Advance Analytics

(Stream, Sensor & Spatial Temporal Analysis project)

Airbnb Geo Boston Geo Spatial Data Analysis project

Prepared By,

Nhu Hoang Duc - BS19BDS013

Uddeshya Bagla - BS18BDS012

Pradeep Sandilya - BS19BDS023

Duurenbat Batchuluun – BS19BDS014

Problem Statement:

The aim of this project is to analyze how the bnbs priced relative to locations, review number of beds, etc. We are going to build a geo spatial analysis/ Spatial Temporal analysis /predictive based model to predict price based on the bnb features

Introduction:

Airbnb is an American company that operates an online marketplace for lodging, primarily homestays for vacation rentals, and tourism activities. Based in San Francisco, California, the platform is accessible via website and mobile app. Airbnb does not own any of the listed properties; instead, it profits by receiving commission from each booking. The company was founded in 2008 by Brian Chesky, Nathan Blecharczyk, and Joe Gebbia. Airbnb is a shortened version of its original name, AirBedandBreakfast.com.

The company has been criticized for enabling bait-and-switch scams, being involved in West Bank settlements, possibly driving up home rents and creating nuisances for those living near leased properties. The company is regulated by many jurisdictions, including the European Union and cities such as San Francisco and New York City.

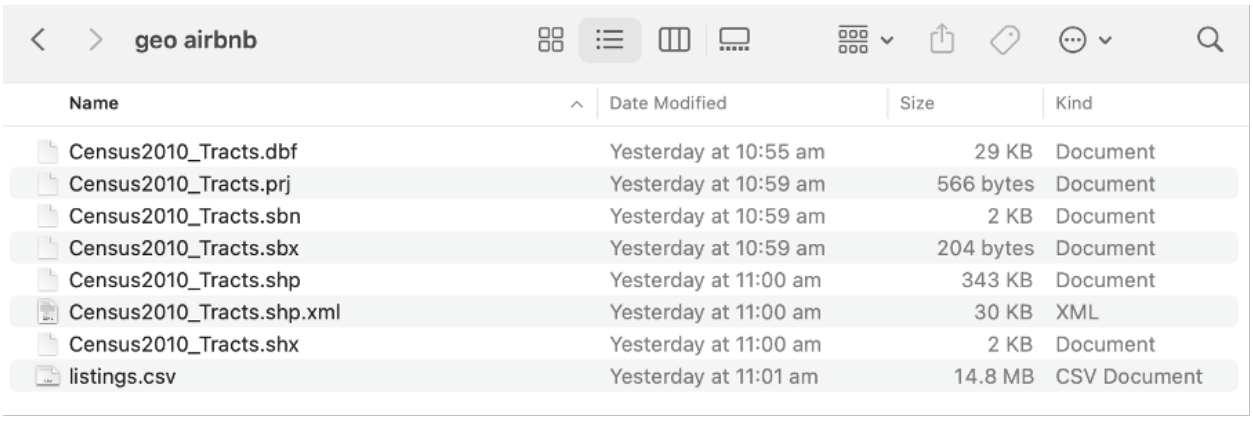
AirBnB and its competitor Vrbo are viewed as a competitive threat by the hotel industries. In all over the the states of America or other countries states the Airbnb is the most accommodation company as compared to other

In this project we are taking the dataset from Airbnb Boston for findings our assumptions on this dataset. Now with this we are implementing predictive analyzing, geopandas, Spatial analysis, spatial regression, spatial clustering.

About the Airbnb Boston Data file format:

This is the Airbnb boston census data folder. This file folder has different file folders format so we are describing here below..

The source of the data [http://insideairbnb.com/get-the-data/]



ArcGIS shapefiles have mandatory and optional files. The mandatory file extensions needed for a shapefile are .shp, .shx and .dbf. But the optional files are: .prj, .xml, .sbn and .sbx

Main File (.SHP)

.shp is a *mandatory* Esri file that gives features their geometry. Every shapefile has its own .shp file that represent spatial vector data. For example, it could be points, lines and polygons in a map.

Index File (.SHX)

.shx are *mandatory* Esri and AutoCAD shape index position. This type of file is used to search forward and backwards.

dBASE File (.DBF).dbf is a standard database file used to store attribute data and object IDs. A .dbf file is *mandatory* for shape files. You can open .DBF files in Microsoft Access or Excel.

Projection File (.PRJ)

.prj is an *optional* file that contains the metadata associated with the shapefiles coordinate and projection system. If this file does not exist, you will get the error “unknown coordinate system”. If you want to fix this error, you have to use the “define projection” tool which generates .prj files.

Extensible Markup Language File (.XML)

.xml file types contains the metadata associated with the shapefile. If you delete this file, you essentially delete your metadata. You can open and edit this *optional* file type (.xml) in any text editor.

Spatial Index File (.SBN)

.sbn is an *optional* spatial index file that optimizes spatial queries. This file type is saved together with a .sbx file. These two files make up a shape index to speed up spatial queries.

Code Page File (.CPG)

.cpg are *optional* plain text files that describes the encoding applied to create the shapefile. If your shapefile doesn’t have a .cpg file, then it has the system default encoding.

Importing Libraries:
Firstly open the jupyter the notebook or Google colab. Now import the libraries required for the analysis. The main libraries we are using here is NumPy, Pandas, GeoPandas, matplotlib.pyplot, seaborn, mplleafnet, shapely geometry, PySal, LibPysal, mapclassify, esda.Moran, mpl_toolkits for 3D axes, statsmodel.api
These are the main libraries we are using for this project now each one we are defining below
NumPy: NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Pandas: Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software. Pandas allows importing data from various file formats such as comma-separated values, JSON, Parquet, SQL database tables or queries, and Microsoft Excel.[8] Pandas allows various data manipulation operations such as merging, reshaping, selecting, as well as data cleaning, and data wrangling features. The development of pandas introduced into Python many comparable features of working with Dataframes that were established in the R programming language. The pandas library is built upon another library NumPy, which is oriented to efficiently working with arrays instead of the features of working on Dataframes.

GeoPandas: The goal of GeoPandas is to make working with geospatial data in python easier. It combines the capabilities of pandas and shapely, providing geospatial operations in pandas and a high-level interface to multiple geometries to shapely. GeoPandas enables you to easily do operations in python that would otherwise require a spatial database such as PostGIS. It is easy to install.

MatPlot: Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK.
Most of the Matplotlib utilities lies under the pyplot submodule and are usually imported under the plt alias.

Seaborn: Seaborn is a library for making statistical graphics in Python. It builds on top of matplotlib and integrates closely with pandas' data structures.Seaborn helps you explore and understand your data. Its plotting functions operate on data frames and arrays containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots. Its dataset-oriented, declarative API lets you focus on

what the different elements of your plots mean, rather than on the details of how to draw them.

Mplleafnet: mplleaflet is a Python library that converts a matplotlib plot into a webpage containing a pannable, zoomable Leaflet map. It can also embed the Leaflet map in an IPython notebook. It converts matplotlibplots into leaflet web maps.
To install this we use pip3 install mplleafnet.

Shapely: Shapely is a BSD-licensed Python package for manipulation and analysis of planar geometric objects. It is based on the widely deployed GEOS (the engine of PostGIS) and JTS (from which GEOS is ported) libraries. Shapely is not concerned with data formats or coordinate systems, but can be readily integrated with packages. Manipulation and analysis of geometric objects in the Cartesian plane.
To install this pip install Shapely

PySal: PySAL, the Python spatial analysis library, is an open source cross-platform library for geospatial data science with an emphasis on geospatial vector data written in Python. It supports the development of high level applications for spatial analysis, such as

- detection of spatial clusters, hot-spots, and outliers
- construction of graphs from spatial data
- spatial regression and statistical modeling on geographically embedded networks
- spatial econometrics
- exploratory spatio-temporal data analysis

Lib Pysal: provides foundational algorithms and data structures that support the rest of the library. This currently includes the following modules: input/output (io), which provides readers and writers for common geospatial file formats; weights (weights), which provides the main class to store spatial weights matrices, as well as several utilities to manipulate and operate on them; computational geometry (cg), with several algorithms, such as Voronoi tessellations or alpha shapes that efficiently process geometric shapes; and an additional module with example data sets.

Map Classify:
implements a family of classification schemes for choropleth maps. Its focus is on the determination of the number of classes, and the assignment of observations to those classes. It is intended for use with upstream mapping and geo visualization packages (see geopandas and geoplot) that handle the rendering of the maps. It is a part of the PySal Library.

ESDA Moran: Moran's Global Autocorrelation Statistic
class esda. Moran(y, w, transformation='r', permutations=999, two_tailed=True) , find below parameters syntax

Parameters

Y *array*
variable measured across n spatial units
w *W*
spatial weights instance
transformation *str*
weights transformation, default is row-standardized "r". Other options include "B": binary, "D": doubly standardized, "U": untransformed (general weights), "V": variance-stabilizing.
Permutations *int*
number of random permutations for calculation of pseudo-p_values
two_tailed *bool*
If True (default) analytical p-values for Moran are two tailed, otherwise if False, they are one-tailed.

Mpl_ToolKit: mpl_toolkits.mplot3d provides some basic 3D plotting (scatter, surf, line, mesh) tools. Not the fastest or most feature complete 3D library out there, but it ships with Matplotlib and thus may be a lighter weight solution for some use cases.

