> Geographic Data Science

with

PySAL

and the

pydata stack

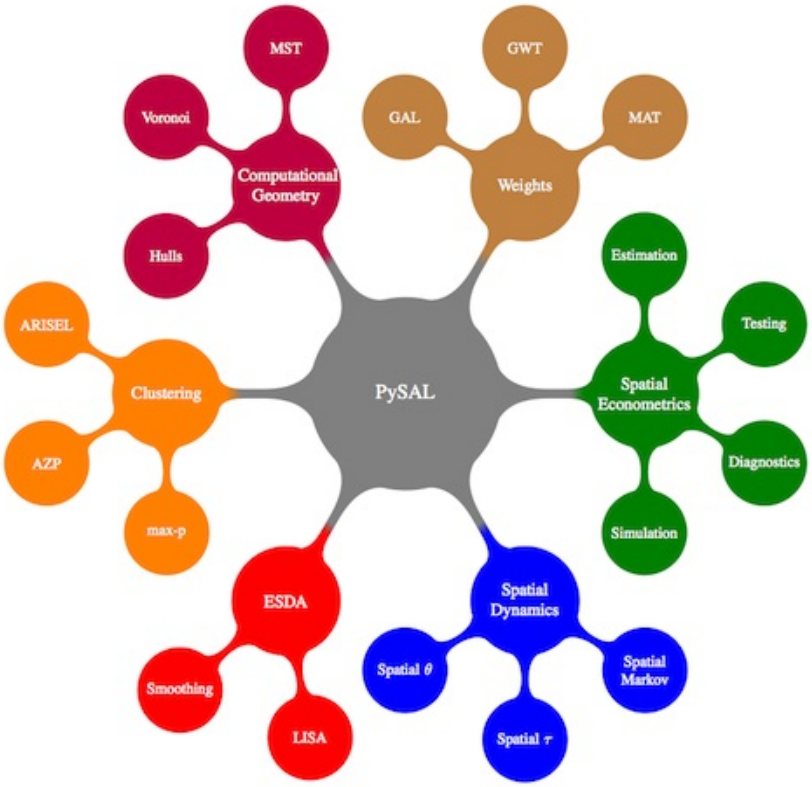
Sergio J. Rey

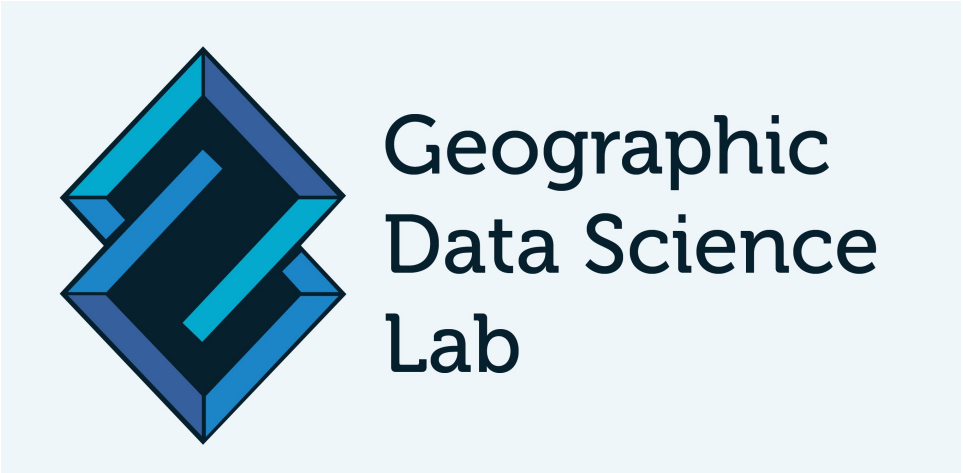
Dani Arribas-Bel

Table of Contents

Geographic Data Science with PySAL and the pydata stack

This two-part tutorial will first provide participants with a gentle introduction to Python for geospatial analysis, and an introduction to version `PySAL` 1.11 and the related eco-system of libraries to facilitate common tasks for Geographic Data Scientists. The first part will cover munging geo-data and exploring relations over space. This includes importing data in different formats (e.g. shapefile, GeoJSON), visualizing, combining and tidying them up for analysis, and will use libraries such as `pandas`, `geopandas`, `PySAL`, or `rasterio`. The second part will provide a gentle overview to demonstrate several techniques that allow to extract geospatial insight from the data. This includes spatial clustering and regression and point pattern analysis, and will use libraries such as `PySAL`, `scikit-learn`, or `clusterpy`. A particular emphasis will be set on presenting concepts through visualization, for which libraries such as `matplotlib`, `seaborn`, and `folium` will be used.



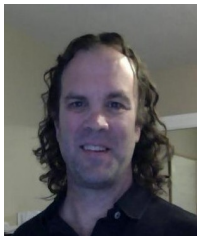


Distribution

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

License

About the authors



Sergio Rey is professor of geographical sciences and core faculty member of the GeoDa Center for Geospatial Analysis and Computation at the Arizona State University. His research interests include open science, spatial and spatio-temporal data analysis, spatial econometrics, visualization, high performance geocomputation, spatial inequality dynamics, integrated multiregional modeling, and regional science. He co-founded the Python Spatial Analysis Library (PySAL) in 2007 and continues to direct the PySAL project. Rey is a fellow of the spatial econometrics association and editor of the journal Geographical Analysis.

Dani Arribas-Bel is Lecturer in Geographic Data Science and member of the Geographic Data Science Lab at the University of Liverpool (UK). Dani is interested in undestanding cities as well as in the quantitative and computational methods required to leverage the power of the large amount of urban data increasingly becoming available. He is also part of the team of core developers of PySAL, the open-source library written in Python for spatial analysis. Dani regularly teaches Geographic Data Science and Python courses at the University of Liverpool and has designed and developed several workshops at different levels on spatial analysis and econometrics, Python and open source scientific computing.

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. Centrography and distance based statistics
2. Exercise (10 min)
3. Spatial clustering a (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.

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

Software and Tools Installation

Dependencies

Participants should have installed the following dependencies:

- Anaconda or MiniConda Python distributions for Python 2.7. See installation instructions on the links.
- `git`
- A `conda` environment loaded with all the dependencies can be installed by running the `pydata.sh` script available as part of the `envs` repository (Github link). To install it, follow these instructions:
    - Clone the repository on your machine:

      `> git clone https://github.com/darribas/envs.git`
    - Navigate into the folder:

      `> cd envs`
    - Run the script:

      `> bash pydata.sh`

Once installed, you need to activate the environment to run the notebooks. In Windows, open up PowerShell and type:

`> activate pydata`

And if you are on GNU/Linux or OSX:

`> source activate pydata`

Get started

Instructions to fire up a notebook here.

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

Spatial Data Processing with PySAL & Pandas

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
```

```
/home/serge/Dropbox/p/pysal/workshops/scipy16/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.

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

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

For the purposes of this part of the workshop, we'll use the `texas.dbf` example data, and the `usjoin.csv` data.

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

Working with shapefiles

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 = ps.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 0x7f1e5423c550>
```

```
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
```

```
[<pysal.cg.shapes.Polygon at 0x7f1e23a751d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75190>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75410>,
 <pysal.cg.shapes.Polygon at 0x7f1e56420490>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a56ed0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a754d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75550>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a755d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75650>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a756d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75750>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a757d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75890>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75910>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75990>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75a10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75a90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75b10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75b90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a56c90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75bd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75c50>,
```

```
 <pysal.cg.shapes.Polygon at 0x7f1e23a75cd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75d50>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75dd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75e50>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75ed0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75f50>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a75fd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87090>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87110>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87190>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87210>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87290>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87310>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87390>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87410>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87490>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87510>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87590>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87610>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87690>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87710>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87790>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87810>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87890>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87910>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87990>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87a10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87a90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87b10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87b90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87c10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87c90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87d10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87d90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87e10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87e90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87f10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87f90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23a87fd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de0d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de150>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de1d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de250>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de2d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de350>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de3d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de450>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de4d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de550>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de5d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de650>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de6d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de750>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de7d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de850>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de8d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de950>,
 <pysal.cg.shapes.Polygon at 0x7f1e231de9d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231dea50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231dead0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231deb50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231debd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231dec50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231decd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ded50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231dedd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231dee50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231deed0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231def50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231defd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed090>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed110>,
```

```
 <pysal.cg.shapes.Polygon at 0x7f1e231ed190>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed210>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed290>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed310>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed390>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed410>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed490>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed510>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed590>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed610>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed690>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed710>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed790>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed810>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed890>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed910>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ed990>,
 <pysal.cg.shapes.Polygon at 0x7f1e231eda10>,
 <pysal.cg.shapes.Polygon at 0x7f1e231eda90>,
 <pysal.cg.shapes.Polygon at 0x7f1e231edb10>,
 <pysal.cg.shapes.Polygon at 0x7f1e231edb90>,
 <pysal.cg.shapes.Polygon at 0x7f1e231edc10>,
 <pysal.cg.shapes.Polygon at 0x7f1e231edc90>,
 <pysal.cg.shapes.Polygon at 0x7f1e231edd10>,
 <pysal.cg.shapes.Polygon at 0x7f1e231edd90>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ede10>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ede90>,
 <pysal.cg.shapes.Polygon at 0x7f1e231edf10>,
 <pysal.cg.shapes.Polygon at 0x7f1e231edf90>,
 <pysal.cg.shapes.Polygon at 0x7f1e231edfd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff0d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff150>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff1d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff250>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff2d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff350>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff3d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff450>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff4d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff550>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff5d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff650>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff6d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff750>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff7d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff850>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff8d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff950>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ff9d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ffa50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ffad0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ffb50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ffbd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ffc50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ffcd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ffd50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ffdd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ffe50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231ffed0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231fff50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231fffd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b090>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b110>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b190>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b210>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b290>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b310>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b390>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b410>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b490>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b510>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b590>,
```

```
 <pysal.cg.shapes.Polygon at 0x7f1e2319b610>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b690>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b710>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b790>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b810>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b890>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b910>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319b990>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319ba10>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319ba90>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319bb10>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319bb90>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319bc10>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319bc90>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319bd10>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319bd90>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319be10>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319be90>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319bf10>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319bf90>,
 <pysal.cg.shapes.Polygon at 0x7f1e2319bfd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b70d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7150>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b71d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7250>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b72d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7350>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b73d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7450>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b74d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7550>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b75d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7650>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b76d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7750>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b77d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7850>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b78d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7950>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b79d0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7a50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7ad0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7b50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7bd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7c50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7cd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7d50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7dd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7e50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7ed0>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7f50>,
 <pysal.cg.shapes.Polygon at 0x7f1e231b7fd0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151090>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151110>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151190>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151290>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151310>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151390>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151410>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151490>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151510>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151590>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151610>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151690>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151710>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151790>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151810>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151890>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151910>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151990>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151a10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151a90>,
```

```
 <pysal.cg.shapes.Polygon at 0x7f1e23151b10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151c90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151d10>,
 <pysal.cg.shapes.Polygon at 0x7f1e54229d90>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151e50>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151ed0>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151f50>,
 <pysal.cg.shapes.Polygon at 0x7f1e23151f10>,
 <pysal.cg.shapes.Polygon at 0x7f1e23169110>,
 <pysal.cg.shapes.Polygon at 0x7f1e23169190>,
 <pysal.cg.shapes.Polygon at 0x7f1e23169210>,
 <pysal.cg.shapes.Polygon at 0x7f1e23169290>,
 <pysal.cg.shapes.Polygon at 0x7f1e23169390>,
 <pysal.cg.shapes.Polygon at 0x7f1e23169410>,
 <pysal.cg.shapes.Polygon at 0x7f1e23169490>,
 <pysal.cg.shapes.Polygon at 0x7f1e23169550>]
```

```
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-16-aa03438a2fa8>", line 1
    polygon. #press tab when the cursor is right after the dot
                                          ^
SyntaxError: invalid syntax
```

Working with Data Tables

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
```

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

So, the header is a list containing the names of all of the fields we can read. If we were interested in getting the `['NAME', 'STATE_NAME', 'HR90', 'HR80']` fields.

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', array([[  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-20-1fef6a3c3a50> in <module>()
----> 1 HRs = f.by_col('HR90', 'HR80')


TypeError: __call__() takes exactly 2 arguments (3 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
```

```
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 ],
       [ 17.3692707 ,  14.53139626],
       [  5.06893755,   4.9960032 ],
       [  9.73560868,   8.84307335],
       [  5.66989851,   4.67049647],
       [  0.        ,   0.        ],
       [  4.08893052,   4.44077341],
       [ 16.49348507,   0.        ],
```

```
       [  0.       ,   7.17154332],
       [  0.       ,   8.1799591 ],
       [  3.48074279,   7.8746358 ],
       [  0.       ,   6.0397415 ],
       [  3.67511944,  12.63104711],
       [  8.19705726,  17.14148582],
       [  0.       ,   0.        ],
       [ 11.19883532,   4.79616307],
       [  8.53606487,   5.95876534],
       [  6.30954634,   5.23450586],
       [ 19.83995768,   8.36942649],
       [ 43.51610096,   0.        ],
       [ 14.83459427,   0.        ],
       [  7.8459064 ,   6.77920141],
       [  5.76850971,  12.41398879],
       [  2.21160651,   5.3564733 ],
       [  4.7187618 ,  16.32386549],
       [  0.       ,   0.        ],
       [ 13.6190056 ,   9.08475248],
       [  0.       ,   0.        ],
       [  5.78904712,   5.74382539],
       [ 11.6411725 ,  10.35132393],
       [ 11.57638838,   5.93938854],
       [  8.66447889,   7.23170379],
       [  7.58454876,  11.06999399],
       [ 16.12643122,   4.11776817],
       [  0.       ,   0.        ],
       [  0.       ,   9.41885655],
       [  0.       ,  11.28795575],
       [  8.68383071,  18.26906873],
       [  4.18077679,   0.        ],
       [  5.50986955,   7.17463051],
       [ 20.67397147,  18.76524676],
       [  0.       ,   0.        ],
       [  0.       ,  13.81692573],
       [  7.75526031,   8.85335741],
       [ 13.7258939 ,  20.66542674],
       [  9.54973022,   4.49964003],
       [  4.80598249,   5.01724679],
       [  5.24022789,   4.89079552],
       [  9.3250237 ,   9.65344145],
       [  3.15613527,   6.68621809],
       [ 15.27205076,  15.54581351],
       [  0.       ,   4.31499461],
       [  0.       ,   0.        ],
       [  3.677958 ,   3.4935108 ],
       [  0.       ,   0.        ],
       [ 16.55903295,  13.85425326],
       [  4.27240878,  24.17911891],
       [ 12.96260289,   6.24687656],
       [  0.       ,   0.        ],
       [  5.04362738,   6.85824018],
       [  0.       ,   4.01654818],
       [  5.78041365,   7.92173328],
       [  0.       ,  29.62152801],
       [ 11.11778178,   7.92841771],
       [ 16.82821755,  10.78167116],
       [  4.53823463,   8.09814957],
       [  5.32162576,  13.85310171],
       [  5.14522395,   3.73616684],
       [ 15.7535989 ,  15.99138866],
       [ 24.88112651,  22.42368558],
       [  7.81127949,   4.58883994],
       [ 14.89203276,  13.77695116],
       [  0.       ,   8.51426139],
       [  6.06428138,   9.65176434],
       [  0.       ,   5.65834889],
       [  7.15537906,  10.99384345],
       [  0.       ,   0.        ],
       [ 16.26129579,   6.17894217],
       [  3.69959304,   6.71636779],
       [  0.       ,  10.13941698],
```

```
       [  4.25034534,  12.82275456],
       [ 13.35470086,  35.39253539],
       [ 22.97970126,  16.23307275],
       [  7.02790077,   6.36415707],
       [  8.69822382,  15.94438598],
       [ 10.13378803,  16.35947213],
       [ 15.88088069,  17.42271184],
       [  5.83200398,   7.39109226],
       [  4.60071541,   7.52700312],
       [  9.78469752,  12.27479928],
       [ 10.30741159,  11.11792209],
       [ 11.33681683,  23.13278217],
       [  8.31669993,  11.00352113],
       [  9.01485648,   5.13347023],
       [  5.62160947,   3.03250849],
       [  0.        ,   7.11642471],
       [  6.97447343,   5.00387801],
       [  6.02627456,   5.7607005 ],
       [  2.38171793,   4.43262411],
       [  9.14599291,  17.72107035],
       [  0.        ,  12.86752879],
       [  9.67949484,  14.86488601],
       [ 12.52316786,  15.0987553 ],
       [  6.21890547,  16.04878832],
       [  9.82342395,  13.32054561],
       [  7.47328301,   5.28387625],
       [  6.61157025,  12.43688282],
       [  9.74445175,   6.11989754],
       [  0.        ,   0.        ],
       [  0.        ,  25.56237219],
       [  7.50391611,  12.90801426],
       [  0.        ,   0.        ],
       [  3.86428627,  23.46473585],
       [ 11.77123447,  21.66866018],
       [  0.        ,   8.42318059],
       [  3.43288706,   6.38630776],
       [  3.87923928,   9.07523369],
       [ 12.49375312,  15.63273495],
       [  4.31053063,   0.        ],
       [ 12.64382349,   2.24769611],
       [ 39.13511398,   0.        ],
       [  6.30835226,  12.65742675],
       [  0.        ,   0.        ],
       [ 11.43510578,   0.        ],
       [  7.66270572,   8.82130059],
       [ 21.17938943,  20.21602264],
       [ 16.56770109,  10.15099607],
       [ 10.95830365,  15.67420453],
       [  7.95696871,  16.48206751],
       [  0.        ,   0.        ],
       [  4.67195116,   2.34878245],
       [  8.125292  ,   7.86311883],
       [  5.08323802,  11.92520511],
       [  8.33437513,  11.38303927],
       [  0.        ,   7.24637681],
       [  7.49568998,  14.43313849],
       [  0.        ,   0.        ],
       [  7.89577576,  17.37196859],
       [  6.95458655,   7.66107408],
       [ 21.83235867,  14.94835344],
       [ 10.95050372,   0.        ],
       [  0.        ,  24.05002405],
       [ 15.26338885,   7.27212284],
       [  7.52785306,  11.14454474],
       [  0.        ,  15.26426254],
       [  6.1716966 ,  10.74575543],
       [  4.93060178,   8.32986256],
       [  8.73743993,   7.05467372],
       [ 13.6286201 ,   4.5605874 ],
       [  6.44703759,  15.92392912],
       [ 12.21077898,  13.08936447],
       [ 17.19605965,   7.54489211],
```

```
       [  7.50219707,   9.74627205],
       [  9.7761267 ,  16.38874094],
       [  7.26343008,  17.59633996],
       [ 17.12915382,   0.        ],
       [  0.        ,  18.78110621],
       [ 11.14827202,   0.        ],
       [  0.        ,   0.        ],
       [ 24.52182442,   7.2337963 ],
       [ 10.01241539,   4.10939202],
       [  7.20126218,  10.36955499],
       [  5.8796725 ,   3.74468722],
       [  4.37653521,   7.4795914 ],
       [ 19.4760931 ,   9.05059281],
       [  2.86590434,   9.85804416],
       [  4.06064211,   3.92049241],
       [ 12.2159785 ,   8.74584572],
       [  5.31123858,  17.18213058],
       [  2.44648318,  18.95016108],
       [ 24.26006793,   8.20411847],
       [ 24.18379686,   0.        ],
       [ 15.35921361,   4.40160218],
       [  0.        ,   0.        ],
       [ 10.04469891,   6.42508353],
       [  8.15396066,  12.39355249],
       [ 11.52573257,  12.19319205],
       [ 12.96613246,   9.13075237],
       [  4.03355921,   9.82294148],
       [  0.        ,   0.        ],
       [ 16.74480911,   7.12098554],
       [ 11.37958502,  22.65262204],
       [ 10.45396336,  14.82919464],
       [  6.37251153,  12.12231415],
       [  5.08021662,  12.3170912 ],
       [  0.        ,   0.        ],
       [  6.88687448,   3.71298617],
       [  7.34537976,   5.79105861],
       [ 14.25110446,  13.46937401],
       [ 10.35080964,   8.34943582],
       [ 15.03778243,  14.47900809],
       [  9.95272456,  10.6202209 ],
       [ 23.54931879,  32.62023941],
       [  4.56965293,   0.        ],
       [  8.40392631,  15.04381511],
       [ 27.63957988,   0.        ],
       [  3.78902698,   8.46131066],
       [  4.50172352,   6.40216576],
       [  5.43980852,  10.62529884],
       [  6.31193587,   4.70543949],
       [ 13.27492367,   8.99054195],
       [  9.24884004,   9.27749707],
       [  8.13293053,  15.03039196],
       [ 23.24905551,  15.79498114],
       [ 18.39050982,  19.38376483],
       [  8.54325327,   4.31704369],
       [ 12.48439451,   0.        ],
       [  7.508447  ,   8.28321985],
       [ 11.42530705,  11.88301175],
       [  0.        ,   0.        ],
       [  5.91179943,  10.22090923],
       [ 15.17946265,  18.03273111],
       [ 10.30169242,  13.92536007],
       [  8.84642604,   7.05355411],
       [  5.11286653,  12.48252447],
       [  5.45857488,  10.64325151],
       [ 10.83188908,  21.14835572],
       [  5.3526027 ,   9.8089703 ],
       [  7.17221841,   8.72004302],
       [ 24.66699556,   8.5719184 ],
       [ 12.37133808,  12.09043646],
       [  8.24674254,   0.        ],
       [  0.        ,   6.41889723],
       [  0.        ,   3.47005344],
```

