## 1 PySAL Components

1.1 Installation ........................................................................................................ 3
  1.1.1 Installing the released version ........................................................................ 3
  1.1.2 Installing development version ...................................................................... 4
  1.1.3 Installing versions supporting Python 2 .......................................................... 4
1.2 API Reference ........................................................................................................ 4
  1.2.1 *pysal.lib*: PySAL Core .................................................................................. 4
  1.2.2 *pysal.explore*: Exploratory spatial data analysis .......................................... 86
  1.2.3 *pysal.viz*: Geovisualization ......................................................................... 169
  1.2.4 *pysal.model*: Linear models for spatial data analysis .................................. 232
1.3 References ........................................................................................................... 321
1.4 Development ........................................................................................................ 321
1.5 Getting Involved ................................................................................................. 321
1.6 Bug reports ......................................................................................................... 321
1.7 Citing PySAL ....................................................................................................... 321
1.8 License information .............................................................................................. 322

Bibliography ............................................................................................................. 323
PySAL 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
PySAL Components

- **explore** - modules to conduct exploratory analysis of spatial and spatio-temporal data, including statistical testing on points, networks, and polygonal lattices. Also includes methods for spatial inequality and distributional dynamics.
- **viz** - visualize patterns in spatial data to detect clusters, outliers, and hot-spots.
- **model** - model spatial relationships in data with a variety of linear, generalized-linear, generalized-additive, and nonlinear models.
- **lib** - solve a wide variety of computational geometry problems:
  - graph construction from polygonal lattices, lines, and points.
  - construction and interactive editing of spatial weights matrices & graphs
  - computation of alpha shapes, spatial indices, and spatial-topological relationships
  - reading and writing of sparse graph data, as well as pure python readers of spatial vector data.

1.1 Installation

As of version 2.0.0, PySAL supports python 3.5 and 3.6 only. Please make sure that you are operating in a python 3 environment.

1.1.1 Installing the released version

PySAL is available on the Python Package Index. Therefore, you can either install directly with `pip` from the command line:

```
pip install -U pysal
```

or download the source distribution (.tar.gz) and decompress it to your selected destination. Open a command shell and navigate to the decompressed folder. Type:
1.1.2 Installing development version

Potentially, you might want to use the newest features in the development version of PySAL on github - pysal/pysal while have not been incorporated in the Pypi released version. You can achieve that by installing pysal/pysal by running the following from a command shell:

```
pip install git+https://github.com/pysal/pysal.git
```

You can also fork the pysal/pysal repo and create a local clone of your fork. By making changes to your local clone and submitting a pull request to pysal/PySAL, you can contribute to the PySAL development.

1.1.3 Installing versions supporting Python 2

Users requiring Python 2 support can install the legacy version of PySAL: 1.1.14 via pip:

```
pip install pysal==1.14.4.post2
```

Note that this version is only receiving bug fixes. All new enhancements to PySAL are Python 3+ only, and are not available in 1.14.4.

1.2 API Reference

This is the class and function reference of pysal.

1.2.1 pysal.lib: PySAL Core

Spatial Weights

```
weights.W(neighbors[, weights, id_order, ...])  # Spatial weights class.
```

```
pysal.lib.weights.W
```

```
class pysal.lib.weights.W(neighbors, weights=None, id_order=None, silence_warnings=False, ids=None)
```

Spatial weights class.

**Parameters**

- **neighbors** [dictionary] Key is region ID, value is a list of neighbor IDS. Example: `{'a':["b"], 'b':["a","c"], 'c':["b"]}
- **weights** [dictionary] Key is region ID, value is a list of edge weights. If not supplied all edge weights are assumed to have a weight of 1. Example: `{'a':[0.5], 'b':[0.5,1.5], 'c':[1.5]}
- **id_order** [list] An ordered list of ids, defines the order of observations when iterating over W if not set, lexicographical ordering is used to iterate and the id_order_set property will return False. This can be set after creation by setting the 'id_order' property.
silent_island_warning: boolean  By default pysal.lib will print a warning if the dataset contains any disconnected observations or islands. To silence this warning set this parameter to True.

silent_connected_components [boolean] By default PySAL will print a warning if the dataset contains any disconnected components in the adjacency matrix. These are disconnected groups of islands. To silence this warning set this parameter to True.

ids [list] Values to use for keys of the neighbors and weights dicts.

Attributes (NOTE: these are described by their docstrings. to view, use the ‘help’ function)

———-
asymmetries
cardinalities
component_labels
diagW2
diagWtW
diagWtW_WW
histogram
id2i
id_order
id_order_set
islands
max_neighbors
mean_neighbors
min_neighbors
n
n_components
neighbor_offsets
nonzero
pct_nonzero
s0
s1
s2
s2array
sd
sparse
trcW2
trcWtW
trcWtW_WW
transform

Examples

```python
>>> from pysal.lib.weights.weights import W
>>> neighbors = {0: [3, 1], 1: [0, 4, 2], 2: [1, 5], 3: [0, 6, 4], 4: [1, 3, 7, 5], 5: [2, 4, 8], 6: [3, 7], 7: [4, 6, 8], 8: [5, 7]}
>>> weights = {0: [1, 1], 1: [1, 1, 1], 2: [1, 1, 1], 3: [1, 1, 1], 4: [1, 1, 1, 1], 5: [1, 1, 1], 6: [1, 1, 1], 7: [1, 1, 1, 1], 8: [1, 1]}
>>> w = W(neighbors, weights)
>>> "%.3f"%w.pct_nonzero
'29.630'
```

Read from external gal file

```python
>>> import pysal.lib
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("stl.gal")).read()
>>> w.n
78
>>> "%.3f"%w.pct_nonzero
'6.542'
```

Set weights implicitly

```python
>>> neighbors = {0: [3, 1], 1: [0, 4, 2], 2: [1, 5], 3: [0, 6, 4], 4: [1, 3, 7, 5], 5: [2, 4, 8], 6: [3, 7], 7: [4, 6, 8], 8: [5, 7]}
>>> w = W(neighbors)
>>> round(w.pct_nonzero,3)
29.63
>>> from pysal.lib.weights import lat2W
>>> w = lat2W(100, 100)
>>> w.trcW2
39600.0
>>> w.trcW2tW
39600.0
>>> w.transform='r'
>>> round(w.trcW2, 3)
2530.722
>>> round(w.trcW2tW, 3)
2533.667
```

Cardinality Histogram

```python
>>> w.histogram [(2, 4), (3, 392), (4, 9604)]
```

Disconnected observations (islands)

```python
>>> from pysal.lib.weights import W
>>> w = W((1:[0], 0:[1], 2:[], 3:[]))
```

WARNING: there are 2 disconnected observations Island ids: [2, 3]

Attributes

- **asymmetries** List of id pairs with asymmetric weights.
- **cardinalities** Number of neighbors for each observation.
- **component_labels** Store the graph component in which each observation falls.
- **diagW2** Diagonal of WW.
**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asymmetry([intrinsic])</code></td>
<td>Asymmetry check.</td>
</tr>
<tr>
<td><code>from_adjlist(adjlist[, focal_col, ...])</code></td>
<td>Return an adjacency list representation of a weights object.</td>
</tr>
<tr>
<td><code>from_networkx(graph[, weight_col])</code></td>
<td>Convert a networkx graph to a PySAL W object.</td>
</tr>
<tr>
<td><code>full()</code></td>
<td>Generate a full numpy array.</td>
</tr>
<tr>
<td><code>get_transform()</code></td>
<td>Getter for transform property.</td>
</tr>
<tr>
<td><code>plot(gdf[, indexed_on, ax, color, node_kws, ...])</code></td>
<td>Plot spatial weights objects.</td>
</tr>
</tbody>
</table>

Continued on next page
### remap_ids(new_ids)

In place modification throughout $W$ of id values from $w.id\_order$ to new_ids in all set_shapefile(shapefile[, idVariable, full])

Adding meta data for writing headers of gal and gwt files.

set_transform([value])

Transformations of weights.

symmetrize([inplace])

Construct a symmetric KNN weight.

to_WSP()

Generate a WSP object.

to_adjlist([remove_symmetric, focal_col, ...])

Compute an adjacency list representation of a weights object.

to_networkx()

Convert a weights object to a networkx graph

<table>
<thead>
<tr>
<th>from_WSP</th>
<th>from_file</th>
<th>from_shapefile</th>
</tr>
</thead>
</table>

__init__(neighbors, weights=None, id_order=None, silence_warnings=False, ids=None)

Initialize self. See help(type(self)) for accurate signature.

## Methods

__init__(neighbors[, weights, id_order, ...])

Initialize self.

asymmetry([intrinsic])

Asymmetry check.

from_WSP(WSP[, silence_warnings])

Return an adjacency list representation of a weights object.

from_adjlist(adjlist[, focal_col, ...])

Convert a networkx graph to a PySAL W object.

from_file([path, format])

Generate a full numpy array.

from_networkx(graph[, weight_col])

Get the transform property.

plot(gdf[, indexed_on, ax, color, node_kws, ...])

Plot spatial weights objects.

remap_ids(new_ids)

In place modification throughout $W$ of id values from $w.id\_order$ to new_ids in all

full()

Adding meta data for writing headers of gal and gwt files.

set_shapefile(shapefile[, idVariable, full])

Transformations of weights.

symmetrize([inplace])

Construct a symmetric KNN weight.

to_WSP()

Generate a WSP object.

to_adjlist([remove_symmetric, focal_col, ...])

Compute an adjacency list representation of a weights object.

to_networkx()

Convert a weights object to a networkx graph

### Attributes

asymmetries

List of id pairs with asymmetric weights.

cardinalities

Number of neighbors for each observation.

Continued on next page
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>component_labels</td>
<td>Store the graph component in which each observation falls.</td>
</tr>
<tr>
<td>diagW2</td>
<td>Diagonal of $WW$.</td>
</tr>
<tr>
<td>diagWtW</td>
<td>Diagonal of $W^TW$.</td>
</tr>
<tr>
<td>diagWtW_WW</td>
<td>Diagonal of $W^TW + WW$.</td>
</tr>
<tr>
<td>histogram</td>
<td>Cardinality histogram as a dictionary where key is the id and value is the number of neighbors for that unit.</td>
</tr>
<tr>
<td>id2i</td>
<td>Dictionary where the key is an ID and the value is that ID's index in Wi.id_order.</td>
</tr>
<tr>
<td>id_order</td>
<td>Returns the ids for the observations in the order in which they would be encountered if iterating over the weights.</td>
</tr>
<tr>
<td>id_order_set</td>
<td>Returns True if user has set id_order, False if not.</td>
</tr>
<tr>
<td>islands</td>
<td>List of ids without any neighbors.</td>
</tr>
<tr>
<td>max_neighbors</td>
<td>Largest number of neighbors.</td>
</tr>
<tr>
<td>mean_neighbors</td>
<td>Average number of neighbors.</td>
</tr>
<tr>
<td>min_neighbors</td>
<td>Minimum number of neighbors.</td>
</tr>
<tr>
<td>n</td>
<td>Number of units.</td>
</tr>
<tr>
<td>n_components</td>
<td>Store whether the adjacency matrix is fully connected.</td>
</tr>
<tr>
<td>neighbor_offsets</td>
<td>Given the current id_order, neighbor_offsets[id] is the offsets of the id’s neighbors in id_order.</td>
</tr>
<tr>
<td>nonzero</td>
<td>Number of nonzero weights.</td>
</tr>
<tr>
<td>pct_nonzero</td>
<td>Percentage of nonzero weights.</td>
</tr>
<tr>
<td>s0</td>
<td>s0 is defined as</td>
</tr>
<tr>
<td>s1</td>
<td>s1 is defined as</td>
</tr>
<tr>
<td>s2</td>
<td>s2 is defined as</td>
</tr>
<tr>
<td>s2array</td>
<td>Individual elements comprising s2.</td>
</tr>
<tr>
<td>sd</td>
<td>Standard deviation of number of neighbors.</td>
</tr>
<tr>
<td>sparse</td>
<td>Sparse matrix object.</td>
</tr>
<tr>
<td>transform</td>
<td>Getter for transform property.</td>
</tr>
<tr>
<td>trcW2</td>
<td>Trace of $WW$.</td>
</tr>
<tr>
<td>trcWtW</td>
<td>Trace of $W^TW$.</td>
</tr>
<tr>
<td>trcWtW_WW</td>
<td>Trace of $W^TW + WW$.</td>
</tr>
</tbody>
</table>

### Distance Weights

- `weights.DistanceBand(data, threshold[, p, ...])`: Spatial weights based on distance band.
- `weights.Kernel(data[, bandwidth, fixed, k, ...])`: Spatial weights based on kernel functions.
- `weights.KNN(data[, k, p, ids, radius, ...])`: Creates nearest neighbor weights matrix based on k nearest neighbors.

#### pysal.lib.weights.DistanceBand

- `class pysal.lib.weights.DistanceBand(data, threshold, p=2, alpha=-1.0, binary=True, ids=None, build_sp=True, silence_warnings=False, distance_metric='euclidean', radius=None)`: Spatial weights based on distance band.

1.2. API Reference
**data** [array] (n,k) or KDTree where KDTree.data is array (n,k) n observations on k characteristics used to measure distances between the n objects

**threshold** [float] distance band

**p** [float] Minkowski p-norm distance metric parameter: 1<=p<=infinity

1: Manhattan distance

**binary** [boolean] If true w_{ij}=1 if d_{i,j}<=threshold, otherwise w_{ij}=0 If false wij=dij^{alpha}

**alpha** [float] distance decay parameter for weight (default -1.0) if alpha is positive the weights will not decline with distance. If binary is True, alpha is ignored

**ids** [list] values to use for keys of the neighbors and weights dicts

**build_sp** [boolean] True to build sparse distance matrix and false to build dense distance matrix; significant speed gains may be obtained dending on the sparsity of the of distance_matrix and threshold that is applied

**silent** [boolean] By default pysal.lib will print a warning if the dataset contains any disconnected observations or islands. To silence this warning set this parameter to True.

### Notes

This was initially implemented running scipy 0.8.0dev (in epd 6.1). earlier versions of scipy (0.7.0) have a logic bug in scipy/sparse/dok.py so serge changed line 221 of that file on sal-dev to fix the logic bug.

### Examples

```python
>>> import pysal.lib
>>> points=[[10, 10], [20, 10], [40, 10], [15, 20], [30, 20], [30, 30]]
>>> wcheck = pysal.lib.weights.W({0: [1, 3], 1: [0, 3], 2: [], 3: [0, 1], 4: [5], 5: [4]})
WARNING: there is one disconnected observation (no neighbors) Island id: [2] >>> w=pysal.lib.weights.distance.DistanceBand(points,threshold=11.2)
WARNING: there is one disconnected observation (no neighbors) Island id: [2] >>> pysal.lib.weights.util.neighbor_equality(w, wcheck) True
>>> w=pysal.lib.weights.distance.DistanceBand(points,threshold=14.2)
>>> wcheck = pysal.lib.weights.W({0: [1, 3], 1: [0, 3, 4], 2: [4], 3: [1, 0], 4: [5, 2, 1], 5: [4]})
>>> pysal.lib.weights.util.neighbor_equality(w, wcheck) True
```

inverse distance weights

```python
>>> w=pysal.lib.weights.distance.DistanceBand(points,threshold=11.2,binary=False)
WARNING: there is one disconnected observation (no neighbors) Island id: [2] >>> w.weights[0] [0.1, 0.08944271909999159] >>> w.neighbors[0].tolist() [1, 3]
```

gravity weights

```python
>>> w=pysal.lib.weights.distance.DistanceBand(points,threshold=11.2,binary=False, alpha=-2.)
WARNING: there is one disconnected observation (no neighbors) Island id: [2] >>> w.weights[0] [0.01, 0.00799999999999998]
```
Attributes

**weights**  [dict] of neighbor weights keyed by observation id

**neighbors**  [dict] of neighbors keyed by observation id

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asymmetry([intrinsic])</td>
<td>Asymmetry check.</td>
</tr>
<tr>
<td>from_adjlist(adjlist[, focal_col, ...])</td>
<td>Return an adjacency list representation of a weights object.</td>
</tr>
<tr>
<td>from_array(array, threshold, **kwargs)</td>
<td>Construct a DistanceBand weights from an array.</td>
</tr>
<tr>
<td>from_dataframe(df, threshold[, geom_col, ids])</td>
<td>Make DistanceBand weights from a dataframe.</td>
</tr>
<tr>
<td>from_networkx(graph[, weight_col])</td>
<td>Convert a networkx graph to a PySAL W object.</td>
</tr>
<tr>
<td>from_shapefile(filepath[, idVariable])</td>
<td>Distance-band based weights from shapefile</td>
</tr>
<tr>
<td>full()</td>
<td>Generate a full numpy array.</td>
</tr>
<tr>
<td>get_transform()</td>
<td>Getter for transform property.</td>
</tr>
<tr>
<td>plot(gdf[, indexed_on, ax, color, node_kws, ...])</td>
<td>Plot spatial weights objects.</td>
</tr>
<tr>
<td>remap_ids(new_ids)</td>
<td>In place modification throughout W of id values from w.id_order to new_ids in all</td>
</tr>
<tr>
<td>set_shapefile(shapefile[, idVariable, full])</td>
<td>Adding meta data for writing headers of gal and gwt files.</td>
</tr>
<tr>
<td>set_transform([value])</td>
<td>Transformations of weights.</td>
</tr>
<tr>
<td>symmetrize([inplace])</td>
<td>Construct a symmetric KNN weight.</td>
</tr>
<tr>
<td>to_WSP()</td>
<td>Generate a WSP object.</td>
</tr>
<tr>
<td>to_adjlist([remove_symmetric, focal_col, ...])</td>
<td>Compute an adjacency list representation of a weights object.</td>
</tr>
<tr>
<td>to_networkx()</td>
<td>Convert a weights object to a networkx graph</td>
</tr>
</tbody>
</table>

Casting to floats is a work around for a bug in scipy.spatial. See detail in pysal issue #126.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(data, threshold[, p, alpha, ...])</td>
<td>Casting to floats is a work around for a bug in scipy.spatial.</td>
</tr>
<tr>
<td>asymmetry([intrinsic])</td>
<td>Asymmetry check.</td>
</tr>
<tr>
<td>from_WSP(WSP[, silence_warnings])</td>
<td>Return an adjacency list representation of a weights object.</td>
</tr>
<tr>
<td>from_adjlist(adjlist[, focal_col, ...])</td>
<td>Construct a DistanceBand weights from an array.</td>
</tr>
<tr>
<td>from_array(array, threshold, **kwargs)</td>
<td>Make DistanceBand weights from a dataframe.</td>
</tr>
<tr>
<td>from_dataframe(df, threshold[, geom_col, ids])</td>
<td>Convert a networkx graph to a PySAL W object.</td>
</tr>
<tr>
<td>from_file([path, format])</td>
<td></td>
</tr>
</tbody>
</table>
from_shapefile(filepath, threshold[, idVariable])

Generate a full numpy array.

get_transform()  

Getter for transform property.

plot(gdf[, indexed_on, ax, color, node_kws, ...])  

Plot spatial weights objects.

remap_ids(new_ids)

In place modification throughout W of id values from w.id_order to new_ids in all

set_shapefile(shapefile[, idVariable, full])

Adding meta data for writing headers of gal and gwt files.

set_transform([value])  

Transformations of weights.

symmetrize([inplace])

Construct a symmetric KNN weight.

to_WSP()  

Generate a WSP object.

to_adjlist([remove_symmetric, focal_col, ...])

Compute an adjacency list representation of a weights object.

to_networkx()  

Convert a weights object to a networkx graph

**Attributes**

asymmetries List of id pairs with asymmetric weights.

cardinalities Number of neighbors for each observation.

component_labels Store the graph component in which each observation falls.

diagW2 Diagonal of $W W$.

diagWtW Diagonal of $W W^t$.

diagWtW_W Diagonal of $W W^t + W W$.

histogram Cardinality histogram as a dictionary where key is the id and value is the number of neighbors for that unit.

id2i Dictionary where the key is an ID and the value is that ID’s index in W.id_order.

id_order Returns the ids for the observations in the order in which they would be encountered if iterating over the weights.

id_order_set Returns True if user has set id_order, False if not.

islands List of ids without any neighbors.

max_neighbors Largest number of neighbors.

mean_neighbors Average number of neighbors.

min_neighbors Minimum number of neighbors.

n Number of units.

n_components Store whether the adjacency matrix is fully connected.

neighbor_offsets Given the current id_order, neighbor_offsets[id] is the offsets of the id’s neighbors in id_order.

nonzero Number of nonzero weights.

pct_nonzero Percentage of nonzero weights.

s0 is defined as

s1 is defined as

s2 is defined as

s2array Individual elements comprising s2.

sd Standard deviation of number of neighbors.
### Table 8 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sparse</code></td>
<td>Sparse matrix object.</td>
</tr>
<tr>
<td><code>transform</code></td>
<td>Getter for transform property.</td>
</tr>
<tr>
<td><code>trcW2</code></td>
<td>Trace of $WW$.</td>
</tr>
<tr>
<td><code>trcWtW</code></td>
<td>Trace of $W'W$.</td>
</tr>
<tr>
<td><code>trcWtW_WW</code></td>
<td>Trace of $W'W + WW$.</td>
</tr>
</tbody>
</table>

#### pysal.lib.weights.Kernel

```python
class pysal.lib.weights.Kernel(data, bandwidth=None, fixed=True, k=2, function='triangular', eps=1.0000001, ids=None, distance_metric='euclidean', radius=None, **kwargs)
```

Spatial weights based on kernel functions.

**Parameters**

- `data` [array] (n,k) or KDTree where KDtree.data is array (n,k) n observations on k characteristics used to measure distances between the n objects
- `bandwidth` [float] or array-like (optional) the bandwidth $h_i$ for the kernel.
- `fixed` [binary] If true then $h_i = h \forall i$. If false then bandwidth is adaptive across observations.
- `k` [int] the number of nearest neighbors to use for determining bandwidth. For fixed bandwidth, $h_i = \max(dknn/i)$ where $dknn$ is a vector of k-nearest neighbor distances (the distance to the kth nearest neighbor for each observation). For adaptive bandwidths, $h_i = dknn_i$
- `diagonal` [boolean] If true, set diagonal weights = 1.0, if false (default), diagonals weights are set to value according to kernel function.
- `function` [{‘triangular’,’uniform’,’quadratic’,’quartic’,’gaussian’}] kernel function defined as follows with 

\[ z_{i,j} = \frac{d_{i,j}}{h_i} \]

- **triangular**
  \[ K(z) = (1 - |z|) \text{ if } |z| \leq 1 \]
- **uniform**
  \[ K(z) = 1/2 \text{ if } |z| \leq 1 \]
- **quadratic**
  \[ K(z) = \frac{3}{4}(1 - z^2) \text{ if } |z| \leq 1 \]
- **quartic**
  \[ K(z) = \frac{15}{16}(1 - z^2)^2 \text{ if } |z| \leq 1 \]
- **gaussian**
  \[ K(z) = (2\pi)^{-1/2}\exp(-z^2/2) \]
- `eps` [float] adjustment to ensure knn distance range is closed on the knnth observations

**Examples**
>>> points=[(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
>>> kw=Kernel(points)
>>> kw.weights[0]
[1.0, 0.500000049999995, 0.4409830615267465]
>>> kw.neighbors[0]
[0, 1, 3]
>>> kw.bandwidth
array([[20.000002,]
       [20.000002,]
       [20.000002,]
       [20.000002,]
       [20.000002,]])
>>> kw15=Kernel(points,bandwidth=15.0)
>>> kw15[0]
{0: 1.0, 1: 0.33333333333333337, 3: 0.2546440075000701}
>>> kw15.neighbors[0]
[0, 1, 3]
>>> kw15.bandwidth
array([[15.],
       [15.],
       [15.],
       [15.],
       [15.]]))

Adaptive bandwidths user specified

>>> bw=[25.0,15.0,25.0,16.0,14.5,25.0]
>>> kwa=Kernel(points,bandwidth=bw)
>>> kwa.weights[0]
[1.0, 0.6, 0.552786404500042, 0.10557280900008403]
>>> kwa.neighbors[0]
[0, 1, 3, 4]
>>> kwa.bandwidth
array([[25. ],
       [15. ],
       [25. ],
       [16. ],
       [14.5],
       [25. ]])

Endogenous adaptive bandwidths

>>> kwea=Kernel(points,fixed=False)
>>> kwa.weights[0]
[1.0, 0.10557289844279438, 9.999990663795e-08]
>>> kwa.neighbors[0]
[0, 1, 3]
>>> kwa.bandwidth
array([[11.18034101,]
       [11.18034101,]
       [20.000002 ],
       [11.18034101,]
       [14.14213704],
       [18.02775818]]))

Endogenous adaptive bandwidths with Gaussian kernel
>>> kweag = Kernel(points, fixed=False, function='gaussian')
>>> kweag.weights[0]
[0.3989422804014327, 0.2674190291577696, 0.241970487162134]
>>> kweag.bandwidth
array([[[11.0034101],
        [11.0034101],
        [20.0000002],
        [11.0034101],
        [14.14213704],
        [18.02775818]]])

Diagonals to 1.0

>>> kq = Kernel(points, function='gaussian')
>>> kq.weights
{0: [0.3989422804014327, 0.35206533556593145, 0.3412334260702758], 1: [0.35206533556593145, 0.3989422804014327, 0.241970487162134, 0.3412334260702758, 0.31069657591175387], 2: [0.241970487162134, 0.3989422804014327, 0.31069657591175387], 3: [0.31069657591175387, 0.3989422804014327, 0.241970487162134, 0.3412334260702758, 0.31069657591175387], 4: [0.31069657591175387, 0.3011374490937829, 0.26575287272131043, 0.35206533556593145, 0.3989422804014327], 5: [0.26575287272131043, 0.35206533556593145, 0.31069657591175387, 0.3011374490937829, 0.26575287272131043, 0.35206533556593145, 0.3989422804014327]}

>>> kqd = Kernel(points, function='gaussian', diagonal=True)
>>> kqd.weights
{0: [1.0, 0.35206533556593145, 0.3412334260702758], 1: [0.35206533556593145, 1.0, 0.241970487162134, 0.3412334260702758, 0.31069657591175387], 2: [0.241970487162134, 1.0, 0.31069657591175387], 3: [0.31069657591175387, 0.3011374490937829, 0.26575287272131043, 0.35206533556593145, 0.3989422804014327], 4: [0.31069657591175387, 0.3011374490937829, 0.26575287272131043, 0.35206533556593145, 1.0], 5: [0.26575287272131043, 0.35206533556593145, 0.31069657591175387, 0.3011374490937829, 1.0]}

Attributes

- **weights** [dict] Dictionary keyed by id with a list of weights for each neighbor
- **neighbors** [dict] of lists of neighbors keyed by observation id
- **bandwidth** [array] array of bandwidths

Methods

- **asymmetry**([intrinsic]) Asymmetry check.
- **from_adjlist**(adjlist[, focal_col, ...]) Return an adjacency list representation of a weights object.
- **from_array**(array,**kwargs) Construct a Kernel weights from an array.
- **from_dataframe**(df[, geom_col, ids]) Make Kernel weights from a dataframe.
- **from_networkx**(graph[, weight_col]) Convert a networkx graph to a PySAL W object.
- **from_shapefile**(filepath[, idVariable]) Kernel based weights from shapefile
- **full**() Generate a full numpy array.
- **get_transform**() Getter for transform property.
- **plot**(gdf[, indexed_on, ax, color, node_kws, ...]) Plot spatial weights objects.
- **remap_ids**(new_ids) In place modification throughout W of id values from w.id_order to new_ids in all

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_shapefile(shapefile[, idVariable, full])</td>
<td>Adding meta data for writing headers of gal and gwt files.</td>
</tr>
<tr>
<td>set_transform([value])</td>
<td>Transformations of weights.</td>
</tr>
<tr>
<td>symmetrize([inplace])</td>
<td>Construct a symmetric KNN weight.</td>
</tr>
<tr>
<td>to_WSP()</td>
<td>Generate a WSP object.</td>
</tr>
<tr>
<td>to_adjlist((remove_symmetric, focal_col, ...))</td>
<td>Compute an adjacency list representation of a weights object.</td>
</tr>
<tr>
<td>to_networkx()</td>
<td>Convert a weights object to a networkx graph</td>
</tr>
</tbody>
</table>

**from_WSP**

**from_file**

```python
__init__ (data, bandwidth=None, fixed=True, k=2, function='triangular', eps=1.0000001, ids=None, diagonal=False, distance_metric='euclidean', radius=None, **kwargs)
```

Initialize self. See help(type(self)) for accurate signature.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong> ([data[, bandwidth, fixed, k, ...]])</td>
<td>Initialize self.</td>
</tr>
<tr>
<td>asymmetry([intrinsic])</td>
<td>Asymmetry check.</td>
</tr>
<tr>
<td>from_WSP(WSP[, silence_warnings])</td>
<td>Return an adjacency list representation of a weights object.</td>
</tr>
<tr>
<td>from_adjlist(adjlist[, focal_col, ...])</td>
<td>Construct a Kernel weights from an array.</td>
</tr>
<tr>
<td>from_array(array, **kwargs)</td>
<td>Make Kernel weights from a dataframe.</td>
</tr>
<tr>
<td>from_dataframe(df[, geom_col, ids])</td>
<td>Convert a networkx graph to a PySAL W object.</td>
</tr>
<tr>
<td>from_shapefile(filepath[, idVariable])</td>
<td>Convert a Kernel weights from shapefile</td>
</tr>
<tr>
<td>full()</td>
<td>Generate a full numpy array.</td>
</tr>
<tr>
<td>get_transform()</td>
<td>Getter for transform property.</td>
</tr>
<tr>
<td>plot(gdf[, indexed_on, ax, color, node_kws, ...])</td>
<td>Plot spatial weights objects.</td>
</tr>
<tr>
<td>remap_ids(new_ids)</td>
<td>In place modification throughout W of id values from w.id_order to new_ids in all</td>
</tr>
<tr>
<td>set_shapefile(shapefile[, idVariable, full])</td>
<td>Adding meta data for writing headers of gal and gwt files.</td>
</tr>
<tr>
<td>set_transform([value])</td>
<td>Transformations of weights.</td>
</tr>
<tr>
<td>symmetrize([inplace])</td>
<td>Construct a symmetric KNN weight.</td>
</tr>
<tr>
<td>to_WSP()</td>
<td>Generate a WSP object.</td>
</tr>
<tr>
<td>to_adjlist((remove_symmetric, focal_col, ...))</td>
<td>Compute an adjacency list representation of a weights object.</td>
</tr>
<tr>
<td>to_networkx()</td>
<td>Convert a weights object to a networkx graph</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asymmetries</td>
<td>List of id pairs with asymmetric weights.</td>
</tr>
<tr>
<td>cardinalities</td>
<td>Number of neighbors for each observation.</td>
</tr>
</tbody>
</table>
Table 11 – continued from previous page

<table>
<thead>
<tr>
<th><strong>component_labels</strong></th>
<th>Store the graph component in which each observation falls.</th>
</tr>
</thead>
<tbody>
<tr>
<td>diagW2</td>
<td>Diagonal of $WW$.</td>
</tr>
<tr>
<td>diagWtW</td>
<td>Diagonal of $W'W$.</td>
</tr>
<tr>
<td>diagWtW_WW</td>
<td>Diagonal of $W'W + WW$.</td>
</tr>
<tr>
<td>histogram</td>
<td>Cardinality histogram as a dictionary where key is the id and value is the number of neighbors for that unit.</td>
</tr>
<tr>
<td>id2i</td>
<td>Dictionary where the key is an ID and the value is that ID's index in W.id_order.</td>
</tr>
<tr>
<td>id_order</td>
<td>Returns the ids for the observations in the order in which they would be encountered if iterating over the weights.</td>
</tr>
<tr>
<td>id_order_set</td>
<td>Returns True if user has set id_order, False if not.</td>
</tr>
<tr>
<td>islands</td>
<td>List of ids without any neighbors.</td>
</tr>
<tr>
<td>max_neighbors</td>
<td>Largest number of neighbors.</td>
</tr>
<tr>
<td>mean_neighbors</td>
<td>Average number of neighbors.</td>
</tr>
<tr>
<td>min_neighbors</td>
<td>Minimum number of neighbors.</td>
</tr>
<tr>
<td>n</td>
<td>Number of units.</td>
</tr>
<tr>
<td>n_components</td>
<td>Store whether the adjacency matrix is fully connected.</td>
</tr>
<tr>
<td>neighbor_offsets</td>
<td>Given the current id_order, neighbor_offsets[id] is the offsets of the id's neighbors in id_order.</td>
</tr>
<tr>
<td>nonzero</td>
<td>Number of nonzero weights.</td>
</tr>
<tr>
<td>pct_nonzero</td>
<td>Percentage of nonzero weights.</td>
</tr>
<tr>
<td>s0</td>
<td>s0 is defined as</td>
</tr>
<tr>
<td>s1</td>
<td>s1 is defined as</td>
</tr>
<tr>
<td>s2</td>
<td>s2 is defined as</td>
</tr>
<tr>
<td>s2array</td>
<td>Individual elements comprising s2.</td>
</tr>
<tr>
<td>sd</td>
<td>Standard deviation of number of neighbors.</td>
</tr>
<tr>
<td>sparse</td>
<td>Sparse matrix object.</td>
</tr>
<tr>
<td>transform</td>
<td>Getter for transform property.</td>
</tr>
<tr>
<td>trcW2</td>
<td>Trace of $WW$.</td>
</tr>
<tr>
<td>trcWtW</td>
<td>Trace of $W'W$.</td>
</tr>
<tr>
<td>trcWtW_WW</td>
<td>Trace of $W'W + WW$.</td>
</tr>
</tbody>
</table>

**pysal.lib.weights.KNN**

```python
class pysal.lib.weights.KNN(data, k=2, p=2, ids=None, radius=None, distance_metric='euclidean', **kwargs)
```

Creates nearest neighbor weights matrix based on k nearest neighbors.

**Parameters**

- **kdtree** [object] PySAL KDTree or ArcKDTree where KDtree.data is array (n,k) n observations on k characteristics used to measure distances between the n objects
- **k** [int] number of nearest neighbors
- **p** [float] Minkowski p-norm distance metric parameter: 1<=p<=infinity 2: Euclidean distance 1: Manhattan distance Ignored if the KDTree is an ArcKDTree
- **ids** [list] identifiers to attach to each observation
Returns

\[ w \] instance Weights object with binary weights

See also:

pysal.lib.weights.weights.W

Notes

Ties between neighbors of equal distance are arbitrarily broken.

Examples

```python
>>> import pysal.lib
>>> points = [(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
>>> kd = pysal.lib.cg.kdtree.KDTree(np.array(points))
>>> wnn2 = pysal.lib.weights.KNN(kd, 2)
>>> [1,3] == wnn2.neighbors[0]
True
```

```python
>>> ids = wnn2 = KNN(kd,2)
>>> wnn2[0]
{1: 1.0, 3: 1.0}
>>> wnn2[1]
{0: 1.0, 3: 1.0}
```

now with 1 rather than 0 offset

```python
>>> wnn2 = pysal.lib.weights.KNN(kd, 2, ids=range(1,7))
>>> wnn2[1]
{2: 1.0, 4: 1.0}
>>> wnn2[2]
{1: 1.0, 4: 1.0}
>>> 0 in wnn2.neighbors
False
```

Attributes

- **asymmetries** List of id pairs with asymmetric weights.
- **cardinalities** Number of neighbors for each observation.
- **component_labels** Store the graph component in which each observation falls.
- **diagW2** Diagonal of \( W \).
- **diagWtW** Diagonal of \( W'W \).
- **diagWtW_WW** Diagonal of \( W'W + WW \).
- **histogram** Cardinality histogram as a dictionary where key is the id and value is the number of neighbors for that unit.
- **id2i** Dictionary where the key is an ID and the value is that ID’s index in Wid_order.
- **id_order** Returns the ids for the observations in the order in which they would be encountered if iterating over the weights.
id_order_set  Returns True if user has set id_order, False if not.

islands  List of ids without any neighbors.

max_neighbors  Largest number of neighbors.

mean_neighbors  Average number of neighbors.

min_neighbors  Minimum number of neighbors.

n  Number of units.

n_components  Store whether the adjacency matrix is fully connected.

neighbor_offsets  Given the current id_order, neighbor_offsets[id] is the offsets of the
                   id’s neighbors in id_order.

nonzero  Number of nonzero weights.

pct_nonzero  Percentage of nonzero weights.

s0  s0 is defined as

s1  s1 is defined as

s2  s2 is defined as

s2array  Individual elements comprising s2.

sd  Standard deviation of number of neighbors.

sparse  Sparse matrix object.

transform  Getter for transform property.

trcW2  Trace of WW.

trcWtW  Trace of W′W.

trcWtW_WW  Trace of W′W + WW.

Methods

asymmetry([intrinsic])  Asymmetry check.

from_adjlist(adjlist[, focal_col, …])  Return an adjacency list representation of a weights
                                               object.

from_array(array, *args, **kwargs)  Creates nearest neighbor weights matrix based on k
                                               nearest neighbors.

from_dataframe(df[, geom_col, ids])  Make KNN weights from a dataframe.

from_networkx(graph[, weight_col])  Convert a networkx graph to a PySAL W object.

from_shapefile(filepath, *args, **kwargs)  Nearest neighbor weights from a shapefile.

full()  Generate a full numpy array.

get_transform()  Getter for transform property.

plot(gdf[, indexed_on, ax, color, node_kws, …])  Plot spatial weights objects.

remap_ids(new_ids)  In place modification throughout W of id values from
                   w.id_order to new_ids in all

reweight([k, p, new_data, new_ids, inplace])  Redo K-Nearest Neighbor weights construction using
                                             given parameters

set_shapefile(shapefile[, idVariable, full])  Adding meta data for writing headers of gal and gwt
                                               files.

Continued on next page
Table 12 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_transform([value])</td>
<td>Transformations of weights.</td>
</tr>
<tr>
<td>symmetrize([inplace])</td>
<td>Construct a symmetric KNN weight.</td>
</tr>
<tr>
<td>to_WSP()</td>
<td>Generate a WSP object.</td>
</tr>
<tr>
<td>to_adjlist([remove_symmetric, focal_col, ...])</td>
<td>Compute an adjacency list representation of a weights object.</td>
</tr>
<tr>
<td>to_networkx()</td>
<td>Convert a weights object to a networkx graph</td>
</tr>
</tbody>
</table>

```
__init__(data, k=2, p=2, ids=None, radius=None, distance_metric='euclidean', **kwargs)
```

Initialize self. See help(type(self)) for accurate signature.

Methods

```
__init__(data[, k, p, ids, radius, ...]) Initialize self.
asymmetry([intrinsic]) Asymmetry check.
from_WSP(WSP[, silence_warnings])
from_adjlist(adjlist[, focal_col, ...]) Return an adjacency list representation of a weights object.
from_array(array, *args, **kwargs) Creates nearest neighbor weights matrix based on k nearest neighbors.
from_dataframe(df[, geom_col, ids]) Make KNN weights from a dataframe.
from_file([path, format])
from_networkx(graph[, weight_col]) Convert a networkx graph to a PySAL W object.
from_shapefile(filepath, *args, **kwargs) Nearest neighbor weights from a shapefile.
full() Generate a full numpy array.
get_transform() Getter for transform property.
plot(gdf[, indexed_on, ax, color, node_kws, ...]) Plot spatial weights objects.
remap_ids(new_ids) In place modification throughout W of id values from w.id_order to new_ids in all
reweight([k, p, new_data, new_ids, inplace]) Redo K-Nearest Neighbor weights construction using given parameters
set_shapefile(shapefile[, idVariable, full]) Adding meta data for writing headers of gal and gwt files.
set_transform([value]) Transformations of weights.
symmetrize([inplace]) Construct a symmetric KNN weight.
to_WSP() Generate a WSP object.
to_adjlist([remove_symmetric, focal_col, ...]) Compute an adjacency list representation of a weights object.
to_networkx() Convert a weights object to a networkx graph
```

Attributes

```
asymmetries List of id pairs with asymmetric weights.
cardinalities Number of neighbors for each observation.
```

Continued on next page
Table 14 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>component_labels</td>
<td>Store the graph component in which each observation falls.</td>
</tr>
<tr>
<td>diagW2</td>
<td>Diagonal of $WW$.</td>
</tr>
<tr>
<td>diagWtW</td>
<td>Diagonal of $W'W$.</td>
</tr>
<tr>
<td>diagWtW_WW</td>
<td>Diagonal of $W'W + WW$.</td>
</tr>
<tr>
<td>histogram</td>
<td>Cardinality histogram as a dictionary where key is the id and value is the number of neighbors for that unit.</td>
</tr>
<tr>
<td>id2I</td>
<td>Dictionary where the key is an ID and the value is that ID’s index in W.id_order.</td>
</tr>
<tr>
<td>id_order</td>
<td>Returns the ids for the observations in the order in which they would be encountered if iterating over the weights.</td>
</tr>
<tr>
<td>id_order_set</td>
<td>Returns True if user has set id_order, False if not.</td>
</tr>
<tr>
<td>islands</td>
<td>List of ids without any neighbors.</td>
</tr>
<tr>
<td>max_neighbors</td>
<td>Largest number of neighbors.</td>
</tr>
<tr>
<td>mean_neighbors</td>
<td>Average number of neighbors.</td>
</tr>
<tr>
<td>min_neighbors</td>
<td>Minimum number of neighbors.</td>
</tr>
<tr>
<td>n</td>
<td>Number of units.</td>
</tr>
<tr>
<td>n_components</td>
<td>Store whether the adjacency matrix is fully connected.</td>
</tr>
<tr>
<td>neighbor_offsets</td>
<td>Given the current id_order, neighbor_offsets[id] is the offsets of the id’s neighbors in id_order.</td>
</tr>
<tr>
<td>nonzero</td>
<td>Number of nonzero weights.</td>
</tr>
<tr>
<td>pct_nonzero</td>
<td>Percentage of nonzero weights.</td>
</tr>
<tr>
<td>s0</td>
<td>s0 is defined as</td>
</tr>
<tr>
<td>s1</td>
<td>s1 is defined as</td>
</tr>
<tr>
<td>s2</td>
<td>s2 is defined as</td>
</tr>
<tr>
<td>s2array</td>
<td>Individual elements comprising s2.</td>
</tr>
<tr>
<td>sd</td>
<td>Standard deviation of number of neighbors.</td>
</tr>
<tr>
<td>sparse</td>
<td>Sparse matrix object.</td>
</tr>
<tr>
<td>transform</td>
<td>Getter for transform property.</td>
</tr>
<tr>
<td>trcW2</td>
<td>Trace of $WW$.</td>
</tr>
<tr>
<td>trcWtW</td>
<td>Trace of $W'W$.</td>
</tr>
<tr>
<td>trcWtW_WW</td>
<td>Trace of $W'W + WW$.</td>
</tr>
</tbody>
</table>

Contiguity Weights

weights.Queen(polygons, **kw) Construct a weights object from a collection of pysal polygons that share at least one vertex.

weights.Rook(polygons, **kw) Construct a weights object from a collection of pysal polygons that share at least one edge.

weights.Voronoi(points) Voronoi weights for a 2-d point set

weights.W(neighbors[, weights, id_order, ...]) Spatial weights class.

pysal.lib.weights.Queen

class pysal.lib.weights.Queen (polygons, **kw)
    Construct a weights object from a collection of pysal polygons that share at least one vertex.
Parameters

polygons [list] a collection of PySAL shapes to build weights from
ids [list] a list of names to use to build the weights
**kw [keyword arguments] optional arguments for pysal.weights.W

See also:
pysal.lib.weights.weights.W

Attributes

asymmetries List of id pairs with asymmetric weights.
cardinalities Number of neighbors for each observation.
component_labels Store the graph component in which each observation falls.
diagW2 Diagonal of WW.
diagWtW Diagonal of W′W.
diagWtW_WW Diagonal of W′W + WW.
histogram Cardinality histogram as a dictionary where key is the id and value is the number of neighbors for that unit.
id2i Dictionary where the key is an ID and the value is that ID’s index in W.id_order.
id_order Returns the ids for the observations in the order in which they would be encountered if iterating over the weights.
id_order_set Returns True if user has set id_order, False if not.
islands List of ids without any neighbors.
max_neighbors Largest number of neighbors.
mean_neighbors Average number of neighbors.
min_neighbors Minimum number of neighbors.
n Number of units.
n_components Store whether the adjacency matrix is fully connected.
neighbor_offsets Given the current id_order, neighbor_offsets[id] is the offsets of the id’s neighbors in id_order.
nonzero Number of nonzero weights.
pct_nonzero Percentage of nonzero weights.
s0 s0 is defined as
s1 s1 is defined as
s2 s2 is defined as
s2array Individual elements comprising s2.
sd Standard deviation of number of neighbors.
sparse Sparse matrix object.
transform Getter for transform property.
trcW2 Trace of WW.
Methods

- `asymmetry([intrinsic])`  
  Asymmetry check.
- `from_adjlist(adjlist[, focal_col, ...])`  
  Return an adjacency list representation of a weights object.
- `from_dataframe(df[, geom_col])`  
  Construct a weights object from a pandas dataframe with a geometry column.
- `from_iterable(iterable[, sparse])`  
  Construct a weights object from a collection of arbitrary polygons.
- `from_networkx(graph[, weight_col])`  
  Convert a networkx graph to a PySAL W object.
- `from_shapefile(filepath[, idVariable, full])`  
  Queen contiguity weights from a polygon shapefile.
- `full()`  
  Generate a full numpy array.
- `get_transform()`  
  Getter for transform property.
- `plot(gdf[, indexed_on, ax, color, node_kws, ...])`  
  Plot spatial weights objects.
- `remap_ids(new_ids)`  
  In place modification throughout W of id values from `w.id_order` to `new_ids` in all.
- `set_shapefile(shapefile[, idVariable, full])`  
  Adding meta data for writing headers of gal and gwt files.
- `set_transform([value])`  
  Transformations of weights.
- `symmetrize([inplace])`  
  Construct a symmetric KNN weight.
- `to_WSP()`  
  Generate a WSP object.
- `to_adjlist([remove_symmetric, focal_col, ...])`  
  Compute an adjacency list representation of a weights object.
- `to_networkx()`  
  Convert a weights object to a networkx graph.

```
from_WSP
from_file
```

**init**(polygons, **kw)

Initialize self. See help(type(self)) for accurate signature.

Methods

```
__init__(polygons, **kw)
  Initialize self.

asymmetry([intrinsic])
  Asymmetry check.

from_WSP(WSP[, silence_warnings])
  Return an adjacency list representation of a weights object.

from_adjlist(adjlist[, focal_col, ...])
  Construct a weights object from a pandas dataframe with a geometry column.

from_dataframe(df[, geom_col])
  Construct a weights object from a collection of arbitrary polygons.

from_networkx(graph[, weight_col])
  Convert a networkx graph to a PySAL W object.
```

Continued on next page
Table 17 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from_shapefile(filepath[, idVariable, full])</td>
<td>Queen contiguity weights from a polygon shapefile.</td>
</tr>
<tr>
<td>full()</td>
<td>Generate a full numpy array.</td>
</tr>
<tr>
<td>get_transform()</td>
<td>Getter for transform property.</td>
</tr>
<tr>
<td>plot(gdf[, indexed_on, ax, color, node_kws, ...])</td>
<td>Plot spatial weights objects.</td>
</tr>
<tr>
<td>remap_ids(new_ids)</td>
<td>In place modification throughout $W$ of id values from $w.id_order$ to new_ids in all</td>
</tr>
<tr>
<td>set_shapefile(shapefile[, idVariable, full])</td>
<td>Adding meta data for writing headers of gal and gwt files.</td>
</tr>
<tr>
<td>set_transform([value])</td>
<td>Transformations of weights.</td>
</tr>
<tr>
<td>symmetrize(inplace)</td>
<td>Construct a symmetric KNN weight.</td>
</tr>
<tr>
<td>to_WSP()</td>
<td>Generate a WSP object.</td>
</tr>
<tr>
<td>to_adjlist([remove_symmetric, focal_col, ...])</td>
<td>Compute an adjacency list representation of a weights object.</td>
</tr>
<tr>
<td>to_networkx()</td>
<td>Convert a weights object to a networkx graph</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asymmetries</td>
<td>List of id pairs with asymmetric weights.</td>
</tr>
<tr>
<td>cardinalities</td>
<td>Number of neighbors for each observation.</td>
</tr>
<tr>
<td>component_labels</td>
<td>Store the graph component in which each observation falls.</td>
</tr>
<tr>
<td>diagW2</td>
<td>Diagonal of $WW$.</td>
</tr>
<tr>
<td>diagWTW</td>
<td>Diagonal of $WW$.</td>
</tr>
<tr>
<td>diagWTW_WW</td>
<td>Diagonal of $WW + WW$.</td>
</tr>
<tr>
<td>histogram</td>
<td>Cardinality histogram as a dictionary where key is the id and value is the number of neighbors for that unit.</td>
</tr>
<tr>
<td>id2i</td>
<td>Dictionary where the key is an ID and the value is that ID's index in $W.id_order$.</td>
</tr>
<tr>
<td>id_order</td>
<td>Returns the ids for the observations in the order in which they would be encountered if iterating over the weights.</td>
</tr>
<tr>
<td>id_order_set</td>
<td>Returns True if user has set id_order, False if not.</td>
</tr>
<tr>
<td>islands</td>
<td>List of ids without any neighbors.</td>
</tr>
<tr>
<td>max_neighbors</td>
<td>Largest number of neighbors.</td>
</tr>
<tr>
<td>mean_neighbors</td>
<td>Average number of neighbors.</td>
</tr>
<tr>
<td>min_neighbors</td>
<td>Minimum number of neighbors.</td>
</tr>
<tr>
<td>n</td>
<td>Number of units.</td>
</tr>
<tr>
<td>n_components</td>
<td>Store whether the adjacency matrix is fully connected.</td>
</tr>
<tr>
<td>neighbor_offsets</td>
<td>Given the current id_order, neighbor_offsets[id] is the offsets of the id's neighbors in id_order.</td>
</tr>
<tr>
<td>nonzero</td>
<td>Number of nonzero weights.</td>
</tr>
<tr>
<td>pct_nonzero</td>
<td>Percentage of nonzero weights.</td>
</tr>
<tr>
<td>s0</td>
<td>$s0$ is defined as</td>
</tr>
<tr>
<td>s1</td>
<td>$s1$ is defined as</td>
</tr>
<tr>
<td>s2</td>
<td>$s2$ is defined as</td>
</tr>
<tr>
<td>s2array</td>
<td>Individual elements comprising $s2$.</td>
</tr>
<tr>
<td>sd</td>
<td>Standard deviation of number of neighbors.</td>
</tr>
<tr>
<td>sparse</td>
<td>Sparse matrix object.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 18 – continued from previous page

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transform</td>
<td>Getter for transform property.</td>
</tr>
<tr>
<td>trcW2</td>
<td>Trace of $WW$.</td>
</tr>
<tr>
<td>trcWtW</td>
<td>Trace of $W'W$.</td>
</tr>
<tr>
<td>trcWtW_WW</td>
<td>Trace of $W'W + WW$.</td>
</tr>
</tbody>
</table>

**pysal.lib.weights.Rook**

```python
class pysal.lib.weights.Rook(polygons, **kw)
```

Construct a weights object from a collection of pysal polygons that share at least one edge.

**Parameters**

- **polygons** [list] a collection of PySAL shapes to build weights from
- **ids** [list] a list of names to use to build the weights
- ****kw [keyword arguments] optional arguments for `pysal.weights.W`

**See also:**

- `pysal.lib.weights.weights.W`

**Attributes**

- **asymmetries** List of id pairs with asymmetric weights.
- **cardinalities** Number of neighbors for each observation.
- **component_labels** Store the graph component in which each observation falls.
- **diagW2** Diagonal of $WW$.
- **diagWtW** Diagonal of $W'W$.
- **diagWtW_WW** Diagonal of $W'W + WW$.
- **histogram** Cardinality histogram as a dictionary where key is the id and value is the number of neighbors for that unit.
- **id2i** Dictionary where the key is an ID and the value is that ID’s index in W.id_order.
- **id_order** Returns the ids for the observations in the order in which they would be encountered if iterating over the weights.
- **id_order_set** Returns True if user has set id_order, False if not.
- **islands** List of ids without any neighbors.
- **max_neighbors** Largest number of neighbors.
- **mean_neighbors** Average number of neighbors.
- **min_neighbors** Minimum number of neighbors.
- **n** Number of units.
- **n_components** Store whether the adjacency matrix is fully connected.
- **neighbor_offsets** Given the current id_order, neighbor_offsets[id] is the offsets of the id’s neighbors in id_order.
- **nonzero** Number of nonzero weights.
- **pct_nonzero** Percentage of nonzero weights.
**Methods**

- `asymmetry([intrinsic])`: Asymmetry check.
- `from_adjlist(adjlist[, focal_col, ...])`: Return an adjacency list representation of a weights object.
- `from_dataframe(df[, geom_col, idVariable, ...])`: Construct a weights object from a pandas dataframe with a geometry column.
- `from_iterable(iterable[, sparse])`: Construct a weights object from a collection of arbitrary polygons.
- `from_networkx(graph[, weight_col])`: Convert a networkx graph to a PySAL W object.
- `from_shapefile(filepath[, idVariable, full])`: Rook contiguity weights from a polygon shapefile.
- `full()`: Generate a full numpy array.
- `get_transform()`: Getter for transform property.
- `plot(gdf[, indexed_on, ax, color, node_kws, ...])`: Plot spatial weights objects.
- `remap_ids(new_ids)`: In place modification throughout W of id values from `w.id_order` to `new_ids` in all.
- `set_shapefile(shapefile[, idVariable, full])`: Adding meta data for writing headers of gal and gwt files.
- `set_transform([value])`: Transformations of weights.
- `symmetrize([inplace])`: Construct a symmetric KNN weight.
- `to_WSP()`: Generate a WSP object.
- `to_adjlist([remove_symmetric, focal_col, ...])`: Compute an adjacency list representation of a weights object.
- `to_networkx()`: Convert a weights object to a networkx graph

**__init__ (polygons, **kw)**

Initialize self. See help(type(self)) for accurate signature.
## __init__ (polygons, **kw)
Initialize self.

## asymmetry ([intrinsic])
Asymmetry check.

## from_WSP (WSP, silence_warnings)

## from_adjlist (adjlist[, focal_col, ...])
Return an adjacency list representation of a weights object.

## from_dataframe (df, geom_col, idVariable, ...)
Construct a weights object from a pandas dataframe with a geometry column.