DATA ANALYSIS, PLOTTING & SPATIAL ANALYSIS:

In this project we defined above libraries which we are implementing here. Now read the csv file data called listings.csv in the folder First, let's boot up and examine our data. Since our data comes in a simple CSV file, we load it into a panda Data Frame.

```
[ ]  listings = pd.read_csv("listings.csv")
```

Now it will read the from the folder

```
listings.head()
```

	id	listing_url	scrape_id	last_scraped	name	summary	space	description	experiences_offered	neighborhood_overview	notes	
147973	https://www.airbnb.com/rooms/12147973	20160906204935	2016-09-07	Sunny Bungalow in the City	Cozy, sunny, family home. Master bedroom high...	The house has an open and cozy feel at the sam...	Cozy, sunny, family home. Master bedroom high...	none	Roslindale is quiet, convenient and friendly. ...	NaN	The bloc
075044	https://www.airbnb.com/rooms/3075044	20160906204935	2016-09-07	Charming room in pet friendly apt	Charming and quiet room in a second floor 1910...	Small but cozy and quite room with a full size...	Charming and quiet room in a second floor 1910...	none	The room is in Roslindale, a diverse and prima...	If you don't have a US cell phone, you can	Plenty parking

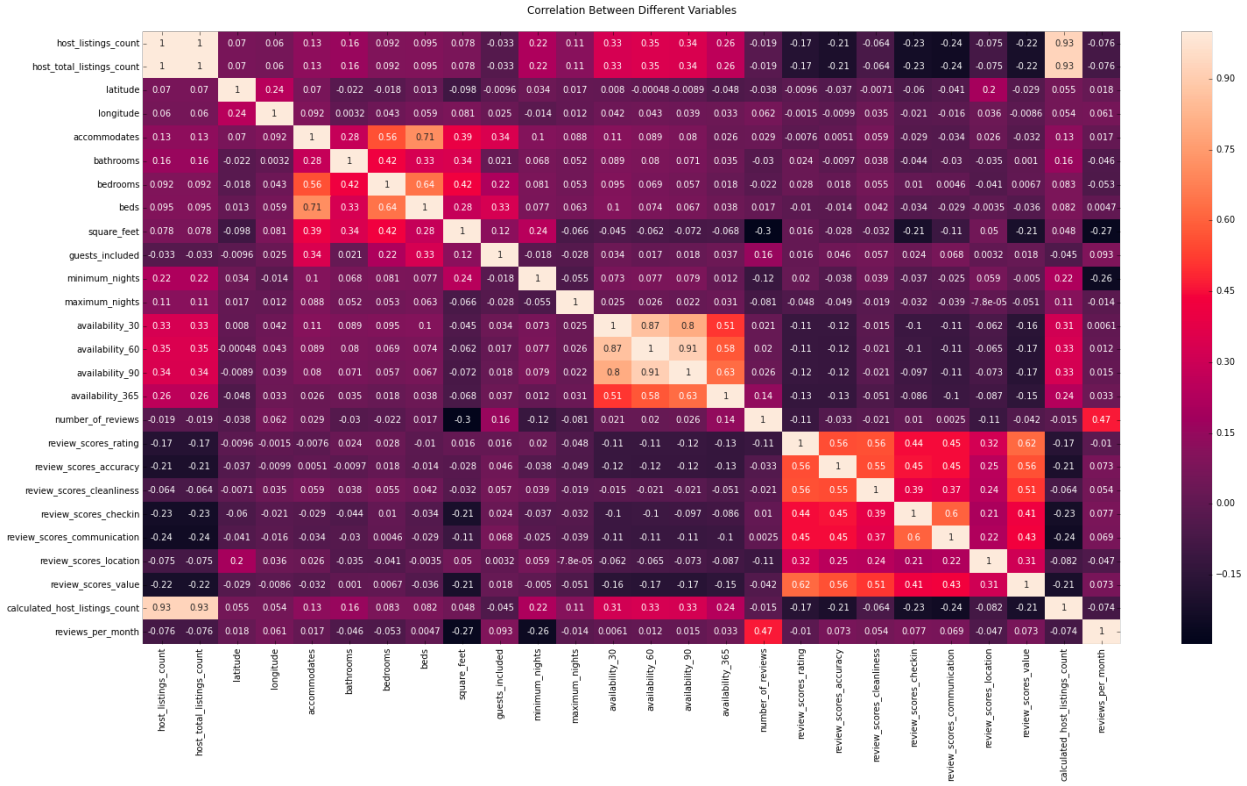
Now find the information of the data whether it has null values or not, for suppose if it has a null values remove null values or drop the null value columns, and we have to clean a lot of data to do exploratory analysis. After dropping the columns we defined in the data frame called listings_1 that can we compared with correlational data frame

```
[ ] listings.info()

<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 3585 entries, 0 to 3584
Data columns (total 97 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   id                              3585 non-null   int64
 1   listing_url                     3585 non-null   object
 2   scrape_id                       3585 non-null   int64
 3   last_scraped                    3585 non-null   object
 4   name                            3585 non-null   object
 5   summary                         3442 non-null   object
 6   space                           2528 non-null   object
 7   description                     3585 non-null   object
 8   experiences_offered             3585 non-null   object
 9   neighborhood_overview           2170 non-null   object
 10  notes                           1610 non-null   object
 11  transit                         2295 non-null   object
 12  access                          2096 non-null   object
 13  interaction                     2031 non-null   object
 14  house_rules                     2393 non-null   object
 15  thumbnail_url                   2986 non-null   object
 16  medium_url                      2986 non-null   object
 17  picture_url                     3585 non-null   object
 18  xl_picture_url                  2986 non-null   object
 19  host_id                         3585 non-null   int64
 20  host_url                        3585 non-null   object
 21  host_name                       3585 non-null   object
 22  host_since                      3585 non-null   object
 23  host_location                   3574 non-null   object
 24  host_about                      2276 non-null   object
 25  host_response_time              3114 non-null   object
 26  host_response_rate              3114 non-null   object
 27  host_acceptance_rate            3114 non-null   object
 28  host_is_superhost               3585 non-null   object
```

```
listings1.corr()
```

	host_listings_count	host_total_listings_count	latitude	longitude	accommodates	bathrooms	bedrooms	beds	square_feet	guests_included
host_listings_count	1.000000	1.000000	0.124228	0.044281	0.160164	0.170443	0.141611	0.038048	-0.028763	-0.105023
host_total_listings_count	1.000000	1.000000	0.124228	0.044281	0.160164	0.170443	0.141611	0.038048	-0.028763	-0.105023
latitude	0.124228	0.124228	1.000000	0.319327	0.053818	-0.020733	-0.045471	-0.012382	-0.074430	-0.019589
longitude	0.044281	0.044281	0.319327	1.000000	0.083658	0.010164	0.015721	0.033736	0.063663	0.016257
accommodates	0.160164	0.160164	0.053818	0.083658	1.000000	0.346914	0.724826	0.815366	0.486852	0.495327
bathrooms	0.170443	0.170443	-0.020733	0.010164	0.346914	1.000000	0.430854	0.347717	0.466834	0.110784
bedrooms	0.141611	0.141611	-0.045471	0.015721	0.724826	0.430854	1.000000	0.710654	0.491993	0.396525
beds	0.038048	0.038048	-0.012382	0.033736	0.815366	0.347717	0.710654	1.000000	0.312219	0.477326
square_feet	-0.028763	-0.028763	-0.074430	0.063663	0.486852	0.466834	0.491993	0.312219	1.000000	0.111401
guests_included	-0.105023	-0.105023	-0.019589	0.016257	0.495327	0.110784	0.396525	0.477326	0.111401	1.000000
minimum_nights	0.057400	0.057400	0.020135	-0.019292	-0.038236	0.021251	-0.004993	-0.020485	0.041722	-0.029542
maximum_nights	-0.004450	-0.004450	0.017591	0.015473	-0.009754	-0.007389	-0.005675	-0.010055	-0.122332	-0.006805
availability_30	0.472413	0.472413	0.004026	0.030450	0.119488	0.117086	0.097908	0.083329	-0.059068	-0.008894
availability_60	0.420898	0.420898	-0.019708	0.044846	0.099504	0.108912	0.075957	0.067515	-0.077331	0.002875
availability_90	0.376304	0.376304	-0.034473	0.053040	0.095363	0.091299	0.058878	0.064910	-0.108102	0.027438
availability_365	0.058604	0.058604	-0.099346	0.046331	0.056993	0.053503	0.028642	0.057344	-0.171845	0.071025
number_of_reviews	-0.140612	-0.140612	-0.039188	0.075634	-0.001810	-0.025705	-0.042669	-0.008720	-0.335045	0.064606
review_scores_rating	-0.125055	-0.125055	-0.014511	0.017748	0.036653	0.002122	0.055272	0.033852	-0.175922	0.047402

CORRELATION HEAT MAP WITH DIFFERENT VARIABLES

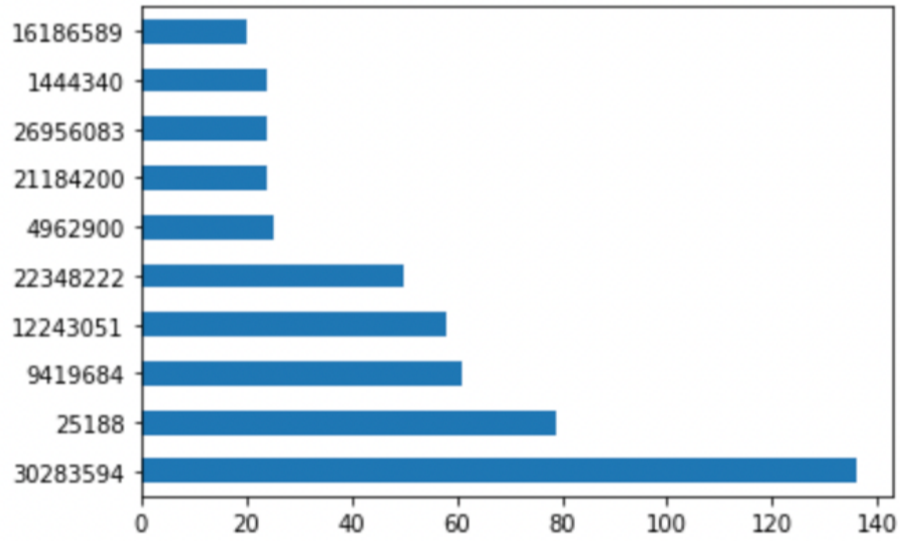


From the above graph we can see the Correlational Between different variables

Visualisations:

```
listings.host_id.value_counts().head(10).plot(kind='barh')
```