```
       [  5.30468802,  16.64745806],
       [ 31.72186271,  18.1356547 ],
       [ 40.7996736 ,   0.        ],
       [  9.58497077,  11.72986129],
       [  4.17920428,   3.58847382],
       [  5.24851729,  11.92057491],
       [  9.25654901,  11.08223015],
       [  6.80862653,  10.9170933 ],
       [  5.5890901 ,   2.33754091],
       [ 18.0626516 ,  26.63044926],
       [  5.3079965 ,  10.95950463],
       [ 10.30414398,  17.39897719],
       [  5.50527405,  13.98964766],
       [ 13.04886801,   6.4499484 ],
       [ 10.77702339,   0.        ],
       [  8.12611734,  27.6854928 ],
       [  0.        ,   0.        ],
       [ 14.80823338,  17.11533289],
       [  7.30031678,   8.23832776],
       [  5.648122  ,   7.62122511],
       [ 12.30201446,  11.76132146]])
```

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([[u'Lipscomb', u'Texas', u'0.0', u'0.0'],
       [u'Sherman', u'Texas', u'0.0', u'10.501995379'],
       [u'Dallam', u'Texas', u'18.31166453', u'5.1038636248'],
       ...,
       [u'Hidalgo', u'Texas', u'7.3003167816', u'8.2383277607'],
       [u'Willacy', u'Texas', u'5.6481219994', u'7.6212251119'],
       [u'Cameron', u'Texas', u'12.302014455', u'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 '/home/serge/anaconda2/envs/pydata/lib/python2.7/site-packages/pysal/contrib/pdutilities/__init__.
pyc'>
```

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	FIPS	STFIPS	COFIP
0	Lake of the Woods	Minnesota	27	077	27077	27	77
1	Ferry	Washington	53	019	53019	53	19
2	Stevens	Washington	53	065	53065	53	65
3	Okanogan	Washington	53	047	53047	53	47
4	Pend Oreille	Washington	53	051	53051	53	51

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	1933	1934	1935
0	Alabama	1	323	267	224	162	166	211	217
1	Arizona	4	600	520	429	321	308	362	416
2	Arkansas	5	310	228	215	157	157	187	207
3	California	6	991	887	749	580	546	603	660
4	Colorado	8	634	578	471	354	353	368	444

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	FIPS	STFIPS	CO
1707	Navajo	Arizona	04	017	04017	4	17
1708	Coconino	Arizona	04	005	04005	4	5
1722	Mohave	Arizona	04	015	04015	4	15
1726	Apache	Arizona	04	001	04001	4	1
2002	Yavapai	Arizona	04	025	04025	4	25
2182	Gila	Arizona	04	007	04007	4	7
2262	Maricopa	Arizona	04	013	04013	4	13
2311	Greenlee	Arizona	04	011	04011	4	11
2326	Graham	Arizona	04	009	04009	4	9
2353	Pinal	Arizona	04	021	04021	4	21
2499	Pima	Arizona	04	019	04019	4	19
2514	Cochise	Arizona	04	003	04003	4	3
2615	Santa Cruz	Arizona	04	023	04023	4	23
3080	La Paz	Arizona	04	012	04012	4	12

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	FIPS	STFIPS	CO
1707	Navajo	Arizona	04	017	04017	4	17
1708	Coconino	Arizona	04	005	04005	4	5
1722	Mohave	Arizona	04	015	04015	4	15
1726	Apache	Arizona	04	001	04001	4	1
2002	Yavapai	Arizona	04	025	04025	4	25
2182	Gila	Arizona	04	007	04007	4	7
2262	Maricopa	Arizona	04	013	04013	4	13
2311	Greenlee	Arizona	04	011	04011	4	11
2326	Graham	Arizona	04	009	04009	4	9
2353	Pinal	Arizona	04	021	04021	4	21
2499	Pima	Arizona	04	019	04019	4	19
2514	Cochise	Arizona	04	003	04003	4	3
2615	Santa Cruz	Arizona	04	023	04023	4	23
3080	La Paz	Arizona	04	012	04012	4	12

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 longidtude 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)
```

```
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      True
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   False
3081   False
3082   False
3083   False
3084   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] < -121)]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS	C
27	Whatcom	Washington	53	073	53073	53	7
31	Skagit	Washington	53	057	53057	53	5
44	Clallam	Washington	53	009	53009	53	9
47	Snohomish	Washington	53	061	53061	53	6
48	Island	Washington	53	029	53029	53	2
57	Jefferson	Washington	53	031	53031	53	3
71	Kitsap	Washington	53	035	53035	53	3
80	King	Washington	53	033	53033	53	3
85	Mason	Washington	53	045	53045	53	4
92	Grays Harbor	Washington	53	027	53027	53	2
107	Pierce	Washington	53	053	53053	53	5
116	Thurston	Washington	53	067	53067	53	6
130	Pacific	Washington	53	049	53049	53	4
131	Lewis	Washington	53	041	53041	53	4
169	Skamania	Washington	53	059	53059	53	5
170	Cowlitz	Washington	53	015	53015	53	1
171	Wahkiakum	Washington	53	069	53069	53	6
181	Clatsop	Oregon	41	007	41007	41	7
183	Columbia	Oregon	41	009	41009	41	9
193	Clark	Washington	53	011	53011	53	1
216	Tillamook	Oregon	41	057	41057	41	5
217	Washington	Oregon	41	067	41067	41	6
224	Multnomah	Oregon	41	051	41051	41	5
225	Hood River	Oregon	41	027	41027	41	2

226	Wasco	Oregon	41	065	41065	41	6
240	Clackamas	Oregon	41	005	41005	41	5
242	Yamhill	Oregon	41	071	41071	41	7
253	Marion	Oregon	41	047	41047	41	4
269	Polk	Oregon	41	053	41053	41	5
272	Lincoln	Oregon	41	041	41041	41	4
...	...	...	...	...	...	...	.
493	Curry	Oregon	41	015	41015	41	1
517	Josephine	Oregon	41	033	41033	41	3
594	Siskiyou	California	06	093	06093	6	9
601	Del Norte	California	06	015	06015	6	1
687	Humboldt	California	06	023	06023	6	2
709	Trinity	California	06	105	06105	6	1
735	Shasta	California	06	089	06089	6	8
907	Tehama	California	06	103	06103	6	1
974	Butte	California	06	007	06007	6	7
1005	Mendocino	California	06	045	06045	6	4
1047	Glenn	California	06	021	06021	6	2
1090	Yuba	California	06	115	06115	6	1
1101	Lake	California	06	033	06033	6	3
1141	Colusa	California	06	011	06011	6	1
1172	Sutter	California	06	101	06101	6	1
1262	Yolo	California	06	113	06113	6	1
1281	Napa	California	06	055	06055	6	5
1282	Sonoma	California	06	097	06097	6	9

1301	Sacramento	California	06	067	06067	6	6
1401	Marin	California	06	041	06041	6	4
1404	San Joaquin	California	06	077	06077	6	7
1447	Solano	California	06	095	06095	6	9
1462	Contra Costa	California	06	013	06013	6	1
1514	Alameda	California	06	001	06001	6	1
1530	San Francisco	California	06	075	06075	6	7
1559	San Mateo	California	06	081	06081	6	8
1609	Santa Clara	California	06	085	06085	6	8
1664	Santa Cruz	California	06	087	06087	6	8
1731	San Benito	California	06	069	06069	6	6
1745	Monterey	California	06	053	06053	6	5

69 rows × 70 columns

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.

First, specify the observation. Here, we'll use Cook County, IL:

```
data_table.query('(NAME == "Cook") & (STATE_NAME == "Illinois")')
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS	COFI
3044	Cook	Illinois	17	031	17031	17	31

1 rows × 70 columns

```
geom = data_table.query('(NAME == "Cook") & (STATE_NAME == "Illinois")').geometry
```

```
geom.values[0].centroid
```

```
(-87.82107391263027, 41.84346628270174)
```

```
cook_county_centroid = geom.values[0].centroid
```

```
import scipy.spatial.distance as d
def near_target_point(polygon, target=cook_county_centroid, threshold=1):
    return d.euclidean(polygon.centroid, target) < threshold
```

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

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS	CO
631	Will	Illinois	17	197	17197	17	197
633	Kendall	Illinois	17	093	17093	17	93
634	Lake	Indiana	18	089	18089	18	89
635	Porter	Indiana	18	127	18127	18	127
686	Grundy	Illinois	17	063	17063	17	63
718	Kankakee	Illinois	17	091	17091	17	91
731	Newton	Indiana	18	111	18111	18	111
3010	Racine	Wisconsin	55	101	55101	55	101
3019	Kenosha	Wisconsin	55	059	55059	55	59
3028	McHenry	Illinois	17	111	17111	17	111
3030	Lake	Illinois	17	097	17097	17	97
3044	Cook	Illinois	17	031	17031	17	31
3045	Kane	Illinois	17	089	17089	17	89
3046	De Kalb	Illinois	17	037	17037	17	37
3059	Du Page	Illinois	17	043	17043	17	43

15 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()
```

```
[u'Lake of the Woods',
 u'Ferry',
 u'Stevens',
 u'Okanogan',
 u'Pend Oreille',
 u'Boundary',
 u'Lincoln',
 u'Flathead',
 u'Glacier',
 u'Toole',
 u'Liberty',
 u'Hill',
 u'Sheridan',
 u'Divide',
 u'Burke',
 u'Renville',
 u'Bottineau',
 u'Rolette',
 u'Towner',
 u'Cavalier',
 u'Pembina',
 u'Kittson',
 u'Roseau',
 u'Blaine',
 u'Phillips',
 u'Valley',
 u'Daniels',
 u'Whatcom',
 u'Bonner',
 u'Ward',
 u'Koochiching',
 u'Skagit',
 u'Williams',
 u'McHenry',
 u'St. Louis',
 u'Roosevelt',
 u'Mountrial',
 u'Marshall',
 u'Ramsey',
 u'Walsh',
 u'Beltrami',
 u'Pierce',
 u'Chelan',
 u'Pondera',
 u'Clallam',
 u'Benson',
 u'Chouteau',
 u'Snohomish',
 u'Island',
 u'Sanders',
 u'Lake',
 u'Nelson',
 u'Grand Forks',
 u'Polk',
 u'Pennington',
 u'Douglas',
 u'McKenzie',
 u'Jefferson',
 u'Richland',
 u'Teton',
 u'McCone',
 u'Shoshone',
 u'Spokane',
 u'Lake',
 u'Clearwater',
 u'Kootenai',
 u'Garfield',
 u'Red Lake',
```

```
 u'Grant',
 u'Lincoln',
 u'Lewis and Clark',
 u'Kitsap',
 u'Itasca',
 u'Sheridan',
 u'Wells',
 u'McLean',
 u'Eddy',
 u'Dunn',
 u'Fergus',
 u'Dawson',
 u'King',
 u'Cascade',
 u'Griggs',
 u'Steele',
 u'Traill',
 u'Mason',
 u'Missoula',
 u'Petroleum',
 u'Powell',
 u'Kittitas',
 u'Foster',
 u'Mercer',
 u'Grays Harbor',
 u'Norman',
 u'Mahnomen',
 u'Mineral',
 u'Cass',
 u'Aroostook',
 u'Judith Basin',
 u'Hubbard',
 u'Benewah',
 u'Wibaux',
 u'Golden Valley',
 u'Billings',
 u'Stutsman',
 u'Kidder',
 u'Burleigh',
 u'Pierce',
 u'Oliver',
 u'Adams',
 u'Whitman',
 u'Barnes',
 u'Cass',
 u'Prairie',
 u'Becker',
 u'Clay',
 u'Thurston',
 u'Latah',
 u'Meagher',
 u'Yakima',
 u'Aitkin',
 u'Stark',
 u'Morton',
 u'Bayfield',
 u'Clearwater',
 u'Custer',
 u'Rosebud',
 u'Granite',
 u'Wadena',
 u'Crow Wing',
 u'Pacific',
 u'Lewis',
 u'Broadwater',
 u'Carlton',
 u'Golden Valley',
 u'Douglas',
 u'Musselshell',
 u'Wheatland',
 u'Franklin',
 u'Benton',
```

```
 u'Grant',
 u'Otter Tail',
 u'Garfield',
 u'Fallon',
 u'Idaho',
 u'Ravalli',
 u'Ashland',
 u'Logan',
 u'Emmons',
 u'La Moure',
 u'Slope',
 u'Hettinger',
 u'Ransom',
 u'Richland',
 u'Wilkin',
 u'Nez Perce',
 u'Columbia',
 u'Walla Walla',
 u'Iron',
 u'Somerset',
 u'Piscataquis',
 u'Jefferson',
 u'Yellowstone',
 u'Treasure',
 u'Lewis',
 u'Asotin',
 u'Sioux',
 u'Pine',
 u'Penobscot',
 u'Skamania',
 u'Cowlitz',
 u'Wahkiakum',
 u'Todd',
 u'Morrison',
 u'McIntosh',
 u'Dickey',
 u'Sargent',
 u'Bowman',
 u'Adams',
 u'Deer Lodge',
 u'Mille Lacs',
 u'Clatsop',
 u'Sweet Grass',
 u'Columbia',
 u'Silver Bow',
 u'Washburn',
 u'Sawyer',
 u'Burnett',
 u'Kanabec',
 u'Carter',
 u'Stillwater',
 u'Grant',
 u'Douglas',
 u'Clark',
 u'Klickitat',
 u'Big Horn',
 u'Traverse',
 u'Umatilla',
 u'Wallowa',
 u'Price',
 u'Campbell',
 u'Harding',
 u'McPherson',
 u'Perkins',
 u'Corson',
 u'Brown',
 u'Beaverhead',
 u'Marshall',
 u'Roberts',
 u'Morrow',
 u'Union',
 u'Madison',
```

```
 u'Benton',
 u'Gilliam',
 u'Powder River',
 u'Stearns',
 u'Tillamook',
 u'Washington',
 u'Pope',
 u'Stevens',
 u'Isanti',
 u'Chisago',
 u'Sherman',
 u'Polk',
 u'Multnomah',
 u'Hood River',
 u'Wasco',
 u'Lemhi',
 u'Washington',
 u'Franklin',
 u'Barron',
 u'Rusk',
 u'Carbon',
 u'Walworth',
 u'Edmunds',
 u'Day',
 u'Big Stone',
 u'Sherburne',
 u'Ziebach',
 u'Dewey',
 u'Clackamas',
 u'Wright',
 u'Yamhill',
 u'Anoka',
 u'Kandiyohi',
 u'Swift',
 u'Taylor',
 u'Oxford',
 u'Grant',
 u'Meeker',
 u'Coos',
 u'Washington',
 u'Chippewa',
 u'Marion',
 u'Lac Qui Parle',
 u'Adams',
 u'Potter',
 u'Faulk',
 u'Hennepin',
 u'Spink',
 u'St. Croix',
 u'Dunn',
 u'Butte',
 u'Valley',
 u'Clark',
 u'Codington',
 u'Chippewa',
 u'Ramsey',
 u'Baker',
 u'Polk',
 u'Wheeler',
 u'Meade',
 u'Lincoln',
 u'Clark',
 u'Essex',
 u'Grand Isle',
 u'Franklin',
 u'Orleans',
 u'Clinton',
 u'Park',
 u'Crook',
 u'Big Horn',
 u'Campbell',
 u'Sheridan',
```

```
 u'Franklin',
 u'Grant',
 u'McLeod',
 u'Carver',
 u'Deuel',
 u'Dakota',
 u'Yellow Medicine',
 u'Sully',
 u'Hyde',
 u'Hand',
 u'Renville',
 u'Pierce',
 u'Custer',
 u'Eau Claire',
 u'Washington',
 u'Jefferson',
 u'Scott',
 u'Hamlin',
 u'Lamoille',
 u'Linn',
 u'Stanley',
 u'Caledonia',
 u'Waldo',
 u'Haakon',
 u'Fremont',
 u'Sibley',
 u'Chittenden',
 u'Kennebec',
 u'Goodhue',
 u'Benton',
 u'Redwood',
 u'Wood',
 u'Pepin',
 u'Teton',
 u'Beadle',
 u'Lyon',
 u'Lincoln',
 u'Lawrence',
 u'Buffalo',
 u'Trempealeau',
 u'Jackson',
 u'Clark',
 u'Crook',
 u'Johnson',
 u'Hughes',
 u'Essex',
 u'Le Sueur',
 u'Rice',
 u'Kingsbury',
 u'Brookings',
 u'Pennington',
 u'Gem',
 u'Brown',
 u'Washington',
 u'Androscoggin',
 u'Nicollet',
 u'Wabasha',
 u'Malheur',
 u'Hancock',
 u'Grafton',
 u'Deschutes',
 u'Knox',
 u'Boise',
 u'Lincoln',
 u'Addison',
 u'Carroll',
 u'Lane',
 u'Blue Earth',
 u'Juneau',
 u'Butte',
 u'Lyman',
 u'Orange',
```

```
 u'Waseca',
 u'Buffalo',
 u'Jerauld',
 u'Moody',
 u'Pipestone',
 u'Dodge',
 u'Sanborn',
 u'Murray',
 u'Cottonwood',
 u'Steele',
 u'Olmsted',
 u'Miner',
 u'Lake',
 u'Winona',
 u'Weston',
 u'Jones',
 u'Cumberland',
 u'Washakie',
 u'Monroe',
 u'Payette',
 u'Hamilton',
 u'Watonwan',
 u'Herkimer',
 u'La Crosse',
 u'Elmore',
 u'Hot Springs',
 u'Jefferson',
 u'Harney',
 u'Sagadahoc',
 u'Fremont',
 u'Jackson',
 u'Blaine',
 u'Teton',
 u'Windsor',
 u'Douglas',
 u'Aurora',
 u'Brule',
 u'Madison',
 u'Canyon',
 u'Mellette',
 u'Camas',
 u'Custer',
 u'Rutland',
 u'Faribault',
 u'Minnehaha',
 u'Rock',
 u'Freeborn',
 u'Nobles',
 u'Jackson',
 u'Martin',
 u'Houston',
 u'Mower',
 u'Fillmore',
 u'Davison',
 u'Hanson',
 u'McCook',
 u'York',
 u'Washington',
 u'Ada',
 u'Warren',
 u'Tripp',
 u'Belknap',
 u'Vernon',
 u'Shannon',
 u'Owyhee',
 u'Bonneville',
 u'Bingham',
 u'Klamath',
 u'Lake',
 u'Coos',
 u'Merrimack',
 u'Sullivan',
```

```
 u'Strafford',
 u'Richland',
 u'Natrona',
 u'Gregory',
 u'Niobrara',
 u'Charles Mix',
 u'Turner',
 u'Lincoln',
 u'Worth',
 u'Mitchell',
 u'Allamakee',
 u'Winnebago',
 u'Winneshiek',
 u'Converse',
 u'Osceola',
 u'Dickinson',
 u'Kossuth',
 u'Howard',
 u'Emmet',
 u'Lyon',
 u'Douglas',
 u'Hutchinson',
 u'Fall River',
 u'Sublette',
 u'Crawford',
 u'Saratoga',
 u'Bennett',
 u'Todd',
 u'Bennington',
 u'Lincoln',
 u'Fulton',
 u'Rockingham',
 u'Sioux',
 u'Windham',
 u"O'Brien",
 u'Cerro Gordo',
 u'Clay',
 u'Hancock',
 u'Palo Alto',
 u'Floyd',
 u'Chickasaw',
 u'Iowa',
 u'Grant',
 u'Hillsborough',
 u'Lincoln',
 u'Gooding',
 u'Minidoka',
 u'Cheshire',
 u'Yankton',
 u'Bon Homme',
 u'Power',
 u'Union',
 u'Clay',
 u'Fayette',
 u'Clayton',
 u'Montgomery',
 u'Caribou',
 u'Bannock',
 u'Sioux',
 u'Dawes',
 u'Sheridan',
 u'Jackson',
 u'Keya Paha',
 u'Boyd',
 u'Cherry',
 u'Curry',
 u'Rensselaer',
 u'Schenectady',
 u'Plymouth',
 u'Cherokee',
 u'Bremer',
 u'Butler',
```

```
 u'Buena Vista',
 u'Twin Falls',
 u'Pocahontas',
 u'Humboldt',
 u'Wright',
 u'Franklin',
 u'Otsego',
 u'Holt',
 u'Essex',
 u'Knox',
 u'Cedar',
 u'Jerome',
 u'Schoharie',
 u'Brown',
 u'Lafayette',
 u'Albany',
 u'Rock',
 u'Josephine',
 u'Dixon',
 u'Berkshire',
 u'Franklin',
 u'Middlesex',
 u'Worcester',
 u'Dubuque',
 u'Cassia',
 u'Webster',
 u'Delaware',
 u'Buchanan',
 u'Black Hawk',
 u'Goshen',
 u'Platte',
 u'Bear Lake',
 u'Woodbury',
 u'Ida',
 u'Sac',
 u'Calhoun',
 u'Hamilton',
 u'Hampshire',
 u'Hardin',
 u'Grundy',
 u'Dakota',
 u'Delaware',
 u'Jo Daviess',
 u'Columbia',
 u'Oneida',
 u'Greene',
 u'Suffolk',
 u'Carbon',
 u'Pierce',
 u'Box Butte',
 u'Antelope',
 u'Albany',
 u'Franklin',
 u'Jackson',
 u'Wayne',
 u'Hampden',
 u'Jones',
 u'Benton',
 u'Linn',
 u'Tama',
 u'Thurston',
 u'Sweetwater',
 u'Plymouth',
 u'Norfolk',
 u'Monona',
 u'Crawford',
 u'Carroll',
 u'Greene',
 u'Boone',
 u'Marshall',
 u'Story',
 u'Ulster',
```

```
u'Cuming',
u'Bristol',
u'Stanton',
u'Madison',
u'Grant',
u'Loup',
u'Hooker',
u'Garfield',
u'Thomas',
u'Wheeler',
u'Blaine',
u'Dutchess',
u'Barnstable',
u'Burt',
u'Litchfield',
u'Hartford',
u'Clinton',
u'Tolland',
u'Windham',
u'Sullivan',
u'Providence',
u'Cache',
u'Siskiyou',
u'Garden',
u'Box Elder',
u'Rich',
u'Scotts Bluff',
u'Morrill',
u'Wayne',
u'Del Norte',
u'Humboldt',
u'Elko',
u'Modoc',
u'Washoe',
u'Cedar',
u'Boone',
u'Jasper',
u'Polk',
u'Poweshiek',
u'Harrison',
u'Guthrie',
u'Shelby',
u'Audubon',
u'Iowa',
u'Dallas',
u'Johnson',
u'Rock Island',
u'Scott',
u'Bristol',
u'Kent',
u'Colfax',
u'Dodge',
u'Platte',
u'Arthur',
u'Greeley',
u'McPherson',
u'Logan',
u'Custer',
u'Valley',
u'Will',
u'Lucas',
u'Kendall',
u'Lake',
u'Porter',
u'Fulton',
u'Geauga',
u'New London',
u'Williams',
u'Banner',
u'Washington',
u'Newport',
u'Fairfield',
```

```
 u'Laramie',
 u'Wyoming',
 u'Washington',
 u'Middlesex',
 u'New Haven',
 u'Lackawanna',
 u'La Salle',
 u'Orange',
 u'Elk',
 u'Cuyahoga',
 u'Venango',
 u'Forest',
 u'Ottawa',
 u'Cameron',
 u'Wood',
 u'Pike',
 u'Lycoming',
 u'Muscatine',
 u'Sullivan',
 u'Bureau',
 u'Henry',
 u'Uinta',
 u'Noble',
 u'De Kalb',
 u'Putnam',
 u'Nance',
 u'Lorain',
 u'Mahaska',
 u'Pottawattamie',
 u'Washington',
 u'Marion',
 u'Madison',
 u'Warren',
 u'Keokuk',
 u'Cass',
 u'Adair',
 u'Sandusky',
 u'Trumbull',
 u'Mercer',
 u'Marshall',
 u'Henry',
 u'Clinton',
 u'Grundy',
 u'Humboldt',
 u'Butler',
 u'Saunders',
 u'Erie',
 u'Kosciusko',
 u'Starke',
 u'Clarion',
 u'Cheyenne',
 u'Defiance',
 u'Weber',
 u'Louisa',
 u'Luzerne',
 u'Polk',
 u'Douglas',
 u'Sherman',
 u'Howard',
 u'Lincoln',
 u'Merrick',
 u'Keith',
 u'Kimball',
 u'Jefferson',
 u'Morgan',
 u'Trinity',
 u'Westchester',
 u'Sussex',
 u'Summit',
 u'Portage',
 u'Mercer',
 u'Rockland',
```

```
 u'Putnam',
 u'Columbia',
 u'Kankakee',
 u'Whitley',
 u'Jasper',
 u'Huron',
 u'Allen',
 u'Medina',
 u'Summit',
 u'Centre',
 u'Clearfield',
 u'Monroe',
 u'Seneca',
 u'Paulding',
 u'Stark',
 u'Newton',
 u'Deuel',
 u'Passaic',
 u'Sarpy',
 u'Shasta',
 u'Lassen',
 u'Northumberland',
 u'Fulton',
 u'Butler',
 u'Armstrong',
 u'Pulaski',
 u'Montour',
 u'Hancock',
 u'Mills',
 u'Putnam',
 u'Montgomery',
 u'Adams',
 u'Clarke',
 u'Wapello',
 u'Jefferson',
 u'Union',
 u'Henry',
 u'Hamilton',
 u'Lucas',
 u'Monroe',
 u'Davis',
 u'Knox',
 u'Marshall',
 u'Union',
 u'Carbon',
 u'Mahoning',
 u'Lawrence',
 u'Bergen',
 u'Livingston',
 u'Warren',
 u'Tooele',
 u'Morris',
 u'Des Moines',
 u'Henderson',
 u'Warren',
 u'Cass',
 u'Ashland',
 u'Wabash',
 u'Seward',
 u'Lancaster',
 u'Buffalo',
 u'Dawson',
 u'York',
 u'Hall',
 u'Huntington',
 u'Iroquois',
 u'Miami',
 u'Ford',
 u'Moffat',
 u'Weld',
 u'Jackson',
 u'Perkins',
```

```
 u'Logan',
 u'Sedgwick',
 u'Routt',
 u'Larimer',
 u'Daggett',
 u'Crawford',
 u'Lander',
 u'Eureka',
 u'Richland',
 u'Wayne',
 u'Van Wert',
 u'Wyandot',
 u'Stark',
 u'Peoria',
 u'Northampton',
 u'Pershing',
 u'Schuylkill',
 u'Adams',
 u'Wells',
 u'Woodford',
 u'Columbiana',
 u'Cass',
 u'White',
 u'Salt Lake',
 u'Allen',
 u'Indiana',
 u'Fremont',
 u'Page',
 u'Taylor',
 u'Ringgold',
 u'Nassau',
 u'Davis',
 u'Van Buren',
 u'Decatur',
 u'Wayne',
 u'Essex',
 u'Appanoose',
 u'Snyder',
 u'Uintah',
 u'Beaver',
 u'Mifflin',
 u'Duchesne',
 u'Hudson',
 u'Lee',
 u'Hardin',
 u'Otoe',
 u'Lehigh',
 u'Hunterdon',
 u'Suffolk',
 u'McLean',
 u'Somerset',
 u'Carroll',
 u'Tazewell',
 u'Blair',
 u'Benton',
 u'Phillips',
 u'Huntingdon',
 u'Cambria',
 u'Union',
 u'Mercer',
 u'Carroll',
 u'Fulton',
 u'Morrow',
 u'Marion',
 u'Saline',
 u'Adams',
 u'Clay',
 u'Fillmore',
 u'Juniata',
 u'Hayes',
 u'Chase',
 u'Frontier',
```

```
u'Gosper',
u'Auglaize',
u'Kearney',
u'Wasatch',
u'Phelps',
u'Berks',
u'Westmoreland',
u'Allegheny',
u'Grant',
u'Holmes',
u'Dauphin',
u'Tuscarawas',
u'Hancock',
u'McDonough',
u'Perry',
u'Hancock',
u'Bucks',
u'Clark',
u'Scotland',
u'Schuyler',
u'Middlesex',
u'Jefferson',
u'Blackford',
u'Putnam',
u'Atchison',
u'Jay',
u'Utah',
u'Nodaway',
u'Mercer',
u'Howard',
u'Harrison',
u'Tippecanoe',
u'Worth',
u'Knox',
u'Nemaha',
u'Lebanon',
u'Logan',
u'Gage',
u'Johnson',
u'Morgan',
u'Union',
u'Vermilion',
u'Warren',
u'Shelby',
u'Washington',
u'Grand',
u'Coshocton',
u'Tehama',
u'Monmouth',
u'Plumas',
u'Delaware',
u'Montgomery',
u'Mason',
u'Clinton',
u'Yuma',
u'Washington',
u'Harrison',
u'Mercer',
u'Tipton',
u'Champaign',
u'Brooke',
u'Madison',
u'Delaware',
u'Gentry',
u'Sullivan',
u'Fountain',
u'Darke',
u'Thayer',
u'Jefferson',
u'Adair',
u'Dundy',
u'Franklin',
```

```
 u'Webster',
 u'Nuckolls',
 u'Hitchcock',
 u'Harlan',
 u'Furnas',
 u'Red Willow',
 u'Logan',
 u'Bedford',
 u'Cumberland',
 u'Randolph',
 u'Lancaster',
 u'Knox',
 u'Franklin',
 u'Piatt',
 u'De Witt',
 u'Somerset',
 u'Schuyler',
 u'Licking',
 u'Champaign',
 u'Holt',
 u'Richardson',
 u'Pawnee',
 u'Grundy',
 u'Boulder',
 u'Lewis',
 u'Chester',
 u'Hamilton',
 u'York',
 u'Montgomery',
 u'Rio Blanco',
 u'Guernsey',
 u'Adams',
 u'Miami',
 u'Ohio',
 u'Boone',
 u'Burlington',
 u'Vermillion',
 u'Menard',
 u'Belmont',
 u'Ocean',
 u'Fulton',
 u'Muskingum',
 u'Butte',
 u'Daviess',
 u'Franklin',
 u'Fayette',
 u'Philadelphia',
 u'Andrew',
 u'Cass',
 u'White Pine',
 u'Madison',
 u'Brown',
 u'Garfield',
 u'Henry',
 u'Adams',
 u'Delaware',
 u'Macon',
 u'De Kalb',
 u'Macon',
 u'Clark',
 u'Linn',
 u'Marshall',
 u'Greene',
 u'Wayne',
 u'Juab',
 u'Churchill',
 u'Norton',
 u'Phillips',
 ...]
```