## from_file ([path, format])

## from_iterable (iterable[, sparse])
Construct a weights object from a collection of arbitrary polygons.

## from_networkx (graph[, weight_col])
Convert a networkx graph to a PySAL W object.

## from_shapefile (filepath[, idVariable, full])
Rook contiguity weights from a polygon shapefile.

## full ()
Generate a full numpy array.

## get_transform ()
Getter for transform property.

## plot (gdf[, indexed_on, ax, color, node_kws, ...])
Plot spatial weights objects.

## remap_ids (new_ids)
In place modification throughout W of id values from w.id_order to new_ids in all

## set_shapefile (shapefile[, idVariable, full])
Adding meta data for writing headers of gal and gwt files.

## set_transform ([value])
Transformations of weights.

## symmetrize ([inplace])
Construct a symmetric KNN weight.

## to_WSP ()
Generate a WSP object.

## to_adjlist ([remove_symmetric, focal_col, ...])
Compute an adjacency list representation of a weights object.

## to_networkx ()
Convert a weights object to a networkx graph

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asymmetries</td>
<td>List of id pairs with asymmetric weights.</td>
</tr>
<tr>
<td>cardinalities</td>
<td>Number of neighbors for each observation.</td>
</tr>
<tr>
<td>component_labels</td>
<td>Store the graph component in which each observation falls.</td>
</tr>
<tr>
<td>diagW2</td>
<td>Diagonal of $WW$.</td>
</tr>
<tr>
<td>diagWtW</td>
<td>Diagonal of $W^T W$.</td>
</tr>
<tr>
<td>diagWtW_WW</td>
<td>Diagonal of $W^T W + WW$.</td>
</tr>
<tr>
<td>histogram</td>
<td>Cardinality histogram as a dictionary where key is the id and value is the number of neighbors for that unit.</td>
</tr>
<tr>
<td>id2i</td>
<td>Dictionary where the key is an ID and the value is that ID’s index in W.id_order.</td>
</tr>
<tr>
<td>id_order</td>
<td>Returns the ids for the observations in the order in which they would be encountered if iterating over the weights.</td>
</tr>
<tr>
<td>id_order_set</td>
<td>Returns True if user has set id_order, False if not.</td>
</tr>
<tr>
<td>islands</td>
<td>List of ids without any neighbors.</td>
</tr>
<tr>
<td>max_neighbors</td>
<td>Largest number of neighbors.</td>
</tr>
<tr>
<td>mean_neighbors</td>
<td>Average number of neighbors.</td>
</tr>
<tr>
<td>min_neighbors</td>
<td>Minimum number of neighbors.</td>
</tr>
<tr>
<td>n</td>
<td>Number of units.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 21 – continued from previous page

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_components</td>
<td>Store whether the adjacency matrix is fully connected.</td>
</tr>
<tr>
<td>neighbor_offsets</td>
<td>Given the current id_order, neighborOffsets[id] is the</td>
</tr>
<tr>
<td></td>
<td>offsets of the id’s neighbors in id_order.</td>
</tr>
<tr>
<td>nonzero</td>
<td>Number of nonzero weights.</td>
</tr>
<tr>
<td>pct_nonzero</td>
<td>Percentage of nonzero weights.</td>
</tr>
<tr>
<td>s0</td>
<td>s0 is defined as</td>
</tr>
<tr>
<td>s1</td>
<td>s1 is defined as</td>
</tr>
<tr>
<td>s2</td>
<td>s2 is defined as</td>
</tr>
<tr>
<td>s2array</td>
<td>Individual elements comprising s2.</td>
</tr>
<tr>
<td>sd</td>
<td>Standard deviation of number of neighbors.</td>
</tr>
<tr>
<td>sparse</td>
<td>Sparse matrix object.</td>
</tr>
<tr>
<td>transform</td>
<td>Getter for transform property.</td>
</tr>
<tr>
<td>trcW2</td>
<td>Trace of $W$.</td>
</tr>
<tr>
<td>trcWtW</td>
<td>Trace of $W^TW$.</td>
</tr>
<tr>
<td>trcWtW_WW</td>
<td>Trace of $W^TW + WW$.</td>
</tr>
</tbody>
</table>

pysal.lib.weights.Voronoi

pysal.lib.weights.Voronoi(points)
Voronoi weights for a 2-d point set
Points are Voronoi neighbors if their polygons share an edge or vertex.

Parameters
points [array] (n,2) coordinates for point locations

Returns
w [W] instance of spatial weights

Examples

```python
>>> import numpy as np
>>> np.random.seed(12345)
>>> points= np.random.random((5,2))*10 + 10
>>> w = Voronoi(points)
>>> w.neighbors
{0: [1, 2, 3, 4], 1: [0, 2], 2: [0, 1, 4], 3: [0, 4], 4: [0, 2, 3]}
```

spint Weights

.weights.WSP(sparse[, id_order]) Thin W class for spreg.
.weights.netW(link_list[, share, transform]) Create a network-contiguity based weight object based on different nodal relationships encoded in a network.
.weights.mat2L(edge_matrix) Convert a matrix denoting network connectivity (edges or flows) to a list denoting edges
.weights.ODW(Wo, Wd[, transform, ...]) Constructs an o*d by o*d origin-destination style spatial weight for o*d flows using standard spatial weights on o origins and d destinations.

Continued on next page
### weights.vecW

Distance-based spatial weight for vectors that is computed using a 4-dimensional distance between the origin x,y-coordinates and the destination x,y-coordinates

---

### pysal.lib.weights.WSP

**class** `pysal.lib.weights.WSP` *(sparse, id_order=None)*

Thin W class for spreg.

**Parameters**

- `sparse` [sparse_matrix] NxN object from scipy.sparse
- `id_order` [list] An ordered list of ids, assumed to match the ordering in sparse.

**Examples**

From GAL information

```python
>>> import scipy.sparse
>>> from pysal.lib.weights import WSP
>>> rows = [0, 1, 1, 2, 2, 3]
>>> cols = [1, 0, 2, 1, 3, 3]
>>> weights = [1, 0.75, 0.25, 0.9, 0.1, 1]
>>> sparse = scipy.sparse.csr_matrix((weights, (rows, cols)), shape=(4,4))
>>> w = WSP(sparse)
>>> w.s0
4.0
>>> w.traceWtW_WW
6.395
>>> w.n
4
```

**Attributes**

- `n` [int] description
- `s0` [float] s0 is defined as:
- `traceWtW_WW` [float] Trace of $W'W + WW$.

**Methods**

- `from_W(W)` Constructs a WSP object from the W’s sparse matrix
- `to_W([silence_warnings])` Convert a pysal WSP object (thin weights matrix) to a pysal W object.
- `__init__(sparse, id_order=None)` Initialize self. See help(type(self)) for accurate signature.
__init__ (sparse[, id_order]) 
Initialize self.

from_W(W) 
Constructs a WSP object from the W’s sparse matrix

to_W([silence_warnings]) 
Convert a pysal WSP object (thin weights matrix) to a pysal W object.

Attributes

diagWtW_WW 
Diagonal of \( W W^T + W W \).
s0 
s0 is defined as:

trcWtW_WW 
Trace of \( W W^T + W W \).

pysal.lib.weights.netW

pysal.lib.weights.netW(link_list, share='A', transform='r')
Create a network-contiguity based weight object based on different nodal relationships encoded in a network.

Parameters

link_list [list] of tuples where each tuple is of the form (o,d) where o is an origin id and d is a destination id

share [string] denoting how to define the nodal relationship used to determine neighboring edges; default is ‘A’ for any shared nodes between two network edges; options include:
‘A’: any shared nodes ‘O’: a shared origin node ‘D’: a shared destination node ‘OD’ a shared origin node or a shared destination node ‘C’: a shared node that is the destination of the first edge and the origin of the second edge - i.e., a directed chain is formed moving from edge one to edge two.

transform [Transformation for standardization of final OD spatial weight; default] is ‘r’ for row standardized

Returns

W [nodal contiguity W object for networkd edges or flows] W Object representing the binary adjacency of the network edges given a definition of nodal relationshipspysal.lib.weights.spintW.

Examples

```python
>>> import pysal.lib
>>> links = [('a', 'b'), ('a', 'c'), ('a', 'd'), ('c', 'd'), ('c', 'b'), ('c', 'a')]
>>> O = pysal.lib.weights.spintW.netW(links, share='O')
>>> O.neighbors[('a', 'b')]
[('a', 'c'), ('a', 'd')]
>>> OD = pysal.lib.weights.spintW.netW(links, share='OD')
>>> OD.neighbors[('a', 'b')]
[('a', 'c'), ('a', 'd'), ('c', 'b')]
>>> any_common = pysal.lib.weights.spintW.netW(links, share='A')
>>> any_common.neighbors[('a', 'b')]
[('a', 'c'), ('a', 'd'), ('c', 'b'), ('c', 'a')]
```
pysal.lib.weights.mat2L

Convert a matrix denoting network connectivity (edges or flows) to a list denoting edges

**Parameters**

- **edge_matrix** [array] where rows denote network edge origins, columns denote network edge destinations, and non-zero entries denote the existence of an edge between a given origin and destination

**Returns**

- **edge_list** [list] of tuples where each tuple is of the form (o,d) where o is an origin id and d is a destination id

pysal.lib.weights.ODW

Constructs an o*d by o*d origin-destination style spatial weight for o*d flows using standard spatial weights on o origins and d destinations. Input spatial weights must be binary or able to be suitably transformed to binary.

**Parameters**

- **Wo** [W object for origin locations] o x o spatial weight object amongst o origins
- **Wd** [W object for destination locations] d x d spatial weight object amongst d destinations
- **transform** [Transformation for standardization of final OD spatial weight; default] is 'r' for row standardized

**Returns**

- **W** [spatial contiguity W object for associations between flows] o*d x o*d spatial weight object amongst o*d flows between o origins and d destinations

**Examples**

```python
>>> import pysal.lib
>>> O = pysal.lib.weights.lat2W(2, 2)
>>> D = pysal.lib.weights.lat2W(2, 2)
>>> OD = pysal.lib.weights.spintW.ODW(O, D)
>>> OD.weights[0]
array([0.25, 0.25, 0.25, 0.25])
>>> OD.neighbors[0]
array([ 5, 6, 9, 10], dtype=int32)
>>> OD.full()[0]
array([[ 0., 0., 0., 0., 0., 0., 0.25, 0.25, 0., 0., 0.25, 0.25, 0.25, 0., 0., 0., 0. ]])
```

pysal.lib.weights.vecW

Distance-based spatial weight for vectors that is computed using a 4-dimensional distance between the origin x,y-coordinates and the destination x,y-coordinates
Parameters

- **origin_x** [list or array] of vector origin x-coordinates
- **origin_y** [list or array] of vector origin y-coordinates
- **dest_x** [list or array] of vector destination x-coordinates
- **dest_y** [list or array] of vector destination y-coordinates
- **threshold** [float] distance band
- **p** [float] Minkowski p-norm distance metric parameter: 1<=p<=infinity
  - 2: Euclidean distance
  - 1: Manhattan distance
- **binary** [boolean] If true w_{ij}=1 if d_{i,j}<=threshold, otherwise w_{i,j}=0 If false wij=dij^{alpha}
- **alpha** [float] distance decay parameter for weight (default -1.0) if alpha is positive the weights will not decline with distance. If binary is True, alpha is ignored
- **ids** [list] values to use for keys of the neighbors and weights dicts
- **build_sp** [boolean] True to build sparse distance matrix and false to build dense distance matrix; significant speed gains may be obtained depending on the sparsity of the distance_matrix and threshold that is applied
- **silence_warnings** [boolean] By default PySAL will print a warning if the dataset contains any disconnected observations or islands. To silence this warning set this parameter to True.

Returns

- **W** [DistanceBand W object that uses 4-dimensional distances between] vectors origin and destination coordinates.

Examples

```python
>>> import pysal.lib
>>> x1 = [5, 6, 3]
>>> y1 = [1, 8, 5]
>>> x2 = [2, 4, 9]
>>> y2 = [3, 6, 1]
>>> W1 = pysal.lib.weights.spintW.vecW(x1, y1, x2, y2, threshold=999)
>>> list(W1.neighbors[0])
[1, 2]
>>> W2 = pysal.lib.weights.spintW.vecW(x1, y2, x1, y2, threshold=8.5)
>>> list(W2.neighbors[0])
[1, 2]
```

Weights Util Classes and Functions

- **weights.block_weights**(regimes[, ids, sparse]) Construct spatial weights for regime neighbors.
- **weights.lat2W**([nrows, ncols, rook, id_type]) Create a W object for a regular lattice.
- **weights.comb**(items[, n]) Combinations of size n taken from items

Continued on next page
weights.order(w[, kmax])
Determine the non-redundant order of contiguity up to a specific order.

weights.higher_order(w[, k])
Contiguity weights object of order k.

weights.shimbel(w)
Find the Shimbel matrix for first order contiguity matrix.

weights.remap_ids(w, old2new[, id_order])
Remaps the IDs in a spatial weights object.

weights.full2W(m[, ids])
Create a PySAL W object from a full array.

weights.full(w)
Generate a full numpy array.

weights.WSP2W(wsp[, silence_warnings])
Convert a pysal WSP object (thin weights matrix) to a pysal W object.

weights.get_ids(in_shps, idVariable)
Gets the IDs from the DBF file that moves with a given shape file or a geopandas.GeoDataFrame.

weights.get_points_array_from_shapefile()
Gets a data array of x and y coordinates from a given shapefile.

pysal.lib.weights.block_weights

pysal.lib.weights.block_weights(regimes, ids=None, sparse=False)
Construct spatial weights for regime neighbors.

Block contiguity structures are relevant when defining neighbor relations based on membership in a regime. For example, all counties belonging to the same state could be defined as neighbors, in an analysis of all counties in the US.

Parameters

regimes [list, array] ids of which regime an observation belongs to
ids [list, array] Ordered sequence of IDs for the observations
sparse [boolean] If True return WSP instance If False return W instance

Returns

W [spatial weights instance]

Examples

```python
>>> from pysal.lib.weights import lat2W
>>> import numpy as np
>>> regimes = np.ones(25)
>>> regimes[range(10,20)] = 2
>>> regimes[range(21,25)] = 3
>>> regimes
array([1., 1., 1., 1., 1., 1., 1., 2., 2., 2., 2., 2., 2., 2.,
       2., 2., 2., 1., 1., 3., 3., 3., 3.])
>>> w = block_weights(regimes)
>>> w.weights[0]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> w.neighbors[0]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 20]
>>> regimes = ['n','n','s','s','e','e','w','w','e']
>>> n = len(regimes)
>>> w = block_weights(regimes)
```
>>> w.neighbors == {0: [1], 1: [0], 2: [3], 3: [2], 4: [5, 8], 5: [4, 8], 6: [7], 7: [6], 8: [4, 5]}
True

**pysal.lib.weights.lat2W**

**pysal.lib.weights.lat2W** *(nrows=5, ncols=5, rook=True, id_type='int')*

Create a W object for a regular lattice.

**Parameters**

- **nrows** [int] number of rows
- **ncols** [int] number of columns
- **rook** [boolean] type of contiguity. Default is rook. For queen, rook =False
- **id_type** [string] string defining the type of IDs to use in the final W object; options are ‘int’ (0, 1, 2; default), ‘float’ (0.0, 1.0, 2.0, etc.) and ‘string’ (‘id0’, ‘id1’, ‘id2’, etc.)

**Returns**

- **w** [W] instance of spatial weights class W

**Notes**

Observations are row ordered: first k observations are in row 0, next k in row 1, and so on.

**Examples**

```python
>>> from pysal.lib.weights import lat2W
>>> w9 = lat2W(3,3)
>>> "%s"%w9.pct_nonzero
'29.630'
>>> w9[0] == {1: 1.0, 3: 1.0}
True
>>> w9[3] == {0: 1.0, 4: 1.0, 6: 1.0}
True
```

**pysal.lib.weights.comb**

**pysal.lib.weights.comb** *(items, n=None)*

Combinations of size n taken from items

**Parameters**

- **items** [list] items to be drawn from
- **n** [integer] size of combinations to take from items

**Returns**

- **implicit** [generator] combinations of size n taken from items
Examples

```python
>>> x = range(4)
>>> for c in comb(x, 2):
...     print(c)
...
[0, 1]
[0, 2]
[0, 3]
[1, 2]
[1, 3]
[2, 3]
```

pysal.lib.weights.order

```python
def order(w, kmax=3):
    """Determine the non-redundant order of contiguity up to a specific order."
    Parameters
    w [W] spatial weights object
    kmax [int] maximum order of contiguity
    Returns
    info [dictionary] observation id is the key, value is a list of contiguity orders with a negative 1 in the ith position
    
    Notes
    Implements the algorithm in Anselin and Smirnov (1996) [Anselin1996b]
    
    Examples
    ```

```python
>>> from pysal.lib.weights import lat2W
>>> from pysal.lib.weights.contiguity import Rook
>>> import pysal.lib

>>> w = Rook.from_shapefile(pysal.lib.examples.get_path('10740.shp'))
WARNING: there is one disconnected observation (no neighbors) Island id: [163] >>> w3 = order(w, kmax = 3) >>> w3[1][0:5] [1, -1, 1, 2, 1]
```

pysal.lib.weights.higher_order

```python
def higher_order(w, k=2):
    """Contiguity weights object of order k."
    Parameters
    w [W] spatial weights object
    k [int] order of contiguity
    Returns
    ```
**Notes**

Proper higher order neighbors are returned such that i and j are k-order neighbors iff the shortest path from i-j is of length k.

**Examples**

```python
>>> from pysal.lib.weights import lat2W
>>> w10 = lat2W(10, 10)
>>> w10_2 = higher_order(w10, 2)
>>> w10_2[0] == {2: 1.0, 11: 1.0, 20: 1.0}
True
>>> w5 = lat2W()
>>> w5[0] == {1: 1.0, 5: 1.0}
True
>>> w5[1] == {0: 1.0, 2: 1.0, 6: 1.0}
True
>>> w5_2 = higher_order(w5, 2)
>>> w5_2[0] == {10: 1.0, 2: 1.0, 6: 1.0}
True
```

**pysal.lib.weights.shimbel**

Find the Shimbel matrix for first order contiguity matrix.

**Parameters**

- `w` [W] spatial weights object

**Returns**

- `info` list list of lists; one list for each observation which stores the shortest order between it and each of the the other observations.

**Examples**

```python
>>> from pysal.lib.weights import lat2W
>>> w5 = lat2W()
>>> w5_shimbel = shimbel(w5)
>>> w5_shimbel[0][24] 8
>>> w5_shimbel[0][0:4]
[-1, 1, 2, 3]
```

**pysal.lib.weights.remap_ids**

Remaps the IDs in a spatial weights object.
Parameters

- **w** [W] Spatial weights object
- **old2new** [dictionary] Dictionary where the keys are the IDs in w (i.e. “old IDs”) and the values are the IDs to replace them (i.e. “new IDs”)
- **id_order** [list] An ordered list of new IDs, which defines the order of observations when iterating over W. If not set then the id_order in w will be used.

Returns

- **implicit** [W] Spatial weights object with new IDs

Examples

```python
>>> from pysal.lib.weights import lat2W
>>> w = lat2W(3, 2)
>>> w.id_order
[0, 1, 2, 3, 4, 5]
>>> w.neighbors[0]
[2, 1]
>>> old_to_new = {0:'a', 1:'b', 2:'c', 3:'d', 4:'e', 5:'f'}
>>> w_new = remap_ids(w, old_to_new)
>>> w_new.id_order
['a', 'b', 'c', 'd', 'e', 'f']
>>> w_new.neighbors['a']
['c', 'b']
```

**pysal.lib.weights.full2W**

Create a PySAL W object from a full array.

Parameters

- **m** [array] nxn array with the full weights matrix
- **ids** [list] User ids assumed to be aligned with m

Returns

- **w** [W] PySAL weights object

Examples

```python
>>> import pysal.lib
>>> import numpy as np
Create an array of zeros

>>> a = np.zeros((4, 4))
For loop to fill it with random numbers
```
```python
>>> for i in range(len(a)):
...   for j in range(len(a[i])):
...     if i!=j:
...       a[i, j] = np.random.random(1)

Create W object

``` pysal.lib.weights.util.full2W(a)
```python
>>> w = pysal.lib.weights.util.full2W(a)
>>> w.full()[0] == a
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])

Create list of user ids

```python
>>> ids = ['myID0', 'myID1', 'myID2', 'myID3']
>>> w = pysal.lib.weights.util.full2W(a, ids=ids)
>>> w.full()[0] == a
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

```python
pysal.lib.weights.full

pysal.lib.weights.full(w)

Generate a full numpy array.

Parameters

w [W] spatial weights object

Returns

(fullw, keys) [tuple] first element being the full numpy array and second element keys being the ids associated with each row in the array.

Examples

```python
>>> from pysal.lib.weights import lat2W, W
>>> neighbors = {'first':['second'], 'second':['first', 'third'], 'third':['second']}
>>> weights = {'first':[1], 'second':[1,1], 'third':[1]}
>>> w = W(neighbors, weights)
>>> wf, ids = full(w)
>>> wf
array([[0., 1., 0.],
       [1., 0., 1.],
       [0., 1., 0.]])
```

```python
>>> ids
['first', 'second', 'third']
```
pysal.lib.weights.WSP2W

pysal.lib.weights.WSP2W(wsp, silence_warnings=False)
Convert a pysal WSP object (thin weights matrix) to a pysal W object.

Parameters

- **wsp** [WSP] PySAL sparse weights object
- **silence_warnings** [boolean] Switch to turn off (default on) print statements for every observation with islands

Returns

- **w** [W] PySAL weights object

Examples

```python
>>> from pysal.lib.weights import lat2W, WSP
Build a 10x10 scipy.sparse matrix for a rectangular 2x5 region of cells (rook contiguity), then construct a PySAL sparse weights object (wsp).

```sp = lat2SW(2, 5)
```wsp = WSP(sp)
```wsp.n
10
```w = WSP2W(wsp)
```w.n
10
```w.full()[0][0]
[0. 1. 0. 0. 0. 1. 0. 0. 0. 0.]
```

pysal.lib.weights.get_ids

pysal.lib.weights.get_ids(in_shps, idVariable)
Gets the IDs from the DBF file that moves with a given shape file or a geopandas.GeoDataFrame.

Parameters

- **in_shps** [str or geopandas.GeoDataFrame] The input geographic data. Either (1) a path to a shapefile including suffix (str); or (2) a geopandas.GeoDataFrame.
- **idVariable** [str] name of a column in the shapefile’s DBF or the geopandas.GeoDataFrame to use for ids.

Returns

- **ids** [list] a list of IDs
Examples

```python
>>> from pysal.lib.weights.util import get_ids
>>> import pysal.lib
>>> polyids = get_ids(pysal.lib.examples.get_path("columbus.shp"), "POLYID")
>>> polyids[:5]
[1, 2, 3, 4, 5]
```

```python
>>> from pysal.lib.weights.util import get_ids
>>> import pysal.lib
>>> import geopandas as gpd
>>> gdf = gpd.read_file(pysal.lib.examples.get_path("columbus.shp"))
>>> polyids = gdf["POLYID"]
>>> polyids[:5]
0    1
1    2
2    3
3    4
4    5
Name: POLYID, dtype: int64
```

**pysal.lib.weights.get_points_array_from_shapefile**

*pysal.lib.weights.get_points_array_from_shapefile*(shapefile)

Gets a data array of x and y coordinates from a given shapefile.

**Parameters**

- shapefile [string] name of a shape file including suffix

**Returns**

- points [array] (n, 2) a data array of x and y coordinates

**Notes**

If the given shape file includes polygons, this function returns x and y coordinates of the polygons’ centroids

**Examples**

Point shapefile

```python
>>> import pysal.lib
>>> from pysal.lib.weights.util import get_points_array_from_shapefile
>>> xy = get_points_array_from_shapefile(pysal.lib.examples.get_path('juvenile.shp'))
>>> xy[:3]
array([[94., 93.],
       [80., 95.],
       [79., 90.]]),
```

Polygon shapefile
```python
>>> xy = get_points_array_from_shapefile(pysal.lib.examples.get_path('columbus.shp'))
>>> xy[:3]
array([[ 8.82721847, 14.36907602],
       [ 8.33265837, 14.03162401],
       [ 9.01226541, 13.81971908]])
```

Weights user Classes and Functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>weights.min_threshold_distance(data[,p])</code></td>
<td>Get the maximum nearest neighbor distance.</td>
</tr>
<tr>
<td><code>weights.lat2SW([nrows, ncols, criterion, row_st])</code></td>
<td>Create a sparse W matrix for a regular lattice.</td>
</tr>
<tr>
<td><code>weights.w_local_cluster(w)</code></td>
<td>Local clustering coefficients for each unit as a node in a graph.</td>
</tr>
<tr>
<td><code>weights.higher_order_sp(w[,k,...])</code></td>
<td>Contiguity weights for either a sparse W or W for order k.</td>
</tr>
<tr>
<td><code>weights.hexLat2W([nrows, ncols])</code></td>
<td>Create a W object for a hexagonal lattice.</td>
</tr>
<tr>
<td><code>weights.attach_islands(w, w_knn1)</code></td>
<td>Attach nearest neighbor to islands in spatial weight w.</td>
</tr>
<tr>
<td><code>weights.nonplanar_neighbors(w, geo-dataframe)</code></td>
<td>Detect neighbors for non-planar polygon collections</td>
</tr>
<tr>
<td><code>weights.fuzzy_contiguity(gdf, tolerance,...)</code></td>
<td>Fuzzy contiguity spatial weights</td>
</tr>
<tr>
<td><code>weights.min_threshold_dist_from_shapefile</code></td>
<td>Kernel weights with adaptive bandwidths.</td>
</tr>
<tr>
<td><code>weights.build_lattice_shapefile(nrows,...)</code></td>
<td>Build a lattice shapefile with nrows rows and ncols cols.</td>
</tr>
<tr>
<td><code>weights.spw_from_gal(galfile)</code></td>
<td>Sparse scipy matrix for w from a gal file.</td>
</tr>
</tbody>
</table>

`pysal.lib.weights.min_threshold_distance`

Get the maximum nearest neighbor distance.

**Parameters**

- `data` [array] (n,k) or KDTree where KDtree.data is array (n,k) n observations on k attributes

**Returns**

- `nnd` [float] maximum nearest neighbor distance between the n observations

**Examples**

```python
>>> from pysal.lib.weights.util import min_threshold_distance
>>> import numpy as np
>>> x, y = np.indices((5, 5))
>>> x.shape = (25, 1)
>>> y.shape = (25, 1)
>>> data = np.hstack([x, y])
>>> min_threshold_distance(data)
1.0
```
pysal.lib.weights.lat2SW

pysal.lib.weights.lat2SW(nrows=3, ncols=5, criterion='rook', row_st=False)
Create a sparse W matrix for a regular lattice.

Parameters

nrows [int] number of rows
ncols [int] number of columns
rook [{“rook”, “queen”, “bishop”}] type of contiguity. Default is rook.
row_st [boolean] If True, the created sparse W object is row-standardized so every row sums up to one. Defaults to False.

Returns

w [scipy.sparse.dia_matrix] instance of a scipy sparse matrix

Notes

Observations are row ordered: first k observations are in row 0, next k in row 1, and so on. This method directly creates the W matrix using the structure of the contiguity type.

Examples

```python
>>> from pysal.lib.weights import lat2W
>>> w9 = lat2SW(3,3)
>>> w9[0,1]
1
>>> w9[3,6]
1
>>> w9r = lat2SW(3,3, row_st=True)
>>> w9r[3,6] == 1./3
True
```

pysal.lib.weights.w_local_cluster

pysal.lib.weights.w_local_cluster(w)
Local clustering coefficients for each unit as a node in a graph. [ws]

Parameters

w [W] spatial weights object

Returns

c [array] (w.n,1) local clustering coefficients

Notes

The local clustering coefficient $c_i$ quantifies how close the neighbors of observation $i$ are to being a clique:

$$c_i = |\{w_{j,k}\}|/(k_i(k_i - 1)) : j, k \in N_i$$
where $N_i$ is the set of neighbors to $i$, $k_i = |N_i|$ and $\{w_{j,k}\}$ is the set of non-zero elements of the weights between pairs in $N_i$. [Watts1998]

Examples

```python
>>> from pysal.lib.weights import lat2W
>>> w = lat2W(3,3, rook=False)
>>> w_local_cluster(w)
array([[1.],
       [0.6],
       [1.],
       [0.6],
       [0.42857143],
       [0.6],
       [1.],
       [0.6],
       [1.],])
```

True

**pysal.lib.weights.higher_order_sp**

pysal.lib.weights.higher_order_sp(w, k=2, shortest_path=True, diagonal=False)

Contiguity weights for either a sparse W or W for order k.

Parameters

- **w** [W] sparse_matrix, spatial weights object or scipy.sparse.csr.csr_instance
- **k** [int] Order of contiguity
- **shortest_path** [boolean] True: i,j and k-order neighbors if the shortest path for i,j is k False: i,j are k-order neighbors if there is a path from i,j of length k
- **diagonal** [boolean] True: keep k-order (i,j) joins when i==j False: remove k-order (i,j) joins when i==j

Returns

- **wk** [W] WSP, type matches type of w argument

Notes

Lower order contiguities are removed.

Examples

```python
>>> from pysal.lib.weights import lat2W
>>> import pysal.lib
>>> w25 = lat2W(5,5)
>>> w25.n
25
>>> w25[0] == {1: 1.0, 5: 1.0}
True
>>> w25_2 = pysal.lib.weights.util.higher_order_sp(w25, 2)
(continues on next page)
```
>>> w25_2[0] == {10: 1.0, 2: 1.0, 6: 1.0}
True
>>> w25_2 = pysal.lib.weights.util.higher_order_sp(w25, 2, diagonal=True)
>>> w25_2[0] == {0: 1.0, 10: 1.0, 2: 1.0, 6: 1.0}
True
>>> w25_3 = pysal.lib.weights.util.higher_order_sp(w25, 3)
>>> w25_3[0] == {15: 1.0, 3: 1.0, 11: 1.0, 7: 1.0}
True
>>> w25_3 = pysal.lib.weights.util.higher_order_sp(w25, 3, shortest_path=False)
>>> w25_3[0] == {1: 1.0, 3: 1.0, 5: 1.0, 7: 1.0, 11: 1.0, 15: 1.0}
True

pysal.lib.weights.hexLat2W

Create a W object for a hexagonal lattice.

Parameters

- nrows [int] number of rows
- ncols [int] number of columns

Returns

- w [W] instance of spatial weights class W

Notes

Observations are row ordered: first k observations are in row 0, next k in row 1, and so on. Construction is based on shifting every other column of a regular lattice down 1/2 of a cell.

Examples

```python
>>> from pysal.lib.weights import lat2W
data = lat2W()
data.neighbors[1]
[0, 6, 2]
data.neighbors[21]
[16, 20, 22]
data.hexLat2W()
data.neighbors[1]
[0, 6, 2, 5, 7]
data.neighbors[21]
[16, 20, 22]
```

pysal.lib.weights.attach_islands

Attach nearest neighbor to islands in spatial weight w.
Parameters

- \(w\) [pysal.lib.weights.W] pysal spatial weight object (unstandardized).
- \(w_{knn1}\) [pysal.lib.weights.W] Nearest neighbor pysal spatial weight object (k=1).

Returns

- : pysal.lib.weights.W pysal spatial weight object \(w\) without islands.

Examples

```python
>>> from pysal.lib.weights import lat2W
>>> import pysal

>>> w = pysal.lib.weights.contiguity.Rook.from_shapefile(pysal.lib.examples.get_path('10740.shp'))

>>> w.islands
[163]

>>> w_knn1 = pysal.lib.weights.distance.KNN.from_shapefile(pysal.lib.examples.get_path('10740.shp'), k=1)

>>> w_attach = attach_islands(w, w_knn1)

>>> w_attach.islands
[]

>>> wAttach[w.islands[0]]
{166: 1.0}
```

`pysal.lib.weights.nonplanar_neighbors`

`pysal.lib.weights.nonplanar_neighbors` \((w, \text{geodataframe}, \text{tolerance}=0.001)\)

Detect neighbors for non-planar polygon collections

Parameters

- \(w\): pysal W A spatial weights object with reported islands
- \(\text{geodataframe}\): GeoDataFrame The polygon dataframe from which \(w\) was constructed.
- \(\text{tolerance}\): float The percentage of the minimum horizontal or vertical extent (minextent) of the dataframe to use in defining a buffering distance to allow for fuzzy contiguity detection. The buffering distance is equal to tolerance*minextent.

Returns

- \(w\): pysal W Spatial weights object that encodes fuzzy neighbors. This will have an attribute `non_planar_joins` to indicate what new joins were detected.

Notes

This relaxes the notion of contiguity neighbors for the case of shapefiles that violate the condition of planar enforcement. It handles three types of conditions present in such files that would result in islands when using the regular PySAL contiguity methods. The first are edges for nearby polygons that should be shared, but are digitized separately for the individual polygons and the resulting edges do not coincide, but instead the edges intersect. The second case is similar to the first, only the resultant edges do not intersect but are “close”. The final case arises when one polygon is “inside” a second polygon but is not encoded to represent a hole in the containing polygon.

The buffering check assumes the geometry coordinates are projected.
References

Planar Enforcement: http://ibis.geog.ubc.ca/courses/klink/gis.notes/ncgia/u12.html#SEC12.6

Examples

```python
>>> import geopandas as gpd
>>> import pysal.lib

>>> df = gpd.read_file(pysal.lib.examples.get_path('map_RS_BR.shp'))

>>> w = pysal.lib.weights.contiguity.Queen.from_dataframe(df)

>>> import pysal.lib

>>> w.islands
[0, 4, 23, 27, 80, 94, 101, 107, 109, 119, 122, 139, 169, 175, 223, 239, 247, 253, 254, 255, 256, 261, 276, 291, 294, 303, 321, 357, 374]

>>> wnp = pysal.lib.weights.util.nonplanar_neighbors(w, df)

>>> wnp.islands
[]

>>> wnp.neighbors[0]
[]

>>> wnp.neighbors[0]

[23, 59, 152, 239]

>>> wnp.neighbors[23]

[0, 45, 59, 107, 152, 185, 246]

Also see nonplanarweights.ipynb

Attributes

- **non_planar_joins** [dictionary] Stores the new joins detected. Key is the id of the focal unit, value is a list of neighbor ids.

**pysal.lib.weights.fuzzy_contiguity**

**pysal.lib.weights.fuzzy_contiguity**(*gdf*, *tolerance=0.005*, *buffering=False*, *drop=True*)

Fuzzy contiguity spatial weights

Parameters

- **gdf**: GeoDataFrame
- **tolerance**: float The percentage of the length of the minimum side of the bounding rectangle for the GeoDataFrame to use in determining the buffering distance.
- **buffering**: boolean If False (default) joins will only be detected for features that intersect (touch, contain, within). If True then features will be buffered and intersections will be based on buffered features.
- **drop**: boolean If True (default), the buffered features are removed from the GeoDataFrame. If False, buffered features are added to the GeoDataFrame.

Returns

- **w**: PySAL W Spatial weights based on fuzzy contiguity. Weights are binary.
Notes

This relaxes the notion of contiguity neighbors for the case of feature collections that violate the condition of planar enforcement. It handles three types of conditions present in such collections that would result in islands when using the regular PySAL contiguity methods. The first are edges for nearby polygons that should be shared, but are digitized separately for the individual polygons and the resulting edges do not coincide, but instead the edges intersect. The second case is similar to the first, only the resultant edges do not intersect but are “close”. The final case arises when one polygon is “inside” a second polygon but is not encoded to represent a hole in the containing polygon.

Detection of the second case will require setting buffering=True and exploring different values for tolerance.

The buffering check assumes the geometry coordinates are projected.

References

Planar Enforcement: http://ibis.geog.ubc.ca/courses/klink/gis.notes/ncgia/u12.html#SEC12.6

Examples

```python
>>> import pysal.lib as lps
>>> import geopandas as gpd
>>> rs = lps.examples.get_path('map_RS_BR.shp')
>>> rs_df = gpd.read_file(rs)
>>> wq = lps.weights.contiguity.Queen.from_dataframe(rs_df)
>>> len(wq.islands)
29
>>> wq[0]
{}
>>> wf = fuzzy_contiguity(rs_df)
>>> wf.islands
[]
>>> wf[0] == dict({239: 1.0, 59: 1.0, 152: 1.0, 23: 1.0, 107: 1.0})
True
```

Example needing to use buffering

```python
>>> import pysal.lib as lps
>>> import geopandas as gpd
>>> from shapely.geometry import Polygon
>>> p0 = Polygon([(0,0), (10,0), (10,10)])
>>> p1 = Polygon([(10,1), (10,2), (15,2)])
>>> p2 = Polygon([(12,2.001), (14, 2.001), (13,10)])
>>> gs = gpd.GeoSeries([p0,p1,p2])
>>> gdf = gpd.GeoDataFrame(geometry=gs)
>>> wf = fuzzy_contiguity(gdf)
>>> wf.islands
[2]
>>> wfb = fuzzy_contiguity(gdf, buffering=True)
>>> wfb.islands
[]
>>> wfb[2]
(1: 1.0)
```
pysal.lib.weights.min_threshold_dist_from_shapefile

Kernel weights with adaptive bandwidths.

Parameters

- **shapefile** [string] shapefile name with shp suffix.
- **radius** [float] If supplied distances will be calculated based on the given radius. p will be ignored.
- **p** [float] Minkowski p-norm distance metric parameter: 1<=p=infinity 2: Euclidean distance 1: Manhattan distance

Returns

- **d** [float] Maximum nearest neighbor distance between the n observations.

Notes

Supports polygon or point shapefiles. For polygon shapefiles, distance is based on polygon centroids. Distances are defined using coordinates in shapefile which are assumed to be projected and not geographical coordinates.

Examples

```python
>>> import pysal.lib
>>> md = pysal.lib.weights.user.min_threshold_dist_from_shapefile(pysal.lib.examples.get_path("columbus.shp"))
>>> md
0.6188641580768541
>>> pysal.lib.weights.user.min_threshold_dist_from_shapefile(pysal.lib.examples.get_path("stl_hom.shp"), pysal.lib.cg.sphere.RADIUS_EARTH_MILES)
31.846942936393717
```

pysal.lib.weights.build_lattice_shapefile

Build a lattice shapefile with nrows rows and ncols cols.

Parameters

- **nrows** [int] Number of rows
- **ncols** [int] Number of cols
- **outFileName** [str] shapefile name with shp suffix

Returns

None

pysal.lib.weights.spw_from_gal

Sparse scipy matrix for w from a gal file.
Parameters

galfile  [string] name of gal file including suffix

Returns

spw  [sparse_matrix] scipy sparse matrix in CSR format

ids  [array] identifiers for rows/cols of spw

Examples

```python
>>> import pysal.lib

spw = pysal.lib.weights.user.spw_from_gal(pysal.lib.examples.get_path("sids2.gal"))

spw.sparse.nnz
```

Set Theoretic Weights

```python

weights.w_union(w1, w2[, silence_warnings])

Parameters

w1  [W] object

w2  [W] object

silence_warnings  [boolean] Switch to turn off (default on) print statements for every observation with islands

Returns

w  [W] object

weights.w_intersection(w1, w2, w_shape, ...)

Returns a binary weights object, w, that includes only those neighbor pairs that exist in both w1 and w2.

weights.w_difference(w1, w2, w_shape, ...)

Returns a binary weights object, w, that includes only neighbor pairs in w1 that are not in w2.

weights.w_symmetric_difference(w1, w2, ...)

Returns a binary weights object, w, that includes only neighbor pairs that are not shared by w1 and w2.

weights.w_subset(w1, ids[, silence_warnings])

Returns a binary weights object, w, that includes only those observations in ids.

weights.w_clip(w1, w2[, outSP, silence_warnings])

Clip a continuous W object (w1) with a different W object (w2) so only cells where w2 has a non-zero value remain with non-zero values in w1.
```

pysal.lib.weights.w_union

Returns a binary weights object, w, that includes all neighbor pairs that exist in either w1 or w2.

Parameters

w1  [W] object

w2  [W] object

silence_warnings  [boolean] Switch to turn off (default on) print statements for every observation with islands

Returns

w  [W] object
Notes

ID comparisons are performed using ==, therefore the integer ID 2 is equivalent to the float ID 2.0. Returns a matrix with all the unique IDs from w1 and w2.

Examples

Construct rook weights matrices for two regions, one is 4x4 (16 areas) and the other is 6x4 (24 areas). A union of these two weights matrices results in the new weights matrix matching the larger one.

```python
>>> from pysal.lib.weights import lat2W
>>> w1 = lat2W(4,4)
>>> w2 = lat2W(6,4)
>>> import pysal.lib
>>> w = pysal.lib.weights.set_operations.w_union(w1, w2)
>>> w1[0] == w[0]
True
>>> w1.neighbors[15]
[11, 14]
>>> w2.neighbors[15]
[11, 14, 19]
>>> w.neighbors[15]
[19, 11, 14]
```

pysal.lib.weights.w_intersection

pysal.lib.weights.w_intersection(w1, w2, w_shape='w1', silence_warnings=False)

Returns a binary weights object, w, that includes only those neighbor pairs that exist in both w1 and w2.

Parameters

- **w1** [W] object
- **w2** [W] object
- **w_shape** [string] Defines the shape of the returned weights matrix. ‘w1’ returns a matrix with the same IDs as w1; ‘all’ returns a matrix with all the unique IDs from w1 and w2; and ‘min’ returns a matrix with only the IDs occurring in both w1 and w2.
- **silence_warnings** [boolean] Switch to turn off (default on) print statements for every observation with islands

Returns

- **w** [W] object

Notes

ID comparisons are performed using ==, therefore the integer ID 2 is equivalent to the float ID 2.0.

Examples

Construct rook weights matrices for two regions, one is 4x4 (16 areas) and the other is 6x4 (24 areas). An intersection of these two weights matrices results in the new weights matrix matching the smaller one.
```python
>>> from pysal.lib.weights import lat2W
>>> w1 = lat2W(4,4)
>>> w2 = lat2W(6,4)
>>> import pysal.lib
>>> w = pysal.lib.weights.set_operations.w_intersection(w1, w2)
>>> w1[0] == w[0]
True
>>> w1.neighbors[15]
[11, 14]
>>> w2.neighbors[15]
[11, 14, 19]
>>> w.neighbors[15]
[11, 14]
```
>>> from pysal.lib.weights import lat2W
>>> w1 = lat2W(4,4,rook=False)
>>> w2 = lat2W(4,4,rook=True)
>>> import pysal.lib

>>> w = pysal.lib.weights.set_operations.w_difference(w1, w2, constrained=False)

>>> w1[0] == w[0]
False

>>> w1.neighbors[15]
[10, 11, 14]

>>> w2.neighbors[15]
[11, 14]

>>> w.neighbors[15]
[10]

pysal.lib.weights.w_symmetric_difference

pysal.lib.weights.

w_symmetric_difference(w1, w2, w_shape='all', constrained=True, silence_warnings=False)

Returns a binary weights object, w, that includes only neighbor pairs that are not shared by w1 and w2. The w_shape and constrained parameters determine which pairs that are not shared by w1 and w2 are returned.

Parameters

w1  [W] object

w2  [W] object

w_shape  [string] Defines the shape of the returned weights matrix. ‘all’ returns a matrix with all the unique IDs from w1 and w2; and ‘min’ returns a matrix with the IDs not shared by w1 and w2.

constrained  [boolean] If False then the full set of neighbor pairs that are not shared by w1 and w2 are returned. If True then those pairs that would not be possible if w_shape=’min’ are dropped. Ignored if w_shape is set to ‘min’.

silence_warnings  [boolean] Switch to turn off (default on) print statements for every observation with islands

Returns

w  [W] object

Notes

ID comparisons are performed using ==, therefore the integer ID 2 is equivalent to the float ID 2.0.

Examples

Construct queen weights matrix for a 4x4 (16 areas) region (w1) and a rook matrix for a 6x4 (24 areas) region (w2). The symmetric difference of these two matrices (with w_shape set to ‘all’ and constrained set to False) contains the corner joins in the overlap area, all the joins in the non-overlap area.
>>> from pysal.lib.weights import lat2W
>>> import pysal.lib

>>> w1 = lat2W(4,4,rook=False)
>>> w2 = lat2W(6,4,rook=True)

>>> w = pysal.lib.weights.set_operations.w_symmetric_difference(w1, w2,
                       constrained=False)

>>> w1[0] == w[0]
False

>>> set(w.neighbors[15]) == set([10, 19])
True

pysal.lib.weights.w_subset

pysal.lib.weights.w_subset(w1, ids, silence_warnings=False)

Returns a binary weights object, w, that includes only those observations in ids.

Parameters

w1 [W] object

ids [list] A list containing the IDs to be included in the returned weights object.

silence_warnings [boolean] Switch to turn off (default on) print statements for every observation with islands

Returns

w [W] object

Examples

Construct a rook weights matrix for a 6x4 region (24 areas). By default PySAL assigns integer IDs to the areas in a region. By passing in a list of integers from 0 to 15, the first 16 areas are extracted from the previous weights matrix, and only those joins relevant to the new region are retained.

>>> from pysal.lib.weights import lat2W
>>> import pysal.lib

>>> w1 = lat2W(6,4)

>>> ids = range(16)

>>> w = pysal.lib.weights.set_operations.w_subset(w1, ids)

>>> w1[0] == w[0]
True

>>> w1.neighbors[15]
[11, 14, 19]

>>> w.neighbors[15]
[11, 14]
pysal Documentation, Release 2.0.0

pysal.lib.weights.w_clip

pysal.lib.weights.w_clip(w1, w2, outSP=True, silence_warnings=False)
Clip a continuous W object (w1) with a different W object (w2) so only cells where w2 has a non-zero value remain with non-zero values in w1.

Checks on w1 and w2 are performed to make sure they conform to the appropriate format and, if not, they are converted.

Parameters

- **w1** [W] W, scipy.sparse.csr.csr_matrix Potentially continuous weights matrix to be clipped. The clipped matrix wc will have at most the same elements as w1.
- **w2** [W] W, scipy.sparse.csr.csr_matrix Weights matrix to use as shell to clip w1. Automatically converted to binary format. Only non-zero elements in w2 will be kept non-zero in wc. 
  NOTE: assumed to be of the same shape as w1
- **outSP** [boolean] If True (default) return sparse version of the clipped W, if False, return W object of the clipped matrix
- **silence_warnings** [boolean] Switch to turn off (default on) print statements for every observation with islands

Returns

- **wc** [W] W, scipy.sparse.csr.csr_matrix Clipped W object (sparse if outSP=Trure). It inherits id_order from w1.

Examples

```python
>>> from pysal.lib.weights import lat2W
```
First create a W object from a lattice using queen contiguity and row-standardize it (note that these weights will stay when we clip the object, but they will not necessarily represent a row-standardization anymore):

```python
>>> w1 = lat2W(3, 2, rook=False)
>>> w1.transform = 'R'
```

We will clip that geography assuming observations 0, 2, 3 and 4 belong to one group and 1, 5 belong to another group and we don’t want both groups to interact with each other in our weights (i.e. w_ij = 0 if i and j in different groups). For that, we use the following method:

```python
>>> import pysal.lib
>>> w2 = pysal.lib.weights.util.block_weights(['r1', 'r2', 'r1', 'r1', 'r1', 'r2 ->'])
```
To illustrate that w2 will only be considered as binary even when the object passed is not, we can row-standardize it

```python
>>> w2.transform = 'R'
```
The clipped object wc will contain only the spatial queen relationships that occur within one group (‘r1’ or ‘r2’) but will have gotten rid of those that happen across groups

```python
>>> wcs = pysal.lib.weights.set_operations.w_clip(w1, w2, outSP=True)
```
This will create a sparse object (recommended when n is large).
If we wanted an original W object, we can control that with the argument `outSP`:

```python
>>> wc = pysal.lib.weights.set_operations.w_clip(w1, w2, outSP=False)
```

**WARNING:** there are 2 disconnected observations Island ids: [1, 5]  
```python
>>> wc.full()[0]
array([[0. , 0. , 0.33333333, 0.33333333, 0. ,
       0. ],
       [0. , 0. , 0. , 0. , 0. ,
       0. ],
       [0.2 , 0. , 0. , 0.2 , 0.2 ,
       0. ],
       [0.2 , 0. , 0. , 0.2 , 0.2 ,
       0. ],
       [0. , 0. , 0.33333333, 0.33333333, 0. ,
       0. ],
       [0. , 0. , 0. , 0. , 0. ,
       0. ]])
```  
You can check they are actually the same:

```python
>>> wcs.sparse.toarray() == wc.full()[0]
array([[ True, True, True, True, True, True],
       [ True, True, True, True, True, True],
       [ True, True, True, True, True, True],
       [ True, True, True, True, True, True],
       [ True, True, True, True, True, True],
       [ True, True, True, True, True, True]])
```  

**Spatial Lag**

- `weights.lag.spatial(w, y)`  
  Spatial lag operator.

- `weights.lag_categorical(w, y[, ties])`  
  Spatial lag operator for categorical variables.
pysal Documentation, Release 2.0.0

pysal.lib.weights.lag_spatial

pysal.lib.weights.lag_spatial(w, y)
Spatial lag operator.

If w is row standardized, returns the average of each observation’s neighbors; if not, returns the weighted sum of each observation’s neighbors.

Parameters

- **w** [W] pysal.lib spatial weights object
- **y** [array] numpy array with dimensionality conforming to w (see examples)

Returns

- **wy** [array] array of numeric values for the spatial lag

Examples

Setup a 9x9 binary spatial weights matrix and vector of data; compute the spatial lag of the vector.

```python
>>> import pysal.lib
>>> import numpy as np
>>> w = pysal.lib.weights.lat2W(3, 3)
>>> y = np.arange(9)
>>> yl = pysal.lib.weights.spatial_lag.lag_spatial(w, y)
>>> yl
array([ 4., 6., 6., 10., 16., 14., 10., 18., 12.])
```

Row standardize the weights matrix and recompute the spatial lag

```python
>>> w.transform = 'r'
>>> yl = pysal.lib.weights.spatial_lag.lag_spatial(w, y)
>>> yl
array([2. , 2. , 3. , 3.33333333, 4. ,
       4.66666667, 5. , 6. , 6. ])
```

Explicitly define data vector as 9x1 and recompute the spatial lag

```python
>>> y.shape = (9, 1)
>>> yl = pysal.lib.weights.spatial_lag.lag_spatial(w, y)
>>> yl
array([[2. ],
       [2. ],
       [3. ],
       [3.33333333],
       [4. ],
       [4.66666667],
       [5. ],
       [6. ],
       [6. ]])
```

Take the spatial lag of a 9x2 data matrix

```python
>>> yr = np.arange(8, -1, -1)
>>> yr.shape = (9, 1)
>>> x = np.hstack((y, yr))
```
>>> yl = pysal.lib.weights.spatial_lag.lag_spatial(w, x)
>>> yl
array([[2., 6.],
        [2., 6.],
        [3., 5.],
        [3.33333333, 4.66666667],
        [4., 4.],
        [4.66666667, 3.33333333],
        [5., 3.],
        [6., 2.],
        [6., 2.]])

pysal.lib.weights.lag_categorical

pysal.lib.weights.lag_categorical(w, y, ties='tryself')

Spatial lag operator for categorical variables.
Constructs the most common categories of neighboring observations, weighted by their weight strength.

Parameters

- w [W] PySAL spatial weights object
- y [iterable] iterable collection of categories (either int or string) with dimensionality conforming to w (see examples)
- ties [str] string describing the method to use when resolving ties. By default, the option is "tryself", and the category of the focal observation is included with its neighbors to try and break a tie. If this does not resolve the tie, a winner is chosen randomly. To just use random choice to break ties, pass "random" instead.

Returns

an (n x k) column vector containing the most common neighboring observation

Notes

This works on any array where the number of unique elements along the column axis is less than the number of elements in the array, for any dtype. That means the routine should work on any dtype that np.unique() can compare.

Examples

Set up a 9x9 weights matrix describing a 3x3 regular lattice. Lag one list of categorical variables with no ties.

```python
>>> import pysal.lib
>>> import numpy as np
>>> np.random.seed(12345)
>>> w = pysal.lib.weights.lat2W(3, 3)
>>> y = ['a', 'b', 'a', 'b', 'c', 'b', 'c', 'b', 'c']
>>> y_l = pysal.lib.weights.spatial_lag.lag_categorical(w, y)
>>> np.array_equal(y_l, np.array(['b', 'a', 'b', 'c', 'b', 'c', 'b', 'c', 'b']))
True
```
Explicitly reshape y into a (9x1) array and calculate lag again

```python
>>> yvect = np.array(y).reshape(9,1)
>>> yvect_l = pysal.lib.weights.spatial_lag.lag_categorical(w,yvect)
>>> check = np.array([ [i] for i in ['b', 'a', 'b', 'c', 'b', 'c', 'b', 'c', 'b']])
>>> np.array_equal(yvect_l, check)
True
```

compute the lag of a 9x2 matrix of categories

```python
>>> y2 = ['a', 'c', 'c', 'd', 'b', 'a', 'd', 'd', 'c']
>>> ym = np.vstack((y,y2)).T
>>> ym_lag = pysal.lib.weights.spatial_lag.lag_categorical(w,ym)
>>> check = np.array([['b', 'd'], ['a', 'c'], ['b', 'c'], ['c', 'd'], ['b', 'd'],
                   ['c', 'c'], ['b', 'd'], ['c', 'd'], ['b', 'c']])
>>> np.array_equal(check, ym_lag)
True
```

**cg: Computational Geometry**

**alpha_shapes**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cg.alpha_shape(xys, alpha)</code></td>
<td>Alpha-shape delineation (Edelsbrunner, Kirkpatrick &amp; Seidel, 1983) from a collection of points ...</td>
</tr>
<tr>
<td><code>cg.alpha_shape_auto(xys, step=1, verbose=False)</code></td>
<td>Computation of alpha-shape delineation with automated selection of alpha.</td>
</tr>
</tbody>
</table>

**pysal.lib.cg.alpha_shape**

`pysal.lib.cg.alpha_shape(xys, alpha)`

Alpha-shape delineation (Edelsbrunner, Kirkpatrick & Seidel, 1983) from a collection of points ...

**Returns**

`shapes` [GeoSeries] Polygon(s) resulting from the alpha shape algorithm. The GeoSeries object remains so even if only a single polygon is returned. There is no CRS included in the object.

**References**


**pysal.lib.cg.alpha_shape_auto**

`pysal.lib.cg.alpha_shape_auto(xys, step=1, verbose=False)`

Computation of alpha-shape delineation with automated selection of alpha. ...

This method uses the algorithm proposed by Edelsbrunner, Kirkpatrick & Seidel (1983) to return the tightest polygon that contains all points in `xys`. The algorithm ranks every point based on its radious and iterates over each point, checking whether the maximum alpha that would keep the point and all the other ones in the set with
smaller radii results in a single polygon. If that is the case, it moves to the next point; otherwise, it retains the previous alpha value and returns the polygon as *shapely* geometry.

