lab_04

September 20, 2016

1 Spatial weights

In this session we will be learning the ins and outs of one of the key pieces in spatial analysis: spatial weights matrices. These are structured sets of numbers that formalize geographical relationships between the observations in a dataset. Essentially, a spatial weights matrix of a given geography is a positive definite matrix of dimensions N by N, where N is the total number of observations:

$$W = \begin{pmatrix} 0 & w_{12} & \dots & w_{1N} \\ w_{21} & \ddots & w_{ij} & \vdots \\ \vdots & w_{ji} & 0 & \vdots \\ w_{N1} & \dots & \dots & 0 \end{pmatrix}$$

where each cell w_{ij} contains a value that represents the degree of spatial contact or interaction between observations i and j. A fundamental concept in this context is that of *neighbor* and *neighborhood*. By convention, elements in the diagonal (w_{ij}) are set to zero. A *neighbor* of a given observation i is another observation with which i has some degree of connection. In terms of W, i and j are thus neighbors if $w_{ij} > 0$. Following this logic, the neighborhood of i will be the set of observations in the system with which it has certain connection, or those observations with a weight greater than zero.

There are several ways to create such matrices, and many more to transform them so they contain an accurate representation that aligns with the way we understand spatial interactions between the elements of a system. In this session, we will introduce the most commonly used ones and will show how to compute them with `PySAL`.

```
In [1]: %matplotlib inline

        import seaborn as sns
        import pandas as pd
        import pysal as ps
        import geopandas as gpd
        import numpy as np
        import matplotlib.pyplot as plt
```

```
/Users/dani/anaconda/envs/gds/lib/python2.7/site-packages/matplotlib/font_manager.p
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take
```

1.1 Data

For this tutorial, we will use again the recently released 2015 Index of Multiple Deprivation (IMD) for England and Wales. This dataset can be most easily downloaded from the CDRC data store (link) and, since it already comes both in tabular as well as spatial data format (shapefile), it does not need merging or joining to additional geometries.

In addition, we will be using the lookup between LSOAs and Medium Super Output Areas (MSOAs), which can be downloaded on this link. This connects each LSOA polygon to the MSOA they belong to. MSOAs are a coarser geographic delineation from the Office of National Statistics (ONS), within which LSOAs are nested. That is, no LSOA boundary crosses any of an MSOA.

As usual, let us set the paths to the folders containing the files before anything so we can then focus on data analysis exclusively (keep in mind the specific paths will probably be different for your computer):

```
In [2]: # This will be different on your computer and will depend on where
        # you have downloaded the files
        imd_shp = '../../../../data/E08000012_IMD/shapefiles/E08000012.shp'
        lookup_path = '../../../../data/Output_areas_(2011)_to_lower_layer_super_ou
```

Let us load the IMD data first:

```
In [3]: # Read the file in
        imd = gpd.read_file(imd_shp)
        # Index it on the LSOA ID
        imd = imd.set_index('LSOA11CD')
        # Display summary
        imd.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Index: 298 entries, E01006512 to E01033768
Data columns (total 12 columns):
crime           298 non-null float64
education       298 non-null float64
employment      298 non-null float64
geometry        298 non-null object
health          298 non-null float64
housing         298 non-null float64
idaci           298 non-null float64
idaopi          298 non-null float64
imd_rank        298 non-null int64
imd_score       298 non-null float64
income          298 non-null float64
living_env      298 non-null float64
dtypes: float64(10), int64(1), object(1)
memory usage: 30.3+ KB
```

1.2 Building spatial weights in PySAL

1.2.1 Contiguity

Contiguity weights matrices define spatial connections through the existence of common boundaries. This makes it directly suitable to use with polygons: if two polygons share boundaries to some degree, they will be labeled as neighbors under these kinds of weights. Exactly how much they need to share is what differenciates the two approaches we will learn: queen and rook.

- **Queen**

Under the queen criteria, two observations only need to share a vortex (a single point) of their boundaries to be considered neighbors. Constructing a weights matrix under these principles can be done by running:

```
In [4]: w_queen = ps.queen_from_shapefile(imd_shp, idVariable='LSOA11CD')
        w_queen

Out[4]: <pysal.weights.weights.W at 0x1046096d0>
```

The command above creates an object `w_queen` of the class `W`. This is the format in which spatial weights matrices are stored in `PySAL`. Note also how we include the attribute `idVariable` and pass in `LSOA11CD`. This ensures that the observations are recorded using the ID codes we are used to, and also that we are using to index the table `imd`. This way we can keep everything in line easily.

A `W` object can be queried to find out about the contiguity relations it contains. For example, if we would like to know who is a neighbor of observation `E01006690`:

```
In [5]: w_queen['E01006690']

Out[5]: {u'E01006691': 1.0,
         u'E01006692': 1.0,
         u'E01006695': 1.0,
         u'E01006697': 1.0,
         u'E01006720': 1.0,
         u'E01006759': 1.0,
         u'E01033763': 1.0}
```

This returns a Python dictionary that contains the ID codes of each neighbor as keys, and the weights they are assigned as values. Since we are looking at a raw queen contiguity matrix, every neighbor gets a weight of one. If we want to access the weight of a specific neighbor, `E01006691` for example, we can do recursive querying:

```
In [6]: w_queen['E01006690']['E01006691']

Out[6]: 1.0
```