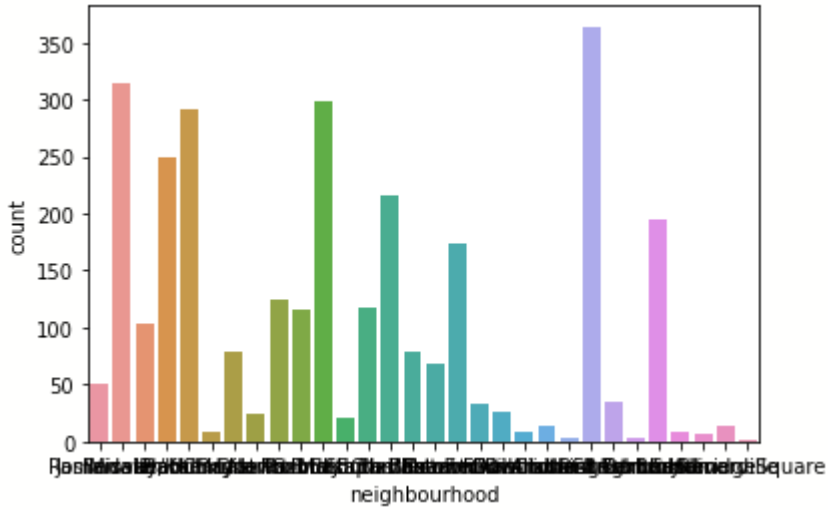
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f526a08f5d0>
```



From the above graph plot between host ID and the number of air bnb that are managed by them. From this graph we can find the top 10 people that owns or manages most number of airbnbs in Boston city.we can see that there is a good distribution between top 10 hosts with the most listings. First host has more than 120+ listings.

Neighbourhood group wise distribution of airbnbs –

This graph below is a plot between Neighbourhood groups and the number of airbnbs that are in that area



For above graph we can observe clearly in the google colab project repository..

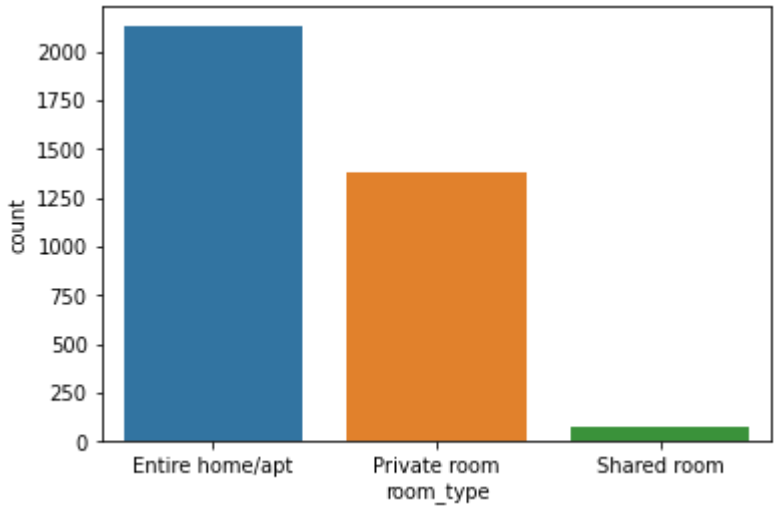
```
listings.neighbourhood.value_counts()
```

```
Allston-Brighton       364
Jamaica Plain          314
South End              298
Back Bay               291
Fenway/Kenmore         249
South Boston           216
Dorchester             195
Beacon Hill            174
North End              125
East Boston            117
Roxbury                116
Mission Hill           103
Charlestown             79
Chinatown               78
West End                68
Roslindale              50
West Roxbury            35
Theater District        33
Downtown Crossing       26
Hyde Park               25
Mattapan                20
Financial District      13
Somerville              13
Downtown                 8
Leather District         8
Brookline                8
Cambridge                7
Chestnut Hill            4
Government Center        3
Harvard Square           2
Name: neighbourhood, dtype: int64
```

The X-axis represent the neighbourhood and y-axis represent the count of the listings from the above graph Allston-Brighton highest count of neighbourhood and least count is Harvard Square..

Airbnb room type and count:

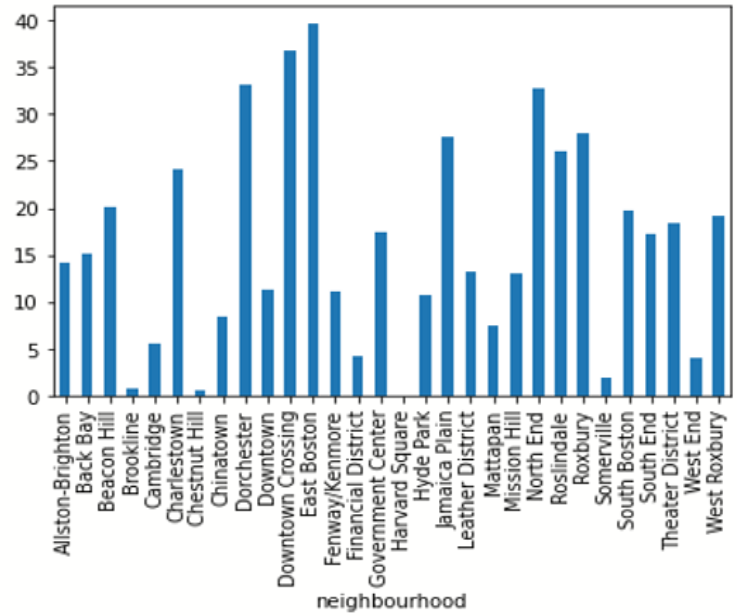
The room type distribution of airbnbs - This graph below is a plot between room types and the number of airbnbs that are of that type.



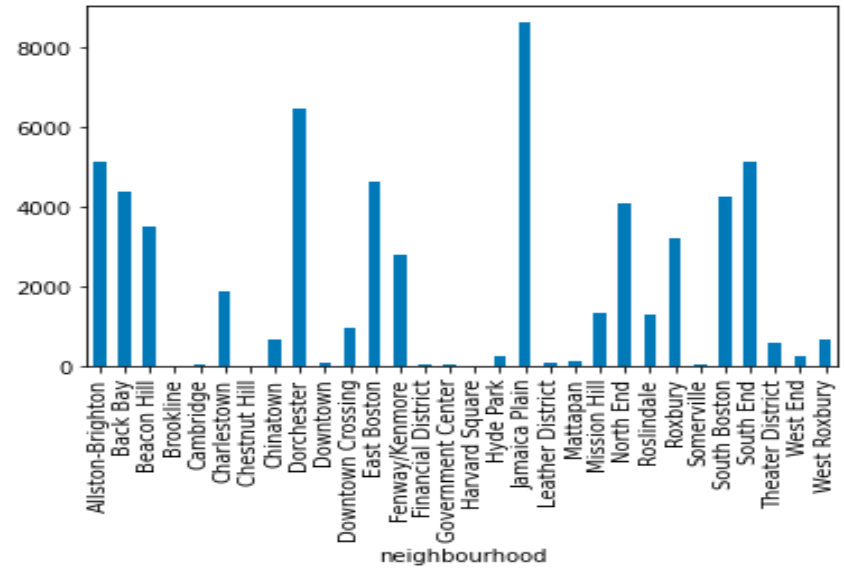
From this graph we infer the maximum number of airbnbs in the whole listings are that from entire home/apt..

Plot between number neighbourhood group Vs the sum and the mean of number of reviews in that place.

Mean number of reviews



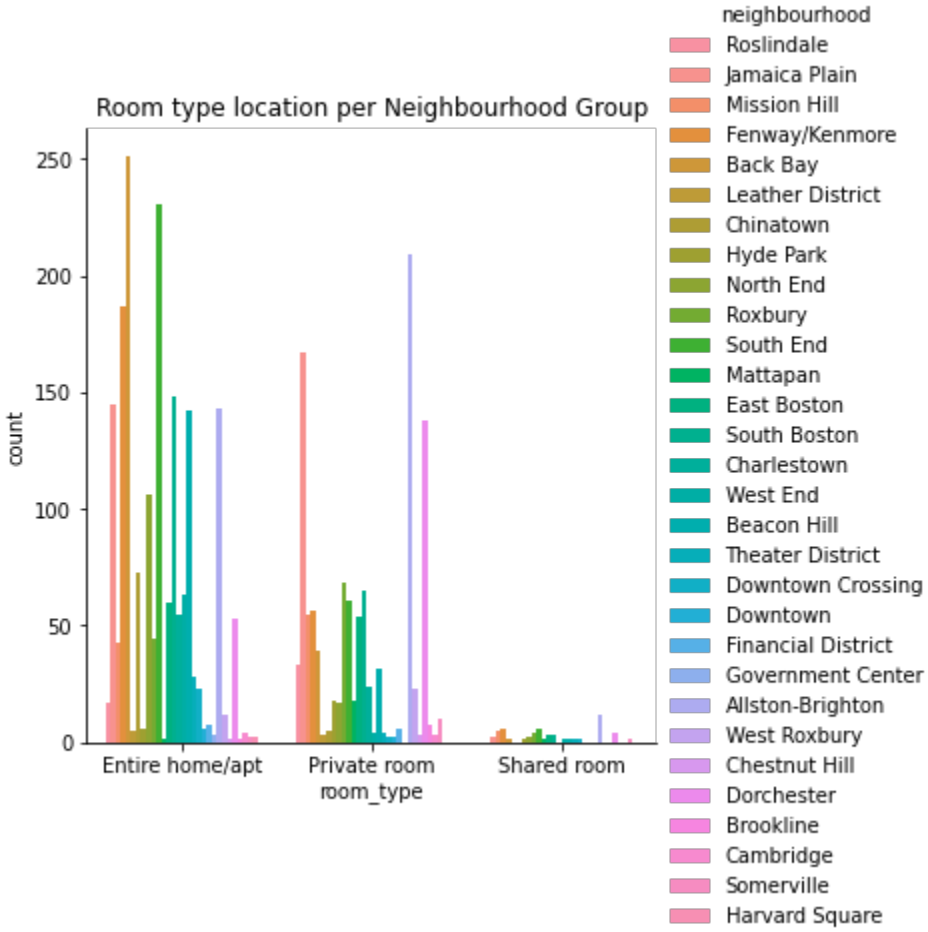
Sum number of reviews



Average is used to summarize the data or when we try to compare among groups where each group has different member counts. From above if we can compare with highest number of sum reviews with mean review there is a lot of smilirality between the two graphs, for example if we see the Jamaica Plain it has highest number of reviews than compared with mean reviews, here the reviews are placed on the Hygienity of the neighbourhood,

cleanliness and maintenance of a certain neighbourhood than compared with average mean of all reviews.

Category count plot between room type and neighborhood group:



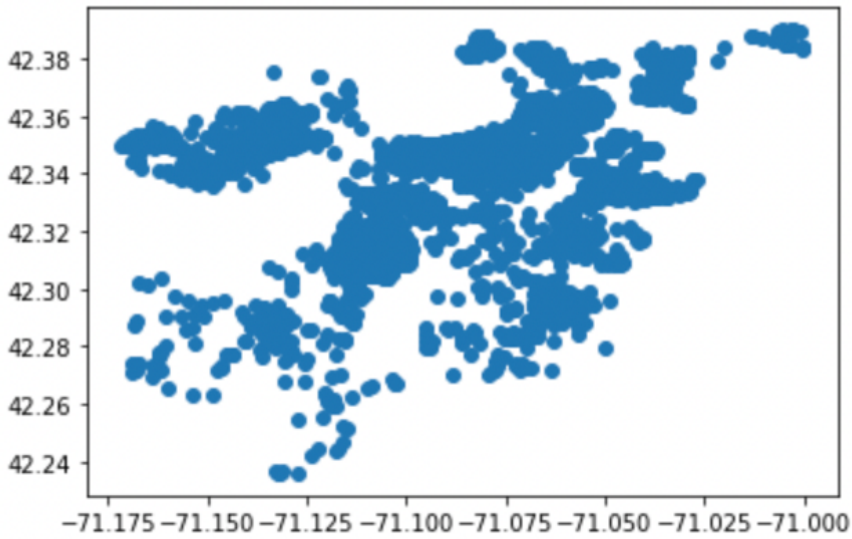
From above graph Room type location per neighbourhood group, if we can see above graph there are more people choosing Entire home/apt than compared with private room type and shared room.

Scatter Plot between Longitude & Latitude of Airbnb:

At the moment our listings have several geospatial variables, the most important of which are longitude and latitude, which give the exact coordinates of the BnB in question. This means that it's easy for us to, say, plot every BnB location on a map:

```
plt.scatter(listings['longitude'], listings['latitude'])
```

```
<matplotlib.collections.PathCollection at 0x7fad37a902d0>
```

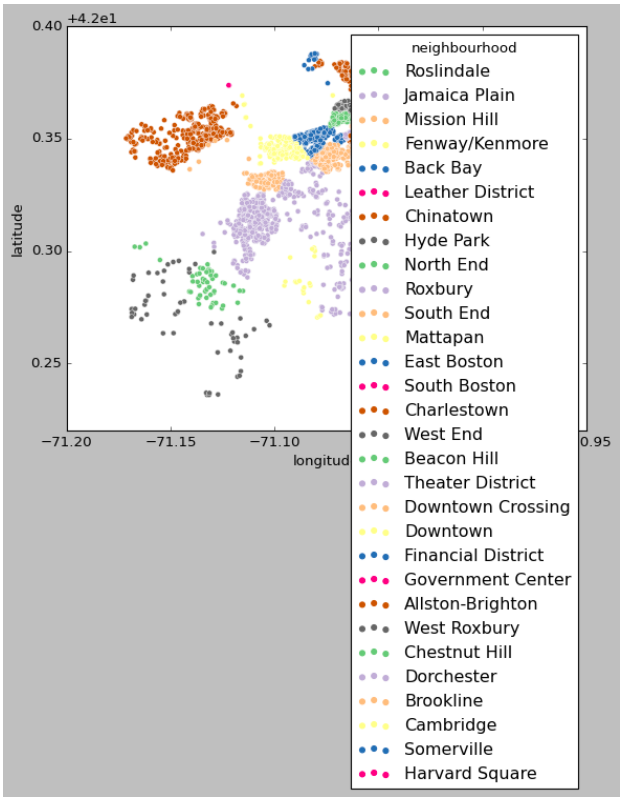


In this plot we see...not much, really. If we are very intimately familiar with the layout of the city of Boston, you will probably be able to make sense of some of these clusters which are, to me, not being from the city, its totally mysterious.

In other words, this plot is missing something important: geospatial context.

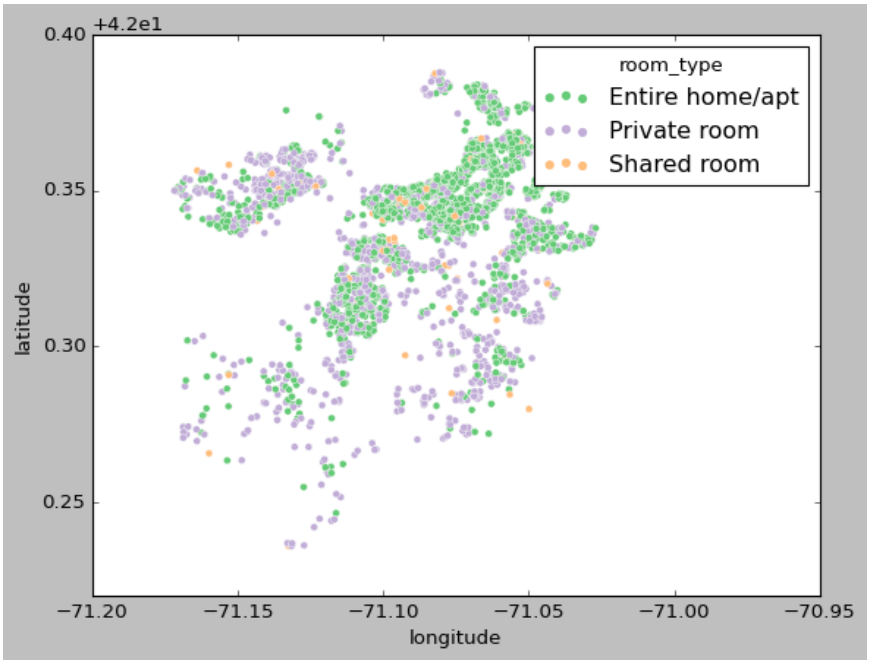
Additionally, this display is unprojected—it's displayed in terms of raw coordinates. The amount of distance contained in a coordinate degree varies greatly depending on where you are, so this naive plot potentially badly distorts distances.

Listings neighborhood with Longitude & Latitude:



From above graph you can see the listings of neighbourhoods with Longitude and latitude

Listings of Room Types with Longitude and Latitude:

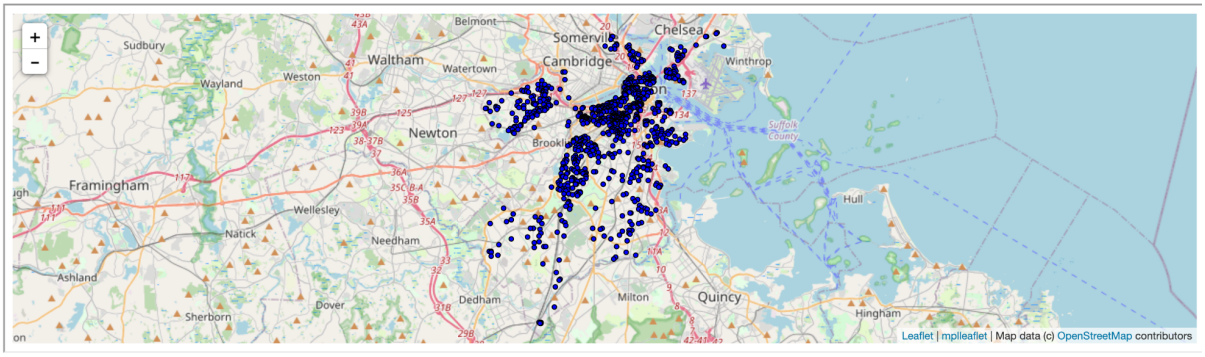


Implementation on the Real Geographical Map:

To implement this first mainly install mpleafflet,Enter mplleaflet. mplleaflet is a tool that automatically takes a coordinate matplotlib plot of any kind and places it on top of a leaflet slippy map. The best part is that it's just one additional line of code. Just throw mplleaflet.display() after generating your plot to drop it inline in your Jupyter notebook. Note, however, that rather than plotting every single point on our map we've taken a sample of 1000 of them and plotted those. This is known as **undersampling**.

This is oftentime necessary, as here, for performance reasons. Because mplleaflet embeds point objects directly on an SVG canvas, it's as much as two orders of magnitude slower and more power-hungry than generating a really basic matplotlib blob.

```
import mplleaflet

sample = listings.sample(1000)
plt.scatter(sample['longitude'], sample['latitude'])

mplleaflet.display()
```



Plotting Geometries

The world around us is split up into lots of different kinds of geometries. The big ones are continents; from there we go down to countries, states, districts, counties, and so on.

On the city level, where we are in this case, there are usually a bunch of federal options (PUMAs or electoral districts for example) as well as a variety of usually more targetted geographical aggregations provided by the city government. Usually no matter where you go, however, the lowest-level geometries in the United States are census tracts. Census tracts are created every ten years for the census, and are built to try to contain either 4000 or 0 people (in the case of parks, beaches, etc.).

Boston GIS released data on the census tracts in their city not long after the 2010 census went out, and that's what we'll use.However, the data comes in the form of a complex and convoluted GIS-standard data format known as a shapefile pandas has no facilities for reading shapefiles.

Now read the shape file given from the data..

```
import geopandas as gpd
boston = gpd.read_file("Census2010_Tracts.shp")
boston
```

	STATEFP10	COUNTYFP10	TRACTCE10	GEOID10	NAME10	NAMELSAD10	MTFCC10	FUNCSTAT10	ALAND10	AWATER10	INTPTLAT10	INTPTLON10	Shape_Leng	Shaj
0	25	025	010405	25025010405	104.05	Census Tract 104.05	G5020	S	363702.0	0.0	+42.3398654	-071.0896052	14629.550361	3.914
1	25	025	010404	25025010404	104.04	Census Tract 104.04	G5020	S	136829.0	0.0	+42.3419667	-071.0886375	5277.643216	1.472
2	25	025	010801	25025010801	108.01	Census Tract 108.01	G5020	S	127905.0	0.0	+42.3541193	-071.0770216	6166.497167	1.376
3	25	025	010702	25025010702	107.02	Census Tract 107.02	G5020	S	299981.0	0.0	+42.3518354	-071.0755159	7818.852369	3.228

After completing of data file reading, here the main thing for shape file is geometry data

```
[113] boston['geometry'].head()

      0    POLYGON ((766978.240 2951616.924, 767000.369 2...
      1    POLYGON ((766835.394 2949104.970, 766736.720 2...
      2    POLYGON ((769261.263 2954196.132, 769572.345 2...
      3    POLYGON ((772221.707 2953536.652, 772276.570 2...
      4    POLYGON ((762449.977 2952359.446, 762888.695 2...
      Name: geometry, dtype: geometry
```

However, these polygons seem more than a little bit weird. The coordinates don't make any sense!

It turns out that the shapefile we're working with is encoded in some kind of alternative **coordinate reference system** (CRS), one with the origin point set to (200000, 750000). We can see this for ourselves by asking our GeoDataFrame its crs property:

```
boston.crs
```

```
<Projected CRS: PROJCS["NAD83 / Massachusetts Mainland (ftUS)",GEO ...>
Name: NAD83 / Massachusetts Mainland (ftUS)
Axis Info [cartesian]:
- [east]: Easting (US survey foot)
- [north]: Northing (US survey foot)
Area of Use:
- undefined
Coordinate Operation:
- name: unnamed
- method: Lambert Conic Conformal (2SP)
Datum: North American Datum 1983
- Ellipsoid: GRS 1980
- Prime Meridian: Greenwich
```

The master directory for all well-known coordinate systems is the **EPSG**. The simple latitude-and-longitude reference system is epsg:4326, and geopandas makes it really, really easy to "fix" our coordinates by just calling the to_crs method:

```
boston = boston.to_crs({'init': 'epsg:4326'})
```

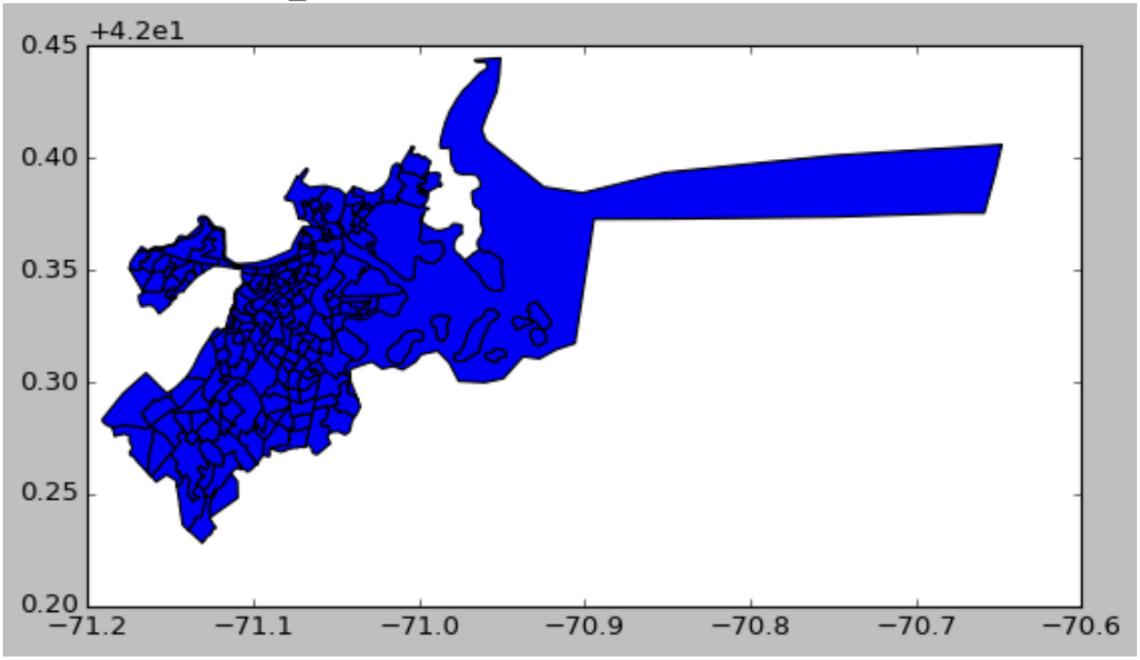
```
boston['geometry'].head()
```

```
0    POLYGON ((-71.09009 42.34666, -71.09000 42.346...
1    POLYGON ((-71.09066 42.33977, -71.09103 42.339...
2    POLYGON ((-71.08159 42.35370, -71.08044 42.354...
3    POLYGON ((-71.07066 42.35185, -71.07045 42.351...
4    POLYGON ((-71.10682 42.34875, -71.10520 42.348...
Name: geometry, dtype: geometry
```

Geopandas objects have a plot method which plops geometry into a matplotlib plot right away

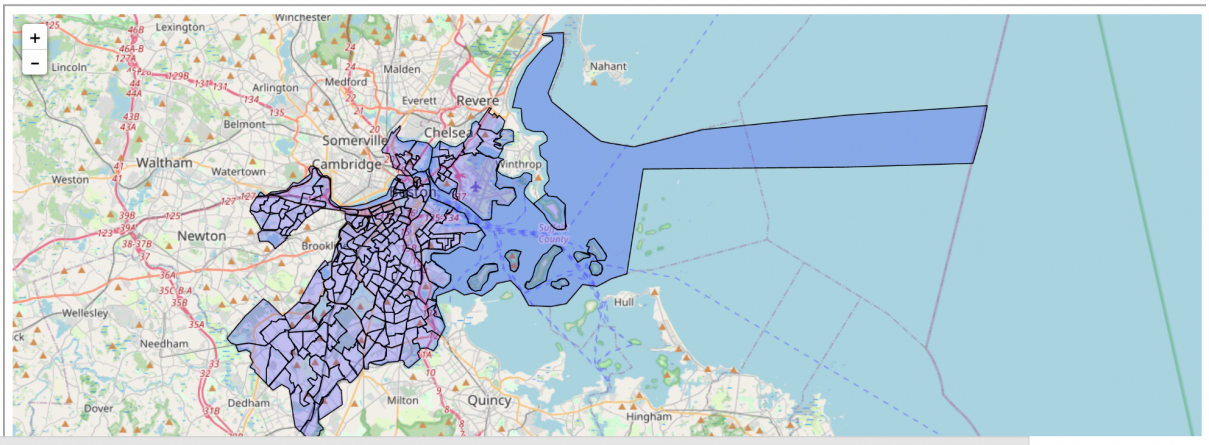
```
boston.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f52628e6f90>
```



Again, to get geographic context, let's use mplleaflet!

```
import mplleaflet

f = plt.figure(figsize=(15, 8))
ax = f.gca()
boston.plot(ax=ax)
mplleaflet.display(fig=f)
```



There it is—all the census tracts of the city of Boston.

Notice that some of these tracts have really "weird" shapes. This is because these are census tracts, remember, which try to isolate parts of the city which have people in them from those (like airports, beaches, islands, or the waterway) which do not.

Upconverting DataFrame to GeoDataFrame Objects

geopandas represents geometries using the shapely library. Just like how when using pandas

From Importing shapely

Recall that our BnB listings was a DataFrame containing two geospatial fields, latitude and longitude. As a quick demonstration of shapely, let's upconvert this representation to true geometries. To do this we wrap our DataFrame in a GeoDataFrame, mapping our geometry column to a shapely Point for each coordinate pair.

```
import shapely
listings = gpd.GeoDataFrame(listings, geometry=listings.apply (parameter) srs: Any
        lambda srs: shapely.geometry.Point(srs['longitude'], srs['latitude']), axis='columns'
    ))
```

Plotting Geometries and Points

We've plotted our points and our polygons separately by seaborn and Kde plot

```
f = plt.figure(figsize=(15, 8))
ax = f.gca()

boston.plot(ax=ax, alpha=0.1, linewidth=0.25, color='white')
sns.kdeplot(data=listings.apply(lambda srs: pd.Series({'x': srs.geometry.x, 'y': srs.geometry.y}), axis='columns'), ax=ax,
            alpha=1)
ax.set_axis_off()
```

This does give us a good sense of where our AirBNBs are concentrated. Unfortunately, however, KDEs are almost always too "blobish"—they're not granular enough to show the details inherent in geographic distributions.

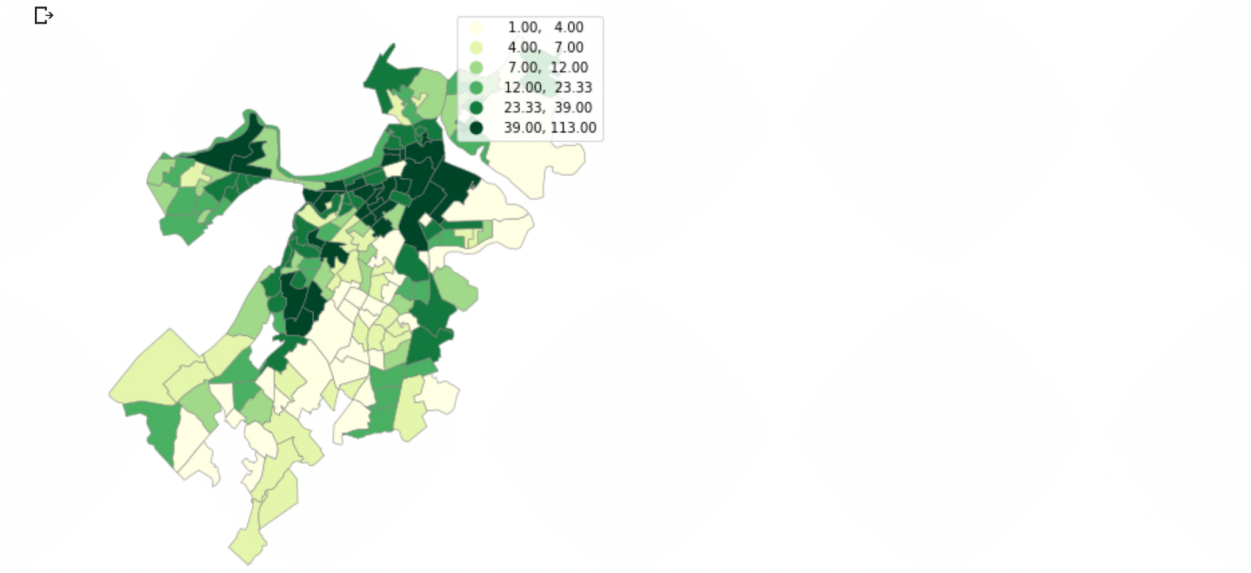
With a little bit more work we can get to the closest thing to a gold standard for such things in cartography, a choropleth map.First let's create a column in our polygonal dataset, BNBs, containing a count of the number of AirBnB locations in the area.

This type of map is, as was mentioned earlier, known as a choropleth. It's a ubiquitious classic of the genre. The darker the shading, the more AirBNBs in that census tract.

Aside from a variety of visual parameters, we specified three important ones. column=BNBs told geopandas which columns to plot data from. scheme=QUANTILES is which "bucketing" system you want to plot with: in this case we specified quantiles, which splits the data into bins equal in *number*, albiet not in *size*. Finally, k tells geopandas how many bins we want.

The trouble with this map, however, is that it doesn't take into account the *size* of the census tracts. If our census tract is bigger than average, it'll naturally contain more homes, and, all else being equal, more AirBnBs.