**Returns**

poly [shapely.Polygon] Tightest alpha-shape polygon containing all points in *xys*

**References**


**voronoi**

```python
cg.voronoi_frames(points[, radius])
```

Composite helper to return Voronoi regions and generator points as individual dataframes

**pysal.lib.cg.voronoi_frames**

**pysal.lib.cg.voronoi_frames**(points, radius=None)

Composite helper to return Voronoi regions and generator points as individual dataframes

**Parameters**

points [array-like] originator points

**Returns**

_ [tuple] (region_df, points_df)

region_df [GeoDataFrame (if geopandas available, otherwise Pandas DataFrame)] Finite Voronoi polygons as geometries

points_df [GeoDataFrame (if geopandas available, otherwise Pandas DataFrame)] Originator points as geometries

**Notes**

If Geopandas is not available the return types will be Pandas DataFrames each with a geometry column populated with PySAL shapes. If Geopandas is available, return types are GeoDataFrames with a geometry column populated with shapely geometry types.

**Examples**

```python
>>> points = [(10.2, 5.1), (4.7, 2.2), (5.3, 5.7), (2.7, 5.3)]
>>> regions_df, points_df = voronoi_frames(points)
>>> regions_df.shape
(4, 1)
>>> regions_df.shape == points_df.shape
True
```
sphere

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cg.RADIUS_EARTH_KM</code></td>
<td>float(x) -&gt; floating point number</td>
</tr>
<tr>
<td><code>cg.RADIUS_EARTH_MILES</code></td>
<td>float(x) -&gt; floating point number</td>
</tr>
<tr>
<td><code>cg.arcdist(pt0, pt1[, radius])</code></td>
<td>Arc distance between two points on a sphere.</td>
</tr>
<tr>
<td><code>cg.arcdist2linear(arc_dist[, radius])</code></td>
<td>Convert an arc distance (spherical earth) to a linear distance (R3) in the unit sphere.</td>
</tr>
<tr>
<td><code>cg.brute_knn(pts, k[, mode])</code></td>
<td>valid modes are ['arc','xrz']</td>
</tr>
<tr>
<td><code>cg.fast_knn(pts, k[, return_dist])</code></td>
<td>Computes k nearest neighbors on a sphere.</td>
</tr>
<tr>
<td><code>cg.fast_threshold(pts, dist[, radius])</code></td>
<td>Find all neighbors on a sphere within a threshold distance</td>
</tr>
<tr>
<td><code>cg.linear2arcdist(linear_dist[, radius])</code></td>
<td>Convert a linear distance in the unit sphere (R3) to an arc distance based on supplied radius</td>
</tr>
<tr>
<td><code>cg.toLngLat(xyz)</code></td>
<td>Convert x,y,z to latitude and longitude</td>
</tr>
<tr>
<td><code>cg.toXYZ(pt)</code></td>
<td>Convert a point’s latitude and longitude to x,y,z</td>
</tr>
<tr>
<td><code>cg.lonlat(pointslist)</code></td>
<td>Converts point order from lat-lon tuples to lon-lat (x,y) tuples</td>
</tr>
<tr>
<td><code>cg.harcdist(p0, p1[, lonx, radius])</code></td>
<td>Alternative arc distance function, uses haversons formula</td>
</tr>
<tr>
<td><code>cg.geointerpolate(p0, p1, t[, lonx])</code></td>
<td>Finds a point on a sphere along the great circle distance between two points on a sphere also known as a way point in great circle navigation</td>
</tr>
<tr>
<td><code>cg.geogrid(pup, pdown, k[, lonx])</code></td>
<td>Computes a k+1 by k+1 set of grid points for a bounding box in lat-lon uses geointerpolate</td>
</tr>
</tbody>
</table>

**pysal.lib.cg.RADIUS_EARTH_KM**

```python
pysal.lib.cg.RADIUS_EARTH_KM = 6371.0
```

Convert a string or number to a floating point number, if possible.

**pysal.lib.cg.RADIUS_EARTH_MILES**

```python
pysal.lib.cg.RADIUS_EARTH_MILES = 3958.755865744055
```

Convert a string or number to a floating point number, if possible.

**pysal.lib.cg.arcdist**

```python
pysal.lib.cg.arcdist(pt0, pt1, radius=6371.0)
```

Arc distance between two points on a sphere.

**Parameters**

- `pt0` [point] assumed to be in form (lng,lat)
- `pt1` [point] assumed to be in form (lng,lat)
- `radius` [radius of the sphere] defaults to Earth’s radius

Returns

The arc distance between pt0 and pt1 using supplied radius

Examples

```python
>>> pt0 = (0,0)
>>> pt1 = (180,0)
>>> d = arcdist(pt0,pt1,RADIUS_EARTH_MILES)
>>> d == math.pi*RADIUS_EARTH_MILES
True
```

pysal.lib.cg.arcdist2linear

```python
pysal.lib.cg.arcdist2linear(arc_dist, radius=6371.0)
```

Convert an arc distance (spherical earth) to a linear distance (R3) in the unit sphere.

Examples

```python
>>> pt0 = (0,0)
>>> pt1 = (180,0)
>>> d = arcdist(pt0,pt1,RADIUS_EARTH_MILES)
>>> d == math.pi*RADIUS_EARTH_MILES
True
>>> arcdist2linear(d,RADIUS_EARTH_MILES)
2.0
```

pysal.lib.cg.brute_knn

```python
pysal.lib.cg.brute_knn(pts, k, mode='arc')
```

valid modes are ['arc','xrz']

pysal.lib.cg.fast_knn

```python
pysal.lib.cg.fast_knn(pts, k, return_dist=False)
```

Computes k nearest neighbors on a sphere.

Parameters

- **pts** [list of x,y pairs]
- **k** [int] Number of points to query
- **return_dist** [bool] Return distances in the ‘wd’ container object

Returns

- **wn** [list] list of neighbors
- **wd** [list] list of neighbor distances (optional)
**pysal.lib.cg.fast_threshold**

**pysal.lib.cg.fast_threshold** *(pts, dist, radius=6371.0)*

Find all neighbors on a sphere within a threshold distance

**Parameters**
- `pointslist` [list of lat-lon tuples (Note, has to be a list, even for one point)]
- `dist`: float threshold distance
- `radius`: float sphere’s radius

**Returns**
- `dict`: key is id of point, value is a list of ids for other points within dist of key point

**pysal.lib.cg.linear2arcdist**

**pysal.lib.cg.linear2arcdist** *(linear_dist, radius=6371.0)*

Convert a linear distance in the unit sphere (R3) to an arc distance based on supplied radius

**Examples**

```python
>>> pt0 = (0,0)
>>> pt1 = (180,0)
>>> d = arcdist(pt0,pt1,RADIUS_EARTH_MILES)
>>> d == linear2arcdist(2.0, radius = RADIUS_EARTH_MILES)
True
```

**pysal.lib.cg.toLngLat**

**pysal.lib.cg.toLngLat** *(xyz)*

Convert x,y,z to latitude and longitude

**pysal.lib.cg.toXYZ**

**pysal.lib.cg.toXYZ** *(pt)*

Convert a point’s latitude and longitude to x,y,z

**Parameters**
- `pt0` [point] assumed to be in form (lng, lat)
- `pt1` [point] assumed to be in form (lng, lat)

**Returns**
- `x, y, z`
pysal.lib.cg.lonlat

pysal.lib.cg.lonlat(pointslist)
Converting point order from lat-lon tuples to lon-lat (x,y) tuples

Parameters
pointslist [list of lat-lon tuples (Note, has to be a list, even for one point)]

Returns
newpts [list with tuples of points in lon-lat order]

pysal.lib.cg.harcdist

pysal.lib.cg.harcdist(p0, p1, lonx=True, radius=6371.0)
Alternative arc distance function, uses haversine formula

Parameters
p0 [first point as a tuple in decimal degrees]
p1 [second point as a tuple in decimal degrees]
lonx [boolean to assess the order of the coordinates,] for lon, lat (default) = True, for lat, lon = False
radius [radius of the earth at the equator as a sphere] default: RADIUS_EARTH_KM (6371.0 km) options: RADIUS_EARTH_MILES (3959.0 miles)

Returns
d [distance in units specified, km, miles or radians (for None)]

pysal.lib.cg.geointerpolate

pysal.lib.cg.geointerpolate(p0, p1, t, lonx=True)
Finds a point on a sphere along the great circle distance between two points on a sphere also known as a way point in great circle navigation

Parameters
p0 [first point as a tuple in decimal degrees]
p1 [second point as a tuple in decimal degrees]
t [proportion along great circle distance between p0 and p1] e.g., t=0.5 would find the mid-point
lonx [boolean to assess the order of the coordinates,] for lon, lat (default) = True, for lat, lon = False

Returns
x,y [tuple in decimal degrees of lon-lat (default) or lat-lon,] depending on setting of lonx; in other words, the same order is used as for the input
**pysal Documentation, Release 2.0.0**

**pysal.lib.cg.geogrid**

```python
pysal.lib.cg.geogrid(pup, pdown, k, lonx=True)
```

Computes a k+1 by k+1 set of grid points for a bounding box in lat-lon uses geointerpolate

**Parameters**

- **pup** [tuple with lat-lon or lon-lat for upper left corner of bounding box]
- **pdown** [tuple with lat-lon or lon-lat for lower right corner of bounding box]
- **k** [number of grid cells (grid points will be one more)]
- **lonx** [boolean to assess the order of the coordinates,] for lon,lat (default) = True, for lat,lon = False

**Returns**

- **grid** [list of tuples with lat-lon or lon-lat for grid points, row by row.] starting with the top row and moving to the bottom; coordinate tuples are returned in same order as input

---

**shapes**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cg.Point(loc)</code></td>
<td>Geometric class for point objects.</td>
</tr>
<tr>
<td><code>cg.LineSegment(start_pt, end_pt)</code></td>
<td>Geometric representation of line segment objects.</td>
</tr>
<tr>
<td><code>cg.Line(m, b)</code></td>
<td>Geometric representation of line objects.</td>
</tr>
<tr>
<td><code>cg.Ray(origin, second_p)</code></td>
<td>Geometric representation of ray objects.</td>
</tr>
<tr>
<td><code>cg.Chain(vert)</code></td>
<td>Geometric representation of a chain, also known as a polyline.</td>
</tr>
<tr>
<td><code>cg.Polygon(vert, [holes])</code></td>
<td>Geometric representation of polygon objects.</td>
</tr>
<tr>
<td><code>cg.Rectangle(left, lower, right, upper)</code></td>
<td>Geometric representation of rectangle objects.</td>
</tr>
<tr>
<td><code>cg.asShape(obj)</code></td>
<td>Returns a pysal shape object from obj.</td>
</tr>
</tbody>
</table>

---

**pysal.lib.cg.Point**

```python
class pysal.lib.cg.Point(loc)
```

Geometric class for point objects.

**Attributes**

- **None**

**__init__(loc)**

Returns an instance of a Point object.

**__init__((number, number)) -> Point**

Test tag: `<tc>#is#Point.__init__</tc>` Test tag: `<tc>#tests#Point.__init__</tc>

**Parameters**

- **loc** [tuple location (number x-tuple, x > 1)]

**Examples**

```python
>>> p = Point((1, 3))
```
### Methods

**__init__(loc)**  
Returns an instance of a Point object.

---

### pysal.lib.cg.LineSegment

**class** `pysal.lib.cg.LineSegment(start_pt, end_pt)`  
Geometric representation of line segment objects.

**Parameters**

- `start_pt` [Point] Point where segment begins
- `end_pt` [Point] Point where segment ends

**Attributes**

- `p1` [Point] HELPER METHOD.
- `p2` [Point] HELPER METHOD.
- `bounding_box` [tuple] Returns the minimum bounding box of a LineSegment object.
- `len` [float] Returns the length of a LineSegment object.
- `line` [Line] Returns a Line object of the line which the segment lies on.

**Methods**

- `get_swap()`  
  Returns a LineSegment object which has its endpoints swapped.
- `intersect(other)`  
  Test whether segment intersects with other segment
- `is_ccw(pt)`  
  Returns whether a point is counterclockwise of the segment.
- `is_cw(pt)`  
  Returns whether a point is clockwise of the segment.
- `sw_ccw(pt)`  
  Sedgewick test for pt being ccw of segment

**__init__(start_pt, end_pt)**  
Creates a LineSegment object.

```python
__init__(Point, Point) -> LineSegment
```

Test tag: `<tc>#is#LineSegment.__init__</tc>`  
Test tag: `<tc>#tests#LineSegment.__init__</tc>`

---

### Examples

```python
>>> ls = LineSegment(Point((1, 2)), Point((5, 6)))
```

---

### Attributes

- None

---

### Methods
**,py**sal Documentation, Release 2.0.0

**py**sal.lib.cg.Line

class **py**sal.lib.cg.Line(m, b)

Geometric representation of line objects.

**Attributes**

- `m` [float]  slope
- `b` [float]  y-intercept

**Methods**

- `x(y)` Returns the x-value of the line at a particular y-value.
- `y(x)` Returns the y-value of the line at a particular x-value.

```
>>> ls = Line(1, 0)
>>> ls.m
```

(continues on next page)
Methods

```python
>>> l = Ray(Point((0, 0)), Point((1, 0)))
>>> str(l.o)
'(0.0, 0.0)'
>>> str(l.p)
'(1.0, 0.0)'
```

### pysal.lib.cg.Ray

**class** *pysal.lib.cg.Ray*(origin, second_p)

Geometric representation of ray objects.

**Attributes**

- **o** [Point] Origin (point where ray originates)
- **p** [Point] Second point on the ray (not point where ray originates)

**__init__(origin, second_p)**

Returns a ray with the values specified.

**Parameters**

- **origin** [the point where the ray originates]
- **second_p** [the second point specifying the ray (not the origin)]

**Examples**

```python
>>> l = Ray(Point((0, 0)), Point((1, 0)))
>>> str(l.o)
'(0.0, 0.0)'
>>> str(l.p)
'(1.0, 0.0)'
```

Methods

```python
__init__(origin, second_p) Returns a ray with the values specified.
```

### pysal.lib.cg.Chain

**class** *pysal.lib.cg.Chain*(vertices)

Geometric representation of a chain, also known as a polyline.

**Attributes**

- **vertices** [list] Returns the vertices of the chain in clockwise order.
len [float] Returns the geometric length of the chain.

__init__(vertices)
Returns a chain created from the points specified.
__init__(Point list or list of Point lists) -> Chain

Parameters
vertices [list – Point list or list of Point lists.]

Examples

>>> c = Chain([Point((0, 0)), Point((1, 0)), Point((1, 1)), Point((2, 1))])

Methods

__init__(vertices) Returns a chain created from the points specified.

Attributes

arclen Returns the geometric length of the chain computed using arcdistance (meters).
bounding_box Returns the bounding box of the chain.
len Returns the geometric length of the chain.
parts Returns the parts of the chain.
segments Returns the segments that compose the Chain
vertices Returns the vertices of the chain in clockwise order.

pysal.lib.cg.Polygon

class pysal.lib.cg.Polygon(vertices, holes=None)
Geometric representation of polygon objects.

Attributes

vertices [list] Returns the vertices of the polygon in clockwise order.
len [int] Returns the number of vertices in the polygon.
perimeter [float] Returns the perimeter of the polygon.
bounding_box [Rectangle] Returns the bounding box of the polygon.
bbox [List] Returns the bounding box of the polygon as a list
area [float] Returns the area of the polygon.
centroid [tuple] Returns the centroid of the polygon

Methods
build_quad_tree_structure()  Build the quad tree structure for this polygon.
contains_point(point)  Test if polygon contains point

__init__(vertices, holes=None)
Returns a polygon created from the objects specified.

__init__(Point list or list of Point lists, holes list) -> Polygon

  Parameters  
  vertices [list – a list of vertices or a list of lists of vertices.]
  holes [list – a list of sub-polygons to be considered as holes.]
  is_quad_tree_structure_built : bool – record if the quad tree structure has been built for
  this polygon. This quad tree structure could help speed up the contains_point test

Examples

```python
>>> p1 = Polygon([Point((0, 0)), Point((1, 0)), Point((1, 1)), Point((0, 1))])
```

Methods

__init__(vertices, holes)  Returns a polygon created from the objects specified.
build_quad_tree_structure()  Build the quad tree structure for this polygon.
contains_point(point)  Test if polygon contains point

Attributes

area  Returns the area of the polygon.
bbox  Returns the bounding box of the polygon as a list
bounding_box  Returns the bounding box of the polygon.
centroid  Returns the centroid of the polygon
holes  Returns the holes of the polygon in clockwise order.
len  Returns the number of vertices in the polygon.
parts  Returns the parts of the polygon in clockwise order.
perimeter  Returns the perimeter of the polygon.
vertices  Returns the vertices of the polygon in clockwise order.

pysal.lib.cg.Rectangle

class pysal.lib.cg.Rectangle (left, lower, right, upper)
Geometric representation of rectangle objects.

  Attributes  
  left [float] Minimum x-value of the rectangle
  lower [float] Minimum y-value of the rectangle
  right [float] Maximum x-value of the rectangle
upper [float] Maximum y-value of the rectangle

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_centroid(new_center)</td>
<td>Moves the rectangle center to a new specified point.</td>
</tr>
<tr>
<td>set_scale(scale)</td>
<td>Rescales the rectangle around its center.</td>
</tr>
</tbody>
</table>

__init__(left, lower, right, upper)

Returns a Rectangle object.

__init__(number, number, number, number) -> Rectangle

Parameters

- left [the minimum x-value of the rectangle]
- lower [the minimum y-value of the rectangle]
- right [the maximum x-value of the rectangle]
- upper [the maximum y-value of the rectangle]

Examples

```python
>>> r = Rectangle(-4, 3, 10, 17)
>>> r.left  #minx
-4.0
>>> r.lower  #miny
3.0
>>> r.right  #maxx
10.0
>>> r.upper  #maxy
17.0
```

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(left, lower, right, upper)</td>
<td>Returns a Rectangle object.</td>
</tr>
<tr>
<td>set_centroid(new_center)</td>
<td>Moves the rectangle center to a new specified point.</td>
</tr>
<tr>
<td>set_scale(scale)</td>
<td>Rescales the rectangle around its center.</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>area</td>
<td>Returns the area of the Rectangle.</td>
</tr>
<tr>
<td>height</td>
<td>Returns the height of the Rectangle.</td>
</tr>
<tr>
<td>width</td>
<td>Returns the width of the Rectangle.</td>
</tr>
</tbody>
</table>

pysal.lib.cg.asShape

pysal.lib.cg.asShape(obj)

Returns a pysal shape object from obj. obj must support the __geo_interface__. 
### cg.bbcommon

**Function:**
Old Stars method for bounding box overlap testing Also defined in pysal.weights._cont_binning

**Parameters:**
- `bb`: First bounding box
- `bbother`: Other bounding box

**Returns:**
- Returns True if there is an overlap.

**Examples**
```python
g0 = [0,0,10,10]
g1 = [10,0,20,10]
bbcommon(g0,g1)
```

### cg.get_bounding_box

**Function:**
Find bounding box for a list of geometries

**Parameters:**
- `items`: A list of geometries

**Returns:**
- Returns the bounding box for the list of geometries.
Parameters

items: list  PySAL shapes

Returns

Rectangle

Examples

```python
>>> bb = get_bounding_box([Point((-1, 5)), Rectangle(0, 6, 11, 12)])
>>> bb.left
-1.0
>>> bb.lower
5.0
>>> bb.right
11.0
>>> bb.upper
12.0
```

**pysal.lib.cg.get_angle_between**

`pysal.lib.cg.get_angle_between(ray1, ray2)`

Returns the angle formed between a pair of rays which share an origin `get_angle_between(Ray, Ray) -> number`

Parameters

ray1 [a ray forming the beginning of the angle measured]

ray2 [a ray forming the end of the angle measured]

Examples

```python
>>> get_angle_between(Ray(Point((0, 0)), Point((1, 0))), Ray(Point((0, 0)),
                   Point((1, 0))))
0.0
```

**pysal.lib.cg.is_collinear**

`pysal.lib.cg.is_collinear(p1, p2, p3)`

Returns whether a triplet of points is collinear.

`is_collinear(Point, Point, Point) -> bool`

Parameters

p1 [a point (Point)]

p2 [another point (Point)]

p3 [yet another point (Point)]
Examples

```python
>>> is_collinear(Point((0, 0)), Point((1, 1)), Point((5, 5)))
True
>>> is_collinear(Point((0, 0)), Point((1, 1)), Point((5, 0)))
False
```

**pysal.lib.cg.get_segments_intersect**

pysal.lib.cg.get_segments_intersect(seg1, seg2)

Returns the intersection of two segments.

get_segments_intersect(LineSegment, LineSegment) -> Point or LineSegment

Parameters

- seg1 [a segment to check intersection for]
- seg2 [a segment to check intersection for]

Examples

```python
>>> seg1 = LineSegment(Point((0, 0)), Point((0, 10)))
>>> seg2 = LineSegment(Point((-5, 5)), Point((5, 5)))
>>> i = get_segments_intersect(seg1, seg2)
>>> isinstance(i, Point)
True
>>> str(i)
'(0.0, 5.0)'
>>> seg3 = LineSegment(Point((100, 100)), Point((100, 101)))
>>> i = get_segments_intersect(seg2, seg3)
```

**pysal.lib.cg.get_segment_point_intersect**

pysal.lib.cg.get_segment_point_intersect(seg, pt)

Returns the intersection of a segment and point.

get_segment_point_intersect(LineSegment, Point) -> Point

Parameters

- seg [a segment to check intersection for]
- pt [a point to check intersection for]

Examples

```python
>>> seg = LineSegment(Point((0, 0)), Point((0, 10)))
>>> pt = Point((0, 5))
>>> i = get_segment_point_intersect(seg, pt)
>>> str(i)
'(0.0, 5.0)'
>>> pt2 = Point((5, 5))
>>> get_segment_point_intersect(seg, pt2)
```
pysal.lib.cg.get_polygon_point_intersect

pysal.lib.cg.get_polygon_point_intersect(poly, pt)

Returns the intersection of a polygon and point.

get_polygon_point_intersect(Polygon, Point) -> Point

Parameters

poly [a polygon to check intersection for]
pt [a point to check intersection for]

Examples

```python
>>> poly = Polygon([Point((0, 0)), Point((1, 0)), Point((1, 1)), Point((0, 1))])
>>> pt = Point((0.5, 0.5))
>>> i = get_polygon_point_intersect(poly, pt)
>>> str(i)
'(0.5, 0.5)'
>>> pt2 = Point((2, 2))
>>> get_polygon_point_intersect(poly, pt2)
```

pysal.lib.cg.get_rectangle_point_intersect

pysal.lib.cg.get_rectangle_point_intersect(rect, pt)

Returns the intersection of a rectangle and point.

get_rectangle_point_intersect(Rectangle, Point) -> Point

Parameters

rect [a rectangle to check intersection for]
pt [a point to check intersection for]

Examples

```python
>>> rect = Rectangle(0, 0, 5, 5)
>>> pt = Point((1, 1))
>>> i = get_rectangle_point_intersect(rect, pt)
>>> str(i)
'(1.0, 1.0)'
>>> pt2 = Point((10, 10))
>>> get_rectangle_point_intersect(rect, pt2)
```

pysal.lib.cg.get_ray_segment_intersect

pysal.lib.cg.get_ray_segment_intersect(ray, seg)

Returns the intersection of a ray and line segment.

get_ray_segment_intersect(Ray, Point) -> Point or LineSegment

Parameters
Example

```python
>>> ray = Ray(Point((0, 0)), Point((0, 1)))
>>> seg = LineSegment(Point((-1, 10)), Point((1, 10)))
>>> i = get_ray_segment_intersect(ray, seg)
>>> isinstance(i, Point)
True
>>> str(i)
'(0.0, 10.0)'
>>> seg2 = LineSegment(Point((10, 10)), Point((10, 11)))
>>> get_ray_segment_intersect(ray, seg2)
```

**pysal.lib.cg.get_rectangle_rectangle_intersection**

```python
pysal.lib.cg.get_rectangle_rectangle_intersection(r0, r1, checkOverlap=True)
```

Returns the intersection between two rectangles.

**Note:** Algorithm assumes the rectangles overlap. checkOverlap=False should be used with extreme caution.

get_rectangle_rectangle_intersection(r0, r1) -> Rectangle, Segment, Point or None

**Parameters**

- r0 [a Rectangle]
- r1 [a Rectangle]

**Examples**

```python
>>> r0 = Rectangle(0, 4, 6, 9)
>>> r1 = Rectangle(4, 0, 9, 7)
>>> ri = get_rectangle_rectangle_intersection(r0, r1)
>>> ri[:]
[4.0, 4.0, 6.0, 7.0]
>>> r0 = Rectangle(0, 0, 4, 4)
>>> r1 = Rectangle(2, 1, 6, 3)
>>> ri = get_rectangle_rectangle_intersection(r0, r1)
>>> ri[:]
[2.0, 1.0, 4.0, 3.0]
>>> r0 = Rectangle(0, 0, 4, 4)
>>> r1 = Rectangle(2, 1, 3, 2)
>>> ri = get_rectangle_rectangle_intersection(r0, r1)
>>> ri[:]== r1[:]
True
```

**pysal.lib.cg.get_polygon_point_dist**

```python
pysal.lib.cg.get_polygon_point_dist(poly, pt)
```

Returns the distance between a polygon and point.

generate_polygon_point_dist(Polygon, Point) -> number
Parameters

poly [a polygon to compute distance from]
pt [a point to compute distance from]

Examples

```python
>>> poly = Polygon([Point((0, 0)), Point((1, 0)), Point((1, 1)), Point((0, 1))])
>>> pt = Point((2, 0.5))
>>> get_polygon_point_dist(poly, pt)
1.0
>>> pt2 = Point((0.5, 0.5))
>>> get_polygon_point_dist(poly, pt2)
0.0
```

**pysal.lib.cg.get_points_dist**

pysal.lib.cg.get_points_dist(pt1, pt2)

Returns the distance between a pair of points.

get_points_dist(Point, Point) -> number

Parameters

pt1 [a point]
pt2 [the other point]

Examples

```python
>>> get_points_dist(Point((4, 4)), Point((4, 8)))
4.0
>>> get_points_dist(Point((0, 0)), Point((0, 0)))
0.0
```

**pysal.lib.cg.get_segment_point_dist**

pysal.lib.cg.get_segment_point_dist(seg, pt)

Returns the distance between a line segment and point and distance along the segment of the closest point on
the segment to the point as a ratio of the length of the segment.

get_segment_point_dist(LineSegment, Point) -> (number, number)

Parameters

seg [a line segment to compute distance from]
pt [a point to compute distance from]

Examples
```python
>>> seg = LineSegment(Point((0, 0)), Point((10, 0)))
>>> pt = Point((5, 5))
>>> get_segment_point_dist(seg, pt)
(5.0, 0.5)
>>> pt2 = Point((0, 0))
>>> get_segment_point_dist(seg, pt2)
(0.0, 0.0)
```

**pysal.lib.cg.get_point_at_angle_and_dist**

*pysal.lib.cg.get_point_at_angle_and_dist*(ray, angle, dist)

Returns the point at a distance and angle relative to the origin of a ray.

get_point_at_angle_and_dist(Ray, number, number) -> Point

**Parameters**

- **ray** [the ray which the angle and distance are relative to]
- **angle** [the angle relative to the ray at which the point is located]
- **dist** [the distance from the ray origin at which the point is located]

**Examples**

```python
>>> ray = Ray(Point((0, 0)), Point((1, 0)))
>>> pt = get_point_at_angle_and_dist(ray, math.pi, 1.0)
>>> isinstance(pt, Point)
True
>>> round(pt[0], 8)
-1.0
>>> round(pt[1], 8)
0.0
```

**pysal.lib.cg.convex_hull**

*pysal.lib.cg.convex_hull*(points)

Returns the convex hull of a set of points.

convex_hull(Point list) -> Polygon

**Parameters**

- **points** [a list of points to compute the convex hull for]

**Examples**

```python
>>> points = [Point((0, 0)), Point((4, 4)), Point((4, 0)), Point((3, 1))]
>>> convex_hull(points)
[(0.0, 0.0), (4.0, 0.0), (4.0, 4.0)]
```
pysal Documentation, Release 2.0.0

pysal.lib.cg.is_clockwise
pysal.lib.cg.is_clockwise(vertices)
Returns whether a list of points describing a polygon are clockwise or counterclockwise.
is_clockwise(Point list) -> bool
Parameters
vertices [a list of points that form a single ring]
Examples
>>> is_clockwise([Point((0, 0)), Point((10, 0)), Point((0, 10))])
False
>>> is_clockwise([Point((0, 0)), Point((0, 10)), Point((10, 0))])
True
>>> v = [(-106.57798, 35.174143999999998), (-106.583412, 35.174141999999996), (˓→106.58417999999999, 35.174143000000001), (-106.58377999999999, 35.
˓→175542999999998), (-106.58287999999999, 35.180543), (-106.58263099999999, 35.
˓→181455), (-106.58257999999999, 35.181643000000001), (-106.58198299999999, 35.
˓→184615000000001), (-106.58148, 35.187242999999995), (-106.58127999999999, 35.
˓→188243), (-106.58138, 35.188243), (-106.58108, 35.189442999999997), (-106.58104,
˓→ 35.189644000000001), (-106.58028, 35.193442999999995), (-106.580029, 35.
˓→194541000000001), (-106.57974399999999, 35.195785999999998), (-106.579475, 35.
˓→196961999999999), (-106.57922699999999, 35.198042999999998), (-106.578397, 35.
˓→201665999999996), (-106.57827999999999, 35.201642999999997), (-106.
˓→57737999999999, 35.201642999999997), (-106.57697999999999, 35.201543000000001),
˓→(-106.56436599999999, 35.200311999999997), (-106.56058, 35.199942999999998), (˓→106.56048, 35.197342999999996), (-106.56048, 35.195842999999996), (-106.56048,
˓→35.194342999999996), (-106.56048, 35.193142999999999), (-106.56048, 35.
˓→191873999999999), (-106.56048, 35.191742999999995), (-106.56048, 35.
˓→190242999999995), (-106.56037999999999, 35.188642999999999), (-106.
˓→56037999999999, 35.187242999999995), (-106.56037999999999, 35.186842999999996),
˓→(-106.56037999999999, 35.186552999999996), (-106.56037999999999, 35.
˓→185842999999998), (-106.56037999999999, 35.184443000000002), (-106.
˓→56037999999999, 35.182943000000002), (-106.56037999999999, 35.181342999999998),
˓→(-106.56037999999999, 35.180433000000001), (-106.56037999999999, 35.
˓→179943000000002), (-106.56037999999999, 35.178542999999998), (-106.
˓→56037999999999, 35.177790999999999), (-106.56037999999999, 35.177143999999998),
˓→(-106.56037999999999, 35.175643999999998), (-106.56037999999999, 35.
˓→174444000000001), (-106.56037999999999, 35.174043999999995), (-106.560526, 35.
˓→174043999999995), (-106.56478, 35.174043999999995), (-106.56627999999999, 35.
˓→174143999999998), (-106.566541, 35.174144999999996), (-106.569023, 35.
˓→174157000000001), (-106.56917199999999, 35.174157999999998), (-106.56938, 35.
˓→174143999999998), (-106.57061499999999, 35.174143999999998), (-106.
˓→57097999999999, 35.174143999999998), (-106.57679999999999, 35.174143999999998),
˓→(-106.57798, 35.174143999999998)]
>>> is_clockwise(v)
True

pysal.lib.cg.point_touches_rectangle
pysal.lib.cg.point_touches_rectangle(point, rect)
Returns True if the point is in the rectangle or touches it’s boundary.

78

Chapter 1. PySAL Components


point_touches_rectangle(point, rect) -> bool

**Parameters**

- **point** [Point or Tuple]
- **rect** [Rectangle]

**Examples**

```python
>>> rect = Rectangle(0,0,10,10)
>>> a = Point((5,5))
>>> b = Point((10,5))
>>> c = Point((11,11))
>>> point_touches_rectangle(a, rect)
1
>>> point_touches_rectangle(b, rect)
1
>>> point_touches_rectangle(c, rect)
0
```

**pysal.lib.cg.get_shared_segments**

```python
from pysal.lib.cg.shapes import Polygon
x = [0, 0, 1, 1]
y = [0, 1, 1, 0]
poly1 = Polygon( list(map(Point, zip(x,y))) )
x = [a+1 for a in x]
poly2 = Polygon( list(map(Point, zip(x,y))) )
get_shared_segments(poly1, poly2, bool_ret=True)
```

**pysal.lib.cg.distance_matrix**

```python
distance_matrix(X, p=2.0, threshold=50000000.0)
```

Distance Matrices

XXX Needs optimization/integration with other weights in pysal

**Parameters**

- **X** [An, n by k numpy.ndarray] Where n is number of observations k is number of dimmensions (2 for x,y)
**p** [float] Minkowski p-norm distance metric parameter: 1<=p<=infinity 2: Euclidean distance 1: Manhattan distance

**threshold** [positive integer] If (n**2)*32 > threshold use scipy.spatial.distance_matrix instead of working in ram, this is roughly the amount of ram (in bytes) that will be used.

**Examples**

```python
>>> x, y = [r.flatten() for r in np.indices((3, 3))]
>>> data = np.array([x, y]).T
>>> d = distance_matrix(data)
>>> np.array(d)
array([[0., 1., 2., 1., 1.41421356, 2.23606798, 2., 2.23606798, 2.82842712],
       [1., 0., 1., 1.41421356, 1.41421356, 2.23606798, 2.23606798, 2.82842712, 2.23606798],
       [2., 1., 0., 2.23606798, 2.23606798, 0. , 1. , 1.41421356, 1.41421356],
       [2.82842712, 2.23606798, 2. , 2.23606798, 2.23606798, 2.82842712, 0. , 1. , 2. ],
       [1.41421356, 1.41421356, 1. , 2. , 1. , 0. , 1. , 1.41421356, 1.41421356],
       [1. , 1.41421356, 1. , 1.41421356, 1.41421356, 1.41421356, 1. , 2. , 1. ],
       [2. , 2.23606798, 2.82842712, 1. , 1.41421356, 2.23606798, 1. , 2. , 1. ],
       [2.23606798, 2. , 2.82842712, 1. , 2. , 1. , 0. , 1. , 1.41421356],
       [1.41421356, 1. , 0. , 1. , 1.41421356, 2.23606798, 2.82842712, 2.23606798, 2.82842712],
       [2.23606798, 2.23606798, 2.82842712, 2.23606798, 2.82842712, 2.23606798, 2.23606798, 2.82842712, 2.82842712]]
```

**locators**

- **`cg.Grid(bounds, resolution)`**
  - Representation of a binning data structure.

- **`cg.PointLocator(points)`**
  - An abstract representation of a point indexing data structure.

- **`cg.PolygonLocator(polygons)`**
  - An abstract representation of a polygon indexing data structure.

**pysal.lib.cg.Grid**

**class** **pysal.lib.cg.Grid**(bounds, resolution)

Representation of a binning data structure.

**Methods**

- **`add(item, pt)`**
  - Adds an item to the grid at a specified location.

- **`bounds(bounds)`**
  - Returns a list of items found in the grid within the bounds specified.
### Table 51 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_grid(loc)</td>
<td>Returns whether a 2-tuple location <em>loc</em> lies inside the grid bounds.</td>
</tr>
<tr>
<td>nearest(pt)</td>
<td>Returns the nearest item to a point.</td>
</tr>
<tr>
<td>proximity(pt, r)</td>
<td>Returns a list of items found in the grid within a specified distance of a point.</td>
</tr>
<tr>
<td>remove(item, pt)</td>
<td>Removes an item from the grid at a specified location.</td>
</tr>
</tbody>
</table>

#### __init__(bounds, resolution)

Returns a grid with specified properties.

**__init__**(Rectangle, number) -> Grid

**Parameters**

- **bounds** [the area for the grid to encompass]
- **resolution** [the diameter of each bin]

#### Examples

TODO: complete this doctest >>> g = Grid(Rectangle(0, 0, 10, 10), 1)

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(bounds, resolution)</td>
<td>Returns a grid with specified properties.</td>
</tr>
<tr>
<td>add(item, pt)</td>
<td>Adds an item to the grid at a specified location.</td>
</tr>
<tr>
<td>bounds(bounds)</td>
<td>Returns a list of items found in the grid within the bounds specified.</td>
</tr>
<tr>
<td>in_grid(loc)</td>
<td>Returns whether a 2-tuple location <em>loc</em> lies inside the grid bounds.</td>
</tr>
<tr>
<td>nearest(pt)</td>
<td>Returns the nearest item to a point.</td>
</tr>
<tr>
<td>proximity(pt, r)</td>
<td>Returns a list of items found in the grid within a specified distance of a point.</td>
</tr>
<tr>
<td>remove(item, pt)</td>
<td>Removes an item from the grid at a specified location.</td>
</tr>
</tbody>
</table>

#### pysal.lib.cg.PointLocator

**class** pysal.lib.cg.PointLocator(points)

An abstract representation of a point indexing data structure.

#### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nearest(query_point)</td>
<td>Returns the nearest point indexed to a query point.</td>
</tr>
<tr>
<td>overlapping(region_rect)</td>
<td>Returns the indexed points located inside a rectangular query region.</td>
</tr>
<tr>
<td>polygon(polygon)</td>
<td>Returns the indexed points located inside a polygon</td>
</tr>
</tbody>
</table>

Continued on next page
proximity(origin, r) Returns the indexed points located within some distance of an origin point.

region(region_rect) Returns the indexed points located inside a rectangular query region.

__init__(points)
Returs a point locator object.
__init__(Point list) -> PointLocator

Parameters
  points [a list of points to index]

Examples

```python
>>> points = [Point((0, 0)), Point((1, 6)), Point((5.4, 1.4))]
>>> pl = PointLocator(points)
```

Methods

__init__(points) Returns a point locator object.
nearest(query_point) Returns the nearest point indexed to a query point.
overlapping(region_rect) Returns the indexed points located inside a rectangular query region.
polygon(polygon) Returns the indexed points located inside a polygon
proximity(origin, r) Returns the indexed points located within some distance of an origin point.
region(region_rect) Returns the indexed points located inside a rectangular query region.

pysal.lib.cg.PolygonLocator

class pysal.lib.cg.PolygonLocator (polygons)
An abstract representation of a polygon indexing data structure.

Methods

contains_point(point) Returns polygons that contain point
inside(query_rectangle) Returns polygons that are inside query_rectangle
nearest(query_point[, rule]) Returns the nearest polygon indexed to a query point based on various rules.
overlapping(query_rectangle) Returns list of polygons that overlap query_rectangle
proximity(origin, r[, rule]) Returns the indexed polygons located within some distance of an origin point based on various rules.
region(region_rect) Returns the indexed polygons located inside a rectangular query region.
__init__(polygons)

Returns a polygon locator object.

__init__(Polygon list) -> PolygonLocator

Parameters

polygons [a list of polygons to index]

Examples

```python
>>> p1 = Polygon([Point((0, 1)), Point((4, 5)), Point((5, 1))])
>>> p2 = Polygon([Point((3, 9)), Point((6, 7)), Point((1, 1))])
>>> pl = PolygonLocator([p1, p2])
>>> isinstance(pl, PolygonLocator)
True
```

Methods

__init__(polygons) Returns a polygon locator object.
contains_point(point) Returns polygons that contain point
inside(query_rectangle) Returns polygons that are inside query_rectangle
nearest(query_point[, rule]) Returns the nearest polygon indexed to a query point based on various rules.
overlapping(query_rectangle) Returns list of polygons that overlap query_rectangle
proximity(origin, r[, rule]) Returns the indexed polygons located within some distance of an origin point based on various rules.
region(region_rect) Returns the indexed polygons located inside a rectangular query region.

kdtree

cg.KDTree(data[, leafsize, distance_metric, ...]) kd-tree built on top of kd-tree functionality in scipy.

pysal.lib.cg.KDTree

pysal.lib.cg.KDTree (data, leafsize=10, distance_metric='Euclidean', radius=6371.0)

kd-tree built on top of kd-tree functionality in scipy. If using scipy 0.12 or greater uses the scipy.spatial.cKDTree, otherwise uses scipy.spatial.KDTree. Offers both Arc distance and Euclidean distance. Note that Arc distance is only appropriate when points in latitude and longitude, and the radius set to meaningful value (see docs below).

Parameters

data [array] The data points to be indexed. This array is not copied, and so modifying this data will result in bogus results. Typically nx2.

leafsize [int] The number of points at which the algorithm switches over to brute-force. Has to be positive. Optional, default is 10.

distance_metric [string] Options: “Euclidean” (default) and “Arc”.

radius [float] Radius of the sphere on which to compute distances. Assumes data in latitude and longitude. Ignored if distance_metric="Euclidean". Typical values:
pysal Documentation, Release 2.0.0

pysal.cg.RADIUS_EARTH_KM (default) pysal.cg.RADIUS_EARTH_MILES

io

<table>
<thead>
<tr>
<th><strong>init</strong></th>
<th>pysal.lib.io.fileio.FileIO(dataPath, mode, dataFormat)</th>
<th>Metaclass for supporting spatial data file read and write</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias of</td>
<td>pysal.lib.io.fileio.FileIO</td>
<td>pysal.lib.io.open</td>
</tr>
<tr>
<td>alias of</td>
<td>pysal.lib.io.fileio.FileIO</td>
<td>pysal.lib.io.fileio.FileIO</td>
</tr>
<tr>
<td>class pysal.lib.io.fileio.FileIO</td>
<td>pysal.lib.io.fileio.FileIO(dataPath='&quot;, mode='r', dataFormat=None)</td>
<td>Metaclass for supporting spatial data file read and write</td>
</tr>
<tr>
<td>How this works: FileIO.open(*args) == FileIO(*args) When creating a new instance of FileIO the <strong>new</strong> method intercepts <strong>new</strong> parses the filename to determine the fileType next, __registry and checked for that type. Each type supports one or more modes ['r','w','a',etc] If we support the type and mode, an instance of the appropriate handler is created and returned. All handlers must inherit from this class, and by doing so are automatically added to the __registry and are forced to conform to the prescribed API. The metaclass takes cares of the registration by parsing the class definition. It doesn’t make much sense to treat weights in the same way as shapefiles and dbfs, ....for now we’ll just return an instance of W on mode='r' .... on mode='w'. .write will expect an instance of W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>by_row</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ids</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rIds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cast(key, typ)</td>
<td>cast key as typ</td>
<td></td>
</tr>
<tr>
<td>check()</td>
<td>Prints the contents of the registry</td>
<td></td>
</tr>
<tr>
<td>close()</td>
<td>subclasses should clean themselves up and then call this method</td>
<td></td>
</tr>
<tr>
<td>get(n)</td>
<td>Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset</td>
<td></td>
</tr>
<tr>
<td>getType(dataPath, mode[, dataFormat])</td>
<td>Parse the dataPath and return the data type</td>
<td></td>
</tr>
<tr>
<td>open(*args, **kwargs)</td>
<td>Alias for FileIO()</td>
<td></td>
</tr>
<tr>
<td>read([n])</td>
<td>Read at most n objects, less if read hits EOF if size is negative or omitted read all objects until EOF returns None if EOF is reached before any objects.</td>
<td></td>
</tr>
<tr>
<td>seek(n)</td>
<td>Seek the FileObj to the beginning of the n'th record, if .ids are set, seeks to the beginning of the record at id, n</td>
<td></td>
</tr>
<tr>
<td>tell()</td>
<td>Return id (or offset) of next object</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 59 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>truncate([size])</td>
<td>Should be implemented by subclasses and redefine this doc string</td>
</tr>
<tr>
<td>write(obj)</td>
<td>Must be implemented by subclasses that support ‘w’ subclasses should increment .pos subclasses should also check if obj is an instance of type(list) and redefine this doc string</td>
</tr>
</tbody>
</table>

```
__init__(dataPath=", mode='r', dataFormat=None)
Initialize self. See help(type(self)) for accurate signature.
```

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>([dataPath, mode, dataFormat])</td>
<td>Initialize self.</td>
</tr>
<tr>
<td>cast(key, typ)</td>
<td>cast key as typ</td>
</tr>
<tr>
<td>check()</td>
<td>Prints the contents of the registry</td>
</tr>
<tr>
<td>close()</td>
<td>subclasses should clean themselves up and then call this method</td>
</tr>
<tr>
<td>flush()</td>
<td></td>
</tr>
<tr>
<td>get(n)</td>
<td>Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset</td>
</tr>
<tr>
<td>getType(dataPath, mode[, dataFormat])</td>
<td>Parse the dataPath and return the data type</td>
</tr>
<tr>
<td>open(*args, **kwargs)</td>
<td>Alias for FileIO()</td>
</tr>
<tr>
<td>read([n])</td>
<td>Read at most n objects, less if read hits EOF if size is negative or omitted read all objects until EOF returns None if EOF is reached before any objects.</td>
</tr>
<tr>
<td>seek(n)</td>
<td>Seek the FileObj to the beginning of the n’th record, if .ids are set, seeks to the beginning of the record at id, n</td>
</tr>
<tr>
<td>tell()</td>
<td>Return id (or offset) of next object</td>
</tr>
<tr>
<td>truncate([size])</td>
<td>Should be implemented by subclasses and redefine this doc string</td>
</tr>
<tr>
<td>write(obj)</td>
<td>Must be implemented by subclasses that support ‘w’ subclasses should increment .pos subclasses should also check if obj is an instance of type(list) and redefine this doc string</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>by_row</td>
<td></td>
</tr>
<tr>
<td>ids</td>
<td></td>
</tr>
<tr>
<td>rIds</td>
<td></td>
</tr>
</tbody>
</table>

### examples
1.2.2 `pysal.explore`: Exploratory spatial data analysis

### `pysal.explore.esda`: Spatial Autocorrelation Analysis

### Gamma Statistic

```python
esda.Gamma(y, w[, operation, standardize, ...])
```
Gamma index for spatial autocorrelation

**Parameters**

- `y` [array] variable measured across n spatial units
- `w` [W] spatial weights instance can be binary or row-standardized
- `operation` [({'c', 's', 'a'})] attribute similarity function where, ‘c’ cross product ‘s’ squared difference ‘a’ absolute difference
- `standardize` [({'no', 'yes'})] standardize variables first ‘no’ keep as is ‘yes’ or ‘y’ standardize to mean zero and variance one
- `permutations` [int] number of random permutations for calculation of pseudo-p_values
Notes

For further technical details see [HGC81].

Examples

use same example as for join counts to show similarity

```python
>>> import pysal.lib, numpy as np
>>> from pysal.explore.esda.gamma import Gamma
>>> w = pysal.lib.weights.lat2W(4,4)
>>> y=np.ones(16)
>>> y[0:8]=0
>>> np.random.seed(12345)
>>> g = Gamma(y,w)
>>> g.g
20.0
>>> round(g.g_z, 3)
3.188
>>> round(g.p_sim_g, 3)
0.003
>>> g.min_g
0.0
>>> g.max_g
20.0
>>> g.mean_g
11.093093093093094
>>> np.random.seed(12345)
>>> g1 = Gamma(y,w,operation='s')
>>> g1.g
8.0
>>> round(g1.g_z, 3)
-3.706
>>> g1.p_sim_g
0.001
>>> g1.min_g
14.0
>>> g1.max_g
48.0
>>> g1.mean_g
25.623623623623622
>>> np.random.seed(12345)
>>> g2 = Gamma(y,w,operation='a')
>>> g2.g
8.0
>>> round(g2.g_z, 3)
-3.706
>>> g2.p_sim_g
0.001
>>> g2.min_g
14.0
>>> g2.max_g
48.0
>>> g2.mean_g
25.623623623623622
>>> np.random.seed(12345)
>>> g3 = Gamma(y,w,standardize='y')
```

(continues on next page)
Attributes

- **y** [array] original variable
- **w** [W] original w object
- **op** [{'c', 's', 'a'}] attribute similarity function, as per parameters attribute similarity function
- **stand** [{'no', 'yes'}] standardization
- **permutations** [int] number of permutations
- **gamma** [float] value of Gamma index
- **sim_g** [array] (if permutations>0) vector of Gamma index values for permuted samples
- **p_sim_g** [array] (if permutations>0) p-value based on permutations (one-sided) null: spatial randomness alternative: the observed Gamma is more extreme than under randomness implemented as a two-sided test
- **mean_g** [float] average of permuted Gamma values
- **min_g** [float] minimum of permuted Gamma values
- **max_g** [float] maximum of permuted Gamma values

Methods

```python
by_col
```

```python
__init__(y, w, operation='c', standardize='no', permutations=999)
```

Initialize self. See help(type(self)) for accurate signature.
Methods

__init__(y, [operation, standardize, ...]) Initialize self.
by_col(df, cols[, w, inplace, pvalue, outvals])

Attributes

p_sim new name to fit with Moran module

Geary Statistic

esda.Geary(y, w[, transformation, permutations]) Global Geary C Autocorrelation statistic

pysal.explore.esda.Geary

class pysal.explore.esda.Geary(y, w, transformation='r', permutations=999) Global Geary C Autocorrelation statistic

Parameters

y [array] (n, 1) attribute vector
w [W] spatial weights
permutations [int] number of random permutations for calculation of pseudo-p_values

Notes

Technical details and derivations can be found in [CO81].

Examples

```python
>>> import pysal.lib
>>> from pysal.explore.esda.geary import Geary
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("book.gal")).read()
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("book.txt"))
>>> y = np.array(f.by_col['y'])
>>> c = Geary(y,w,permutations=0)
>>> round(c.C,7)
0.3330108
>>> round(c.p_norm,7)
9.2e-05
```
Spatial weights

permutations [int] number of permutations
C [float] value of statistic
EC [float] expected value
VC [float] variance of G under normality assumption
z_norm [float] z-statistic for C under normality assumption
z_rand [float] z-statistic for C under randomization assumption
p_norm [float] p-value under normality assumption (one-tailed)
p_rand [float] p-value under randomization assumption (one-tailed)
sim [array] (if permutations!=0) vector of I values for permutated samples
p_sim [float] (if permutations!=0) p-value based on permutations (one-tailed)
null: spatial randomness alternative: the observed C is extreme it is either extremely high or extremely low
EC_sim [float] (if permutations!=0) average value of C from permutations
VC_sim [float] (if permutations!=0) variance of C from permutations
seC_sim [float] (if permutations!=0) standard deviation of C under permutations.
z_sim [float] (if permutations!=0) standardized C based on permutations
p_z_sim [float] (if permutations!=0) p-value based on standard normal approximation from permutations

Methods

by_col(df, cols[, w, inplace, pvalue, outvals]) Function to compute a Geary statistic on a dataframe

__init__ (y, w[, transformation='r', permutations=999])
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (y, w[, transformation, permutations])
by_col(df, cols[, w, inplace, pvalue, outvals])

Getis-Ord Statistics

esda.G(y, w[, permutations]) Global G Autocorrelation Statistic
esda.G_Local(y, w[, transform, ...]) Generalized Local G Autocorrelation

pysal.explore.esda.G

class pysal.explore.esda.G (y, w[, permutations=999])
  Global G Autocorrelation Statistic

Parameters
y [array (n,1)] Attribute values
w [W] DistanceBand W spatial weights based on distance band
permutations [int] the number of random permutations for calculating pseudo p_values

Notes
Moments are based on normality assumption.
For technical details see [GO10] and [OG10].