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')]
```

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

```
data_table[HRs]
```

	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
5	0.000000	0.000000	4.573101	4.000640
6	7.976390	5.536179	5.633168	5.720497
7	1.011173	1.689475	4.490115	2.814460
8	11.529039	9.273857	28.227324	5.500096
9	0.000000	5.708740	0.000000	6.605892
10	0.000000	0.000000	0.000000	0.000000
11	3.574045	3.840688	7.413585	1.888146
12	0.000000	0.000000	0.000000	0.000000
13	0.000000	0.000000	0.000000	0.000000
14	0.000000	0.000000	0.000000	0.000000
15	0.000000	0.000000	0.000000	0.000000
16	2.945942	0.000000	0.000000	0.000000
17	0.000000	0.000000	19.161808	5.219752
18	0.000000	0.000000	0.000000	0.000000
19	0.000000	8.117213	0.000000	0.000000
20	0.000000	0.000000	0.000000	0.000000
21	0.000000	4.864050	0.000000	0.000000
22	0.000000	0.000000	0.000000	2.218377
23	4.119804	9.910312	14.287755	14.863258
24	5.530668	0.000000	12.421589	0.000000
25	5.854801	5.811757	6.504065	4.045798
26	0.000000	10.811980	0.000000	0.000000
27	1.422131	2.439530	4.061193	2.086920
28	2.138534	2.142245	5.518079	3.756292
29	0.708135	1.138434	0.570854	1.150993

...	...	...	...	...
3055	0.000000	0.000000	2.107526	0.739919
3056	0.000000	0.970572	4.400324	0.825491
3057	0.611430	2.568301	1.974919	2.828994
3058	2.022286	2.033140	0.000000	3.987956
3059	0.850723	1.981012	2.782690	2.260130
3060	1.790825	3.732470	5.757329	4.007173
3061	0.556604	1.060271	1.010560	2.769193
3062	0.000000	1.756836	5.505395	2.907653
3063	0.427592	3.688132	2.625587	0.000000
3064	0.896660	2.704530	4.229303	2.938915
3065	1.051403	5.379304	7.057466	3.424679
3066	2.095434	6.671377	9.243279	7.285916
3067	1.872835	3.951663	5.339935	3.201065
3068	1.917913	6.382639	1.304631	0.000000
3069	1.939789	4.960564	0.000000	6.072530
3070	5.452765	15.156192	24.841012	28.268787
3071	7.520089	9.163383	10.590286	6.443839
3072	7.202448	9.746302	11.850014	12.561604
3073	8.253379	15.655752	21.173432	16.479507
3074	2.181802	3.074760	3.191133	3.300700
3075	4.902862	11.782264	7.680787	18.362582
3076	18.513376	17.133324	15.034136	12.027015
3077	4.159907	4.126434	3.967782	6.585273
3078	5.403098	5.970974	4.127839	2.586787
3079	1.121183	1.096311	2.442074	2.806112
3080	5.046682	13.152054	13.251761	5.521552
3081	3.411368	7.393533	11.453817	8.691999
3082	1.544425	6.023552	5.280349	4.367330
3083	9.302820	1.800148	3.000030	3.727712
3084	3.396162	2.284879	1.194743	2.048855

3085 rows × 4 columns

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.

Exploratory Geovisualization with PySAL

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 geovisualization within a self-contained exploratory workflow. The idea here is the 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.

PySAL Viz Module

Contributors:

- Dani Arribas-Bel daniel.arribas.bel@gmail.com
- Serge Rey sjsrey@gmail.com

This document describes the main structure, components and usage of the mapping module in PySAL. The 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 random as rdm
from pysal.contrib.viz 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:

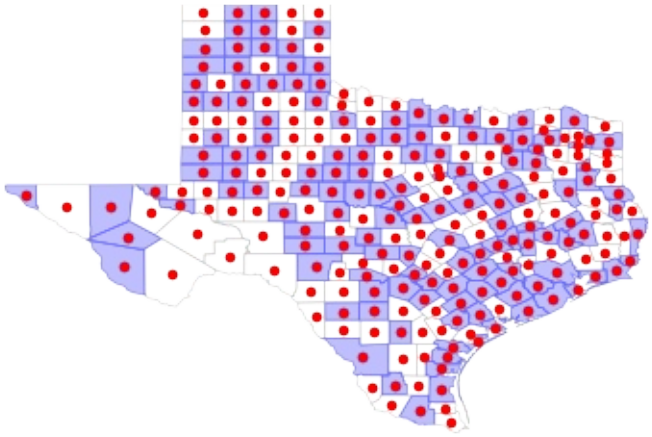
- map_poly_shape: to read in polygon shapefiles
- map_line_shape: to read in line shapefiles
- map_point_shape: to read in point shapefiles

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 = ps.open(shp_link)
some = [bool(rdm.getrandbits(1)) for i in ps.open(shp_link)]

fig = figure()

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()
```



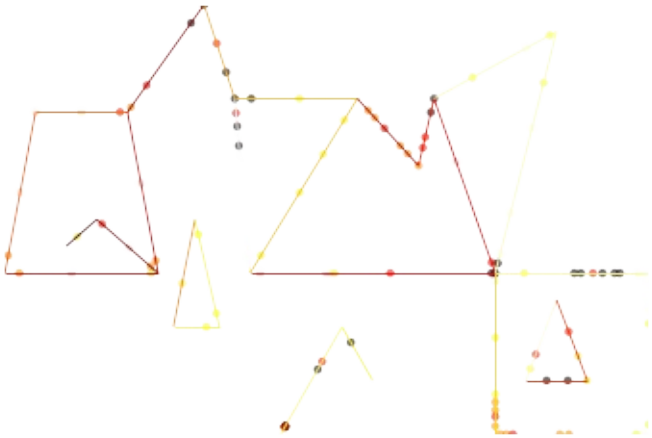
Medium-level component

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

Example

```
net_link = ps.examples.get_path('eberly_net.shp')
net = ps.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 = ps.open(pts_link)

fig = figure()

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')
```

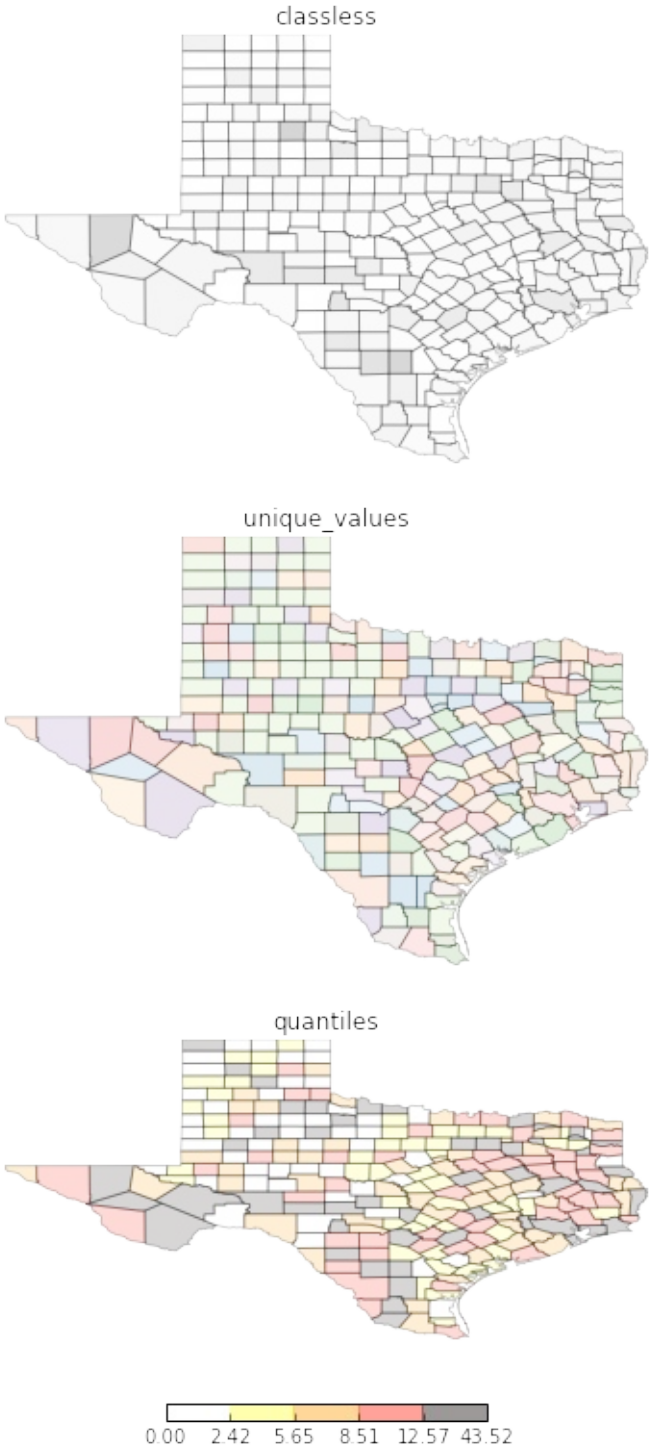
```
callng plt.show()
```

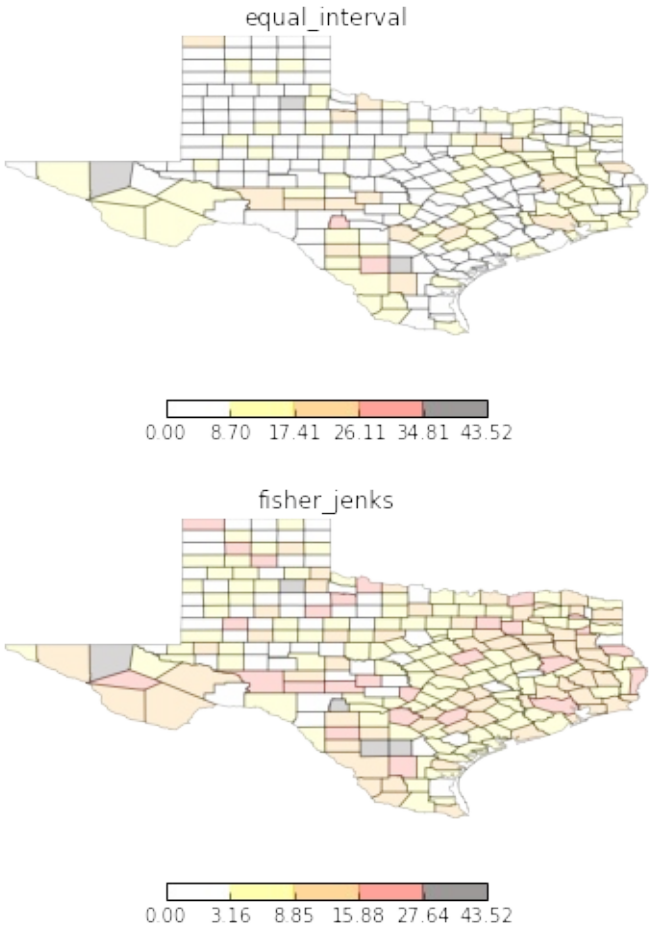
Higher-level component

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_jenks']
for typ in types:
    maps.plot_choropleth(shp_link, values, typ, title=typ)
```





Folium

Contributors:

- Levi Wolf levi.john.wolf@gmail.com
- Serge Rey sjsrey@gmail.com

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
```

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

ImportError                               Traceback (most recent call last)

<ipython-input-6-677ccf6438e5> in <module>()
      1 import pysal as ps
----> 2 import geojson as gj
      3 from pysal.contrib.viz import folium_mapping as fm


ImportError: No module named geojson
```

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()
```

```
[u'type', 'bbox', 'features']
```

```
js.type
```

```
'FeatureCollection'
```

```
js.bbox
```

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

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

{"bbox": [-100.5494155883789, 36.05754852294922, -99.99715423583984, 36.49387741088867], "geometry": {"coordinates"
686645507812, 36.49387741088867], [-100.00114440917969, 36.49251937866211], [-99.99715423583984, 36.05754852294
.54059600830078, 36.058135986328125], [-100.5494155883789, 36.48944854736328], [-100.00686645507812, 36.4938774
"type": "Polygon"}, "properties": {"BLK60": 0.029359953, "BLK70": 0.0286861733, "BLK80": 0.0265533723, "BLK90": 0.031816
TY_FIPS": "295", "COFIPS": 295, "DNL60": 1.293817423, "DNL70": 1.3170337879, "DNL80": 1.3953635084, "DNL90": 1.2153856
": 1.4948859166, "DV70": 2.2709475333, "DV80": 3.5164835165, "DV90": 6.1016949153, "FH60": 6.7245119306, "FH70": 4.5,
353601497, "FH90": 6.0935799782, "FIPS": "48295", "FIPSNO": 48295, "FP59": 22.4, "FP69": 12.1, "FP79": 10.851262862, "FP8
99674, "GI59": 0.2869290401, "GI69": 0.378218563, "GI79": 0.4070049836, "GI89": 0.3730049522, "HC60": 0.0, "HC70": 0.0, "
"HC90": 0.0, "HR60": 0.0, "HR70": 0.0, "HR80": 0.0, "HR90": 0.0, "MA60": 32.4, "MA70": 34.3, "MA80": 31.0, "MA90": 35.8, "MFIL59": 8.531
8847402, "MFIL69": 8.9704320743, "MFIL79": 9.8020637224, "MFIL89": 10.252241206, "NAME": "Lipscomb", "PO60": 3406, "PO70": 3486,
"PO80": 3766, "PO90": 3143, "POL60": 8.1332938612, "POL70": 8.1565102261, "POL80": 8.2337687092, "POL90": 8.0529330368, "PS60":
-1.514026445, "PS70": -1.449058083, "PS80": -1.476411495, "PS90": -1.571799202, "RD60": -0.917851658, "RD70": -0.602337681, "
": -0.355503211, "RD90": -0.605606852, "SOUTH": 1, "STATE_FIPS": "48", "STATE_NAME": "Texas", "STFIPS": 48, "UE60": 2.0, "UE70": 1
UE80": 1.9411764706, "UE90": 1.7328519856}, "type": "Feature"}

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.