```
f = plt.figure(figsize=(15, 8))
ax = f.gca()
kw = dict(column='BNBs', k=6, cmap='YlGn', alpha=1, legend=True, edgecolor='gray', linewidth=0.5)
boston.plot(scheme='QUANTILES', ax=ax, **kw)
ax.set_axis_off()
```



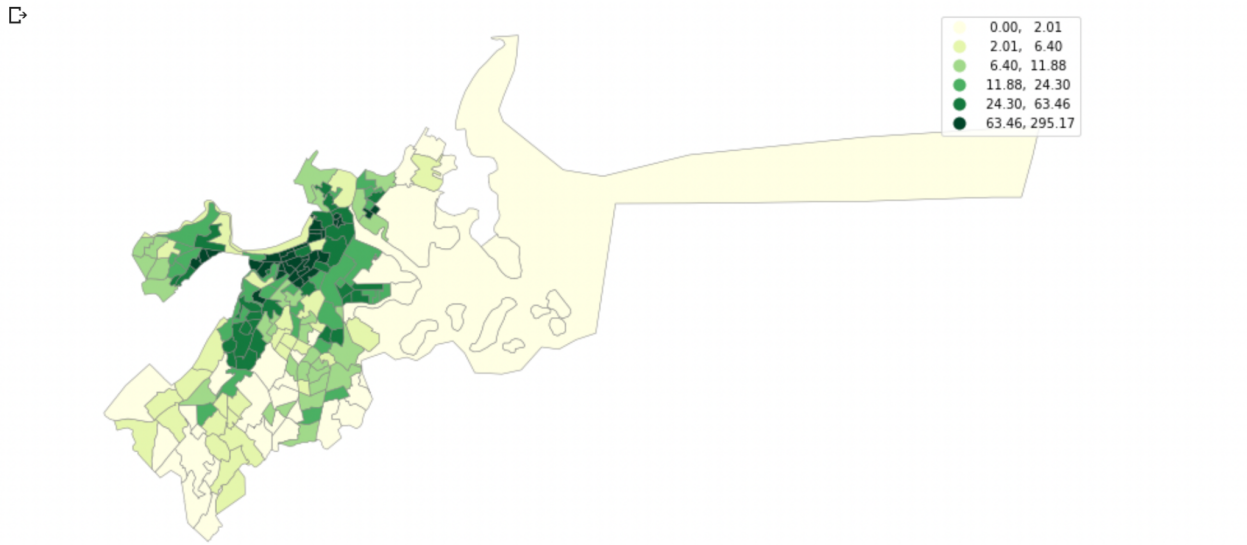
Notice how the plot on the left is completely scrambled and doesn't actually tell us anything, while the plot on the right is smooth, structured, and informative.

In our case we'll fix it by figuring out AirBnB density per square kilometer, not just the raw number, and plot that.

Remember, however, that areas in the latitude-longitude coordinate system are not equal. To get an accurate measurement of area we need to first reproject to what's called an "equal-area" projection (epsg:3395 will do nicely), and then take the areas of *those* polygons instead. Note also that shapely returns areas in square meters; we divide by a constant to get square kilometers. All this is done by the workhourse one-liner below.

```
boston['BNBDensity'] = (boston['BNBs'] / boston['geometry']\
                                .to_crs({'init': 'epsg:3395'})\
                                .map(lambda p: p.area / 10**6))\
                                .fillna(0)
```

```
f = plt.figure(figsize=(15, 8))
ax = f.gca()
kw = dict(column='BNBDensity', k=6, cmap='YlGn', alpha=1, legend=True, edgecolor='gray', linewidth=0.5)
boston.plot(scheme='QUANTILES', ax=ax, **kw)
ax.set_axis_off()
```



This more contiguous plot, Here we see that in the inner city AirBnB density reaches from 60 to almost 300 AirBnB locations per square kilometer. Since there are approximately 150 city blocks per kilometer squared, a bac

k-of-the-envelope calculation says that this makes for anywhere from an AirBnB every three blocks to two AirBnBs per block.

SPATIAL WEIGHTS:

Spatial data analytics requires understanding spaces, and understanding spaces requires tying them together somehow. Here we are using new tool library PySal

Spatial weights tie together areas which are neighbors with one another in some sense. There are many ways of defining this concept of neighborhood-ness or closeness. We could, for example, pick all of the other shapes which are within a certain distance of our chosen shape. Or we could choose all shapes which directly border ours, or at least touch ours. The former of these is known as rook continuity and the latter as queen continuity, after the way those pieces move on a checkerboard, and they're two of the most common methods for defining spatial weights.

we'll pick queen continuity. This means that we will consider our census tracts to be neighbors so long as they touch at least at an edge. We can read such facts directly out of the shapefile using lps.weights.queen_from_shapefile:

We can see the below code screen shot from our code file

```
import libpysal as lps
import numpy as np
```

```
[133] shp_path = 'Census2010_Tracts.shp'
```

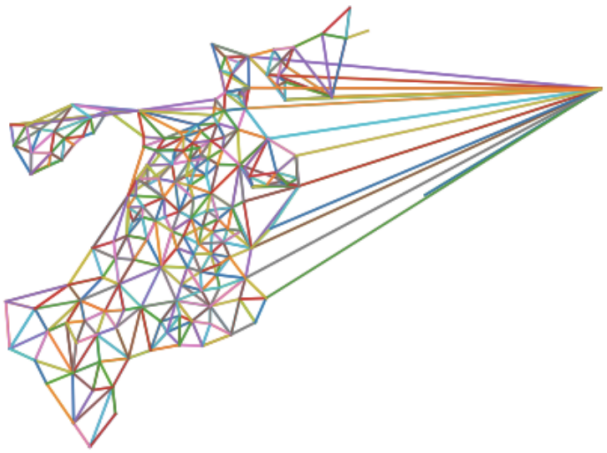
```
[134] qW = lps.weights.Queen.from_shapefile(shp_path)
```

```
[135] qW[4]
```

```
{6: 1.0, 10: 1.0, 50: 1.0, 80: 1.0}
```

This tells us that, for instance, our fourth census tract has four "touching" neighbors: tracts 6, 10, 50, and 80

```
f = plt.figure(figsize=(15, 8))
ax = f.gca()

for i, (k, neighbors) in enumerate(qW.neighbors.items()):
    origin = boston.geometry.iloc[i].centroid
    for nabe_i in neighbors:
        nabe_centroid = boston.geometry.iloc[nabe_i].centroid
        plt.plot([origin.x, nabe_centroid.x], [origin.y, nabe_centroid.y], '-')

boston.plot(ax=ax, linewidth=0.5, facecolor='white')

ax.set_axis_off()
```



To get a better sense of what we've just created, let's plot our neighborhood vectors on a map

SPATIAL LAGS:

Spacial weights are necessary in order to get a sense of the spatial lag inherent in our data. Spatial lag is a geographic version of autocorrelation, a perhaps familar term from time-series analysis: it's a measure of how strongly the data in one polygon is predicted by its neighbors.

If our data is totally randomly distributed on the map—if, in effect, it has *no* spatial component—then we expect that the BnB densities in our neighboring census tracts will be very poor predictors of the BnB density in our current tract, because we'll just be predicting noise with noise. If, on the other hand, the densities we predict are close to the real value, then

we *can* predict density based on geography, and our data does indeed have a strong geographic component.

Of course, examining our plots from earlier, this may seem like a moot point—the choropleths make it pretty obvious that there's geography involved. But what we'd like to do now is assign a probability to that geography not being just random chance. Examining spatial lag allows us, essentially, to understand the statistical significance of our result.

Let's generate lags, split them into five quantiles, and see where they end up.

SPATIAL LAG

```
[ ] import mapclassify as mc
    bnb_spatial_lags = lps.weights.lag_spatial(qW, boston['BNBDensity'])
    spatial_lag_classes = mc.Quantiles(bnb_spatial_lags, k=5)
```

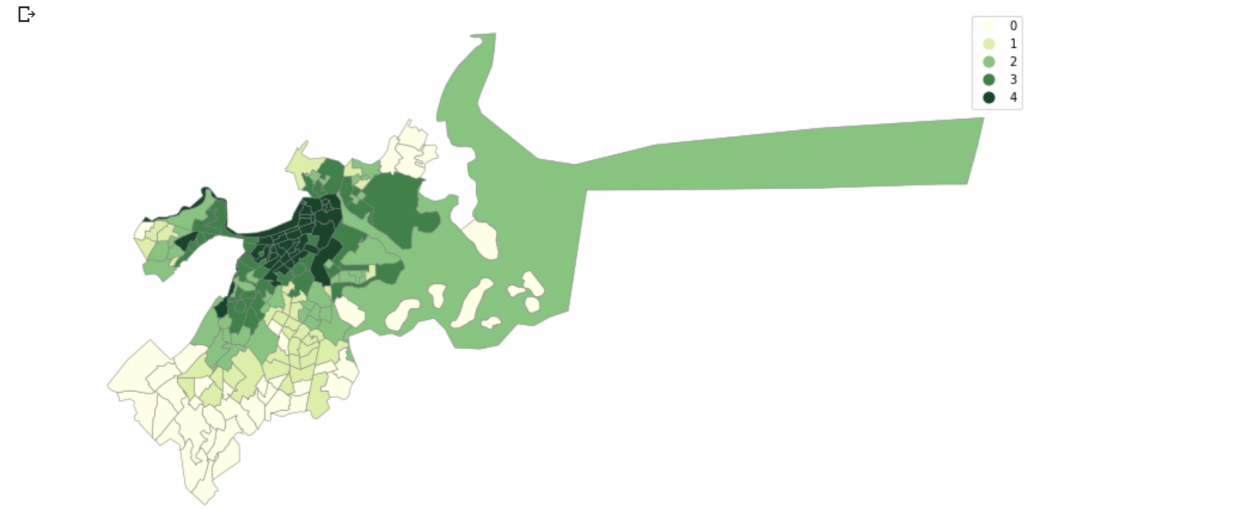
```
[ ] spatial_lag_classes
```

```
    Quantiles

         Interval        Count
    ---------------------------
    [   0.00,   24.42] |    37
    (  24.42,   59.60] |    36
    (  59.60,  136.94] |    36
    ( 136.94,  281.67] |    36
    ( 281.67, 1132.69] |    36
```

```
spatial_lag_classes.yb
```

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