Examples
```python
>>> import pysal.lib
>>> import numpy
>>> numpy.random.seed(10)

Preparing a point data set
>>> points = [(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]

Creating a weights object from points
>>> w = pysal.lib.weights.DistanceBand(points, threshold=15)
>>> w.transform = “B”

Preparing a variable
>>> y = numpy.array([2, 3, 3.2, 5, 8, 7])

Applying Getis and Ord G test
>>> from pysal.explore.esda.getisord import G
>>> g = G(y, w)

Examining the results
>>> round(g.G, 3)
0.557
>>> round(g.p_norm, 3)
0.173
```

Attributes
- y [array] original variable
- w [W] DistanceBand W spatial weights based on distance band
- permutations [int] the number of permutations
- G [float] the value of statistic
- EG [float] the expected value of statistic
- VG [float] the variance of G under normality assumption
- z_norm [float] standard normal test statistic
- p_norm [float] p-value under normality assumption (one-sided)
- sim [array] (if permutations > 0) vector of G values for permutated samples
- p_sim [float] p-value based on permutations (one-sided) null: spatial randomness alternative: the observed G is extreme it is either extremely high or extremely low
- EG_sim [float] average value of G from permutations
- VG_sim [float] variance of G from permutations
- seG_sim [float] standard deviation of G under permutations.
- z_sim [float] standardized G based on permutations
pysal Documentation, Release 2.0.0

**p_z_sim** [float] p-value based on standard normal approximation from permutations (one-sided)

**Methods**

```python
by_col(df, cols[, w, inplace, pvalue, outvals]) Function to compute a G statistic on a dataframe
```

```python
__init__(y, w, permutations=999)
Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

```python
__init__(y, w[, permutations]) Initialize self.
by_col(df, cols[, w, inplace, pvalue, outvals]) Function to compute a G statistic on a dataframe
```

**pysal.explore.esda.G_Local**

```python
class pysal.explore.esda.G_Local(y, w, transform='R', permutations=999, star=False)
Generalized Local G Autocorrelation
```

**Parameters**

- **y** [array] variable
- **w** [W] DistanceBand, weights instance that is based on threshold distance and is assumed to be aligned with y
- **transform** [{‘R’, ‘B’}] the type of w, either ‘B’ (binary) or ‘R’ (row-standardized)
- **permutations** [int] the number of random permutations for calculating pseudo p values
- **star** [boolean] whether or not to include focal observation in sums (default: False)

**Notes**

To compute moments of Gs under normality assumption, PySAL considers w is either binary or row-standardized. For binary weights object, the weight value for self is 1. For row-standardized weights object, the weight value for self is 1/(the number of its neighbors + 1).

For technical details see [GO10] and [OG10].

**Examples**

```python
>>> import pysal.lib
>>> import numpy
>>> numpy.random.seed(10)
Preparing a point data set
```  
```python
>>> points = [(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
Creating a weights object from points
```
```python
>>> w = pysal.lib.weights.DistanceBand(points, threshold=15)

Preparing a variable

```n
g
```n
```n
numpy.array([2, 3, 3.2, 5, 8, 7])
```n
```n
Applying Getis and Ord local G test using a binary weights object
```n
```n
from pysal.explore.esda.getisord import G_Local
```n
```n
lg = G_Local(y, w, transform='B')

Examining the results

```n
```n
lg.Zs
```n
array([-1.0136729 , -0.04361589, 1.31558703, -0.31412676, 1.15373986,
       1.77833941])
```n
```n
round(lg.p_sim[0], 3)
```n
0.101

```n
```n
numpy.random.seed(10)

Applying Getis and Ord local G* test using a binary weights object

```n
```n
lg_star = G_Local(y, w, transform='B', star=True)

Examining the results

```n
```n
lg_star.Zs
```n
array([-1.39727626, -0.28917762, 0.65064964, -0.28917762, 1.23452088,
       2.02424331])
```n
```n
round(lg_star.p_sim[0], 3)
```n
0.101

```n
```n
numpy.random.seed(12345)

Applying Getis and Ord local G test using a row-standardized weights object

```n
```n
lg = G_Local(y, w, transform='R')

Examining the results

```n
```n
lg.Zs
```n
array([-0.62074534, -0.01780611, 1.31558703, -0.12824171, 0.28843496,
       1.77833941])
```n
```n
round(lg.p_sim[0], 3)
```n
0.103

```n
```n
numpy.random.seed(10)

Applying Getis and Ord local G* test using a row-standardized weights object

```n
```n
lg_star = G_Local(y, w, transform='R', star=True)

Examining the results

```n
```n
lg_star.Zs
```n
array([-0.62488094, -0.09144599, 0.41150696, -0.09144599, 0.24690418,
       1.28024388])
```n
```n
round(lg_star.p_sim[0], 3)
```n
0.101

Attributes
y [array] original variable
w [array] original weights object
permutations [int] the number of permutations
Gs [array] of floats, the value of the original G statistic in Getis & Ord (1992)
EGs [float] expected value of Gs under normality assumption the values is scalar, since the expectation is identical across all observations
VGs [array] of floats, variance values of Gs under normality assumption
Zs [array] of floats, standardized Gs
p_norm [array] of floats, p-value under normality assumption (one-sided) for two-sided tests, this value should be multiplied by 2
sim [array] of arrays of floats (if permutations>0), vector of I values for permuted samples
p_sim [array] of floats, p-value based on permutations (one-sided) null - spatial randomness alternative - the observed G is extreme it is either extremely high or extremely low
EG_sim [array] of floats, average value of G from permutations
VG_sim [array] of floats, variance of G from permutations
seG_sim [array] of floats, standard deviation of G under permutations.
z_sim [array] of floats, standardized G based on permutations
p_z_sim [array] of floats, p-value based on standard normal approximation from permutations (one-sided)

Methods

by_col(df, cols[, w, inplace, pvalue, outvals]) Function to compute a G_Local statistic on a dataframe

__init__(y, w[, transform='R', permutations=999, star=False])
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, w[, transform, permutations, star]) Initialize self.
by_col(df, cols[, w, inplace, pvalue, outvals]) Function to compute a G_Local statistic on a dataframe
calc()

Join Count Statistics

esda.Join_Counts(y, w[, permutations]) Binary Join Counts
pysal.explore.esda.Join_Counts

**class pysal.explore.esda.Join_Counts**(y, w, permutations=999)

Binary Join Counts

**Parameters**

- **y** [array] binary variable measured across n spatial units
- **w** [W] spatial weights instance
- **permutations** [int] number of random permutations for calculation of pseudo-p_values

**Notes**

Technical details and derivations can be found in [CO81].

**Examples**

Replicate example from anselin and rey

```python
>>> import numpy as np
>>> import pysal.lib
>>> w = pysal.lib.weights.lat2W(4, 4)
>>> y = np.ones(16)
>>> y[0:8] = 0
>>> np.random.seed(12345)
>>> from pysal.explore.esda.join_counts import Join_Counts
>>> jc = Join_Counts(y, w)
>>> jc.bb
10.0
>>> jc.bw
4.0
>>> jc.ww
10.0
>>> jc.J
24.0
>>> len(jc.sim_bb)
999
>>> round(jc.p_sim_bb, 3)
0.003
>>> round(np.mean(jc.sim_bb), 3)
5.547
>>> np.max(jc.sim_bb)
10.0
>>> np.min(jc.sim_bb)
0.0
>>> len(jc.sim_bw)
999
>>> jc.p_sim_bw
1.0
>>> np.mean(jc.sim_bw)
12.811811811811811
>>> np.max(jc.sim_bw)
24.0
>>> np.min(jc.sim_bw)
```
Attributes

y [array] original variable
w [W] original w object
permutations [int] number of permutations
bb [float] number of black-black joins
ww [float] number of white-white joins
bw [float] number of black-white joins
J [float] number of joins
sim_bb [array] (if permutations>0) vector of bb values for permuted samples
p_sim_bb [array]
   (if permutations>0) p-value based on permutations (one-sided) null: spatial randomness
   alternative: the observed bb is greater than under randomness
mean_bb [float] average of permuted bb values
min_bb [float] minimum of permuted bb values
max_bb [float] maximum of permuted bb values
sim_bw [array] (if permutations>0) vector of bw values for permuted samples
p_sim_bw [array] (if permutations>0) p-value based on permutations (one-sided) null: spatial randomness
   alternative: the observed bw is greater than under randomness
mean_bw [float] average of permuted bw values
min_bw [float] minimum of permuted bw values
max_bw [float] maximum of permuted bw values

Methods

by_col(df, cols[, w, inplace, pvalue, outvals]) Function to compute a Join_Count statistic on a dataframe

__init__(y, w[, permutations]) Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, w[, permutations]) Initialize self.
by_col(df, cols[, w, inplace, pvalue, outvals]) Function to compute a Join_Count statistic on a dataframe
Moran Statistics

```
esda.Moran(y, w[, transformation, ...])  Moran’s I Global Autocorrelation Statistic
esda.Moran_BV(x, y, w[, ...])             Bivariate Moran’s I
esda.Moran_BV_matrix(variables, w[,...]) Bivariate Moran Matrix
esda.Moran_Local(y, w[, ...])            Local Moran Statistics
esda.Moran_Local_BV(x, y, w[,...])      Bivariate Local Moran Statistics
esda.Moran_Rate(e, b, w[, adjusted, ...]) Adjusted Moran’s I Global Autocorrelation Statistic for Rate Variables [AR99]
esda.Moran_Local_Rate(e, b, w[, adjusted, ...]) Adjusted Local Moran Statistics for Rate Variables [As-suncao1999]
```

```
pysal.explore.esda.Moran

class pysal.explore.esda.Moran(y, w, transformation='r', permutations=999, two_tailed=True)
    Moran’s I Global Autocorrelation Statistic

Parameters

y  [array] variable measured across n spatial units
w  [W] spatial weights instance

transformation  [string] weights transformation, default is row-standardized “r”. Other options
    variance-stabilizing.

permutations  [int] number of random permutations for calculation of pseudo-p_values

two_tailed  [boolean] If True (default) analytical p-values for Moran are two tailed, otherwise
    if False, they are one-tailed.

Notes

Technical details and derivations can be found in [CO81].

Examples

```
>>> import pysal.lib
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("stl.gal")) .read()
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("stl_hom.txt"))
>>> y = np.array(f.by_col['HR8893'])
>>> from pysal.explore.esda.moran import Moran
>>> mi = Moran(y, w)
>>> round(mi.I, 3)
0.244
>>> mi.EI
-0.012987012987012988
>>> mi.p_norm
0.0002714786277093761
>>> round(mi.I, 3)
0.244
>>> mi.EI
-0.012987012987012988
>>> mi.p_norm
0.0002714786277093761
```

SIDS example replicating OpenGeoda >>> w = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.gal")) .read()
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.dbf")) >>> SIDR = np.array(f.by_col("SIDR74"))
>>> mi = Moran(SIDR, w) >>> round(mi.I, 3) 0.248 >>> mi.p_norm 0.0001158330781489969
One-tailed

```python
>>> mi_1 = Moran(SIDR, w, two_tailed=False)
>>> round(mi_1.I, 3)
0.248
>>> round(mi_1.p_norm, 4)
0.0001
```

Attributes

- `y` [array] original variable
- `w` [W] original w object
- `permutations` [int] number of permutations
- `I` [float] value of Moran’s I
- `EI` [float] expected value under normality assumption
- `VI_norm` [float] variance of I under normality assumption
- `sel_norm` [float] standard deviation of I under normality assumption
- `z_norm` [float] z-value of I under normality assumption
- `p_norm` [float] p-value of I under normality assumption
- `VI_rand` [float] variance of I under randomization assumption
- `sel_rand` [float] standard deviation of I under randomization assumption
- `z_rand` [float] z-value of I under randomization assumption
- `p_rand` [float] p-value of I under randomization assumption
- `two_tailed` [boolean] If True `p_norm` and `p_rand` are two-tailed, otherwise they are one-tailed.
- `sim` [array] (if permutations>0) vector of I values for permuted samples
- `p_sim` [array] (if permutations>0) p-value based on permutations (one-tailed) null: spatial randomness alternative: the observed I is extreme if it is either extremely greater or extremely lower than the values obtained based on permutations
- `EI_sim` [float] (if permutations>0) average value of I from permutations
- `VI_sim` [float] (if permutations>0) variance of I from permutations
- `sel_sim` [float] (if permutations>0) standard deviation of I under permutations.
- `z_sim` [float] (if permutations>0) standardized I based on permutations
- `p_z_sim` [float] (if permutations>0) p-value based on standard normal approximation from permutations

Methods

- `by_col(df, cols[, w, inplace, pvalue, outvals])` Function to compute a Moran statistic on a dataframe

- `__init__(y, w, transformation= 'r', permutations=999, two_tailed=True)`
  Initialize self. See help(type(self)) for accurate signature.
Methods

```python
__init__(y, w[, transformation, ...])
```
Initializes self.

```python
by_col(df, cols[, w, inplace, pvalue, outvals])
```
Function to compute a Moran statistic on a dataframe

```python
pysal.explore.esda.Moran_BV
```

```python
class pysal.explore.esda.Moran_BV(x, y, w, transformation='r', permutations=999)
```
Bivariate Moran’s I

### Parameters
- **x** [array] x-axis variable
- **y** [array] yw will be on y axis
- **w** [W] weight instance assumed to be aligned with y
- **permutations** [int] number of random permutations for calculation of pseudo p_values

### Notes
Inference is only based on permutations as analytical results are not too reliable.

### Examples
```python
>>> import pysal.lib
>>> import numpy as np

Set random number generator seed so we can replicate the example
```n
```python
>>> np.random.seed(10)
```

Open the sudden infant death dbf file and read in rates for 74 and 79 converting each to a numpy array
```python
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.dbf"))
>>> SIDR74 = np.array(f.by_col['SIDR74'])
>>> SIDR79 = np.array(f.by_col['SIDR79'])
```

Read a GAL file and construct our spatial weights object
```python
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.gal"))
```

Create an instance of Moran_BV >>> from pysal.explore.esda.moran import Moran_BV >>> mbi = Moran_BV(SIDR79, SIDR74, w)

What is the bivariate Moran’s I value
```python
>>> round(mbi.I, 3)
0.156
```

Based on 999 permutations, what is the p-value of our statistic
Attributes

- **zx** [array] original x variable standardized by mean and std
- **zy** [array] original y variable standardized by mean and std
- **w** [W] original w object
- **permutation** [int] number of permutations
- **I** [float] value of bivariate Moran’s I
- **sim** [array] (if permutations>0) vector of I values for permuted samples
- **p_sim** [float] (if permutations>0) p-value based on permutations (one-sided) null: spatial randomness alternative: the observed I is extreme it is either extremely high or extremely low
- **EI_sim** [array] (if permutations>0) average value of I from permutations
- **VI_sim** [array] (if permutations>0) variance of I from permutations
- **seI_sim** [array] (if permutations>0) standard deviation of I under permutations.
- **z_sim** [array] (if permutations>0) standardized I based on permutations
- **p_z_sim** [float] (if permutations>0) p-value based on standard normal approximation from permutations

Methods

- **by_col(df, x[, y, w, inplace, pvalue, outvals])** Function to compute a Moran_BV statistic on a dataframe

- **__init__(x, y, w[, transformation, permutations])** Initialize self. See help(type(self)) for accurate signature.

Methods

- **__init__(x, y, w[, transformation, permutations])** Initialize self.
- **by_col(df, x[, y, w, inplace, pvalue, outvals])** Function to compute a Moran_BV statistic on a dataframe

**pysal.explore.esda.Moran_BV_matrix**

**pysal.explore.esda.Moran_BV_matrix**(variables, w, permutations=0, varnames=None)

Bivariate Moran Matrix

Calculates bivariate Moran between all pairs of a set of variables.

**Parameters**

- **variables** [array or pandas.DataFrame] sequence of variables to be assessed
- **w** [W] a spatial weights object
**permutations** [int] number of permutations

**varnames** [list, optional if variables is an array] Strings for variable names. Will add an attribute to Moran_BV objects in results needed for plotting in splot or .plot(). Default =None. Note: If variables is a pandas.DataFrame varnames will automatically be generated

**Returns**

**results** [dictionary] (i, j) is the key for the pair of variables, values are the Moran_BV objects.

### Examples

**open dbf**

```python
>>> import pysal.lib
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.dbf"))
```

**pull of selected variables from dbf and create numpy arrays for each**

```python
>>> varnames = ['SIDR74', 'SIDR79', 'NWR74', 'NWR79]
>>> vars = [np.array(f.by_col[var]) for var in varnames]
```

**create a contiguity matrix from an external gal file**

```python
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.gal").read()
```

**create an instance of Moran_BV_matrix**

```python
>>> from pysal.explore.esda.moran import Moran_BV_matrix
>>> res = Moran_BV_matrix(vars, w, varnames = varnames)
```

**check values**

```python
>>> round(res[(0, 1)].I, 7)
0.1936261
>>> round(res[(3, 0)].I, 7)
0.3770138
```

### pysal.explore.esda.Moran_Local

**class** pysal.explore.esda.Moran_Local(y, w, transformation='r', permutations=999, geoda_quads=False)

Local Moran Statistics

**Parameters**

- **y** [array] (n,1), attribute array
- **w** [W] weight instance assumed to be aligned with y
- **transformation** ["R", "B", "D", "U", "V"] 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
**geoda_quads** [boolean] (default=False) If True use GeoDa scheme: HH=1, LL=2, LH=3, HL=4 If False use PySAL Scheme: HH=1, LH=2, LL=3, HL=4

**Notes**

For technical details see [Ans95].

**Examples**

```python
>>> import pysal.lib
>>> import numpy as np
>>> np.random.seed(10)
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("desmith.gal")).read()
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("desmith.txt"))
>>> y = np.array(f.by_col["z"])
>>> from pysal.explore.esda.moran import Moran_Local

```

```python
>>> lm = Moran_Local(y, w, transformation = "r", permutations = 99)
```

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

```python
>>> lm.p_z_sim[0]
0.24669152541631179
```

```python
>>> lm = Moran_Local(y, w, transformation = "r", permutations = 99, geoda_quads=True)
```

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

Note random components result is slightly different values across architectures so the results have been removed from doctests and will be moved into unitests that are conditional on architectures

**Attributes**

- **y** [array] original variable
- **w** [W] original w object
- **permutations** [int] number of random permutations for calculation of pseudo p_values
- **Is** [array] local Moran’s I values
- **q** [array] (if permutations>0) values indicate quadranr location 1 HH, 2 LH, 3 LL, 4 HL
- **sim** [array (permutations by n)] (if permutations>0) I values for permuted samples
- **p_sim** [array (if permutations>0) p-values based on permutations (one-sided) null: spatial randomness alternative: the observed li is further away or extreme from the median of simulated values. It is either extremely high or extremely low in the distribution of simulated Is.
- **Els_sim** [array (if permutations>0) average values of local Is from permutations
- **Vls_sim** [array (if permutations>0) variance of Is from permutations
- **sels_sim** [array (if permutations>0) standard deviations of Is under permutations
- **z_sim** [array (if permutations>0) standardized Is based on permutations
- **p_z_sim** [array (if permutations>0) p-values based on standard normal approximation from permutations (one-sided) for two-sided tests, these values should be multiplied by 2
Methods

by_col(df, cols[, w, inplace, pvalue, outvals])

Function to compute a Moran_Local statistic on a dataframe

__init__(y, w[, transformation='r', permutations=999, geoda_quads=False])

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, w[, transformation, ...])

Initialize self.

by_col(df, cols[, w, inplace, pvalue, outvals])

Function to compute a Moran_Local statistic on a dataframe

calc(w, z)

pysal.explore.esda.Moran_Local_BV

class pysal.explore.esda.Moran_Local_BV(x, y, w, transformation='r', permutations=999, geoda_quads=False)

Bivariate Local Moran Statistics

Parameters

x [array] x-axis variable
y [array] (n,1), wy will be on y axis
w [W] weight instance assumed to be aligned with y
transformation ["R", "B", "D", "U", "V"] 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
geoda_quads [boolean] (default=False) If True use GeoDa scheme: HH=1, LL=2, LH=3, HL=4 If False use PySAL Scheme: HH=1, LH=2, LL=3, HL=4

Examples

```python
>>> import pysal.lib
>>> import numpy as np
>>> np.random.seed(10)
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.gal")).read()
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.dbf"))
>>> x = np.array(f.by_col['SIDR79'])
>>> y = np.array(f.by_col['SIDR74'])
>>> from pysal.explore.esda.moran import Moran_Local_BV
>>> lm = Moran_Local_BV(x, y, w, transformation = "r", permutations = 99)
```

(continues on next page)
>>> lm.q[:10]
array([3, 4, 3, 4, 2, 1, 4, 4, 2, 4])
>>> lm = Moran_Local_BV(x, y, w, transformation = "r", permutations = 99, geoda_quads=True)
>>> lm.q[:10]
array([2, 4, 2, 4, 3, 1, 4, 4, 3, 4])

Note random components result is slightly different values across architectures so the results have been removed from doctests and will be moved into unittests that are conditional on architectures

Attributes

- **zx**: [array] original x variable standardized by mean and std
- **zy**: [array] original y variable standardized by mean and std
- **w**: [W] original w object
- **permutations**: [int] number of random permutations for calculation of pseudo p_values
- **Is**: [float] value of Moran’s I
- **q**: [array] (if permutations>0) values indicate quadrat location 1 HH, 2 LH, 3 LL, 4 HL
- **sim**: [array] (if permutations>0) vector of I values for permuted samples
- **p_sim**: [array] (if permutations>0) p-value based on permutations (one-sided) null: spatial randomness alternative: the observed li is further away or extreme from the median of simulated values. It is either extremely high or extremely low in the distribution of simulated Is.
- **EI_sim**: [array] (if permutations>0) average values of local Is from permutations
- **VI_sim**: [array] (if permutations>0) variance of Is from permutations
- **sel_sim**: [array] (if permutations>0) standard deviations of Is under permutations.
- **z_sim**: [array] (if permutations>0) standardized Is based on permutations
- **p_z_sim**: [array] (if permutations>0) p-values based on standard normal approximation from permutations (one-sided) for two-sided tests, these values should be multiplied by 2

Methods

- **by_col**(df, x[, y, w, inplace, pvalue, outvals]) Function to compute a Moran_Local_BV statistic on a dataframe

- **calc**

```python
__init__**(x, y, w[, transformation, ...])
Initialize self. See help(type(self)) for accurate signature.
```

Methods

```python
__init__**(x, y, w[, transformation, ...])
Initialize self.
```
Table 86 – continued from previous page

| by_col(df, x[, y, w, inplace, pvalue, outvals]) | Function to compute a Moran_Local_BV statistic on a dataframe |
| calc(w, zx, zy) |

**pysal.explore.esda.Moran_Rate**

**class pysal.explore.esda.Moran_Rate**

Adjusted Moran’s I Global Autocorrelation Statistic for Rate Variables [AR99]

**Parameters**

- **e** [array] an event variable measured across n spatial units
- **b** [array] a population-at-risk variable measured across n spatial units
- **w** [W] spatial weights instance
- **adjusted** [boolean] whether or not Moran’s I needs to be adjusted for rate variable
- **two_tailed** [boolean] If True (default), analytical p-values for Moran’s I are two-tailed, otherwise they are one tailed.
- **permutations** [int] number of random permutations for calculation of pseudo p_values

**Examples**

```python
>>> import pysal.lib
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.gal")).read()
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.dbf"))
>>> e = np.array(f.by_col('SID79'))
>>> b = np.array(f.by_col('BIR79'))
>>> from pysal.explore.esda.moran import Moran_Rate
>>> mi = Moran_Rate(e, b, w, two_tailed=False)
>>> "\%6.4f" % mi.I
'0.1662'
>>> "\%6.4f" % mi.p_norm
'0.0042'
```

**Attributes**

- **y** [array] rate variable computed from parameters e and b if adjusted is True, y is standardized rates otherwise, y is raw rates
- **w** [W] original w object
- **permutations** [int] number of permutations
- **I** [float] value of Moran’s I
- **EI** [float] expected value under normality assumption
- **VI_norm** [float] variance of I under normality assumption
- **seI_norm** [float] standard deviation of I under normality assumption
**z_norm** [float] z-value of I under normality assumption

**p_norm** [float] p-value of I under normality assumption

**VI_rand** [float] variance of I under randomization assumption

**sel_rand** [float] standard deviation of I under randomization assumption

**z_rand** [float] z-value of I under randomization assumption

**p_rand** [float] p-value of I under randomization assumption

**two_tailed** [boolean] If True, p_norm and p_rand are two-tailed p-values, otherwise they are one-tailed.

**sim** [array] (if permutations>0) vector of I values for permuted samples

**p_sim** [array] (if permutations>0) p-value based on permutations (one-sided) null: spatial randomness alternative: the observed I is extreme if it is either extremely greater or extremely lower than the values obtained from permutations

**EI_sim** [float] (if permutations>0) average value of I from permutations

**VI_sim** [float] (if permutations>0) variance of I from permutations

**sel_sim** [float] (if permutations>0) standard deviation of I under permutations.

**z_sim** [float] (if permutations>0) standardized I based on permutations

**p_z_sim** [float] (if permutations>0) p-value based on standard normal approximation from

---

### Methods

**by_col** (df, events, populations[, w, ...]) Function to compute a Moran_Rate statistic on a dataframe

```
__init__(e, b, w[, adjusted, ...]) Initialize self. See help(type(self)) for accurate signature.
```

---

**pysal.explore.esda.Moran_Local_Rate**

**class pysal.explore.esda.Moran_Local_Rate** *(e, b, w, adjusted=True, transformation='r', permutations=999, two_tailed=True)*

Adjusted Local Moran Statistics for Rate Variables [Assuncao1999]

**Parameters**

- e [array] (n,1), an event variable across n spatial units
- b [array] (n,1), a population-at-risk variable across n spatial units
- w [W] weight instance assumed to be aligned with y
adjusted [boolean] whether or not local Moran statistics need to be adjusted for rate variable


permutations [int] number of random permutations for calculation of pseudo p_values

geoda_quads [boolean] (default=False) If True use GeoDa scheme: HH=1, LL=2, LH=3, HL=4 If False use PySAL Scheme: HH=1, LH=2, LL=3, HL=4

Attributes
———-

y [array] rate variables computed from parameters e and b if adjusted is True, y is standardized rates otherwise, y is raw rates

w [W] original w object

permutations [int] number of random permutations for calculation of pseudo p_values

I [float] value of Moran’s I

q [array] (if permutations>0) values indicate quadrant location 1 HH, 2 LH, 3 LL, 4 HL

sim [array] (if permutations>0) vector of I values for permuted samples

p_sim [array] (if permutations>0) p-value based on permutations (one-sided) null: spatial randomness alternative: the observed Ii is further away or extreme from the median of simulated Is. It is either extremely high or extremely low in the distribution of simulated Is

EI_sim [float] (if permutations>0) average value of I from permutations

VI_sim [float] (if permutations>0) variance of I from permutations

sel_sim [float] (if permutations>0) standard deviation of I under permutations.

z_sim [float] (if permutations>0) standardized I based on permutations

p_z_sim [float] (if permutations>0) p-value based on standard normal approximation from permutations (one-sided) for two-sided tests, these values should be multiplied by 2

Examples

>>> import pysal.lib
>>> import numpy as np
>>> np.random.seed(10)
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.gal")).read()
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("sids2.dbf"))
>>> e = np.array(f.by_col('SID79'))
>>> b = np.array(f.by_col('BIR79'))
>>> from pysal.explore.esda.moran import Moran_Local_Rate
>>> lm = Moran_Local_Rate(e, b, w, transformation = "r", permutations = 99)
>>> lm.q[:10]
array([2, 4, 3, 1, 2, 1, 1, 4, 2, 4])
>>> lm = Moran_Local_Rate(e, b, w, transformation = "r", permutations = 99, geoda_quads=True)
>>> lm.q[:10]
array([3, 4, 2, 1, 3, 1, 1, 4, 3, 4])

Note random components result is slightly different values across architectures so the results have been removed from doctests and will be moved into unitests that are conditional on architectures.
Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>by_col(df, events, populations[, w, ...])</code></td>
<td>Function to compute a Moran_Local_Rate statistic on a dataframe</td>
</tr>
</tbody>
</table>

__init__ (e, b, w, adjusted=True, transformation='r', permutations=999, geoda_quads=False)
Initialize self. See help(type(self)) for accurate signature.

Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong> (e, b, w[, adjusted, ...])</td>
<td>Initialize self</td>
</tr>
<tr>
<td><code>by_col(df, events, populations[, w, ...])</code></td>
<td>Function to compute a Moran_Local_Rate statistic on a dataframe</td>
</tr>
<tr>
<td><code>calc(w, z)</code></td>
<td></td>
</tr>
</tbody>
</table>

### pysal.explore.giddy: Geospatial Distribution Dynamics

#### Markov Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>giddy.markov.Markov(class_ids[, classes])</code></td>
<td>Classic Markov transition matrices.</td>
</tr>
<tr>
<td><code>giddy.markov.Spatial_Markov(y, w[, k, m, ...])</code></td>
<td>Markov transitions conditioned on the value of the spatial lag.</td>
</tr>
<tr>
<td><code>giddy.markov.LISA_Markov(y, w[, ...])</code></td>
<td>Markov for Local Indicators of Spatial Association</td>
</tr>
<tr>
<td><code>giddy.markov.kullback(F)</code></td>
<td>Kullback information based test of Markov Homogeneity.</td>
</tr>
<tr>
<td><code>giddy.markov.prais(pmat)</code></td>
<td>Prais conditional mobility measure.</td>
</tr>
<tr>
<td><code>giddy.markov.homogeneity(transition_matrices)</code></td>
<td>Test for homogeneity of Markov transition probabilities across regimes.</td>
</tr>
<tr>
<td><code>giddy.ergodic.steady_state(P)</code></td>
<td>Calculates the steady state probability vector for a regular Markov transition matrix P.</td>
</tr>
<tr>
<td><code>giddy.ergodic.fmpt(P)</code></td>
<td>Calculates the matrix of first mean passage times for an ergodic transition probability matrix.</td>
</tr>
<tr>
<td><code>giddy.ergodic.var_fmpt(P)</code></td>
<td>Variances of first mean passage times for an ergodic transition probability matrix.</td>
</tr>
</tbody>
</table>

### pysal.explore.giddy.markov.Markov

#### class pysal.explore.giddy.markov.Markov (class_ids, classes=None)
Classic Markov transition matrices.

Parameters

- **class_ids** [array] (n, t), one row per observation, one column recording the state of each observation, with as many columns as time periods.
- **classes** [array] (k, 1), all different classes (bins) of the matrix.
Examples

```python
>>> import numpy as np
>>> from pysal.explore.giddy.markov import Markov
>>> c = [['b', 'a', 'c'], ['c', 'c', 'a'], ['c', 'b', 'c']]
>>> c.extend([['a', 'a', 'b'], ['a', 'b', 'c']])
>>> c = np.array(c)
>>> m = Markov(c)
>>> m.classes.tolist()
['a', 'b', 'c']
>>> m.p
array([[0.25 , 0.5 , 0.25 ],
       [0.33333333, 0. , 0.66666667],
       [0.33333333, 0.33333333, 0.33333333]])
>>> m.steady_state
array([0.30769231, 0.28846154, 0.40384615])

US nominal per capita income 48 states 81 years 1929-2009

```python
>>> import pysal.lib
>>> import pysal.viz.mapclassify as mc
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("usjoin.csv"))
>>> pci = np.array([f.by_col[str(y)]
                     for y in range(1929,2010)])

set classes to quintiles for each year

```python
>>> q5 = np.array([mc.Quantiles(y).yb for y in pci]).transpose()
>>> m = Markov(q5)
>>> m.transitions
array([[729., 71., 1., 0., 0.],
       [ 72., 567., 80., 3., 0.],
       [ 0., 81., 631., 86., 2.],
       [ 0., 3., 86., 573., 56.],
       [ 0., 0., 1., 57., 741.]])
>>> m.p
array([[0.91011236, 0.0886392 , 0.00124844, 0. , 0. ],
       [0.09972299, 0.78531856, 0.11080332, 0.00415512, 0. ],
       [0. , 0.10125 , 0.78875 , 0.1075 , 0.0025 ],
       [0. , 0.00417827, 0.11977716, 0.79805014, 0.07799443],
       [0. , 0. , 0.00125156, 0.07133917, 0.92740926]])
>>> m.steady_state
array([0.20774716, 0.18725774, 0.20740537, 0.18821787, 0.20937187])

Relative incomes

```python
>>> pci = pci.transpose()
>>> rpci = pci/(pci.mean(axis=0))
>>> rq = mc.Quantiles(rpci.flatten()).yb.reshape(pci.shape)
>>> mq = Markov(rq)
>>> mq.transitions
array([[707., 58., 7., 1., 0.],
       [ 50., 629., 80., 1., 1.],
       [ 4., 79., 610., 73., 2.],
       [ 0., 7., 72., 650., 37.],
       [ 0., 0., 0., 48., 724.]])
>>> mq.steady_state
array([0.17957376, 0.21631443, 0.21499942, 0.21134662, 0.17776576])
```
Attributes

- **p** ([array], (k, k)), transition probability matrix.
- **steady_state** ([array], (k,)), ergodic distribution.
- **transitions** ([array], (k, k)), count of transitions between each state i and j.

__init__ (class_ids[, classes=None])
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(class_ids[, classes]) Initialize self.

Attributes

**steady_state**

---

**pysal.explore.giddy.markov.Spatial_Markov**

class pysal.explore.giddy.markov.Spatial_Markov(y, w, k=4, m=4, permutations=0, fixed=True, discrete=False, cutoffs=None, lag_cutoffs=None, variable_name=None)

Markov transitions conditioned on the value of the spatial lag.

Parameters

- **y** ([array], (n, t)), one row per observation, one column per state of each observation, with as many columns as time periods.
- **w** ([W] spatial weights object).
- **k** [integer, optional] number of classes (quantiles) for input time series y. Default is 4. If discrete=True, k is determined endogenously.
- **m** [integer, optional] number of classes (quantiles) for the spatial lags of regional time series. Default is 4. If discrete=True, m is determined endogenously.
- **permutations** [int, optional] number of permutations for use in randomization based inference (the default is 0).
- **fixed** [bool, optional] If true, discretization are taken over the entire n*t pooled series and cutoffs can be user-defined. If cutoffs and lag_cutoffs are not given, quantiles are used. If false, quantiles are taken each time period over n. Default is True.
- **discrete** [bool, optional] If true, categorical spatial lags which are most common categories of neighboring observations serve as the conditioning and fixed is ignored; if false, weighted averages of neighboring observations are used. Default is false.
- **cutoffs** [array, optional] users can specify the discretization cutoffs for continuous time series. Default is None, meaning that quantiles will be used for the discretization.
- **lag_cutoffs** [array, optional] users can specify the discretization cutoffs for the spatial lags of continuous time series. Default is None, meaning that quantiles will be used for the discretization.
variable_name  [string] name of variable.

Notes

Based on [Rey01].

The shtest and chi2 tests should be used with caution as they are based on classic theory assuming random transitions. The x2 based test is preferable since it simulates the randomness under the null. It is an experimental test requiring further analysis.

Examples

```python
>>> import pysal.lib
>>> from pysal.explore.giddy.markov import Spatial_Markov
>>> import numpy as np

>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("usjoin.csv"))
>>> pci = np.array([f.by_col[str(y)] for y in range(1929, 2010)])
>>> pci = pci.transpose()
>>> rpci = pci/(pci.mean(axis=0))
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("states48.gal")).read()
>>> w.transform = 'r'

Now we create a Spatial_Markov instance for the continuous relative per capita income time series for 48 US lower states 1929-2009. The current implementation allows users to classify the continuous incomes in a more flexible way.

(1) Global quintiles to discretize the income data (k=5), and global quintiles to discretize the spatial lags of incomes (m=5).

```python
>>> sm = Spatial_Markov(rpci, w, fixed=True, k=5, m=5, variable_name='rpci')
```
The probability of a poor state remaining poor is 0.963 if their neighbors are in the 1st quintile and 0.798 if their neighbors are in the 2nd quintile. The probability of a rich economy remaining rich is 0.976 if their neighbors are in the 5th quintile, but if their neighbors are in the 4th quintile this drops to 0.903.

The global transition probability matrix is estimated:

```
>>> print(sm.p)
[[0.91461837 0.07503234 0.00905563 0.00129366 0. 
 0.06570302 0.82654402 0.10512484 0.00131406 0.00131406
 0.00520833 0.10286458 0.79427083 0.09505208 0.00260417
 0. 0.00913838 0.09399478 0.84856397 0.04830287
 0. 0. 0.00775194 0.09411765 0.79274611 0.0984456 0.00518135]
[[0.8852459 0.09836066 0. 0.01639344 0.01020408 0.87058824 0.3529412
 0.03875969 0.81395349 0.13953488 0.00775194]
[[0.049505 0.09405941 0.77722772 0.11881188 0.0049505 ]
[[0. 0.2339181 0.12865497 0.75438596 0.09356725
 0. 0. 0.09661836 0.90338164]]
[[0.33333333 0.66666667 0. 0. 0. 
 0.0483871 0.77419355 0.16129032 0.01612903 0. 
 0.01149425 0.16091954 0.74712644 0.08045977 0. 
 0. 0.01036269 0.06217617 0.89637306 0.03108808
 0. 0. 0.02352941 0.97647059]]
```

The Q and likelihood ratio statistics are both significant indicating the dynamics are not homogeneous across the lag classes:

```
>>> "%3f" %sm.LR
'170.659'
>>> "%3f" %sm.Q
'200.624'
>>> "%3f" %sm.LR_p_value
'0.000'
>>> "%3f" %sm.Q_p_value
'0.000'
>>> sm.dof_hom
60
```

The long run distribution for states with poor (rich) neighbors has 0.435 (0.018) of the values in the first quintile, 0.263 (0.200) in the second quintile, 0.204 (0.190) in the third, 0.0684 (0.255) in the fourth and 0.029 (0.337) in the fifth quintile.

```
>>> sm.S
array([(0.43509425, 0.2635327, 0.20363044, 0.06841983, 0.02932278],
      [0.13391287, 0.33993305, 0.25153036, 0.23343016, 0.04119356],
      [0.12124869, 0.21137444, 0.2653101 , 0.29013417, 0.1137326 ],
      [0.0776413 , 0.19748806, 0.25352636, 0.22480415, 0.24654013],
      [0.01776781, 0.19964349, 0.19009833, 0.25524697, 0.33724343])
```

States with incomes in the first quintile with neighbors in the first quintile return to the first quartile after 2.298 years, after leaving the first quintile. They enter the fourth quintile after 80.810 years after leaving the first quintile, on average. Poor states within neighbors in the fourth quintile return to the first quintile, on average, after 12.88 years, and would enter the fourth quintile after 28.473 years.
>>> for f in sm.F:
    ...    print(f)
    ...
    [
[ 2.29835259  28.95614035  46.14285714  80.80952381  279.42857143]
[ 33.86549708  3.79459555  22.57142857  57.23809524  255.85714286]
[ 37.6865497  12.76315789  6.25714286  14.61564626  198.61904762]
[ 52.6865497  18.76315789  12.25714286  6.  34.1031746]
[ 7.46754205  9.70574606  25.76785714  74.53116883  194.23446197]
[ 2.91475577  24.97142857  73.73474026  193.4380334]
[ 53.74777115  28.48447637  48.76331169  168.4660482]
[ 72.0363562  46.9601483  18.46153846  4.28393646  119.70329314]
[ 77.1917276  52.08887197  23.6043956  5.14285714  24.27564033]
[ 8.24751154  6.53333333  18.38765432  40.70864198  112.76732026]
[ 47.35040872  4.73094099  34.17530864  106.23398693]
[ 69.42288828  24.76666667  94.3796594]
[ 83.72288828  39.06666667  76.36702977]
[ 93.52288828  48.86666667  8.79255406]
[ 99.4614206  5.06359731  10.54545198  49.68944423]
[117.7677159  23.03735526  43.57927247]
[127.89752089  32.4393006  31.63099455]
[138.24752089  42.7893006  4.05613474]
[ 56.2815534  1.5  10.57236842  25.01273913  110.54347826]
[ 82.9223301  5.00892857  9.07236842  25.52173913  104.94565217]
[ 97.17718447  19.53125  21.42391304  43.57927247]
[127.1407767  48.74107143  83.52173913]
[169.6407767  91.24107143  2.96521739]]

(2) Global quintiles to discretize the income data (k=5), and global quartiles to discretize the spatial lags of incomes (m=4).

>>> sm = Spatial_Markov(rpci, w, fixed=True, k=5, m=4, variable_name='rpci')

We can also examine the cutoffs for the incomes and cutoffs for the spatial lags:

>>> sm.cutoffs
array([0.83999133, 0.94707545, 1.03242697, 1.14911154])

>>> sm.lag_cutoffs
array([0.91440247, 0.98583079, 1.08698351])

We now look at the estimated spatially lag conditioned transition probability matrices.

>>> for p in sm.P:
    ...    print(p)
    ...
    [
[ 0.95708955  0.03544776  0.00746269  0.  0.  ]
[ 0.05825243  0.83980583  0.18461538  0.  0.  ]
[ 0.  0.1294964  0.76258993  0.10791367  0.  ]
[ 0.  0.01538462  0.8461538  0.72307692  0.07692308]
[ 0.  0.  0.  0.14285714  0.85714286]
[ 0.7421875  0.234375  0.0234375  0.  0.  ]
[ 0.08550186  0.85130112  0.6319703  0.  0.  ]
[ 0.0685801  0.6926407  0.8614786  0.05627706  0.004329]
[ 0.  0.  0.5363985  0.92337165  0.02298851]
[ 0.  0.  0.  0.13432836  0.85657164]
[ 0.95145631  0.04854369  0.  0.  0.  ]
[ 0.06  0.79  0.145  0.  0.005  ]

(continues on next page)
We now obtain 4 5×5 spatial lag conditioned transition probability matrices instead of 5 as in case (1).

The Q and likelihood ratio statistics are still both significant.

```python
>>> %.3f
'sm.LR
'172.105'
>>> %.3f
'sm.Q
'321.128'
>>> %.3f
'sm.LR_p_value
'0.000'
>>> %.3f
'sm.Q_p_value
'0.000'
>>> sm.dof_hom
45
```

(3) We can also set the cutoffs for relative incomes and their spatial lags manually. For example, we want the defining cutoffs to be [0.8, 0.9, 1, 1.2], meaning that relative incomes: 2.1 smaller than 0.8 : class 0 2.2 between 0.8 and 0.9: class 1 2.3 between 0.9 and 1.0 : class 2 2.4 between 1.0 and 1.2: class 3 2.5 larger than 1.2: class 4

```python
>>> cc = np.array([0.8, 0.9, 1, 1.2])
>>> sm = Spatial_Markov(rpci, w, cutoffs=cc, lag_cutoffs=cc, variable_name='rpci')
>>> sm.cutoffs
array([0.8, 0.9, 1. , 1.2])
>>> sm.k
5
>>> sm.lag_cutoffs
array([0.8, 0.9, 1. , 1.2])
>>> sm.m
5
>>> for p in sm.P:
...    print(p)
[[0.96703297 0.03296703 0. 0. 0. ]
[0.10638298 0.68085106 0.21276596 0. 0. ]
[0. 0.14285714 0.7755102 0.08163265 0. ]
[0. 0. 0.5 0.5 0. ]
[0. 0. 0. 0. 0. ]]
[[0.88636364 0.10960606 0.00757576 0. 0. ]
[0.04402516 0.89308176 0.06289308 0. 0. ]
[0. 0.05882353 0.8627451 0.07843137 0. ]
[0. 0. 0.13846154 0.86153846 0. ]
[0. 0. 0. 0. 0.1]]
[[0.7808219 0.17808219 0.02739726 0.01369863 0. ]
[0.0348372 0.90406977 0.05813953 0.00290698 0. ]
[0. 0.05919003 0.84735202 0.09034268 0.00311526]
[0. 0. 0.05811623 0.92985972 0.01202405]
[0. 0. 0. 0.14285714 0.85741286]]
```

(continues on next page)


(4) Spatial_Markov also accept discrete time series and calculate categorical spatial lags on which several transition probability matrices are conditioned. Let’s still use the US state income time series to demonstrate. We first discretize them into categories and then pass them to Spatial_Markov.

```python
>>> import pysal.viz.mapclassify as mc
>>> y = mc.Quantiles(rpci.flatten(), k=5).yb.reshape(rpci.shape)
>>> np.random.seed(5)
>>> sm = Spatial_Markov(y, w, discrete=True, variable_name='discretized rpci')
>>> sm.k
5
>>> sm.m
5
>>> for p in sm.P:
...     print(p)
[[0.00295858 0.06213018 0.82248521 0.10946746 0.00295858]
 [0.00185529 0.07606679 0.88497217 0.03710575]
 [0.0 0.0 0.07803468 0.92196532]]
[0.0 0.05660377 0.90566038 0.03773585]
[0.0 0.0 0.03932584 0.96067416]]

Attributes

- `class_ids` [array] (n, t), discretized series if y is continuous. Otherwise it is identical to y.
- `classes` [array] (k, 1), all different classes (bins).
- `lclass_ids` [array] (n, t), spatial lag series.
- `lclasses` [array] (k, 1), all different classes (bins) for spatial lags.
p [array] (k, k), transition probability matrix for a-spatial Markov.
s [array] (k, 1), ergodic distribution for a-spatial Markov.
transitions [array] (k, k), counts of transitions between each state \( i \) and \( j \) for a-spatial Markov.
T [array] (k, k, k), counts of transitions for each conditional Markov. \( T[0] \) is the matrix of transitions for observations with lags in the 0th quantile; \( T[k-1] \) is the transitions for the observations with lags in the \( k-1 \)th.
P [array] (k, k, k), transition probability matrix for spatial Markov first dimension is the conditioned on the lag.
S [array] (k, k), steady state distributions for spatial Markov. Each row is a conditional steady_state.
F [array] (k, k, k), first mean passage times. First dimension is conditioned on the lag.
shtest [list] (k elements), each element of the list is a tuple for a multinomial difference test between the steady state distribution from a conditional distribution versus the overall steady state distribution: first element of the tuple is the chi2 value, second its p-value and the third the degrees of freedom.
chi2 [list] (k elements), each element of the list is a tuple for a chi-squared test of the difference between the conditional transition matrix against the overall transition matrix: first element of the tuple is the chi2 value, second its p-value and the third the degrees of freedom.
x2 [float] sum of the chi2 values for each of the conditional tests. Has an asymptotic chi2 distribution with \( k(k-1)(k-1) \) degrees of freedom. Under the null that transition probabilities are spatially homogeneous. (see chi2 above)
x2_dof [int] degrees of freedom for homogeneity test.
x2_pvalue [float] p-value for homogeneity test based on analytic distribution
x2_rpvalue [float] (if permutations>0) pseudo p-value for x2 based on random spatial permutations of the rows of the original transitions.
x2_realizations [array] (permutations,1), the values of x2 for the random permutations.
Q [float] Chi-square test of homogeneity across lag classes based on [BB03].
Q_p_value [float] p-value for Q.
LR [float] Likelihood ratio statistic for homogeneity across lag classes based on [BB03].
LR_p_value [float] p-value for LR.
dof_hom [int] degrees of freedom for LR and Q, corrected for 0 cells.

### Methods

<table>
<thead>
<tr>
<th>summary([file_name])</th>
<th>A summary method to call the Markov homogeneity test to test for temporally lagged spatial dependence.</th>
</tr>
</thead>
</table>

**__init__**

```
y, w, k=4, m=4, permutations=0, fixed=True, discrete=False, cutoffs=None, lag_cutoffs=None, variable_name=None
```

Initialize self. See help(type(self)) for accurate signature.
Methods

**__init__**(y, w[, k, m, permutations, fixed, ...]) Initialize self.

**summary**(file_name) A summary method to call the Markov homogeneity test to test for temporally lagged spatial dependence.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
<tr>
<td>LR</td>
</tr>
<tr>
<td>LR_p_value</td>
</tr>
<tr>
<td>Q</td>
</tr>
<tr>
<td>Q_p_value</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>chi2</td>
</tr>
<tr>
<td>dof_hom</td>
</tr>
<tr>
<td>ht</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>shtest</td>
</tr>
<tr>
<td>x2</td>
</tr>
<tr>
<td>x2_dof</td>
</tr>
<tr>
<td>x2_pvalue</td>
</tr>
</tbody>
</table>

**pysal.explore.giddy.markov.LISA_Markov**

*class* pysal.explore.giddy.markov.LISA_Markov**(y, w, permutations=0, significance_level=0.05, geoda_quads=False)**

Markov for Local Indicators of Spatial Association

**Parameters**

- **y** [array] (n, t), n cross-sectional units observed over t time periods.
- **w** [W] spatial weights object.
- **permutations** [int, optional] number of permutations used to determine LISA significance (the default is 0).
- **significance_level** [float, optional] significance level (two-sided) for filtering significant LISA endpoints in a transition (the default is 0.05).
- **geoda_quads** [bool] If True use GeoDa scheme: HH=1, LL=2, LH=3, HL=4. If False use PySAL Scheme: HH=1, LH=2, LL=3, HL=4. (the default is False).

**Examples**

```python
>>> import pysal.lib
>>> import numpy as np
>>> from pysal.explore.giddy.markov import LISA_Markov
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("usjoin.csv"))
>>> years = list(range(1929, 2010))
>>> pci = np.array([f.by_col[str(y)] for y in years]).transpose()
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("states48.gal"))
```
```python
>>> lm = LISA_Markov(pci, w)
>>> lm.classes
array([1, 2, 3, 4])
>>> lm.steady_state
array([0.28561505, 0.14190226, 0.40493672, 0.16754598])
>>> lm.transitions
array([[1.087e+03, 4.400e+01, 4.000e+00, 3.400e+01],
        [4.100e+01, 4.700e+02, 3.600e+01, 1.000e+00],
        [5.000e+00, 3.400e+01, 1.422e+03, 3.900e+01],
        [3.000e+01, 1.000e+00, 4.000e+01, 5.520e+02]])
>>> lm.p
array([[0.92985458, 0.03763901, 0.00342173, 0.02908469],
        [0.07481752, 0.85766423, 0.06569343, 0.00182482],
        [0.00333333, 0.02266667, 0.948 , 0.026 ],
        [0.04815409, 0.00160514, 0.06420546, 0.88603531]])
>>> lm.move_types[0,:3]
array([11, 11, 11])
>>> lm.move_types[0,-3:]
array([11, 11, 11])
```

Now consider only moves with one, or both, of the LISA end points being significant

```python
>>> np.random.seed(10)
>>> lm_random = LISA_Markov(pci, w, permutations=99)
>>> lm_random.significant_moves[0,:3]
array([11, 11, 11])
>>> lm_random.significant_moves[0,-3:]
array([59, 43, 27])
```

Any value less than 49 indicates at least one of the LISA end points was significant. So for example, the first spatial unit experienced a transition of type 11 (LL, LL) during the first three and last tree intervals (according to `lm.move_types`), however, the last three of these transitions involved insignificant LISAS in both the start and ending year of each transition.

Test whether the moves of `y` are independent of the moves of `wy`

```python
>>> "Chi2: $%8.3f$, p: $%5.2f$, dof: $%d$ \% lm.chi_2
'Chi2: 1058.208, p: 0.00, dof: 9'
```

Actual transitions of LISAs

```python
>>> lm.transitions
array([[1.087e+03, 4.400e+01, 4.000e+00, 3.400e+01],
        [4.100e+01, 4.700e+02, 3.600e+01, 1.000e+00],
        [5.000e+00, 3.400e+01, 1.422e+03, 3.900e+01],
        [3.000e+01, 1.000e+00, 4.000e+01, 5.520e+02]])
```

Expected transitions of LISAs under the null `y` and `wy` are moving independently of one another

```python
>>> lm.expected_t
array([[1.12328098e+03, 1.15377356e+01, 3.47522158e-01, 3.38337644e+01],
        [4.100e+01, 4.700e+02, 3.600e+01, 1.000e+00],
        [5.000e+00, 3.400e+01, 1.422e+03, 3.900e+01],
        [3.000e+01, 1.000e+00, 4.000e+01, 5.520e+02]])
```

If the LISA classes are to be defined according to GeoDa, the `geoda_quad` option has to be set to true
```python
>>> lm.q[0:5,0]
anarray([3, 2, 3, 1, 4])
>>> lm = LISA_Markov(pci, w, geoda_quads=True)
>>> lm.q[0:5,0]
anarray([2, 3, 2, 1, 4])
```

Attributes

- **chi_2** [tuple] (3 elements) (chi square test statistic, p-value, degrees of freedom) for test that dynamics of y are independent of dynamics of wy.
- **classes** [array] (4, 1) 1=HH, 2=LH, 3=LL, 4=HL (own, lag) 1=HH, 2=LL, 3=LH, 4=HL (own, lag) (if geoda_quads=True)
- **expected_t** [array] (4, 4), expected number of transitions under the null that dynamics of y are independent of dynamics of wy.
- **move_types** [matrix] (n, t-1), integer values indicating which type of LISA transition occurred (q1 is quadrant in period 1, q2 is quadrant in period 2).

<table>
<thead>
<tr>
<th>q1</th>
<th>q2</th>
<th>move_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

- **p** [array] (k, k), transition probability matrix.
- **p_values** [matrix] (n, t), LISA p-values for each end point (if permutations > 0).
- **significant_moves** [matrix] (n, t-1), integer values indicating the type and significance of a LISA transition. st = 1 if significant in period t, else st=0 (if permutations > 0).
<table>
<thead>
<tr>
<th>q1</th>
<th>q2</th>
<th>s1</th>
<th>s2</th>
<th>move_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>33</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>

steady_state [array] (k, ), ergodic distribution.

transitions [array] (4, 4), count of transitions between each state i and j.

spillover [array] Detect spillover locations for diffusion in LISA Markov.

Methods

spillover([quadrant, neighbors_on]) Detect spillover locations for diffusion in LISA Markov.

_init_ (y, w, permutations=0, significance_level=0.05, geoda_quads=False) Initialize self. See help(type(self)) for accurate signature.
Methods

```python
__init__(y, w[, permutations, ...]) Initialize self.
spillover([quadrant, neighbors_on]) Detect spillover locations for diffusion in LISA Markov.
```

Attributes

```python
steady_state
```

cyasal.explore.giddy.markov.kullback

cyasal.explore.giddy.markov.kullback(F)
  Kullback information based test of Markov Homogeneity.

Parameters

```python
F [array] (s, r, r), values are transitions (not probabilities) for s strata, r initial states, r terminal states.
```

Returns

```python
Results [dictionary] (key - value)
  Conditional homogeneity - (float) test statistic for homogeneity of transition probabilities across strata.
  Conditional homogeneity pvalue - (float) p-value for test statistic.
  Conditional homogeneity dof - (int) degrees of freedom = r(s-1)(r-1).
```

Notes

Based on [KKK62]. Example below is taken from Table 9.2.

Examples

```python
>>> import numpy as np
>>> from pysal.explore.giddy.markov import kullback
>>> s1 = np.array([...
  ... [22, 11, 24, 2, 2, 7],
  ... [5, 23, 15, 3, 42, 6],
  ... [4, 21, 190, 25, 20, 34],
  ... [0, 2, 14, 56, 14, 28],
  ... [32, 15, 20, 10, 56, 14],
  ... [5, 22, 31, 18, 13, 134]
  ... ])
>>> s2 = np.array([...
  ... [3, 6, 9, 3, 0, 8],
  ... [1, 9, 3, 12, 27, 5],
  ... [2, 9, 208, 32, 5, 18],
  ... [0, 14, 32, 108, 40, 40],
  ... [22, 14, 9, 26, 224, 14],
  ... [1, 5, 13, 53, 13, 116]
  ... ])
```

(continues on next page)
... })

>>> F = np.array([s1, s2])
>>> res = kullback(F)
>>> "%.3f"%res['Conditional homogeneity']
'160.961'
>>> "%d"%res['Conditional homogeneity dof']
'30'
>>> "%3.1f"%res['Conditional homogeneity pvalue']
'0.0'

pysal.explore.giddy.markov.prais

pysal.explore.giddy.markov.prais(pmat)

Prais conditional mobility measure.

Parameters

pmat [matrix] (k, k), Markov probability transition matrix.

Returns

pr [matrix] (1, k), conditional mobility measures for each of the k classes.

Notes

Prais’ conditional mobility measure for a class is defined as:

\[ pr_i = 1 - p_{i,i} \]

Examples

```python
>>> import numpy as np
>>> import pysal.lib
>>> from pysal.explore.giddy.markov import Markov, prais
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("usjoin.csv"))
>>> pci = np.array([f.by_col[str(y)]
for y in range(1929,2010)])
>>> q5 = np.array([mc.Quantiles(y).yb for y in pci]).transpose()
>>> m = Markov(q5)
>>> m.transitions
array([[ 729., 71., 1., 0., 0.],
       [ 72., 567., 80., 3., 0.],
       [ 0., 81., 631., 86., 2.],
       [ 0., 3., 86., 573., 56.],
       [ 0., 0., 1., 57., 741.]])
>>> m.p
array([[0.91011236, 0.08988764, 0.00124844, 0. , 0. ],
       [0.09972299, 0.78531856, 0.11080332, 0.00415512, 0. ],
       [0. , 0.10125 , 0.78875 , 0.1075 , 0.0025 ],
       [0. , 0.00417827, 0.11977716, 0.79805014, 0.07799443],
       [0. , 0. , 0.00125156, 0.07133917, 0.92740926]])
>>> prais(m.p)
array([0.08988764, 0.21468144, 0.211250, 0.20194986, 0.07259074])
```
pysal.explore.giddy.markov.homogeneity

Test for homogeneity of Markov transition probabilities across regimes.

**Parameters**

- `transition_matrices` [list] of transition matrices for regimes, all matrices must have same size (r, c). r is the number of rows in the transition matrix and c is the number of columns in the transition matrix.
- `regime_names` [sequence] Labels for the regimes.
- `class_names` [sequence] Labels for the classes/states of the Markov chain.
- `title` [string] name of test.

**Returns**

: `implicit` an instance of Homogeneity_Results.

pysal.explore.giddy.ergodic.steady_state

Calculates the steady state probability vector for a regular Markov transition matrix P.

**Parameters**

- `P` [array] (k, k), an ergodic Markov transition probability matrix.

**Returns**

: `array` (k, ), steady state distribution.

**Examples**

Taken from [KS67]. Land of Oz example where the states are Rain, Nice and Snow, so there is 25 percent chance that if it rained in Oz today, it will snow tomorrow, while if it snowed today in Oz there is a 50 percent chance of snow again tomorrow and a 25 percent chance of a nice day (nice, like when the witch with the monkeys is melting).

