns.kdeplot(data.HR90)
```

```
/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j





<matplotlib.axes._subplots.AxesSubplot at 0x7fd6ccc17358>
```



```
sns.kdeplot(data.HR90)
sns.kdeplot(data.HR80)
sns.kdeplot(data.HR70)
sns.kdeplot(data.HR60)
```

```
/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j




<matplotlib.axes._subplots.AxesSubplot at 0x7fd6da838908>
```



```
data.HR90.mean()
```

```
8.302494460285041
```

```
data.HR90.median()
```

```
7.23234613355
```

Exercises

1. Repeat the global analysis for the years 1960, 70, 80 and compare the results to what we found in 1990.
2. The local analysis can also be repeated for the other decades. How many counties are hot spots in each of the periods?
3. The recent Brexit vote provides a timely example where local spatial autocorrelation analysis can provide interesting insights. One local analysis of the vote to leave has recently been repored. Extend this to do an analysis of the attribute `Pct_remain`. Do the hot spots for the leave vote concord with the cold spots for the remain vote?

Exploratory Spatial and Temporal Data Analysis (ESTDA)

IPYNB

```
import matplotlib
import numpy as np
import pysal as ps
import matplotlib.pyplot as plt
%matplotlib inline
```

```
f = ps.open(ps.examples.get_path('usjoin.csv'), 'r')
```

To determine what is in the file, check the `header` attribute on the file object:

```
f.header[0:10]
```

```
['Name',
 'STATE_FIPS',
 '1929',
 '1930',
 '1931',
 '1932',
 '1933',
 '1934',
 '1935',
 '1936']
```

Ok, lets pull in the `name` variable to see what we have.

```
name = f.by_col('Name')
```

```
name
```

```
['Alabama',
 'Arizona',
 'Arkansas',
 'California',
 'Colorado',
 'Connecticut',
 'Delaware',
 'Florida',
 'Georgia',
 'Idaho',
 'Illinois',
 'Indiana',
 'Iowa',
 'Kansas',
 'Kentucky',
 'Louisiana',
 'Maine',
 'Maryland',
 'Massachusetts',
 'Michigan',
 'Minnesota',
 'Mississippi',
 'Missouri',
 'Montana',
 'Nebraska',
 'Nevada',
 'New Hampshire',
 'New Jersey',
 'New Mexico',
 'New York',
 'North Carolina',
 'North Dakota',
 'Ohio',
 'Oklahoma',
 'Oregon',
 'Pennsylvania',
 'Rhode Island',
 'South Carolina',
 'South Dakota',
 'Tennessee',
 'Texas',
 'Utah',
 'Vermont',
 'Virginia',
 'Washington',
 'West Virginia',
 'Wisconsin',
 'Wyoming']
```

Now obtain per capital incomes in 1929 which is in the column associated with `1929`.

```
y1929 = f.by_col('1929')
```

```
y1929[:10]
```

```
[323, 600, 310, 991, 634, 1024, 1032, 518, 347, 507]
```

And now 2009

```
y2009 = f.by_col("2009")
```

```
y2009[:10]
```

```
[32274, 32077, 31493, 40902, 40093, 52736, 40135, 36565, 33086, 30987]
```

These are read into regular Python lists which are not particularly well suited to efficient data analysis. So let's convert them to numpy arrays.

```
y2009 = np.array(y2009)
```

```
y2009
```

```
array([32274, 32077, 31493, 40902, 40093, 52736, 40135, 36565, 33086,
       30987, 40933, 33174, 35983, 37036, 31250, 35151, 35268, 47159,
       49590, 34280, 40920, 29318, 35106, 32699, 37057, 38009, 41882,
       48123, 32197, 46844, 33564, 38672, 35018, 33708, 35210, 38827,
       41283, 30835, 36499, 33512, 35674, 30107, 36752, 43211, 40619,
       31843, 35676, 42504])
```

Much better. But pulling these in and converting them a column at a time is tedious and error prone. So we will do all of this in a list comprehension.

```
Y = np.array( [ f.by_col(str(year)) for year in range(1929,2010) ] ) * 1.0
```

```
Y.shape
```

```
(81, 48)
```

```
Y = Y.transpose()
```

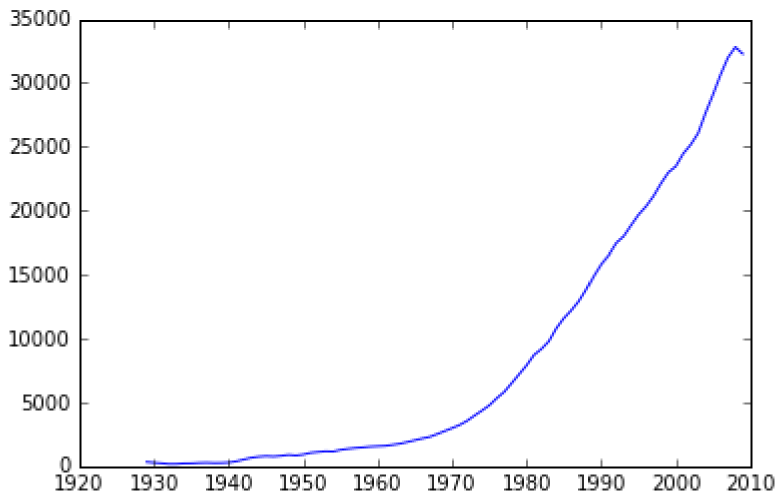
```
Y.shape
```

```
(48, 81)
```

```
years = np.arange(1929,2010)
```

```
plt.plot(years,Y[0])
```

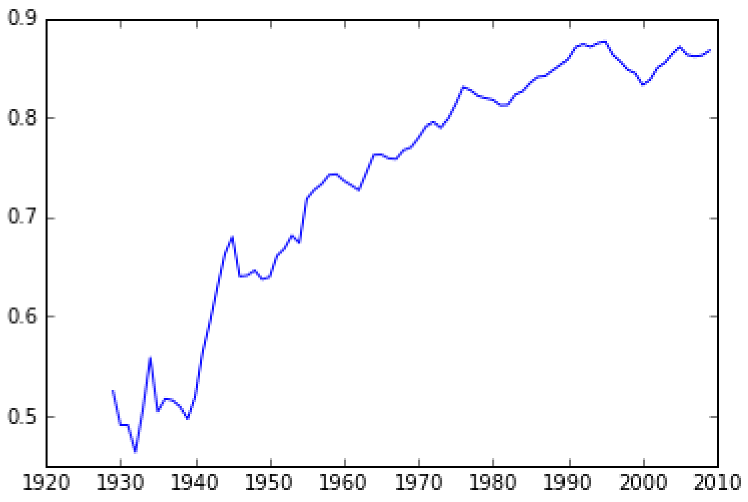
```
[<matplotlib.lines.Line2D at 0x110ba1a58>]
```



```
RY = Y / Y.mean(axis=0)
```

```
plt.plot(years,RY[0])
```

```
[<matplotlib.lines.Line2D at 0x113575e10>]
```



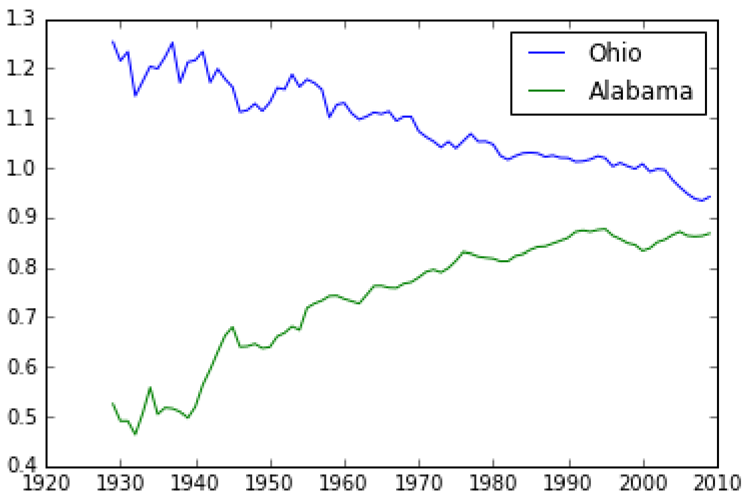
```
name = np.array(name)
```

```
np.nonzero(name=='Ohio')
```

```
(array([32]),)
```

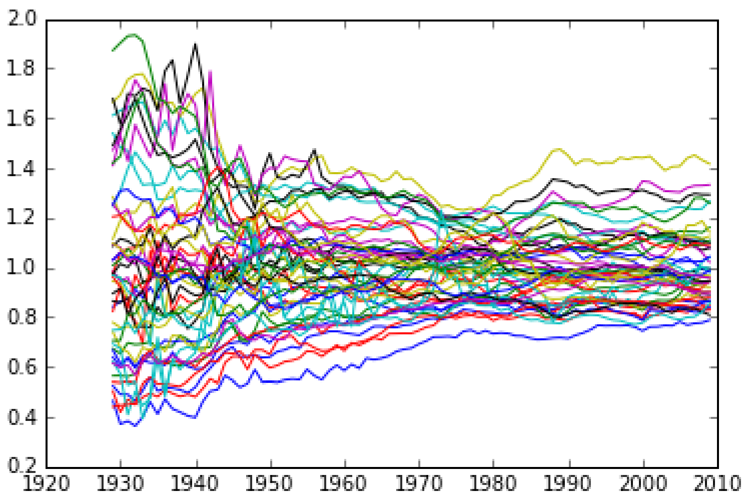
```
plt.plot(years, RY[32], label='Ohio')
plt.plot(years, RY[0], label='Alabama')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1137d9eb8>
```



Spaghetti Plot

```
for row in RY:
    plt.plot(years, row)
```



Kernel Density (univariate, aspatial)

```
from scipy.stats.kde import gaussian_kde
```

```
density = gaussian_kde(Y[:,0])
```

```
Y[:,0]
```

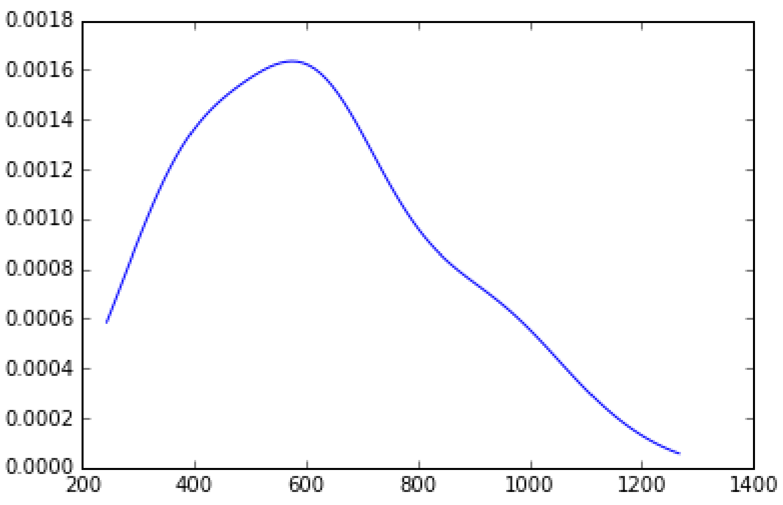
```
array([  323.,   600.,   310.,   991.,   634.,  1024.,  1032.,   518.,
         347.,   507.,   948.,   607.,   581.,   532.,   393.,   414.,
         601.,   768.,   906.,   790.,   599.,   286.,   621.,   592.,
         596.,   868.,   686.,   918.,   410.,  1152.,   332.,   382.,
         771.,   455.,   668.,   772.,   874.,   271.,   426.,   378.,
         479.,   551.,   634.,   434.,   741.,   460.,   673.,   675.])
```

```
density = gaussian_kde(Y[:,0])
```

```
minY0 = Y[:,0].min()*.90
maxY0 = Y[:,0].max()*1.10
x = np.linspace(minY0, maxY0, 100)
```

```
plt.plot(x,density(x))
```

```
[<matplotlib.lines.Line2D at 0x113d2a748>]
```

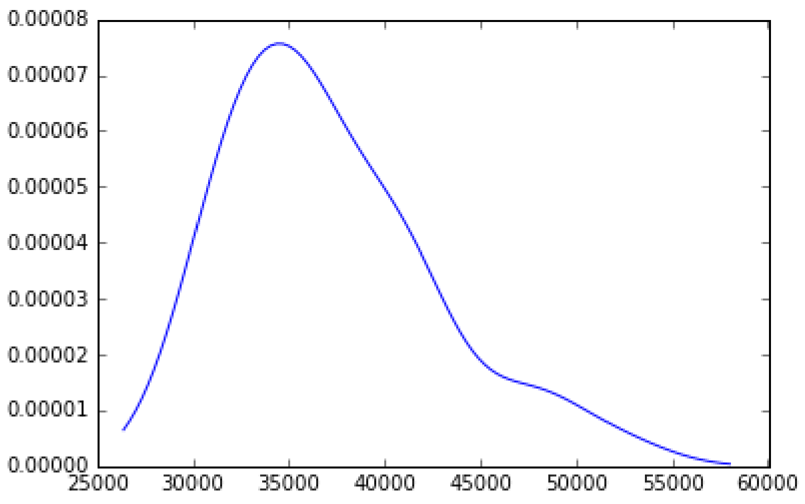


```
d2009 = gaussian_kde(Y[:,-1])
```

```
minY0 = Y[:,-1].min()*.90
maxY0 = Y[:,-1].max()*1.10
x = np.linspace(minY0, maxY0, 100)
```