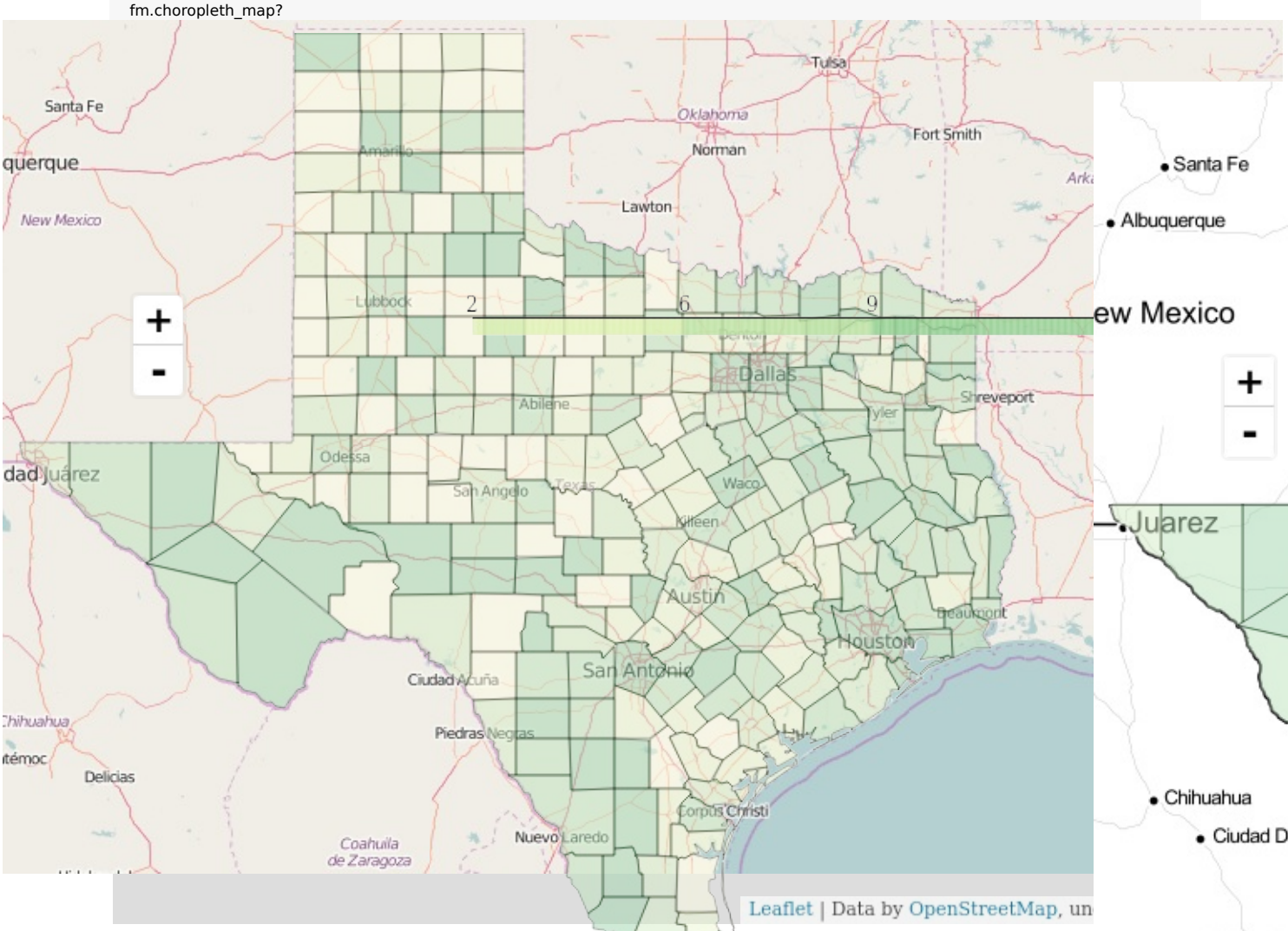
```
list(js.features[0].properties.keys())[:5]
```

```
[u'HR90', u'PS90', u'FH80', u'HC60', u'FP79']
```

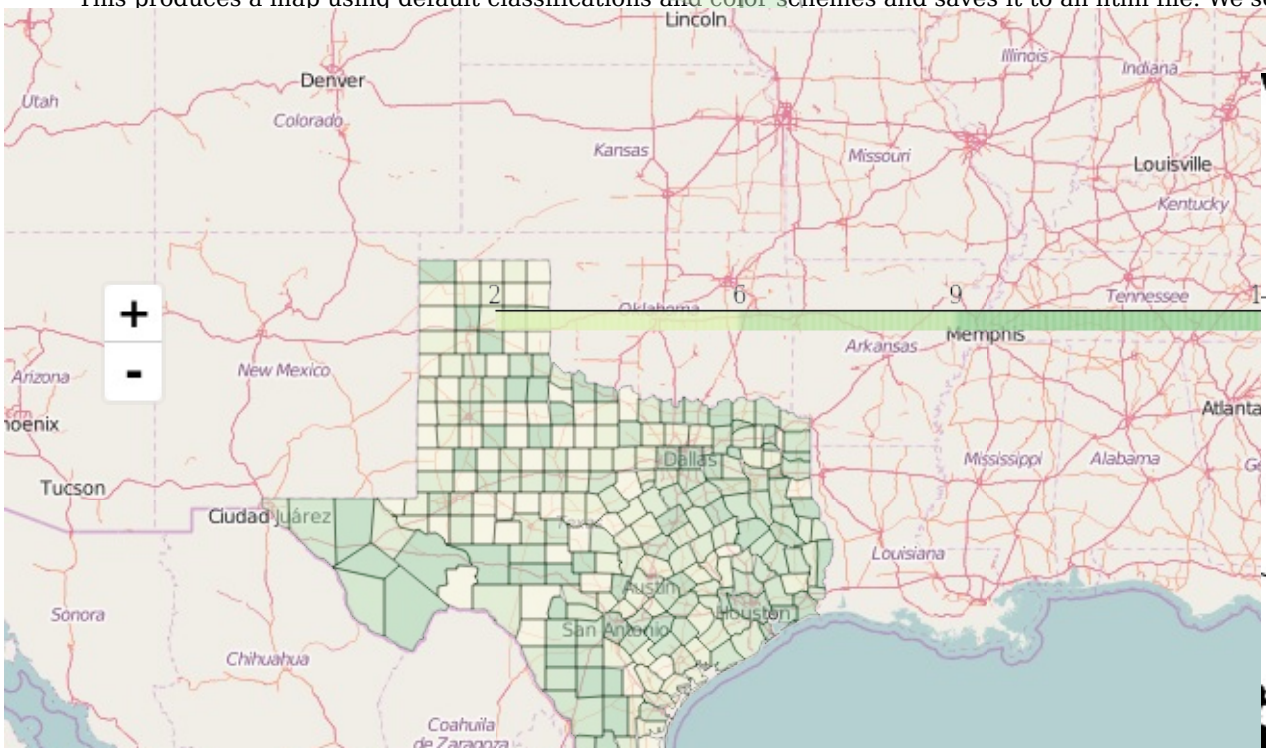
We can map these attributes by calling them as arguments to the choropleth mapping function:

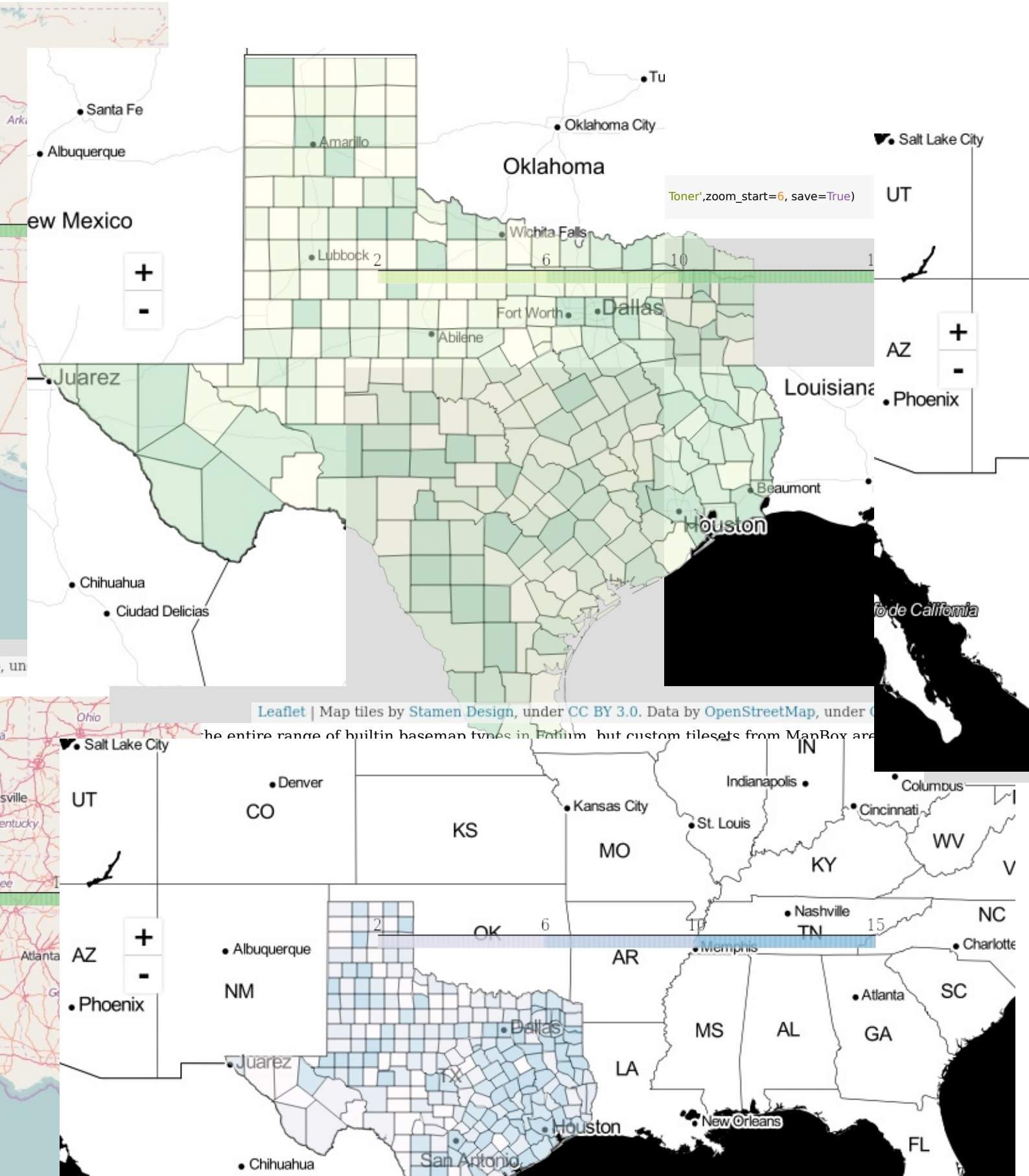
```
import folium as fol
```

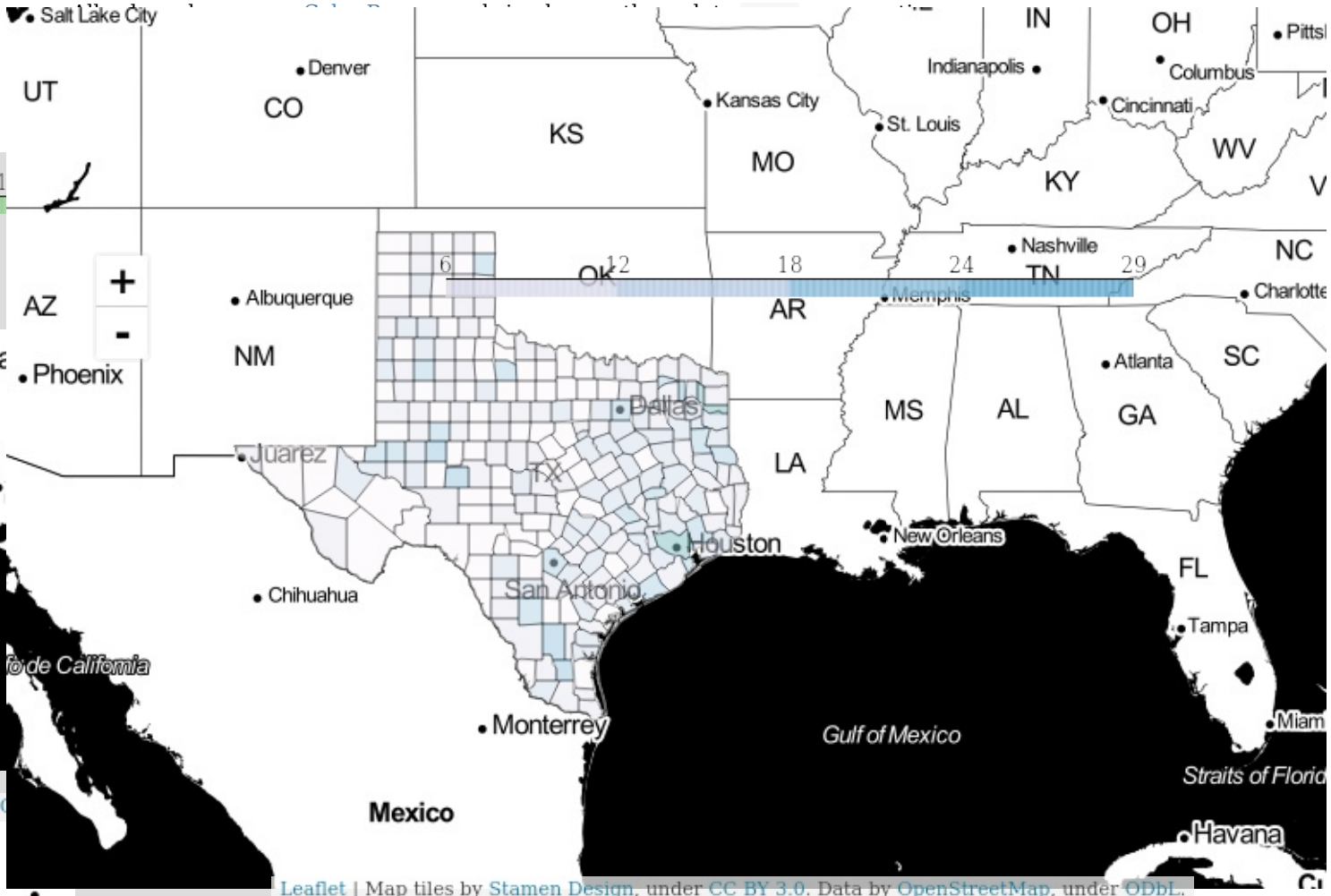
```
fm.choropleth_map?
```



This produces a map using default classifications and color schemes and saves it to an html file. We set the




Santa Fe
Albuquerque
Amarillo
Oklahoma City
Oklahoma
ew Mexico
Wichita Falls
Lubbock
Fort Worth
Dallas
Abilene
Juarez
Louisiana
Beaumont
Houston
Chihuahua
Ciudad Delicias
Salt Lake City
UT
AZ
Phoenix
Leaflet | Map tiles by Stamen Design, under CC BY 3.0. Data by OpenStreetMap, under
Denver
CO
KS
Kansas City
MO
St. Louis
IN
Indianapolis
Columbus
Cincinnati
KY
WV
Nashville
TN
NC
Charlotte
OK
AR
Memphis
Albuquerque
NM
Dallas
TX
Juarez
LA
Houston
New Orleans
MS
AL
GA
Atlanta
SC
FL
San Antonio
Chihuahua



orts up to 6 classes.

py

e don't have time to go into the details here today, we note that for for publication ready maps
to Cartopy. For an example of a recent publication and code see Rey (2016).


OH
Columbus
Cincinnati
WV
NC
29
Charlotte
Atlanta
SC
GA
FL
Tampa
Straits of Florid
Havana

Spatial Weights

Spatial weights are mathematical structures used to represent spatial relationships. They characterize the relationship of each observation to every other observation using some concept of proximity or closeness that depends on the weight type.

They can be build in PySAL from shapefiles, as well as some types of files.

```
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 verted each 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 0x7fac6769e590>
```

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: 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	FIPS	STFIPS	COFI
4	Ochiltree	Texas	48	357	48357	48	357
0	Lipscomb	Texas	48	295	48295	48	295
3	Hansford	Texas	48	195	48195	48	195
5	Roberts	Texas	48	393	48393	48	393
6	Hemphill	Texas	48	211	48211	48	211
7	Hutchinson	Texas	48	233	48233	48	233

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.]])
```

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 dimensions
```

```
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.])
```

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

```
qW.sparse
```

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

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	FIPS	STFIPS	COFIP
0	Lipscomb	Texas	48	295	48295	48	295
1	Sherman	Texas	48	421	48421	48	421
2	Dallam	Texas	48	111	48111	48	111
3	Hansford	Texas	48	195	48195	48	195
4	Ochiltree	Texas	48	357	48357	48	357

5 rows × 70 columns

The observation we were discussing above is in the fifth row: Pend Oreille county, Washington. Note that its FIPS code is 53051.