`W` objects also have a direct way to provide a list of all the neighbors or their weights for a given observation. This is thanks to the `neighbors` and `weights` attributes:

```
In [7]: w_queen.neighbors['E01006690']
```

```
Out[7]: [u'E01006720',
         u'E01033763',
         u'E01006691',
         u'E01006692',
         u'E01006695',
         u'E01006697',
         u'E01006759']
```

```
In [8]: w_queen.weights['E01006690']
```

```
Out[8]: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

Once created, `W` objects can provide much information about the matrix, beyond the basic attributes one would expect. We have direct access to the number of neighbors each observation has via the attribute `cardinalities`. For example, to find out how many neighbors observation `E01006524` has:

```
In [9]: w_queen.cardinalities['E01006524']
```

```
Out[9]: 6
```

Since `cardinalities` is a dictionary, it is direct to convert it into a `Series` object:

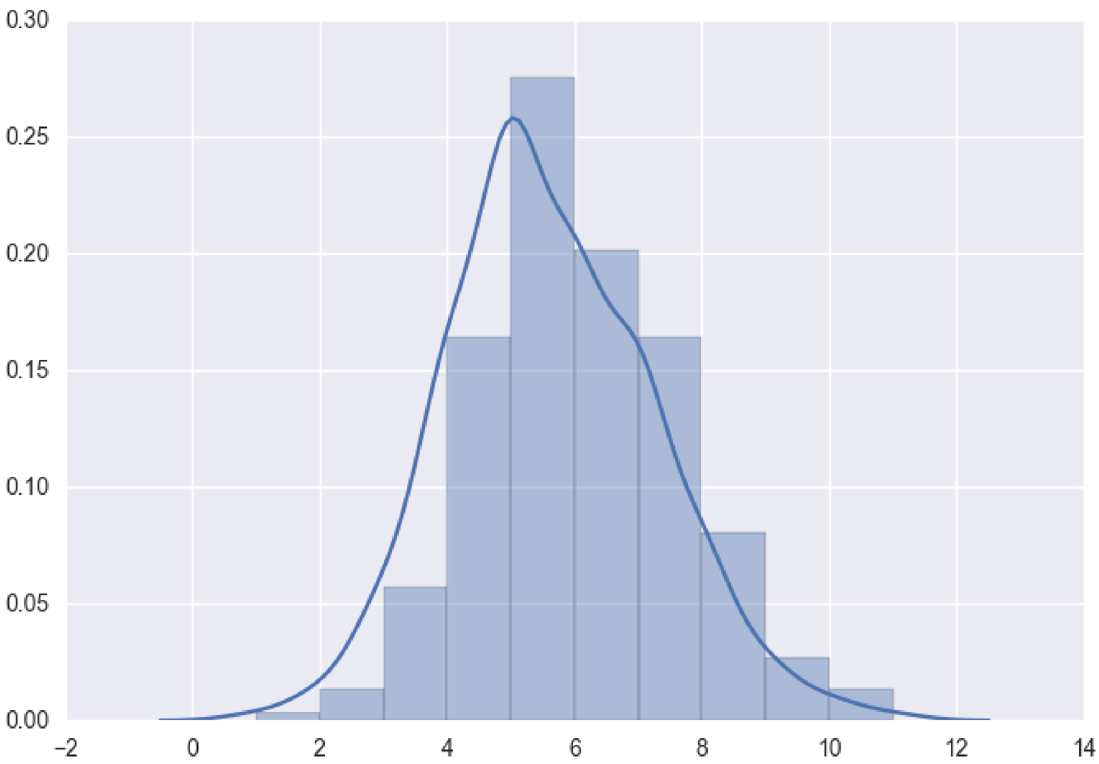
```
In [10]: queen_card = pd.Series(w_queen.cardinalities)
         queen_card.head()
```

```
Out[10]: E01006512    6
         E01006513    9
         E01006514    5
         E01006515    8
         E01006518    5
         dtype: int64
```

This allows, for example, to access quick plotting, which comes in very handy to get an overview of the size of neighborhoods in general:

```
In [11]: sns.distplot(queen_card, bins=10)
```

```
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x117d2ed50>
```



The figure above shows how most observations have around five neighbors, but there is some variation around it. The distribution also seems to follow a symmetric form, where deviations from the average occur both in higher and lower values almost evenly.

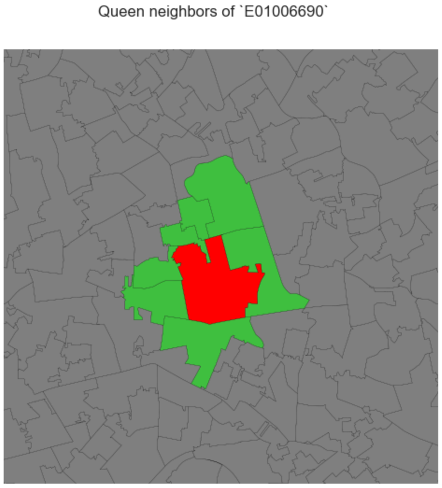
Some additional information about the spatial relationships contained in the matrix are also easily available from a `W` object. Let us tour over some of them:

```
In [12]: # Number of observations
         w_queen.n

Out[12]: 298

In [13]: # Average number of neighbors
         w_queen.mean_neighbors

Out[13]: 5.6174496644295306

In [14]: # Min number of neighbors
         w_queen.min_neighbors

Out[14]: 1

In [15]: # Max number of neighbors
         w_queen.max_neighbors

Out[15]: 11
```

```
In [16]: # Islands (observations disconnected)
         w_queen.islands

Out[16]: []