```
f = plt.figure(figsize=(15, 8))
ax = f.gca()

kw = dict(column='spatial_class', k=5, cmap='YlGn', alpha=1, legend=True, edgecolor='gray', linewidth=0.5,
          categorical=True)
boston.assign(spatial_class=spatial_lag_classes.yb).plot(ax=ax, **kw)
ax.set_axis_off()
```



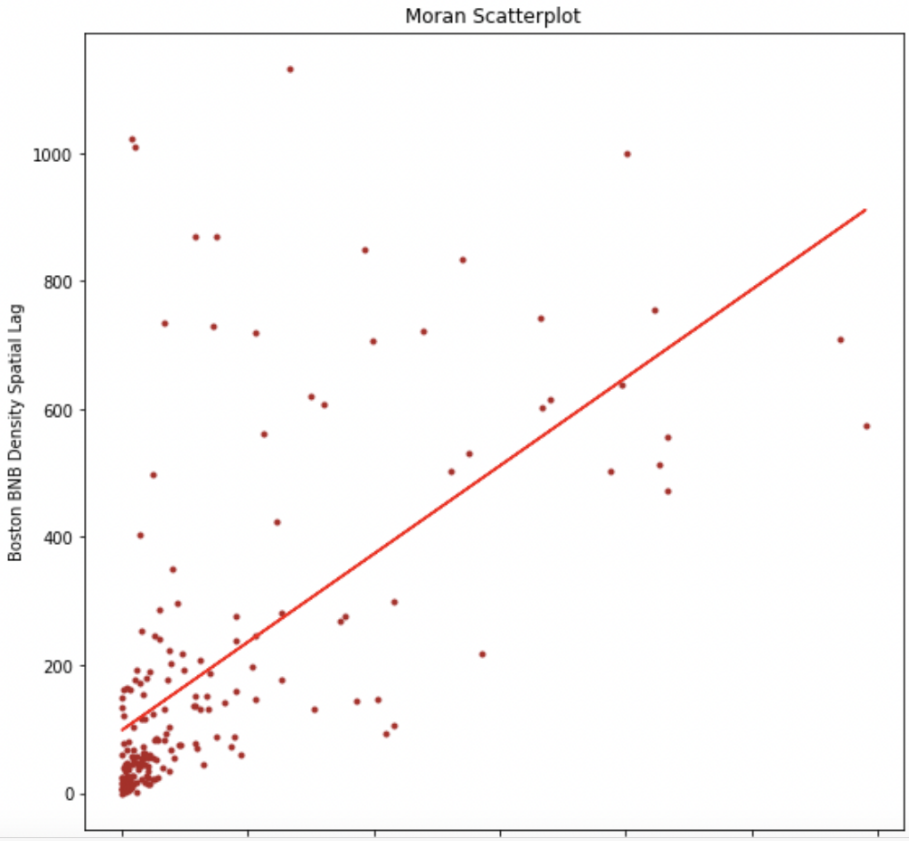
This geographic pattern is distinctly *not* random, strong evidence again that our data is geographically dependent ones.

We have another Plot called Moran Scatter Plot

The Moran scatter plot is a useful visual tool for exploratory analysis, because it enables you to assess how similar an observed value is to its neighboring observations. Its horizontal axis is based on the values of the observations and is also known as the response axis.

The tool of choice for this is the a Moran scatterplot. If geography matters our data should show a strong liner structure of some kind: high lag values correspond with high actual values, and low lags with low ones.

```
f, ax = plt.subplots(1, figsize=(9, 9))
plt.plot(boston['BNBDensity'], bnb_spatial_lags, '.', color='firebrick')

# Calculate and plot a line of best fit.
b,a = np.polyfit(boston['BNBDensity'], bnb_spatial_lags, 1)
plt.plot(boston['BNBDensity'], a + b*boston['BNBDensity'], 'r')

plt.title('Moran Scatterplot')
plt.ylabel('Boston BNB Density Spatial Lag')
plt.xlabel('Boston BNB Density Actual')
plt.show()
```



Now we can get p-value

```
[ ]  from esda.moran import Moran
```

We can get a p-value

```
moran = Moran(boston['BNBDensity'].values, qW)
moran.I, moran.p_sim
```

```
(0.5279124549042942, 0.001)
```

Our p-value of 0.001 tells us that there is a less than 0.1% chance that our data distribution is this heavily skewed by randomness alone.

SPATIAL CLUSTERING:

Spatial clustering aims to partition spatial data into a series of meaningful subclasses, called spatial clusters, such that spatial objects in the same cluster are similar to each other, and are dissimilar to those in different clusters.Whatever the analysis objective or the computational schema adopted, the clustering task heavily depends on the specific characteristics of the data considered. In particular, the spatio-temporal context is a large container, which includes several kinds of data types that exhibit extremely different properties and offer sensibly different opportunities of extracting useful knowledge. In this section we provide a taxonomy of the data types that are available in the spatio-temporal domain.

In our project Suppose that we're interested in segmenting the Boston AirBnB markets based on the types of properties available. This is, after all, a part of our dataset.

Leaving aside the "exotic" options, we've really got two major types of BnBs: apartments and houses.

```
listings['property_type'].value_counts()
```

```
Apartment          2612
House               562
Condominium         231
Townhouse            54
Bed & Breakfast      41
Loft                 39
Other                17
Boat                 12
Villa                 6
Entire Floor          4
Dorm                  2
Guesthouse            1
Camper/RV             1
Name: property_type, dtype: int64
```

```
apartments = listings.query('property_type == "Apartment" or property_type == "Condominium"').groupby('census_tract').count()['id']
houses = listings.query('property_type == "House" or property_type == "Townhouse"').groupby('census_tract').count()['id']
```

We have three predictor variables that we'd like to use: the number of apartments, the numbers of houses, and their density combined.

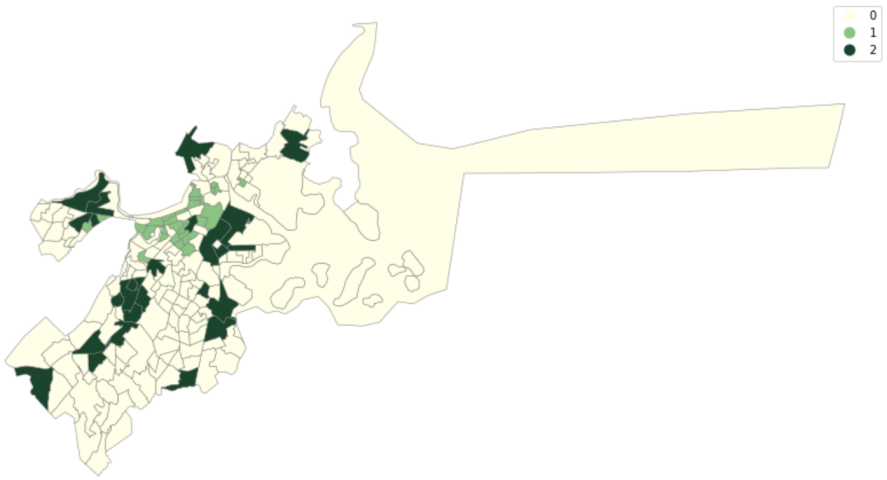
```
bnb_market = boston[['BNBDensity', 'BNBDensity_Houses', 'BNBDensity_Apartments']]
```

Now import the libraries of sklearn.cluster & sklearn.preprocessing for findings k-mean clusterings

```
import sklearn.cluster
import sklearn.preprocessing

clf = sklearn.cluster.KMeans(n_clusters=3)
X = sklearn.preprocessing.scale(bnb_market.values)
classes = clf.fit(X)
```

Now we're dealing with geometries, not just points in space! Our clustering algorithm doesn't know about this; it just naively classifies even very distant tracts into bins based on characteristic similarity alone. We can see this when we plot the result:

```
f = plt.figure(figsize=(15, 8))
ax = f.gca()

kw = dict(column='cluster', k=3, cmap='YlGn', alpha=1, legend=True, edgecolor='gray', linewidth=0.5, categorical=True)
boston.assign(cluster=classes.labels_).plot(ax=ax, **kw)
ax.set_axis_off()
```



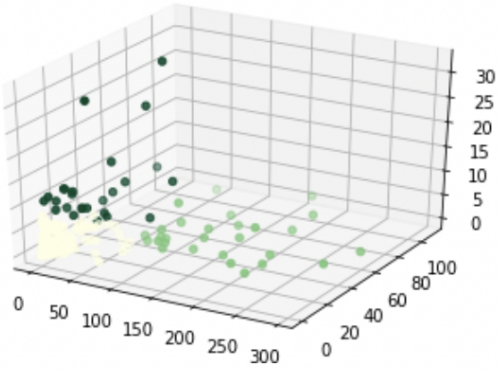
Our classifier does seem to have learned, in effect, the difference between low-density outlying tracts, medium-density tracts, and high-density inner city tracts:

```
[ ] bnb_market.assign(cluster=classes.labels_).groupby('cluster').mean()
```

cluster	BNBDensity	BNBDensity_Houses	BNBDensity_Apartments
0	12.233624	2.062992	6.740157
1	149.154101	1.571429	47.464286
2	31.235665	11.615385	23.884615

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(bnb_market['BNBDensity'], bnb_market['BNBDensity_Apartments'],
           zs=bnb_market['BNBDensity_Houses'], c=classes.labels_, cmap='YlGn')
```

```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7fad10bfe110>
```

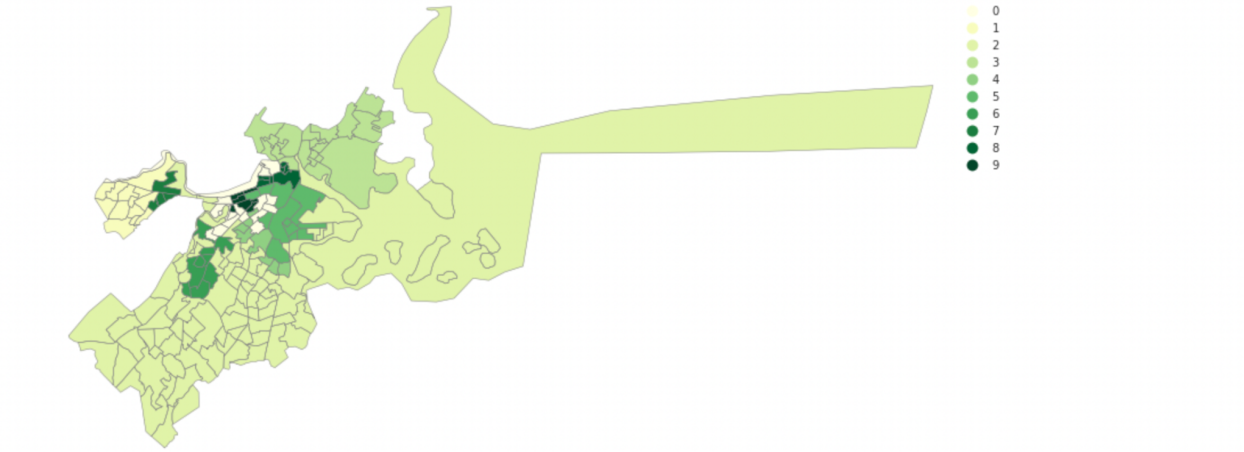


The pale census tracts are those which have very little AirBnB activity at all. They tend to be furthest from the city center. The second class of tracts, the dark green ones, have a significant BnB density, but consist primarily of houses, not apartments; these correspond with middle-of-the-pack residential areas. Finally, the densest areas, tracts containing plenty of apartments for rent but no homes and located in the heart of the city, are pale green.