```python
>>> import numpy as np
>>> from pysal.explore.giddy.ergodic import steady_state
>>> p=np.array([[.5, .25, .25], [.5,0,.5], [.25,.25,.5]])
>>> steady_state(p)
array([0.4, 0.2, 0.4])
```

Thus, the long run distribution for Oz is to have 40 percent of the days classified as Rain, 20 percent as Nice, and 40 percent as Snow (states are mutually exclusive).

pysal.explore.giddy.ergodic.fmpt

Calculates the matrix of first mean passage times for an ergodic transition probability matrix.

**Parameters**
P [array] (k, k), an ergodic Markov transition probability matrix.

Returns

M [array] (k, k), elements are the expected value for the number of intervals required for a chain starting in state i to first enter state j. If i=j then this is the recurrence time.

Notes

Uses formulation (and examples on p. 218) in [KS67].

Examples

```python
>>> import numpy as np
>>> from pysal.explore.giddy.ergodic import fmpt
>>> p=np.array([[.5, .25, .25], [.5, 0, .5], [.25, .25, .5]])
>>> fm=fmpt(p)
>>> fm
array([[ 2.5 ,  4. , 3.33333333],
        [ 2.66666667,  5. , 2.66666667],
        [ 3.33333333,  4. ,  2.5]])
```

Thus, if it is raining today in Oz we can expect a nice day to come along in another 4 days, on average, and snow to hit in 3.33 days. We can expect another rainy day in 2.5 days. If it is nice today in Oz, we would experience a change in the weather (either rain or snow) in 2.67 days from today. (That wicked witch can only die once so I reckon that is the ultimate absorbing state).

pysal.explore.giddy.ergodic.var_fmpt

pysal.explore.giddy.ergodic.var_fmpt(P)

Variances of first mean passage times for an ergodic transition probability matrix.

Parameters

P [array] (k, k), an ergodic Markov transition probability matrix.

Returns

: array (k, k), elements are the variances for the number of intervals required for a chain starting in state i to first enter state j.

Notes

Uses formulation (and examples on p. 83) in [KS67].

Examples

```python
>>> import numpy as np
>>> from pysal.explore.giddy.ergodic import var_fmpt
>>> p=np.array([[.5, .25, .25], [.5, 0, .5], [.25, .25, .5]])
>>> vfm=var_fmpt(p)
>>> vfm
array([[ 5.58333333, 12. , 6.88888889],
       [ 5.58333333,  4. , 2.58333333],
       [ 5.58333333, 12. , 6.88888889]])
```
Directional LISA

```
[ 6.22222222, 12. , 6.22222222],
[ 6.88888889, 12. , 5.58333333]]
```

**giddy.directional.Rose**(*Y*, *w[, k]*)  
Rose diagram based inference for directional LISAs.

**pysal.explore.giddy.directional.Rose**

**class**  

```
class pysal.explore.giddy.directional.Rose(Y, w, k=8)
```

Rose diagram based inference for directional LISAs.

For n units with LISA values at two points in time, the Rose class provides the LISA vectors, their visualization, and computationally based inference.

**Parameters**

- **Y** [array (n,2)] Columns correspond to end-point time periods to calculate LISA vectors for n object.
- **w** [PySAL W] Spatial weights object.
- **k** [int] Number of circular sectors in rose diagram.

**Attributes**

- **cuts** [(k, 1) ndarray] Radian cuts for rose diagram (circular histogram).
- **counts**: (k, 1) ndarray Number of vectors contained in each sector.
- **r** [(n, 1) ndarray] Vector lengths.
- **theta** [(n,1) ndarray] Signed radians for observed LISA vectors.

If self.permute is called the following attributes are available:

- **alternative** [string] Form of the specified alternative hypothesis ['two-sided'(default) | 'positive' | 'negative']
- **counts_perm** [(permutations, k) ndarray] Counts obtained for each sector for every permutation
- **expected_perm** [(k, 1) ndarray] Average number of counts for each sector taken over all permutations.
- **p** [(k, 1) ndarray] Psuedo p-values for the observed sector counts under the specified alternative.
- **larger_perm** [(k, 1) ndarray] Number of times realized counts are as large as observed sector count.
- **smaller_perm** [(k, 1) ndarray] Number of times realized counts are as small as observed sector count.

**Methods**

```
permute([permutations, alternative])
```

Generate ransom spatial permutations for inference on LISA vectors.

```
Table 102 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plot(attribute, ax)</code></td>
<td>Plot the rose diagram.</td>
</tr>
<tr>
<td><code>plot_origin()</code></td>
<td>Plot vectors of positional transition of LISA values starting from the same origin.</td>
</tr>
<tr>
<td><code>plot_vectors(arrows)</code></td>
<td>Plot vectors of positional transition of LISA values within quadrant in scatterplot in a polar plot.</td>
</tr>
</tbody>
</table>

__init__(Y, w, k=8)

Calculation of rose diagram for local indicators of spatial association.

**Parameters**

- **Y** [(n, 2) ndarray] Variable observed on n spatial units over 2 time periods
- **w** [W] Spatial weights object.
- **k** [int] number of circular sectors in rose diagram (the default is 8).

**Notes**

Based on [RMA11].

**Examples**

Constructing data for illustration of directional LISA analytics. Data is for the 48 lower US states over the period 1969-2009 and includes per capita income normalized to the national average.

Load comma delimited data file in and convert to a numpy array

```python
>>> import pysal.lib
>>> from pysal.explore.giddy.directional import Rose
>>> import matplotlib.pyplot as plt
>>> file_path = pysal.lib.examples.get_path("spi_download.csv")
>>> f=open(file_path,'r')
>>> lines=f.readlines()
>>> f.close()
>>> lines=[line.strip().split(',') for line in lines]
>>> names=[line[2] for line in lines[1:-5]]
>>> data=np.array([list(map(int,line[3:])) for line in lines[1:-5]])
```

Bottom of the file has regional data which we don’t need for this example so we will subset only those records that match a state name

```python
>>> sids=list(range(60))
>>> out=['"United States 3/"',
...      '"Alaska 3/"',
...      '"District of Columbia"',
...      '"Hawaii 3/"',
...      '"New England"',
...      '"Mideast"',
...      '"Great Lakes"',
...      '"Plains"',
...      '"Southeast"',
...      '"Southwest"',
...      '"Rocky Mountain"',
...      '"Far West 3/"']
```
Now we convert state incomes to express them relative to the national average

```python
>>> rel = states/(us*1.)
```

Create our contiguity matrix from an external GAL file and row standardize the resulting weights

```python
>>> gal = pysal.lib.io.open(pysal.lib.examples.get_path('states48.gal'))
>>> w = gal.read()
>>> w.transform = 'r'
```

Take the first and last year of our income data as the interval to do the directional directional analysis

```python
>>> Y = rel[:, [0, -1]]
```

Set the random seed generator which is used in the permutation based inference for the rose diagram so that we can replicate our example results

```python
>>> np.random.seed(100)
```

Call the rose function to construct the directional histogram for the dynamic LISA statistics. We will use four circular sectors for our histogram

```python
>>> r4 = Rose(Y, w, k=4)
```

What are the cut-offs for our histogram - in radians

```python
>>> r4.cuts
array([0. , 1.57079633, 3.14159265, 4.71238898, 6.28318531])
```

How many vectors fell in each sector

```python
>>> r4.counts
array([32, 5, 9, 2])
```

We can test whether these counts are different than what would be expected if there was no association between the movement of the focal unit and its spatial lag.

To do so we call the `permute` method of the object

```python
>>> r4.permute()
```

and then inspect the `p` attribute:

```python
>>> r4.p
array([0.04, 0. , 0.02, 0. ])
```

Repeat the exercise but now for 8 rather than 4 sectors

```python
>>> r8 = Rose(Y, w, k=8)
>>> r8.counts
array([32, 5, 9, 2, 7, 3, 6, 3])
```
The default is a two-sided alternative. There is an option for a directional alternative reflecting positive co-movement of the focal series with its spatial lag. In this case the number of vectors in quadrants I and III should be much larger than expected, while the counts of vectors falling in quadrants II and IV should be much lower than expected.

Finally, there is a second directional alternative for examining the hypothesis that the focal unit and its lag move in opposite directions.

We can call the plot method to visualize directional LISAs as a rose diagram conditional on the starting relative income:

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code>(Y, w[, k])</td>
<td>Calculation of rose diagram for local indicators of spatial association.</td>
</tr>
<tr>
<td><code>permute</code>([permutations, alternative])</td>
<td>Generate ransom spatial permutations for inference on LISA vectors.</td>
</tr>
<tr>
<td><code>plot</code>([attribute, ax])</td>
<td>Plot the rose diagram.</td>
</tr>
<tr>
<td><code>plot_origin</code>()</td>
<td>Plot vectors of positional transition of LISA values starting from the same origin.</td>
</tr>
<tr>
<td><code>plot_vectors</code>([arrows])</td>
<td>Plot vectors of positional transition of LISA values within quadrant in scatterplot in a polar plot.</td>
</tr>
</tbody>
</table>

### Economic Mobility Indices

#### pysal.explore.giddy.mobility.markov_mobility

Markov-based mobility index.

Parameters
**p** [array] (k, k), Markov transition probability matrix.

**measure** [string] If measure=“P”, $M_P = \frac{m-\sum_{i=1}^{m} P_{ii}}{m-1}$; if measure = “D”, $M_D = 1 - |\det(P)|$, where $\det(P)$ is the determinant of $P$; if measure = “L2”, $M_{L2} = 1 - |\lambda_2|$, where $\lambda_2$ is the second largest eigenvalue of $P$; if measure = “B1”, $M_{B1} = \frac{m}{m-1} \sum_{i=1}^{m} \pi_i P_{ii}$, where $\pi$ is the initial income distribution; if measure = “B2”, $M_{B2} = \frac{1}{m-1} \sum_{i=1}^{m} \sum_{j=1}^{m} \pi_i P_{ij} |i - j|$, where $\pi$ is the initial income distribution.

**ini** [array] (k,), initial distribution. Need to be specified if measure = “B1” or “B2”. If not, the initial distribution would be treated as a uniform distribution.

**Returns**

**mobi** [float] Mobility value.

**Notes**

The mobility indices are based on [FSZ04].

**Examples**

```python
>>> import numpy as np
>>> import pysal.lib
>>> import pysal.viz.mapclassify as mc
>>> from pysal.explore.giddy.markov import Markov
>>> from pysal.explore.giddy.mobility import markov_mobility

>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("usjoin.csv"))
>>> pci = np.array([f.by_col[str(y)] for y in range(1929,2010)])
>>> q5 = np.array([mc.Quantiles(y).yb for y in pci]).transpose()
>>> m = Markov(q5)
>>> m.p
array([[0.91011236, 0.0886392 , 0.00124844, 0.      , 0.      ],
       [0.09972299, 0.78531856, 0.11080332, 0.00415512, 0.      ],
       [0.      , 0.10125  , 0.78875  , 0.1075  , 0.0025  ],
       [0.      , 0.00417827, 0.11977716, 0.79805014, 0.07799443],
       [0.      , 0.00125156, 0.07133917, 0.92740926, 0.92740926]])

(1) Estimate Shorrock1 mobility index:

```python
>>> mobi_1 = markov_mobility(m.p, measure="P")
>>> print("{:.5f}".format(mobi_1))
0.19759
```

(2) Estimate Shorrock2 mobility index:

```python
>>> mobi_2 = markov_mobility(m.p, measure="D")
>>> print("{:.5f}".format(mobi_2))
0.60685
```

(3) Estimate Sommers and Conlisk mobility index:

```python
```
>>> mobi_3 = markov_mobility(m.p, measure="L2")
>>> print("{:.5f}".format(mobi_3))
0.03978

(4) Estimate Bartholomew1 mobility index (note that the initial distribution should be given):

```python
>>> ini = np.array([0.1,0.2,0.2,0.4,0.1])
>>> mobi_4 = markov_mobility(m.p, measure = "B1", ini=ini)
>>> print("{:.5f}".format(mobi_4))
0.22777
```

(5) Estimate Bartholomew2 mobility index (note that the initial distribution should be given):

```python
>>> ini = np.array([0.1,0.2,0.2,0.4,0.1])
>>> mobi_5 = markov_mobility(m.p, measure = "B2", ini=ini)
>>> print("{:.5f}".format(mobi_5))
0.04637
```

### Exchange Mobility Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>giddy.rank.Theta(y, regime[, permutations])</code></td>
<td>Regime mobility measure.</td>
</tr>
<tr>
<td><code>giddy.rank.Tau(x, y)</code></td>
<td>Kendall’s Tau is based on a comparison of the number of pairs of n observations that have concordant ranks between two variables.</td>
</tr>
<tr>
<td><code>giddy.rank.SpatialTau(x, y, w[, permutations])</code></td>
<td>Spatial version of Kendall’s rank correlation statistic.</td>
</tr>
<tr>
<td><code>giddy.rank.Tau_Local(x, y)</code></td>
<td>Local version of the classic Tau.</td>
</tr>
<tr>
<td><code>giddy.rank.Tau_Local_Neighbor(x, y, w[, ...])</code></td>
<td>Neighbor set LIMA.</td>
</tr>
<tr>
<td><code>giddy.rank.Tau_Local_Neighborhood(x, y, w[, ...])</code></td>
<td>Neighborhood set LIMA.</td>
</tr>
<tr>
<td><code>giddy.rank.Tau_Regional(x, y, regime[, ...])</code></td>
<td>Inter and intraregional decomposition of the classic Tau.</td>
</tr>
</tbody>
</table>

### pysal.explore.giddy.rank.Theta

**class** `pysal.explore.giddy.rank.Theta(y, regime[, permutations]=999)`

Regime mobility measure. [Rey04]

For sequence of time periods Theta measures the extent to which rank changes for a variable measured over n locations are in the same direction within mutually exclusive and exhaustive partitions (regimes) of the n locations.

Theta is defined as the sum of the absolute sum of rank changes within the regimes over the sum of all absolute rank changes.

**Parameters**

- **y** [array] (n, k) with k>=2, successive columns of y are later moments in time (years, months, etc).
- **regime** [array] (n,), values corresponding to which regime each observation belongs to.
- **permutations** [int] number of random spatial permutations to generate for computationally based inference.
Examples

```python
>>> import pysal.lib as ps
>>> from pysal.explore.giddy.rank import Theta
>>> import numpy as np

>>> f=ps.io.open(ps.examples.get_path("mexico.csv"))
>>> vnames=['pcgdp%d' %dec for dec in range(1940,2010,10)]
>>> y=np.transpose(np.array([f.by_col[v] for v in vnames]))
>>> regime=np.array(f.by_col['esquivel99'])
>>> np.random.seed(10)
>>> t=Theta(y,regime,999)

array([[0.41538462, 0.28070175, 0.61363636, 0.62222222, 0.33333333,
        0.47222222]])

>>> t.pvalue_left
array([0.307, 0.077, 0.823, 0.552, 0.045, 0.735])

>>> t.total
array([130., 114., 88., 90., 90., 72.])

>>> t.max_total
512
```

Attributes

- **ranks** [array] ranks of the original y array (by columns).
- **regimes** [array] the original regimes array.
- **total** [array] (k-1, ), the total number of rank changes for each of the k periods.
- **max_total** [int] the theoretical maximum number of rank changes for n observations.
- **theta** [array] (k-1,), the theta statistic for each of the k-1 intervals.
- **permutations** [int] the number of permutations.
- **pvalue_left** [float] p-value for test that observed theta is significantly lower than its expectation under complete spatial randomness.
- **pvalue_right** [float] p-value for test that observed theta is significantly greater than its expectation under complete spatial randomness.

```python
__init__(y, regime, permutations=999)
```

Initialize self. See help(type(self)) for accurate signature.

Methods

```python
__init__(y, regime[, permutations])
```

Initialize self.

**pysal.explore.giddy.rank.Tau**

```python
class pysal.explore.giddy.rank.Tau(x, y)
```

Kendall’s Tau is based on a comparison of the number of pairs of n observations that have concordant ranks between two variables.

**Parameters**

- **x** [array] (n, ), first variable.
y [array] (n,), second variable.

Notes
Modification of algorithm suggested by [Chr05]. PySAL/giddy implementation uses a list based representation of a binary tree for the accumulation of the concordance measures. Ties are handled by this implementation (in other words, if there are ties in either x, or y, or both, the calculation returns Tau_b, if no ties classic Tau is returned.)

Examples
# from scipy example

```python
>>> from scipy.stats import kendalltau
>>> from pysal.explore.giddy.rank import Tau
>>> x1 = [12, 2, 1, 12, 2]
>>> x2 = [1, 4, 7, 1, 0]
>>> kt = Tau(x1,x2)
>>> kt.tau
-0.47140452079103173
>>> kt.tau_p
0.24821309157521476
>>> tau, p = kendalltau(x1,x2)
>>> tau
-0.4714045207910316
>>> p
0.2827454599327748
```

Attributes
- tau [float] The classic Tau statistic.
- tau_p [float] asymptotic p-value.

_init__(x, y)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(x, y) Initialize self.

pysal.explore.giddy.rank.SpatialTau
class pysal.explore.giddy.rank.SpatialTau(x, y, w, permutations=0)
Spatial version of Kendall’s rank correlation statistic.

Kendall’s Tau is based on a comparison of the number of pairs of n observations that have concordant ranks between two variables. The spatial Tau decomposes these pairs into those that are spatial neighbors and those that are not, and examines whether the rank correlation is different between the two sets relative to what would be expected under spatial randomness.

Parameters
- x [array] (n,), first variable.
y [array] (n, ), second variable.

w [W] spatial weights object.

**permutations** [int] number of random spatial permutations for computationally based inference.

**Notes**

Algorithm has two stages. The first calculates classic Tau using a list based implementation of the algorithm from [Chr05]. Second stage calculates concordance measures for neighboring pairs of locations using a modification of the algorithm from [PTVF07]. See [Rey14] for details.

**Examples**

```python
>>> import pysal.lib as ps
>>> import numpy as np
>>> from pysal.explore.giddy_rank import SpatialTau
>>> f=ps.io.open(ps.examples.get_path("mexico.csv"))
>>> vnames=['pcgdp%d'%'dec for dec in range(1940,2010,10)]
>>> y=np.transpose(np.array([f.by_col[v] for v in vnames]))
>>> regime=np.array(f.by_col['esquivel99'])
>>> w=ps.weights.block_weights(regime)
>>> np.random.seed(12345)
>>> res=[SpatialTau(y[:,i],y[:,i+1],w,99) for i in range(6)]
>>> for r in res:
...     ev = r.taus.mean()
...     "%.3f %.3f %.3f"%(r.tau_spatial, ev, r.tau_spatial_psim)
... ' 0.397 0.659 0.010'
' 0.492 0.706 0.010'
' 0.651 0.772 0.020'
' 0.714 0.752 0.210'
' 0.683 0.705 0.270'
' 0.810 0.819 0.280'
```

**Attributes**

- **tau** [float] The classic Tau statistic.
- **tau_spatial** [float] Value of Tau for pairs that are spatial neighbors.
- **taus** [array] (permutations, 1), values of simulated tau_spatial values under random spatial permutations in both periods. (Same permutation used for start and ending period).
- **pairs_spatial** [int] Number of spatial pairs.
- **concordant** [float] Number of concordant pairs.
- **concordant_spatial** [float] Number of concordant pairs that are spatial neighbors.
- **extraX** [float] Number of extra X pairs.
- **extraY** [float] Number of extra Y pairs.
- **discordant** [float] Number of discordant pairs.
- **discordant_spatial** [float] Number of discordant pairs that are spatial neighbors.
- **taus** [float] spatial tau values for permuted samples (if permutations>0).
**tau_spatial_psim** [float] one-sided pseudo p-value for observed tau_spatial under the null of spatial randomness of rank exchanges (if permutations>0).

```python
__init__(x, y, w, permutations=0)
```
Initialize self. See help(type(self)) for accurate signature.

### Methods

- `__init__(x, y, w[, permutations])` Initialize self.

---

**pysal.explore.giddy.rank.Tau_Local**

class `pysal.explore.giddy.rank.Tau_Local(x, y)`

Local version of the classic Tau.

Decomposition of the classic Tau into local components.

**Parameters**

- `x` [array] (n,), first variable.
- `y` [array] (n,), second variable.

**Notes**

The equation for calculating local concordance statistic can be found in [Rey16] Equation (9).

**Examples**

```python
>>> import pysal.lib as ps
>>> import numpy as np
>>> from pysal.explore.giddy.rank import Tau_Local, Tau

>>> np.random.seed(10)
>>> f = ps.io.open(ps.examples.get_path("mexico.csv"))
>>> vnames = ["pcgdp%d"%dec for dec in range(1940, 2010, 10)]
>>> y = np.transpose(np.array([f.by_col[v] for v in vnames]))
>>> r = y / y.mean(axis=0)
>>>
>>> tau_local = Tau_Local(r[:,0],r[:,1])
>>> tau_local.tau_local
array([-0.03225806, 0.93548387, 0.80645161, 0.74193548, 0.93548387,
        0.74193548, 0.67741935, 0.41935484, 1.0, 0.54838710,
        0.74193548, 0.93548387, 0.67741935, 0.74193548, 0.80645161,
        0.74193548, 0.54838710, 0.67741935, 0.74193548, 0.74193548,
        0.54838710, -0.16129032, 0.93548387, 0.61290323, 0.67741935,
        0.48387097, 0.93548387, 0.61290323, 0.74193548, 0.41935484,
        0.61290323, 0.61290323])

>>> tau_local.tau_local
0.6612903225806451
>>> tau_classic = Tau(r[:,0],r[:,1])
>>> tau_classic.tau
0.6612903225806451
```

**Attributes**
n [int] number of observations.
tau [float] The classic Tau statistic.
tau_local [array] (n,), local concordance (local version of the classic tau).
S [array] (n, n), concordance matrix, s_{ij}=1 if observation i and j are concordant, s_{ij}=-1 if observation i and j are discordant, and s_{ij}=0 otherwise.

__init__ (x, y)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (x, y) Initialize self.

pysal.explore.giddy.rank.Tau_Local_Neighbor
class pysal.explore.giddy.rank.Tau_Local_Neighbor (x, y, w, permutations=0)
Neighbor set LIMA.
Local concordance relationships between a focal unit and its neighbors. A decomposition of local Tau into neighbor and non-neighbor components.

Parameters

x [array] (n,), first variable.
y [array] (n,), second variable.
w [W] spatial weights object.
permutations [int] number of random spatial permutations for computationally based inference.

Notes

The equation for calculating neighbor set LIMA statistic can be found in [Rey16] Equation (16).

Examples

```python
>>> import pysal.lib as ps
>>> import numpy as np
>>> from pysal.explore.giddy.rank import Tau_Local_Neighbor, SpatialTau
>>> np.random.seed(10)
>>> f = ps.io.open(ps.examples.get_path("mexico.csv"))
>>> vnames = ["pcgdp%d"%dec for dec in range(1940, 2010, 10)]
>>> y = np.transpose(np.array([f.by_col[v] for v in vnames]))
>>> r = y / y.mean(axis=0)
>>> regime = np.array(f.by_col["esquivel99"])
>>> w = ps.weights.block_weights(regime)
>>> res = Tau_Local_Neighbor(r[:,0], r[:,1], w, permutations=999)
>>> res.tau_ln
array([-0.2 , 1. , 1. , 1. , 0.33333333,
        0.6 , 0.6 , -0.5 , 1. , 1. , 1.2])
```
Attributes

n [int] number of observations.

tau_local [array] (n,), local concordance (local version of the classic tau).

S [array] (n,n), concordance matrix, s_(i,j)=1 if observation i and j are concordant, s_(i,j)=-1 if observation i and j are discordant, and s_(i,j)=0 otherwise.

tau_ln [array] (n,), observed neighbor set LIMA values.

tau_ln_weights [array] (n,), weights for neighbor set LIMA at each location. GIMA is the weighted average of neighbor set LIMA.

tau_ln_sim [array] (n, permutations), neighbor set LIMA values for permuted samples (if permutations>0).

tau_ln_pvalues [array] (n, ), one-sided pseudo p-values for observed neighbor set LIMA values under the null that concordance relationship between the focal state and its neighbors is not different from what could be expected from randomly distributed rank changes.

sign [array] (n,), values indicate concordant or discordant: 1 concordant, -1 discordant

__init__ (x, y, w, permutations=0)
Initialize self. See help(type(self)) for accurate signature.

Methods
Neighborhood set LIMA.

An extension of neighbor set LIMA. Consider local concordance relationships for a subset of states, defined as the focal state and its neighbors.

Parameters

- **x** [array] (n,), first variable.
- **y** [array] (n,), second variable.
- **w** [W] spatial weights object.
- **permutations** [int] number of random spatial permutations for computationally based inference.

Notes

The equation for calculating neighborhood set LIMA statistic can be found in [Rey16] Equation (22).

Examples

```python
>>> import pysal.lib as ps
>>> from pysal.explore.giddy.rank import Tau_Local_Neighborhood
>>> import numpy as np

>>> np.random.seed(10)

>>> f = ps.io.open(ps.examples.get_path("mexico.csv"))
>>> vnames = ["pcgdp\%d\%d" for dec in range(1940, 2010, 10)]
>>> y = np.transpose(np.array([f.by_col[v] for v in vnames]))
>>> r = y / y.mean(axis=0)

>>> regime = np.array(f.by_col["esquivel99"])

>>> w = ps.weights.block_weights(regime)

>>> res = Tau_Local_Neighborhood(r[:,0],r[:,1],w,permutations=999)

>>> res.tau_lnhood
array([0.06666667, 0.6 , 0.2 , 0.8 , 0.33333333, 0.6 , 0.2 , 1. , 0.66666667, 0.33333333, 0.33333333, 0.33333333, 0.2 , 1., 0.33333333, 0.33333333, 0.33333333, 0.2 , 0.6 , 0.33333333, 0.33333333, 0.66666667, 0.8 , 0.33333333, 0.2 , 0.6 , 0.8 , 0.33333333, 0.8 , 0.8 , 0.66666667])

>>> res.tau_lnhood_pvalues
array([0.106, 0.33 , 0.107, 0.535, 0.137, 0.414, 0.432, 0.169, 1. , 0.03 , 0.019, 0.146, 0.249, 0.1 , 0.908, 0.225, 0.311, 0.125, 0.399, 0.215, 0.334, 0.115, 0.669, 0.045, 0.11 , 0.525, 0.655, 0.466, 0.236, 0.413, 0.504, 0.038])

>>> res.sign
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

Attributes
n  [int] number of observations.

tau_local  [array] (n, ), local concordance (local version of the classic tau).

S  [array] (n,n), concordance matrix. s_{i,j}=1 if observation i and j are concordant, s_{i,j}=-1 if observation i and j are discordant, and s_{i,j}=0 otherwise.

tau_inhood  [array] (n, ), observed neighborhood set LIMA values.

tau_inhood_sim  [array] (n, permutations), neighborhood set LIMA values for permuted samples (if permutations>0).

tau_inhood_pvalues  [array] (n, 1), one-sided pseudo p-values for observed neighborhood set LIMA values under the null that the concordance relationships for a subset of states, defined as the focal state and its neighbors, is different from what would be expected from randomly distributed rank changes.

sign  [array] (n, ), values indicate concordant or discordant: 1 concordant, -1 discordant

__init__ (x, y, w[, permutations=0])
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (x, y, w[, permutations]) Initialize self.

pysal.explore.giddy.rank.Tau_Regional

class pysal.explore.giddy.rank.Tau_Regional (x, y, regime, permutations=0)
Inter and intraregional decomposition of the classic Tau.

Parameters

x  [array] (n, ), first variable.
y  [array] (n, ), second variable.
regime  [array] (n, ), ids of which regime an observation belongs to.
permutations  [int] number of random spatial permutations for computationally based inference.

Notes

The equation for calculating inter and intraregional Tau statistic can be found in [Rey16] Equation (27).

Examples

```python
>>> import pysal.lib as ps
>>> import numpy as np
>>> from pysal.explore.giddy.rank importTau_Regional
>>> np.random.seed(10)
>>> f = ps.io.open(ps.examples.get_path("mexico.csv"))
>>> vnames = ["pcgdp%d"%dec for dec in range(1940, 2010, 10)]
>>> y = np.transpose(np.array([f.by_col[v] for v in vnames]))
```
```python
>>> r = y / y.mean(axis=0)
>>> regime = np.array(f.by_col['esquivel99'])
>>> res = Tau_Regional(y[:,0],y[:,-1],regime,permutations=999)
>>> res.tau_reg
array([[1. , 0.25 , 0.5 , 0.6 , 0.83333333,
    0.6 , 1. , 0.25 , 0.33333333, 0.5 , 0.3 , 0.91666667,
    0.4 , 0.75 , 0.5 , 0.6 , 0.38888889,
    0.53333333, 0.83333333],
   [0.6 , 0.3 , 0.4 , 0.2 , 0.4 , 0.28 , 0.8 ],
   [0.83333333, 0.91666667, 0.38888889, 0.4 , 0.6 , 0.73333333, 1. ],
   [0.6 , 0.4 , 0.53333333, 0.28 , 0.73333333, 0.8 , 0.8 ],
   [1. , 0.75 , 0.83333333, 0.8 , 1. , 0.8 , 0.33333333]])
>>> res.tau_reg_pvalues
array([[0.782, 0.227, 0.464, 0.638, 0.294, 0.627, 0.201],
   [0.227, 0.352, 0.391, 0.14 , 0.048, 0.252, 0.327],
   [0.464, 0.391, 0.587, 0.198, 0.107, 0.423, 0.124],
   [0.638, 0.14 , 0.198, 0.141, 0.184, 0.089, 0.217],
   [0.294, 0.048, 0.107, 0.184, 0.583, 0.25 , 0.005],
   [0.627, 0.252, 0.423, 0.089, 0.25 , 0.38 , 0.227],
   [0.201, 0.327, 0.124, 0.217, 0.005, 0.227, 0.322]])
```

### Attributes

- **n** [int] number of observations.
- **S** [array] (n .n), concordance matrix, s_[i,j]=1 if observation i and j are concordant, s_[i,j]=-1 if observation i and j are discordant, and s_[i,j]=0 otherwise.
- **tau_reg** [array] (k, k), observed concordance matrix with diagonal elements measuring concordance between units within a regime and the off-diagonal elements denoting concordance between observations from a specific pair of different regimes.
- **tau_reg_sim** [array] (permutations, k, k), concordance matrices for permuted samples (if permutations>0).
- **tau_reg_pvalues** [array] (k, k), one-sided pseudo p-values for observed concordance matrix under the null that income mobility were random in its spatial distribution.

#### __init__ (x, y, regime, permutations=0)

Initialize self. See help(type(self)) for accurate signature.

### Methods

#### __init__(x, y, regime[, permutations])

Initialize self.

**pysal.explore.inequality**: Spatial Inequality Analysis

1.2. API Reference 139
Theil Inequality Measures

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inequality.theil.Theil(y)</code></td>
<td>Classic Theil measure of inequality</td>
</tr>
<tr>
<td><code>inequality.theil.TheilD(y, partition)</code></td>
<td>Decomposition of Theil’s T based on partitioning of observations into exhaustive and mutually exclusive groups</td>
</tr>
<tr>
<td><code>inequality.theil.TheilDSim(y, partition[, ...])</code></td>
<td>Random permutation based inference on Theil’s inequality decomposition.</td>
</tr>
</tbody>
</table>

```python
class pysal.explore.inequality.theil.Theil(y):
    Classic Theil measure of inequality
    
    \[ T = \sum_{i=1}^{n} \left( \frac{y_i}{\sum_{i=1}^{n} y_i} \ln \left[ \frac{N}{\sum_{i=1}^{n} y_i} \right] \right) \]
    
    Parameters
    
    - `y` [array (n,t) or (n,)] with n taken as the observations across which inequality is calculated. If y is (n,) then a scalar inequality value is determined. If y is (n,t) then an array of inequality values are determined, one value for each column in y.

    Notes
    
    This computation involves natural logs. To prevent ln[0] from occurring, a small value is added to each element of y before beginning the computation.

    Examples
    
    ```python
    >>> import pysal.lib
    >>> import numpy as np
    >>> from pysal.explore.inequality.theil import Theil
    >>> f=pysal.lib.io.open(pysal.lib.examples.get_path("mexico.csv"))
    >>> vnames=["pcgdp%d"%dec for dec in range(1940,2010,10)]
    >>> y=np.array([f.by_col[v] for v in vnames]).T
    >>> theil_y=Theil(y)
    >>> theil_y.T
    array([0.20894344, 0.15222451, 0.10472941, 0.10194725, 0.09560113, 0.10511256, 0.10660832])
    ```
```

Attributes

- `T` [array (t,) or (1,)] Theil’s T for each column of y

    ```python
    __init__(y)
    Initialize self. See help(type(self)) for accurate signature.
    ```

Methods
pysal.explore.inequality.theil.TheilID

class pysal.explore.inequality.theil.TheilID(y, partition)
   Decomposition of Theil’s T based on partitioning of observations into exhaustive and mutually exclusive groups

   Parameters
   y [array (n,t) or (n, )] with n taken as the observations across which inequality is calculated. If y is (n,) then a scalar inequality value is determined. If y is (n,t) then an array of inequality values are determined, one value for each column in y.

   partition [array (n, )] elements indicating which partition each observation belongs to. These are assumed to be exhaustive.

Examples

```python
>>> import pysal.lib
>>> from pysal.explore.inequality.theil import TheilID
>>> import numpy as np
>>> f = pysal.lib.io.open(pysal.lib.examples.get_path("mexico.csv"))
>>> vnames=['pcgdp%d'%'dec for dec in range(1940,2010,10)]
>>> y = np.array([f.by_col[v] for v in vnames]).T
>>> regimes=np.array(f.by_col('hanson98'))
>>> theil_d=TheilID(y,regimes)
>>> theil_d.bg
array([0.0345889, 0.02816853, 0.05260921, 0.05931219, 0.03205257, 0.02963731, 0.03635872])
>>> theil_d.wg
array([0.17435454, 0.12405598, 0.0521202, 0.04263506, 0.06354856, 0.07547525, 0.0702496])
```

Attributes

   T [array (n,t) or (n,)] global inequality T
   bg [array (n,t) or (n,)] between group inequality
   wg [array (n,t) or (n,)] within group inequality

   __init__(y, partition)
   Initialize self. See help(type(self)) for accurate signature.

Methods

   __init__(y, partition) Initialize self.

pysal.explore.inequality.theil.TheilDSim

class pysal.explore.inequality.theil.TheilDSim(y, partition, permutations=99)
   Random permutation based inference on Theil’s inequality decomposition.

   Provides for computationally based inference regarding the inequality decomposition using random spatial per-
mutations. See [RSastreGutierrez10].

**Parameters**

*y* [array (n,t) or (n, )] with n taken as the observations across which inequality is calculated. If *y* is (n,) then a scalar inequality value is determined. If *y* is (n,t) then an array of inequality values are determined, one value for each column in *y*.

**partition** [array (n, )] elements indicating which partition each observation belongs to. These are assumed to be exhaustive.

**permutations** [int] Number of random spatial permutations for computationally based inference on the decomposition.

**Examples**

```python
>>> import pysal.lib
>>> from pysal.explore.inequality.theil import TheilDSim
>>> import numpy as np

>>> f=pysal.lib.io.open(pysal.lib.examples.get_path("mexico.csv"))
>>> vnames=["pcgdp%d%dec"%dec for dec in range(1940,2010,10)]
>>> y=np.array([f.by_col[v] for v in vnames]).T
>>> regimes=np.array(f.by_col('hanson98'))
>>> np.random.seed(10)
>>> theil_ds=TheilDSim(y,regimes,999)
>>> theil_ds.bg_pvalue
array([0.4 , 0.344, 0.001, 0.001, 0.034, 0.072, 0.032])
```

**Attributes**

- **observed** [array (n,t) or (n,)] TheilD instance for the observed data.
- **bg** [array (permutations+1,t)] between group inequality
- **bg_pvalue** [array (t,1)] p-value for the between group measure. Measures the percentage of the realized values that were greater than or equal to the observed bg value. Includes the observed value.
- **wg** [array (size=permutations+1)] within group inequality Depending on the shape of *y*, 1 or 2-dimensional

**Methods**

- **__init__**(y, partition[, permutations=99])
  Initialize self. See help(type(self)) for accurate signature.

**Gini Inequality Measures**

- **inequality.gini.Gini_Spatial**(x, w[,...]) Spatial Gini coefficient
pysal.explore.inequality.gini.Gini_Spatial

class pysal.explore.inequality.gini.Gini_Spatial(x, w, permutations=99)
Spatial Gini coefficient

Provides for computationally based inference regarding the contribution of spatial neighbor pairs to overall inequality across a set of regions. See [RS13].

Parameters

- y [array (n,1)] attribute
- w [binary spatial weights object]
- permutations [int (default = 99)] number of permutations for inference

Examples

```python
>>> import pysal.lib
>>> import numpy as np
>>> from pysal.explore.inequality.gini import Gini_Spatial

Use data from the 32 Mexican States, Decade frequency 1940-2010

```pysal.lib.io.open(pysal.lib.examples.get_path("mexico.csv"))
>>> vnames=["pcgdp%d"%dec for dec in range(1940,2010,10)]
>>> y=np.transpose(np.array([f.by_col[v] for v in vnames]))

Define regime neighbors

```pysal.lib.weights.block_weights(regimes)
>>> np.random.seed(12345)
>>> gs = Gini_Spatial(y[:,0],w)
>>> gs.p_sim
0.04
>>> gs.wcg
4353856.0
>>> gs.e_wcg
4170356.7474747472

Thus, the amount of inequality between pairs of states that are not in the same regime (neighbors) is significantly higher than what is expected under the null of random spatial inequality.

Attributes

- g [float] Gini coefficient
- wg [float] Neighbor inequality component (geographic inequality)
- wcg [float] Non-neighbor inequality component (geographic complement inequality)
- wcg_share [float] Share of inequality in non-neighbor component

If Permutations > 0
- p_sim [float] pseudo p-value for spatial gini
- e_wcg [float] expected value of non-neighbor inequality component (level) from permutations
- s_wcg [float] standard deviation non-neighbor inequality component (level) from permutations
**Methods**

```
__init__(x, w[, permutations]) Initialize self.
```

**pysal.explore.spaghetti:**

**spaghetti.Network**

```
spaghetti.Network([in_data, node_sig,...]) Spatially-constrained network representation and analytical functionality.

spaghetti.Network.extractgraph() Using the existing network representation, create a graph-theoretic representation by removing all nodes with a neighbor incidence of two (non-articulation points).

spaghetti.Network.contiguityweights([graph,...]) Create a contiguity based W object.

spaghetti.Network.distancebandweights(threshold) Create distance based weights.

spaghetti.Network.snapobservations(in_data, name) Snap a point pattern shapefile to this network object.

spaghetti.Network.compute_distance_to_nodes(x,...) Given an observation on a network edge, return the distance to the two nodes that bound that end.

spaghetti.Network.compute_snap_dist(pattern, idx) Given an observation snapped to a network edge, calculate the distance from the original location to the snapped location.

spaghetti.Network.count_per_edge(obs_on_network) Compute the counts per edge.

spaghetti.Network.simulate_observations(count) Generate a simulated point pattern on the network.

spaghetti.Network.enum_links_node(v0) Returns the edges (links) around node.

spaghetti.Network.node_distance_matrix(...) Called from within allneighbordistances(), nearestneighbordistances(), and distancebandweights().

spaghetti.Network.allneighbordistances(...) Compute either all distances between i and j in a single point pattern or all distances between each i from a source pattern and all j from a destination pattern.

spaghetti.Network.nearestneighbordistances(...) Compute the interpattern nearest neighbor distances or the intrapattern nearest neighbor distances between a source pattern and a destination pattern.

spaghetti.Network.NetworkF(pointpattern[, ...]) Computes a network constrained F-Function

spaghetti.Network.NetworkG(pointpattern[, ...]) Computes a network constrained G-Function
```

Continued on next page
Table 119 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spaghetti.Network.NetworkK(pointpattern[, ...])</code></td>
<td>Computes a network constrained K-Function</td>
</tr>
<tr>
<td><code>spaghetti.Network.segment_edges(distance)</code></td>
<td>Segment all of the edges in the network at either a</td>
</tr>
<tr>
<td><code>spaghetti.Network.savenetwork(filename)</code></td>
<td>Save a network to disk as a binary file.</td>
</tr>
<tr>
<td><code>spaghetti.Network.loadnetwork(filename)</code></td>
<td>Load a network from a binary file saved on disk.</td>
</tr>
</tbody>
</table>

```python
import pysal.explore.spaghetti as spgh
streets_file = examples.get_path('streets.shp')
ntw = spgh.Network(in_data=streets_file)
```

Snap point observations to the network with attribute information.

```python
>>> crimes_file = examples.get_path('crimes.shp')
>>> ntw.snapobservations(crimes_file, 'crimes', attribute=True)
```

And without attribute information.

```python
>>> schools_file = examples.get_path('schools.shp')
>>> ntw.snapobservations(schools_file, 'schools', attribute=False)
```

Attributes

- **in_data** [str] The input shapefile. This must be in .shp format.
- **adjacencylist** [list] List of lists storing node adjacency.
- **nodes** [dict] Keys are tuples of node coords and values are the node ID.
- **edge_lengths** [dict] Keys are tuples of sorted node IDs representing an edge and values are the length.
pointpatterns [dict] Keys are a string name of the pattern and values are point pattern class instances.

node_coords [dict] Keys are the node ID and values are the (x,y) coordinates inverse to nodes.

edges [list] List of edges, where each edge is a sorted tuple of node IDs.

node_list [list] List of node IDs.

alldistances [dict] Keys are the node IDs (int). Values are tuples with two elements as follows (1) a list of the shortest path distances; (2) a dict with the key being the id of the destination node and the value being a list of the shortest path.

distancematrix [numpy.ndarray] all network nodes (non-observations) distance matrix.

graphedges [list] tuples of graph edge ids.

graph_lengths [dict] Keys are the graph edge ids (tuple). Values are the graph edge length (float).

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetworkF(pointpattern[, nsteps, ...])</td>
<td>Computes a network constrained F-Function</td>
</tr>
<tr>
<td>NetworkG(pointpattern[, nsteps, ...])</td>
<td>Computes a network constrained G-Function</td>
</tr>
<tr>
<td>NetworkK(pointpattern[, nsteps, ...])</td>
<td>Computes a network constrained K-Function</td>
</tr>
<tr>
<td>allneighbordistances(sourcepattern[,...])</td>
<td>Compute either all distances between i and j in a single point pattern or all distances between each i from a source pattern and all j from a destination pattern.</td>
</tr>
<tr>
<td>compute_distance_to_nodes(x, y, edge)</td>
<td>Given an observation on a network edge, return the distance to the two nodes that bound that end.</td>
</tr>
<tr>
<td>compute_snap_dist(pattern, idx)</td>
<td>Given an observation snapped to a network edge, calculate the distance from the original location to the snapped location.</td>
</tr>
<tr>
<td>contiguityweights([graph, weightings])</td>
<td>Create a contiguity based W object.</td>
</tr>
<tr>
<td>count_per_edge(obs_on_network[, graph])</td>
<td>Compute the counts per edge.</td>
</tr>
<tr>
<td>distancebandweights(threshold[, n_proccess, ...])</td>
<td>Create distance based weights.</td>
</tr>
<tr>
<td>enum_links_node(v0)</td>
<td>Returns the edges (links) around node.</td>
</tr>
<tr>
<td>extractgraph()</td>
<td>Using the existing network representation, create a graph-theoretic represention by removing all nodes with a neighbor incidence of two (non-articulation points).</td>
</tr>
<tr>
<td>loadnetwork(filename)</td>
<td>Load a network from a binary file saved on disk.</td>
</tr>
<tr>
<td>nearestneighbordistances(sourcepattern[, ...])</td>
<td>Compute the interpattern nearest neighbor distances or the intrapattern nearest neighbor distances between a source pattern and a destination pattern.</td>
</tr>
<tr>
<td>node_distance_matrix(n_processes[, gen_tree])</td>
<td>Called from within allneighbordistances(), nearest-neighbordistances(), and distancebandweights().</td>
</tr>
<tr>
<td>savenetwork(filename)</td>
<td>Save a network to disk as a binary file.</td>
</tr>
<tr>
<td>segment_edges(distance)</td>
<td>Segment all of the edges in the network at either a</td>
</tr>
<tr>
<td>simulate_observations(count[, distribution])</td>
<td>Generate a simulated point pattern on the network.</td>
</tr>
<tr>
<td>snapobservations(in_data, name[, ...])</td>
<td>Snap a point pattern shapefile to this network object.</td>
</tr>
</tbody>
</table>
pysal.explore.spaghetti.Network.NetworkF

Network.F(pointpattern, nsteps=10, permutations=99, threshold=0.2, distribution='uniform', lowerbound=None, upperbound=None)

Computes a network constrained F-Function

**Parameters**

- **pointpattern** [spaghetti.network.PointPattern] A spaghetti point pattern object.
- **nsteps** [int] The number of steps at which the count of the nearest neighbors is computed.
- **permutations** [int] The number of permutations to perform (default 99).
- **threshold** [float] The level at which significance is computed. – 0.5 would be 97.5% and 2.5%
- **distribution** [str] The distribution from which random points are sampled – uniform or poisson
- **lowerbound** [float] The lower bound at which the F-function is computed. (Default 0).
- **upperbound** [float] The upper bound at which the F-function is computed. Defaults to the maximum observed nearest neighbor distance.

**Returns**


**Examples**

```python
>>> import pysal.explore.spaghetti as spgh
>>> ntw = spgh.Network(in_data=examples.get_path('streets.shp'))
>>> pt_str = 'crimes'
>>> in_data = examples.get_path('{}\shp'.format(pt_str))
>>> ntw.snapobservations(in_data, pt_str, attribute=True)
>>> crimes = ntw.pointpatterns['crimes']
>>> sim = ntw.simulate_observations(crimes.npoints)
>>> fres = ntw.NetworkF(crimes, permutations=5, nsteps=10)
>>> fres.lowerenvelope.shape[0]
10
```

pysal.explore.spaghetti.Network.NetworkG

Network.G(pointpattern, nsteps=10, permutations=99, threshold=0.5, distribution='uniform', lowerbound=None, upperbound=None)

Computes a network constrained G-Function

**Parameters**

- **pointpattern** [spaghetti.network.PointPattern] A spaghetti point pattern object.
- **nsteps** [int] The number of steps at which the count of the nearest neighbors is computed.
- **permutations** [int] The number of permutations to perform (default 99).
- **threshold** [float] The level at which significance is computed. – 0.5 would be 97.5% and 2.5%
**distribution** [str] The distribution from which random points are sampled – uniform or poisson

**lowerbound** [float] The lower bound at which the G-function is computed. (Default 0).

**upperbound** [float] The upper bound at which the G-function is computed. Defaults to the maximum observed nearest neighbor distance.

**Returns**


**Examples**

```python
>>> import pysal.explore.spaghetti as spgh
>>> ntw = spgh.Network(in_data=examples.get_path('streets.shp'))
>>> pt_str = 'crimes'
>>> in_data = examples.get_path('{}').format(pt_str)
>>> ntw.snapobservations(in_data, pt_str, attribute=True)
>>> crimes = ntw.pointpatterns['crimes']
>>> sim = ntw.simulate_observations(crimes.npoints)
>>> gres = ntw.NetworkG(crimes, permutations=5, nsteps=10)
>>> gres.lowerenvelope.shape[0]
10
```

**pysal.explore.spaghetti.Network.NetworkK**

Network.K(pointpattern, nsteps=10, permutations=99, threshold=0.5, distribution='uniform', lowerbound=None, upperbound=None)

Computes a network constrained K-Function

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsteps</td>
<td>[int] The number of steps at which the count of the nearest neighbors is computed.</td>
</tr>
<tr>
<td>permutations</td>
<td>[int] The number of permutations to perform (default 99).</td>
</tr>
<tr>
<td>threshold</td>
<td>[float] The level at which significance is computed. – 0.5 would be 97.5% and 2.5%</td>
</tr>
<tr>
<td>distribution</td>
<td>[str] The distribution from which random points are sampled – uniform or poisson</td>
</tr>
<tr>
<td>lowerbound</td>
<td>[float] The lower bound at which the K-function is computed. (Default 0).</td>
</tr>
<tr>
<td>upperbound</td>
<td>[float] The upper bound at which the K-function is computed. Defaults to the maximum observed nearest neighbor distance.</td>
</tr>
</tbody>
</table>

**Returns**


**Examples**
```python
>>> import pysal.explore.spaghetti as spgh
>>> ntw = spgh.Network(in_data=examples.get_path('streets.shp'))
>>> pt_str = 'crimes'
>>> in_data = examples.get_path('{}.shp'.format(pt_str))
>>> ntw.snapobservations(in_data, pt_str, attribute=True)
>>> crimes = ntw.pointpatterns['crimes']
>>> sim = ntw.simulate_observations(crimes.npoints)
>>> kres = ntw.NetworkK(crimes, permutations=5, nsteps=10)
>>> kres.lowerenvelope.shape[0]
10
```

**pysal.explore.spaghetti.Network.allneighbordistances**

Network.allneighbordistances(sourcepattern, destpattern=None, fill_diagonal=None, n_processes=None, gen_tree=False, snap_dist=False)

Compute either all distances between i and j in a single point pattern or all distances between each i from a source pattern and all j from a destination pattern.

**Parameters**

- **sourcepattern** [str or spaghetti.network.PointPattern] The key of a point pattern snapped to the network OR the full spaghetti.network.PointPattern object.

- **destpattern** [str] (Optional) The key of a point pattern snapped to the network OR the full spaghetti.network.PointPattern object.

- **fill_diagonal** [float, int] (Optional) Fill the diagonal of the cost matrix. Default in None and will populate the diagonal with numpy.nan Do not declare a destpattern for a custom fill_diagonal.

- **n_processes** [int, str] (Optional) Specify the number of cores to utilize. Default is 1 core. Use (int) to specify an exact number or cores. Use (“all”) to request all available cores.

- **gen_tree** [bool] rebuild shortest path {True}, or skip {False}

- **snap_dist** [bool] include the distance from the original location to the snapped location along the network. Default is False.

**Returns**

- **nearest** [numpy.ndarray] An array of shape (n,n) storing distances between all points.

- **tree_nearest** [dict] nearest network node to point pattern node shortest path lookup

**Examples**

```python
>>> import pysal.explore.spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.snapobservations(examples.get_path('crimes.shp'),...
... 'crimes',...
... attribute=True)

>>> s2s_dist = ntw.allneighbordistances('crimes')
>>> s2s_dist[0,0], s2s_dist[1,0]
(nan, 3105.189475447081)
```
pysal Documentation, Release 2.0.0

```python
>>> ntw.snapobservations(examples.get_path('schools.shp'),
    ...                           'schools',
    ...   attribute=False)
```

```python
>>> s2d_dist = ntw.allneighbordistances('crimes',
    ...   destpattern='schools')
>>> s2d_dist[0,0], s2d_dist[1,0]
(4520.72353741989, 6340.422971967316)

>>> s2d_dist, tree = ntw.allneighbordistances('schools',
    ...   gen_tree=True)
>>> tree[(6, 7)]
(173, 64)
```

**pysal.explore.spaghetti.Network.compute_distance_to_nodes**

Network.compute_distance_to_nodes(x, y, edge)

Given an observation on a network edge, return the distance to the two nodes that bound that end.

**Parameters**

- **x** [float] x-coordinate of the snapped point.
- **y** [float] y-coordinate of the snapped point.
- **edge** [tuple] (node0, node1) representation of the network edge.

**Returns**

- **d1** [float] The distance to node0. Always the node with the lesser id.
- **d2** [float] The distance to node1. Always the node with the greater id.

**pysal.explore.spaghetti.Network.compute_snap_dist**

Network.compute_snap_dist(pattern, idx)

Given an observation snapped to a network edge, calculate the distance from the original location to the snapped location.

**Parameters**

- **pattern** [spaghetti.network.PointPattern] point pattern object
- **idx** [int] point id

**pysal.explore.spaghetti.Network.contiguityweights**

Network.contiguityweights(graph=True, weightings=None)

Create a contiguity based W object.

**Parameters**

- **graph** [bool] {True, False} controls whether the W is generated using the spatial representation or the graph representation.
- **weightings** [dict] Dict of lists of weightings for each edge.
### Returns

\[ W \] A PySAL W Object representing the binary adjacency of the network.

### Examples

Instantiate an instance of a network.

```python
>>> import pysal.explore.spaghetti as spgh
>>> from pysal.lib import examples
>>> import pysal.explore.esda
>>> import numpy as np

>>> ntw = spgh.Network(examples.get_path('streets.shp'))
```

Snap point observations to the network with attribute information.

```python
>>> ntw.snapobservations(examples.get_path('crimes.shp'), ...
... 'crimes', attribute=True)
```

Find counts per network edge.

```python
>>> counts = ntw.count_per_edge(ntw.pointpatterns['crimes'] ...
... .obs_to_edge, graph=False)
>>> counts[(50, 165)]
4
```

Create a contiguity based W object.

```python
>>> w = ntw.contiguityweights(graph=False)
```

Using the W object, access to ESDA functionality is provided. First, a vector of attributes is created for all edges with observations.

```python
>>> w = ntw.contiguityweights(graph=False)
>>> edges = w.neighbors.keys()
>>> y = np.zeros(len(edges))
>>> for i, e in enumerate(edges):
... if e in counts.keys():
... y[i] = counts[e]
>>> y[3]
3.0
```

Next, a standard call of Moran is made and the result placed into `res`.

```python
>>> res = pysal.explore.esda.moran.Moran(y, w, permutations=99)
>>> type(res)
<class 'pysal.explore.esda.moran.Moran'>
```

---

**pysal.explore.spaghetti.Network.count_per_edge**

Network.\texttt{count\_per\_edge}(\textit{obs\_on\_network}, \textit{graph=True})

Compute the counts per edge.

**Parameters**
**obs_on_network**  [dict] Dict of observations on the network. {edge: {pt_id: (coords)}} or {edge: [(coord), (coord), (coord)]}

**Returns**

**counts**  [dict] {edge: count}

**Examples**

Note that this passes the obs_to_edge attribute of a point pattern snapped to the network.