In [17]: # Order of IDs (first five only in this case)
         w_queen.id_order[:5]

Out[17]: [u'E01006512', u'E01006513', u'E01006514', u'E01006515', u'E01006518']
```

Spatial weight matrices can be explored visually in other ways. For example, we can pick an observation and visualize it in the context of its neighborhood. The following plot does exactly that by zooming into the surroundings of LSOA `E01006690` and displaying its polygon as well as those of its neighbors:

```
In [18]: # Setup figure
         f, ax = plt.subplots(1, figsize=(6, 6))
         # Plot base layer of polygons
         imd.plot(ax=ax, facecolor='k', linewidth=0.1)
         # Select focal polygon
         # NOTE we pass both the area code and the column name
         #      (`geometry`) within brackets!!!
         focus = imd.loc[['E01006690'], ['geometry']]
         # Plot focal polygon
         focus.plot(facecolor='red', alpha=1, linewidth=0, ax=ax)
         # Plot neighbors
         neis = imd.loc[w_queen['E01006690'], :]
         neis.plot(ax=ax, facecolor='lime', linewidth=0)
         # Title
         f.suptitle("Queen neighbors of `E01006690`")
         # Style and display on screen
         ax.set_axis_off()
         plt.axis('equal')
         ax.set_ylim(388000, 393500)
         ax.set_xlim(336000, 339500)
         plt.show()
```



Note how the figure is built gradually, from the base map (L. 4-5), to the focal point (L. 9), to its neighborhood (L. 11-13). Once the entire figure is plotted, we zoom into the area of interest (L. 19-20).

- **Rook**

Rook contiguity is similar to and, in many ways, superseded by queen contiguity. However, since it sometimes comes up in the literature, it is useful to know about it. The main idea is the same: two observations are neighbors if they share some of their boundary lines. However, in the rook case, it is not enough with sharing only one point, it needs to be at least a segment of their boundary. In most applied cases, these differences usually boil down to how the geocoding was done, but in some cases, such as when we use raster data or grids, this approach can differ more substantively and it thus makes more sense.

From a technical point of view, constructing a rook matrix is very similar:

```
In [19]: w_rook = ps.rook_from_shapefile(imd_shp, idVariable='LSOA11CD')
         w_rook
```

```
Out[19]: <pysal.weights.weights.W at 0x119d97ed0>
```

The output is of the same type as before, a `W` object that can be queried and used in very much the same way as any other one.

[Optional exercise]

Create a similar map for the rook neighbors of polygon `E01006580`.

How would it differ if the spatial weights were created based on the queen criterion?

1.2.2 Distance

Distance based matrices assign the weight to each pair of observations as a function of how far from each other they are. How this is translated into an actual weight varies across types and variants, but they all share that the ultimate reason why two observations are assigned some weight is due to the distance between them.

- **K-Nearest Neighbors**

One approach to define weights is to take the distances between a given observation and the rest of the set, rank them, and consider as neighbors the k closest ones. That is exactly what the k-nearest neighbors (KNN) criterium does.

To calculate KNN weights, we can use a similar function as before and derive them from a shapefile:

```
In [20]: knn5 = ps.knnW_from_shapefile(imd_shp, k=5, idVariable='LSOA11CD')
         knn5
```

```
Out[20]: <pysal.weights.weights.W at 0x119d4a610>
```

Note how we need to specify the number of nearest neighbors we want to consider with the argument `k`. Since it is a polygon shapefile that we are passing, the function will automatically compute the centroids to derive distances between observations. Alternatively, we can provide the points in the form of an array, skipping this way the dependency of a file on disk:

```
In [21]: # Extract centroids
         cents = imd.centroid
         # Extract coordinates into an array
         pts = np.array([(pt.x, pt.y) for pt in cents])
         # Compute KNN weights
         knn5_from_pts = ps.knnW_from_array(pts, k=5)
         knn5_from_pts

Out[21]: <pysal.weights.weights.W at 0x11ceb5210>
```

- **Distance band**

Another approach to build distance-based spatial weights matrices is to draw a circle of certain radious and consider neighbor every observation that falls within the circle. The technique has two main variations: binary and continuous. In the former one, every neighbor is given a weight of one, while in the second one, the weights can be further tweaked by the distance to the observation of interest.

To compute binary distance matrices in `PySAL`, we can use the following command:

```
In [22]: w_dist1kmB = ps.threshold_binaryW_from_shapefile(imd_shp,
                                                 1000, idVariable='LSOA11CD')
```

This creates a binary matrix that considers neighbors of an observation every polygon whose centroid is closer than 1,000 metres (1Km) of the centroid of such observation. Check, for example, the neighborhood of polygon `E01006690`:

```
In [23]: w_dist1kmB['E01006690']

Out[23]: {u'E01006691': 1,
          u'E01006692': 1,
          u'E01006695': 1,
          u'E01006697': 1,
          u'E01006720': 1,
          u'E01006725': 1,
          u'E01006726': 1,
          u'E01033763': 1}
```

Note that the units in which you specify the distance directly depend on the CRS in which the spatial data are projected, and this has nothing to do with the weights building but it can affect it significantly. Recall how you can check the CRS of a `GeoDataFrame`:

```
In [24]: imd.crs

Out[24]: {u'datum': u'OSGB36',
          u'k': 0.9996012717,
          u'lat_0': 49,
          u'lon_0': -2,
          u'no_defs': True,
          u'proj': u'tmerc',
          u'units': u'm',
          u'x_0': 400000,
          u'y_0': -100000}
```

In this case, you can see the unit is expressed in metres (`m`), hence we set the threshold to 1,000 for a circle of 1km of radious.

An extension of the weights above is to introduce further detail by assigning different weights to different neighbors within the radious circle based on how far they are from the observation of interest. For example, we could think of assigning the inverse of the distance between observations i and j as w_{ij}. This can be computed with the following command:

```
In [25]: w_dist1kmC = ps.threshold_continuousW_from_shapefile(imd_shp,
                                                         1000, idVariable='LSOA11CD')
```

In `w_dist1kmC`, every observation within the 1km circle is assigned a weight equal to the inverse distance between the two:

$$w_{ij} = \frac{1}{d_{ij}}$$

This way, the further apart i and j are from each other, the smaller the weight w_{ij} will be.

Contrast the binary neighborhood with the continuous one for `E01006690`:

```
In [26]: w_dist1kmC['E01006690']
```

```
Out[26]: {u'E01006691': 0.0013201155583939742,
          u'E01006692': 0.0016898105326562619,
          u'E01006695': 0.0011209237936802396,
          u'E01006697': 0.0014034695245779395,
          u'E01006720': 0.0013390450966253398,
          u'E01006725': 0.0010090443283278022,
          u'E01006726': 0.0010528397241717365,
          u'E01033763': 0.0012983249276301231}
```

[Optional exercise]

Explore the help for functions `ps.threshold_binaryW_from_array` and `ps.threshold_continuousW_from_array` and try to use them to replicate `w_dist1kmB` and `w_dist1kmC`.

Following this logic of more detailed weights through distance, there is a temptation to take it further and consider everyone else in the dataset as a neighbor whose weight will then get modulated by the distance effect shown above. However, although conceptually correct, this approach is not always the most computationally or practical one. Because of the nature of spatial weights matrices, particularly because of the fact their size is N by N, they can grow substantially large. A way to cope with this problem is by making sure they remain fairly *sparse* (with many zeros). Sparsity is typically ensured in the case of contiguity or KNN by construction but, with inverse distance, it needs to be imposed as, otherwise, the matrix could be potentially entirely dense (no zero values other than the diagonal). In practical terms, what is usually done is to impose a distance threshold beyond which no weight is assigned and interaction is assumed to be non-existent. Beyond being computationally feasible and scalable, results from this approach usually do not differ much from a fully "dense" one as the additional information that is included from further observations is almost ignored due to the small weight they receive. In this context, a commonly used threshold, although not always best, is that which makes every observation to have at least one neighbor.

Such a threshold can be calculated as follows:

```
In [27]: min_thr = ps.min_threshold_dist_from_shapefile(imd_shp)
         min_thr
```

```
Out[27]: 939.73728445042354
```

Which can then be used to calculate an inverse distance weights matrix:

```
In [28]: w_min_dist = ps.threshold_continuousW_from_shapefile(imd_shp,
                                          min_thr, idVariable='LSOA11CD')
```

```
WARNING: there is one disconnected observation (no neighbors)
Island id:  [214]
```

NOTE The warning that it returns appears to be a precission issue caused by the point being exactly at the threshold distance. One solution until this gets resolved is to make the threshold marginally larger:

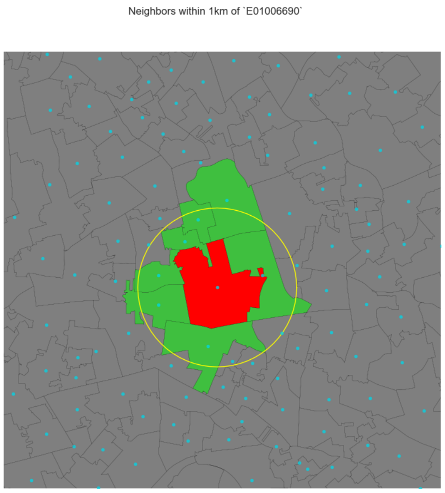
```
In [29]: # Add a marginally small amount to the threshold
         new_thr = min_thr + (0.000001 * min_thr)
         # Compute weights matrix
         w_min_dist = ps.threshold_continuousW_from_shapefile(imd_shp,
                                          new_thr, idVariable='LSOA11CD')
```

[Optional extension. Lecture figure]

Below is how to build a visualization for distance-based weights that displays the polygons, highlighting the focus and its neighbors, and then overlays the centroids and the buffer used to decide whether a polygon is a neighbor or not. Since this is distance-based weights, there needs to be a way to establish distance between two polygons and, in this case, the distance between their centroids is used.