```
plt.plot(x,d2009(x))
```

```
[<matplotlib.lines.Line2D at 0x113a48358>]
```



```
minR0 = RY.min()
```

```
maxR0 = RY.max()
```

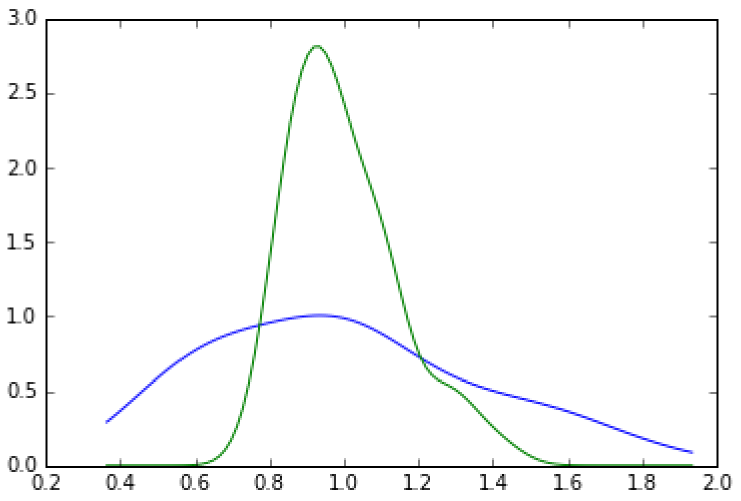
```
x = np.linspace(minR0, maxR0, 100)
```

```
d1929 = gaussian_kde(RY[:,0])
```

```
d2009 = gaussian_kde(RY[:,-1])
```

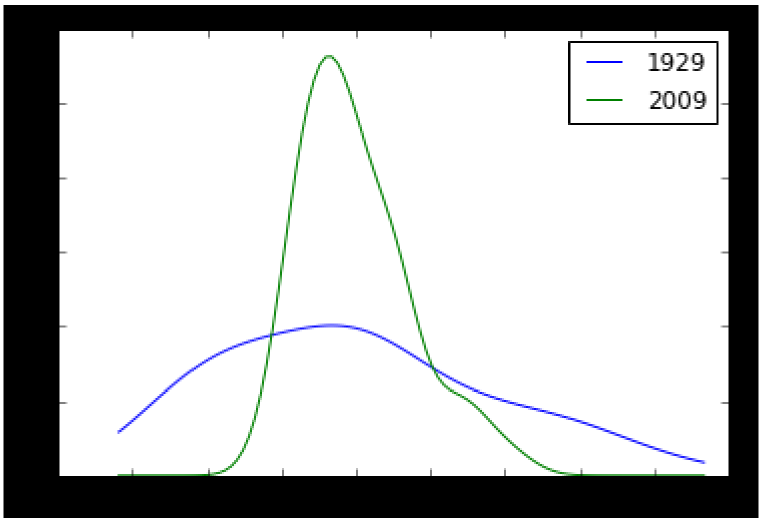
```
plt.plot(x, d1929(x))
plt.plot(x, d2009(x))
```

```
[<matplotlib.lines.Line2D at 0x113d035c0>]
```



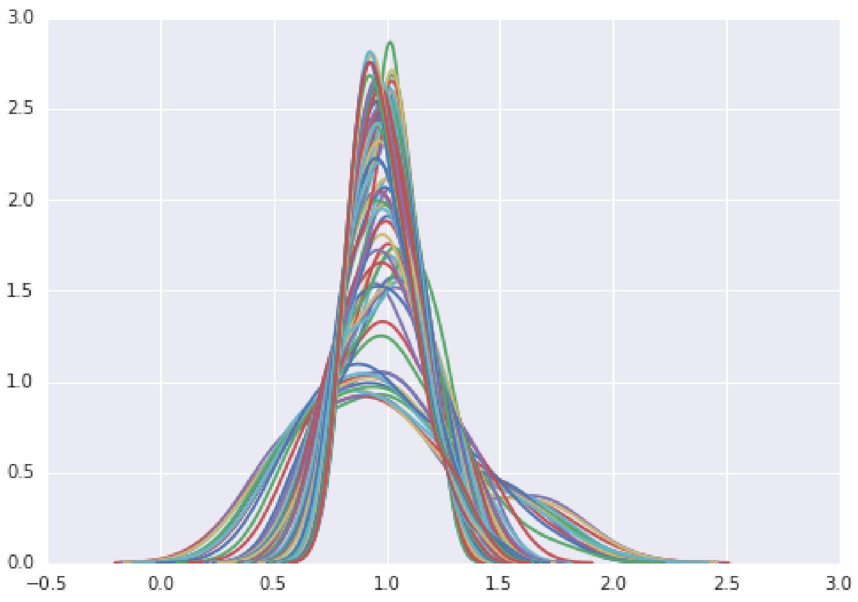
```
plt.plot(x, d1929(x), label='1929')
plt.plot(x, d2009(x), label='2009')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x113a4a908>
```



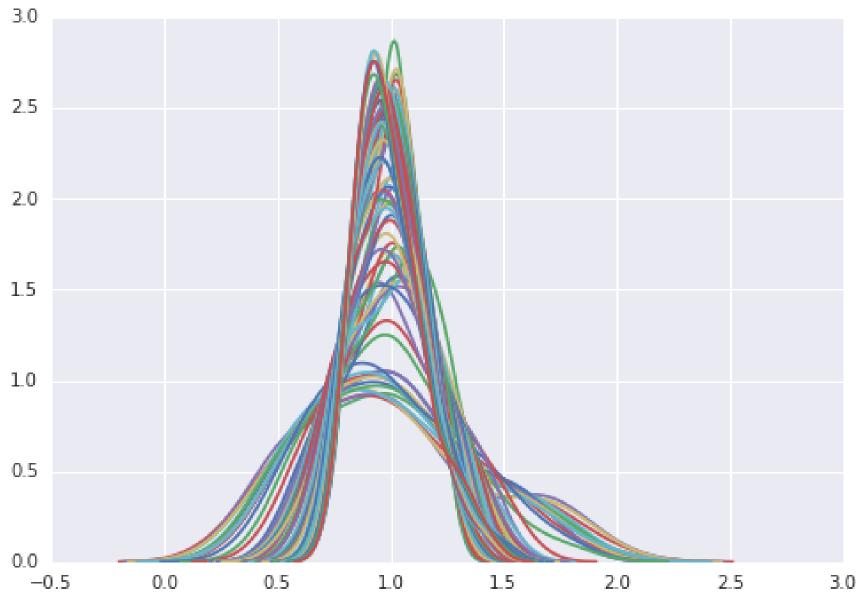
```
import seaborn as sns
for y in range(2010-1929):
    sns.kdeplot(RY[:,y])
#sns.kdeplot(data.HR80)
#sns.kdeplot(data.HR70)
#sns.kdeplot(data.HR60)
```

```
/Users/dani/anaconda/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels/
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

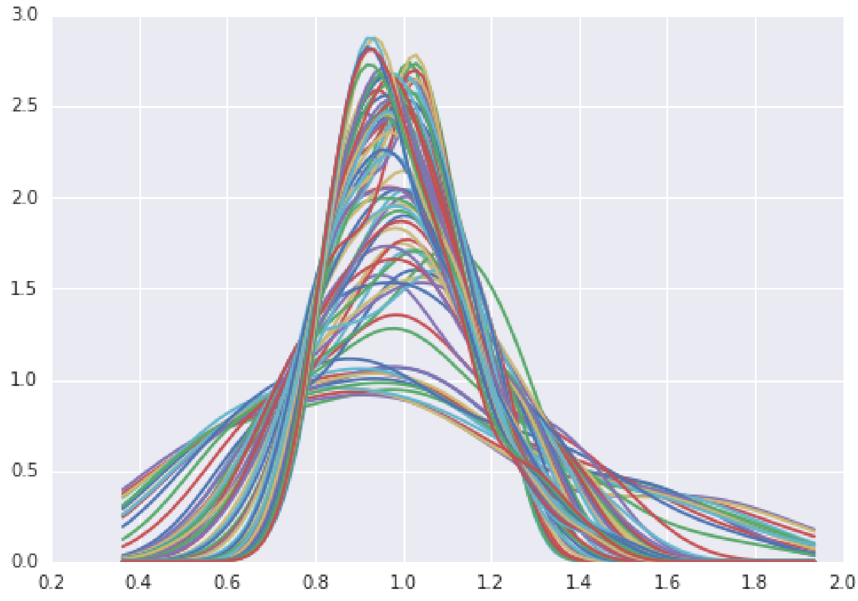


```
import seaborn as sns
for y in range(2010-1929):
    sns.kdeplot(RY[:,y])
```

```
/Users/dani/anaconda/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels/
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```



```
for cs in RY.T: # take cross sections
    plt.plot(x, gaussian_kde(cs)(x))
```

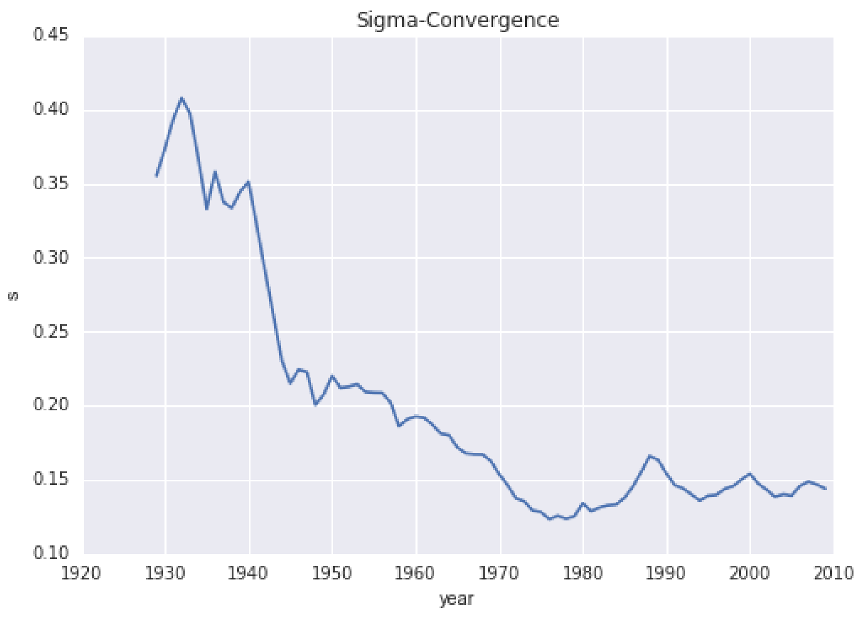


```
cs[0]
```

```
0.86746356478544273
```

```
sigma = RY.std(axis=0)
plt.plot(years, sigma)
plt.ylabel('s')
plt.xlabel('year')
plt.title("Sigma-Convergence")
```

```
<matplotlib.text.Text at 0x11439c470>
```



So the distribution is becoming less dispersed over time.

But what about internal mixing? Do poor (rich) states remain poor (rich), or is there movement within the distribuiton over time?

Markov Chains

```
c = np.array([
['b','a','c'],
['c','c','a'],
['c','b','c'],
['a','a','b'],
['a','b','c']])
```

```
c
```

```
array([['b', 'a', 'c'],
       ['c', 'c', 'a'],
       ['c', 'b', 'c'],
       ['a', 'a', 'b'],
       ['a', 'b', 'c']],
      dtype='<U1')
```

```
m = ps.Markov(c)
```

```
m.classes
```

```
array(['a', 'b', 'c'],
      dtype='<U1')
```

```
m.transitions
```

```
array([[ 1.,  2.,  1.],
       [ 1.,  0.,  2.],
       [ 1.,  1.,  1.]])
```

```
m.p
```

```
matrix([[ 0.25      ,  0.5       ,  0.25      ],
        [ 0.33333333,  0.        ,  0.66666667],
        [ 0.33333333,  0.33333333,  0.33333333]])
```

State Per Capita Incomes

```
ps.examples.explain('us_income')
```

```
{'description': 'Per-capita income for the lower 47 US states 1929-2010',
 'explanation': [' * us48.shp: shapefile ',
  ' * us48.dbf: dbf for shapefile',
  ' * us48.shx: index for shapefile',
  ' * usjoin.csv: attribute data (comma delimited file)'],
 'name': 'us_income'}
```

```
data = ps.pdio.read_files(ps.examples.get_path("us48.dbf"))
W = ps.queen_from_shapefile(ps.examples.get_path("us48.shp"))
W.transform = 'r'
```

```
data.STATE_NAME
```

```
0         Washington
1            Montana
2              Maine
3       North Dakota
4       South Dakota
5            Wyoming
6          Wisconsin
7              Idaho
8            Vermont
9          Minnesota
10            Oregon
11     New Hampshire
12              Iowa
13     Massachusetts
14          Nebraska
15          New York
16      Pennsylvania
17       Connecticut
18      Rhode Island
19        New Jersey
20           Indiana
21            Nevada
22              Utah
23        California
24              Ohio
25          Illinois
26          Delaware
27     West Virginia
28          Maryland
29          Colorado
30          Kentucky
31            Kansas
32          Virginia
33          Missouri
34           Arizona
35          Oklahoma
36    North Carolina
37         Tennessee
38             Texas
39        New Mexico
40           Alabama
41       Mississippi
42           Georgia
43    South Carolina
44          Arkansas
45         Louisiana
46           Florida
47          Michigan
Name: STATE_NAME, dtype: object
```