```python
>>> import pysal.explore.spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
... ntw.snapobservations(examples.get_path('crimes.shp'),
... 'crimes',
... attribute=True)
... counts = ntw.count_per_edge(ntw.pointpatterns['crimes'].obs_to_edge, graph=False)
... counts[(140, 142)]
10
... s = sum([v for v in list(counts.values())])
... s
287
```

**pysal.explore.spaghetti.Network.distancebandweights**

Network.distancebandweights (threshold, n_processes=None, gen_tree=False)

Create distance based weights.

**Parameters**

- **threshold**  [float] Distance threshold value.
- **n_processes**  [int, str] (Optional) Specify the number of cores to utilize. Default is 1 core. Use (int) to specify an exact number or cores. Use (“all”) to request all available cores.
- **gen_tree**  [bool] rebuild shortest path {True}, or skip {False}

**Returns**

**w**  [pysal.lib.weights.weights.W] A PySAL W Object representing the binary adjacency of the network.

**Examples**

```python
>>> import pysal.explore.spaghetti as spgh
>>> streets_file = examples.get_path('streets.shp')
>>> ntw = spgh.Network(in_data=streets_file)
>>> w = ntw.distancebandweights(threshold=500)
>>> w.n
230
>>> w.histogram
[(1, 22), (2, 58), (3, 63), (4, 40), (5, 36), (6, 3), (7, 5), (8, 3)]
```
pysal.explore.spaghetti.Network.enum_links_node

Network.enum_links_node(v0)
Returns the edges (links) around node.

Parameters
v0 [int] Node id

Returns
links [list] List of tuple edges adjacent to the node.

Examples

```python
>>> import pysal.explore.spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.enum_links_node(24)
[(24, 48), (24, 25), (24, 26)]
```

pysal.explore.spaghetti.Network.extractgraph

Network.extractgraph()
Using the existing network representation, create a graph-theoretic representation by removing all nodes with a neighbor incidence of two (non-articulation points). That is, we assume these nodes are bridges between nodes with higher incidence.

pysal.explore.spaghetti.Network.loadnetwork

static Network.loadnetwork(filename)
Load a network from a binary file saved on disk.

Parameters
filename [str] The filename where the network should be saved.

Returns
self [spaghetti.Network] spaghetti Network object

pysal.explore.spaghetti.Network.nearestneighbordistances

Network.nearestneighbordistances(sourcepattern, destpattern=None, n_processes=None, gen_tree=False, all_dists=None, snap_dist=False, keep_zero_dist=True)
Compute the interpattern nearest neighbor distances or the intrapattern nearest neighbor distances between a source pattern and a destination pattern.

Parameters
sourcepattern [str] The key of a point pattern snapped to the network.

destpattern [str] (Optional) The key of a point pattern snapped to the network.

n_processes [int, str] (Optional) Specify the number of cores to utilize. Default is 1 core. Use (int) to specify an exact number or cores. Use (“all”) to request all available cores.
**gen_tree** [bool] rebuild shortest path {True}, or skip {False}

**all_dists** [numpy.ndarray] An array of shape (n,n) storing distances between all points.

**snap_dist** [bool] include the distance from the original location to the snapped location along the network. Default is False.

**keep_zero_dist** [bool] Include zero values in minimum distance (True) or exclude (False). Default is True. If the source pattern is the same as the destination pattern the diagonal is filled with nans

**Returns**

**nearest** [dict] key is source point id, value is tuple of list containing nearest destination point ids and distance.

**Examples**

```python
>>> import pysal.explore.spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.snapobservations(examples.get_path('crimes.shp'), 'crimes')
>>> nn = ntw.nearestneighbor distances('crimes', keep_zero_dist=True)
>>> nn[11], nn[18]
([(18, 19), 165.33982412719126], ([19], 0.0))

>>> nn = ntw.nearestneighbor distances('crimes', keep_zero_dist=False)
>>> nn[11], nn[18]
([(18, 19), 165.33982412719126], ([11], 165.33982412719126))
```

**pysal.explore.spaghetti.Network.node_distance_matrix**

Network.node_distance_matrix(n_processes, gen_tree=False)

Called from within allneighbor distances(), nearestneighbor distances(), and distancebandweights().

**Parameters**

- **n_processes** [int] cpu cores for multiprocessing.
- **gen_tree** [bool] rebuild shortest path {True}, or skip {False}

**pysal.explore.spaghetti.Network.savenetwork**

Network.savenetwork(filename)

Save a network to disk as a binary file.

**Parameters**

- **filename** [str] The filename where the network should be saved. This should be a full path or it will be save in the current directory.

**Examples**

```python
>>> import pysal.explore.spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.savenetwork('mynetwork.pkl')
```
pysal.explore.spaghetti.Network.segment_edges

Network.segment_edges(distance)

Segment all of the edges in the network at either a fixed distance or a fixed number of segments.

Parameters

distance [float] The distance at which edges are split.

Returns


Examples

```python
>>> import pysal.explore.spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> n200 = ntw.segment_edges(200.0)
>>> len(n200.edges)
688
```

pysal.explore.spaghetti.Network.simulate_observations

Network.simulate_observations(count, distribution='uniform')

Generate a simulated point pattern on the network.

Parameters

count [int] The number of points to create or mean of the distribution if not ‘uniform’.

distribution [str] {'uniform', 'poisson'} distribution of random points. If poisson if the distribution the poisson is calculated from half the total network length.

Returns

random_pts [dict] Keys are the edge tuple. Values are lists of new point coordinates.

Examples

```python
>>> import pysal.explore.spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.snapobservations(examples.get_path('crimes.shp'),...
...'crimes',
...attribute=True)
>>> npts = ntw.pointpatterns['crimes'].npoints
>>> sim = ntw.simulate_observations(npts)
>>> isinstance(sim, spgh.network.SimulatedPointPattern)
True
```
pysal.explore.spaghetti.Network.snapobservations

Network snapobservations (in_data, name, idvariable=None, attribute=None)

Snap a point pattern shapefile to this network object. The point pattern is stored in the network.pointpattern['key'] attribute of the network object.

Parameters

in_data [geopandas.GeoDataFrame or str] The input geographic data. Either (1) a path to a shapefile (str); or (2) a geopandas.GeoDataFrame.

name [str] Name to be assigned to the point dataset.

idvariable [str] Column name to be used as ID variable.

attribute [bool] Defines whether attributes should be extracted. True for attribute extraction. False for no attribute extraction.

Examples

```python
>>> import pysal.explore.spaghetti as spgh

streets_file = examples.get_path('streets.shp')

ntw = spgh.Network(in_data=streets_file)

pt_str = 'crimes'

in_data = examples.get_path('{}\shp'.format(pt_str))

ntw.snapobservations(in_data, pt_str, attribute=True)

ntw.pointpatterns[pt_str].npoints
```

__init__ (in_data=None, node_sig=11, unique_segs=True, extractgraph=True)

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetworkF(pointpattern[, nsteps, ...])</td>
<td>Computes a network constrained F-Function</td>
</tr>
<tr>
<td>NetworkG(pointpattern[, nsteps, ...])</td>
<td>Computes a network constrained G-Function</td>
</tr>
<tr>
<td>NetworkK(pointpattern[, nsteps, ...])</td>
<td>Computes a network constrained K-Function</td>
</tr>
<tr>
<td><strong>init</strong> ([in_data, node_sig, unique_segs, ...])</td>
<td></td>
</tr>
<tr>
<td>allneighbordistances(sourcepattern[, ...])</td>
<td>Compute either all distances between i and j in a single point pattern or all distances between each i from a source pattern and all j from a destination pattern.</td>
</tr>
<tr>
<td>compute_distance_to_nodes(x, y, edge)</td>
<td>Given an observation on a network edge, return the distance to the two nodes that bound that end.</td>
</tr>
<tr>
<td>compute_snap_dist(pattern, idx)</td>
<td>Given an observation snapped to a network edge, calculate the distance from the original location to the snapped location.</td>
</tr>
<tr>
<td>contiguityweights([graph, weightings])</td>
<td>Create a contiguity based W object.</td>
</tr>
<tr>
<td>count_per_edge(obs_on_network[, graph])</td>
<td>Compute the counts per edge.</td>
</tr>
<tr>
<td>distancebandweights(threshold[, n_proccess, ...])</td>
<td>Create distance based weights.</td>
</tr>
<tr>
<td>enum_links_node(v0)</td>
<td>Returns the edges (links) around node.</td>
</tr>
</tbody>
</table>
extractgraph() Using the existing network representation, create a graph-theoretic representation by removing all nodes with a neighbor incidence of two (non-articulation points).

loadnetwork(filename) Load a network from a binary file saved on disk.

nearestneighborDistances(sourcepattern[, ...]) Compute the interpattern nearest neighbor distances or the intrapattern nearest neighbor distances between a source pattern and a destination pattern.

node_distance_matrix(n_processes[, gen_tree]) Called from within allneighborDistances(), nearestneighborDistances(), and distancebandweights().

savenetwork(filename) Save a network to disk as a binary file.

segment_edges(distance) Segment all of the edges in the network at either a

simulate_observations(count[, distribution]) Generate a simulated point pattern on the network.

snapobservations(in_data, name[, ...]) Snap a point pattern shapefile to this network object.

spaghetti.NetworkBase

spaghetti.NetworkBase(ntw, pointpattern[, ...]) Base object for performing network analysis on a spaghetti.Network object.

spaghetti.NetworkBase.computeenvelope() compute upper and lower bounds of envelope

spaghetti.NetworkBase.setbounds(nearest) set upper and lower bounds

spaghetti.NetworkBase.validatedistribution() ensure statistical distribution is supported

pysal.explore.spaghetti.NetworkBase

class pysal.explore.spaghetti.NetworkBase(ntw, pointpattern, nsteps=10, permutations=99, threshold=0.5, distribution='poisson', lowerbound=None, upperbound=None)

Base object for performing network analysis on a spaghetti.Network object.

Parameters


nsteps [int] The number of steps at which the count of the nearest neighbors is computed.

permutations [int] The number of permutations to perform (default 99).

threshold [float] The level at which significance is computed. – 0.5 would be 97.5% and 2.5%

distribution [str] The distribution from which random points are sampled – uniform or poisson

lowerbound [float] The lower bound at which the function is computed. (Default 0).

upperbound [float] The upper bound at which the function is computed. Defaults to the maximum observed nearest neighbor distance.

Attributes

sim [numpy.ndarray] simulated distance matrix

npts [int] pointpattern.npoints
xaxis [numpy.ndarray] observed x-axis of values
observed [numpy.ndarray] observed y-axis of values

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>computeenvelope()</td>
<td>compute upper and lower bounds of envelope</td>
</tr>
<tr>
<td>setbounds(nearest)</td>
<td>set upper and lower bounds</td>
</tr>
<tr>
<td>validatedistribution()</td>
<td>ensure statistical distribution is supported</td>
</tr>
</tbody>
</table>

pysal.explore.spaghetti.NetworkBase.computeenvelope

NetworkBase.computeenvelope()
compute upper and lower bounds of envelope

pysal.explore.spaghetti.NetworkBase.setbounds

NetworkBase.setbounds(nearest)
set upper and lower bounds

pysal.explore.spaghetti.NetworkBase.validatedistribution

NetworkBase.validatedistribution()
ensure statistical distribution is supported

__init__(ntw, pointpattern[, nsteps, ...]) Initialize self. See help(type(self)) for accurate signature.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(ntw, pointpattern[, nsteps, ...])</td>
<td>Initialize self.</td>
</tr>
<tr>
<td>computeenvelope()</td>
<td>compute upper and lower bounds of envelope</td>
</tr>
<tr>
<td>setbounds(nearest)</td>
<td>set upper and lower bounds</td>
</tr>
<tr>
<td>validatedistribution()</td>
<td>ensure statistical distribution is supported</td>
</tr>
</tbody>
</table>

spaghetti.NetworkF

spaghetti.NetworkF(ntw, pointpattern[...]) Compute a network constrained F statistic.
spaghetti.NetworkF.computeenvelope() compute upper and lower bounds of envelope
spaghetti.NetworkF.setbounds(nearest) set upper and lower bounds
spaghetti.NetworkF.validatedistribution() ensure statistical distribution is supported
spaghetti.NetworkF.computeobserved() compute the observed nearest and simulated nearest
spaghetti.NetworkF.computepermutations() compute permutations of the nearest
pysal.explore.spaghetti.NetworkF

class pysal.explore.spaghetti.NetworkF(ntw, pointpattern, nsteps=10, permutations=99, threshold=0.5, distribution='poisson', lowerbound=None, upperbound=None)

Compute a network constrained F statistic. This requires the capability to compute a distance matrix between two point patterns. In this case one will be observed and one will be simulated.

Attributes

fsim  [spaghetti.network.SimulatedPointPattern] simulated point pattern of self.npts points

Methods

table

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>computeenvelope()</td>
<td>compute upper and lower bounds of envelope</td>
</tr>
<tr>
<td>computeobserved()</td>
<td>compute the observed nearest and simulated nearest</td>
</tr>
<tr>
<td>computepermutations()</td>
<td>compute permutations of the nearest</td>
</tr>
<tr>
<td>setbounds(nearest)</td>
<td>set upper and lower bounds</td>
</tr>
<tr>
<td>validatedistribution()</td>
<td>ensure statistical distribution is supported</td>
</tr>
</tbody>
</table>

pysal.explore.spaghetti.NetworkF.computeenvelope

NetworkF.computeenvelope()  
compute upper and lower bounds of envelope

pysal.explore.spaghetti.NetworkF.computeobserved

NetworkF.computeobserved()  
compute the observed nearest and simulated nearest

pysal.explore.spaghetti.NetworkF.computepermutations

NetworkF.computepermutations()  
compute permutations of the nearest

pysal.explore.spaghetti.NetworkF.setbounds

NetworkF.setbounds(nearest)  
set upper and lower bounds

pysal.explore.spaghetti.NetworkF.validatedistribution

NetworkF.validatedistribution()  
ensure statistical distribution is supported

__init__(ntw, pointpattern, nsteps=10, permutations=99, threshold=0.5, distribution='poisson', lowerbound=None, upperbound=None)

Initialize self. See help(type(self)) for accurate signature.
Methods

```python
__init__ (ntw, pointpattern[, nsteps, ...]) Initialize self.
computeenvelope() compute upper and lower bounds of envelope
computeobserved() compute the observed nearest and simulated nearest
computepermutations() compute permutations of the nearest
setbounds(nearest) set upper and lower bounds
validatedistribution() ensure statistical distribution is supported
```

### spaghetti.NetworkG

```python
spaghetti.NetworkG(ntw, pointpattern[, ...]) Compute a network constrained G statistic.
spaghetti.NetworkG.computeenvelope() compute upper and lower bounds of envelope
spaghetti.NetworkG.setbounds(nearest) set upper and lower bounds
spaghetti.NetworkG.validatedistribution() ensure statistical distribution is supported
spaghetti.NetworkG.computeobserved() compute the observed nearest
spaghetti.NetworkG.computepermutations() compute permutations of the nearest
```

### pysal.explore.spaghetti.NetworkG

#### pysal.explore.spaghetti.NetworkG.computeenvelope

```python
NetworkG.computeenvelope() compute upper and lower bounds of envelope
```

#### pysal.explore.spaghetti.NetworkG.computeobserved

```python
NetworkG.computeobserved() compute the observed nearest
```
**pysal.explore.spaghetti.NetworkG.computepermutations**

NetworkG.computepermutations()
compute permutations of the nearest

**pysal.explore.spaghetti.NetworkG.setbounds**

NetworkG.setbounds(nearest)
set upper and lower bounds

**pysal.explore.spaghetti.NetworkG.validatedistribution**

NetworkG.validatedistribution()
ensure statistical distribution is supported

```python
__init__(ntw, pointpattern, nsteps=10, permutations=99, threshold=0.5, distribution='poisson', lowerbound=None, upperbound=None)
```
Initialize self. See help(type(self)) for accurate signature.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Initialize self.</td>
</tr>
<tr>
<td>computeenvelope()</td>
<td>compute upper and lower bounds of envelope</td>
</tr>
<tr>
<td>computeobserved()</td>
<td>compute the observed nearest</td>
</tr>
<tr>
<td>computepermutations()</td>
<td>compute permutations of the nearest</td>
</tr>
<tr>
<td>setbounds(nearest)</td>
<td>set upper and lower bounds</td>
</tr>
<tr>
<td>validatedistribution()</td>
<td>ensure statistical distribution is supported</td>
</tr>
</tbody>
</table>

**spaghetti.NetworkK**

```python
spaghetti.NetworkK(ntw, pointpattern[, ...])
```
Compute a network constrained K statistic. This requires the capability to compute a distance matrix between two point patterns. In this case one will be observed and one will be simulated.

1.2. API Reference 161
Notes

Based on [OY01].

Attributes

- \texttt{lam} [float] lambda value

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{computeenvelope()}</td>
<td>compute upper and lower bounds of envelope</td>
</tr>
<tr>
<td>\texttt{computeobserved()}</td>
<td>compute the observed nearest</td>
</tr>
<tr>
<td>\texttt{computepermutations()}</td>
<td>compute permutations of the nearest</td>
</tr>
<tr>
<td>\texttt{setbounds(nearest)}</td>
<td>set upper and lower bounds</td>
</tr>
<tr>
<td>\texttt{validatedistribution()}</td>
<td>ensure statistical distribution is supported</td>
</tr>
</tbody>
</table>

\texttt{pysal.explore.spaghetti.NetworkK.computeenvelope}

\texttt{NetworkK.\texttt{computeenvelope}()}  
compute upper and lower bounds of envelope

\texttt{pysal.explore.spaghetti.NetworkK.computeobserved}

\texttt{NetworkK.\texttt{computeobserved}()}  
compute the observed nearest

\texttt{pysal.explore.spaghetti.NetworkK.computepermutations}

\texttt{NetworkK.\texttt{computepermutations}()}  
compute permutations of the nearest

\texttt{pysal.explore.spaghetti.NetworkK.setbounds}

\texttt{NetworkK.\texttt{setbounds(nearest)}}  
set upper and lower bounds

\texttt{pysal.explore.spaghetti.NetworkK.validatedistribution}

\texttt{NetworkK.\texttt{validatedistribution}()}  
ensure statistical distribution is supported

\texttt{\_\_init\_\_}  
Initialize self. See help(type(self)) for accurate signature.
spaghetti.PointPattern

spaghetti.PointPattern([in_data,...]) A stub point pattern class used to store a point pattern.

pysal.explore.spaghetti.PointPattern

class pysal.explore.spaghetti.PointPattern(in_data=None, idvariable=None, attribute=False)
    A stub point pattern class used to store a point pattern. This class is monkey patched with network specific attributes when the points are snapped to a network. In the future this class may be replaced with a generic point pattern class.

Parameters

in_data [geopandas.GeoDataFrame or str] The input geographic data. Either (1) a path to a shapefile (str); or (2) a geopandas.GeoDataFrame.

idvariable [str] Field in the shapefile to use as an id variable.

attribute [bool] {False, True} A flag to indicate whether all attributes are tagged to this class.

Attributes

points [dict] Keys are the point ids (int). Values are the x,y coordinates (tuple).

npoints [int] The number of points.

obs_to_edge [dict] Keys are edge ids (tuple). Values are snapped point information (dict). Within the snapped point information (dict) keys are observation id (int), and values are snapped coordinates.

obs_to_node [list] list of incident network nodes to snapped observation points converted from default_dict. Originally in the form of paired left/right nearest network nodes - {netnode1: obs_id1, netnode2: obs_id1, netnode1: obs_id2... netnode1: obs_idn}, then simplified to a list in the form - [netnode1, netnode2, netnode1, netnode2, ...].

dist_to_node [dict] Keys are observations ids (int). Values are distance lookup (dict). Within distance lookup (dict) keys are the two incident nodes of the edge and values are distance to each of those edges.

snapped_coordinates [dict] Keys are the point ids (int). Values are the snapped x,y coordinates (tuple).

snap_dist [bool] include the distance from the original location to the snapped location along the network. Default is False.

__init__ (in_data=None, idvariable=None, attribute=False)
Initialize self. See help(type(self)) for accurate signature.
Methods

```
__init__([in_data, idvariable, attribute]) Initialize self.
```

**spaghetti.SimulatedPointPattern**

```
spaghetti.SimulatedPointPattern() Struct style class to mirror the Point Pattern Class.
```

**pysal.explore.spaghetti.SimulatedPointPattern**

class pysal.explore.spaghetti.SimulatedPointPattern
Struct style class to mirror the Point Pattern Class. If the PointPattern class has methods, it might make sense to make this a child of that class. This class is not intended to be used by the external user.

**Attributes**

- **npoints** [int] The number of points.
- **obs_to_edge** [dict] Keys are edge ids (tuple). Values are snapped point information (dict). Within the snapped point information (dict) keys are observation id (int), and values are snapped coordinates.
- **obs_to_node** [list] list of incident network nodes to snapped observation points converted from default_dict. Originally in the form of paired left/right nearest network nodes - {netnode1: obs_id1, netnode2: obs_id1, netnode1: obs_id2… netnode1: obs_idn}, then simplified to a list in the form - [netnode1, netnode2, netnode1, netnode2, …].
- **dist_to_node** [dict] Keys are observations ids (int). Values are distance lookup (dict). Within distance lookup (dict) keys are the two incident nodes of the edge and values are distance to each of those edges.
- **snapped_coordinates** [dict] Keys are the point ids (int). Values are the snapped x,y coordinates (tuple).

```
__init__()
Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

```
__init__() Initialize self.
```

**spaghetti**

```
spaghetti.compute_length(v0, v1) Compute the euclidean distance between two points.
spaghetti.dijkstra(ntw, cost, v0[, n]) Compute the shortest path between a start node and all other nodes in an origin-destination matrix.
spaghetti.dijkstra_mp(ntw_cost_node) Compute the shortest path between a start node and all other nodes in the web utilizing multiple cores upon request.
spaghetti.generatetree(pred) Rebuild the shortest path from root origin to destination
```

Continued on next page
Table 138 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spaghetti.get_neighbor_distances(ntw, v0, l)</td>
<td>Get distances to the nearest node neighbors along connecting edges.</td>
</tr>
<tr>
<td>spaghetti.snap_points_on_segments(points, ...)</td>
<td>Place points onto closest segment in a set of segments.</td>
</tr>
<tr>
<td>spaghetti.squared_distance_point_segment(...)</td>
<td>Find the squared distance between a point and a segment.</td>
</tr>
<tr>
<td>spaghetti.ffunction(nearest, lowerbound, ...)</td>
<td>Compute an F-Function</td>
</tr>
<tr>
<td>spaghetti.gfunction(nearest, lowerbound, ...)</td>
<td>Compute a G-Function</td>
</tr>
<tr>
<td>spaghetti.kfunction(nearest, upperbound, ...)</td>
<td>Compute a K-Function</td>
</tr>
</tbody>
</table>

**pysal.explore.spaghetti.compute_length**

**pysal.explore.spaghetti.compute_length**(v0, v1)

Compute the euclidean distance between two points.

**Parameters**

- **v0** [tuple] sequence in the form x, y
- **vq** [tuple] sequence in the form x, y

**Returns**

- **euc_dist** [float] Euclidean distance

**Examples**

```python
>>> import pysal.explore.spaghetti as spgh
>>> point1, point2 = (0,0), (1,1)
>>> spgh.util.compute_length(point1, point2)
1.4142135623730951
```

**pysal.explore.spaghetti.dijkstra**

**pysal.explore.spaghetti.dijkstra**(ntw, cost, v0, n=inf)

Compute the shortest path between a start node and all other nodes in an origin-destination matrix.

**Parameters**

- **cost** [dict] key is tuple (start node, end node); value is float. Cost per edge to travel, e.g. distance.
- **v0** [int] Start node ID
- **n** [float] integer break point to stop iteration and return n neighbors. Default is (‘inf’).

**Returns**

- **distance** [list] List of distances from node to all other nodes.
- **pred** [list] List of preceeding nodes for traversal route.
Notes

Based on [Dij59].

Examples

```python
>>> import pysal.explore.spaghetti as spgh
>>> from pysal.lib import examples

ntw = spgh.Network(examples.get_path('streets.shp'))
distance, pred = spgh.util.dijkstra(ntw, ntw.edge_lengths, 0)
round(distance[196], 4)
5505.6682
>>> pred[196]
133
```

pysal.explore.spaghetti.dijkstra_mp

pysal.explore.spaghetti.dijkstra_mp(ntw_cost_node)

Compute the shortest path between a start node and all other nodes in the web utilizing multiple cores upon request.

Parameters

- **ntw_cost_node** [tuple] tuple of arguments to pass into dijkstra (1) ntw - spaghetti.Network; spaghetti Network object; (2) cost - dict; key is tuple (start node, end node); value is float - Cost per edge to travel, e.g. distance; (3) node - int; Start node ID

Returns

- **distance** [list] List of distances from node to all other nodes.
- **pred** [list] List of preceeding nodes for traversal route.

Notes

Based on [Dij59].

Examples

```python
>>> import pysal.explore.spaghetti as spgh
>>> from pysal.lib import examples

ntw = spgh.Network(examples.get_path('streets.shp'))
distance, pred = spgh.util.dijkstra(ntw, ntw.edge_lengths, 0)
round(distance[196], 4)
5505.6682
>>> pred[196]
133
```

pysal.explore.spaghetti.generatetree

pysal.explore.spaghetti.generatetree(pred)

Rebuild the shortest path from root origin to destination
Parameters

pred [list] List of preceeding nodes for traversal route.

Returns

tree [dict] key is root origin; value is root origin to destination.

Examples

```python
>>> import pysal.explore.spaghetti as spgh
>>> from pysal.lib import examples

>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> distance, pred = spgh.util.dijkstra(ntw, ntw.edge_lengths, 0)
>>> tree = spgh.util.generatedtree(pred)
>>> tree[3]
[23, 22, 20, 19, 170, 2, 0]
```

**pysal.explore.spaghetti.get_neighbor_distances**

Get distances to the nearest node neighbors along connecting edges.

Parameters


v0 [int] Node id

l [dict] key is tuple (start node, end node); value is float. Cost per edge to travel, e.g. distance.

Returns

neighbors [dict] key is int (node id); value is float (distance)

Examples

```python
>>> import pysal.explore.spaghetti as spgh
>>> from pysal.lib import examples

>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> neighs = spgh.util.get_neighbor_distances(ntw, 0, ntw.edge_lengths)
>>> neighs[1]
102.62353453439829
```

**pysal.explore.spaghetti.snap_points_on_segments**

Place points onto closet segment in a set of segments

Parameters

points [dict] Point id as key and (x,y) coordinate as value

segments [list] Elements are of type pysal.lib.cg.shapes.Chain ** Note ** each element is a segment represented as a chain with one head and one tail node in other words one link only.
Returns

p2s [dict] key [point id (see points in arguments)]; value [a 2-tuple ((head, tail), point) where (head, tail) is the target segment, and point is the snapped location on the segment.

Examples

```python
>>> import pysal.explore.spaghetti as spgh
>>> from pysal.lib.cg.shapes import Point, Chain
>>> points = {0: Point((1, 1))}
>>> segments = [Chain([Point((0, 0)), Point((2, 0))])]
>>> spgh.util.snap_points_on_segments(points, segments)
{0: ([[(0.0, 0.0), (2.0, 0.0)]], array([1., 0.]))}
```

```python
pysal.explore.spaghetti.squared_distance_point_segment

pysal.explore.spaghetti.squared_distance_point_segment(point, segment)

Find the squared distance between a point and a segment.

Parameters

point [tuple] point coordinates (x,y)
segment [list] List of 2 point coordinate tuples [(x0,y0), (x1,y1)].

Returns

sqd [float] distance squared between point and segment
nearp [numpy.ndarray] array of (xb, yb); the nearest point on the segment

Examples

```python
>>> import pysal.explore.spaghetti as spgh
>>> point, segment = (1, 1), ((0, 0), (2, 0))
>>> spgh.util.squared_distance_point_segment(point, segment)
(1.0, array([1., 0.]))
```

```python
pysal.explore.spaghetti.ffunction

pysal.explore.spaghetti.ffunction(nearest, lowerbound, upperbound, npts, nsteps=10)

Compute an F-Function

Parameters

lowerbound [int or float] The starting value of the sequence.
upperbound [int or float] The end value of the sequence.
npts [int] pointpattern.npoints
nsteps [int] The number of distance bands. Default is 10. Must be non-negative.

Returns

168 Chapter 1. PySAL Components
pysal.explore.spaghetti.gfunction

Computes a G-Function

Parameters

- **nearest** [numpy.ndarray] A vector of nearest neighbor distances.
- **lowerbound** [int or float] The starting value of the sequence.
- **upperbound** [int or float] The end value of the sequence.
- **nsteps** [int] The number of distance bands. Default is 10. Must be non-negative.

Returns

- **x** [numpy.ndarray] x-axis of values
- **y** [numpy.ndarray] y-axis of values

pysal.explore.spaghetti.kfunction

Computes a K-Function

Parameters

- **nearest** [numpy.ndarray] A vector of nearest neighbor distances.
- **upperbound** [int or float] The end value of the sequence.
- **intensity** [float] lambda value
- **nsteps** [int] The number of distance bands. Default is 10. Must be non-negative.

Returns

- **x** [numpy.ndarray] x-axis of values
- **y** [numpy.ndarray] y-axis of values

1.2.3 pysal.viz: Geovisualization

pysal.viz.mapclassify: Choropleth map classification

Classifiers

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapclassify.Box_Plot(y[, hinge])</td>
<td>Box_Plot Map Classification</td>
</tr>
<tr>
<td>mapclassify.Equal_Interval(y[, k])</td>
<td>Equal Interval Classification</td>
</tr>
<tr>
<td>mapclassify.Fisher_Jenks(y[, k])</td>
<td>Fisher Jenks optimal classifier - mean based</td>
</tr>
<tr>
<td>mapclassify.Fisher_Jenks_Sampled(y[, k])</td>
<td>Fisher Jenks optimal classifier - mean based using random sample</td>
</tr>
</tbody>
</table>

Continued on next page
Table 139 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mapclassify.HeadTail_Breaks(y)</code></td>
<td>Head/tail Breaks Map Classification for Heavy-tailed Distributions</td>
</tr>
<tr>
<td><code>mapclassify.Jenks_Caspall(y[, k])</code></td>
<td>Jenks Caspall Map Classification</td>
</tr>
<tr>
<td><code>mapclassify.Jenks_Caspall_Forced(y[, k])</code></td>
<td>Jenks Caspall Map Classification with forced movements</td>
</tr>
<tr>
<td><code>mapclassify.Jenks_Caspall_Sampled(y[, k, pct])</code></td>
<td>Jenks Caspall Map Classification using a random sample</td>
</tr>
<tr>
<td><code>mapclassify.Max_P_Classifier(y[, k, initial])</code></td>
<td>Max_P Map Classification</td>
</tr>
<tr>
<td><code>mapclassify.Maximum_Breaks(y[, k, mindiff])</code></td>
<td>Maximum Breaks Map Classification</td>
</tr>
<tr>
<td><code>mapclassify.Natural_Breaks(y[, k, initial])</code></td>
<td>Natural Breaks Map Classification</td>
</tr>
<tr>
<td><code>mapclassify.Quantiles(y[, k])</code></td>
<td>Quantile Map Classification</td>
</tr>
<tr>
<td><code>mapclassify.Percentiles(y[, pct])</code></td>
<td>Percentiles Map Classification</td>
</tr>
<tr>
<td><code>mapclassify.Std_Mean(y[, multiples])</code></td>
<td>Standard Deviation and Mean Map Classification</td>
</tr>
<tr>
<td><code>mapclassify.User_Defined(y, bins)</code></td>
<td>User Specified Binning</td>
</tr>
</tbody>
</table>

**pysal.viz.mapclassify.Box_Plot**

**class** `pysal.viz.mapclassify.Box_Plot(y, hinge=1.5)`

Box_Plot Map Classification

**Parameters**

- `y` [array] attribute to classify
- `hinge` [float] multiplier for IQR

**Notes**

The bins are set as follows:

```python
bins[0] = q[0]-hinge*IQR
bins[1] = q[0]
bins[5] = inf (see Notes)
```

where `q` is an array of the first three quartiles of `y` and IQR=q[2]-q[0]

If q[2]+hinge*IQR > max(y) there will only be 5 classes and no high outliers, otherwise, there will be 6 classes and at least one high outlier.

**Examples**

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> bp = mc.Box_Plot(cal)
>>> bp.bins
array([-5.287625e+01, 2.567500e+00, 9.365000e+00, 3.953000e+01,
       9.497375e+01, 4.111450e+03])
>>> bp.counts
array([ 0, 15, 14, 14, 6, 9])
```

(continues on next page)
>>> bp.high_outlier_ids
array([ 0,  6, 18, 29, 33, 36, 37, 40, 42])

```python
cal[bp.high_outlier_ids].values
array([ 329.92, 181.27, 370.5 , 722.85, 192.05, 110.74, 4111.45,
       317.11, 264.93])
```

```python
bx = mc.Box_Plot(np.arange(100))
bx.bins
array([-49.5 , 24.75, 49.5 , 74.25, 148.5 ])
```

Attributes

- **yb** [array] (n,1), bin ids for observations
- **bins** [array] (n,1), the upper bounds of each class (monotonic)
- **k** [int] the number of classes
- **counts** [array] (k,1), the number of observations falling in each class
- **low_outlier_ids** [array] indices of observations that are low outliers
- **high_outlier_ids** [array] indices of observations that are high outliers

Methods

- **__call__**(arg, **kwargs)** This will allow the classifier to be called like it’s a function.
- **find_bin**(x)** Sort input or inputs according to the current bin estimate
- **get_adcm()** Absolute deviation around class median (ADCM).
- **get_gadf()** Goodness of absolute deviation of fit
- **get_tss()** Total sum of squares around class means
- **make**(arg, **kwargs)** Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.
- **update**(y, inplace)** Add data or change classification parameters.

- **__init__**(y, **kwargs)**

  Parameters

  - **y** [array (n,1)] attribute to classify
  - **hinge** [float] multiple of inter-quartile range (default=1.5)

Methods

- **__init__**(y, hinge)

  Parameters

  - **find_bin**(x)** Sort input or inputs according to the current bin estimate
  - **get_adcm()** Absolute deviation around class median (ADCM).
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_gadf()</td>
<td>Goodness of absolute deviation of fit</td>
</tr>
<tr>
<td>get_tss()</td>
<td>Total sum of squares around class means</td>
</tr>
<tr>
<td>make(*args, **kwargs)</td>
<td>Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.</td>
</tr>
<tr>
<td>update([y, inplace])</td>
<td>Add data or change classification parameters.</td>
</tr>
</tbody>
</table>

**pysal.viz.mapclassify.Equal_Interval**

```python
class pysal.viz.mapclassify.Equal_Interval(y, k=5)
    Equal Interval Classification

    Parameters
    y [array] (n,1), values to classify
    k [int] number of classes required

    Notes
    Intervals defined to have equal width:

    \[ bins_j = \min(y) + w \cdot (j + 1) \]

    with \( w = \frac{\max(y) - \min(y)}{k} \)

    Examples

    ```python
    >>> import pysal.viz.mapclassify as mc
    >>> cal = mc.load_example()
    >>> ei = mc.Equal_Interval(cal, k=5)
    >>> ei.k
    5
    >>> ei.counts
    array([57, 0, 0, 0, 1])
    >>> ei.bins
    array([ 822.394, 1644.658, 2466.922, 3289.186, 4111.45])
    ```

    Attributes

    yb [array] (n,1), bin ids for observations, each value is the id of the class the observation belongs to yb[i] = j for j>=1 if bins[j-1] < y[i] <= bins[j], yb[i] = 0 otherwise
    bins [array] (k,1), the upper bounds of each class
    k [int] the number of classes
    counts [array] (k,1), the number of observations falling in each class

    Methods
## `__call__`(*args, **kwargs)
This will allow the classifier to be called like it’s a function.

## `find_bin`(x)
Sort input or inputs according to the current bin estimate

## `get_adcm`()
Absolute deviation around class median (ADCM).

## `get_gadf`()
Goodness of absolute deviation of fit

## `get_tss`()
Total sum of squares around class means

## `make`(*args, **kwargs)
Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.

## `update`([y, inplace])
Add data or change classification parameters.

### `__init__`(y[, k=5])
see class docstring

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code>(y[, k])</td>
<td>see class docstring</td>
</tr>
<tr>
<td><code>find_bin</code>(x)</td>
<td>Sort input or inputs according to the current bin estimate</td>
</tr>
<tr>
<td><code>get_adcm</code>()</td>
<td>Absolute deviation around class median (ADCM).</td>
</tr>
<tr>
<td><code>get_gadf</code>()</td>
<td>Goodness of absolute deviation of fit</td>
</tr>
<tr>
<td><code>get_tss</code>()</td>
<td>Total sum of squares around class means</td>
</tr>
<tr>
<td><code>make</code>(*args, **kwargs)</td>
<td>Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.</td>
</tr>
<tr>
<td><code>update</code>([y, inplace])</td>
<td>Add data or change classification parameters.</td>
</tr>
</tbody>
</table>

### pysal.viz.mapclassify.Fisher_Jenks

#### class pysal.viz.mapclassify.Fisher_Jenks(y, k=5)
Fisher Jenks optimal classifier - mean based

#### Parameters

- **y** [array] (n,1), values to classify
- **k** [int] number of classes required

#### Examples

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> fj = mc.Fisher_Jenks(cal)
>>> fj.adcm
799.24
>>> fj.bins
array([ 75.29, 192.05, 370.5 , 722.85, 4111.45])
>>> fj.counts
array([49, 3, 4, 1, 1])
```
Attributes

- **yb** [array] (n,1), bin ids for observations
- **bins** [array] (k,1), the upper bounds of each class
- **k** [int] the number of classes
- **counts** [array] (k,1), the number of observations falling in each class

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*)</td>
<td>This will allow the classifier to be called like it’s a function.</td>
</tr>
<tr>
<td>find_bin(x)</td>
<td>Sort input or inputs according to the current bin estimate</td>
</tr>
<tr>
<td>get_adcm()</td>
<td>Absolute deviation around class median (ADCM).</td>
</tr>
<tr>
<td>get_gadf()</td>
<td>Goodness of absolute deviation of fit</td>
</tr>
<tr>
<td>get_tss()</td>
<td>Total sum of squares around class means</td>
</tr>
<tr>
<td>make(*)</td>
<td>Configure and create a classifier that will consume data and produce classi-</td>
</tr>
<tr>
<td>update([y, inplace])</td>
<td>Add data or change classification parameters.</td>
</tr>
</tbody>
</table>

**init**(y[, k])

Initialize self. See help(type(self)) for accurate signature.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(y[, k])</td>
<td>Initialize self.</td>
</tr>
<tr>
<td>find_bin(x)</td>
<td>Sort input or inputs according to the current bin estimate</td>
</tr>
<tr>
<td>get_adcm()</td>
<td>Absolute deviation around class median (ADCM).</td>
</tr>
<tr>
<td>get_gadf()</td>
<td>Goodness of absolute deviation of fit</td>
</tr>
<tr>
<td>get_tss()</td>
<td>Total sum of squares around class means</td>
</tr>
<tr>
<td>make(*)</td>
<td>Configure and create a classifier that will consume data and produce classi-</td>
</tr>
<tr>
<td>update([y, inplace])</td>
<td>Add data or change classification parameters.</td>
</tr>
</tbody>
</table>

**pysal.viz.mapclassify.Fisher_Jenks_Sampled**

class **pysal.viz.mapclassify.Fisher_Jenks_Sampled**(y, k=5, pct=0.1, truncate=True)

Fisher Jenks optimal classifier - mean based using random sample

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>[array] (n,1), values to classify</td>
</tr>
<tr>
<td>k</td>
<td>[int] number of classes required</td>
</tr>
<tr>
<td>pct</td>
<td>[float] The percentage of n that should form the sample If pct is specified such that n*pct &gt; 1000, then pct = 1000/n, unless truncate is False</td>
</tr>
</tbody>
</table>
**Truncate** [boolean] truncate pct in cases where pct * n > 1000., (Default True)

### Examples

(Turned off due to timing being different across hardware)

For theoretical details see [RSL16].

**Attributes**

- **yb** [array] (n,1), bin ids for observations
- **bins** [array] (k,1), the upper bounds of each class
- **k** [int] the number of classes
- **counts** [array] (k,1), the number of observations falling in each class

### Methods

- **__call__(**args, **kwargs)** This will allow the classifier to be called like it’s a function.
- **find_bin(x)** Sort input or inputs according to the current bin estimate
- **get_adcm()** Absolute deviation around class median (ADCM).
- **get_gadf()** Goodness of absolute deviation of fit
- **get_tss()** Total sum of squares around class means
- **make(**args, **kwargs)** Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.
- **update([y, inplace])** Add data or change classification parameters.

- **__init__(y[, k=5, pct=0.1, truncate=True])** Initialize self. See help(type(self)) for accurate signature.

### Methods

- **__init__(y[, k, pct, truncate])** Initialize self.
- **find_bin(x)** Sort input or inputs according to the current bin estimate
- **get_adcm()** Absolute deviation around class median (ADCM).
- **get_gadf()** Goodness of absolute deviation of fit
- **get_tss()** Total sum of squares around class means
- **make(**args, **kwargs)** Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.
- **update([y, inplace])** Add data or change classification parameters.

**pysal.viz.mapclassify.HeadTail_Breaks**

class pysal.viz.mapclassify.HeadTail_Breaks(y)

Head/tail Breaks Map Classification for Heavy-tailed Distributions
Parameters

- y [array] (n,1), values to classify

Notes

Head/tail Breaks is a relatively new classification method developed for data with a heavy-tailed distribution. Implementation based on contributions by Alessandra Sozzi <alessandra.sozzi@gmail.com>. For theoretical details see [Jia13].

Examples

```python
>>> import numpy as np
>>> import pysal.viz.mapclassify as mc
>>> np.random.seed(10)
>>> cal = mc.load_example()
>>> htb = mc.HeadTail_Breaks(cal)
>>> htb.k
3
>>> htb.counts
array([50, 7, 1])
>>> htb.bins
array([ 125.92810345, 811.26 , 4111.45])
>>> np.random.seed(123456)
>>> x = np.random.lognormal(3, 1, 1000)
>>> htb = mc.HeadTail_Breaks(x)
>>> htb.bins
array([ 32.26204423, 72.50205622, 128.07150107, 190.2899093 ,
       264.82847377, 457.88157946, 576.76046949])
>>> htb.counts
array([695, 209, 62, 22, 10, 1, 1])
```

Attributes

- yb [array] (n,1), bin ids for observations,
- bins [array] (k,1), the upper bounds of each class
- k [int] the number of classes
- counts [array] (k,1), the number of observations falling in each class

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args, *<em>kwargs)</em></td>
<td>This will allow the classifier to be called like it's a function.</td>
</tr>
<tr>
<td>find_bin(x)</td>
<td>Sort input or inputs according to the current bin estimate.</td>
</tr>
<tr>
<td>get_adcm()</td>
<td>Absolute deviation around class median (ADCM).</td>
</tr>
<tr>
<td>get_gadf()</td>
<td>Goodness of absolute deviation of fit</td>
</tr>
<tr>
<td>get_tss()</td>
<td>Total sum of squares around class means</td>
</tr>
</tbody>
</table>

Continued on next page
make(*args, **kwargs)
Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.

update([y, inplace])
Add data or change classification parameters.

__init__(y)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y)
Initialize self.
find_bin(x)
Sort input or inputs according to the current bin estimate.
get_adcm()
Absolute deviation around class median (ADCM).
get_gadf()
Goodness of absolute deviation of fit
get_tss()
Total sum of squares around class means
make(*args, **kwargs)
Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.
update([y, inplace])
Add data or change classification parameters.

pysal.viz.mapclassify.Jenks_Caspall

class pysal.viz.mapclassify.Jenks_Caspall(y,k=5)
Jenks Caspall Map Classification

Parameters

y [array] (n,1), values to classify
k [int] number of classes required

Examples

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> jc = mc.Jenks_Caspall(cal, k = 5)
>>> jc.bins
array([1.81000e+00, 7.60000e+00, 2.98200e+01, 1.81270e+02, 4.11145e+03])
>>> jc.counts
array([14, 13, 14, 10, 7])
```

Attributes

yb [array] (n,1), bin ids for observations,
bins [array] (k,1), the upper bounds of each class
k [int] the number of classes
counts [array] (k,1), the number of observations falling in each class
Methods

__call__(*args, **kwargs) This will allow the classifier to be called like it’s a function.

find_bin(x) Sort input or inputs according to the current bin estimate.

get_adcm() Absolute deviation around class median (ADCM).

get_gadf() Goodness of absolute deviation of fit

get_tss() Total sum of squares around class means

make(*args, **kwargs) Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.

update([y, inplace]) Add data or change classification parameters.

__init__(y, k=5) Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, k) Initialize self.

find_bin(x) Sort input or inputs according to the current bin estimate.

get_adcm() Absolute deviation around class median (ADCM).

get_gadf() Goodness of absolute deviation of fit

get_tss() Total sum of squares around class means

make(*args, **kwargs) Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.

update([y, inplace]) Add data or change classification parameters.

pysal.viz.mapclassify.Jenks_Caspall_Forced

class pysal.viz.mapclassify.Jenks_Caspall_Forced(y, k=5) Jenks Caspall Map Classification with forced movements

Parameters

y [array] (n,1), values to classify

k [int] number of classes required

Examples

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> jcf = mc.Jenks_Caspall_Forced(cal, k = 5)
>>> jcf.k
5
>>> jcf.bins
array([[1.34000e+00],
       [5.90000e+00]],
       dtype=...)"
[1.67000e+01],
[5.06500e+01],
[4.11145e+03])
>>> jcf.counts
array([12, 12, 13, 9, 12])
>>> jcf4 = mc.Jenks_Caspall_Forced(cal, k = 4)
>>> jcf4.k
4
>>> jcf4.bins
array([[2.51000e+00],
[8.70000e+00],
[3.66800e+01],
[4.11145e+03]])
>>> jcf4.counts
array([15, 14, 14, 15])

Attributes

yb [array] (n,1), bin ids for observations
bins [array] (k,1), the upper bounds of each class
k [int] the number of classes
counts [array] (k,1), the number of observations falling in each class

Methods

__call__(*args, **kwargs) This will allow the classifier to be called like it’s a function.
find_bin(x) Sort input or inputs according to the current bin estimate
get_adcm() Absolute deviation around class median (ADCM).
get_gadf() Goodness of absolute deviation of fit
get_tss() Total sum of squares around class means
make(*args, **kwargs) Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.
update([y, inplace]) Add data or change classification parameters.

__init__(y[, k=5]) Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(yl, k]
find_bin(x) Sort input or inputs according to the current bin estimate
get_adcm() Absolute deviation around class median (ADCM).
get_gadf() Goodness of absolute deviation of fit
get_tss() Total sum of squares around class means

Continued on next page
Table 153 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>make(*args, **kwargs)</td>
<td>Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.</td>
</tr>
<tr>
<td>update([y, inplace])</td>
<td>Add data or change classification parameters.</td>
</tr>
</tbody>
</table>

**pysal.viz.mapclassify.Jenks_Caspall_Sampled**

**class pysal.viz.mapclassify.Jenks_Caspall_Sampled(y, k=5, pct=0.1)**

Jenks Caspall Map Classification using a random sample

**Parameters**

- **y** [array] (n,1), values to classify
- **k** [int] number of classes required
- **pct** [float] The percentage of n that should form the sample If pct is specified such that n*pct > 1000, then pct = 1000/n

**Notes**

This is intended for large n problems. The logic is to apply Jenks_Caspall to a random subset of the y space and then bin the complete vector y on the bins obtained from the subset. This would trade off some “accuracy” for a gain in speed.

**Examples**

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> x = np.random.random(100000)
>>> jc = mc.Jenks_Caspall(x)
>>> jcs = mc.Jenks_Caspall_Sampled(x)
>>> jc.bins
array([0.1988721 , 0.39624334, 0.59441487, 0.79624357, 0.99999251])
>>> jcs.bins
array([0.20998558, 0.42112792, 0.62752937, 0.80543819, 0.99999251])
>>> jc.counts
array([19943, 19510, 19547, 20297, 20703])
>>> jcs.counts
array([21039, 20908, 20425, 17813, 19815])
```

# not for testing since we get different times on different hardware # just included for documentation of likely speed gains #>>> t1 = time.time(); jc = Jenks_Caspall(x); t2 = time.time() #>>> t1s = time.time(); jcs = Jenks_Caspall_Sampled(x); t2s = time.time() #>>> t2 - t1; t2s - t1s #1.8292930126190186 #0.061631917953491211

**Attributes**

- **yb** [array] (n,1), bin ids for observations,
- **bins** [array] (k,1), the upper bounds of each class
- **k** [int] the number of classes
- **counts** [array] (k,1), the number of observations falling in each class
**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args, **kwargs)</td>
<td>This will allow the classifier to be called like it’s a function.</td>
</tr>
<tr>
<td>find_bin(x)</td>
<td>Sort input or inputs according to the current bin estimate.</td>
</tr>
<tr>
<td>get_adcm()</td>
<td>Absolute deviation around class median (ADCM).</td>
</tr>
<tr>
<td>get_gadf()</td>
<td>Goodness of absolute deviation of fit</td>
</tr>
<tr>
<td>get_tss()</td>
<td>Total sum of squares around class means</td>
</tr>
<tr>
<td>make(*args, **kwargs)</td>
<td>Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.</td>
</tr>
<tr>
<td>update([y, inplace])</td>
<td>Add data or change classification parameters.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>()</td>
<td>Initialize self. See help(type(self)) for accurate signature.</td>
</tr>
</tbody>
</table>

```python
import pysal.viz.mapclassify as mc
cal = mc.load_example()
mp = mc.Max_P_Classifier(cal)
mp.bins
```

**pysal.viz.mapclassify.Max_P_Classifier**

**class pysal.viz.mapclassify.Max_P_Classifier**(y, k=5, initial=1000)

Max_P Map Classification

Based on Max_p regionalization algorithm

**Parameters**

- y [array] (n,1), values to classify
- k [int] number of classes required
- initial [int] number of initial solutions to use prior to swapping

**Examples**

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> mp = mc.Max_P_Classifier(cal)
>>> mp.bins
```

(continues on next page)
Attributes

- **yb** [array] (n,1), bin ids for observations,
- **bins** [array] (k,1), the upper bounds of each class
- **k** [int] the number of classes
- **counts** [array] (k,1), the number of observations falling in each class

Methods

- **__call__**(args, **kwargs) This will allow the classifier to be called like it’s a function.
- **find_bin**(x) Sort input or inputs according to the current bin estimate
- **get_adcm**() Absolute deviation around class median (ADCM).
- **get_gadf**() Goodness of absolute deviation of fit
- **get_tss**() Total sum of squares around class means
- **make**(args, **kwargs) Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.
- **update**(y, inplace) Add data or change classification parameters.

**__init__**(y, k=5, initial=1000)
Initialize self. See help(type(self)) for accurate signature.

Methods

- **__init__**(y, k, initial) Initialize self.
- **find_bin**(x) Sort input or inputs according to the current bin estimate
- **get_adcm**() Absolute deviation around class median (ADCM).
- **get_gadf**() Goodness of absolute deviation of fit
- **get_tss**() Total sum of squares around class means
- **make**(args, **kwargs) Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.
- **update**(y, inplace) Add data or change classification parameters.