```
In [30]: # Setup figure
         f, ax = plt.subplots(1, figsize=(9, 9))
         # Plot base layer of polygons
         imd.plot(ax=ax, facecolor='k', linewidth=0.1)
         # Select focal polygon
         # NOTE we pass both the area code and the column name
         #      (`geometry`) within brackets!!!
         focus = imd.loc[['E01006690'], ['geometry']]
         # Plot focal polygon
         focus.plot(facecolor='red', alpha=1, linewidth=0, ax=ax)
```

```
# Plot neighbors
neis = imd.loc[w_queen['E01006690'], :]
neis.plot(ax=ax, facecolor='lime', linewidth=0)
# Plot 1km buffer
buf = focus.centroid.buffer(1000)
buf.plot(edgecolor='yellow', alpha=0, ax=ax)
# Plot centroids of neighbor
pts = np.array([(pt.x, pt.y) for pt in imd.centroid])
ax.plot(pts[:, 0], pts[:, 1], color='#00d8ea',
        linewidth=0, alpha=0.75, marker='o', markersize=4)
#imd.centroid.plot(axes=ax)
# Title
f.suptitle("Neighbors within 1km of `E01006690`")
# Style, zoom and display on screen
ax.set_axis_off()
plt.axis('equal')
ax.set_ylim(388000, 393500)
ax.set_xlim(336000, 339500)
plt.show()
```


Neighbors within 1km of `E01006690`

1.2.3 Block weights

Block weights connect every observation in a dataset that belongs to the same category in a list provided ex-ante. Usually, this list will have some relation to geography an the location of the observations but, technically speaking, all one needs to create block weights is a list of memberships. In this class of weights, neighboring observations, those in the same group, are assigned a weight of one, and the rest receive a weight of zero.

In this example, we will build a spatial weights matrix that connects every LSOA with all the other ones in the same MSOA. To do this, we first need a lookup list that connects both kinds of geographies:

```
In [31]: # NOTE: disregard the warning in pink that might come from running
         #       this cell
         lookup = pd.read_csv(lookup_path+'OA11_LSOA11_MSOA11_LAD11_EW_LUv2.csv')
         lookup = lookup[['LSOA11CD', 'MSOA11CD']].drop_duplicates(keep='last')\
                                                  .set_index('LSOA11CD')['MSOA11CD'
         lookup.head()

/Users/dani/anaconda/envs/gds/lib/python2.7/site-packages/IPython/core/interactives
  interactivity=interactivity, compiler=compiler, result=result)


Out[31]: LSOA11CD
         E01000002    E02000001
         E01032740    E02000001
         E01000005    E02000001
         E01000009    E02000017
         E01000008    E02000016
         Name: MSOA11CD, dtype: object
```

Since the original file contains much more information than we need for this exercise, note how in line 2 we limit the columns we keep to only two, `LSOA11CD` and `MSOA11CD`. We also add an additional command, `drop_duplicates`, which removes elements whose index is repeated more than once, as is the case in this dataset (every LSOA has more than one row in this table). By adding the `take_last` argument, we make sure that one and only one element of each index value is retained. For ease of use later on, we set the index, that is the name of the rows, to `LSOA11CD`. This will allow us to perform easy lookups without having to perform full `DataFrame` queries, and it is also a more computationally efficient way to select observations.

For example, if we want to know in which MSOA the polygon `E01000003` is, we just need to type:

```
In [32]: lookup.loc['E01000003']

Out[32]: 'E02000001'
```

With the lookup in hand, let us append it to the IMD table to keep all the necessary pieces in one place only:

```
In [33]: imd['MSOA11CD'] = lookup
```

Now we are ready to build a block spatial weights matrix that connects as neighbors all the LSOAs in the same MSOA. Using `PySAL`, this is a one-line task:

```
In [34]: w_block = ps.block_weights(imd['MSOA11CD'])
```

In this case, `PySAL` does not allow to pass the argument `idVariable` as above. As a result, observations are named after the the order the occupy in the list:

```
In [35]: w_block[0]
```

```
Out[35]: {218: 1.0, 219: 1.0, 220: 1.0, 292: 1.0}
```

The first element is neighbor of observations 218, 129, 220, and 292, all of them with an assigned weight of 1. However, it is easy enough to correct this by using the additional method `remap_ids`:

```
In [36]: w_block.remap_ids(imd.index)
```

Now if you try `w_bloc[0]`, it will return an error. But if you query for the neighbors of an observation by its LSOA id, it will work:

```
In [37]: w_block['E01006512']
```

```
Out[37]: {u'E01006747': 1.0, u'E01006748': 1.0, u'E01006751': 1.0, u'E01033763': 1.
```

[Optional exercise]
For block weights, create a similar map to that of queen neighbors of polygon `E01006690`.

1.3 Standardizing `W` matrices

In the context of many spatial analysis techniques, a spatial weights matrix with raw values (e.g. ones and zeros for the binary case) is not always the best suiting one for analysis and some sort of transformation is required. This implies modifying each weight so they conform to certain rules. `PySAL` has transformations baked right into the `W` object, so it is easy to check the state of an object as well as to modify it.

Consider the original queen weights, for observation `E01006690`:

```
In [38]: w_queen['E01006690']
```

```
Out[38]: {u'E01006691': 1.0,
          u'E01006692': 1.0,
          u'E01006695': 1.0,
          u'E01006697': 1.0,
          u'E01006720': 1.0,
          u'E01006759': 1.0,
          u'E01033763': 1.0}
```

Since it is contiguity, every neighbor gets one, the rest zero weight. We can check if the object `w_queen` has been transformed or not by calling the argument `transform`:

```
In [39]: w_queen.transform
```

```
Out[39]: 'O'
```

where `O` stands for "original", so no transformations have been applied yet. If we want to apply a row-based transformation, so every row of the matrix sums up to one, we modify the `transform` attribute as follows:

```
In [40]: w_queen.transform = 'R'
```

Now we can check the weights of the same observation as above and find they have been modified:

```
In [41]: w_queen['E01006690']
```

```
Out[41]: {u'E01006691': 0.14285714285714285,
          u'E01006692': 0.14285714285714285,
          u'E01006695': 0.14285714285714285,
          u'E01006697': 0.14285714285714285,
          u'E01006720': 0.14285714285714285,
          u'E01006759': 0.14285714285714285,
          u'E01033763': 0.14285714285714285}
```

Save for precission issues, the sum of weights for all the neighbors is one:

```
In [42]: pd.Series(w_queen['E01006690']).sum()
```

```
Out[42]: 0.99999999999999978
```

Returning the object back to its original state is as simple as assigning `transform` back to original:

```
In [43]: w_queen.transform = 'O'
```

```
In [44]: w_queen['E01006690']
```

```
Out[44]: {u'E01006691': 1.0,
          u'E01006692': 1.0,
          u'E01006695': 1.0,
          u'E01006697': 1.0,
          u'E01006720': 1.0,
          u'E01006759': 1.0,
          u'E01033763': 1.0}
```

`PySAL` supports the following transformations:

- `O`: original, returning the object to the initial state.
- `B`: binary, with every neighbor having assigned a weight of one.
- `R`: row, with all the neighbors of a given observation adding up to one.
- `V`: variance stabilizing, with the sum of all the weights being constrained to the number of observations.

1.4 Reading and Writing spatial weights in **PySAL**

Sometimes, if a dataset is very detailed or large, it can be costly to build the spatial weights matrix of a given geography and, despite the optimizations in the PySAL code, the computation time can quickly grow out of hand. In these contexts, it is useful to not have to re-build a matrix from scratch every time we need to re-run the analysis. A useful solution in this case is to build the matrix once, and save it to a file where it can be reloaded at a later stage if needed.

PySAL has a common way to write any kind of W object into a file using the command open. The only element we need to decide for ourselves beforehand is the format of the file. Although there are several formats in which spatial weight matrices can be stored (have a look at the list of supported ones by PySAL), we will focused on the two most commonly used ones:

- **.gal** files for contiguity weights

Contiguity spatial weights can be saved into a .gal file with the following commands:

```
In [45]: # Open file to write into
         fo = ps.open('imd_queen.gal', 'w')
         # Write the matrix into the file
         fo.write(w_queen)
         # Close the file
         fo.close()
```

The process is composed by the following three steps:

1. Open a target file for writing the matrix, hence the w argument. In this case, if a file imd_queen.gal already exists, it will be overwritten, so be careful.
2. Write the W object into the file.
3. Close the file. This is important as some additional information is written into the file at this stage, so failing to close the file might have unintended consequences.

Once we have the file written, it is possible to read it back into memory with the following command:

```
In [46]: w_queen2 = ps.open('imd_queen.gal', 'r').read()
         w_queen2

Out[46]: <pysal.weights.weights.W at 0x11a9c1850>
```

Note how we now use r instead of w because we are reading the file, and also notice how we open the file and, in the same line, we call read() directly.

- **.gwt** files for distance-based weights.

A very similar process to the one above can be used to read and write distance based weights. The only difference is specifying the right file format, .gwt in this case. So, if we want to write w_dist1km into a file, we will run:

```
In [47]: # Open file
         fo = ps.open('imd_dist1km.gwt', 'w')
         # Write matrix into the file
         fo.write(w_dist1kmC)
         # Close file
         fo.close()
```

And if we want to read the file back in, all we need to do is:

```
In [48]: w_dist1km2 = ps.open('imd_dist1km.gwt', 'r').read()
```

```
/Users/dani/code/pysal_darribas/pysal/core/IOHandlers/gwt.py:148: RuntimeWarning: D
  warn("DBF relating to GWT was not found, proceeding with unordered string ids.",
```

Note how, in this case, you will probably receive a warning alerting you that there was not a DBF relating to the file. This is because, by default, PySAL takes the order of the observations in a .gwt from a shapefile. If this is not provided, PySAL cannot entirely determine all the elements and hence the resulting W might not be complete (islands, for example, can be missing). To fully complete the reading of the file, we can remap the ids as we have seen above:

```
In [49]: w_dist1km2.remap_ids(imd.index)
```

1.5 Spatial Lag

One of the most direct applications of spatial weight matrices is the so-called *spatial lag*. The spatial lag of a given variable is the product of a spatial weight matrix and the variable itself:

$$Y_{sl} = WY$$

where Y is a Nx1 vector with the values of the variable. Recall that the product of a matrix and a vector equals the sum of a row by column element multiplication for the resulting value of a given row. In terms of the spatial lag:

$$y_{sl-i} = \sum_j w_{ij} y_j$$

If we are using row-standardized weights, w_{ij} becomes a proportion between zero and one, and y_{sl-i} can be seen as the average value of Y in the neighborhood of i.

The spatial lag is a key element of many spatial analysis techniques, as we will see later on and, as such, it is fully supported in PySAL. To compute the spatial lag of a given variable, imd_score for example:

```
In [50]: # Row-standardize the queen matrix
         w_queen.transform = 'R'
         # Compute spatial lag of `imd_score`
         w_queen_score = ps.lag_spatial(w_queen, imd['imd_score'])
         # Print the first five elements
         w_queen_score[:5]
```

```
Out[50]: array([ 48.27833333,  34.96777778,  46.538     ,  40.02375   ,  63.738
```

Line 4 contains the actual computation, which is highly optimized in PySAL. Note that, despite passing in a pd.Series object, the output is a numpy array. This however, can be added directly to the table imd:

```
In [51]: imd['w_queen_score'] = w_queen_score
```

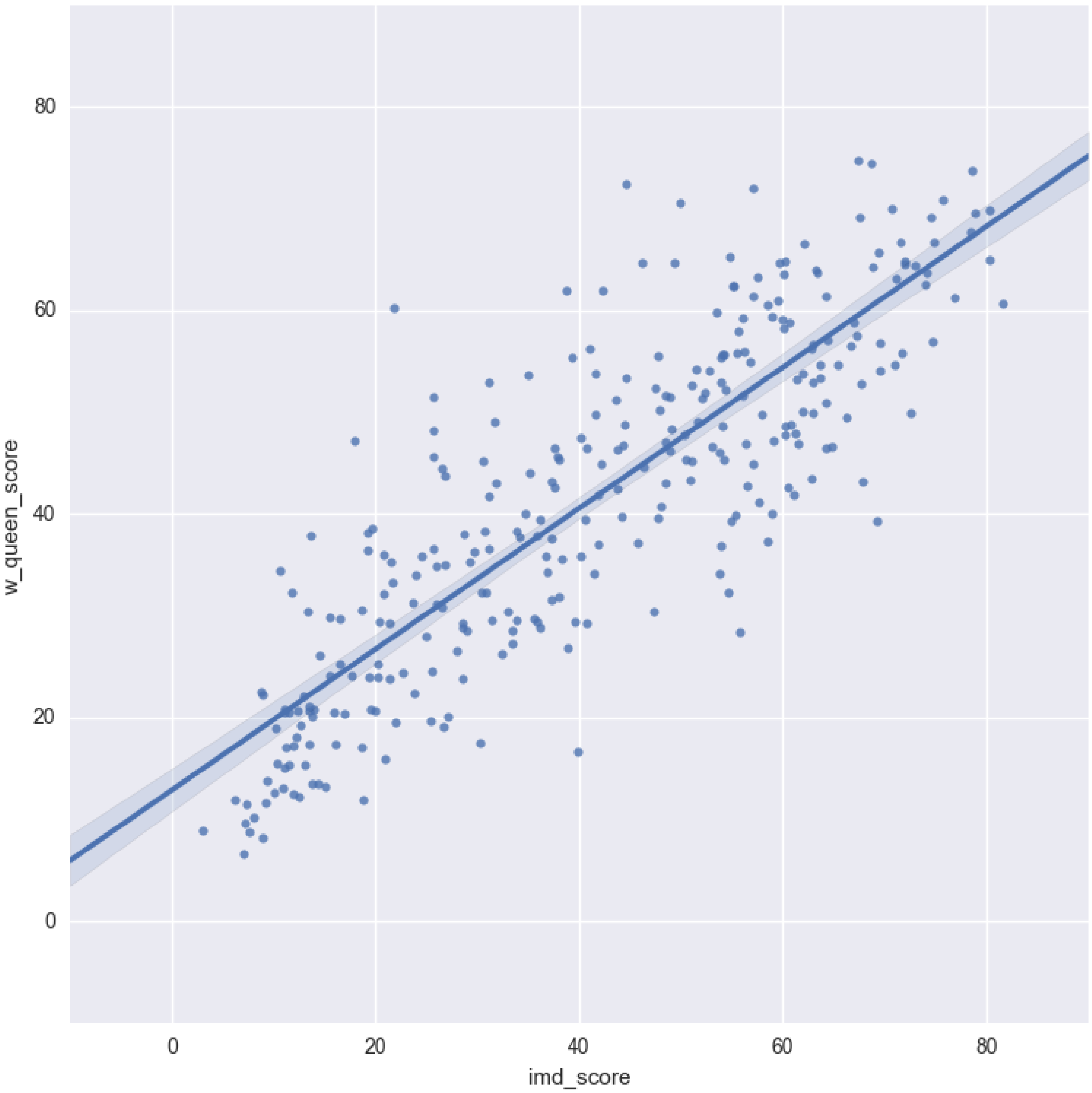
[Optional exercise]

Explore the spatial lag of w_queen_score by constructing a density/histogram plot similar to those created in Lab 2. Compare these with one for imd_score. What differences can you tell?

1.6 Moran Plot

The Moran Plot is a graphical way to start exploring the concept of spatial autocorrelation, and it is an easy application of spatial weight matrices and the spatial lag. In essence, it is a simple scatter plot in which a given variable (imd_score, for example) is plotted against *its own* spatial lag. Usually, a fitted line is added to include more information:

```
In [52]: # Setup the figure and axis
         f, ax = plt.subplots(1, figsize=(9, 9))
         # Plot values
         sns.regplot(x="imd_score", y="w_queen_score", data=imd)
         # Display
         plt.show()
```



In order to easily compare different scatter plots and spot outlier observations, it is common practice to standardize the values of the variable before computing its spatial lag and plotting it. This can be accomplished by substracting the average value and dividing the result by the standard deviation:

$$z_i = \frac{y - \bar{y}}{\sigma_y}$$

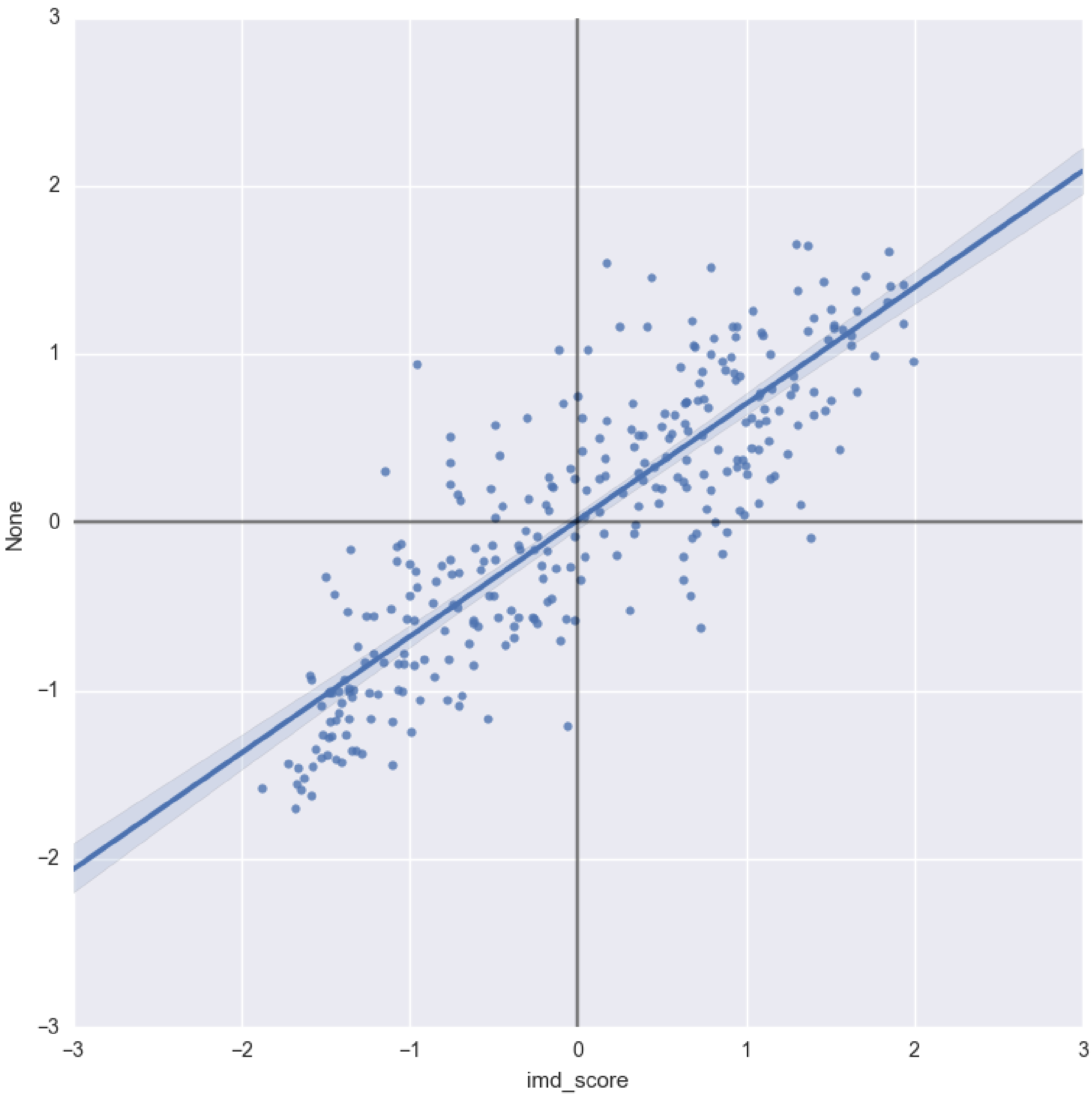
where z_i is the standardized version of y_i, $\bar{y}$ is the average of the variable, and σ its standard deviation.

Creating a standardized Moran Plot implies that average values are centered in the plot (as they are zero when standardized) and dispersion is expressed in standard deviations, with the rule of thumb of values greater or smaller than two standard deviations being *outliers*. A standardized Moran Plot also partitions the space into four quadrants that represent different situations:

1. High-High (*HH*): values above average surrounded by values above average.
2. Low-Low (*LL*): values below average surrounded by values below average.
3. High-Low (*HL*): values above average surrounded by values below average.
4. Low-High (*LH*): values below average surrounded by values above average.

These will be further explored once spatial autocorrelation has been properly introduced in subsequent lectures.

```
In [53]: # Standardize the IMD scores
         std_imd = (imd['imd_score'] - imd['imd_score'].mean()) / imd['imd_score'].
         # Compute the spatial lag of the standardized version and save is as a
         # Series indexed as the original variable
         std_w_imd = pd.Series(ps.lag_spatial(w_queen, std_imd), index=std_imd.inde
         # Setup the figure and axis
         f, ax = plt.subplots(1, figsize=(9, 9))
         # Plot values
         sns.regplot(x=std_imd, y=std_w_imd)
         # Add vertical and horizontal lines
         plt.axvline(0, c='k', alpha=0.5)
         plt.axhline(0, c='k', alpha=0.5)
         # Display
         plt.show()
```



[Optional exercise]
Create a standardized Moran Plot for each of the components of the IMD:

- Crime
- Education
- Employment
- Health
- Housing
- Income
- Living environment

Bonus if you can generate all the plots with a `for` loop.

Bonus-II if you explore the functionality of Seaborn's `jointplot` (link and link) to create a richer Moran plot.