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')
```

```
ps.knnW_from_shapefile??
```

Now, Pend Oreille county has a different index:

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

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

KeyError                                  Traceback (most recent call last)

<ipython-input-17-1d8a3009bc1e> in <module>()
----> 1 qW[4] #fails, since no FIPS is 4.


/home/serge/anaconda2/envs/pydata/lib/python2.7/site-packages/pysal/weights/weights.pyc in __getitem__(self, key)
    504         {1: 1.0, 4: 1.0, 101: 1.0, 85: 1.0, 5: 1.0}
    505         """
--> 506         return dict(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['48111']
```

```
{u'48205': 1.0, u'48341': 1.0, u'48421': 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 = ['48111']
self_and_neighbors.extend(qW.neighbors['48111'])
```

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

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

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS	COFIPS
1	Sherman	Texas	48	421	48421	48	421
2	Dallam	Texas	48	111	48111	48	111
8	Hartley	Texas	48	205	48205	48	205
9	Moore	Texas	48	341	48341	48	341

4 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 neighs') will fail because there is no column called 'neighs'
```

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	STFIPS	C
FIPS							
48295	Lipscomb	Texas	48	295	48295	48	29
48421	Sherman	Texas	48	421	48421	48	42
48111	Dallam	Texas	48	111	48111	48	11
48195	Hansford	Texas	48	195	48195	48	19
48357	Ochiltree	Texas	48	357	48357	48	35

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	STFIPS	CC
FIPS							
48111	Dallam	Texas	48	111	48111	48	11
48205	Hartley	Texas	48	205	48205	48	20
48421	Sherman	Texas	48	421	48421	48	42
48341	Moore	Texas	48	341	48341	48	34

4 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['48111']
```

```
{u'48205': 1.0, u'48421': 1.0}
```

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

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) #silence because there will be a lot of warnings
```

```
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
```

```
dataframe.query('FIPS not in @islands')
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS
0	Lipscomb	Texas	48	295	48295	48
1	Sherman	Texas	48	421	48421	48
2	Dallam	Texas	48	111	48111	48
3	Hansford	Texas	48	195	48195	48
4	Ochiltree	Texas	48	357	48357	48
5	Roberts	Texas	48	393	48393	48
6	Hemphill	Texas	48	211	48211	48
7	Hutchinson	Texas	48	233	48233	48
8	Hartley	Texas	48	205	48205	48
9	Moore	Texas	48	341	48341	48
10	Potter	Texas	48	375	48375	48
11	Carson	Texas	48	065	48065	48
12	Gray	Texas	48	179	48179	48
13	Wheeler	Texas	48	483	48483	48
14	Oldham	Texas	48	359	48359	48
15	Randall	Texas	48	381	48381	48
16	Armstrong	Texas	48	011	48011	48
17	Collingsworth	Texas	48	087	48087	48
18	Donley	Texas	48	129	48129	48

27	Wilbarger	Texas	48	487	48487	48	
28	Motley	Texas	48	345	48345	48	
29	Cottle	Texas	48	101	48101	48	
30	Floyd	Texas	48	153	48153	48	
31	Hale	Texas	48	189	48189	48	
32	Lamb	Texas	48	279	48279	48	
33	Bailey	Texas	48	017	48017	48	
35	Wichita	Texas	48	485	48485	48	
38	Red River	Texas	48	387	48387	48	
41	Lamar	Texas	48	277	48277	48	
43	King	Texas	48	269	48269	48	
...	...	...	...	...	...	...	
119	Midland	Texas	48	329	48329	48	
126	Anderson	Texas	48	001	48001	48	
128	Freestone	Texas	48	161	48161	48	
132	Hudspeth	Texas	48	229	48229	48	
140	Tom Green	Texas	48	451	48451	48	
144	Upton	Texas	48	461	48461	48	
145	Reagan	Texas	48	383	48383	48	
146	Leon	Texas	48	289	48289	48	
149	Concho	Texas	48	095	48095	48	
158	Robertson	Texas	48	395	48395	48	
165	Madison	Texas	48	313	48313	48	
166	Schleicher	Texas	48	413	48413	48	
167	Menard	Texas	48	327	48327	48	

172	Brazos	Texas	48	041	48041	48	
183	Presidio	Texas	48	377	48377	48	
191	Bastrop	Texas	48	021	48021	48	
200	Fayette	Texas	48	149	48149	48	
205	Caldwell	Texas	48	055	48055	48	
212	Gonzales	Texas	48	177	48177	48	
214	Medina	Texas	48	325	48325	48	
217	Uvalde	Texas	48	463	48463	48	
218	Kinney	Texas	48	271	48271	48	
224	Atascosa	Texas	48	013	48013	48	
228	Zavala	Texas	48	507	48507	48	
229	Frio	Texas	48	163	48163	48	
230	Maverick	Texas	48	323	48323	48	
234	La Salle	Texas	48	283	48283	48	
235	McMullen	Texas	48	311	48311	48	
239	Webb	Texas	48	479	48479	48	
242	Duval	Texas	48	131	48131	48	

93 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. To use these measures, we first must extract the polygons' centroids.

For each polygon `poly` in `dataframe.geometry`, we want `poly.centroid`. So, one way to do this is to make a list of all of the centroids:

```
>>> ps.min_threshold_dist_from_shapefile(ps.examples.get_path("stl_hom.shp"), ps.cg.sphere.RADIUS_EARTH_MILES)
```

```
31.846942936393717
```

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

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

```
3958.755865744055
```

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

```
threshold
```

```
1.0040319244447573
```

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

```
threshold
```

```
60.47758554135752
```

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

```
knn4 = ps.knnW_from_shapefile?
```

```
knn4 = ps.knnW_from_shapefile
```

```
knn4 = ps.knnW_from_shapefile
```

```
knn4 = ps.knnW_from_shapefile
```

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

```
knn4.histogram
```

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

```
knn4_bad.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}
```

If we were working with point data, this step would be unncessary.

KnnW

If we wanted to consider only the `k`-nearest neighbors to an observation's centroid, we could use the `knnW` function in PySAL.

This specific type of distance weights requires that we first build a `KDTree`, a special representation for spatial point data. Fortunately, this is built in to PySAL:

```
kdtree = ps.cg.KDTree(centroids)
```

Then, we can use this to build a spatial weights object where only the closest `k` observations are considered "neighbors." In this example, let's do the closest 5:

```
nn5 = ps.knnW(kdtree, k=5)
```

```
nn5.histogram
```

```
[(5, 254)]
```

So, all observations have exactly 5 neighbors. Sometimes, these neighbors are actually different observations than the ones identified by contiguity neighbors.

For example, Pend Oreille gets a new neighbor, Kootenai county:

```
nn5[4]
```

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

```
dataframe.loc[nn5.neighbors[4] + [4]]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS	COFI
5	Roberts	Texas	48	393	48393	48	393
3	Hansford	Texas	48	195	48195	48	195
0	Lipscomb	Texas	48	295	48295	48	295
7	Hutchinson	Texas	48	233	48233	48	233
6	Hemphill	Texas	48	211	48211	48	211
4	Ochiltree	Texas	48	357	48357	48	357

6 rows × 70 columns

Kernel W

Kernel Weights are continuous distance-based weights that use kernel densities to provide an indication of neighborliness.

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 bandwidth can be estimated at each point or over the entire map.

For example, if we wanted to use a single estimated bandwidth for the entire map and weight according to a gaussian kernel:

```
kernelW = ps.Kernel(centroids, fixed=True, function='gaussian')
#ps.Kernel(centroids, fixed=False, function='gaussian') #same kernel, but bandwidth changes at each observation
```

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

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS	C
15	Randall	Texas	48	381	48381	48	38
16	Armstrong	Texas	48	011	48011	48	1
11	Carson	Texas	48	065	48065	48	6
10	Potter	Texas	48	375	48375	48	3
9	Moore	Texas	48	341	48341	48	3
3	Hansford	Texas	48	195	48195	48	1
7	Hutchinson	Texas	48	233	48233	48	2
1	Sherman	Texas	48	421	48421	48	4
25	Hall	Texas	48	191	48191	48	1
17	Collingsworth	Texas	48	087	48087	48	8
18	Donley	Texas	48	129	48129	48	1
13	Wheeler	Texas	48	483	48483	48	4
12	Gray	Texas	48	179	48179	48	1
6	Hemphill	Texas	48	211	48211	48	2
5	Roberts	Texas	48	393	48393	48	3
4	Ochiltree	Texas	48	357	48357	48	3
0	Lipscomb	Texas	48	295	48295	48	2
4	Ochiltree	Texas	48	357	48357	48	3

18 rows × 70 columns

As you can see, this provides our target observation, Pend Oreille, with many new neighbors. \

```
mind = ps.threshold_binaryW_from_shapefile('../data/texas.shp', radius=radius, threshold=50)
```

```
WARNING: there are 5 disconnected observations
Island ids:  [132, 133, 181, 182, 195]
```

```
mind = ps.threshold_binaryW_from_shapefile('../data/texas.shp', radius=radius, threshold=threshold)
```

```
mind.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]])
```

```
mind[0]
```

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

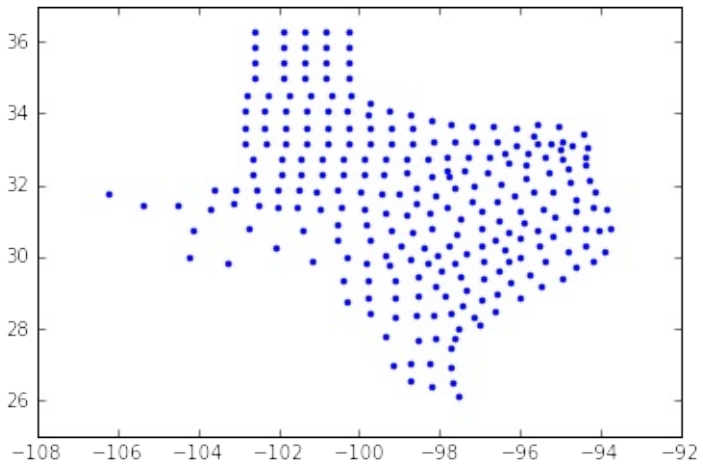
```
knn4[0]
```

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

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

```
plt.plot(centroids[:,0], centroids[:,1],'.')
plt.ylim([25,37])
```

```
(25, 37)
```



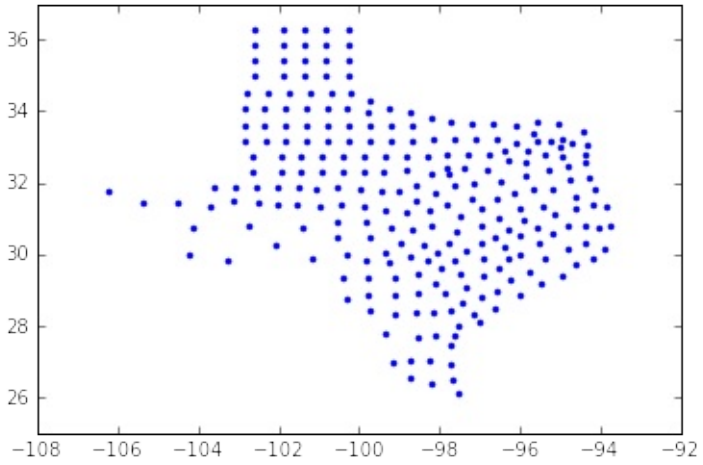

```
wq = ps.queen_from_shapefile('../data/texas.shp')
```

```
wq[0]
```

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

```
plt.plot(centroids[:,0], centroids[:,1],'.')
plt.ylim([25,37])
```

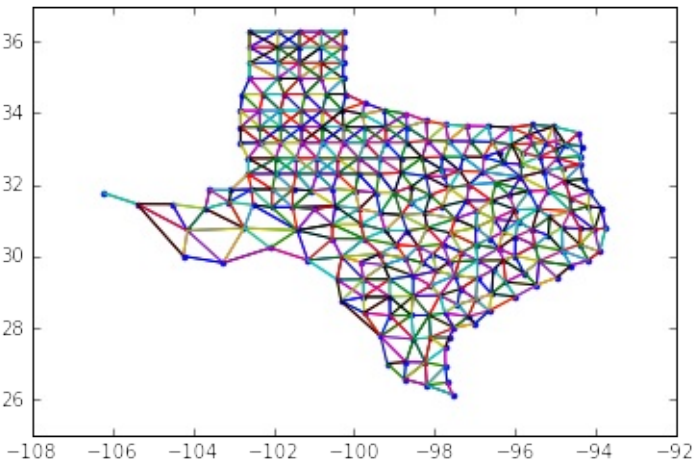
```
(25, 37)
```




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

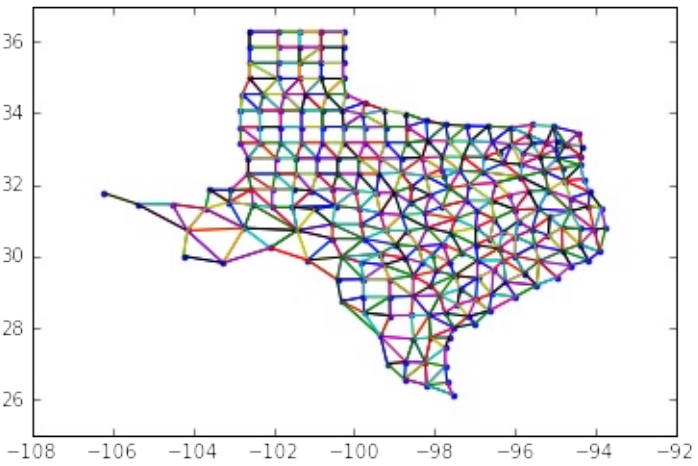
```
[4, 5, 6]
```

```
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], '-')
```

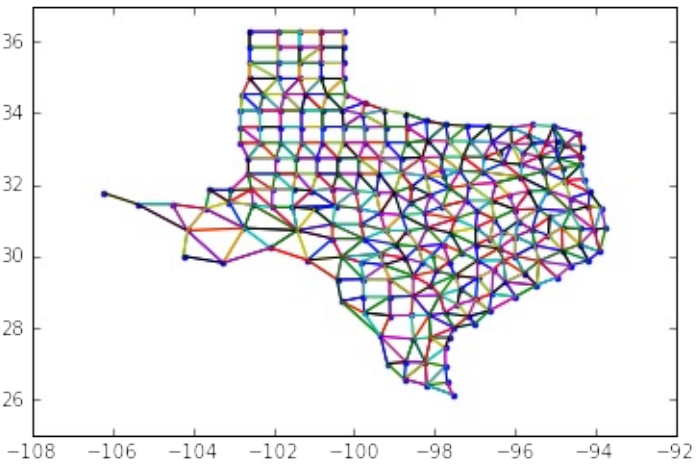



```
wr = ps.rook_from_shapefile('../data/texas.shp')
```

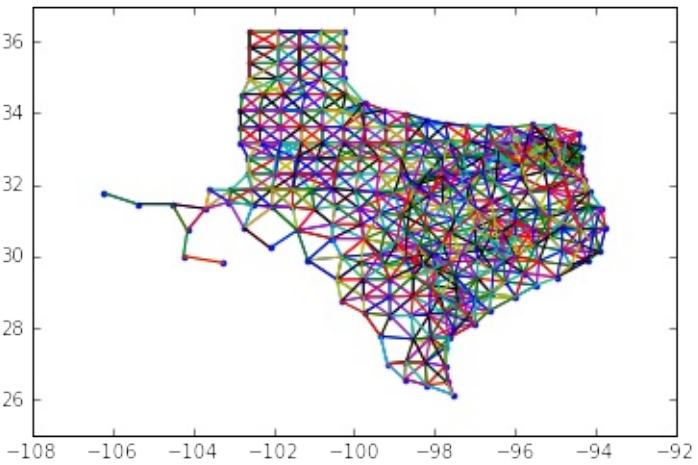
```
plt.plot(centroids[:,0], centroids[:,1],'.')
#plt.plot(s04[:,0], s04[:,1], '-')
plt.ylim([25,37])
for k,neighs in wr.neighbors.items():
    #print(k,neighs)
    origin = centroids[k]
    for neigh in neighs:
        segment = centroids[[k,neigh]]
        plt.plot(segment[:,0], segment[:,1], '-')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



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



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

```
data.head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS	COFIP
0	Lipscomb	Texas	48	295	48295	48	295
1	Sherman	Texas	48	421	48421	48	421
2	Dallam	Texas	48	111	48111	48	111
3	Hansford	Texas	48	195	48195	48	195
4	Ochiltree	Texas	48	357	48357	48	357

5 rows × 70 columns

```

```

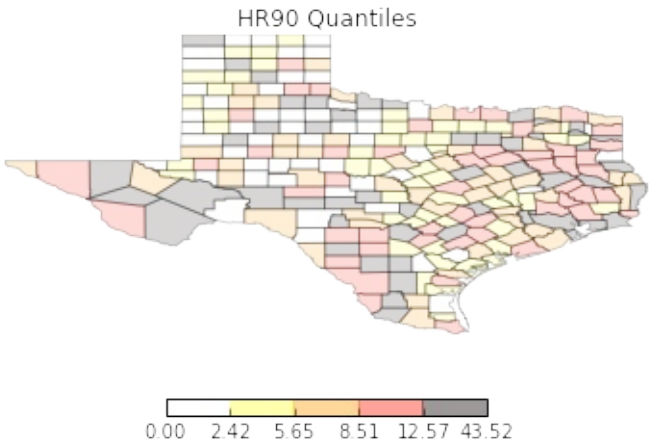
```
data.head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS	COFIP
0	Lipscomb	Texas	48	295	48295	48	295
1	Sherman	Texas	48	421	48421	48	421
2	Dallam	Texas	48	111	48111	48	111
3	Hansford	Texas	48	195	48195	48	195
4	Ochiltree	Texas	48	357	48357	48	357

5 rows × 70 columns

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

```
shp_link = "../data/texas.shp"
maps.plot_choropleth(shp_link, data.HR90, 'quantiles', title='HR90 Quantiles')
```



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, 0.014)
```

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

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

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

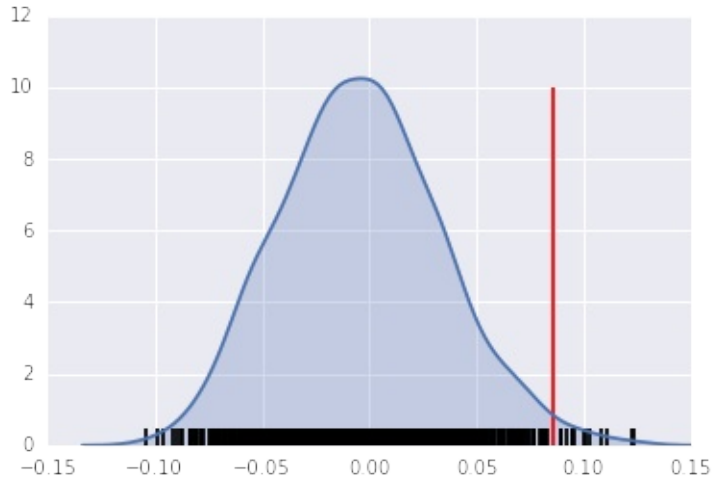
```
array([-0.04389574,  0.01129696,  0.02567161,  0.01608838, -0.04648693])
```

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])
```

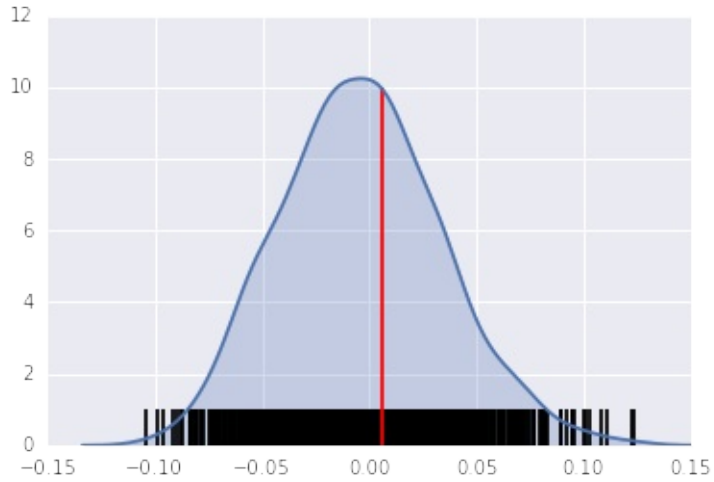
```
(-0.15, 0.15)
```



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])
```

```
(-0.15, 0.15)
```



In addition, we can compute a global Bivariate Moran statistic, which relates an observation to the spatial lag of another observation:

```
bv_HRBLK = ps.Moran_BV(data.HR90.values, data.BLK90.values, W)
```

```
bv_HRBLK.I, bv_HRBLK.p_sim
```

```
(0.080041444794258343, 0.0050000000000000001)
```

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, LMo_HR90.p_sim
```

```
(array([ 1.12087323e+00,  4.74852232e-01, -1.22758423e+00,
        9.38686608e-01,  6.89742960e-01,  7.85031726e-01,
        7.10475151e-01,  4.10606860e-01,  7.40368367e-03,
        1.48663520e-01, -5.99072474e-01,  8.11752611e-02,
       -1.20482997e-01,  3.18800252e-01,  3.47276792e-01,
        4.39609038e-02, -3.04514513e-01,  2.01610062e-01,
        2.85061414e-01,  5.84462378e-01,  7.56336432e-01,
        3.83077442e-01,  3.99898754e-03, -9.16855756e-01,
       -4.87000938e-02,  1.46381951e-02, -1.20220927e-01,
       -7.65176192e-01, -2.57853261e+00,  2.03465774e-01,
       -1.00025313e-02,  1.51706786e-01,  4.93433817e-01,
        4.35124305e-01, -3.12259940e-01, -2.32723324e-01,
        2.82455741e-03,  1.22002377e-01, -1.27741850e-02,
       -3.54088100e-04, -9.99830522e-03, -4.29411693e-02,
        1.14955949e-01, -5.76707109e-01, -5.79367567e-01,
       -2.44357064e-01, -2.45937883e-02,  1.28248005e-01,
        2.40904907e-01, -1.30657056e+00,  8.75684587e-02,
        7.60257313e-01,  7.46769544e-03,  8.46525604e-02,
       -9.91242446e-02, -3.63476666e-03, -1.84994288e-01,
        6.10806717e-02, -4.10962997e-01, -1.55435599e-02,
        3.34932522e-01,  5.84998647e-01,  3.97457030e-01,
        4.84549106e-01, -7.05237462e-01, -1.89267505e-01,
       -5.36300770e-01,  2.18228666e-01,  2.45387767e-01,
        1.04747657e+00, -5.03740235e-02, -4.99501600e-01,
       -6.79039774e-02, -4.28204631e-01, -5.19255631e-02,
        1.62667971e-01,  3.96182300e-02,  1.39001221e-01,
        7.85033384e-01, -6.34873174e-02, -2.05239166e-01,
        6.70092095e-01,  1.19432435e-01,  8.33565709e-02,
        6.08109031e-02,  4.71129836e-02, -7.53103370e-01,
        4.12179684e-01,  4.18690212e-01, -1.24823445e-01,
       -2.20055897e-01,  8.71366261e-01, -8.79710509e-02,
        1.68820593e-03,  1.07166595e-02, -3.54370029e-02,
        1.20592772e-03,  2.34359115e-01,  1.74921357e-01,
       -1.82542919e-01, -2.17921039e-01, -1.29920391e-03,
       -5.76690555e-02,  1.45723641e-01,  3.62521949e-01,
        4.39212267e-02,  1.62791100e-01,  2.34754317e-01,
        4.24252755e-02, -7.44382557e-01,  1.01380671e-01,
        1.68446607e-01,  1.36810620e-01,  4.56977527e-02,
        6.92868233e-02,  3.82546883e-02,  1.23206557e-01,
        5.81014061e-01,  6.19044577e-01,  5.97840100e-02,
        6.02552478e-01,  1.98331508e-01, -2.08124490e-01,
        2.89307856e-01,  2.94694807e-01,  3.17901639e-01,
        3.26156089e-01,  3.01034558e-01,  1.49504893e-01,
        1.95148623e+00, -2.54303828e-01,  5.08956169e-01,
        6.08264082e-01, -3.81888449e-02, -3.00282633e-01,
       -2.65739969e-02,  2.24230408e-01, -1.44008615e-02,
        4.66594113e-01, -4.31489357e-02,  1.87061533e-02,
        1.45607307e-01,  2.47182058e-03, -6.64057756e-01,
        1.54192078e-02,  2.36636263e-01, -8.45056351e-03,
       -1.34851321e-01,  1.18537899e-01, -2.27964598e-01,
       -4.18648686e-01,  3.39507930e-01, -2.63719989e-02,
        1.55681623e-01,  1.22301226e-01,  1.77438253e-01,
        4.10677287e-02,  1.49101636e-01,  1.04297773e-01,
       -7.67634652e-02, -5.64248729e-03, -3.05586580e-02,
        5.53503759e-02,  1.37140043e-02,  1.29212068e+00,
       -8.56707799e-02,  2.62786886e-01, -6.36623531e-01,
       -3.90746796e-01,  2.74114791e-02, -3.42998234e-02,
        1.96470454e-02,  2.66085647e-01, -5.86103398e-01,
       -1.40050219e-01, -9.58183688e-02,  1.06002105e-01,
        5.78305864e-02,  9.12034797e-02,  1.75316202e-01,
        1.53146979e+00,  2.55223569e-01, -1.07521719e+00,
```

```
        2.10421553e-01,   1.79559685e-03,   2.46623711e-01,
       -1.37221563e-01,  -1.93104397e-01,  -6.70944099e-01,
       -6.21994147e-01,   2.62134357e-01,   8.44258361e-02,
        2.15034457e-03,  -1.13695914e-02,  -9.85881978e-01,
       -3.88051312e-02,  -3.94113412e-02,   2.54665339e-01,
        1.03033517e-01,   1.87014888e-01,   6.57921237e-02,
        8.93885114e-01,  -4.12463002e-02,   5.93523798e-04,
       -7.48641130e-01,  -2.33977229e-01,  -1.81364976e-01,
       -2.12719096e-02,  -1.77624803e-01,   7.56458111e-01,
        4.65743492e-02,  -1.17037830e-02,   1.42744115e-01,
       -2.52132245e-01,   2.20273885e-02,   1.32249407e-01,
        1.50840035e-02,   1.58872364e-01,  -3.07330315e-01,
       -2.15826555e-01,   7.78727321e-01,   1.53798953e-01,
        1.39830232e-02,   1.14430614e-02,  -4.13603775e-01,
       -1.13601903e-01,   3.98036147e-02,   5.00495648e-02,
        1.85345062e-02,   8.20513963e-01,  -2.85154006e-03,
        3.37194545e-01,  -6.54881438e-01,   2.87571205e-01,
        3.40025967e+00,   2.79223250e+00,   2.18713536e-01,
        2.57852929e-01,   1.00405907e-01,   1.92499398e-01,
        9.07540052e-02,   1.49438551e-01,   1.33219486e+00,
        9.58722126e-03,  -9.26242513e-02,   1.26208606e-01,
        3.53039195e-01,   1.91842139e-01,  -2.84025227e-03,
        2.62219645e-01,   1.87256013e-01,   2.39837728e-03,
        8.94400168e-02,  -1.39329420e-01]),
 array([ 0.009,  0.173,  0.029,  0.009,  0.001,  0.008,  0.064,  0.057,
         0.434,  0.135,  0.074,  0.315,  0.043,  0.009,  0.254,  0.45 ,
         0.238,  0.393,  0.262,  0.015,  0.055,  0.051,  0.263,  0.035,
         0.444,  0.129,  0.169,  0.124,  0.061,  0.242,  0.319,  0.092,
         0.031,  0.009,  0.22 ,  0.283,  0.431,  0.247,  0.497,  0.424,
         0.364,  0.172,  0.377,  0.097,  0.082,  0.238,  0.074,  0.271,
         0.038,  0.009,  0.466,  0.04 ,  0.479,  0.374,  0.034,  0.465,
         0.149,  0.108,  0.079,  0.467,  0.21 ,  0.068,  0.037,  0.107,
         0.035,  0.188,  0.003,  0.308,  0.053,  0.016,  0.302,  0.126,
         0.42 ,  0.209,  0.371,  0.134,  0.426,  0.325,  0.191,  0.045,
         0.359,  0.06 ,  0.139,  0.443,  0.143,  0.483,  0.014,  0.041,
         0.207,  0.264,  0.279,  0.145,  0.133,  0.432,  0.435,  0.457,
         0.463,  0.144,  0.027,  0.028,  0.069,  0.02 ,  0.041,  0.149,
         0.204,  0.289,  0.067,  0.256,  0.156,  0.124,  0.072,  0.2  ,
         0.141,  0.28 ,  0.061,  0.371,  0.07 ,  0.073,  0.061,  0.071,
         0.081,  0.273,  0.142,  0.273,  0.125,  0.058,  0.11 ,  0.05 ,
         0.256,  0.168,  0.033,  0.226,  0.016,  0.251,  0.414,  0.476,
         0.1  ,  0.209,  0.167,  0.383,  0.005,  0.201,  0.115,  0.108,
         0.397,  0.289,  0.301,  0.094,  0.404,  0.044,  0.227,  0.146,
         0.262,  0.4  ,  0.139,  0.17 ,  0.077,  0.27 ,  0.127,  0.379,
         0.438,  0.234,  0.238,  0.427,  0.011,  0.384,  0.052,  0.102,
         0.322,  0.347,  0.263,  0.483,  0.08 ,  0.174,  0.299,  0.329,
         0.299,  0.413,  0.442,  0.352,  0.065,  0.265,  0.045,  0.073,
         0.45 ,  0.096,  0.319,  0.184,  0.099,  0.06 ,  0.059,  0.221,
         0.466,  0.441,  0.026,  0.305,  0.222,  0.201,  0.203,  0.285,
         0.173,  0.116,  0.373,  0.379,  0.323,  0.174,  0.213,  0.399,
         0.07 ,  0.023,  0.165,  0.141,  0.358,  0.331,  0.079,  0.246,
         0.41 ,  0.165,  0.257,  0.102,  0.099,  0.12 ,  0.307,  0.488,
         0.005,  0.287,  0.418,  0.203,  0.434,  0.002,  0.192,  0.242,
         0.063,  0.039,  0.013,  0.059,  0.012,  0.108,  0.331,  0.001,
         0.119,  0.241,  0.012,  0.489,  0.314,  0.245,  0.117,  0.152,
         0.36 ,  0.354,  0.251,  0.485,  0.378,  0.41 ]))
```

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]
```

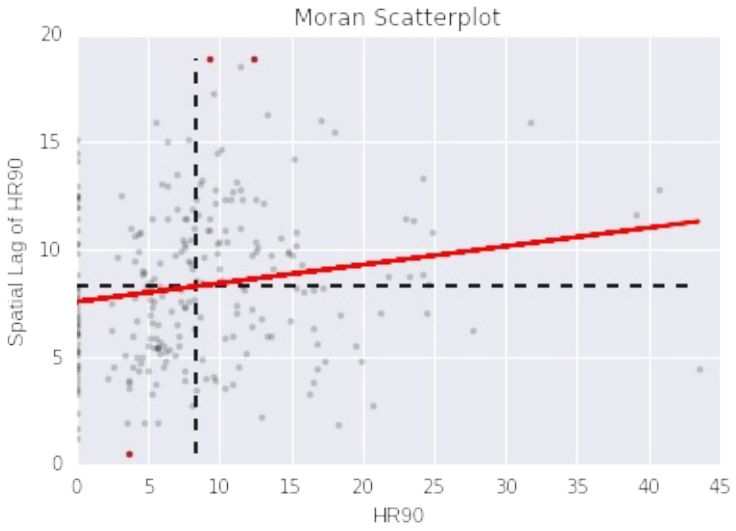
Then, since we have a lot of points, we can plot the points with a statistically insignficant LISA value lighter using the `alpha` keyword. In addition, we would like to plot the statistically significant points in a dark red color.

```
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 0x7f5a57011e10>
```



We can also make a LISA map of the data.

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

```
sig.sum()
```

```
42
```

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

```
hotspots.sum()
```

```
9
```

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

```
coldspots.sum()
```

```
16
```

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

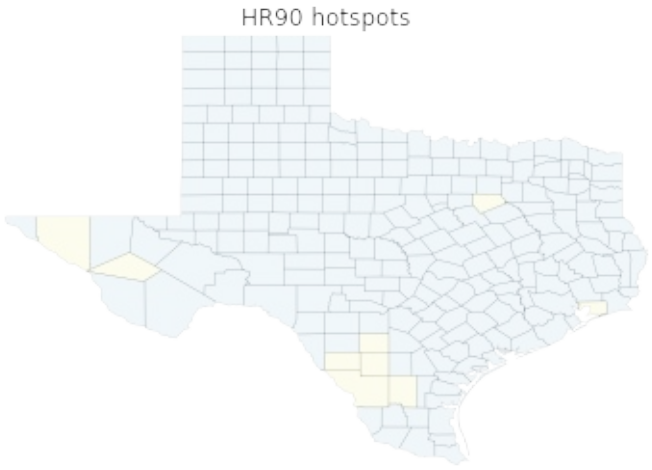
```
98     9.784698
132    11.435106
164    17.129154
209    13.274924
229    12.371338
234    31.721863
236     9.584971
239     9.256549
242    18.062652
Name: HR90, dtype: float64
```

```
data[hotspots]
```

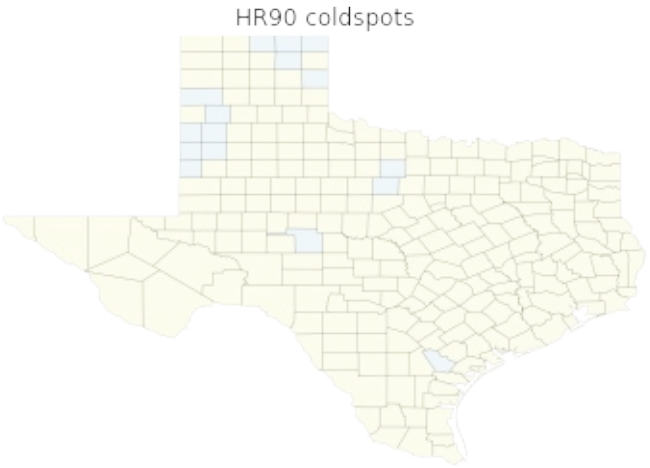
	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS	STFIPS	COI
98	Ellis	Texas	48	139	48139	48	139
132	Hudspeth	Texas	48	229	48229	48	229
164	Jeff Davis	Texas	48	243	48243	48	243
209	Chambers	Texas	48	071	48071	48	71
229	Frio	Texas	48	163	48163	48	163
234	La Salle	Texas	48	283	48283	48	283
236	Dimmit	Texas	48	127	48127	48	127
239	Webb	Texas	48	479	48479	48	479
242	Duval	Texas	48	131	48131	48	131

9 rows × 70 columns

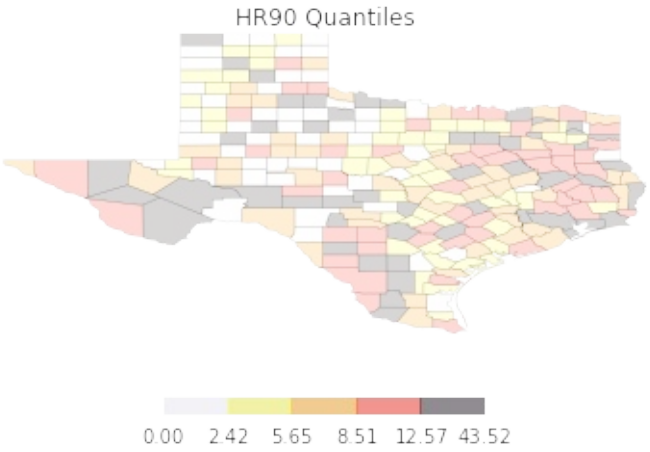
```
maps.plot_choropleth(shp_link, hotspots*1., 'unique_values', title='HR90 hotspots')
```




```
maps.plot_choropleth(shp_link, coldspots*1., 'unique_values', title='HR90 coldspots')
```

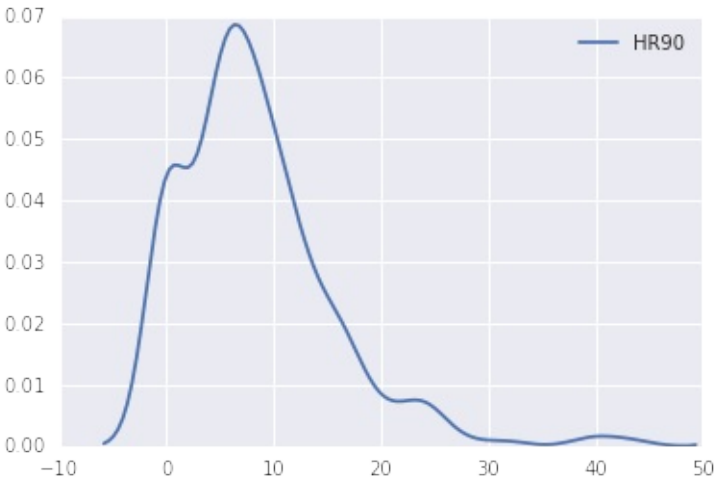



```
maps.plot_choropleth(shp_link, data.HR90, 'quantiles', title='HR90 Quantiles')
```




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

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5a5a448e90>
```




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

```
8.302494460285041
```

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

```
7.23234613355
```

```
import pysal as ps
import numpy as np
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
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
```

```
['Name',
 'STATE_FIPS',
 '1929',
 '1930',
 '1931',
 '1932',
 '1933',
 '1934',
 '1935',
 '1936',
 '1937',
 '1938',
 '1939',
 '1940',
 '1941',
 '1942',
 '1943',
 '1944',
 '1945',
 '1946',
 '1947',
 '1948',
 '1949',
 '1950',
 '1951',
 '1952',
 '1953',
 '1954',
 '1955',
 '1956',
 '1957',
 '1958',
 '1959',
 '1960',
 '1961',
 '1962',
 '1963',
 '1964',
 '1965',
 '1966',
 '1967',
 '1968',
 '1969',
 '1970',
 '1971',
 '1972',
 '1973',
 '1974',
 '1975',
 '1976',
 '1977',
 '1978',
 '1979',
```

```
 '1980',
 '1981',
 '1982',
 '1983',
 '1984',
 '1985',
 '1986',
 '1987',
 '1988',
 '1989',
 '1990',
 '1991',
 '1992',
 '1993',
 '1994',
 '1995',
 '1996',
 '1997',
 '1998',
 '1999',
 '2000',
 '2001',
 '2002',
 '2003',
 '2004',
 '2005',
 '2006',
 '2007',
 '2008',
 '2009']
```

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
```

```
[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]
```

And now 2009

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

```
y2009
```

```
[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]
```

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()
```

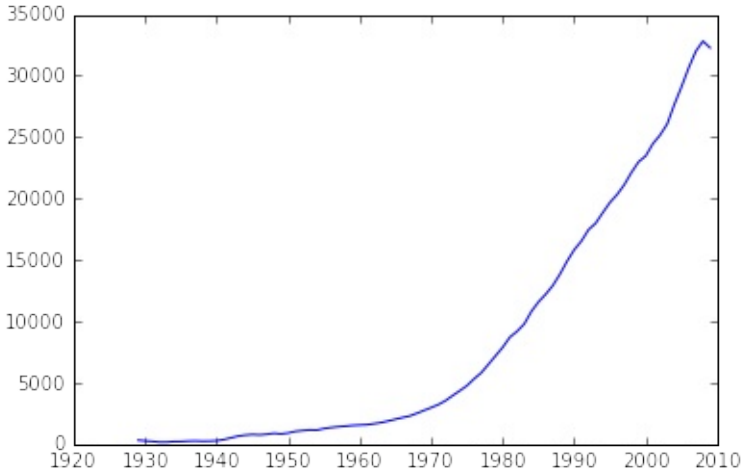
```
Y.shape
```

```
(48, 81)
```

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

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

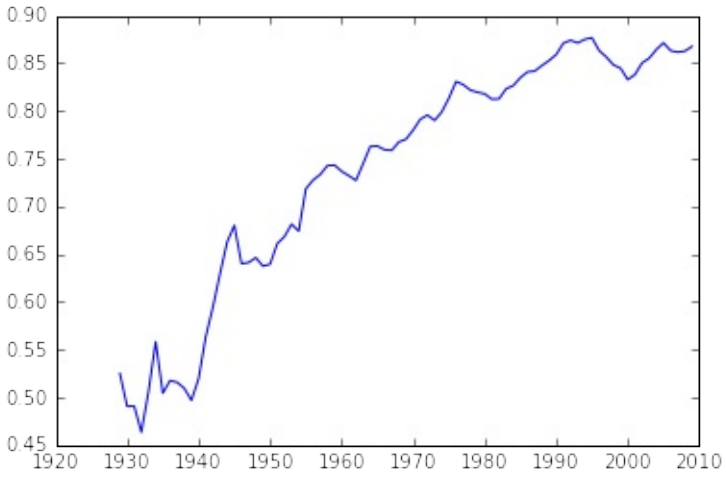
```
[<matplotlib.lines.Line2D at 0x7fcee352a1d0>]
```




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

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

```
[<matplotlib.lines.Line2D at 0x7fcee342d690>]
```



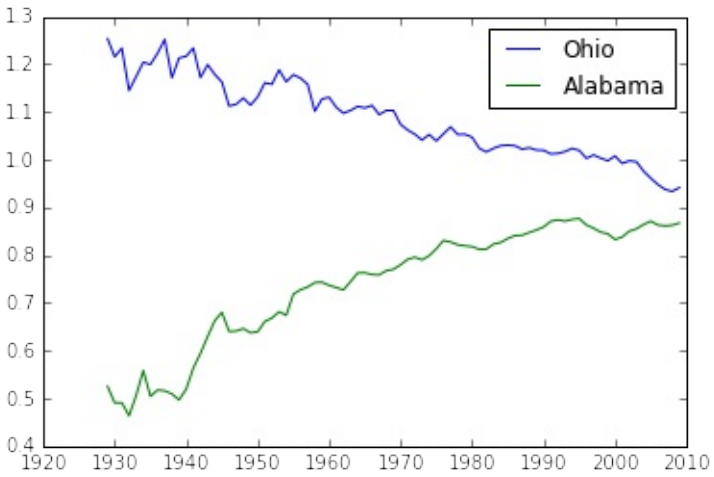

```
name = np.array(name)
```

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

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

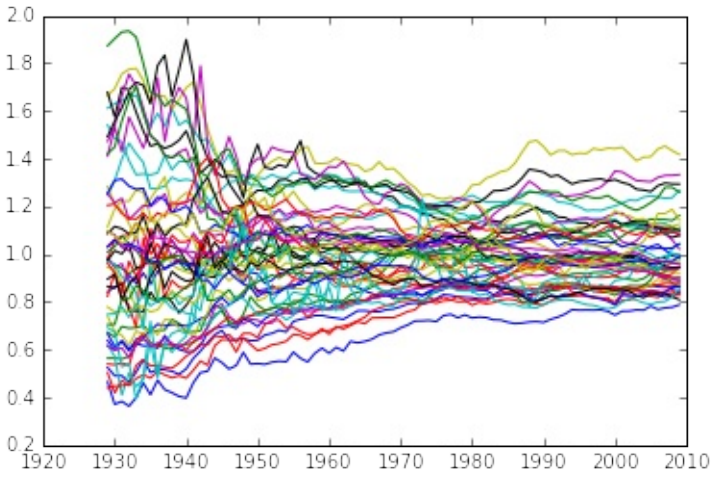
```
plot(years, RY[32], label='Ohio')
plot(years, RY[0], label='Alabama')
legend()
```

```
<matplotlib.legend.Legend at 0x7fcee3436890>
```



Spaghetti Plot

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



Kernel Density (univariate, aspatial)