**pysal.viz.mapclassify.Maximum_Breaks**

**class** **pysal.viz.mapclassify.Maximum_Breaks**(y, k=5, mindiff=0)
Maximum Breaks Map Classification

**Parameters**
y [array] (n, 1), values to classify
k [int] number of classes required
mindiff [float] The minimum difference between class breaks

Examples

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> mb = mc.Maximum_Breaks(cal, k = 5)
>>> mb.k
5
>>> mb.bins
array([ 146.005, 228.49 , 546.675, 2417.15 , 4111.45 ])
>>> mb.counts
array([50, 2, 4, 1, 1])
```

Attributes

yb [array] (n, 1), bin ids for observations
bins [array] (k, 1), the upper bounds of each class
k [int] the number of classes
counts [array] (k, 1), the number of observations falling in each class (numpy array k x 1)

Methods

```python
__call__(*args, **kwargs)
find_bin(x)
get_adcm()
get_gadf()
get_tss()
make(*args, **kwargs)
update([y, inplace])
```

__init__ (y[, k, mindiff])
Initialize self. See help(type(self)) for accurate signature.

Methods

```python
__init__ (y[, k, mindiff])
find_bin(x)
get_adcm()
get_gadf()
```
Table 159 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_tss()</td>
<td>Total sum of squares around class means</td>
</tr>
<tr>
<td>make(*args, **kwargs)</td>
<td>Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.</td>
</tr>
<tr>
<td>update([y, inplace])</td>
<td>Add data or change classification parameters.</td>
</tr>
</tbody>
</table>

**pysal.viz.mapclassify.Natural_Breaks**

**class** `pysal.viz.mapclassify.Natural_Breaks(y, k=5, initial=100)`

Natural Breaks Map Classification

**Parameters**

- `y` [array] (n,1), values to classify
- `k` [int] number of classes required
- `initial` [int] number of initial solutions to generate, (default=100)

**Notes**

There is a tradeoff here between speed and consistency of the classification. If you want more speed, set `initial` to a smaller value (0 would result in the best speed, if you want more consistent classes in multiple runs of `Natural_Breaks` on the same data, set `initial` to a higher value.

**Examples**

```python
>>> import numpy as np
>>> import pysal.viz.mapclassify as mc
>>> np.random.seed(123456)
>>> cal = mc.load_example()
>>> nb = mc.Natural_Breaks(cal, k=5)
>>> nb.k
5
>>> nb.counts
array([41, 9, 6, 1, 1])
>>> nb.bins
array([ 29.82, 110.74, 370.5 , 722.85, 4111.45])
>>> x = np.array([1] * 50)
>>> x[-1] = 20
>>> nb = mc.Natural_Breaks(x, k = 5, initial = 0)

Warning: Not enough unique values in array to form k classes Warning: setting k to 2

>>> nb.bins
array([ 1, 20])
>>> nb.counts
array([49, 1])
```

**Attributes**

- `yb` [array] (n,1), bin ids for observations,
- `bins` [array] (k,1), the upper bounds of each class
k [int] the number of classes

counts [array] (k,1), the number of observations falling in each class

Methods

__call__(*args, **kwargs) This will allow the classifier to be called like it’s a function.

find_bin(x) Sort input or inputs according to the current bin estimate

get_adcm() Absolute deviation around class median (ADCM).

get_gadf() Goodness of absolute deviation of fit

get_tss() Total sum of squares around class means

make(*args, **kwargs) Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.

update([y, inplace]) Add data or change classification parameters.

__init__ (y, k=5, initial=100) Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (y[, k, initial]) Initialize self.

find_bin(x) Sort input or inputs according to the current bin estimate

get_adcm() Absolute deviation around class median (ADCM).

get_gadf() Goodness of absolute deviation of fit

get_tss() Total sum of squares around class means

make(*args, **kwargs) Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.

update([y, inplace]) Add data or change classification parameters.

pysal.viz.mapclassify.Quantiles

class pysal.viz.mapclassify.Quantiles (y, k=5) Quantile Map Classification

Parameters

y [array] (n,1), values to classify

k [int] number of classes required

Examples

>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> q = mc.Quantiles(cal, k = 5)
Attributes

  yb  [array] (n,1), bin ids for observations, each value is the id of the class the observation belongs to yb[i] = j for j>=1 if bins[j-1] < y[i] <= bins[j], yb[i] = 0 otherwise

  bins  [array] (k,1), the upper bounds of each class

  k  [int] the number of classes

  counts  [array] (k,1), the number of observations falling in each class

Methods

__call__(*args, **kwargs)  This will allow the classifier to be called like it’s a function.

find_bin(x)  Sort input or inputs according to the current bin estimate

get_adcm()  Absolute deviation around class median (ADCM).

get_gadf()  Goodness of absolute deviation of fit

get_tss()  Total sum of squares around class means

make(*args, **kwargs)  Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.

update([y, inplace])  Add data or change classification parameters.

__init__(y, k=5)  Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, k)  Initialize self.

find_bin(x)  Sort input or inputs according to the current bin estimate

get_adcm()  Absolute deviation around class median (ADCM).

get_gadf()  Goodness of absolute deviation of fit

get_tss()  Total sum of squares around class means

make(*args, **kwargs)  Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.

update([y, inplace])  Add data or change classification parameters.

\textit{pysal.viz.mapclassify.Percentiles}

\texttt{class \textit{pysal.viz.mapclassify.Percentiles}(y, \textit{pct}=[1, 10, 50, 90, 99, 100])}

Percentiles Map Classification
Parameters

y [array] attribute to classify
pct [array] percentiles default=[1, 10, 50, 99, 100]

Examples

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> p = mc.Percentiles(cal)
>>> p.bins
array([ 1.357000e-01, 5.530000e-01, 9.365000e+00, 2.139140e+02,
       2.179948e+03, 4.111450e+03])
>>> p.counts
array([ 1, 5, 23, 23, 5, 1])
>>> p2 = mc.Percentiles(cal, pct = [50, 100])
>>> p2.bins
array([ 9.365, 4111.45 ])
>>> p2.counts
array([29, 29])
>>> p2.k
2
```

Attributes

yb [array] bin ids for observations (numpy array n x 1)

bins [array] the upper bounds of each class (numpy array k x 1)
k [int] the number of classes

counts [int] the number of observations falling in each class (numpy array k x 1)

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__call__</code></td>
<td>This will allow the classifier to be called like it's a function.</td>
</tr>
<tr>
<td><code>find_bin(x)</code></td>
<td>Sort input or inputs according to the current bin estimate</td>
</tr>
<tr>
<td><code>get_adcm()</code></td>
<td>Absolute deviation around class median (ADCM).</td>
</tr>
<tr>
<td><code>get_gadf()</code></td>
<td>Goodness of absolute deviation of fit</td>
</tr>
<tr>
<td><code>get_tss()</code></td>
<td>Total sum of squares around class means</td>
</tr>
<tr>
<td><code>make(*args, **kwargs)</code></td>
<td>Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.</td>
</tr>
<tr>
<td><code>update([y, inplace])</code></td>
<td>Add data or change classification parameters.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code></td>
<td>(y, pct=[1, 10, 50, 99, 100]) Initialize self. See help(type(self)) for accurate signature.</td>
</tr>
</tbody>
</table>
pysal Documentation, Release 2.0.0

__init__(y[, pct])
Initialize self.

find_bin(x)
Sort input or inputs according to the current bin estimate.

get_adcm()
Absolute deviation around class median (ADCM).

get_gadf()
Goodness of absolute deviation of fit.

get_tss()
Total sum of squares around class means.

make(*args, **kwargs)
Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.

update([y, inplace])
Add data or change classification parameters.

pysal.viz.mapclassify.Std_Mean

class pysal.viz.mapclassify.Std_Mean(y, multiples=[-2, -1, 1, 2])
Standard Deviation and Mean Map Classification

Parameters

y [array] (n,1), values to classify

multiples [array] the multiples of the standard deviation to add/subtract from the sample mean to define the bins, default=[-2,-1,1,2]

Examples

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> st = mc.Std_Mean(cal)
>>> st.k
5
>>> st.bins
array([[-967.36235382, -420.71712519, 672.57333208, 1219.21856072,
       4111.45]])
>>> st.counts
array([ 0, 0, 56, 1, 1])
```

>>> st3 = mc.Std_Mean(cal, multiples = [-3, -1.5, 1.5, 3])
>>> st3.bins
array([[-1514.00758246, -694.03973951, 945.8959464 , 1765.86378936,
       4111.45]])
>>> st3.counts
array([ 0, 0, 57, 0, 1])

Attributes

yb [array] (n,1), bin ids for observations,

bins [array] (k,1), the upper bounds of each class

k [int] the number of classes

counts [array] (k,1), the number of observations falling in each class

Chapter 1. PySAL Components
### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args, **kwargs)</td>
<td>This will allow the classifier to be called like it’s a function.</td>
</tr>
<tr>
<td>find_bin(x)</td>
<td>Sort input or inputs according to the current bin estimate.</td>
</tr>
<tr>
<td>get_adcm()</td>
<td>Absolute deviation around class median (ADCM).</td>
</tr>
<tr>
<td>get_gadf()</td>
<td>Goodness of absolute deviation of fit.</td>
</tr>
<tr>
<td>get_tss()</td>
<td>Total sum of squares around class means.</td>
</tr>
<tr>
<td>make(*args, **kwargs)</td>
<td>Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.</td>
</tr>
<tr>
<td>update([y, inplace])</td>
<td>Add data or change classification parameters.</td>
</tr>
</tbody>
</table>

**__init__**(y[, multiples=\{-2, -1, 1, 2\}])  
Initialize self. See help(type(self)) for accurate signature.

### pysal.viz.mapclassify.User_Defined

**class** pysal.viz.mapclassify.User_Defined(y, bins)  
User Specified Binning

**Parameters**

- `y` [array] (n,1), values to classify
- `bins` [array] (k,1), upper bounds of classes (have to be monotonically increasing)

**Notes**

If upper bound of user bins does not exceed max(y) we append an additional bin.

**Examples**

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> bins = [20, max(cal)]
```

(continues on next page)
>>> bins
[20, 4111.45]
>>> ud = mc.User_Defined(cal, bins)
>>> ud.bins
array([ 20. , 4111.45])
>>> ud.counts
array([37, 21])

>>> bins = [20, 30]
>>> ud = mc.User_Defined(cal, bins)
>>> ud.bins
array([ 20. , 30. , 4111.45])
>>> ud.counts
array([37, 4, 17])

Attributes

- **yb** [array] (n,1), bin ids for observations,
- **bins** [array] (k,1), the upper bounds of each class
- **k** [int] the number of classes
- **counts** [array] (k,1), the number of observations falling in each class

Methods

- **__call__(**args, **kwargs)** This will allow the classifier to be called like it’s a function.
- **find_bin(x)** Sort input or inputs according to the current bin estimate
- **get_adcm()** Absolute deviation around class median (ADCM).
- **get_gadf()** Goodness of absolute deviation of fit
- **get_tss()** Total sum of squares around class means
- **make(**args, **kwargs)** Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.
- **update(y, inplace)** Add data or change classification parameters.

- **__init__(y, bins)**
  Initialize self. See help(type(self)) for accurate signature.

Methods

- **__init__(y, bins)**
  Initialize self.
- **find_bin(x)** Sort input or inputs according to the current bin estimate
- **get_adcm()** Absolute deviation around class median (ADCM).
- **get_gadf()** Goodness of absolute deviation of fit
- **get_tss()** Total sum of squares around class means

Continued on next page
Table 169 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>make(*args, **kwargs)</td>
<td>Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.</td>
</tr>
<tr>
<td>update([y, inplace])</td>
<td>Add data or change classification parameters.</td>
</tr>
</tbody>
</table>

Utilities

mapclassify.K_classifiers(y[, pct])
Evaluate all k-classifiers and pick optimal based on k and GADF

mapclassify.gadf(y[, method, maxk, pct])
Evaluate the Goodness of Absolute Deviation Fit of a Classifier Finds the minimum value of k for which gadf>pct

pysal.viz.mapclassify.K_classifiers

class pysal.viz.mapclassify.K_classifiers(y, pct=0.8)
Evaluate all k-classifiers and pick optimal based on k and GADF

Parameters

y [array] (n,1), values to be classified
pct [float] The percentage of GADF to exceed

See also:
gadf

Notes

This can be used to suggest a classification scheme.

Examples

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> ks = mc.classifiers.K_classifiers(cal)
>>> ks.best.name
'Fisher_Jenks'
>>> ks.best.k
4
>>> ks.best.gadf
0.8481032719908105
```

Attributes

- best [object] instance of the optimal Map_Classifier
- results [dictionary] keys are classifier names, values are the Map_Classifier instances with the best pct for each classifier

__init__(y, pct=0.8)
Initialize self. See help(type(self)) for accurate signature.
Methods

```python
__init__(y[, pct])
```

Initialize self.

```python
pysal.viz.mapclassify.gadf
```

Evaluates the Goodness of Absolute Deviation Fit of a Classifier Finds the minimum value of \(k\) for which \(\text{gadf} > \text{pct}\).

**Parameters**

- **y** [array] (n, 1) values to be classified
- **maxk** [int] maximum value of \(k\) to evaluate
- **pct** [float] The percentage of GADF to exceed

**Returns**

- **k** [int] number of classes
- **cl** [object] instance of the classifier at \(k\)
- **gadf** [float] goodness of absolute deviation fit

See also:

`K_classifiers`

**Notes**

The GADF is defined as:

\[
GADF = 1 - \sum_{c} \sum_{i \in c} \frac{|y_i - y_{c,med}|}{\sum_{i} |y_i - y_{med}|}
\]

where \(y_{med}\) is the global median and \(y_{c,med}\) is the median for class \(c\).

**Examples**

```python
>>> import pysal.viz.mapclassify as mc
>>> cal = mc.load_example()
>>> qgadf = mc.classifiers.gadf(cal)
>>> qgadf[0]
15
>>> qgadf[-1]
0.3740257590909283
```

Quantiles fail to exceed 0.80 before 15 classes. If we lower the bar to 0.2 we see quintiles as a result.
```python
>>> qgadf2 = mc.classifiers.gadf(cal, pct = 0.2)
>>> qgadf2[0]
5
>>> qgadf2[-1]
0.21710231966462412

pysal.viz.splot: Lightweight visualization interface

Giddy

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>splot.giddy.dynamic_lisa_heatmap(rose, p, ax)</code></td>
<td>Heatmap indicating significant transition of LISA values over time inbetween Moran Scatterplot quadrants</td>
</tr>
<tr>
<td><code>splot.giddy.dynamic_lisa_rose(rose,...)</code></td>
<td>Plot dynamic LISA values in a rose diagram.</td>
</tr>
<tr>
<td><code>splot.giddy.dynamic_lisa_vectors(rose, ax,...)</code></td>
<td>Plot vectors of positional transition of LISA values in Moran scatterplot</td>
</tr>
<tr>
<td><code>splot.giddy.dynamic_lisa_composite(rose, gdf)</code></td>
<td>Composite visualisation for dynamic LISA values over two points in time.</td>
</tr>
<tr>
<td><code>splot.giddy.dynamic_lisa_composite_explore(...)</code></td>
<td>Interactive exploration of dynamic LISA values for different dates in a dataframe.</td>
</tr>
</tbody>
</table>

```python
pysal.viz.splot.giddy.dynamic_lisa_heatmap

Heatmap indicating significant transition of LISA values over time inbetween Moran Scatterplot quadrants

Parameters

- `rose` [giddy.directional.Rose instance] A Rose object, which contains (among other attributes) LISA values at two points in time, and a method to perform inference on those.
- `p` [float, optional] The p-value threshold for significance. Default = 0.05
- `ax` [Matplotlib Axes instance, optional] If given, the figure will be created inside this axis. Default = None.
- **kwargs [keyword arguments, optional] Keywords used for creating and designing the heatmap. These are passed on to seaborn.heatmap(). See seaborn documentation for valid keywords. Note: “Start time” refers to `y1` in `Y = np.array([y1, y2]).T` with `giddy.Rose(Y, w, k=5), “End time” referst to `y2`.

Returns

- `fig` [Matplotlib Figure instance] Heatmap figure
- `ax` [Matplotlib Axes instance] Axes in which the figure is plotted

Examples

```python
>>> import geopandas as gpd
>>> import pandas as pd
>>> from pysal.lib.weights.contiguity import Queen
>>> from pysal.lib import examples

(continues on next page)
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from pysal.explore.giddy.directional import Rose
>>> from pysal.viz.splot.giddy import dynamic_lisa_heatmap

get csv and shp files

>>> shp_link = examples.get_path('us48.shp')
>>> df = gpd.read_file(shp_link)
>>> income_table = pd.read_csv(examples.get_path("usjoin.csv"))

calculate relative values

>>> for year in range(1969, 2010):
...     income_table[str(year) + '_rel'] = (income_table[str(year)] / income_table[str(year)].mean())

merge to one gdf

>>> gdf = df.merge(income_table,left_on='STATE_NAME',right_on='Name')

retrieve spatial weights and data for two points in time

>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
>>> y1 = gdf['1969_rel'].values
>>> y2 = gdf['2000_rel'].values

calculate rose Object

>>> Y = np.array([y1, y2]).T
>>> rose = Rose(Y, w, k=5)

plot

>>> dynamic_lisa_heatmap(rose)
>>> plt.show()

customize plot

>>> dynamic_lisa_heatmap(rose, cbar='GnBu')
>>> plt.show()

pysal.viz.splot.giddy.dynamic_lisa_rose

pysal.viz.splot.giddy.dynamic_lisa_rose(rose, attribute=None, ax=None, **kwargs)
Plot dynamic LISA values in a rose diagram.

Parameters

rose [giddy.directional.Rose instance] A Rose object, which contains (among other attributes) LISA values at two points in time, and a method to perform inference on those.

attribute [(n,) ndarray, optional] Points will be colored by chosen attribute values. Variable to specify colors of the colorbars. Default =None.
End time

Start time

0 0 0 1
2 0 0 0
0 0 0 0
2 0 0 0
0 0 0 0

1.2. API Reference 195
ax  [Matplotlib Axes instance, optional] If given, the figure will be created inside this axis. Default = None. Note: This axis should have a polar projection.

**kwargs  [keyword arguments, optional] Keywords used for creating and designing the matplotlib.pyplot.scatter(). Note: ‘c’ and ‘color’ cannot be passed when attribute is not None.

**Returns**

fig  [Matplotlib Figure instance] LISA rose plot figure

ax  [Matplotlib Axes instance] Axes in which the figure is plotted

**Examples**

```python
>>> import geopandas as gpd
gdf = gpd.read_file(shp_link)

>>> import pandas as pd
>>> income_table = pd.read_csv(examples.get_path("usjoin.csv"))

>>> for year in range(1969, 2010):
...    income_table[str(year) + '_rel'] = (income_table[str(year)] / income_table[str(year)].mean())

>>> gdf = df.merge(income_table,left_on='STATE_NAME',right_on='Name')

>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
>>> y1 = gdf['1969_rel'].values
>>> y2 = gdf['2000_rel'].values

>>> Y = np.array([y1, y2]).T
>>> rose = Rose(Y, w, k=5)

>>> dynamic_lisa_rose(rose, attribute=y1)
>>> plt.show()
```

```python
get csv and shp files

```python
>>> shp_link = examples.get_path('us48.shp')
>>> df = gpd.read_file(shp_link)
>>> income_table = pd.read_csv(examples.get_path("usjoin.csv"))
```
```
>>> dynamic_lisa_rose(rose, c='r')
>>> plt.show()
```

```
pysal.viz.splot.giddy.dynamic_lisa_vectors

pysal.viz.splot.giddy.dynamic_lisa_vectors(rose, ax=None, arrows=True, **kwargs)

Plot vectors of positional transition of LISA values in Moran scatterplot

Parameters

- **rose** [giddy.directional.Rose instance] A Rose object, which contains (among other attributes) LISA values at two points in time, and a method to perform inference on those.
- **ax** [Matplotlib Axes instance, optional] If given, the figure will be created inside this axis. Default = None.
- **arrows** [boolean, optional] If True show arrowheads of vectors. Default = True
- ****kwargs** [keyword arguments, optional] Keywords used for creating and designing the matplotlib.pyplot.plot() Note: ‘c’ and ‘color’ cannot be passed when attribute is not None.

Returns

- **fig** [Matplotlib Figure instance] Figure of dynamic LISA vectors
- **ax** [matplotlib Axes instance] Axes in which the figure is plotted
```
Examples

```python
>>> import geopandas as gpd
>>> import pandas as pd
>>> from pysal.lib.weights.contiguity import Queen
>>> from pysal.lib import examples
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from pysal.explore.giddy.directional import Rose
>>> from pysal.viz.splot.giddy import dynamic_lisa_vectors

get csv and shp files

>>> shp_link = examples.get_path('us48.shp')
>>> df = gpd.read_file(shp_link)
>>> income_table = pd.read_csv(examples.get_path('usjoin.csv'))

calculate relative values

>>> for year in range(1969, 2010):
...     income_table[str(year) + '_rel'] = (income_table[str(year)] / income_table[str(year)].mean())

merge to one gdf

>>> gdf = df.merge(income_table, left_on='STATE_NAME', right_on='Name')

retrieve spatial weights and data for two points in time

>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
>>> y1 = gdf['1969_rel'].values
>>> y2 = gdf['2000_rel'].values

calculate rose Object

>>> Y = np.array([y1, y2]).T
>>> rose = Rose(Y, w, k=5)

plot

>>> dynamic_lisa_vectors(rose)
>>> plt.show()

customize plot

>>> dynamic_lisa_vectors(rose, arrows=False, c='r')
>>> plt.show()

pysal.viz.splot.giddy.dynamic_lisa_composite

pysal.viz.splot.giddy.dynamic_lisa_composite(rose, gdf, p=0.05, figsize=(13, 10))
Composite visualisation for dynamic LISA values over two points in time. Includes dynamic lisa heatmap, dynamic lisa rose plot, and LISA cluster plots for both, compared points in time.
Parameters

rose [giddy.directional.Rose instance] A Rose object, which contains (among other attributes) LISA values at two points in time, and a method to perform inference on those.

gdf [geopandas dataframe instance] The GeoDataFrame containing information and polygons to plot.

p [float, optional] The p-value threshold for significance. Default = 0.05.

figsize: tuple, optional W, h of figure. Default = (13,10)

Returns

fig [Matplotlib Figure instance] Dynamic lisa composite figure.

axs [matplotlib Axes instance] Axes in which the figure is plotted.

Examples

```python
>>> import geopandas as gpd
>>> import pandas as pd
>>> from pysal.lib.weights.contiguity import Queen
>>> from pysal.lib import examples
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from pysal.explore.giddy.directional import Rose
>>> from pysal.viz.splot.giddy import dynamic_lisa_composite

get csv and shp files

```python
>>> shp_link = examples.get_path('us48.shp')
>>> df = gpd.read_file(shp_link)
>>> income_table = pd.read_csv(examples.get_path("usjoin.csv"))
```
calculate relative values

```python
>>> for year in range(1969, 2010):
...     income_table[str(year) + '_rel'] = (income_table[str(year)] / income_table[str(year)].mean())
```merge to one gdf

```python
>>> gdf = df.merge(income_table, left_on='STATE_NAME', right_on='Name')
```retrieve spatial weights and data for two points in time

```python
>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
>>> y1 = gdf['1969_rel'].values
>>> y2 = gdf['2000_rel'].values
```calculate rose Object

```python
>>> Y = np.array([y1, y2]).T
>>> rose = Rose(Y, w, k=5)
```plot
```python
>>> dynamic_lisa_composite(rose, gdf)
>>> plt.show()
```

### Space-time autocorrelation

![Space-time autocorrelation](image)

**customize plot**

```python
>>> fig, axs = dynamic_lisa_composite(rose, gdf)
>>> axs[0].set_ylabel('1996')
>>> axs[0].set_xlabel('2009')
>>> axs[1].set_title('LISA cluster for 1996')
>>> axs[3].set_title('LISA cluster for 2009')
>>> plt.show()
```

**pysal.viz.splot.giddy.dynamic_lisa_composite_explore**

**pysal.viz.splot.giddy.** `dynamic_lisa_composite_explore(rose, gdf, pattern=", p=0.05, figsize=(13, 10))`

Interactive exploration of dynamic LISA values for different dates in a dataframe. Note: only possible in jupyter notebooks

**Parameters**

- `rose` [giddy.directional.Rose instance] A Rose object, which contains (among other attributes) weights to calculate `esda.moran.Moran_local` values
Space-time autocorrelation

LISA cluster for 1996

LISA cluster for 2009
**gdf** [geopandas dataframe instance] The Dataframe containing information and polygons to plot.

**pattern** [str, optional] Option to extract all columns ending with a specific pattern. Only extracted columns will be used for comparison.

**p** [float, optional] The p-value threshold for significance. Default = 0.05

**figsize**: tuple, optional W, h of figure. Default = (13,10)

**Returns**

None

**Examples**

**Note**: this function creates Jupyter notebook widgets, so is meant only to run in a notebook.

```python
>>> import geopandas as gpd
>>> import pandas as pd
>>> from pysal.lib.weights.contiguity import Queen
>>> from pysal.lib import examples
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

If you want to see figures embedded inline in a Jupyter notebook, add a line `%matplotlib inline` at the top of your notebook.

```python
>>> from pysal.explore.giddy.directional import Rose
>>> from pysal.viz.splot.giddy import dynamic_lisa_composite_explore
```

get csv and shp files

```python
>>> shp_link = examples.get_path('us48.shp')
>>> df = gpd.read_file(shp_link)
>>> income_table = pd.read_csv(examples.get_path("usjoin.csv"))
```

calculate relative values

```python
>>> for year in range(1969, 2010):
...     income_table[str(year) + '_rel'] = (income_table[str(year)] / income_table[str(year)].mean())
```

merge to one gdf

```python
>>> gdf = df.merge(income_table,left_on='STATE_NAME',right_on='Name')
```

retrieve spatial weights and data for two points in time

```python
>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
>>> y1 = gdf['1969_rel'].values
>>> y2 = gdf['2000_rel'].values
```

calculate rose Object

```python
>>> Y = np.array([y1, y2]).T
>>> rose = Rose(Y, w, k=5)
```
plot

```python
>>> fig = dynamic_lisa_composite_explore(rose, gdf, pattern='rel')
>>> # plt.show()
```

## ESDA

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>splot.esda.moran_scatterplot(moran[, ...])</code></td>
<td>Moran Scatterplot</td>
</tr>
<tr>
<td><code>splot.esda.plot_moran(moran[, ...])</code></td>
<td>Global Moran’s I simulated reference distribution and scatterplot.</td>
</tr>
<tr>
<td><code>splot.esda.plot_moran_simulation(moran[, ...])</code></td>
<td>Global Moran’s I simulated reference distribution.</td>
</tr>
<tr>
<td><code>splot.esda.plot_moran_bv(moran_bv[, ...])</code></td>
<td>Bivariate Moran’s I simulated reference distribution and scatterplot.</td>
</tr>
<tr>
<td><code>splot.esda.plot_moran_bv_simulation(moran_bv)</code></td>
<td>Bivariate Moran’s I simulated reference distribution.</td>
</tr>
<tr>
<td><code>splot.esda.lisa_cluster(moran_loc, gdf[, p, ...])</code></td>
<td>Create a LISA Cluster map</td>
</tr>
<tr>
<td><code>splot.esda.plot_local_autocorrelation(...)</code></td>
<td>Produce three-plot visualisation of Moran Scatterplot, LISA cluster and Choropleth maps, with Local Moran region and quadrant masking</td>
</tr>
<tr>
<td><code>splot.esda.moran_facet(moran_matrix[, ...])</code></td>
<td>Moran Facet visualization.</td>
</tr>
</tbody>
</table>

### pysal.viz.splot.esda.moran_scatterplot

**pysal.viz.splot.esda.moran_scatterplot**

```python
pysal.viz.splot.esda.moran_scatterplot(moran, zstandard=True, p=None, ax=None, scatter_kwds=None, fitline_kwds=None)
```

**Moran Scatterplot**

**Parameters**

- **moran** [esda.moran instance] Values of Moran’s I Global, Bivariate and Local Autocorrelation Statistics
- **zstandard** [bool, optional] If True, Moran Scatterplot will show z-standardized attribute and spatial lag values. Default = True.
- **p** [float, optional] If given, the p-value threshold for significance for Local Autocorrelation analysis. Points will be colored by significance. By default it will not be colored. Default = None.
- **ax** [Matplotlib Axes instance, optional] If given, the Moran plot will be created inside this axis. Default = None.
- **scatter_kwds** [keyword arguments, optional] Keywords used for creating and designing the scatter points. Default = None.
- **fitline_kwds** [keyword arguments, optional] Keywords used for creating and designing the moran fitline. Default = None.

**Returns**

- **fig** [Matplotlib Figure instance] Moran scatterplot figure
- **ax** [matplotlib Axes instance] Axes in which the figure is plotted
Examples

Imports

```python
>>> import matplotlib.pyplot as plt
>>> from pysal.lib.weights.contiguity import Queen
>>> from pysal.lib import examples
>>> import geopandas as gpd
>>> from pysal.explore.esda.moran import (Moran, Moran_BV,
... Moran_Local, Moran_Local_BV)
>>> from pysal.viz.splot.esda import moran_scatterplot
```

Load data and calculate weights

```python
>>> link_to_data = examples.get_path('Guerry.shp')
>>> gdf = gpd.read_file(link_to_data)
>>> x = gdf['Suicids'].values
>>> y = gdf['Donatns'].values
>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
```

Calculate esda.moran Objects

```python
>>> moran = Moran(y, w)
>>> moran_bv = Moran_BV(y, x, w)
>>> moran_loc = Moran_Local(y, w)
>>> moran_loc_bv = Moran_Local_BV(y, x, w)
```

Plot

```python
>>> fig, axs = plt.subplots(2, 2, figsize=(10,10),
... subplot_kw={'aspect': 'equal'})
>>> moran_scatterplot(moran, p=0.05, ax=axs[0,0])
>>> moran_scatterplot(moran_loc, p=0.05, ax=axs[1,0])
>>> moran_scatterplot(moran_bv, p=0.05, ax=axs[0,1])
>>> moran_scatterplot(moran_loc_bv, p=0.05, ax=axs[1,1])
>>> plt.show()
```

### pysal.viz.splot.esda.plot_moran

`pysal.viz.splot.esda.plot_moran(moran, zstandard=True, scatter_kwds=None, fitline_kwds=None, **kwargs)`

Global Moran’s I simulated reference distribution and scatterplot.

**Parameters**

- **moran** [esda.moran.Moran instance] Values of Moran’s I Global Autocorrelation Statistics
- **zstandard** [bool, optional] If True, Moran Scatterplot will show z-standardized attribute and spatial lag values. Default =True.
- **scatter_kwds** [keyword arguments, optional] Keywords used for creating and designing the scatter points. Default =None.
- **fitline_kwds** [keyword arguments, optional] Keywords used for creating and designing the moran fitline and vertical fitline. Default =None.
- **kwargs** [keyword arguments, optional] Keywords used for creating and designing the figure, passed to seaborn.kdeplot.
Returns

- **fig**: [Matplotlib Figure instance] Moran scatterplot and reference distribution figure
- **ax**: [matplotlib Axes instance] Axes in which the figure is plotted

Examples

Imports

```python
>>> import matplotlib.pyplot as plt
>>> from pysal.lib.weights_contiguity import Queen
>>> from pysal.lib import examples
>>> import geopandas as gpd
>>> from pysal.explore.esda.moran import Moran
>>> from pysal.viz.splot.esda import plot_moran
```

Load data and calculate weights

```python
>>> link_to_data = examples.get_path('Guerry.shp')
>>> gdf = gpd.read_file(link_to_data)
>>> y = gdf['Donatns'].values
>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'

Calculate Global Moran

```python
>>> moran = Moran(y, w)
```

plot

```python
>>> plot_moran(moran)
>>> plt.show()
```

customize plot

```python
>>> plot_moran(moran, zstandard=False,
...           fitline_kwds=dict(color='#4393c3'))
>>> plt.show()
```
Global Moran’s I simulated reference distribution.

**Parameters**

- **moran** [esda.moran.Moran instance] Values of Moran’s I Global Autocorrelation Statistics
- **ax** [Matplotlib Axes instance, optional] If given, the Moran plot will be created inside this axis. Default =None.
- **fitline_kwds** [keyword arguments, optional] Keywords used for creating and designing the vertical moran fitline. Default =None.
- ****kwargs** [keyword arguments, optional] Keywords used for creating and designing the figure, passed to seaborn.kdeplot.

**Returns**

- **fig** [Matplotlib Figure instance] Simulated reference distribution figure
- **ax** [matplotlib Axes instance] Axes in which the figure is plotted

**Examples**

**Imports**

```python
>>> import matplotlib.pyplot as plt
>>> from pysal.lib.weights.contiguity import Queen
>>> from pysal.lib import examples
>>> import geopandas as gpd
>>> from pysal.explore.esda.moran import Moran
>>> from pysal.viz.splot.esda import plot_moran_simulation
```

**Load data and calculate weights**

```python
>>> link_to_data = examples.get_path('Guerry.shp')
>>> gdf = gpd.read_file(link_to_data)
```
Calculate Global Moran

```python
>>> y = gdf['Donatns'].values
>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
```

```python
>>> moran = Moran(y, w)
```

```python
>>> plot_moran_simulation(moran)
>>> plt.show()
```
customize plot

```python
>>> plot_moran_simulation(moran, fitline_kwds=dict(color='#4393c3'))
>>> plt.show()
```

```
Moran I: 0.35
```

Bivariate Moran’s I simulated reference distribution and scatterplot.

```python
pysal.viz.splot.esda.plot_moran_bv
```

**Parameters**
scatter_kwds [keyword arguments, optional] Keywords used for creating and designing the scatter points. Default = None.
fitline_kwds [keyword arguments, optional] Keywords used for creating and designing the moran fitline and vertical fitline. Default = None.
**kwargs [keyword arguments, optional] Keywords used for creating and designing the figure, passed to seaborn.kdeplot.

Returns

fig [Matplotlib Figure instance] Bivariate moran scatterplot and reference distribution figure
ax [matplotlib Axes instance] Axes in which the figure is plotted

Examples

Imports

```python
>>> import matplotlib.pyplot as plt
>>> from pysal.lib.weights.contiguity import Queen
>>> from pysal.lib import examples
>>> import geopandas as gpd
>>> from pysal.explore.esda.moran import Moran_BV
>>> from pysal.viz.splot.esda import plot_moran_bv
```

Load data and calculate weights

```python
>>> link_to_data = examples.get_path('Guerry.shp')
>>> gdf = gpd.read_file(link_to_data)
>>> x = gdf['Suicids'].values
>>> y = gdf['Donatns'].values
>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
```

Calculate Bivariate Moran

```python
>>> moran_bv = Moran_BV(x, y, w)
```

plot

```python
>>> plot_moran_bv(moran_bv)
>>> plt.show()
```

customize plot

```python
>>> plot_moran_bv(moran_bv, fitline_kwds=dict(color='#4393c3'))
>>> plt.show()
```

pysal.viz.splot.esda.plot_moran_bv_simulation

```python
pysal.viz.splot.esda.plot_moran_bv_simulation(moran_bv, ax=None, fitline_kwds=None, **kwargs)
```

Bivariate Moran’s I simulated reference distribution.
Parameters

- **moran_bv** [esda.moran.Moran_BV instance] Values of Bivariate Moran’s I Autocorrelation Statistics
- **ax** [Matplotlib Axes instance, optional] If given, the Moran plot will be created inside this axis. Default = None.
- **fitline_kwds** [keyword arguments, optional] Keywords used for creating and designing the vertical Moran fitline. Default = None.
- **kwargs** [keyword arguments, optional] Keywords used for creating and designing the figure, passed to seaborn.kdeplot.

Returns

- **fig** [Matplotlib Figure instance] Bivariate moran reference distribution figure
- **ax** [matplotlib Axes instance] Axes in which the figure is plotted

Examples

Imports

```python
>>> import matplotlib.pyplot as plt
>>> from pysal.lib.weights.contiguity import Queen
>>> from pysal.lib import examples
>>> import geopandas as gpd
>>> from pysal.explore.esda.moran import Moran_BV
>>> from pysal.viz.splot.esda import plot_moran_bv_simulation
```

Load data and calculate weights

```python
>>> link_to_data = examples.get_path('Guerry.shp')
>>> gdf = gpd.read_file(link_to_data)
>>> x = gdf['Suicides'].values
>>> y = gdf['Donatns'].values
>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
```

Calculate Bivariate Moran

```python
>>> moran_bv = Moran_BV(x, y, w)
```

plot

```python
>>> plot_moran_bv_simulation(moran_bv)
>>> plt.show()
```

customize plot

```python
>>> plot_moran_bv_simulation(moran_bv,
...                          **fitline_kwds=**dict(color='#4393c3'))
>>> plt.show()
```
Reference Distribution

Bivariate Moran I: -0.02
Bivariate Moran I: -0.02

Reference Distribution

Bivariate Moran I: -0.02
pysal.viz.splot.esda.lisa_cluster

Create a LISA Cluster map

Parameters

moran_loc [esda.moran.Moran_Local or Moran_Local_BV instance] Values of Moran’s Local Autocorrelation Statistic
gdf [geopandas dataframe instance] The Dataframe containing information to plot. Note that gdf will be modified, so calling functions should use a copy of the user provided gdf. (either using gdf.assign() or gdf.copy())
p [float, optional] The p-value threshold for significance. Points will be colored by significance.
ax [matplotlib Axes instance, optional] Axes in which to plot the figure in multiple Axes layout. Default = None
legend [boolean, optional] If True, legend for maps will be depicted. Default = True
legend_kwds [dict, optional] Dictionary to control legend formatting options. Example: legend_kwds={'loc': 'upper left', 'bbox_to_anchor': (0.92, 1.05)} Default = None
**kwargs [keyword arguments, optional] Keywords designing and passed to geopandas.GeoDataFrame.plot().

Returns

fig [matplotlib Figure instance] Figure of LISA cluster map
ax [matplotlib Axes instance] Axes in which the figure is plotted

Examples

Imports

```python
>>> import matplotlib.pyplot as plt
>>> from pysal.lib.weights.contiguity import Queen
>>> from pysal.lib import examples
>>> import geopandas as gpd
>>> from pysal.explore.esda.moran import Moran_Local
>>> from pysal.viz.splot.esda.import lisa_cluster
```

Data preparation and statistical analysis

```python
>>> link = examples.get_path('Guerry.shp')
>>> gdf = gpd.read_file(link)
>>> y = gdf['Donatns'].values
>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
>>> moran_loc = Moran_Local(y, w)
```

Plotting

```python
>>> fig = lisa_cluster(moran_loc, gdf)
>>> plt.show()
```
pysal.viz.splot.esda.plot_local_autocorrelation

pysal.viz.splot.esda.plot_local_autocorrelation(moran_loc, gdf, attribute, p=0.05, region_column=None, mask=None, mask_color='#636363', quadrant=None, legend=True, scheme='Quantiles', cmap='YlGnBu', figsize=(15, 4), scatter_kwds=None, fitline_kwds=None)

Produce three-plot visualisation of Moran Scatterplot, LISA cluster and Choropleth maps, with Local Moran region and quadrant masking

Parameters

moran_loc [esda.moran.Moran_Local or Moran_Local_BV instance] Values of Moran’s Local Autocorrelation Statistic

gdf [geopandas dataframe] The Dataframe containing information to plot the two maps.

attribute [str] Column name of attribute which should be depicted in Choropleth map.

p [float, optional] The p-value threshold for significance. Points and polygons will be colored by significance. Default = 0.05.

region_column: string, optional Column name containing mask region of interest. Default = None

mask: str, optional Identifier or name of the region to highlight. Default = None

mask_color: str, optional Color of mask. Default = '#636363'

quadrant [int, optional] Quadrant 1-4 in scatterplot masking values in LISA cluster and Choropleth maps. Default = None

figsize: tuple, optional W, h of figure. Default = (15,4)

legend: boolean, optional If True, legend for maps will be depicted. Default = True

scheme: str, optional Name of PySAL classifier to be used. Default = ‘Quantiles’

cmap: str, optional Name of matplotlib colormap used for plotting the Choropleth. Default = ‘YlGnBu’

scatter_kwds [keyword arguments, optional] Keywords used for creating and designing the scatter points. Default =None.

fitline_kwds [keyword arguments, optional] Keywords used for creating and designing the moran fitline in the scatterplot. Default =None.

Returns

fig [Matplotlib figure instance] Moran Scatterplot, LISA cluster map and Choropleth.

axs [list of Matplotlib axes] Lisat of Matplotlib axes plotted.

Examples

Imports
Data preparation and analysis

```python
>>> link = examples.get_path('Guerry.shp')
>>> gdf = gpd.read_file(link)
>>> y = gdf['Donatns'].values
>>> w = Queen.from_dataframe(gdf)
>>> w.transform = 'r'
>>> moran_loc = Moran_Local(y, w)
```

Plotting with quadrant mask and region mask

```python
>>> fig = plot_local_autocorrelation(moran_loc, gdf, 'Donatns', p=0.05,
... region_column='Dprtmnt',
... mask=['Ain'], quadrant=1)
>>> plt.show()
```

**pysal.viz.splot.esda.moran_facet**

**pysal.viz.splot.esda.moran_facet** *(moran_matrix, figsize=(16, 12), scatter_bv_kwds=None, fitline_bv_kwds=None, scatter_glob_kwds=None, fitline_glob_kwds=None)*

Moran Facet visualization. Includes BV Morans and Global Morans on the diagonal.

**Parameters**

- **moran_matrix** [esda.moran.Moran_BV_matrix instance] Dictionary of Moran_BV objects
- **figsize** [tuple, optional] W, h of figure. Default =(16,12)
- **scatter_bv_kwds** [keyword arguments, optional] Keywords used for creating and designing the scatter points of off-diagonal Moran_BV plots. Default =None.
- **fitline_bv_kwds** [keyword arguments, optional] Keywords used for creating and designing the moran fitline of off-diagonal Moran_BV plots. Default =None.
- **scatter_glob_kwds** [keyword arguments, optional] Keywords used for creating and designing the scatter points of diagonal Moran plots. Default =None.
- **fitline_glob_kwds** [keyword arguments, optional] Keywords used for creating and designing the moran fitline of diagonal Moran plots. Default =None.

**Returns**

- **fig** [Matplotlib Figure instance] Bivariate Moran Local scatterplot figure
- **axarr** [matplotlib Axes instance] Axes in which the figure is plotted

**Examples**

Imports
>>> import matplotlib.pyplot as plt
>>> import pysal.lib as lp
>>> import numpy as np
>>> import geopandas as gpd
>>> from pysal.explore.esda.moran import Moran_BV_matrix
>>> from pysal.viz.splot.esda import moran_facet

Load data and calculate Moran Local statistics

```python
>>> f = gpd.read_file(lp.examples.get_path("sids2.dbf"))
>>> varnames = ['SIDR74', 'SIDR79', 'NWR74', 'NWR79']
>>> vars = [np.array(f[var]) for var in varnames]
>>> w = lp.io.open(lp.examples.get_path("sids2.gal")).read()
>>> Moran_matrix = Moran_BV_matrix(vars, w, varnames = varnames)
```

Plot

```python
>>> fig, axarr = moran_facet(Moran_matrix)
>>> plt.show()
```

Customize plot

```python
>>> fig, axarr = moran_facet(Moran_matrix,
...                          fitline_bv_kwds=dict(color='#4393c3'))
```
Weights

`splot.libpysal.plot_spatial_weights`(w, gdf)  
Plot spatial weights network.

**pysal.viz.splot.libpysal.plot_spatial_weights**

Plot spatial weights network. NOTE: Additionally plots `w.non_planar_joins` if `pysal.lib.weights.util.nonplanar_neighbors()` was applied.

**Returns**

- **fig** [matplotlib Figure instance] Figure of spatial weight network.
- **ax** [matplotlib Axes instance] Axes in which the figure is plotted.
Examples

Imports

```python
>>> from pysal.lib.weights.contiguity import Queen
>>> import geopandas as gpd
>>> import pysal.lib
>>> from pysal.lib import examples
>>> import matplotlib.pyplot as plt
>>> from pysal.viz.splot.libpysal import plot_spatial_weights
```

Data preparation and statistical analysis

```python
>>> gdf = gpd.read_file(examples.get_path('map_RS_BR.shp'))
>>> weights = Queen.from_dataframe(gdf)
>>> wnp = pysal.lib.weights.util.nonplanar_neighbors(weights, gdf)
```

Plot weights

```python
>>> plot_spatial_weights(weights, gdf)
>>> plt.show()
```

Plot corrected weights

```python
>>> plot_spatial_weights(wnp, gdf)
>>> plt.show()
```

mapping

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>splot.mapping.value_by_alpha_cmap(x, y[, ...,])</code></td>
<td>Calculates Value by Alpha rgba values</td>
</tr>
<tr>
<td><code>splot.mapping.vba_choropleth(x, y, gdf[, ...,])</code></td>
<td>Value by Alpha Choropleth</td>
</tr>
<tr>
<td><code>splot.mapping.vba_legend(rgb_bins[, ..., ax])</code></td>
<td>Creates Value by Alpha heatmap used as choropleth legend</td>
</tr>
<tr>
<td><code>splot.mapping.mapclassify_bin(y, classifier)</code></td>
<td>Classify your data with <code>pysal.mapclassify</code> Note: Input parameters are dependent on classifier used.</td>
</tr>
</tbody>
</table>

**pysal.viz.splot.mapping.value_by_alpha_cmap**

`pysal.viz.splot.mapping.value_by_alpha_cmap(x, y, cmap='GnBu', revert_alpha=False, divergent=False)`

Calculates Value by Alpha rgba values

**Parameters**

- `x` [array] Variable determined by color
- `y` [array] Variable determining alpha value
- `cmap` [str or list of str] Matplotlib Colormap or list of colors used to create vba_layer
- `revert_alpha` [bool, optional] If True, high y values will have a low alpha and low values will be transparent. Default =False.
**divergent** [bool, optional] Creates a divergent alpha array with high values at the extremes and low, transparent values in the middle of the input values.

**Returns**

**rgba** [ndarray (n,4)] RGBA colormap, where the alpha channel represents one attribute (x) and the rgb color the other attribute (y)

**cmap** [str or list of str] Original Matplotlib Colormap or list of colors used to create vba_layer

**Examples**

**Imports**

```python
>>> from pysal.lib import examples
>>> import geopandas as gpd
>>> import matplotlib.pyplot as plt
>>> import matplotlib
>>> import numpy as np
>>> from pysal.viz.splot.mapping import value_by_alpha_cmap
```

**Load Example Data**

```python
>>> link_to_data = examples.get_path('columbus.shp')
>>> gdf = gpd.read_file(link_to_data)
>>> x = gdf['HOVAL'].values
>>> y = gdf['CRIME'].values
```

**Create rgba values**

```python
>>> rgba, _ = value_by_alpha_cmap(x, y)
```

**Create divergent rgba and change Colormap**

```python
>>> div_rgba, _ = value_by_alpha_cmap(x, y, cmap='seismic', divergent=True)
```

**Create rgba values with reverted alpha values**

```python
>>> rev_rgba, _ = value_by_alpha_cmap(x, y, cmap='RdBu', revert_alpha=True)
```

**pysal.viz.splot.mapping.vba_choropleth**

```
pysal.viz.splot.mapping.vba_choropleth(x, y, gdf, cmap='GnBu', divergent=False, revert_alpha=False, alpha_mapclassify=None, rgb_mapclassify=None, ax=None, legend=False)
```

**Value by Alpha Choropleth**

**Parameters**

- **x** [array] Variable determined by color
- **y** [array] Variable determining alpha value
- **gdf** [geopandas dataframe instance] The Dataframe containing information to plot.
- **cmap** [str or list of str] Matplotlib Colormap or list of colors used to create vba_layer
- **divergent** [bool, optional] Creates a divergent alpha array with high values at the extremes and low, transparent values in the middle of the input values.
revert_alpha [bool, optional] If True, high y values will have a low alpha and low values will be transparent. Default = False.

alpha_mapclassify [dict] Keywords used for binning input values and classifying alpha values with mapclassify. Note: valid keywords are eg. dict(classifier='quantiles', k=5, hinge=1.5). For other options check splot.mapping.mapclassify_bin.

rgb_mapclassify [dict] Keywords used for binning input values and classifying rgb values with mapclassify. Note: valid keywords are eg.g dict(classifier='quantiles', k=5, hinge=1.5). For other options check splot.mapping.mapclassify_bin.

ax [matplotlib Axes instance, optional] Axes in which to plot the figure in multiple Axes layout. Default = None

legend [bool, optional] Adds a legend. Note: currently only available if data is classified, hence if alpha_mapclassify and rgb_mapclassify are used.

Returns

fig [matplotlib Figure instance] Figure of Value by Alpha choropleth

ax [matplotlib Axes instance] Axes in which the figure is plotted

Examples

Imports

```python
>>> from pysal.lib import examples
>>> import geopandas as gpd
>>> import matplotlib.pyplot as plt
>>> import matplotlib
>>> import numpy as np
>>> from pysal.viz.splot.mapping import vba_choropleth
```

Load Example Data

```python
>>> link_to_data = examples.get_path('columbus.shp')
>>> gdf = gpd.read_file(link_to_data)
>>> x = gdf['HOVAL'].values
>>> y = gdf['CRIME'].values
```

Plot a Value-by-Alpha map

```python
>>> fig, _ = vba_choropleth(x, y, gdf)
>>> plt.show()
```

Plot a Value-by-Alpha map with reverted alpha values

```python
>>> fig, _ = vba_choropleth(x, y, gdf, cmap='RdBu',
... revert_alpha=True)
>>> plt.show()
```

Plot a Value-by-Alpha map with classified alpha and rgb values

```python
>>> fig, axs = plt.subplots(2,2, figsize=(20,10))
>>> vba_choropleth(y, x, gdf, cmap='viridis', ax = axs[0,0],
... rgb_mapclassify=dict(classifier='quantiles', k=3),
... alpha_mapclassify=dict(classifier='quantiles', k=3))
>>> vba_choropleth(y, x, gdf, cmap='viridis', ax = axs[0,1],
... (continues on next page)```
... rgb_mapclassify=dict(classifier='natural_breaks'),
... alpha_mapclassify=dict(classifier='natural_breaks'))
>>> vba_choropleth(y, x, gdf, cmap='viridis', ax = axs[1,0],
... rgb_mapclassify=dict(classifier='std_mean'),
... alpha_mapclassify=dict(classifier='std_mean'))
>>> vba_choropleth(y, x, gdf, cmap='viridis', ax = axs[1,1],
... rgb_mapclassify=dict(classifier='fisher_jenks', k=3),
... alpha_mapclassify=dict(classifier='fisher_jenks', k=3))
>>> plt.show()

Pass in a list of colors instead of a cmap

>>> color_list = ['#a1dab4','#41b6c4','#225ea8']
>>> vba_choropleth(y, x, gdf, cmap=color_list,
... rgb_mapclassify=dict(classifier='quantiles', k=3),
... alpha_mapclassify=dict(classifier='quantiles'))
>>> plt.show()

Add a legend and use divergent alpha values

>>> fig = plt.figure(figsize=(15,10))
>>> ax = fig.add_subplot(111)
>>> vba_choropleth(x, y, gdf, divergent=True,
... alpha_mapclassify=dict(classifier='quantiles', k=5),
... rgb_mapclassify=dict(classifier='quantiles', k=5),
... legend=True, ax=ax)
>>> plt.show()

`pysal.viz.splot.mapping.vba_legend`

`pysal.viz.splot.mapping.vba_legend(rgb_bins, alpha_bins, cmap, ax=None)`

Creates Value by Alpha heatmap used as choropleth legend.
Parameters

- **rgb_bins** [pysal.mapclassify instance] Object of classified values used for rgb. Can be created with `mapclassify_bin()` or `pysal.mapclassify`.
- **alpha_bins** [pysal.mapclassify instance] Object of classified values used for alpha. Can be created with `mapclassify_bin()` or `pysal.mapclassify`.
- **ax** [matplotlib Axes instance, optional] Axes in which to plot the figure in multiple Axes layout. Default = None

Returns

- **fig** [matplotlib Figure instance] Figure of Value by Alpha heatmap
- **ax** [matplotlib Axes instance] Axes in which the figure is plotted

Examples

Imports

```python
>>> from pysal.lib import examples
>>> import geopandas as gpd
>>> import matplotlib.pyplot as plt
>>> import matplotlib
>>> import numpy as np
>>> from pysal.viz.splot.mapping import vba_legend, mapclassify_bin
```

Load Example Data
```python
>>> link_to_data = examples.get_path('columbus.shp')
>>> gdf = gpd.read_file(link_to_data)
>>> x = gdf['HOVAL'].values
>>> y = gdf['CRIME'].values

Classify your data
```n
```python
>>> rgb_bins = mapclassify_bin(x, 'quantiles')
>>> alpha_bins = mapclassify_bin(y, 'quantiles')

Plot your legend
```n
```python
>>> fig, _ = vba_legend(rgb_bins, alpha_bins, cmap='RdBu')
>>> plt.show()
```

**pysal.viz.splot.mapping.mapclassify_bin**

`pysal.viz.splot.mapping.mapclassify_bin(y, classifier, k=5, pct=[1, 10, 50, 90, 99, 100], hinge=1.5, multiples=[-2, -1, 1, 2], mindiff=0, initial=100, bins=None)`

Classify your data with `pysal.mapclassify` Note: Input parameters are dependent on classifier used.

**Parameters**

- `y` [array] (n,1), values to classify
- `classifier` [str] `pysal.mapclassify` classification scheme
- `k` [int, optional] The number of classes. Default=5.
- `pct` [array, optional] Percentiles used for classification with `percentiles`. Default=[1,10,50,90,99,100]
- `hinge` [float, optional] Multiplier for IQR when `Box_Plot` classifier used. Default=1.5.
- `multiples` [array, optional] The multiples of the standard deviation to add/subtract from the sample mean to define the bins using `std_mean`. Default=[-2,-1,1,2].
- `mindiff` [float, optional] The minimum difference between class breaks if using `maximum_breaks` classifier. Default=0.
- `initial` [int] Number of initial solutions to generate or number of runs when using `natural_breaks` or `max_p_classifier`. Default=100. Note: setting initial to 0 will result in the quickest calculation of bins.
- `bins` [array, optional] (k,1), upper bounds of classes (have to be monotonically increasing) if using `user_defined` classifier. Default=None, Example = [20, max(y)].

**Returns**

- `bins` [pysal.mapclassify instance] Object containing bin ids for each observation (.yb), upper bounds of each class (.bins), number of classes (.k) and number of observations falling in each class (.counts)

**Note:** Supported classifiers include: quantiles, box_plot, euqal_interval, fisher_jenks, headtail_breaks, jenks_caspall, jenks_caspall_forced, max_p_classifier, maximum_breaks, natural_breaks, percentiles, std_mean, user_defined
Examples

Imports

```python
>>> from pysal.lib import examples
>>> import geopandas as gpd
>>> from pysal.viz.splot.mapping import pysal.viz.mapclassify_bin
```

Load Example Data

```python
>>> link_to_data = examples.get_path('columbus.shp')
>>> gdf = gpd.read_file(link_to_data)
>>> x = gdf['HOVAL'].values
```

Classify values by quantiles

```python
>>> quantiles = mapclassify_bin(x, 'quantiles')
```

Classify values by box_plot and set hinge to 2

```python
>>> box_plot = mapclassify_bin(x, 'box_plot', hinge=2)
```

1.2.4 pysal.model: Linear models for spatial data analysis

pysal.model.spreg: Spatial Econometrics

These are the standard spatial regression models supported by the `spreg` package. Each of them contains a significant amount of detail in their docstring discussing how they’re used, how they’re fit, and how to interpret the results.

```plaintext
sreg.OLS(y, x[, w, robust, gwk, sig2n_k, ...]) Ordinary least squares with results and diagnostics.
sreg.ML_Lag(y, x, w[, method, epsilon, ...]) ML estimation of the spatial lag model with all results and diagnostics; Anselin (1988) [Anselin1988]
sreg.ML_Error(y, x, w[, method, epsilon, ...]) ML estimation of the spatial lag model with all results and diagnostics; Anselin (1988) [Anselin1988]
sreg.GM_Lag(y, x[, yend, q, w, w_lags, ...]) Spatial two stage least squares (S2SLS) with results and diagnostics; Anselin (1988) [Anselin1988]
sreg.GM_Error(y, x, w[, vm, name_y, ...]) GMM method for a spatial error model, with results and diagnostics; based on Kelejian and Prucha (1998, 1999) [Kelejian1998] [Kelejian1999].
sreg.GM_Error_Het(y, x, w[, max_iter, ...]) GMM method for a spatial error model with heteroskedasticity, with results and diagnostics; based on Arraiz et al [Arraiz2010], following Anselin [Anselin2011].
sreg.GM_Error_Hom(y, x, w[, max_iter, ...]) GMM method for a spatial error model with homoskedasticity, with results and diagnostics; based on Drukker et al.
sreg.GM_Comb[]
```

Continued on next page
Table 176 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spreg.GM_Combo_Het</code></td>
<td>GMM method for a spatial lag and error model with heteroskedasticity and endogenous variables, with results and diagnostics; based on Arbia et al [Arbia2010], following Anselin [Anselin2011].</td>
</tr>
<tr>
<td><code>spreg.GM_Combo_Hom</code></td>
<td>GMM method for a spatial lag and error model with homoskedasticity and endogenous variables, with results and diagnostics; based on Drukker et al.</td>
</tr>
<tr>
<td><code>spreg.GM_Endog_Error</code></td>
<td>GMM method for a spatial error model with endogenous variables, with results and diagnostics; based on Kelejian and Prucha (1998, 1999) [Kelejian1998] [Kelejian1999].</td>
</tr>
<tr>
<td><code>spreg.GM_Endog_Error_Het</code></td>
<td>GMM method for a spatial error model with heteroskedasticity and endogenous variables, with results and diagnostics; based on Arbia et al [Arbia2010], following Anselin [Anselin2011].</td>
</tr>
<tr>
<td><code>spreg.GM_Endog_Error_Hom</code></td>
<td>GMM method for a spatial error model with homoskedasticity and endogenous variables, with results and diagnostics; based on Drukker et al.</td>
</tr>
<tr>
<td><code>spreg.TSLS</code></td>
<td>Two stage least squares with results and diagnostics.</td>
</tr>
<tr>
<td><code>spreg.ThreeSLS</code></td>
<td>User class for 3SLS estimation</td>
</tr>
</tbody>
</table>

**pysal.model.spreg.OLS**

Class `pysal.model.spreg.OLS(y, x, w=None, robust=None, gwk=None, sig2n_k=True, nonspat_diag=True, spat_diag=False, moran=False, white_test=False, vm=False, name_y=None, name_x=None, name_w=None, name_gwk=None, name_ds=None)`

Ordinary least squares with results and diagnostics.