```
f = ps.open(ps.examples.get_path("usjoin.csv"))
pci = np.array([f.by_col[str(y)] for y in range(1929,2010)])
pci.shape
```

```
(81, 48)
```

```
pci = pci.T
```

```
pci.shape
```

```
(48, 81)
```

```
cnames = f.by_col('Name')
```

```
cnames[:10]
```

```
['Alabama',
 'Arizona',
 'Arkansas',
 'California',
 'Colorado',
 'Connecticut',
 'Delaware',
 'Florida',
 'Georgia',
 'Idaho']
```

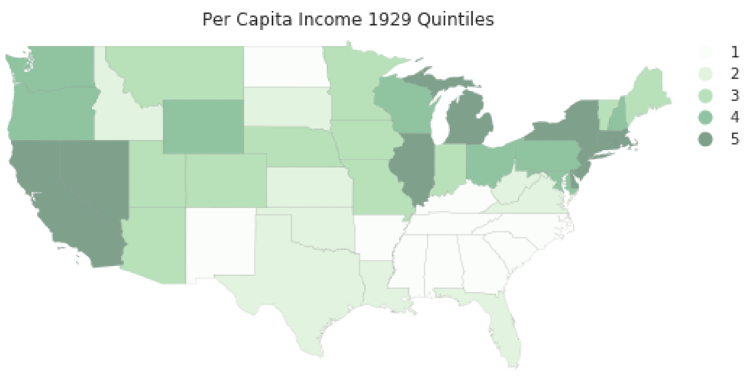
```
ids = [ cnames.index(name) for name in data.STATE_NAME]
```

```
ids[:10]
```

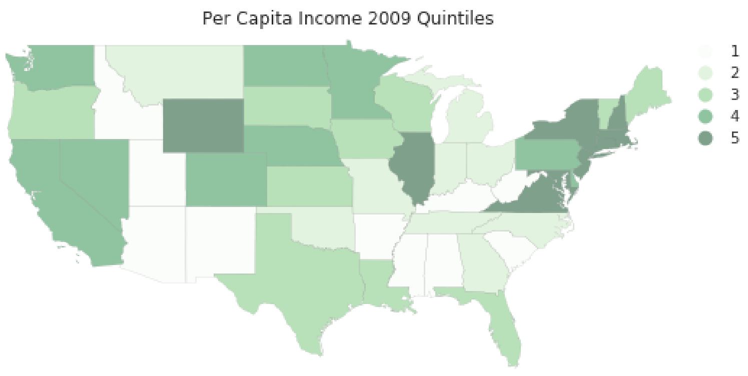
```
[44, 23, 16, 31, 38, 47, 46, 9, 42, 20]
```

```
pci = pci[ids]
RY = RY[ids]
```

```
import matplotlib.pyplot as plt

import geopandas as gpd
shp_link = ps.examples.get_path('us48.shp')
tx = gpd.read_file(shp_link)
pci29 = ps.Quantiles(pci[:,0], k=5)
f, ax = plt.subplots(1, figsize=(10, 5))
tx.assign(cl=pci29.yb+1).plot(column='cl', categorical=True, \
        k=5, cmap='Greens', linewidth=0.1, ax=ax, \
        edgecolor='grey', legend=True)
ax.set_axis_off()
plt.title('Per Capita Income 1929 Quintiles')

plt.show()
```



```
pci2009 = ps.Quantiles(pci[:,-1], k=5)
f, ax = plt.subplots(1, figsize=(10, 5))
tx.assign(cl=pci2009.yb+1).plot(column='cl', categorical=True, \
        k=5, cmap='Greens', linewidth=0.1, ax=ax, \
        edgecolor='grey', legend=True)
ax.set_axis_off()
plt.title('Per Capita Income 2009 Quintiles')
plt.show()
```



convert to a code cell to generate a time series of the maps

for y in range(2010-1929): pciy = ps.Quantiles(pci[:,y], k=5) f, ax = plt.subplots(1, figsize=(10, 5)) tx.assign(cl=pciy.yb+1).plot(column='cl', categorical=True, \ k=5, cmap='Greens', linewidth=0.1, ax=ax, \ edgecolor='grey', legend=True) ax.set_axis_off() plt.title("Per Capita Income %d Quintiles"%(1929+y)) plt.show()

Put series into cross-sectional quintiles (i.e., quintiles for each year).

```
q5 = np.array([ps.Quantiles(y).yb for y in pci.T]).transpose()
```

```
q5.shape
```

```
(48, 81)
```

```
q5[:,0]
```

```
array([3, 2, 2, 0, 1, 3, 3, 1, 2, 2, 3, 3, 2, 4, 2, 4, 3, 4, 4, 4, 2, 4, 2,
       4, 3, 4, 4, 1, 3, 2, 0, 1, 1, 2, 2, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1,
       1, 4])
```

```
pci.shape
```

```
(48, 81)
```

```
pci[0]
```

```
array([  741,   658,   534,   402,   376,   443,   490,   569,   599,
         582,   614,   658,   864,  1196,  1469,  1527,  1419,  1401,
        1504,  1624,  1595,  1721,  1874,  1973,  2066,  2077,  2116,
        2172,  2262,  2281,  2380,  2436,  2535,  2680,  2735,  2858,
        3078,  3385,  3566,  3850,  4097,  4205,  4381,  4731,  5312,
        5919,  6533,  7181,  7832,  8887,  9965, 10913, 11903, 12431,
       13124, 14021, 14738, 15522, 16300, 17270, 18670, 20026, 20901,
       21917, 22414, 23119, 23878, 25287, 26817, 28632, 30392, 31528,
       32053, 32206, 32934, 34984, 35738, 38477, 40782, 41588, 40619])
```

we are looping over the rows of y which is ordered $T \times n$ (rows are cross sections, row 0 is the cross-section for period 0.

```
m5 = ps.Markov(q5)
```

```
m5.classes
```

```
array([0, 1, 2, 3, 4])
```

```
m5.transitions
```

```
array([[ 729.,   71.,    1.,    0.,    0.],
       [  72.,  567.,   80.,    3.,    0.],
       [   0.,   81.,  631.,   86.,    2.],
       [   0.,    3.,   86.,  573.,   56.],
       [   0.,    0.,    1.,   57.,  741.]])
```

```
np.set_printoptions(3, suppress=True)
m5.p
```

```
matrix([[ 0.91 ,  0.089,  0.001,  0.   ,  0.   ],
        [ 0.1  ,  0.785,  0.111,  0.004,  0.   ],
        [ 0.   ,  0.101,  0.789,  0.107,  0.003],
        [ 0.   ,  0.004,  0.12 ,  0.798,  0.078],
        [ 0.   ,  0.   ,  0.001,  0.071,  0.927]])
```

```
m5.steady_state #steady state distribution
```

```
matrix([[ 0.208],
        [ 0.187],
        [ 0.207],
        [ 0.188],
        [ 0.209]])
```

```
fmpt = ps.ergodic.fmpt(m5.p) #first mean passage time
fmpt
```

```
matrix([[   4.814,   11.503,   29.609,   53.386,  103.598],
        [  42.048,    5.34 ,   18.745,   42.5  ,   92.713],
        [  69.258,   27.211,    4.821,   25.272,   75.433],
        [  84.907,   42.859,   17.181,    5.313,   51.61 ],
        [  98.413,   56.365,   30.66 ,   14.212,    4.776]])
```

For a state with income in the first quintile, it takes on average 11.5 years for it to first enter the second quintile, 29.6 to get to the third quintile, 53.4 years to enter the fourth, and 103.6 years to reach the richest quintile.

But, this approach assumes the movement of a state in the income distribution is independent of the movement of its neighbors or the position of the neighbors in the distribution. Does spatial context matter?

Dynamics of Spatial Dependence

Create a queen contiguity matrix that is row standardized

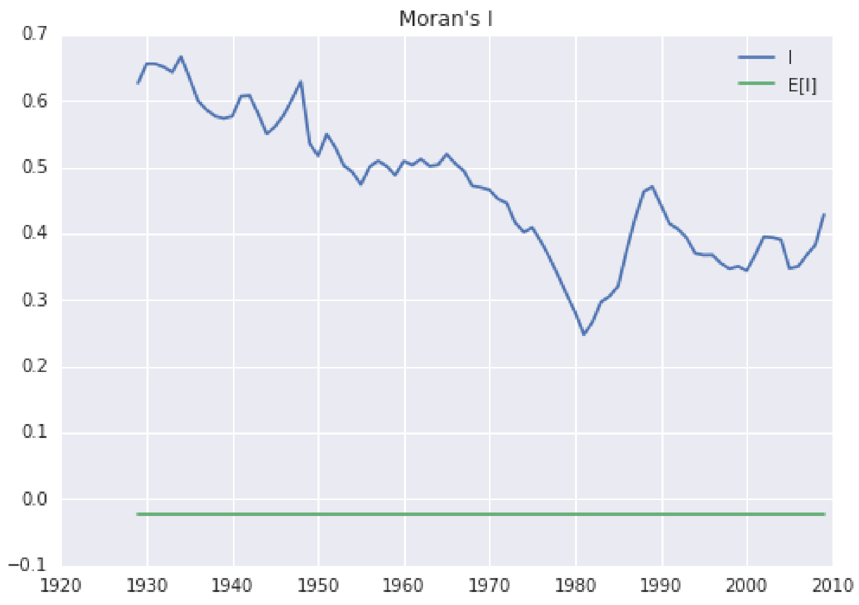
```
w = ps.queen_from_shapefile(ps.examples.get_path('us48.shp'))
w.transform = 'R'
```

```
mits = [ps.Moran(cs, w) for cs in RY.T]
```

```
res = np.array([(m.I, m.EI, m.p_sim, m.z_sim) for m in mits])
```

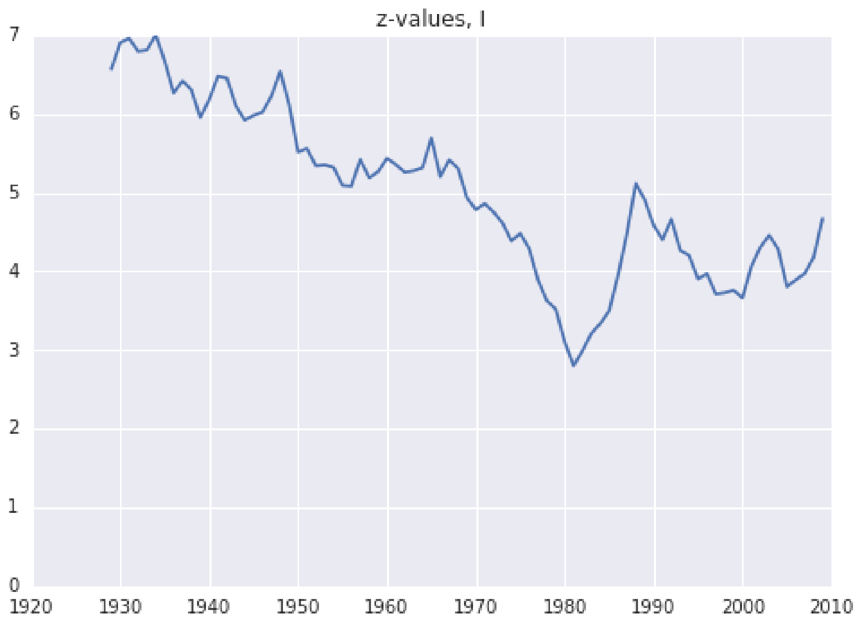
```
plt.plot(years, res[:,0], label='I')
plt.plot(years, res[:,1], label='E[I]')
plt.title("Moran's I")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f912bf8d438>
```



```
plt.plot(years, res[:,-1])
plt.ylim(0,7.0)
plt.title('z-values, I')
```

```
<matplotlib.text.Text at 0x7f912beb4da0>
```



Spatial Markov

```
pci.shape
```

```
(48, 81)
```

```
rpci = pci / pci.mean(axis=0)
```

```
rpci[:,0]
```

```
array([ 1.204,  0.962,  0.977,  0.621,  0.692,  1.097,  1.094,  0.824,
        1.031,  0.974,  1.086,  1.115,  0.944,  1.473,  0.969,  1.873,
        1.255,  1.664,  1.421,  1.492,  0.987,  1.411,  0.896,  1.611,
        1.253,  1.541,  1.677,  0.748,  1.248,  1.031,  0.639,  0.865,
        0.705,  1.009,  0.975,  0.74 ,  0.54 ,  0.614,  0.779,  0.666,
        0.525,  0.465,  0.564,  0.441,  0.504,  0.673,  0.842,  1.284])
```

```
rpci[:,0].mean()
```

```
0.99999999999999989
```

```
sm = ps.Spatial_Markov(rpci, W, fixed=True, k=5)
```

```
sm.p
```

```
matrix([[ 0.915,  0.075,  0.009,  0.001,  0.   ],
        [ 0.066,  0.827,  0.105,  0.001,  0.001],
        [ 0.005,  0.103,  0.794,  0.095,  0.003],
        [ 0.   ,  0.009,  0.094,  0.849,  0.048],
        [ 0.   ,  0.   ,  0.   ,  0.062,  0.938]])
```