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

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

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

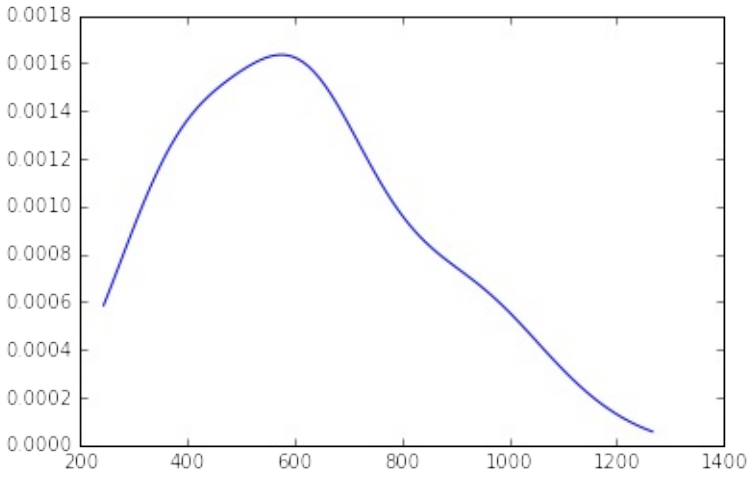
```
array([  323.,   600.,   310.,   991.,   634.,  1024.,  1032.,   518.,
         347.,   507.,   948.,   607.,   581.,   532.,   393.,   414.,
         601.,   768.,   906.,   790.,   599.,   286.,   621.,   592.,
         596.,   868.,   686.,   918.,   410.,  1152.,   332.,   382.,
         771.,   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)
```

```
plot(x,density(x))
```

```
[<matplotlib.lines.Line2D at 0x7fcee3050350>]
```

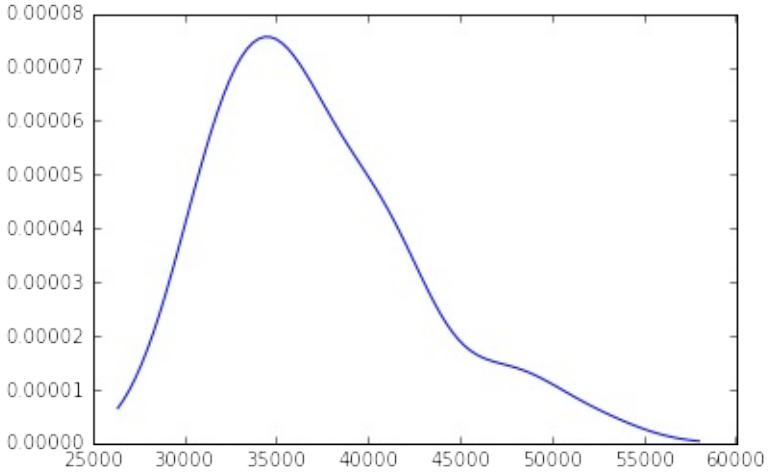



```
d2009 = gaussian_kde(Y[:,-1])
```

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

```
plot(x,d2009(x))
```

```
[<matplotlib.lines.Line2D at 0x7fcee178a5d0>]
```




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

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

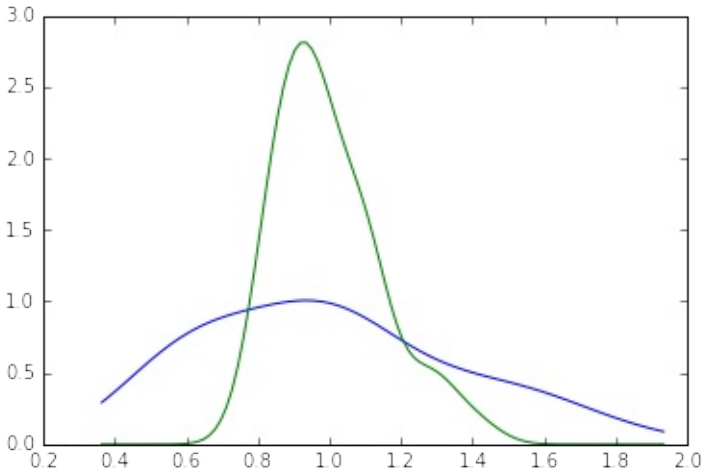
```
x = np.linspace(minR0, maxR0, 100)
```

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

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

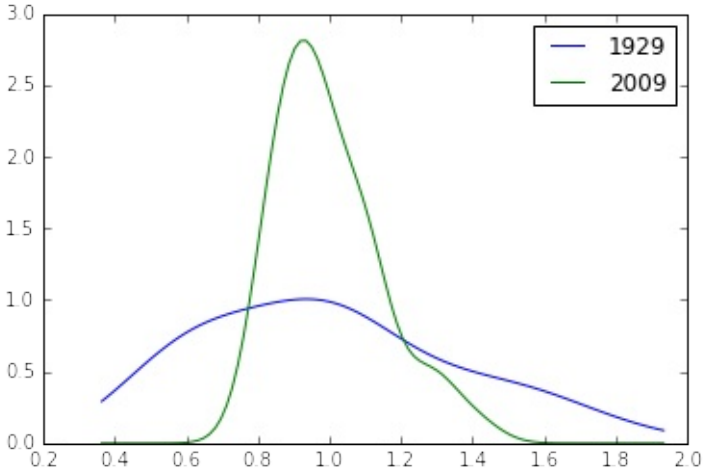
```
plot(x, d1929(x))
plot(x, d2009(x))
```

```
[<matplotlib.lines.Line2D at 0x7fcee17c3c50>]
```

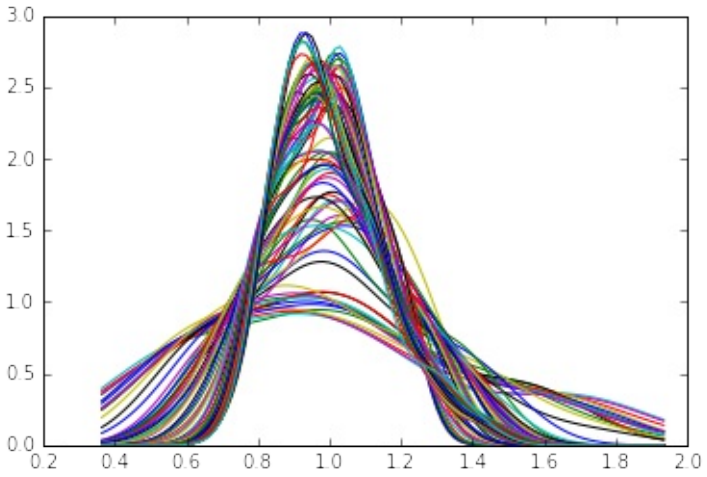



```
plot(x, d1929(x), label='1929')
plot(x, d2009(x), label='2009')
legend()
```

```
<matplotlib.legend.Legend at 0x7fcee3161b50>
```

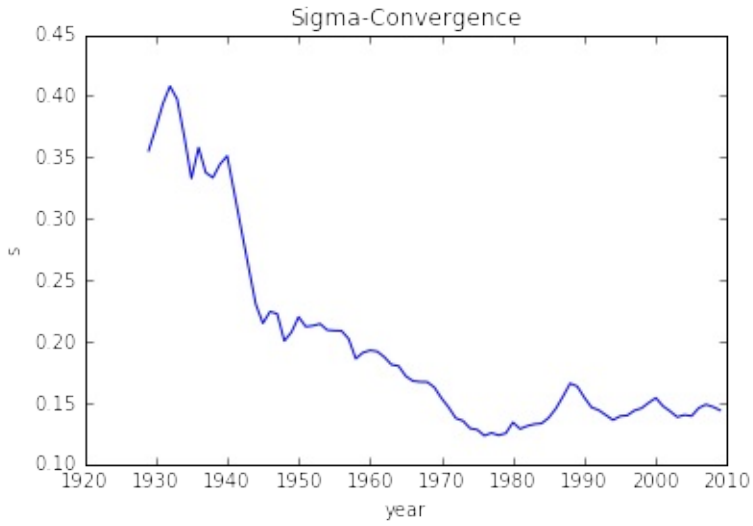



```
for cs in RY.T: # take cross sections
    plot(x, gaussian_kde(cs)(x))
```




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

```
<matplotlib.text.Text at 0x7fcee14d13d0>
```



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='|S1')
```

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

```
m.classes
```

```
array(['a', 'b', 'c'],
      dtype='|S1')
```

```
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

```
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)
```

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

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

```
q5.shape
```

```
(48, 81)
```

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

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

```
pci.shape
```

```
(81, 48)
```

```
pci[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])
```

we are looping over the rows of y which is ordered Txn (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.]])
```

```
m5.p
```

```
matrix([[ 0.91011236,  0.0886392 ,  0.00124844,  0.        ,  0.        ],
        [ 0.09972299,  0.78531856,  0.11080332,  0.00415512,  0.        ],
        [ 0.        ,  0.10125   ,  0.78875   ,  0.1075    ,  0.0025    ],
        [ 0.        ,  0.00417827,  0.11977716,  0.79805014,  0.07799443],
        [ 0.        ,  0.        ,  0.00125156,  0.07133917,  0.92740926]])
```

```
m5.steady_state
```

```
matrix([[ 0.20774716],
        [ 0.18725774],
        [ 0.20740537],
        [ 0.18821787],
        [ 0.20937187]])
```

```
fmpt = ps.ergodic.fmpt(m5.p)
```

```
fmpt
```

```
matrix([[  4.81354357,  11.50292712,  29.60921231,  53.38594954,
        103.59816743],
       [ 42.04774505,   5.34023324,  18.74455332,  42.50023268,
         92.71316899],
       [ 69.25849753,  27.21075248,   4.82147603,  25.27184624,
         75.43305672],
       [ 84.90689329,  42.85914824,  17.18082642,   5.31299186,
         51.60953369],
       [ 98.41295543,  56.36521038,  30.66046735,  14.21158356,
          4.77619083]])
```

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

Read in a GAL file to construct our W

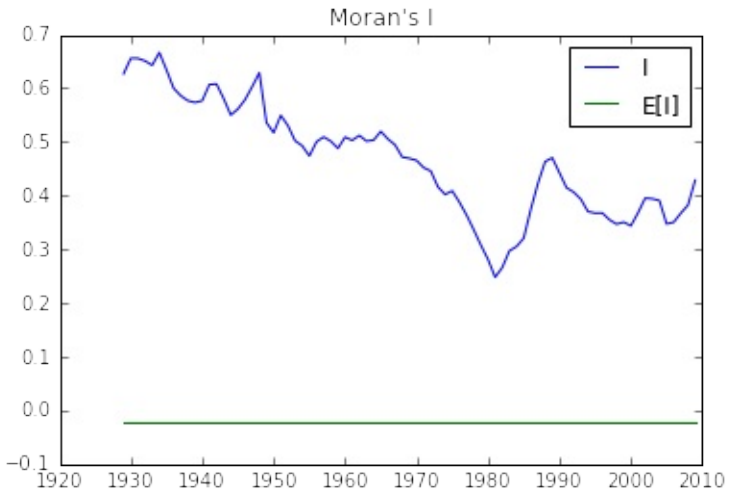
```
w = ps.open(ps.examples.get_path("states48.gal")).read()

w.n

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])
```

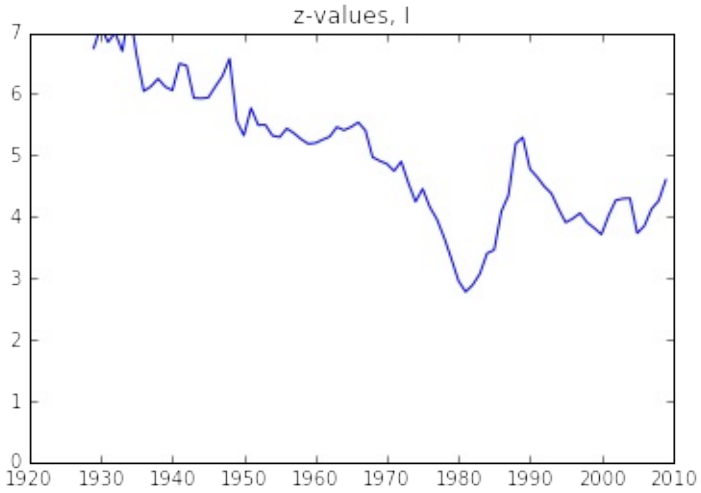
```
plot(years, res[:,0], label='I')
plot(years, res[:,1], label='E[I]')
title("Moran's I")
legend()
```

```
<matplotlib.legend.Legend at 0x7fcee13b3c90>
```



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

```
<matplotlib.text.Text at 0x7fcee12e2a90>
```



Spatial Markov

```
pci.shape
```

```
(81, 48)
```

```
pci = pci.T
```

```
pci.shape
```

```
(48, 81)
```

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

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

```
array([ 0.5250254 ,  0.97527938,  0.50389434,  1.61083644,  1.03054521,
        1.6644768 ,  1.67748053,  0.8419912 ,  0.56403657,  0.82411107,
        1.54094142,  0.98665764,  0.94439553,  0.86474771,  0.63880799,
        0.67294277,  0.97690484,  1.2483576 ,  1.47267186,  1.28411785,
        0.97365391,  0.46488317,  1.00941416,  0.96227565,  0.96877751,
        1.41090417,  1.11506942,  1.49217745,  0.66644091,  1.8725364 ,
        0.53965459,  0.62092787,  1.253234  ,  0.73958686,  1.08581104,
        1.25485946,  1.42065696,  0.44050119,  0.69244836,  0.61442601,
        0.77859804,  0.89563156,  1.03054521,  0.70545208,  1.20447003,
        0.74771419,  1.09393837,  1.0971893 ])
```

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

```
0.99999999999999989
```

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

```
sm.p
```

```
matrix([[ 0.91461837,  0.07503234,  0.00905563,  0.00129366,  0.        ],
        [ 0.06570302,  0.82654402,  0.10512484,  0.00131406,  0.00131406],
        [ 0.00520833,  0.10286458,  0.79427083,  0.09505208,  0.00260417],
        [ 0.        ,  0.00913838,  0.09399478,  0.84856397,  0.04830287],
        [ 0.        ,  0.        ,  0.        ,  0.06217617,  0.93782383]])
```

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

```
[[ 0.96341463  0.0304878   0.00609756  0.        0.        ]
 [ 0.06040268  0.83221477  0.10738255  0.        0.        ]
 [ 0.        0.14        0.74        0.12        0.        ]
 [ 0.        0.03571429  0.32142857  0.57142857  0.07142857]
 [ 0.        0.        0.        0.16666667  0.83333333]]
[[ 0.79831933  0.16806723  0.03361345  0.        0.        ]
 [ 0.0754717   0.88207547  0.04245283  0.        0.        ]
 [ 0.00537634  0.06989247  0.8655914   0.05913978  0.        ]
 [ 0.        0.        0.06372549  0.90196078  0.03431373]
 [ 0.        0.        0.        0.19444444  0.80555556]]
[[ 0.84693878  0.15306122  0.        0.        0.        ]
 [ 0.08133971  0.78947368  0.1291866   0.        0.        ]
 [ 0.00518135  0.0984456   0.79274611  0.0984456   0.00518135]
 [ 0.        0.        0.09411765  0.87058824  0.03529412]
 [ 0.        0.        0.        0.10204082  0.89795918]]
[[ 0.8852459   0.09836066  0.        0.01639344  0.        ]
 [ 0.03875969  0.81395349  0.13953488  0.        0.00775194]
 [ 0.0049505   0.09405941  0.77722772  0.11881188  0.0049505 ]
 [ 0.        0.02339181  0.12865497  0.75438596  0.09356725]
 [ 0.        0.        0.        0.09661836  0.90338164]]
[[ 0.33333333  0.66666667  0.        0.        0.        ]
 [ 0.0483871   0.77419355  0.16129032  0.01612903  0.        ]
 [ 0.01149425  0.16091954  0.74712644  0.08045977  0.        ]
 [ 0.        0.01036269  0.06217617  0.89637306  0.03108808]
 [ 0.        0.        0.        0.02352941  0.97647059]]
```

```
sm.S
```

```
array([[ 0.43509425,  0.2635327 ,  0.20363044,  0.06841983,  0.02932278],
       [ 0.13391287,  0.33993305,  0.25153036,  0.23343016,  0.04119356],
       [ 0.12124869,  0.21137444,  0.2635101 ,  0.29013417,  0.1137326 ],
       [ 0.0776413 ,  0.19748806,  0.25352636,  0.22480415,  0.24654013],
       [ 0.01776781,  0.19964349,  0.19009833,  0.25524697,  0.3372434 ]])
```

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

```
[[  2.29835259  28.95614035  46.14285714  80.80952381 279.42857143]
 [ 33.86549708   3.79459555  22.57142857  57.23809524 255.85714286]
 [ 43.60233918   9.73684211   4.91085714  34.66666667 233.28571429]
 [ 46.62865497  12.76315789   6.25714286  14.61564626 198.61904762]
 [ 52.62865497  18.76315789  12.25714286   6.         34.1031746 ]]
[[  7.46754205   9.70574606  25.76785714  74.53116883 194.23446197]
 [ 27.76691978   2.94175577  24.97142857  73.73474026 193.4380334 ]
 [ 53.57477715  28.48447637   3.97566318  48.76331169 168.46660482]
 [ 72.03631562  46.94601483  18.46153846   4.28393653 119.70329314]
 [ 77.17917276  52.08887197  23.6043956    5.14285714  24.27564033]]
[[  8.24751154   6.53333333  18.38765432  40.70864198 112.76732026]
 [ 47.35040872   4.73094099  11.85432099  34.17530864 106.23398693]
 [ 69.42288828  24.76666667   3.794921    22.32098765  94.37966594]
 [ 83.72288828  39.06666667  14.3         3.44668119  76.36702977]
 [ 93.52288828  48.86666667  24.1         9.8         8.79255406]]
[[ 12.87974382  13.34847151  19.83446328  28.47257282  55.82395142]
 [ 99.46114206   5.06359731  10.54545198  23.05133495  49.68944423]
 [117.76777159  23.03735526   3.94436301  15.0843986   43.57927247]
 [127.89752089  32.4393006   14.56853107   4.44831643  31.63099455]
 [138.24752089  42.7893006   24.91853107  10.35         4.05613474]]
[[ 56.2815534   1.5         10.57236842  27.02173913 110.54347826]
 [ 82.9223301   5.00892857   9.07236842  25.52173913 109.04347826]
 [ 97.17718447  19.53125     5.26043557  21.42391304 104.94565217]
 [127.1407767   48.74107143  33.29605263   3.91777427  83.52173913]
 [169.6407767   91.24107143  75.79605263  42.5         2.96521739]]
```

LISA Markov

```
lm = ps.LISA_Markov(pci,w)
```

```
lm.classes
```

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

```
lm.transitions
```

```
array([[  1.08700000e+03,   4.40000000e+01,   4.00000000e+00,
          3.40000000e+01],
       [  4.10000000e+01,   4.70000000e+02,   3.60000000e+01,
          1.00000000e+00],
       [  5.00000000e+00,   3.40000000e+01,   1.42200000e+03,
          3.90000000e+01],
       [  3.00000000e+01,   1.00000000e+00,   4.00000000e+01,
          5.52000000e+02]])
```

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

```
lm.transitions
```

```
array([[ 1087.,    44.,     4.,    34.],
       [   41.,   470.,    36.,     1.],
       [    5.,    34.,  1422.,    39.],
       [   30.,     1.,    40.,   552.]])
```

```
lm.p
```

```
matrix([[ 0.93 ,  0.038,  0.003,  0.029],
       [ 0.075,  0.858,  0.066,  0.002],
       [ 0.003,  0.023,  0.948,  0.026],
       [ 0.048,  0.002,  0.064,  0.886]])
```

```
lm.steady_state
```

```
matrix([[ 0.286],
       [ 0.142],
       [ 0.405],
       [ 0.168]])
```

```
ps.ergodic.fmpt(lm.p)
```

```
matrix([[  3.501,  37.93 ,  40.558,  43.174],
       [ 31.728,   7.047,  28.682,  49.915],
       [ 52.445,  47.421,   2.47 ,  43.756],
       [ 38.768,  51.518,  26.316,   5.969]])
```

Test of independence of own chains and lag chains

```
lm.transitions
```

```
array([[ 1087.,    44.,     4.,    34.],
     [   41.,   470.,    36.,     1.],
     [    5.,    34.,  1422.,    39.],
     [   30.,     1.,    40.,   552.]])
```

```
lm.expected_t
```

```
array([[ 1123.281,    11.538,     0.348,    33.834],
     [    3.503,   528.474,    15.918,     0.106],
     [    0.154,    23.216,  1466.907,     9.723],
     [    9.608,     0.099,     6.235,   607.058]])
```

```
lm.chi_2
```

```
(1058.2079036003051, 0.0, 9)
```

Part II

Point Patterns

IPYNB

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
```

```
/home/dani/anaconda/envs/pydata/lib/python2.7/site-packages/matplotlib/font_manager.py:273: UserWarning: Matplotlib is building the fon
t cache using fc-list. This may take a moment.
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a moment.')
```