except that the areas we've binned are discontinuous. Because our classifier disregards geography it doesn't put in any effort keeping. pysal contains classifiers that allow us to do just that. Chief amongst them is the Maxp clustering algorithm, which chooses clusters which match each other characteristically *and* are also all next to one another (according to whatever definition of neighbors you choose—we'll keep going with queen contiguity). Here's what we get when we run it. Notice the place of our spatial weights in the input, as well as a minimum number of observations per class—150.

```
clf = ps.region.Maxp(qW, bnb_market.values, 150, boston['BNBs'].values[:,None])
```

```
# n = 200
f = plt.figure(figsize=(15, 8))
ax = f.gca()

kw = dict(column='cluster', k=10, cmap='YlGn', alpha=1, legend=True, edgecolor='gray', linewidth=0.5, categorical=True)
boston.assign(cluster=clf.area2region.values()).plot(ax=ax, **kw)
ax.set_axis_off()
```



Because of the additional constraint the algorithm imposes, it tends to result in significantly more clusters than comparable non-spatial clustering algorithms. But, as you can see here, the algorithm did arrive at essentially the same conclusion as our kNN did: apartment-dense

innercities, house-dense middle burbs, and outlying areas.

```
bnb_market.assign(cluster=clf.area2region.values()).groupby('cluster').mean()
```

cluster	BNBDensity	BNBDensity_Houses	BNBDensity_Apartments
0	78.451566	1.764706	36.647059
1	16.821212	5.833333	11.166667
2	7.442751	2.882353	4.423529
3	21.453374	2.428571	10.142857
4	53.346294	3.375000	15.250000
5	29.089423	3.333333	29.333333
6	39.003481	10.000000	24.777778
7	83.277575	10.500000	39.750000
8	205.586501	1.285714	43.857143
9	166.723335	0.666667	49.500000

SPATIAL REGRESSION:

Regression (and prediction more generally) provides us a perfect case to examine how spatial structure can help us understand and analyze our data. In this chapter, we discuss how spatial structure can be used to both validate and improve prediction algorithms, focusing on linear regression specifically. Usually, spatial structure helps regression models in one of two ways. The first (and most clear) way space can have an impact on our data is when the process *generating* the data is itself explicitly spatial. Here, think of something like the prices for single family homes. It's often the case that individuals pay a premium on their house price in order to live in a better school district for the same quality house. Alternatively, homes closer to noise or chemical polluters like wastewater treatment plants, recycling facilities, or wide highways, may actually be cheaper than we would otherwise anticipate. In cases like asthma incidence, the locations individuals tend to travel to throughout the day, such as their places of work or recreation, may have more impact on their health than their residential addresses. In this case, it may be necessary to use data from *other sites* to predict the asthma incidence at a given site.

In this project we are interesting in modeling of price of Bnb, Here we taken explanatory variables are ['bathrooms', 'bedrooms', 'beds', 'guests_included']

One thing we can do is the need to merge together the price and cleaning fee into an actual_price, BnB hosts like to be sneaky and sock away as much as 50$ or more into that

"fee"!. But if you've run a basic linear regression before, this should be pretty rote. We're going to use statsmodels to do it.

```
explanatory_variables = ['bathrooms', 'bedrooms', 'beds', 'guests_included']
```

```
import statsmodels.api as sm

def convert_price(price):
    try:
        return float(price.replace("$", "").replace(".", "")[:-2])
    except (ValueError, AttributeError):
        return np.nan


price_variables = ['price', 'cleaning_fee']
geom = ['latitude', 'longitude']  # we'll need this later
valid_listings = listings[explanatory_variables + price_variables + geom].dropna()
valid_listings['actual_price'] = valid_listings['price'].map(convert_price) +\
                                 valid_listings['cleaning_fee'].map(convert_price)
valid_listings = valid_listings.dropna()

clf = sm.OLS(valid_listings['actual_price'], valid_listings[explanatory_variables])
basic_ols_res = clf.fit()
```

```
basic_ols_res.summary()
```

OLS Regression Results

Dep. Variable:	actual_price	**R-squared (uncentered):**	0.821
Model:	OLS	**Adj. R-squared (uncentered):**	0.820
Method:	Least Squares	**F-statistic:**	2808.
Date:	Sat, 21 May 2022	**Prob (F-statistic):**	0.00
Time:	08:09:59	**Log-Likelihood:**	-15293.
No. Observations:	2460	**AIC:**	3.059e+04
Df Residuals:	2456	**BIC:**	3.062e+04
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
bathrooms	86.6214	4.185	20.699	0.000	78.415	94.828
bedrooms	38.2459	4.722	8.099	0.000	28.986	47.505
beds	36.4452	3.642	10.006	0.000	29.303	43.588
guests_included	12.0422	2.316	5.199	0.000	7.500	16.585

Omnibus:	172.427	**Durbin-Watson:**	1.296
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	241.583
Skew:	0.592	**Prob(JB):**	3.48e-53
Kurtosis:	3.977	**Cond. No.**	7.49

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

From the above graph the R-squre is 82%

A common theme found in the field of spatial econometrics is the idea of mutual price-setting. That is, when someone sets the price of a spatial good, they take into account the price that good set by their neighbors, and vice versa.

If price-setting has a spatial element of this kind (a beating-the-Joneses effect of sorts), then we should be able to improve the results of our model by mixing in observations about nearby BnBs.

To do that we'll first need to grab ahold of what we mean by "neighbors". Since we're dealing with points, not shapes, contiguities don't apply here anymore; we in our usual k-nearest neighbor space. It's possible to grab our kNN from scipy, but it's much faster coding-wise to take a pysal built-in instead.

Then to run our spatial regression we simply pass in the same parameters as before, but now with the inclusion of our weights (w). name_x and name_y are optional parameters which simply add a little bit more information to the summary display

```
import spreg
geospatial_ols = spreg.GM_Lag(valid_listings['actual_price'].values[:, None],
                              valid_listings[explanatory_variables].values,
                              w=w,
                              name_x=explanatory_variables,
                              name_y='actual_price')
```

From below we can observe the summary of the geospatial analysis here the dependent variable is actual price and there are 2460 total number of observations in the geo_spatial analysis, the mean dependent variable of this analysis is 246.6 and pseudo R-squared is 0.2 for this model we can develop with different independent variables thus this must be valid but we have to develop the analysis by increasing the R-square value, and spatial Pseudo R- Square is 0.3265. Thus the further concluded analysis is by increasing the R- Square value for better spatial analysis..

```
print(geospatial_ols.summary)
```

```
REGRESSION
----------
SUMMARY OF OUTPUT: SPATIAL TWO STAGE LEAST SQUARES
--------------------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :actual_price               Number of Observations:        2460
Mean dependent var  :    246.6793               Number of Variables   :           6
S.D. dependent var  :    145.2765               Degrees of Freedom    :        2454
Pseudo R-squared    :      0.2996
Spatial Pseudo R-squared:  0.3265

------------------------------------------------------------------------------------
            Variable     Coefficient       Std.Error     z-Statistic     Probability
------------------------------------------------------------------------------------
            CONSTANT       74.5257846      11.4683038       6.4984139       0.0000000
           bathrooms       48.5721848       6.0617495       8.0128988       0.0000000
            bedrooms       39.4847512       4.7342175       8.3402908       0.0000000
                beds       35.5915425       3.6648137       9.7116921       0.0000000
     guests_included        5.9026820       2.4268024       2.4322878       0.0150038
      W_actual_price       -0.0212334       0.0196799      -1.0789394       0.2806148
------------------------------------------------------------------------------------
Instrumented: W_actual_price
Instruments: W_bathrooms, W_bedrooms, W_beds, W_guests_included
================================ END OF REPORT =====================================
```

So in this case we've shown pretty conclusively just the opposite: that the price of a BnB has little to do with the prices set by its neighbors.

```
geospatial_ols_mse = ((valid_listings['actual_price'] - geospatial_ols.predy.flatten()) ** 2).mean()
print("OLS".ljust(5), ":", basic_ols_res.mse_total)
print("OLS+W".ljust(5), ":", geospatial_ols_mse)
```

```
OLS   : 81947.35243902438
OLS+W : 14788.387329874082
```

Limitations:

Our approach has four main limitations, which will inform our future research agenda. First, regression and classification analyses cannot determine casual relationships. While we can justifiably argue that Airbnb's spatial penetration can be explained and predicted for the eight cities under study, we cannot establish any causal relationship. As future work, it would be interesting to perform a longitudinal study of Airbnb penetration and observe changes to city neighborhoods.

Second, even though our list of cities aimed at capturing a variety of socio-economic conditions, it is not comprehensive. Future work should replicate this study upon more cities, which are not necessarily in the U.S. and have not been early adopters of Airbnb (as most of our cities were). It would be interesting to test whether the growth dynamics in late-adopting cities is similar to those in early-adopting ones, that is, whether the presence of the creative class would still matter.

Finally, this study analyzes Airbnb spatial penetration from the point of view of 'offerings' only (i.e., number of Airbnb listings per area). We have already performed a similar study of Airbnb 'demand' (i.e., number of properties actually rented per area) on the same eight U.S. cities and we have found very similar results. An interesting future study would be to analyze Airbnb penetration while segmenting by users' demographics, to shed light onto the impact of the service on different classes of users (for example, on tourists vs. business travelers).

Conclusion:

This is the first time that Airbnb's adoption has been analyzed for a wide range of different U.S. cities example Boston we have used that data . We have extracted a variety of geographic, economic, and socio-demographic indicators and shown that, despite the US city analyzed being rather different in terms of ethnic composition and socio-economic characteristics, in most of them central areas with a strong presence of educated and creative people are those with highest Airbnb penetration. Finally, we have presented a generic prediction model for forecasting Airbnb penetration by exploiting a variety of indicators that also captures non-traditional socio-demographic dimensions such as the presence of creative workers in these

areas. We have shown that the proposed model can effectively be used to predict Airbnb penetration in the U.S. cities considered in this study.

Project Presentation Link:

https://drive.google.com/file/d/1wuDlx5CrG4_WiNr3WbME7I_aZD5c092T/view?usp=sharing