```
for p in sm.P:
    print(p)
```

```
[[ 0.963  0.03   0.006  0.     0.   ]
 [ 0.06   0.832  0.107  0.     0.   ]
 [ 0.     0.14   0.74   0.12   0.   ]
 [ 0.     0.036  0.321  0.571  0.071]
 [ 0.     0.     0.     0.167  0.833]]
[[ 0.798  0.168  0.034  0.     0.   ]
 [ 0.075  0.882  0.042  0.     0.   ]
 [ 0.005  0.07   0.866  0.059  0.   ]
 [ 0.     0.     0.064  0.902  0.034]
 [ 0.     0.     0.     0.194  0.806]]
[[ 0.847  0.153  0.     0.     0.   ]
 [ 0.081  0.789  0.129  0.     0.   ]
 [ 0.005  0.098  0.793  0.098  0.005]
 [ 0.     0.     0.094  0.871  0.035]
 [ 0.     0.     0.     0.102  0.898]]
[[ 0.885  0.098  0.     0.016  0.   ]
 [ 0.039  0.814  0.14   0.     0.008]
 [ 0.005  0.094  0.777  0.119  0.005]
 [ 0.     0.023  0.129  0.754  0.094]
 [ 0.     0.     0.     0.097  0.903]]
[[ 0.333  0.667  0.     0.     0.   ]
 [ 0.048  0.774  0.161  0.016  0.   ]
 [ 0.011  0.161  0.747  0.08   0.   ]
 [ 0.     0.01   0.062  0.896  0.031]
 [ 0.     0.     0.     0.024  0.976]]
```

```
sm.S
```

```
array([[ 0.435,  0.264,  0.204,  0.068,  0.029],
       [ 0.134,  0.34 ,  0.252,  0.233,  0.041],
       [ 0.121,  0.211,  0.264,  0.29 ,  0.114],
       [ 0.078,  0.197,  0.254,  0.225,  0.247],
       [ 0.018,  0.2  ,  0.19 ,  0.255,  0.337]])
```

```
for f in sm.F:
    print(f)
```

```
[[   2.298   28.956   46.143   80.81   279.429]
 [  33.865    3.795   22.571   57.238  255.857]
 [  43.602    9.737    4.911   34.667  233.286]
 [  46.629   12.763    6.257   14.616  198.619]
 [  52.629   18.763   12.257    6.      34.103]]
[[   7.468    9.706   25.768   74.531  194.234]
 [  27.767    2.942   24.971   73.735  193.438]
 [  53.575   28.484    3.976   48.763  168.467]
 [  72.036   46.946   18.462    4.284  119.703]
 [  77.179   52.089   23.604    5.143   24.276]]
[[   8.248    6.533   18.388   40.709  112.767]
 [  47.35     4.731   11.854   34.175  106.234]
 [  69.423   24.767    3.795   22.321   94.38 ]
 [  83.723   39.067   14.3      3.447   76.367]
 [  93.523   48.867   24.1      9.8      8.793]]
[[  12.88    13.348   19.834   28.473   55.824]
 [  99.461    5.064   10.545   23.051   49.689]
 [ 117.768   23.037    3.944   15.084   43.579]
 [ 127.898   32.439   14.569    4.448   31.631]
 [ 138.248   42.789   24.919   10.35     4.056]]
[[  56.282    1.5     10.572   27.022  110.543]
 [  82.922    5.009    9.072   25.522  109.043]
 [  97.177   19.531    5.26    21.424  104.946]
 [ 127.141   48.741   33.296    3.918   83.522]
 [ 169.641   91.241   75.796   42.5      2.965]]
```

```
sm.summary()
```

```
--------------------------------------------------------------
                     Spatial Markov Test
--------------------------------------------------------------
Number of classes: 5
Number of transitions: 3840
Number of regimes: 5
Regime names: LAG0, LAG1, LAG2, LAG3, LAG4
--------------------------------------------------------------
   Test                   LR                Chi-2
  Stat.              170.659              200.624
    DOF                   60                   60
p-value                0.000                0.000
--------------------------------------------------------------
P(H0)           C0         C1         C2         C3         C4
     C0      0.915      0.075      0.009      0.001      0.000
     C1      0.066      0.827      0.105      0.001      0.001
     C2      0.005      0.103      0.794      0.095      0.003
     C3      0.000      0.009      0.094      0.849      0.048
     C4      0.000      0.000      0.000      0.062      0.938
--------------------------------------------------------------
P(LAG0)         C0         C1         C2         C3         C4
     C0      0.963      0.030      0.006      0.000      0.000
     C1      0.060      0.832      0.107      0.000      0.000
     C2      0.000      0.140      0.740      0.120      0.000
     C3      0.000      0.036      0.321      0.571      0.071
     C4      0.000      0.000      0.000      0.167      0.833
--------------------------------------------------------------
P(LAG1)         C0         C1         C2         C3         C4
     C0      0.798      0.168      0.034      0.000      0.000
     C1      0.075      0.882      0.042      0.000      0.000
     C2      0.005      0.070      0.866      0.059      0.000
     C3      0.000      0.000      0.064      0.902      0.034
     C4      0.000      0.000      0.000      0.194      0.806
--------------------------------------------------------------
P(LAG2)         C0         C1         C2         C3         C4
     C0      0.847      0.153      0.000      0.000      0.000
     C1      0.081      0.789      0.129      0.000      0.000
     C2      0.005      0.098      0.793      0.098      0.005
     C3      0.000      0.000      0.094      0.871      0.035
     C4      0.000      0.000      0.000      0.102      0.898
--------------------------------------------------------------
P(LAG3)         C0         C1         C2         C3         C4
     C0      0.885      0.098      0.000      0.016      0.000
     C1      0.039      0.814      0.140      0.000      0.008
     C2      0.005      0.094      0.777      0.119      0.005
     C3      0.000      0.023      0.129      0.754      0.094
     C4      0.000      0.000      0.000      0.097      0.903
--------------------------------------------------------------
P(LAG4)         C0         C1         C2         C3         C4
     C0      0.333      0.667      0.000      0.000      0.000
     C1      0.048      0.774      0.161      0.016      0.000
     C2      0.011      0.161      0.747      0.080      0.000
     C3      0.000      0.010      0.062      0.896      0.031
     C4      0.000      0.000      0.000      0.024      0.976
--------------------------------------------------------------
```

Part II

Point Patterns

> `IPYNB`
>
> **NOTE**: some of this material has been ported and adapted from "Lab 9" in Arribas-Bel (2016).

This notebook covers a brief introduction on how to visualize and analyze point patterns. To demonstrate this, we will use a dataset of all the AirBnb listings in the city of Austin (check the Data section for more information about the dataset).

Before anything, let us load up the libraries we will use:

```
%matplotlib inline

import numpy as np
import pandas as pd
import geopandas as gpd
import seaborn as sns
import matplotlib.pyplot as plt
import mplleaflet as mpll
```

Data preparation

Let us first set the paths to the datasets we will be using:

```
# Adjust this to point to the right file in your computer
listings_link = '../data/listings.csv.gz'
```

The core dataset we will use is `listings.csv`, which contains a lot of information about each individual location listed at AirBnb within Austin:

```
lst = pd.read_csv(listings_link)
lst.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5835 entries, 0 to 5834
Data columns (total 92 columns):
id                                    5835 non-null int64
listing_url                           5835 non-null object
scrape_id                             5835 non-null int64
last_scraped                          5835 non-null object
name                                  5835 non-null object
summary                               5373 non-null object
space                                 4475 non-null object
description                           5832 non-null object
experiences_offered                   5835 non-null object
neighborhood_overview                 3572 non-null object
notes                                 2413 non-null object
transit                               3492 non-null object
thumbnail_url                         5542 non-null object
medium_url                            5542 non-null object
picture_url                           5835 non-null object
xl_picture_url                        5542 non-null object
host_id                               5835 non-null int64
host_url                              5835 non-null object
host_name                             5820 non-null object
host_since                            5820 non-null object
host_location                         5810 non-null object
host_about                            3975 non-null object
host_response_time                    4177 non-null object
host_response_rate                    4177 non-null object
host_acceptance_rate                  3850 non-null object
host_is_superhost                     5820 non-null object
host_thumbnail_url                    5820 non-null object
host_picture_url                      5820 non-null object
host_neighbourhood                    4977 non-null object
host_listings_count                   5820 non-null float64
host_total_listings_count             5820 non-null float64
host_verifications                    5835 non-null object
host_has_profile_pic                  5820 non-null object
host_identity_verified                5820 non-null object
street                                5835 non-null object
neighbourhood                         4800 non-null object
neighbourhood_cleansed                5835 non-null int64
neighbourhood_group_cleansed          0 non-null float64
city                                  5835 non-null object
state                                 5835 non-null object
zipcode                               5810 non-null float64
market                                5835 non-null object
smart_location                        5835 non-null object
country_code                          5835 non-null object
country                               5835 non-null object
latitude                              5835 non-null float64
longitude                             5835 non-null float64
is_location_exact                     5835 non-null object
property_type                         5835 non-null object
room_type                             5835 non-null object
accommodates                          5835 non-null int64
bathrooms                             5789 non-null float64
bedrooms                              5829 non-null float64
beds                                  5812 non-null float64
bed_type                              5835 non-null object
amenities                             5835 non-null object
square_feet                           302 non-null float64
price                                 5835 non-null object
weekly_price                          2227 non-null object
monthly_price                         1717 non-null object
security_deposit                      2770 non-null object
cleaning_fee                          3587 non-null object
```

```
guests_included                    5835 non-null int64
extra_people                       5835 non-null object
minimum_nights                     5835 non-null int64
maximum_nights                     5835 non-null int64
calendar_updated                   5835 non-null object
has_availability                   5835 non-null object
availability_30                    5835 non-null int64
availability_60                    5835 non-null int64
availability_90                    5835 non-null int64
availability_365                   5835 non-null int64
calendar_last_scraped              5835 non-null object
number_of_reviews                  5835 non-null int64
first_review                       3827 non-null object
last_review                        3829 non-null object
review_scores_rating               3789 non-null float64
review_scores_accuracy             3776 non-null float64
review_scores_cleanliness          3778 non-null float64
review_scores_checkin              3778 non-null float64
review_scores_communication        3778 non-null float64
review_scores_location             3779 non-null float64
review_scores_value                3778 non-null float64
requires_license                   5835 non-null object
license                            1 non-null float64
jurisdiction_names                 0 non-null float64
instant_bookable                   5835 non-null object
cancellation_policy                5835 non-null object
require_guest_profile_picture      5835 non-null object
require_guest_phone_verification   5835 non-null object
calculated_host_listings_count     5835 non-null int64
reviews_per_month                  3827 non-null float64
dtypes: float64(20), int64(14), object(58)
memory usage: 4.1+ MB
```

It turns out that one record displays a very odd location and, for the sake of the illustration, we will remove it:

```
odd = lst.loc[lst.longitude>-80, ['longitude', 'latitude']]
odd
```

	longitude	latitude
5832	-5.093682	43.214991

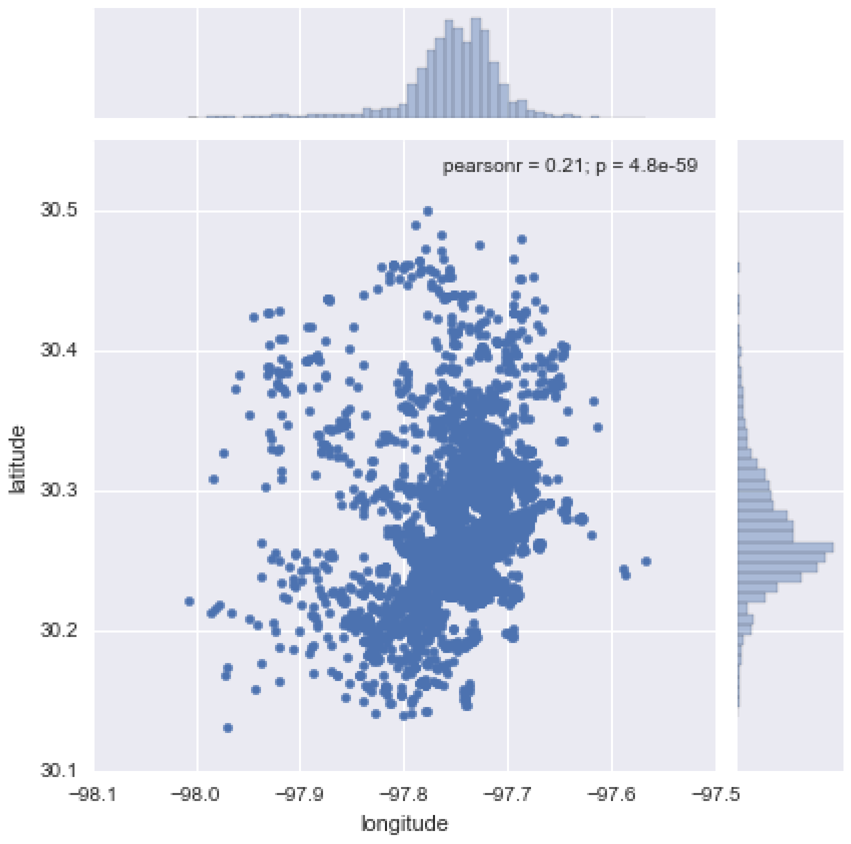
```
lst = lst.drop(odd.index)
```

Point Visualization

The most straighforward way to get a first glimpse of the distribution of the data is to plot their latitude and longitude:

```
sns.jointplot?
```

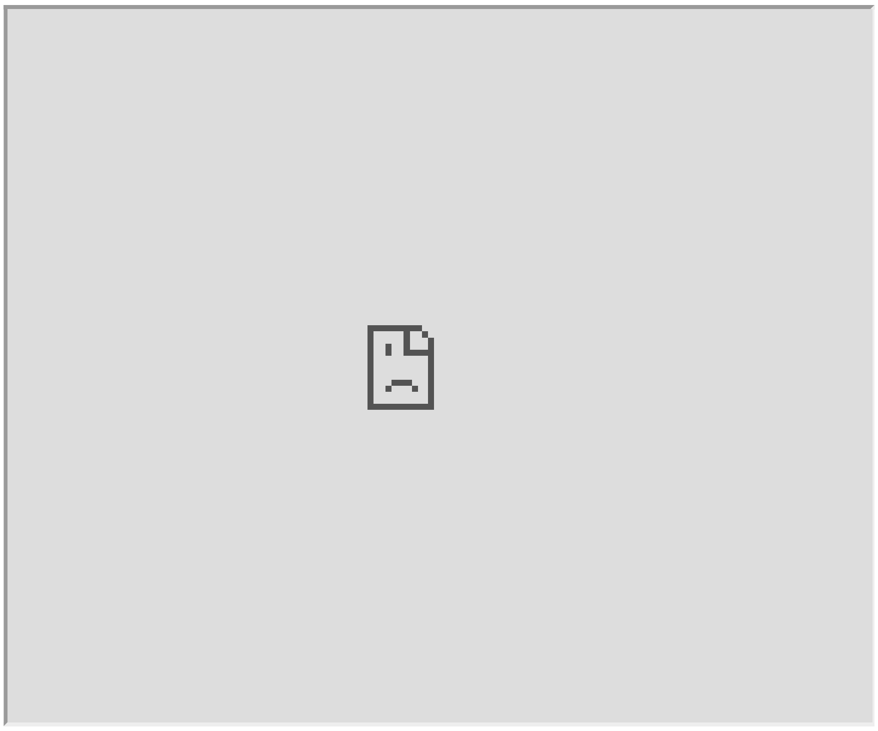
```
sns.jointplot(x="longitude", y="latitude", data=lst);
```



Now this does not neccesarily tell us much about the dataset or the distribution of locations within Austin. There are two main challenges in interpreting the plot: one, there is lack of context, which means the points are not identifiable over space (unless you are so familiar with lon/lat pairs that they have a clear meaning to you); and two, in the center of the plot, there are so many points that it is hard to tell any pattern other than a big blurb of blue.

Let us first focus on the first problem, geographical context. The quickest and easiest way to provide context to this set of points is to overlay a general map. If we had an image with the map or a set of several data sources that we could aggregate to create a map, we could build it from scratch. But in the XXI Century, the easiest is to overlay our point dataset on top of a web map. In this case, we will use Leaflet, and we will convert our underlying `matplotlib` points with `mplleaflet`. The full dataset (+5k observations) is a bit too much for leaflet to plot it directly on screen, so we will obtain a random sample of 100 points:

```
# NOTE: `mpll.display` turned off to be able to compile the website,
#       comment out the last line of this cell for rendering Leaflet map.
rids = np.arange(lst.shape[0])
np.random.shuffle(rids)
f, ax = plt.subplots(1, figsize=(6, 6))
lst.iloc[rids[:100], :].plot(kind='scatter', x='longitude', y='latitude', \
                              s=30, linewidth=0, ax=ax);
mpll.display(fig=f,)
```



This map allows us to get a much better sense of where the points are and what type of location they might be in. For example, now we can see that the big blue blurb has to do with the urbanized core of Austin.

`bokeh` alternative

Leaflet is not the only technology to display data on maps, although it is probably the default option in many cases. When the data is larger than "acceptable", we need to resort to more technically sophisticated alternatives. One option is provided by `bokeh` and its `datashaded` submodule (see here for a very nice introduction to the library, from where this example has been adapted).

Before we delve into `bokeh`, let us reproject our original data (lon/lat coordinates) into Web Mercator, as `bokeh` will expect them. To do that, we turn the coordinates into a `GeoSeries`:

```python
from shapely.geometry import Point
xys_wb = gpd.GeoSeries(lst[['longitude', 'latitude']].apply(Point, axis=1), \
                      crs="+init=epsg:4326")
xys_wb = xys_wb.to_crs(epsg=3857)
x_wb = xys_wb.apply(lambda i: i.x)
y_wb = xys_wb.apply(lambda i: i.y)
```

Now we are ready to setup the plot in `bokeh`:

```
from bokeh.plotting import figure, output_notebook, show
from bokeh.tile_providers import STAMEN_TERRAIN
output_notebook()

minx, miny, maxx, maxy = xys_wb.total_bounds
y_range = miny, maxy
x_range = minx, maxx

def base_plot(tools='pan,wheel_zoom,reset',plot_width=600, plot_height=400, **
    p = figure(tools=tools, plot_width=plot_width, plot_height=plot_height,
        x_range=x_range, y_range=y_range, outline_line_color=None,
        min_border=0, min_border_left=0, min_border_right=0,
        min_border_top=0, min_border_bottom=0, **plot_args)

    p.axis.visible = False
    p.xgrid.grid_line_color = None
    p.ygrid.grid_line_color = None
    return p

options = dict(line_color=None, fill_color='#800080', size=4)
```

Loading BokehJS ...
And good to go for mapping!

```
# NOTE: `show` turned off to be able to compile the website,
#       comment out the last line of this cell for rendering.
p = base_plot()
p.add_tile(STAMEN_TERRAIN)
p.circle(x=x_wb, y=y_wb, **options)
#show(p)
```

```
<bokeh.models.renderers.GlyphRenderer at 0x1052bb5f8>
```

As you can quickly see, `bokeh` is substantially faster at rendering larger amounts of data.

The second problem we have spotted with the first scatter is that, when the number of points grows, at some point it becomes impossible to discern anything other than a big blur of color. To some extent, interactivity gets at that problem by allowing the user to zoom in until every point is an entity on its own. However, there exist techniques that allow to summarize the data to be able to capture the overall pattern at once. Traditionally, kernel density estimation (KDE) has been one of the most common solutions by approximating a continuous surface of point intensity. In this context, however, we will explore a more recent alternative suggested by the `datashader` library (see the paper if interested in more details).

Arguably, our dataset is not large enough to justify the use of a reduction technique like datashader, but we will create the plot for the sake of the illustration. Keep in mind, the usefulness of this approach increases the more points you need to be plotting.

```
# NOTE: `show` turned off to be able to compile the website,
#       comment out the last line of this cell for rendering.

import datashader as ds
from datashader.callbacks import InteractiveImage
from datashader.colors import viridis
from datashader import transfer_functions as tf
from bokeh.tile_providers import STAMEN_TONER

p = base_plot()
p.add_tile(STAMEN_TONER)

pts = pd.DataFrame({'x': x_wb, 'y': y_wb})
pts['count'] = 1
def create_image90(x_range, y_range, w, h):
    cvs = ds.Canvas(plot_width=w, plot_height=h, x_range=x_range, y_range=y_ra
    agg = cvs.points(pts, 'x', 'y',  ds.count('count'))
    img = tf.interpolate(agg.where(agg > np.percentile(agg,90)), \
                         cmap=viridis, how='eq_hist')
    return tf.dynspread(img, threshold=0.1, max_px=4)

#InteractiveImage(p, create_image90)
```

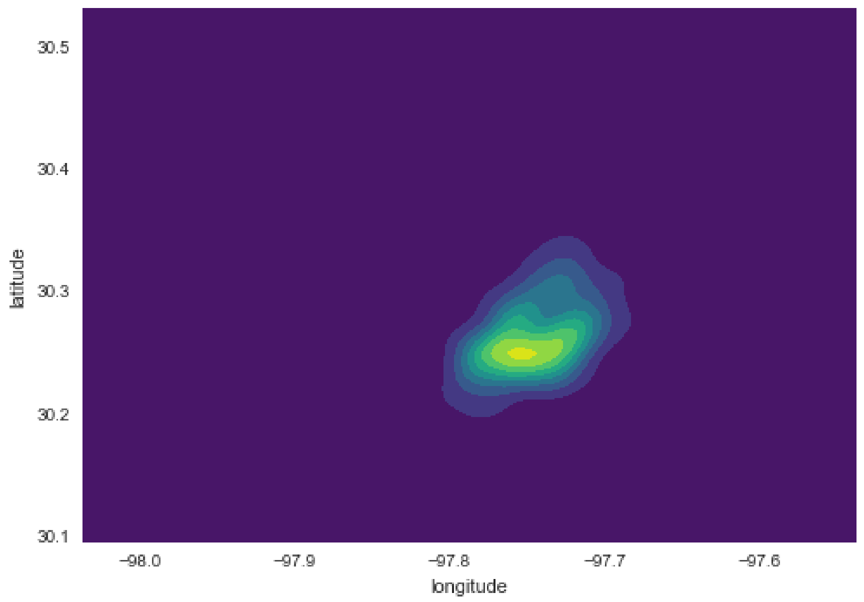
The key advandage of `datashader` is that is decouples the point processing from the plotting. That is the bit that allows it to be scalable to truly large datasets (e.g. millions of points). Essentially, the approach is based on generating a very fine grid, counting points within pixels, and encoding the count into a color scheme. In our map, this is not particularly effective because we do not have too many points (the previous plot is probably a more effective one) and esssentially there is a pixel per location of every point. However, hopefully this example shows how to create this kind of scalable maps.

Kernel Density Estimation

A common alternative when the number of points grows is to replace plotting every single point by estimating the continuous observed probability distribution. In this case, we will not be visualizing the points themselves, but an abstracted surface that models the probability of point density over space. The most commonly used method to do this is the so called kernel density estimate (KDE). The idea behind KDEs is to count the number of points in a continious way. Instead of using discrete counting, where you include a point in the count if it is inside a certain boundary and ignore it otherwise, KDEs use functions (kernels) that include points but give different weights to each one depending of how far of the location where we are counting the point is.

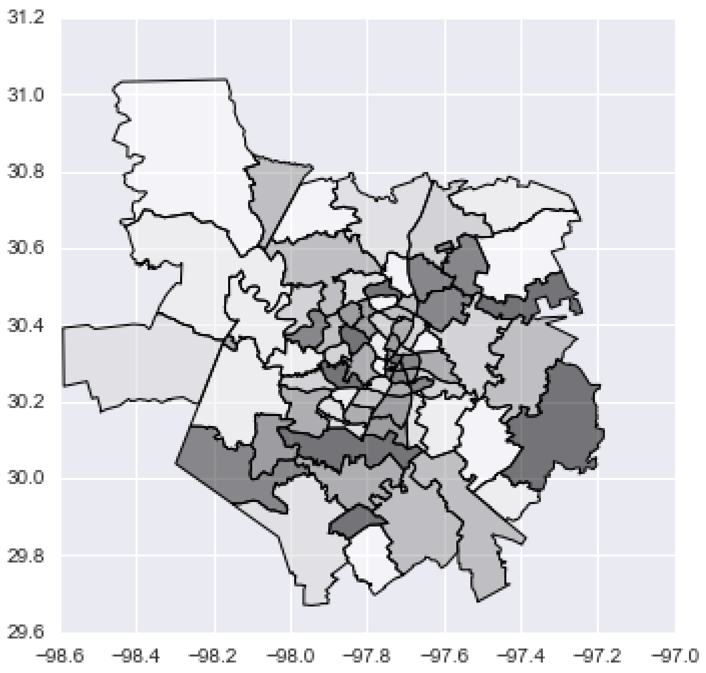
Creating a KDE is very straightfoward in Python. In its simplest form, we can run the following single line of code:

```
sns.kdeplot(lst['longitude'], lst['latitude'], shade=True, cmap='viridis');
```



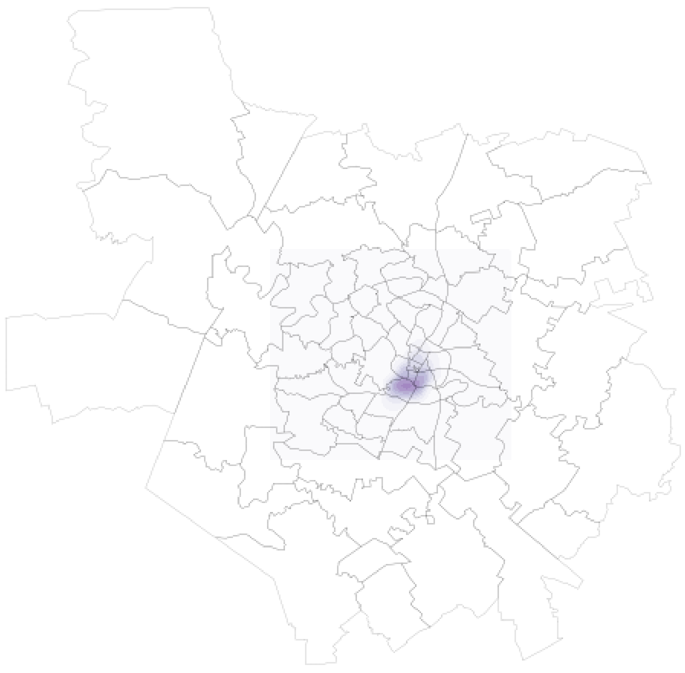
Now, if we want to include additional layers of data to provide context, we can do so in the same way we would layer up different elements in `matplotlib`. Let us load first the Zip codes in Austin, for example:

```
zc = gpd.read_file('../data/Zipcodes.geojson')
zc.plot();
```



And, to overlay both layers:

```
f, ax = plt.subplots(1, figsize=(9, 9))

zc.plot(color='white', linewidth=0.1, ax=ax)

sns.kdeplot(lst['longitude'], lst['latitude'], \
            shade=True, cmap='Purples', \
            ax=ax);

ax.set_axis_off()
plt.axis('equal')
plt.show()
```



Exercise

> *Split the dataset by type of property and create a map for the five most common types.*

Consider the following sorting of property types:

```
lst.property_type.groupby(lst.property_type)\
                 .count()\
                 .sort_values(ascending=False)
```

```
property_type
House              3549
Apartment          1855
Condominium         106
Loft                 83
Townhouse            57
Other                47
Bed & Breakfast      37
Camper/RV            34
Bungalow             18
Cabin                17
Tent                 11
Villa                 7
Treehouse             7
Earth House           2
Chalet                1
Hut                   1
Boat                  1
Tipi                  1
Name: property_type, dtype: int64
```

Spatial Clustering

> `IPYNB`
>
> **NOTE**: much of this material has been ported and adapted from "Lab 8" in Arribas-Bel (2016).

This notebook covers a brief introduction to spatial regression. To demonstrate this, we will use a dataset of all the AirBnb listings in the city of Austin (check the Data section for more information about the dataset).

Many questions and topics are complex phenomena that involve several dimensions and are hard to summarize into a single variable. In statistical terms, we call this family of problems *multivariate*, as oposed to *univariate* cases where only a single variable is considered in the analysis. Clustering tackles this kind of questions by reducing their dimensionality -the number of relevant variables the analyst needs to look at- and converting it into a more intuitive set of classes that even non-technical audiences can look at and make sense of. For this reason, it is widely use in applied contexts such as policymaking or marketing. In addition, since these methods do not require many preliminar assumptions about the structure of the data, it is a commonly used exploratory tool, as it can quickly give clues about the shape, form and content of a dataset.

The core idea of statistical clustering is to summarize the information contained in several variables by creating a relatively small number of categories. Each observation in the dataset is then assigned to one, and only one, category depending on its values for the variables originally considered in the classification. If done correctly, the exercise reduces the complexity of a multi-dimensional problem while retaining all the meaningful information contained in the original dataset. This is because, once classified, the analyst only needs to look at in which category every observation falls into, instead of considering the multiple values associated with each of the variables and trying to figure out how to put them together in a coherent sense. When the clustering is performed on observations that represent areas, the technique is often called geodemographic analysis.

The basic premise of the exercises we will be doing in this notebook is that, through the characteristics of the houses listed in AirBnb, we can learn about the geography of Austin. In particular, we will try to classify the city's zipcodes into a small number of groups that will allow us to extract some patterns about the main kinds of houses and areas in the city.

Data

Before anything, let us load up the libraries we will use:

```
%matplotlib inline

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pysal as ps
import geopandas as gpd
from sklearn import cluster
from sklearn.preprocessing import scale

sns.set(style="whitegrid")
```

Let us also set the paths to all the files we will need throughout the tutorial:

```
# Adjust this to point to the right file in your computer
abb_link = '../data/listings.csv.gz'
zc_link = '../data/Zipcodes.geojson'
```

Before anything, let us load the main dataset:

```
lst = pd.read_csv(abb_link)
```

Originally, this is provided at the individual level. Since we will be working in terms of neighborhoods and areas, we will need to aggregate them to that level. For this illustration, we will be using the following subset of variables:

```
varis = ['bedrooms', 'bathrooms', 'beds']
```

This will allow us to capture the main elements that describe the "look and feel" of a property and, by aggregation, of an area or neighborhood. All of the variables above are numerical values, so a sensible way to aggregate them is by obtaining the average (of bedrooms, etc.) per zipcode.

```
aves = lst.groupby('zipcode')[varis].mean()
aves.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 47 entries, 33558.0 to 78759.0
Data columns (total 3 columns):
bedrooms     47 non-null float64
bathrooms    47 non-null float64
beds         47 non-null float64
dtypes: float64(3)
memory usage: 1.5 KB
```

In addition to these variables, it would be good to include also a sense of what proportions of different types of houses each zipcode has. For example, one can imagine that neighborhoods with a higher proportion of condos than single-family homes will probably look and feel more urban. To do this, we need to do some data munging:

```
types = pd.get_dummies(lst['property_type'])
prop_types = types.join(lst['zipcode'])\
                  .groupby('zipcode')\
                  .sum()
prop_types_pct = (prop_types * 100.).div(prop_types.sum(axis=1), axis=0)
prop_types_pct.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 47 entries, 33558.0 to 78759.0
Data columns (total 18 columns):
Apartment          47 non-null float64
Bed & Breakfast    47 non-null float64
Boat               47 non-null float64
Bungalow           47 non-null float64
Cabin              47 non-null float64
Camper/RV          47 non-null float64
Chalet             47 non-null float64
Condominium        47 non-null float64
Earth House        47 non-null float64
House              47 non-null float64
Hut                47 non-null float64
Loft               47 non-null float64
Other              47 non-null float64
Tent               47 non-null float64
Tipi               47 non-null float64
Townhouse          47 non-null float64
Treehouse          47 non-null float64
Villa              47 non-null float64
dtypes: float64(18)
memory usage: 7.0 KB
```

Now we bring both sets of variables together:

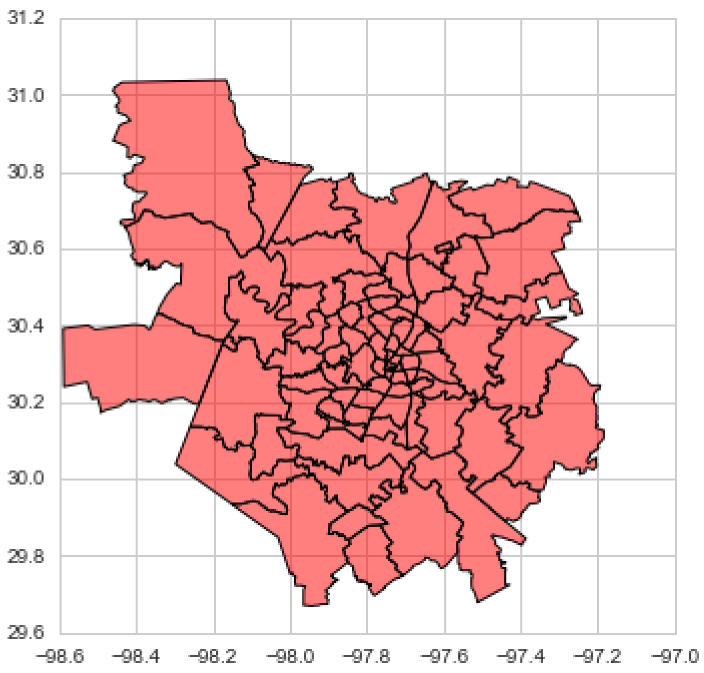
```
aves_props = aves.join(prop_types_pct)
```

And since we will be feeding this into the clustering algorithm, we will first standardize the columns:

```
db = pd.DataFrame(\
                 scale(aves_props), \
                 index=aves_props.index, \
                 columns=aves_props.columns)\
       .rename(lambda x: str(int(x)))
```

Now let us bring geography in:

```
zc = gpd.read_file(zc_link)
zc.plot(color='red');
```

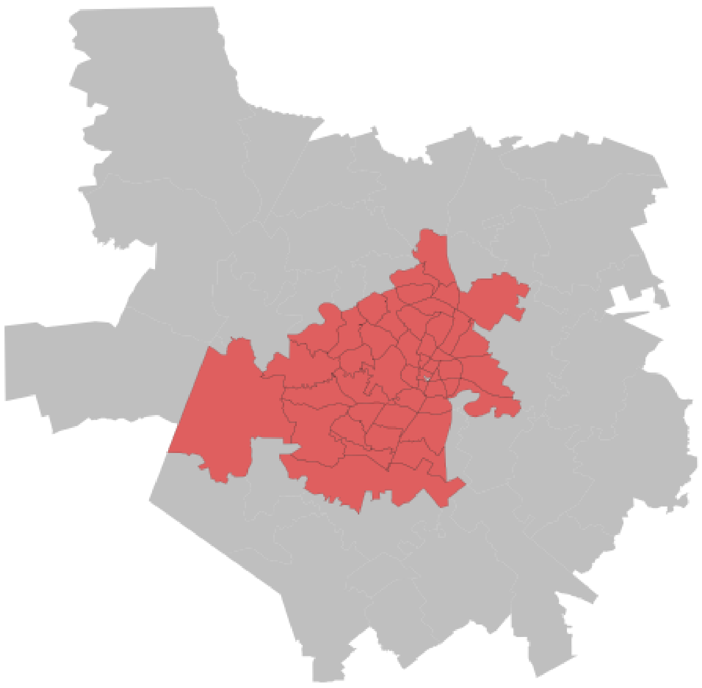


And combine the two:

```
zdb = zc[['geometry', 'zipcode', 'name']].join(db, on='zipcode')\
                                          .dropna()
```

To get a sense of which areas we have lost:

```
f, ax = plt.subplots(1, figsize=(9, 9))

zc.plot(color='grey', linewidth=0, ax=ax)
zdb.plot(color='red', linewidth=0.1, ax=ax)

ax.set_axis_off()

plt.show()
```



Geodemographic analysis

The main intuition behind geodemographic analysis is to group disparate areas of a city or region into a small set of classes that capture several characteristics shared by those in the same group. By doing this, we can get a new perspective not only on the types of areas in a city, but on how they are distributed over space. In the context of our AirBnb data analysis, the idea is that we can group different zipcodes of Austin based on the type of houses listed on the website. This will give us a hint into the geography of AirBnb in the Texan tech capital.

Although there exist many techniques to statistically group observations in a dataset, all of them are based on the premise of using a set of attributes to define classes or categories of observations that are similar *within* each of them, but differ *between* groups. How similarity within groups and dissimilarity between them is defined and how the classification algorithm is operationalized is what makes techniques differ and also what makes each of them particularly well suited for specific problems or types of data. As an illustration, we will only dip our toes into one of these methods, K-means, which is probably the most commonly used technique for statistical clustering.

Technically speaking, we describe the method and the parameters on the following line of code, where we specifically ask for five groups:

```
cluster.KMeans?
```

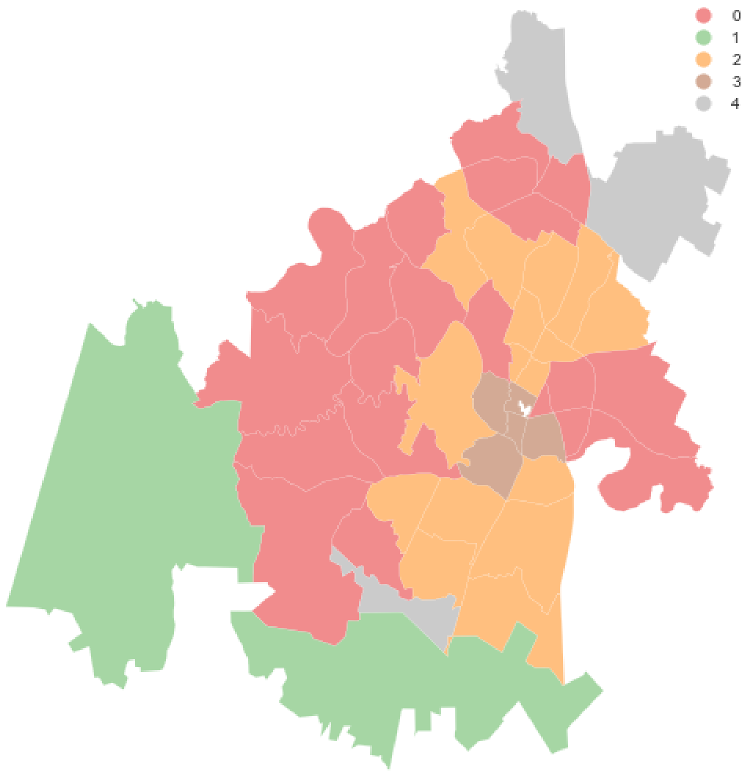
```
km5 = cluster.KMeans(n_clusters=5)
```

Following the `sklearn` pipeline approach, all the heavy-lifting of the clustering happens when we `fit` the model to the data:

```
km5cls = km5.fit(zdb.drop(['geometry', 'name'], axis=1).values)
```

Now we can extract the classes and put them on a map:

```
f, ax = plt.subplots(1, figsize=(9, 9))

zdb.assign(cl=km5cls.labels_)\
   .plot(column='cl', categorical=True, legend=True, \
         linewidth=0.1, edgecolor='white', ax=ax)

ax.set_axis_off()

plt.show()
```

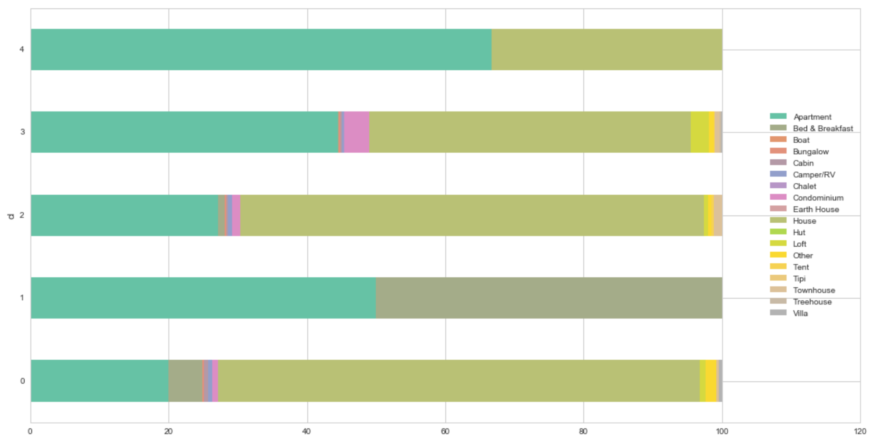


The map above shows a clear pattern: there is a class at the core of the city (number 0, in red), then two other ones in a sort of "urban ring" (number 1 and 3, in green and brown, respectively), and two peripheral sets of areas (number 2 and 4, yellow and green).

This gives us a good insight into the geographical structure, but does not tell us much about what are the defining elements of these groups. To do that, we can have a peak into the characteristics of the classes. For example, let us look at how the proportion of different types of properties are distributed across clusters:

```
cl_pcts = prop_types_pct.rename(lambda x: str(int(x)))\
                        .reindex(zdb['zipcode'])\
                        .assign(cl=km5cls.labels_)\
                        .groupby('cl')\
                        .mean()
```

```
f, ax = plt.subplots(1, figsize=(18, 9))
cl_pcts.plot(kind='barh', stacked=True, ax=ax, \
             cmap='Set2', linewidth=0)
ax.legend(ncol=1, loc="right");
```



A few interesting, albeit maybe not completely unsurprising, characteristics stand out. First, most of the locations we have in the dataset are either apartments or houses. However, how they are distributed is interesting. The urban core -cluster 0- distinctively has the highest proportion of condos and lofts. The suburban ring -clusters 1 and 3- is very consistent, with a large share of houses and less apartments, particularly so in the case of cluster 3. Class 4 has only two types of properties, houses and apartments, suggesting there are not that many places listed at AirBnb. Finally, class 3 arises as a more rural and leisure one: beyond apartments, it has a large share of bed & breakfasts.

Mini Exercise

> *What are the average number of beds, bedrooms and bathrooms for every class?*

Regionalization analysis: building (meaningful) regions

In the case of analysing spatial data, there is a subset of methods that are of particular interest for many common cases in Geographic Data Science. These are the so-called regionalization techniques. Regionalization methods can take also many forms and faces but, at their core, they all involve statistical clustering of observations with the additional constraint that observations need to be geographical neighbors to be in the same category. Because of this, rather than category, we will use the term area for each observation and region for each class or cluster -hence regionalization, the construction of regions from smaller areas.

As in the non-spatial case, there are many different algorithms to perform regionalization, and they all differ on details relating to the way they measure (dis)similarity, the process to regionalize, etc. However, same as above too, they all share a few common aspects. In particular, they all take a set of input attributes *and* a representation of space in the form of a binary spatial weights matrix. Depending on the algorithm, they also require the desired number of output regions into which the areas are aggregated.

In this example, we are going to create aggregations of zipcodes into groups that have areas where the AirBnb listed location have similar ratings. In other words, we will create delineations for the "quality" or "satisfaction" of AirBnb users. In other words, we will explore what are the boundaries that separate areas where AirBnb users tend to be satisfied about their experience versus those where the ratings are not as high. To do this, we will focus on the `review_scores_X` set of variables in the original dataset:

```
ratings = [i for i in lst if 'review_scores_' in i]
ratings
```

```
['review_scores_rating',
 'review_scores_accuracy',
 'review_scores_cleanliness',
 'review_scores_checkin',
 'review_scores_communication',
 'review_scores_location',
 'review_scores_value']
```

Similarly to the case above, we now bring this at the zipcode level. Note that, since they are all scores that range from 0 to 100, we can use averages and we do not need to standardize.

```
rt_av = lst.groupby('zipcode')[ratings]\
           .mean()\
           .rename(lambda x: str(int(x)))
```

And we link these to the geometries of zipcodes:

```
zrt = zc[['geometry', 'zipcode']].join(rt_av, on='zipcode')\
                                 .dropna()
zrt.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Int64Index: 43 entries, 0 to 78
Data columns (total 9 columns):
geometry                       43 non-null object
zipcode                        43 non-null object
review_scores_rating           43 non-null float64
review_scores_accuracy         43 non-null float64
review_scores_cleanliness      43 non-null float64
review_scores_checkin          43 non-null float64
review_scores_communication    43 non-null float64
review_scores_location         43 non-null float64
review_scores_value            43 non-null float64
dtypes: float64(7), object(2)
memory usage: 3.4+ KB
```

In contrast to the standard clustering techniques, regionalization requires a formal representation of topology. This is so the algorithm can impose spatial constraints during the process of clustering the observations. We will use exactly the same approach as in the previous sections of this tutorial for this and build spatial weights objects `W` with `PySAL`. For the sake of this illustration, we will consider queen contiguity, but any other rule should work fine as long as there is a rational behind it. Weights constructors currently only work from shapefiles on disk, so we will write our `GeoDataFrame` first, then create the `W` object, and remove the files.

```
zrt.to_file('tmp')
w = ps.queen_from_shapefile('tmp/tmp.shp', idVariable='zipcode')
# NOTE: this might not work on Windows
! rm -r tmp
w
```

```
<pysal.weights.weights.W at 0x11bd5ff98>
```

Now we are ready to run the regionalization algorithm. In this case we will use the `max-p` (Duque, Anselin & Rey, 2012), which does not require a predefined number of output regions but instead it takes a target variable that you want to make sure a minimum threshold is met. In our case, since it is based on ratings, we will impose that every resulting region has at least 10% of the total number of reviews. Let us work through what that would mean:

```
n_rev = lst.groupby('zipcode')\
           .sum()\
           ['number_of_reviews']\
           .rename(lambda x: str(int(x)))\
           .reindex(zrt['zipcode'])
thr = np.round(0.1 * n_rev.sum())
thr
```

```
6271.0
```

This means we want every resulting region to be based on at least 6,271 reviews. Now we have all the pieces, let us glue them together through the algorithm:

```
# Set the seed for reproducibility
np.random.seed(1234)

z = zrt.drop(['geometry', 'zipcode'], axis=1).values
maxp = ps.region.Maxp(w, z, thr, n_rev.values[:, None], initial=1000)
```

We can check whether the solution is better (lower within sum of squares) than we would have gotten from a purely random regionalization process using the `cinference` method:

```
%%time
np.random.seed(1234)
maxp.cinference(nperm=999)
```

```
CPU times: user 26.2 s, sys: 185 ms, total: 26.4 s
Wall time: 32.1 s
```

Which allows us to obtain an empirical p-value:

```
maxp.cpvalue
```

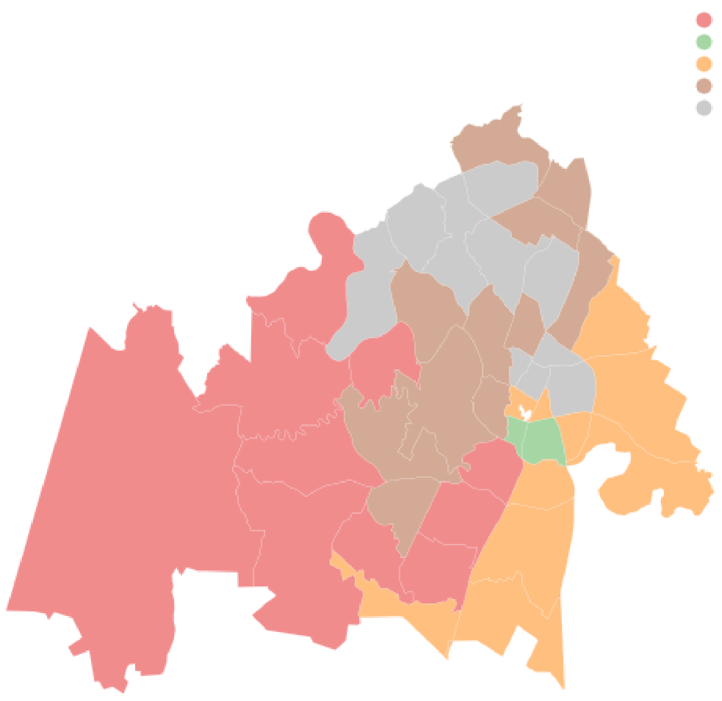
```
0.022
```

Which gives us reasonably good confidence that the solution we obtain is more meaningful than pure chance.

With that out of the way, let us see what the result looks like on a map! First we extract the labels:

```
lbls = pd.Series(maxp.area2region).reindex(zrt['zipcode'])
```

```
f, ax = plt.subplots(1, figsize=(9, 9))

zrt.assign(cl=lbls.values)\
   .plot(column='cl', categorical=True, legend=True, \
         linewidth=0.1, edgecolor='white', ax=ax)

ax.set_axis_off()

plt.show()
```



The map shows a clear geographical pattern with a western area, another in the North and a smaller one in the East. Let us unpack what each of them is made of:

```
zrt[ratings].groupby(lbls.values).mean().T
```

	0	**1**	**2**	
review_scores_rating	96.911817	95.326614	92.502135	9
review_scores_accuracy	9.767500	9.605032	9.548751	9
review_scores_cleanliness	9.678277	9.558179	8.985408	9
review_scores_checkin	9.922450	9.797086	9.765563	9
review_scores_communication	9.932211	9.827390	9.794794	9
review_scores_location	9.644754	9.548761	8.904775	9
review_scores_value	9.678822	9.341224	9.491638	9

Although very similar, there are some patterns to be extracted. For example, the East area seems to have lower overall scores.

Exercise

> *Obtain a geodemographic classification with eight classes instead of five and replicate the analysis above*
>
> *Re-run the regionalization exercise imposing a minimum of 5% reviews per area*

Spatial Regression

> IPYNB
>
> **NOTE**: some of this material has been ported and adapted from the Spatial Econometrics note in Arribas-Bel (2016b).

This notebook covers a brief and gentle introduction to spatial econometrics in Python. To do that, we will use a set of Austin properties listed in AirBnb.

The core idea of spatial econometrics is to introduce a formal representation of space into the statistical framework for regression. This can be done in many ways: by including predictors based on space (e.g. distance to relevant features), by splitting the datasets into subsets that map into different geographical regions (e.g. spatial regimes), by exploiting close distance to other observations to borrow information in the estimation (e.g. kriging), or by introducing variables that put in relation their value at a given location with those in nearby locations, to give a few examples. Some of these approaches can be implemented with standard non-spatial techniques, while others require bespoke models that can deal with the issues introduced. In this short tutorial, we will focus on the latter group. In particular, we will introduce some of the most commonly used methods in the field of spatial econometrics.

The example we will use to demonstrate this draws on hedonic house price modelling. This a well-established methodology that was developed by Rosen (1974) that is capable of recovering the marginal willingness to pay for goods or services that are not traded in the market. In other words, this allows us to put an implicit price on things such as living close to a park or in a neighborhood with good quality of air. In addition, since hedonic models are based on linear regression, the technique can also be used to obtain predictions of house prices.

Data

Before anything, let us load up the libraries we will use:

```
%matplotlib inline

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pysal as ps
import geopandas as gpd

sns.set(style="whitegrid")
```

Let us also set the paths to all the files we will need throughout the tutorial, which is only the original table of listings:

```
# Adjust this to point to the right file in your computer
abb_link = '../data/listings.csv.gz'
```

And go ahead and load it up too:

```
lst = pd.read_csv(abb_link)
```

Baseline (nonspatial) regression

Before introducing explicitly spatial methods, we will run a simple linear regression model. This will allow us, on the one hand, set the main principles of hedonic modeling and how to interpret the coefficients, which is good because the spatial models will build on this; and, on the other hand, it will provide a baseline model that we can use to evaluate how meaningful the spatial extensions are.

Essentially, the core of a linear regression is to explain a given variable -the price of a listing i on AirBnb (P_i)- as a linear function of a set of other characteristics we will collectively call X_i:

$$\ln(P_i) = \alpha + \beta X_i + \epsilon_i$$

For several reasons, it is common practice to introduce the price in logarithms, so we will do so here. Additionally, since this is a probabilistic model, we add an error term ϵ_i that is assumed to be well-behaved (i.i.d. as a normal).

For our example, we will consider the following set of explanatory features of each listed property:

```
x = ['host_listings_count', 'bathrooms', 'bedrooms', 'beds', 'guests_included'
```

Additionally, we are going to derive a new feature of a listing from the `amenities` variable. Let us construct a variable that takes 1 if the listed property has a pool and 0 otherwise:

```
def has_pool(a):
    if 'Pool' in a:
        return 1
    else:
        return 0

lst['pool'] = lst['amenities'].apply(has_pool)
```

For convenience, we will re-package the variables:

```
yxs = lst.loc[:, x + ['pool', 'price']].dropna()
y = np.log(\
           yxs['price'].apply(lambda x: float(x.strip('$').replace(',', '')))\
           + 0.000001
          )
```

To run the model, we can use the `spreg` module in `PySAL`, which implements a standard OLS routine, but is particularly well suited for regressions on spatial data. Also, although for the initial model we do not need it, let us build a spatial weights matrix that connects every observation to its 8 nearest neighbors. This will allow us to get extra diagnostics from the baseline model.