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
```

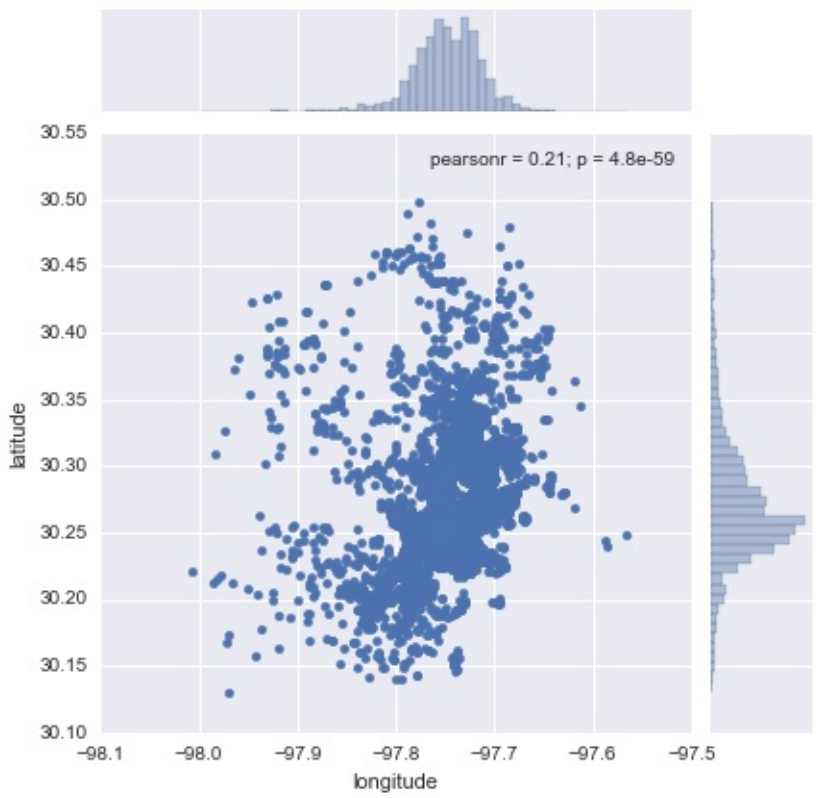
	longitude	latitude
5832	-5.093682	43.214991

```
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:

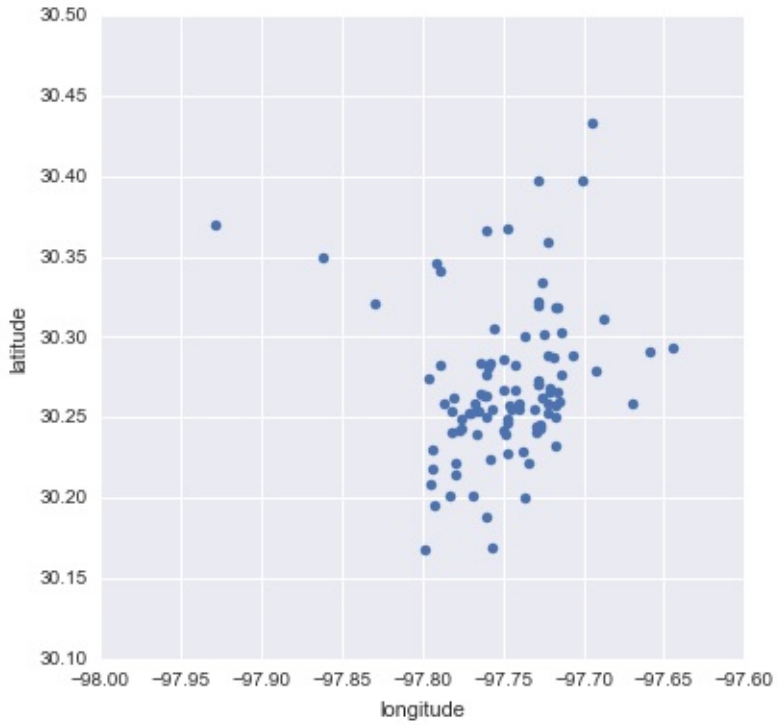
```
sns.jointplot(x="longitude", y="latitude", data=lst);
```



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 patter 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 bigh 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`:

```
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, **plot_args):
    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)
```

```
<div class="bk-banner">
    <a href="http://bokeh.pydata.org" target="_blank" class="bk-logo bk-logo-small bk-logo-notebook"></a>
    <span id="a0125a61-aa40-4d88-9db2-d2b800c39acb">Loading BokehJS ...</span>
</div>
```

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 0x7f6c701fa7d0>
```

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_range)
    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.

Centrography and distance based statistics

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 marketting. 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(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:

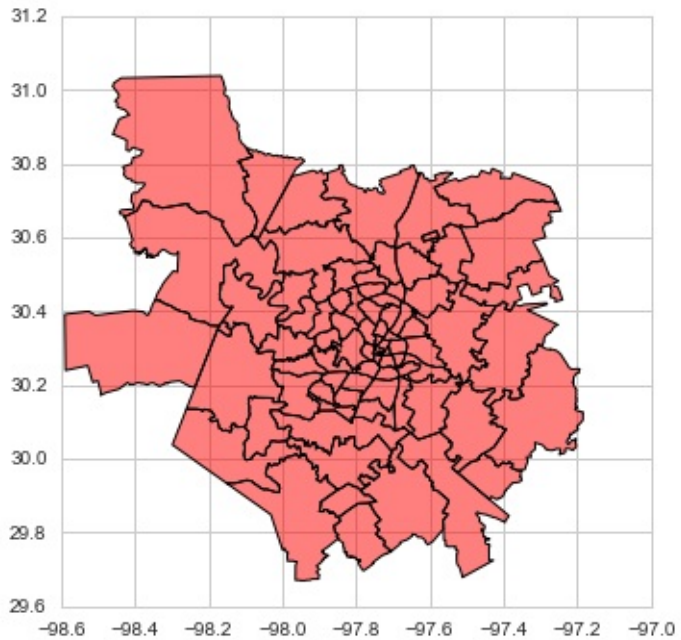
```
aves_props = aves.join(prop_types_pct)
```

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');
```

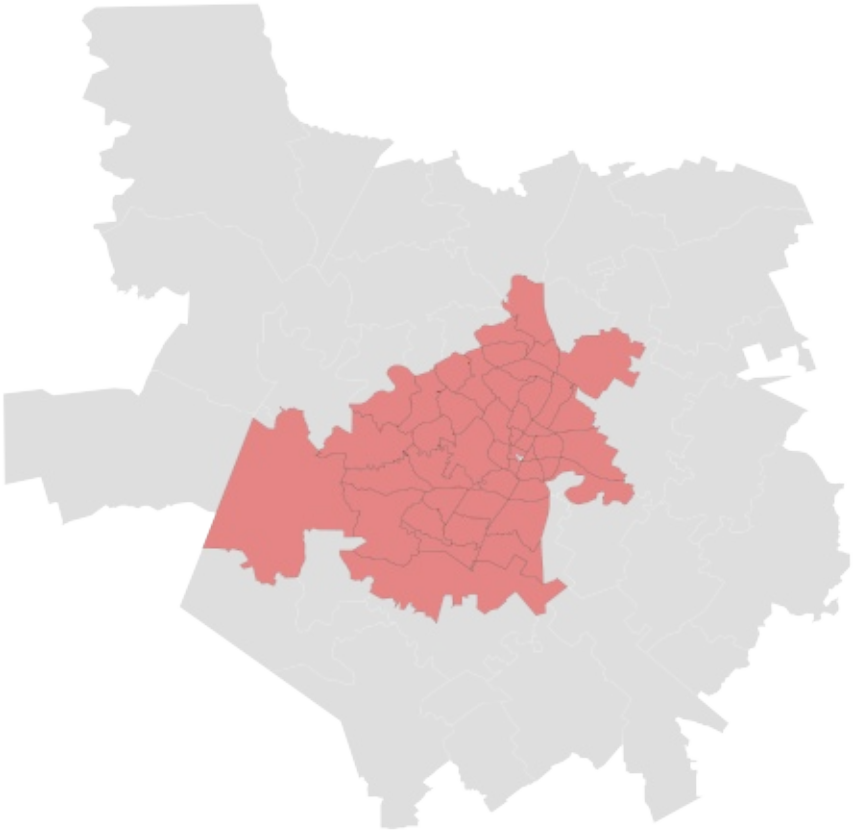


And combine the two:

```
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:

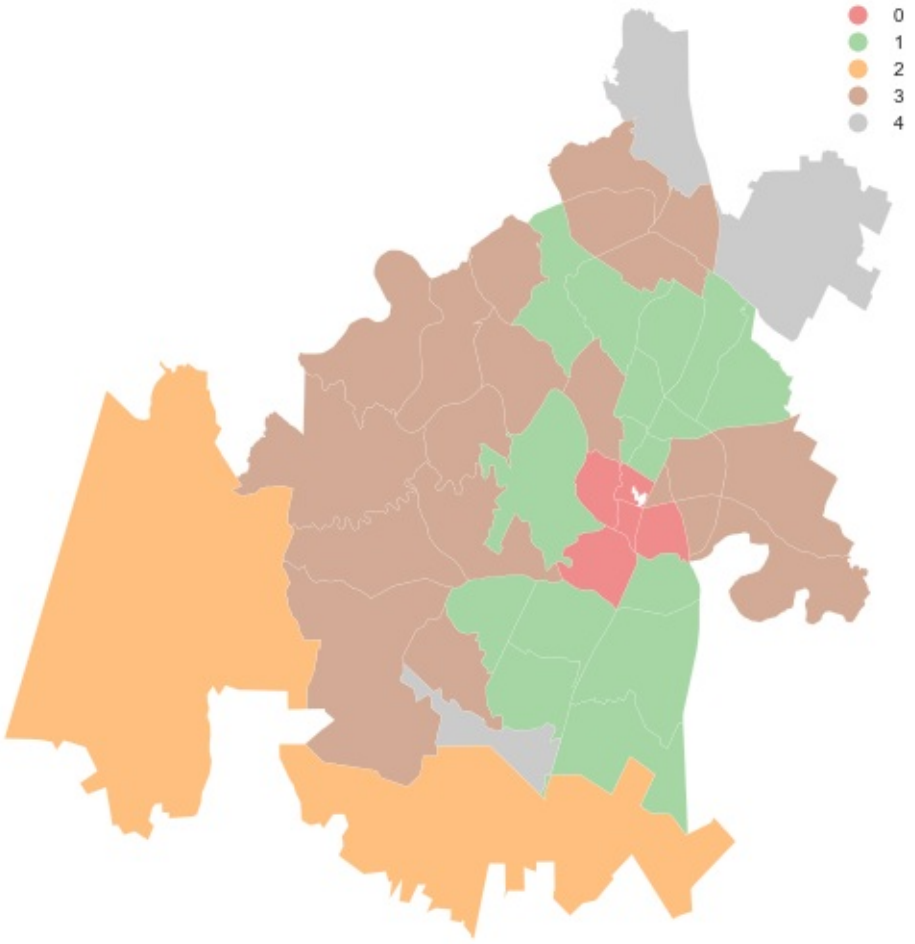
```
km5 = cluster.KMeans(n_clusters=5)
```

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()
```

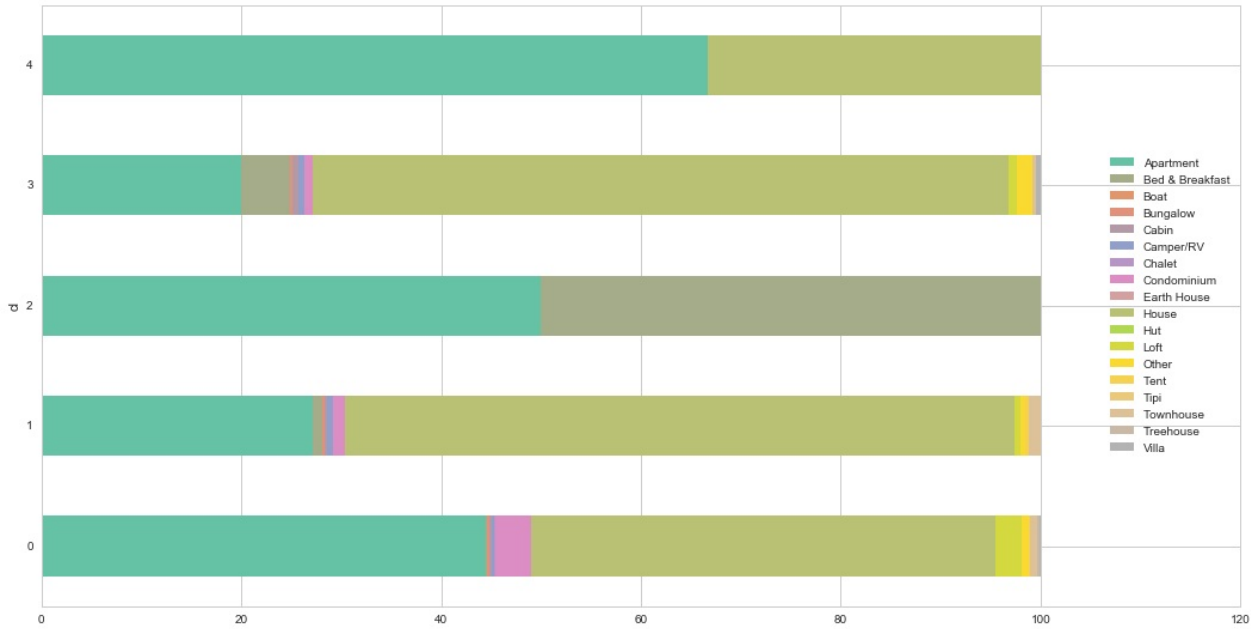


The map above shows a clear pattern: there is a class at the core of the city (number 0, in red), then two other ones in a sort of "urban ring" (number 1 and 3, in green and brown, respectively), and two peripheral sets of areas (number 2 and 4, yellow and green).

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 apartmentes, 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 0x7f0cc9b433d0>
```

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 27.4 s, sys: 0 ns, total: 27.4 s
Wall time: 27.4 s
```

Which allows us to obtain an empirical p-value:

```
maxp.cpvalue
```

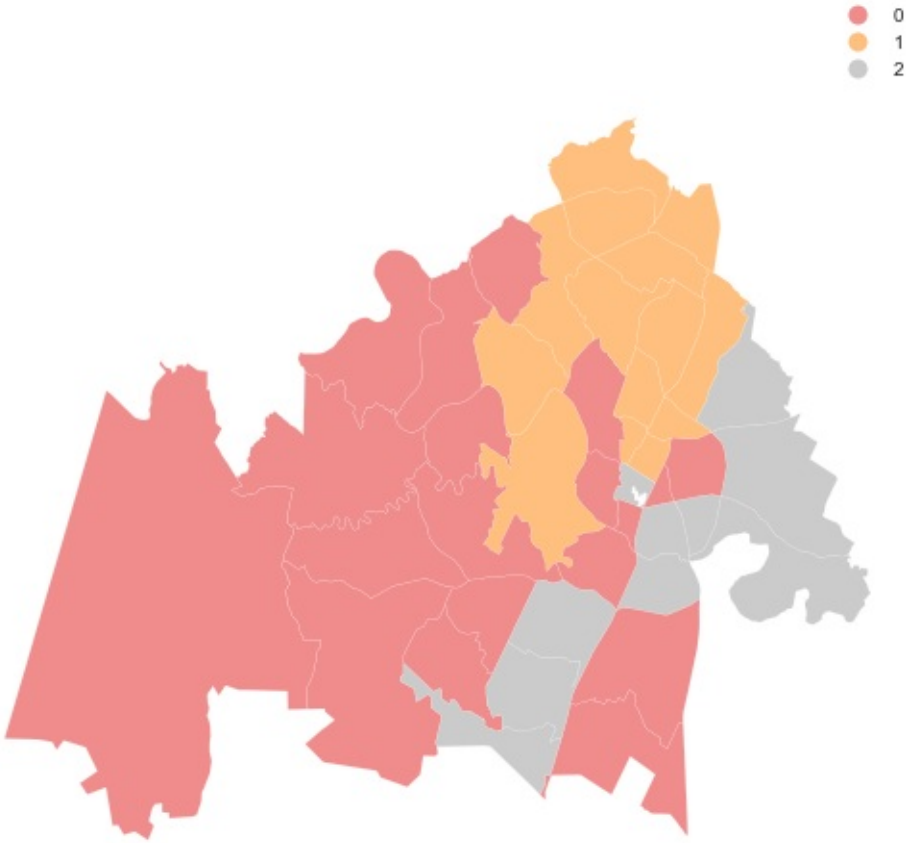
```
0.001
```

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	96.425334	95.264603	92.111148
review_scores_accuracy	9.703800	9.561277	9.548701
review_scores_cleanliness	9.597942	9.610098	8.965437
review_scores_checkin	9.876563	9.821887	9.764887
review_scores_communication	9.909209	9.821964	9.775652
review_scores_location	9.609283	9.483582	8.893100
review_scores_value	9.618152	9.543733	9.441086

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**: much 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")
```

```
/home/dani/anaconda/envs/pydata/lib/python2.7/site-packages/matplotlib/font_manager.py:273: UserWarning: Matplotlib is building the fon
t cache using fc-list. This may take a moment.
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a moment.')
```

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 = 'T'
w
```

```
unsupported weights transformation
```

```
<pysal.weights.weights.W at 0x7f3c066107d0>
```

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='ln(price)')
```

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:        5767
Mean dependent var  :      5.1952             Number of Variables   :           7
S.D. dependent var  :      0.9455             Degrees of Freedom    :        5760
R-squared           :      0.4042
Adjusted R-squared  :      0.4036
Sum squared residual:    3071.189             F-statistic           :    651.3958
Sigma-square        :       0.533             Prob(F-statistic)     :           0
S.E. of regression  :       0.730             Log likelihood        :   -6366.162
Sigma-square ML     :       0.533             Akaike info criterion :   12746.325
S.E of regression ML:      0.7298             Schwarz criterion     :   12792.944

------------------------------------------------------------------------------------
            Variable     Coefficient       Std.Error     t-Statistic     Probability
------------------------------------------------------------------------------------
            CONSTANT       4.0976886       0.0223530     183.3171506       0.0000000
  host_listings_count      -0.0000130       0.0001790      -0.0726772       0.9420655
           bathrooms       0.2947079       0.0194817      15.1273879       0.0000000
            bedrooms       0.3274226       0.0159666      20.5067654       0.0000000
               beds        0.0245741       0.0097379       2.5235601       0.0116440
     guests_included       0.0075119       0.0060551       1.2406028       0.2148030
                pool       0.0888039       0.0221903       4.0019209       0.0000636
------------------------------------------------------------------------------------

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.

```
yxs_w = yxs.assign(w_pool=ps.lag_spatial(w, yxs['pool'].values))

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='ln(price)')
```

```
print(m2.summary)
```

```
REGRESSION
----------
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES
-----------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :  ln(price)              Number of Observations:        5767
Mean dependent var  :      5.1952             Number of Variables   :           8
S.D. dependent var  :      0.9455             Degrees of Freedom    :        5759
R-squared           :      0.4044
Adjusted R-squared  :      0.4037
Sum squared residual:    3070.363             F-statistic           :    558.6139
Sigma-square        :       0.533             Prob(F-statistic)     :           0
S.E. of regression  :       0.730            Log likelihood        :   -6365.387
Sigma-square ML     :       0.532            Akaike info criterion :   12746.773
S.E of regression ML:      0.7297            Schwarz criterion     :   12800.053

------------------------------------------------------------------------------
            Variable     Coefficient       Std.Error     t-Statistic     Probability
------------------------------------------------------------------------------
            CONSTANT       4.0906444       0.0230571     177.4134022       0.0000000
  host_listings_count      -0.0000108       0.0001790      -0.0603617       0.9518697
           bathrooms       0.2948787       0.0194813      15.1365024       0.0000000
            bedrooms       0.3277450       0.0159679      20.5252404       0.0000000
               beds        0.0246650       0.0097377       2.5329419       0.0113373
     guests_included       0.0076894       0.0060564       1.2696250       0.2042695
                pool       0.0725756       0.0257356       2.8200486       0.0048181
              w_pool       0.0188875       0.0151729       1.2448141       0.2132508
------------------------------------------------------------------------------

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 =====================================
```

Spatially lagged endogenous regressors (`WY`)

```
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='ln(price)')
```

```
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:        5767
Mean dependent var  :      5.1952             Number of Variables   :           8
S.D. dependent var  :      0.9455             Degrees of Freedom    :        5759
Pseudo R-squared    :      0.4224
Spatial Pseudo R-squared:  0.4056

------------------------------------------------------------------------------------
            Variable     Coefficient       Std.Error     z-Statistic     Probability
------------------------------------------------------------------------------------
            CONSTANT       3.7085715       0.1075621      34.4784213       0.0000000
  host_listings_count      -0.0000587       0.0001765      -0.3324585       0.7395430
           bathrooms       0.2857932       0.0193237      14.7897969       0.0000000
            bedrooms       0.3272598       0.0157132      20.8270544       0.0000000
               beds        0.0239548       0.0095848       2.4992528       0.0124455
     guests_included       0.0065147       0.0059651       1.0921407       0.2747713
                pool       0.0891100       0.0218383       4.0804521       0.0000449
         W_ln(price)       0.0392530       0.0106212       3.6957202       0.0002193
------------------------------------------------------------------------------------
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 =====================================
```

Prediction performance of spatial models

```
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
```

Exercise

> *Run a regression including both the spatial lag of pools and of the price. How does its predictive performance compare?*