```
w = ps.knnW_from_array(lst.loc[\
                               yxs.index, \
                              ['longitude', 'latitude']\
                              ].values)
w.transform = 'R'
w
```

```
<pysal.weights.weights.W at 0x11bdb5358>
```

At this point, we are ready to fit the regression:

```
m1 = ps.spreg.OLS(y.values[:, None], yxs.drop('price', axis=1).values, \
                  w=w, spat_diag=True, \
                  name_x=yxs.drop('price', axis=1).columns.tolist(), name_y='l
```

To get a quick glimpse of the results, we can print its summary:

```
print(m1.summary)
```

```
REGRESSION
----------
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES
-----------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :   ln(price)                Number of Observations:
Mean dependent var  :      5.1952                Number of Variables   :
S.D. dependent var  :      0.9455                Degrees of Freedom    :
R-squared           :      0.4042
Adjusted R-squared  :      0.4036
Sum squared residual:    3071.189                F-statistic           :    65
Sigma-square        :       0.533                Prob(F-statistic)     :
S.E. of regression  :       0.730                Log likelihood        :   -63
Sigma-square ML     :       0.533                Akaike info criterion :   127
S.E of regression ML:      0.7298                Schwarz criterion     :   127

-------------------------------------------------------------------------------
            Variable     Coefficient       Std.Error     t-Statistic     Proba
-------------------------------------------------------------------------------
            CONSTANT       4.0976886       0.0223530     183.3171506       0.0
 host_listings_count      -0.0000130       0.0001790      -0.0726772       0.9
           bathrooms       0.2947079       0.0194817      15.1273879       0.0
            bedrooms       0.3274226       0.0159666      20.5067654       0.0
                beds       0.0245741       0.0097379       2.5235601       0.0
     guests_included       0.0075119       0.0060551       1.2406028       0.2
                pool       0.0888039       0.0221903       4.0019209       0.0
-------------------------------------------------------------------------------

REGRESSION DIAGNOSTICS
MULTICOLLINEARITY CONDITION NUMBER            9.260

TEST ON NORMALITY OF ERRORS
TEST                             DF        VALUE           PROB
Jarque-Bera                       2     1358479.047           0.0000

DIAGNOSTICS FOR HETEROSKEDASTICITY
RANDOM COEFFICIENTS
TEST                             DF        VALUE           PROB
Breusch-Pagan test                6        1414.297           0.0000
Koenker-Bassett test              6          36.756           0.0000

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                           MI/DF       VALUE           PROB
Lagrange Multiplier (lag)         1         255.796           0.0000
Robust LM (lag)                   1          13.039           0.0003
Lagrange Multiplier (error)       1         278.752           0.0000
Robust LM (error)                 1          35.995           0.0000
Lagrange Multiplier (SARMA)       2         291.791           0.0000

================================ END OF REPORT ================================
```

Results are largely unsurprising, but nonetheless reassuring. Both an extra bedroom and an extra bathroom increase the final price around 30%. Accounting for those, an extra bed pushes the price about 2%. Neither the number of guests included nor the number of listings the host has in total have a significant effect on the final price.

Including a spatial weights object in the regression buys you an extra bit: the summary provides results on the diagnostics for spatial dependence. These are a series of statistics that test whether the residuals of the regression are spatially correlated, against the null of a random distribution over space. If the latter is rejected a key assumption of OLS, independently distributed error terms, is violated. Depending on the structure of the spatial pattern, different strategies have been defined within the spatial econometrics literature to deal with them. If you are interested in this, a very recent and good resource to check out is Anselin & Rey (2015). The main summary from the diagnostics for spatial dependence is that there is clear evidence to reject the null of spatial randomness in the residuals, hence an explicitly spatial approach is warranted.

Spatially lagged exogenous regressors (`WX`)

The first and most straightforward way to introduce space is by "spatially lagging" one of the explanatory variables. Mathematically, this can be expressed as follows:

$$\ln(P_i) = \alpha + \beta X_i + \delta \sum_j w_{ij} X'_i + \epsilon_i$$

where $X'*i$ is a subset of X_i, although it could encompass all of the explanatory variables, and $w*{ij}$ is the ij-th cell of a spatial weights matrix W. Because W assigns non-zero values only to spatial neighbors, if W is row-standardized (customary in this context), then $\sum*j w*{ij} X'_i$ captures the average value of X'_i in the surroundings of location i. This is what we call the *spatial lag* of X_i. Also, since it is a spatial transformation of an explanatory variable, the standard estimation approach - OLS- is sufficient: spatially lagging the variables does not violate any of the assumptions on which OLS relies.

Usually, we will want to spatially lag variables that we think may affect the price of a house in a given location. For example, one could think that pools represent a visual amenity. If that is the case, then listed properties surrounded by other properties with pools might, everything else equal, be more expensive. To calculate the number of pools surrounding each property, we can build an alternative weights matrix that we do not row-standardize:

```
w_pool = ps.knnW_from_array(lst.loc[\
                                yxs.index, \
                               ['longitude', 'latitude']\
                               ].values)
yxs_w = yxs.assign(w_pool=ps.lag_spatial(w_pool, yxs['pool'].values))
```

And now we can run the model, which has the same setup as `m1`, with the exception that it includes the number of AirBnb properties with pools surrounding each house:

```
m2 = ps.spreg.OLS(y.values[:, None], \
                  yxs_w.drop('price', axis=1).values, \
                  w=w, spat_diag=True, \
                  name_x=yxs_w.drop('price', axis=1).columns.tolist(), name_y=
```

```
print(m2.summary)
```

```
REGRESSION
----------
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES
-----------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :   ln(price)               Number of Observations:
Mean dependent var  :      5.1952               Number of Variables   :
S.D. dependent var  :      0.9455               Degrees of Freedom    :
R-squared           :      0.4044
Adjusted R-squared  :      0.4037
Sum squared residual:    3070.363               F-statistic           :    55
Sigma-square        :       0.533               Prob(F-statistic)     :
S.E. of regression  :       0.730               Log likelihood        :   -63
Sigma-square ML     :       0.532               Akaike info criterion :   127
S.E of regression ML:      0.7297               Schwarz criterion     :   128

------------------------------------------------------------------------------
            Variable     Coefficient       Std.Error     t-Statistic     Proba
------------------------------------------------------------------------------
            CONSTANT       4.0906444       0.0230571     177.4134022       0.0
 host_listings_count      -0.0000108       0.0001790      -0.0603617       0.9
           bathrooms       0.2948787       0.0194813      15.1365024       0.0
            bedrooms       0.3277450       0.0159679      20.5252404       0.0
                beds       0.0246650       0.0097377       2.5329419       0.0
     guests_included       0.0076894       0.0060564       1.2696250       0.2
                pool       0.0725756       0.0257356       2.8200486       0.0
              w_pool       0.0188875       0.0151729       1.2448141       0.2
------------------------------------------------------------------------------

REGRESSION DIAGNOSTICS
MULTICOLLINEARITY CONDITION NUMBER            9.605

TEST ON NORMALITY OF ERRORS
TEST                             DF        VALUE           PROB
Jarque-Bera                       2     1368880.320           0.0000

DIAGNOSTICS FOR HETEROSKEDASTICITY
RANDOM COEFFICIENTS
TEST                             DF        VALUE           PROB
Breusch-Pagan test                7        1565.566           0.0000
Koenker-Bassett test              7          40.537           0.0000

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                           MI/DF       VALUE           PROB
Lagrange Multiplier (lag)         1         255.124           0.0000
Robust LM (lag)                   1          13.448           0.0002
Lagrange Multiplier (error)       1         276.862           0.0000
Robust LM (error)                 1          35.187           0.0000
Lagrange Multiplier (SARMA)       2         290.310           0.0000

================================ END OF REPORT ================================
```

Results are largely consistent with the original model. Also, incidentally, the number of pools surrounding a property does not appear to have any significant effect on the price of a given property. This could be for a host of reasons: maybe AirBnb customers do not value the number of pools surrounding a property where they are looking to stay; but maybe they do but our dataset only allows us to capture the number of pools in *other* AirBnb properties, which is not necessarily a good proxy of the number of pools in the immediate surroundings of a given property.

Spatially lagged endogenous regressors (`WY`)

In a similar way to how we have included the spatial lag, one could think the prices of houses surrounding a given property also enter its own price function. In math terms, this implies the following:

$$\ln(P_i) = \alpha + \lambda \sum_j w_{ij} \ln(P_i) + \beta X_i + \epsilon_i$$

This is essentially what we call a *spatial lag* model in spatial econometrics. Two calls for caution:

1. Unlike before, this specification *does* violate some of the assumptions on which OLS relies. In particular, it is including an endogenous variable on the right-hand side. This means we need a new estimation method to obtain reliable coefficients. The technical details of this go well beyond the scope of this workshop (although, if you are interested, go check Anselin & Rey, 2015). But we can offload those to `PySAL` and use the `GM_Lag` class, which implements the state-of-the-art approach to estimate this model.
2. A more conceptual *gotcha*: you might be tempted to read the equation above as the effect of the price in neighboring locations j on that of location i. This is not exactly the exact interpretation. Instead, we need to realize this is all assumed to be a "joint decission": rather than some houses setting their price first and that having a subsequent effect on others, what the equation models is an interdependent process by which each owner sets her own price *taking into account* the price that will be set in neighboring locations. This might read a bit like a technical subtlety and, to some extent, it is; but it is important to keep it in mind when you are interpreting the results.

Let us see how you would run this using `PySAL` :

```
m3 = ps.spreg.GM_Lag(y.values[:, None], yxs.drop('price', axis=1).values, \
                  w=w, spat_diag=True, \
                  name_x=yxs.drop('price', axis=1).columns.tolist(), name_y='l
```

```
print(m3.summary)
```

```
REGRESSION
----------
SUMMARY OF OUTPUT: SPATIAL TWO STAGE LEAST SQUARES
--------------------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :   ln(price)               Number of Observations:
Mean dependent var  :      5.1952               Number of Variables   :
S.D. dependent var  :      0.9455               Degrees of Freedom    :
Pseudo R-squared    :      0.4224
Spatial Pseudo R-squared:  0.4056

------------------------------------------------------------------------------
            Variable     Coefficient       Std.Error     z-Statistic     Proba
------------------------------------------------------------------------------
            CONSTANT       3.7085715       0.1075621      34.4784213       0.0
 host_listings_count      -0.0000587       0.0001765      -0.3324585       0.7
           bathrooms       0.2857932       0.0193237      14.7897969       0.0
            bedrooms       0.3272598       0.0157132      20.8270544       0.0
                beds       0.0239548       0.0095848       2.4992528       0.0
     guests_included       0.0065147       0.0059651       1.0921407       0.2
                pool       0.0891100       0.0218383       4.0804521       0.0
        W_ln(price)       0.0785059       0.0212424       3.6957202       0.0
------------------------------------------------------------------------------
Instrumented: W_ln(price)
Instruments: W_bathrooms, W_bedrooms, W_beds, W_guests_included,
             W_host_listings_count, W_pool

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                           MI/DF       VALUE           PROB
Anselin-Kelejian Test             1          31.545          0.0000
================================ END OF REPORT ===============================
```

As we can see, results are again very similar in all the other variable. It is also very clear that the estimate of the spatial lag of price is statistically significant. This points to evidence that there are processes of spatial interaction between property owners when they set their price.

Prediction performance of spatial models

Even if we are not interested in the interpretation of the model to learn more about how alternative factors determine the price of an AirBnb property, spatial econometrics can be useful. In a purely predictive setting, the use of explicitly spatial models is likely to improve accuracy in cases where space plays a key role in the data generating process. To have a quick look at this issue, we can use the mean squared error (MSE), a standard metric of accuracy in the machine learning literature, to evaluate whether explicitly spatial models are better than traditional, non-spatial ones:

```python
from sklearn.metrics import mean_squared_error as mse

mses = pd.Series({'OLS': mse(y, m1.predy.flatten()), \
                     'OLS+W': mse(y, m2.predy.flatten()), \
                     'Lag': mse(y, m3.predy_e)
                    })
mses.sort_values()
```

```
Lag      0.531327
OLS+W    0.532402
OLS      0.532545
dtype: float64
```

We can see that the inclusion of the number of surrounding pools (which was insignificant) only marginally reduces the MSE. The inclusion of the spatial lag of price, however, does a better job at improving the accuracy of the model.

Exercise

> *Run a regression including both the spatial lag of pools and of the price. How does its predictive performance compare?*

Development workflow

Dependencies

In addition to the packages required to run the tutorial (see the install guide for more detail), you will need the following libraries:

- `npm` and `node.js`
- `gitbook`
- `make`
- `cp` , `rm` , and `zip` Unix utilities.

Workflow

The overall structure of the workflow is as follows:

1. Develop material on Jupyter notebooks and place them under the `content/` folder.
2. When you want to build the website with the new content run on, the root folder:

   `> make notebooks`

3. When you want to obtain a new version of the pdf or ebook formats, run on the root folder:

   `> make book`

4. When you want to push a new version to the website to Github Pages, make sure to commit all your changes first on the `master` branch (assuming your remote is named as `origin`):

   ```
   > git add .
   > git commit -m "commit message"
   > git push origin master
   ```

   Then you can run:

   `> make website`

   This will compile a new version of the website, pdf, eupb and mobi files, check them in, switch to the `gh-pages` branch, check the new version of the website and push it to Github.