**Parameters**

- **y** [array] nx1 array for dependent variable
- **x** [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
- **w** [pysal W object] Spatial weights object (required if running spatial diagnostics)
- **robust** [string] If ‘white’, then a White consistent estimator of the variance-covariance matrix is given. If ‘hac’, then a HAC consistent estimator of the variance-covariance matrix is given. Default set to None.
- **gwk** [pysal W object] Kernel spatial weights needed for HAC estimation. Note: matrix must have ones along the main diagonal.
- **sig2n_k** [boolean] If True, then use n-k to estimate sigma^2. If False, use n.
- **nonspat_diag** [boolean] If True, then compute non-spatial diagnostics on the regression.
- **spat_diag** [boolean] If True, then compute Lagrange multiplier tests (requires w). Note: see moran for further tests.
- **moran** [boolean] If True, compute Moran’s I on the residuals. Note: requires spat_diag=True.
- **white_test** [boolean] If True, compute White’s specification robust test. (requires nonspat_diag=True)
- **vm** [boolean] If True, include variance-covariance matrix in summary results
name_y [string] Name of dependent variable for use in output
name_x [list of strings] Names of independent variables for use in output
name_w [string] Name of weights matrix for use in output
name_gwk [string] Name of kernel weights matrix for use in output
name_ds [string] Name of dataset for use in output

Examples

>>> import numpy as np
>>> import pysal.lib

Open data on Columbus neighborhood crime (49 areas) using pysal.lib.io.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.lib.io.open() also reads data in CSV format; also, the actual OLS class requires data to be passed in as numpy arrays so the user can read their data in using any method.

>>> db = pysal.lib.io.open(pysal.lib.examples.get_path('columbus.dbf'), 'r')

Extract the HOVAL column (home values) from the DBF file and make it the dependent variable for the regression. Note that PySAL requires this to be an nx1 numpy array.

>>> hoval = db.by_col("HOVAL")
>>> y = np.array(hoval)
>>> y.shape = (len(hoval), 1)

Extract CRIME (crime) and INC (income) vectors from the DBF to be used as independent variables in the regression. Note that PySAL requires this to be an nxj numpy array, where j is the number of independent variables (not including a constant). pysal.model.spreg.OLS adds a vector of ones to the independent variables passed in.

>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("CRIME"))
>>> X = np.array(X).T

The minimum parameters needed to run an ordinary least squares regression are the two numpy arrays containing the independent variable and dependent variables respectively. To make the printed results more meaningful, the user can pass in explicit names for the variables used; this is optional.

>>> ols = OLS(y, X, name_y='home value', name_x=['income','crime'], name_ds='columbus', white_test=True)

pysal.model.spreg.OLS computes the regression coefficients and their standard errors, t-stats and p-values. It also computes a large battery of diagnostics on the regression. In this example we compute the white test which by default isn’t (‘white_test=True’). All of these results can be independently accessed as attributes of the regression object created by running pysal.model.spreg.OLS. They can also be accessed at one time by printing the summary attribute of the regression object. In the example below, the parameter on crime is -0.4849, with a t-statistic of -2.6544 and p-value of 0.01087.

>>> ols.betas
array([[ 46.42818268],
       [ 0.62898397],
       [-0.48488854]])

>>> print round(ols.t_stat[2][0],3)
Or we can easily obtain a full summary of all the results nicely formatted and ready to be printed:

```python
>>> print ols.summary
REGRESSION
---------
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES
-----------------------------------------
Data set : columbus
Dependent Variable : home value Number of Observations: 49
Mean dependent var : 38.4362 Number of Variables : 3
S.D. dependent var : 18.4661 Degrees of Freedom : 46
R-squared : 0.3495
Adjusted R-squared : 0.3212
Sum squared residual: 10647.015 F-statistic : 12.3582
Sigma-square : 231.457 Prob(F-statistic) : 5.064e-05
S.E. of regression : 15.214
Log likelihood : -201.368
Sigma-square ML : 217.286 Akaike info criterion : 408.735
S.E of regression ML: 14.7406 Schwarz criterion : 414.411
<BLANKLINE>
----------------------------------------------------------------------------------
Variable Coefficient Std.Error t-Statistic Probability
----------------------------------------------------------------------------------
CONSTANT 46.4281827 13.1917570 3.5194844 0.
crime -0.4848885 0.1826729 -2.6544086 0.
income 0.6289840 0.5359104 1.1736736 0.
----------------------------------------------------------------------------------
<BLANKLINE>
REGRESSION DIAGNOSTICS
MULTICOLLINEARITY CONDITION NUMBER 12.538
<BLANKLINE>
TEST ON NORMALITY OF ERRORS
TEST DF VALUE PROB
Jarque-Bera 2 39.706 0.0000
<BLANKLINE>
DIAGNOSTICS FOR HETEROSKEDASTICITY
RANDOM COEFFICIENTS
```
TEST  DF  VALUE  PROB
Breusch-Pagan test  2  5.767  0.0559
Koenker-Bassett test  2  2.270  0.3214

SPECIFICATION ROBUST TEST
TEST  DF  VALUE  PROB
White  5  2.906  0.7145

If the optional parameters w and spat_diag are passed to pysal.model.spreg.OLS, spatial diagnostics will also be computed for the regression. These include Lagrange multiplier tests and Moran’s I of the residuals. The w parameter is a PySAL spatial weights matrix. In this example, w is built directly from the shapefile columbus.shp, but w can also be read in from a GAL or GWT file. In this case a rook contiguity weights matrix is built, but PySAL also offers queen contiguity, distance weights and k nearest neighbor weights among others. In the example, the Moran’s I of the residuals is 0.204 with a standardized value of 2.592 and a p-value of 0.0095.

```python
>>> w = pysal.lib.weights.Rook.from_shapefile(pysal.lib.examples.get_path("columbus.shp"))
>>> ols = OLS(y, X, w, spat_diag=True, moran=True, name_y="home value", name_x=["income", "crime"], name_ds="columbus")
>>> ols.betas
array([[ 46.42818268],
       [ 0.62898397],
       [-0.48488854]])
>>> print round(ols.moran_res[0],3)
0.204
>>> print round(ols.moran_res[1],3)
2.592
>>> print round(ols.moran_res[2],4)
0.0095
```

Attributes

- **summary**  [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
- **betas**  [array] kx1 array of estimated coefficients
- **u**  [array] nx1 array of residuals
- **predy**  [array] nx1 array of predicted y values
- **n**  [integer] Number of observations
- **k**  [integer] Number of variables for which coefficients are estimated (including the constant)
- **y**  [array] nx1 array for dependent variable
- **x**  [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
- **robust**  [string] Adjustment for robust standard errors
- **mean_y**  [float] Mean of dependent variable
- **std_y**  [float] Standard deviation of dependent variable
- **vm**  [array] Variance covariance matrix (kxk)
- **r2**  [float] R squared
ar2  [float] Adjusted R squared
utu  [float] Sum of squared residuals
sig2  [float] Sigma squared used in computations
sig2ML [float] Sigma squared (maximum likelihood)
fs stat [tuple] Statistic (float), p-value (float)
logll [float] Log likelihood
aic  [float] Akaike information criterion
schwarz [float] Schwarz information criterion
std_err [array] 1xk array of standard errors of the betas
t_stat [list of tuples] t statistic; each tuple contains the pair (statistic, p-value), where each is a float
mulColli [float] Multicollinearity condition number
jarque_bera [dictionary] ‘jb’: Jarque-Bera statistic (float); ‘pvalue’: p-value (float); ‘df’: degrees of freedom (int)
breusch_pagan [dictionary] ‘bp’: Breusch-Pagan statistic (float); ‘pvalue’: p-value (float); ‘df’: degrees of freedom (int)
koenker_bassett [dictionary] ‘kb’: Koenker-Bassett statistic (float); ‘pvalue’: p-value (float); ‘df’: degrees of freedom (int)
white [dictionary] ‘wh’: White statistic (float); ‘pvalue’: p-value (float); ‘df’: degrees of freedom (int)
lm_error [tuple] Lagrange multiplier test for spatial error model; tuple contains the pair (statistic, p-value), where each is a float
lm_lag [tuple] Lagrange multiplier test for spatial lag model; tuple contains the pair (statistic, p-value), where each is a float
rlm_error [tuple] Robust lagrange multiplier test for spatial error model; tuple contains the pair (statistic, p-value), where each is a float
rlm_lag [tuple] Robust lagrange multiplier test for spatial lag model; tuple contains the pair (statistic, p-value), where each is a float
lm_sarma [tuple] Lagrange multiplier test for spatial SARMA model; tuple contains the pair (statistic, p-value), where each is a float
moran_res [tuple] Moran’s I for the residuals; tuple containing the triple (Moran’s I, standardized Moran’s I, p-value)
name_y [string] Name of dependent variable for use in output
name_x [list of strings] Names of independent variables for use in output
name_w [string] Name of weights matrix for use in output
name_gwk [string] Name of kernel weights matrix for use in output
name_ds [string] Name of dataset for use in output
title [string] Name of the regression method used
sig2n  [float] Sigma squared (computed with n in the denominator)
sig2n_k [float] Sigma squared (computed with n-k in the denominator)
import numpy as np
import pandas as pd
import scipy
import scipy.stats
import scipy.optimize

xtx = np.dot(X.T, X)
xtxi = np.linalg.inv(xtx)

__init__(y, x, w=None, robust=None, gwk=None, sig2n_k=True, spat_diag=False, moran=False, white_test=False, vm=False, name_y=None, name_x=None, name_w=None, name_gwk=None, name_ds=None)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, x, robust, gwk, sig2n_k, . . . ) Initialize self.

Attributes

mean_y
sig2n
sig2n_k
std_y
utu
vm

class pysal.model.spreg.ML_Lag(y, x, w, method='full', epsilon=1e-07, spat_diag=False, vm=False, name_y=None, name_x=None, name_w=None, name_gwk=None, name_ds=None)

ML estimation of the spatial lag model with all results and diagnostics; Anselin (1988) [Anselin1988]

Parameters

y [array] nx1 array for dependent variable
x [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
w [pysal W object] Spatial weights object
method [string] if ‘full’, brute force calculation (full matrix expressions) if ‘ord’, Ord eigen-value method
epsilon [float] tolerance criterion in minimize_scalar function and inverse_product
spat_diag [boolean] if True, include spatial diagnostics
vm [boolean] if True, include variance-covariance matrix in summary results
name_y [string] Name of dependent variable for use in output
name_x [list of strings] Names of independent variables for use in output
name_w [string] Name of weights matrix for use in output
name_ds [string] Name of dataset for use in output

Attributes

betas [array] (k+1)x1 array of estimated coefficients (rho first)
**rho** [float] estimate of spatial autoregressive coefficient

**u** [array] nx1 array of residuals

**predy** [array] nx1 array of predicted y values

**n** [integer] Number of observations

**k** [integer] Number of variables for which coefficients are estimated (including the constant, excluding the rho)

**y** [array] nx1 array for dependent variable

**x** [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant

**method** [string] log Jacobian method if ‘full’: brute force (full matrix computations)

**epsilon** [float] tolerance criterion used in minimize_scalar function and inverse_product

**mean_y** [float] Mean of dependent variable

**std_y** [float] Standard deviation of dependent variable

**vm** [array] Variance covariance matrix (k+1 x k+1), all coefficients

**vm1** [array] Variance covariance matrix (k+2 x k+2), includes sig2

**sig2** [float] Sigma squared used in computations

**logll** [float] maximized log-likelihood (including constant terms)

**aic** [float] Akaike information criterion

**schwarz** [float] Schwarz criterion

**predy_e** [array] predicted values from reduced form

**e_pred** [array] prediction errors using reduced form predicted values

**pr2** [float] Pseudo R squared (squared correlation between y and ypred)

**pr2_e** [float] Pseudo R squared (squared correlation between y and ypred_e (using reduced form))

**utu** [float] Sum of squared residuals

**std_err** [array] 1xk array of standard errors of the betas

**z_stat** [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float

**name_y** [string] Name of dependent variable for use in output

**name_x** [list of strings] Names of independent variables for use in output

**name_w** [string] Name of weights matrix for use in output

**name_ds** [string] Name of dataset for use in output

**title** [string] Name of the regression method used

### Examples

```python
>>> import numpy as np
>>> import pysal.lib
```
```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path("baltim.dbf"),'r')
>>> ds_name = "baltim.dbf"
>>> y_name = "PRICE"
>>> y = np.array(db.by_col(y_name)).T
>>> y.shape = (len(y),1)
>>> x_names = ["NROOM","NBATH","PATIO","FIREPL","AC","GAR","AGE","LOTSZ","SQFT"]

>>> x = np.array([db.by_col(var) for var in x_names]).T
>>> ww = ps.open(ps.examples.get_path("baltim_q.gal"))
>>> w = ww.read()
>>> ww.close()
>>> w_name = "baltim_q.gal"
>>> w.transform = 'r'

>>> mllag = ML_Lag(y,x,w,name_y=y_name,name_x=x_names, name_w=w_name,name_ds=ds_name)  #doctest: +SKIP
>>> np.around(mllag.betas, decimals=4)  #doctest: +SKIP
array([[ 4.3675],
       [ 0.7502],
       [ 5.6116],
       [ 7.0497],
       [ 7.7246],
       [ 6.1231],
       [ 4.6375],
       [-0.1107],
       [ 0.0679],
       [ 0.0794],
       [ 0.4259]])

>>> '{0:.6f}'.format(mllag.rho)  #doctest: +SKIP
'0.425885'

>>> '{0:.6f}'.format(mllag.mean_y)  #doctest: +SKIP
'44.307180'

>>> '{0:.6f}'.format(mllag.std_y)  #doctest: +SKIP
'23.606077'

>>> np.around(np.diag(mllag.vm1), decimals=4)  #doctest: +SKIP
array([ 23.8716,  1.1222,  3.0593,  7.3416,  5.6695,  5.4698,
       [ 2.8684,  0.0026,  0.0002,  0.0266,
       [ 0.0032,  220.1292]])

>>> '{0:.6f}'.format(mllag.sig2)  #doctest: +SKIP
'151.458698'

>>> '{0:.6f}'.format(mllag.logll)  #doctest: +SKIP
'1687.874348'

>>> '{0:.6f}'.format(mllag.schwarz)  #doctest: +SKIP
'1724.744787'
```
>>> "[0:.6f]".format(mllag.pr2) #doctest: +SKIP
'0.727081'

>>> "[0:.4f]".format(mllag.pr2_e) #doctest: +SKIP
'0.7062'

>>> "[0:.4f]".format(mllag.utu) #doctest: +SKIP
'31957.7853'

>>> np.around(mllag.std_err, decimals=4) #doctest: +SKIP
array([ 4.8859, 1.0593, 1.7491, 2.7095, 2.3811, 2.3388, 1.6936, 0.0508, 0.0146, 0.1631, 0.057])

>>> np.around(mllag.z_stat, decimals=4) #doctest: +SKIP
array([[ 0.8939, 0.3714], [ 0.7082, 0.4788], [ 3.2083, 0.0013], [ 2.6018, 0.0093], [ 3.2442, 0.0012], [ 2.6181, 0.0088], [ 2.7382, 0.0062], [-2.1780, 0.0294], [ 4.6487, 0.0013], [ 0.4866, 0.6266], [ 7.4775, 0.0012], [ 2.6181, 0.0088], [ 2.7382, 0.0062], [-2.1780, 0.0294], [ 4.6487, 0.0013], [ 0.4866, 0.6266], [ 7.4775, 0.0012]])

>>> mllag.name_y #doctest: +SKIP
'PRICE'

>>> mllag.name_x #doctest: +SKIP
['CONSTANT', 'NROOM', 'NBATH', 'PATIO', 'FIREPL', 'AC', 'GAR', 'AGE', 'LOTSZ', 'SQFT', 'W_PRICE'

>>> mllag.name_w #doctest: +SKIP
'baltim_q.gal'

>>> mllag.name_ds #doctest: +SKIP
'baltim.dbf'

>>> mllag.title #doctest: +SKIP
'MAXIMUM LIKELIHOOD SPATIAL LAG (METHOD = FULL)'

>>> mllag = ML_Lag(y,x,w,method='ord',name_y=y_name,name_x=x_names, name_w=w_name,name_ds=ds_name) #doctest: +SKIP

>>> np.around(mllag.betas, decimals=4) #doctest: +SKIP
array([[ 4.3675], [ 0.7502], [ 5.6116], [ 7.0497], [ 7.7246], [ 6.1231], [ 4.6375], [-0.1107], [ 0.0679], [ 0.0794], [ 0.4259]])
>>> np.around(np.diag(mllag.vm), decimals=4) #doctest: +SKIP
array([ 23.8716,  1.1222,  3.0593,  7.3416,  5.6695,  5.4698, 
        2.8684,  0.0026,  0.0002,  0.0266,  0.0032])

>>> "[0:.6f]".format(mllag.sig2) #doctest: +SKIP
'151.458698'

>>> "[0:.6f]".format(mllag.logl) #doctest: +SKIP
'-832.937174'

>>> "[0:.6f]".format(mllag.aic) #doctest: +SKIP
'1687.874348'

>>> "[0:.6f]".format(mllag.schwarz) #doctest: +SKIP
'1724.744787'

>>> "[0:.6f]".format(mllag.pr2) #doctest: +SKIP
'0.727081'

>>> "[0:.6f]".format(mllag.pr2_e) #doctest: +SKIP
'0.706198'

>>> "[0:.4f]".format(mllag.utu) #doctest: +SKIP
'31957.7853'

>>> np.around(mllag.std_err, decimals=4) #doctest: +SKIP
array([ 4.8859,  1.0593,  1.7491,  2.7095,  2.3811,  2.3388,  1.6936, 
        0.0508,  0.0146,  0.1631,  0.0570])

>>> np.around(mllag.z_stat, decimals=4) #doctest: +SKIP
array([[ 0.8939,  0.3714], [ 0.7082,  0.4788], [ 3.2083,  0.0013], [ 2.6018,  0.0093], [ 3.2442,  0.0012], [ 2.6181,  0.0088], [ 2.7382,  0.0062], [-2.1780,  0.0294], [ 4.6487,  0.0000], [ 0.4866,  0.6266], [ 7.4775,  0.0000]])

>>> mllag.name_y #doctest: +SKIP
'PRICE'

>>> mllag.name_x #doctest: +SKIP
['CONSTANT', 'NROOM', 'NBATH', 'PATIO', 'FIREPL', 'AC', 'GAR', 'AGE', 'LOTSZ', 'SQFT', 'W_PRICE']

>>> mllag.name_w #doctest: +SKIP
'baltim_q.gal'

>>> mllag.name_ds #doctest: +SKIP
'baltim.dbf'

>>> mllag.title #doctest: +SKIP
'MAXIMUM LIKELIHOOD SPATIAL LAG (METHOD = ORD)'

__init__(y, x, w, method='full', epsilon=1e-07, spat_diag=False, vm=False, name_y=None, name_x=None, name_w=None, name_ds=None)
Initialize self. See help(type(self)) for accurate signature.
### Methods

```python
__init__(y, x[, method, epsilon, ...]) Initialize self.
```

### Attributes

- `mean_y`
- `sig2n`
- `sig2n_k`
- `std_y`
- `utu`
- `vm`

### pysal.model.spreg.ML_Error

**class pysal.model.spreg.ML_Error**

```python
class pysal.model.spreg.ML_Error(y, x, w[, method='full', epsilon=1e-07, spat_diag=False, vm=False, name_y=None, name_x=None, name_w=None, name_ds=None])
```

ML estimation of the spatial lag model with all results and diagnostics; Anselin (1988) [Anselin1988]

#### Parameters

- `y` [array] nx1 array for dependent variable
- `x` [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
- `w` [Sparse matrix] Spatial weights sparse matrix
- `epsilon` [float] tolerance criterion in minimize_scalar function and inverse_product
- `spat_diag` [boolean] if True, include spatial diagnostics (not implemented yet)
- `vm` [boolean] if True, include variance-covariance matrix in summary results
- `name_y` [string] Name of dependent variable for use in output
- `name_x` [list of strings] Names of independent variables for use in output
- `name_w` [string] Name of weights matrix for use in output
- `name_ds` [string] Name of dataset for use in output

#### Examples

```python
>>> import numpy as np
>>> import pysal.lib
>>> from pysal.lib import examples
>>> np.set_printoptions(suppress=True)  # prevent scientific format
>>> db = pysal.lib.io.open(examples.get_path("south.dbf"),'r')
>>> y_name = "HR90"
>>> y = np.array(db.by_col(y_name))
>>> y.shape = (len(y),1)
```

(continues on next page)
>>> x_names = ['RD90', 'PS90', 'UE90', 'DV90']
>>> x = np.array([db.by_col(var) for var in x_names]).T
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("south_q.gal"))
>>> w = w.read()
>>> w.close()
>>> w_name = "south_q.gal"
>>> w.transform = 'r'
>>> mlerr = ML_Error(y, x, w, name_y=y_name, name_x=x_names, name_w=w_name,
    name_ds=ds_name) #doctest: +SKIP
>>> np.around(mlerr.betas, decimals=4) #doctest: +SKIP
array([[ 6.1492],
       [ 4.4024],
       [ 1.7784],
       [-0.3781],
       [ 0.4858],
       [ 0.2991]])
>>>
>>> '{0:.4f}'.format(mlerr.lam) #doctest: +SKIP
'0.2991'
>>> '{0:.4f}'.format(mlerr.mean_y) #doctest: +SKIP
'9.5493'
>>> '{0:.4f}'.format(mlerr.std_y) #doctest: +SKIP
'7.0389'
>>> np.around(np.diag(mlerr.vm), decimals=4) #doctest: +SKIP
array([ 1.0648,  0.0555,  0.0454,  0.0061,  0.0148,  0.0014])
>>> np.around(mlerr.sig2, decimals=4) #doctest: +SKIP
array([[ 32.4069]])
>>> '{0:.4f}'.format(mlerr.logll) #doctest: +SKIP
'-4471.4071'
>>> '{0:.4f}'.format(mlerr.aic) #doctest: +SKIP
'8952.8141'
>>> '{0:.4f}'.format(mlerr.schwarz) #doctest: +SKIP
'8979.0779'
>>> '{0:.4f}'.format(mlerr.pr2) #doctest: +SKIP
'0.3058'
>>> '{0:.4f}'.format(mlerr.utu) #doctest: +SKIP
'48534.9148'
>>> np.around(mlerr.std_err, decimals=4) #doctest: +SKIP
array([[ 1.0319,  0.2355,  0.2132,  0.0784,  0.1217,  0.0378]])
>>> np.around(mlerr.z_stat, decimals=4) #doctest: +SKIP
array([[ 5.9593,  0.   ],
       [18.6902,  0.   ],
       [ 8.3422,  0.   ],
       [-4.8233,  0.   ],
       [ 3.9913,  0.0001],
       [ 7.9089,  0.   ]])
>>> mlerr.name_y #doctest: +SKIP
'HR90'
>>> mlerr.name_x #doctest: +SKIP
['CONSTANT', 'RD90', 'PS90', 'UE90', 'DV90', 'lambda']
>>> mlerr.name_w #doctest: +SKIP
'south_q.gal'
>>> mlerr.name_ds #doctest: +SKIP
'south.dbf'
>>> mlerr.title #doctest: +SKIP
'MAXIMUM LIKELIHOOD SPATIAL ERROR (METHOD = FULL)'

Attributes
betas [array] (k+1)x1 array of estimated coefficients (rho first)
lam [float] estimate of spatial autoregressive coefficient
u [array] nx1 array of residuals
e_filtered [array] nx1 array of spatially filtered residuals
predy [array] nx1 array of predicted y values
n [integer] Number of observations
k [integer] Number of variables for which coefficients are estimated (including the constant, excluding lambda)
y [array] nx1 array for dependent variable
x [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
method [string] log Jacobian method if ‘full’: brute force (full matrix computations)
epsilon [float] tolerance criterion used in minimize_scalar function and inverse_product
mean_y [float] Mean of dependent variable
std_y [float] Standard deviation of dependent variable
varb [array] Variance covariance matrix (k+1 x k+1) - includes var(lambda)
vm1 [array] variance covariance matrix for lambda, sigma (2 x 2)
sig2 [float] Sigma squared used in computations
logll [float] maximized log-likelihood (including constant terms)
pr2 [float] Pseudo R squared (squared correlation between y and ypred)
utu [float] Sum of squared residuals
std_err [array] 1xk array of standard errors of the betas
z_stat [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float
name_y [string] Name of dependent variable for use in output
name_x [list of strings] Names of independent variables for use in output
name_w [string] Name of weights matrix for use in output
name_ds [string] Name of dataset for use in output
title [string] Name of the regression method used

Methods

__init__(y, x, w, method='full', epsilon=1e-07, spat_diag=False, vm=False, name_y=None, name_x=None, name_w=None, name_ds=None)
Initialize self. See help(type(self)) for accurate signature.
Methods

```
__init__(y, x, w[, method, epsilon, ...]) Initialize self.
get_x_lag(w, regimes_att)
```

Attributes

```
mean_y
sig2n
sig2n_k
std_y
utu
vm
```

```
pysal.model.spreg.GM_Lag
``` 

class pysal.model.spreg.GM_Lag(y, x, yend=None, q=None, w=None, w_lags=1, lag_q=True, robust=None, gwk=None, sig2n_k=False, spat_diag=False, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_gwk=None, name_ds=None)

Spatial two stage least squares (S2SLS) with results and diagnostics; Anselin (1988) [Anselin1988]

Parameters

- `y` [array] nx1 array for dependent variable
- `x` [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
- `yend` [array] Two dimensional array with n rows and one column for each endogenous variable
- `q` [array] Two dimensional array with n rows and one column for each external exogenous variable to use as instruments (note: this should not contain any variables from x); cannot be used in combination with h
- `w` [pysal W object] Spatial weights object
- `w_lags` [integer] Orders of W to include as instruments for the spatially lagged dependent variable. For example, w_lags=1, then instruments are WX; if w_lags=2, then WX, WWX; and so on.
- `lag_q` [boolean] If True, then include spatial lags of the additional instruments (q).
- `robust` [string] If ‘white’, then a White consistent estimator of the variance-covariance matrix is given. If ‘hac’, then a HAC consistent estimator of the variance-covariance matrix is given. Default set to None.
- `gwk` [pysal W object] Kernel spatial weights needed for HAC estimation. Note: matrix must have ones along the main diagonal.
- `sig2n_k` [boolean] If True, then use n-k to estimate sigma^2. If False, use n.
- `spat_diag` [boolean] If True, then compute Anselin-Kelejian test
- `vm` [boolean] If True, include variance-covariance matrix in summary results
- `name_y` [string] Name of dependent variable for use in output
**Examples**

We first need to import the needed modules, namely numpy to convert the data we read into arrays that `spreg` understands and *pysal* to perform all the analysis. Since we will need some tests for our model, we also import the diagnostics module.

```python
>>> import numpy as np
>>> import pysal.lib
>>> import pysal.model.spreg.diagnostics as D
```

Open data on Columbus neighborhood crime (49 areas) using `pysal.lib.io.open()`. This is the DBF associated with the Columbus shapefile. Note that `pysal.lib.io.open()` also reads data in CSV format; since the actual class requires data to be passed in as numpy arrays, the user can read their data in using any method.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path("columbus.dbf"), 'r')
```

Extract the HOVAL column (home value) from the DBF file and make it the dependent variable for the regression. Note that *PySAL* requires this to be an numpy array of shape (n, 1) as opposed to the also common shape of (n,) that other packages accept.

```python
>>> y = np.array(db.by_col("HOVAL"))
>>> y = np.reshape(y, (49,1))
```

Extract INC (income) and CRIME (crime rates) vectors from the DBF to be used as independent variables in the regression. Note that *PySAL* requires this to be an nxj numpy array, where j is the number of independent variables (not including a constant). By default this model adds a vector of ones to the independent variables passed in, but this can be overridden by passing constant=False.

```python
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("CRIME"))
>>> X = np.array(X).T
```

Since we want to run a spatial error model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations into the error component of the model. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `columbus.shp`.

```python
>>> w = pysal.lib.weights.Rook.from_shapefile(pysal.lib.examples.get_path("columbus.shp"))
```

Unless there is a good reason not to do it, the weights have to be row-standardized so every row of the matrix sums to one. Among other things, this allows to interpret the spatial lag of a variable as the average value of the neighboring observations. In *PySAL*, this can be easily performed in the following way:

```python
>>> w.transform = 'r'
```
This class runs a lag model, which means that includes the spatial lag of the dependent variable on the right-hand side of the equation. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional. The default most basic model to be run would be:

```python
>>> reg=GM_Lag(y, X, w=w, w_lags=2, name_x=['inc', 'crime'], name_y='hoval', name_ds='columbus') >>> reg.betas
array([[ 45.30170561],
       [ 0.62088862],
       [ 0.02836221]])
```

Once the model is run, we can obtain the standard error of the coefficient estimates by calling the diagnostics module:

```python
>>> D.se_betas(reg)
array([17.91278862, 0.52486082, 0.18228150, 0.31740089])
```

But we can also run models that incorporates corrected standard errors following the White procedure. For that, we will have to include the optional parameter `robust='white'`:

```python
>>> reg=GM_Lag(y, X, w=w, w_lags=2, robust='white', name_x=['inc', 'crime'], name_y='hoval', name_ds='columbus') >>> reg.betas
array([[ 45.30170561],
       [ 0.62088862],
       [ 0.02836221]])
```

And we can access the standard errors from the model object:

```python
>>> reg.std_err
array([20.47077481, 0.50613931, 0.20138425, 0.38028295])
```

The class is flexible enough to accomodate a spatial lag model that, besides the spatial lag of the dependent variable, includes other non-spatial endogenous regressors. As an example, we will assume that CRIME is actually endogenous and we decide to instrument for it with DISCBD (distance to the CBD). We reload the X including INC only and define CRIME as endogenous and DISCBD as instrument:

```python
>>> X = np.array(db.by_col("INC")) >>> X = np.reshape(X, (49,1)) >>> yd = np.array(db.by_col("CRIME")) >>> yd = np.reshape(yd, (49,1)) >>> q = np.array(db.by_col("DISCBD")) >>> q = np.reshape(q, (49,1))
```

And we can run the model again:

```python
>>> reg=GM_Lag(y, X, w=w, yend=yd, q=q, w_lags=2, name_x=['inc'], name_y='hoval', name_yend=['crime'], name_q=['discbd'], name_ds='columbus') >>> reg.betas
array([[100.79359082],
       [-0.50215501],
       [-1.14881711],
       [-0.38235022]])
```

Once the model is run, we can obtain the standard error of the coefficient estimates by calling the diagnostics module:
Attributes

- **summary** [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
- **betas** [array] kx1 array of estimated coefficients
- **u** [array] nx1 array of residuals
- **e_pred** [array] nx1 array of residuals (using reduced form)
- **predy** [array] nx1 array of predicted y values
- **predy_e** [array] nx1 array of predicted y values (using reduced form)
- **n** [integer] Number of observations
- **k** [integer] Number of variables for which coefficients are estimated (including the constant)
- **kstar** [integer] Number of endogenous variables.
- **y** [array] nx1 array for dependent variable
- **x** [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
- **yend** [array] Two dimensional array with n rows and one column for each endogenous variable
- **q** [array] Two dimensional array with n rows and one column for each external exogenous variable used as instruments
- **z** [array] nxk array of variables (combination of x and yend)
- **h** [array] nxl array of instruments (combination of x and q)
- **robust** [string] Adjustment for robust standard errors
- **mean_y** [float] Mean of dependent variable
- **std_y** [float] Standard deviation of dependent variable
- **vm** [array] Variance covariance matrix (kxk)
- **pr2** [float] Pseudo R squared (squared correlation between y and ypred)
- **pr2_e** [float] Pseudo R squared (squared correlation between y and ypred_e (using reduced form))
- **utu** [float] Sum of squared residuals
- **sig2** [float] Sigma squared used in computations
- **std_err** [array] 1xk array of standard errors of the betas
- **z_stat** [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float
- **ak_test** [tuple] Anselin-Kelejian test; tuple contains the pair (statistic, p-value)
- **name_y** [string] Name of dependent variable for use in output
- **name_x** [list of strings] Names of independent variables for use in output
- **name_yend** [list of strings] Names of endogenous variables for use in output
name_z [list of strings] Names of exogenous and endogenous variables for use in output
name_q [list of strings] Names of external instruments
name_h [list of strings] Names of all instruments used in output
name_w [string] Name of weights matrix for use in output
name_gwk [string] Name of kernel weights matrix for use in output
name_ds [string] Name of dataset for use in output
title [string] Name of the regression method used
sig2n [float] Sigma squared (computed with n in the denominator)
sig2n_k [float] Sigma squared (computed with n-k in the denominator)
hth [float] H'H
hthi [float] (H'H)^{-1}
varb [array] (Z'H (H'H)^{-1} H'Z)^{-1}
zthhthi [array] Z'H(H'H)^{-1}
pfora1a2 [array] n(zthhthi)'varb

__init__(y, x, yend=None, q=None, w=None, w_lags=1, lag_q=True, robust=None, gwk=None, sig2n_k=False, spat_diag=False, vm=False, name_y=None, name_x=None, name_w=None, name_gwk=None, name_ds=None)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, x, yend, q, w, w_lags, lag_q, ...)

Attributes

mean_y
pfora1a2
sig2n
sig2n_k
std_y
utu
vm

pysal.model.spreg.GM_Error

class pysal.model.spreg.GM_Error(y, x, w, vm=False, name_y=None, name_x=None, name_w=None, name_ds=None)

Parameters

y [array] nx1 array for dependent variable
x [array] Two dimensional array with n rows and one column for each independent (exogenous)
variable, excluding the constant

**w** [pysal W object] Spatial weights object (always needed)

**vm** [boolean] If True, include variance-covariance matrix in summary results

**name_y** [string] Name of dependent variable for use in output

**name_x** [list of strings] Names of independent variables for use in output

**name_w** [string] Name of weights matrix for use in output

**name_ds** [string] Name of dataset for use in output

### Examples

We first need to import the needed modules, namely numpy to convert the data we read into arrays that `spreg` understands and `pysal` to perform all the analysis.

```python
>>> import pysal.lib
>>> import numpy as np
```

Open data on Columbus neighborhood crime (49 areas) using `pysal.lib.io.open()`. This is the DBF associated with the Columbus shapefile. Note that `pysal.lib.io.open()` also reads data in CSV format; since the actual class requires data to be passed in as numpy arrays, the user can read their data in using any method.

```python
>>> dbf = pysal.lib.io.open(pysal.lib.examples.get_path('columbus.dbf'), 'r')
```

Extract the HOV AL column (home values) from the DBF file and make it the dependent variable for the regression. Note that PySAL requires this to be a numpy array of shape (n, 1) as opposed to the also common shape of (n,) that other packages accept.

```python
>>> y = np.array([dbf.by_col('HOVAL')]).T
```

Extract CRIME (crime) and INC (income) vectors from the DBF to be used as independent variables in the regression. Note that PySAL requires this to be an nxj numpy array, where j is the number of independent variables (not including a constant). By default this class adds a vector of ones to the independent variables passed in.

```python
>>> names_to_extract = ['INC', 'CRIME']
>>> x = np.array([dbf.by_col(name) for name in names_to_extract]).T
```

Since we want to run a spatial error model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations into the error component of the model. To do that, we can open an already existing gal file or create a new one. In this case, we will use `columbus.gal`, which contains contiguity relationships between the observations in the Columbus dataset we are using throughout this example. Note that, in order to read the file, not only to open it, we need to append `.read()` at the end of the command.

```python
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("columbus.gal"), 'r').read()
```

Unless there is a good reason not to do it, the weights have to be row-standardized so every row of the matrix sums to one. Among other things, this allows to interpret the spatial lag of a variable as the average value of the neighboring observations. In PySAL, this can be easily performed in the following way:

```python
>>> w.transform='r'
```
We are all set with the preliminars, we are good to run the model. In this case, we will need the variables and the weights matrix. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional.

```
>>> model = GM_Error(y, x, w=w, name_y='hoval', name_x=['income', 'crime'], name_ds='columbus')
```

Once we have run the model, we can explore a little bit the output. The regression object we have created has many attributes so take your time to discover them. Note that because we are running the classical GMM error model from 1998/99, the spatial parameter is obtained as a point estimate, so although you get a value for it (there are for coefficients under model.betas), you cannot perform inference on it (there are only three values in model.se_betas).

```
>>> print model.name_x
['CONSTANT', 'income', 'crime', 'lambda']

>>> np.around(model.betas, decimals=4)
array([[ 47.6946],
       [ 0.7105],
       [-0.5505],
       [ 0.3257]])

>>> np.around(model.std_err, decimals=4)
array([ 12.4120,  0.5044,  0.1785])

>>> np.around(model.z_stat, decimals=6)  # doctest: +SKIP
array([[ 3.84261100e+00,  1.22000000e-04],
       [ 1.40839200e+00,  1.59015000e-01],
       [-3.08424700e+00,  2.04100000e-03]])

>>> round(model.sig2,4)
198.5596
```

Attributes

- **summary** [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
- **betas** [array] kx1 array of estimated coefficients
- **u** [array] nx1 array of residuals
- **e_filtered** [array] nx1 array of spatially filtered residuals
- **predy** [array] nx1 array of predicted y values
- **n** [integer] Number of observations
- **k** [integer] Number of variables for which coefficients are estimated (including the constant)
- **y** [array] nx1 array for dependent variable
- **x** [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
- **mean_y** [float] Mean of dependent variable
- **std_y** [float] Standard deviation of dependent variable
- **pr2** [float] Pseudo R squared (squared correlation between y and ypred)
- **vm** [array] Variance covariance matrix (kxk)
- **sig2** [float] Sigma squared used in computations
- **std_err** [array] 1xk array of standard errors of the betas
z_stat [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float

name_y [string] Name of dependent variable for use in output

name_x [list of strings] Names of independent variables for use in output

name_w [string] Name of weights matrix for use in output

name_ds [string] Name of dataset for use in output

title [string] Name of the regression method used

__init__ (y, x, w[, vm=False, name_y=None, name_x=None, name_w=None, name_ds=None])
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (y, x, w[, vm, name_y, name_x, ...]) Initialize self.

Attributes

mean_y

std_y

pysal.model.spreg.GM_Error_Het

class pysal.model.spreg.GM_Error_Het (y, x, w, max_iter=1, epsilon=1e-05, step1c=False, vm=False, name_y=None, name_x=None, name_w=None, name_ds=None)
GMM method for a spatial error model with heteroskedasticity, with results and diagnostics; based on Arriaz et al [Arriaz2010], following Anselin [Anselin2011].

Parameters

y [array] nx1 array for dependent variable

x [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant

w [pysal W object] Spatial weights object

max_iter [int] Maximum number of iterations of steps 2a and 2b from Arriaz et al. Note: epsilon provides an additional stop condition.

epsilon [float] Minimum change in lambda required to stop iterations of steps 2a and 2b from Arriaz et al. Note: max_iter provides an additional stop condition.

step1c [boolean] If True, then include Step 1c from Arriaz et al.

vm [boolean] If True, include variance-covariance matrix in summary results

name_y [string] Name of dependent variable for use in output

name_x [list of strings] Names of independent variables for use in output

name_w [string] Name of weights matrix for use in output

name_ds [string] Name of dataset for use in output
Examples

We first need to import the needed modules, namely numpy to convert the data we read into arrays that spreg understands and pysal to perform all the analysis.

```python
>>> import numpy as np
>>> import pysal.lib
```

Open data on Columbus neighborhood crime (49 areas) using pysal.lib.io.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.lib.io.open() also reads data in CSV format; since the actual class requires data to be passed in as numpy arrays, the user can read their data in using any method.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path('columbus.dbf'),'r')
```

Extract the HOV AL column (home values) from the DBF file and make it the dependent variable for the regression. Note that PySAL requires this to be an n by 1 array of shape (n, 1) as opposed to the also common shape of (n,) that other packages accept.

```python
>>> y = np.array(db.by_col("HOVAL"))
```

```python
>>> y = np.reshape(y, (49,1))
```

Extract INC (income) and CRIME (crime) vectors from the DBF to be used as independent variables in the regression. Note that PySAL requires this to be an nxJ numpy array, where J is the number of independent variables (not including a constant). By default this class adds a vector of ones to the independent variables passed in.

```python
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("CRIME"))
>>> X = np.array(X).T
```

Since we want to run a spatial error model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations into the error component of the model. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from columbus.shp.

```python
>>> w = pysal.lib.weights.Rook.from_shapefile(pysal.lib.examples.get_path("columbus.shp"))
```

Unless there is a good reason not to do it, the weights have to be row-standardized so every row of the matrix sums to one. Among other things, this allows to interpret the spatial lag of a variable as the average value of the neighboring observations. In PySAL, this can be easily performed in the following way:

```python
>>> w.transform = 'r'
```

We are all set with the preliminaries, we are good to run the model. In this case, we will need the variables and the weights matrix. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional.

```python
>>> reg = GM_Error_Het(y, X, w=w, step1c=True, name_y='home value', name_x=['income', 'crime'], name_ds='columbus')
```

Once we have run the model, we can explore a little bit the output. The regression object we have created has many attributes so take your time to discover them. This class offers an error model that explicitly accounts for heteroskedasticity and that unlike the models from spreg.error_sp, it allows for inference on the spatial parameter.
Hence, we find the same number of betas as of standard errors, which we calculate taking the square root of the diagonal of the variance-covariance matrix:

```python
>>> print np.around(np.hstack((reg.betas, np.sqrt(reg.vm.diagonal()).reshape(4, 1))), 4)
[[ 47.9963  11.479 ]
 [  0.7105   0.3681]
 [-0.5588   0.1616]
 [  0.4118   0.168 ]]
```

Attributes

- **summary** [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
- **betas** [array] kx1 array of estimated coefficients
- **u** [array] nx1 array of residuals
- **e_filtered** [array] nx1 array of spatially filtered residuals
- **predy** [array] nx1 array of predicted y values
- **n** [integer] Number of observations
- **k** [integer] Number of variables for which coefficients are estimated (including the constant)
- **y** [array] nx1 array for dependent variable
- **x** [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
- **iter_stop** [string] Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al.
- **iteration** [integer] Number of iterations of steps 2a and 2b from Arraiz et al.
- **mean_y** [float] Mean of dependent variable
- **std_y** [float] Standard deviation of dependent variable
- **pr2** [float] Pseudo R squared (squared correlation between y and ypred)
- **vm** [array] Variance covariance matrix (kxk)
- **std_err** [array] 1xk array of standard errors of the betas
- **z_stat** [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float
- **xtx** [float] X’X
- **name_y** [string] Name of dependent variable for use in output
- **name_x** [list of strings] Names of independent variables for use in output
- **name_w** [string] Name of weights matrix for use in output
- **name_ds** [string] Name of dataset for use in output
- **title** [string] Name of the regression method used
__init__(y, x, w, max_iter=1, epsilon=1e-05, step1c=False, vm=False, name_y=None, name_x=None, name_w=None, name_ds=None)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, x, w[, max_iter, epsilon, ...]) Initialize self.

Attributes

mean_y
std_y

pysal.model.spreg.GM_Error_Hom

class pysal.model.spreg.GM_Error_Hom(y, x, w, max_iter=1, epsilon=1e-05, A1='hom_sc', vm=False, name_y=None, name_x=None, name_w=None, name_ds=None)
GMM method for a spatial error model with homoskedasticity, with results and diagnostics; based on Drukker et al. (2013) [Drukker2013], following Anselin (2011) [Anselin2011].

Parameters

y [array] nx1 array for dependent variable
x [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
w [pysal W object] Spatial weights object
max_iter [int] Maximum number of iterations of steps 2a and 2b from Arraiz et al. Note: epsilon provides an additional stop condition.
epsilon [float] Minimum change in lambda required to stop iterations of steps 2a and 2b from Arraiz et al. Note: max_iter provides an additional stop condition.
A1 [string] If A1='het', then the matrix A1 is defined as in Arraiz et al. If A1='hom', then as in Anselin (2011). If A1='hom_sc' (default), then as in Drukker, Egger and Prucha (2010) and Drukker, Prucha and Raciborski (2010).
vm [boolean] If True, include variance-covariance matrix in summary results
name_y [string] Name of dependent variable for use in output
name_x [list of strings] Names of independent variables for use in output
name_w [string] Name of weights matrix for use in output
name_ds [string] Name of dataset for use in output

Examples

We first need to import the needed modules, namely numpy to convert the data we read into arrays that spreg understands and pysal to perform all the analysis.
>>> import numpy as np
>>> import pysal.lib

Open data on Columbus neighborhood crime (49 areas) using pysal.lib.io.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.lib.io.open() also reads data in CSV format; since the actual class requires data to be passed in as numpy arrays, the user can read their data in using any method.

>>> db = pysal.lib.io.open(pysal.lib.examples.get_path('columbus.dbf'),'r')

Extract the HOV AL column (home values) from the DBF file and make it the dependent variable for the regression. Note that PySAL requires this to be an numpy array of shape (n, 1) as opposed to the also common shape of (n, ) that other packages accept.

>>> y = np.array(db.by_col("HOVAL"))
>>> y = np.reshape(y, (49,1))

Extract INC (income) and CRIME (crime) vectors from the DBF to be used as independent variables in the regression. Note that PySAL requires this to be an nxj numpy array, where j is the number of independent variables (not including a constant). By default this class adds a vector of ones to the independent variables passed in.

>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("CRIME"))
>>> X = np.array(X).T

Since we want to run a spatial error model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations into the error component of the model. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from columbus.shp.

>>> w = pysal.lib.weights.Rook.from_shapefile(pysal.lib.examples.get_path("columbus.shp"))

Unless there is a good reason not to do it, the weights have to be row-standardized so every row of the matrix sums to one. Among other things, his allows to interpret the spatial lag of a variable as the average value of the neighboring observations. In PySAL, this can be easily performed in the following way:

>>> w.transform = 'r'

We are all set with the preliminaries, we are good to run the model. In this case, we will need the variables and the weights matrix. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional.

>>> reg = GM_Error_Hom(y, X, w=w, A1='hom_sc', name_y='home value', name_x=['income', 'crime'], name_ds='columbus')

Once we have run the model, we can explore a little bit the output. The regression object we have created has many attributes so take your time to discover them. This class offers an error model that assumes homoskedasticity but that unlike the models from spreg.error_sp, it allows for inference on the spatial parameter. This is why you obtain as many coefficient estimates as standard errors, which you calculate taking the square root of the diagonal of the variance-covariance matrix of the parameters:

>>> print np.around(np.hstack((reg.betas,np.sqrt(reg.vm.diagonal()).reshape(4,1))),4)

[ 47.9479 12.3021]
Attributes

- **summary**: [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
- **betas**: [array] kx1 array of estimated coefficients
- **u**: [array] nx1 array of residuals
- **e_filtered**: [array] nx1 array of spatially filtered residuals
- **predy**: [array] nx1 array of predicted y values
- **n**: [integer] Number of observations
- **k**: [integer] Number of variables for which coefficients are estimated (including the constant)
- **y**: [array] nx1 array for dependent variable
- **x**: [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
- **iter_stop**: [string] Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al.
- **iteration**: [integer] Number of iterations of steps 2a and 2b from Arraiz et al.
- **mean_y**: [float] Mean of dependent variable
- **std_y**: [float] Standard deviation of dependent variable
- **pr2**: [float] Pseudo R squared (squared correlation between y and ypred)
- **vm**: [array] Variance covariance matrix (kxk)
- **sig2**: [float] Sigma squared used in computations
- **std_err**: [array] 1xk array of standard errors of the betas
- **z_stat**: [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float
- **xtx**: [float] X’X
- **name_y**: [string] Name of dependent variable for use in output
- **name_x**: [list of strings] Names of independent variables for use in output
- **name_w**: [string] Name of weights matrix for use in output
- **name_ds**: [string] Name of dataset for use in output
- **title**: [string] Name of the regression method used

__init__(y, x, w, max_iter=1, epsilon=1e-05, A1='hom_sc', vm=False, name_y=None, name_x=None, name_w=None, name_ds=None)

Initialize self. See help(type(self)) for accurate signature.

Methods
__init__(y, x, w[, max_iter, epsilon, A1, ...]) Initialize self.

Attributes

- mean_y
- std_y

pysal.model.spreg.GM_Combo

class pysal.model.spreg.GM_Combo(y, x, yend=None, q=None, w=None, w_lags=1, 
lag_q=True, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, 
names_ds=None)

GMM method for a spatial lag and error model with endogenous variables, with results and diagnostics; based on Kelejian and Prucha (1998, 1999) [Kelejian1998] [Kelejian1999].

Parameters

- y [array] nx1 array for dependent variable
- x [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
- yend [array] Two dimensional array with n rows and one column for each endogenous variable
- q [array] Two dimensional array with n rows and one column for each external exogenous variable to use as instruments (note: this should not contain any variables from x)
- w [pysal W object] Spatial weights object (always needed)
- w_lags [integer] Orders of W to include as instruments for the spatially lagged dependent variable. For example, w_lags=1, then instruments are WX; if w_lags=2, then WX, WWX; and so on.
- lag_q [boolean] If True, then include spatial lags of the additional instruments (q).
- vm [boolean] If True, include variance-covariance matrix in summary results
- name_y [string] Name of dependent variable for use in output
- name_x [list of strings] Names of independent variables for use in output
- name_yend [list of strings] Names of endogenous variables for use in output
- name_q [list of strings] Names of instruments for use in output
- name_w [string] Name of weights matrix for use in output
- name_ds [string] Name of dataset for use in output

Examples

We first need to import the needed modules, namely numpy to convert the data we read into arrays that spreg understands and pysal to perform all the analysis.

```python
>>> import numpy as np
>>> import pysal.lib
```
Open data on Columbus neighborhood crime (49 areas) using `pysal.lib.io.open()`. This is the DBF associated with the Columbus shapefile. Note that `pysal.lib.io.open()` also reads data in CSV format; since the actual class requires data to be passed in as numpy arrays, the user can read their data in using any method.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path("columbus.dbf"), 'r')
```

Extract the CRIME column (crime rates) from the DBF file and make it the dependent variable for the regression. Note that PySAL requires this to be a numpy array of shape (n, 1) as opposed to the also common shape of (n, ) that other packages accept.

```python
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Extract INC (income) vector from the DBF to be used as independent variables in the regression. Note that PySAL requires this to be an nxj numpy array, where j is the number of independent variables (not including a constant). By default this model adds a vector of ones to the independent variables passed in.

```python
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X = np.array(X).T
```

Since we want to run a spatial error model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations into the error component of the model. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `columbus.shp`.

```python
>>> w = pysal.lib.weights.Rook.from_shapefile(pysal.lib.examples.get_path("columbus.shp"))
```

Unless there is a good reason not to do it, the weights have to be row-standardized so every row of the matrix sums to one. Among other things, this allows to interpret the spatial lag of a variable as the average value of the neighboring observations. In PySAL, this can be easily performed in the following way:

```python
>>> w.transform = 'r'
```

The Combo class runs an SARAR model, that is a spatial lag+error model. In this case we will run a simple version of that, where we have the spatial effects as well as exogenous variables. Since it is a spatial model, we have to pass in the weights matrix. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional.

```python
>>> reg = GM_Combo(y, X, w=w, name_y='crime', name_x=['income'], name_ds='columbus')
```

Once we have run the model, we can explore a little bit the output. The regression object we have created has many attributes so take your time to discover them. Note that because we are running the classical GMM error model from 1998/99, the spatial parameter is obtained as a point estimate, so although you get a value for it (there are for coefficients under model.betas), you cannot perform inference on it (there are only three values in model.se_betas). Also, this regression uses a two stage least squares estimation method that accounts for the endogeneity created by the spatial lag of the dependent variable. We can check the betas:

```python
>>> print reg.name_z
['CONSTANT', 'income', 'W_crime', 'lambda']
>>> print np.around(np.hstack((reg.betas[:-1], np.sqrt(reg.vm.diagonal()).reshape(3,1))),3)
[[ 39.059  11.86 ]
 [ -1.404  0.391]
 [  0.467  0.2 ]]
```
And lambda:

```python
>>> print 'lambda: ', np.around(reg.betas[-1], 3)
lambda: [-0.048]
```

This class also allows the user to run a spatial lag+error model with the extra feature of including non-spatial endogenous regressors. This means that, in addition to the spatial lag and error, we consider some of the variables on the right-hand side of the equation as endogenous and we instrument for this. As an example, we will include HOVAL (home value) as endogenous and will instrument with DISCBD (distance to the CSB). We first need to read in the variables:

```python
>>> yd = []
>>> yd.append(db.by_col("HOVAL"))
>>> yd = np.array(yd).T
>>> q = []
>>> q.append(db.by_col("DISCBD"))
>>> q = np.array(q).T
```

And then we can run and explore the model analogously to the previous combo:

```python
>>> reg = GM_Combo(y, X, yd, q, w=w, name_x=["inc"], name_y="crime", name_yend=[
˓→'hoval'], name_q=["discbd"], name_ds='columbus')
>>> print reg.name_z
['CONSTANT', 'inc', 'hoval', 'W_crime', 'lambda']
>>> names = np.array(reg.name_z).reshape(5,1)
>>> print np.hstack((names[0:4, :], np.around(np.hstack((reg.betas[:-1], np.sqrt(reg.vm.diagonal()).reshape(4,1))),4)))
[['CONSTANT' '50.0944' '14.3593']
['inc' '-0.2552' '0.5667']
['hoval' '-0.6885' '0.3029']
['W_crime' '0.4375' '0.2314']]

>>>
```

```python
>>> print 'lambda: ', np.around(reg.betas[-1], 3)
lambda: [ 0.254]
```

Attributes

- **summary** [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
- **betas** [array] kx1 array of estimated coefficients
- **u** [array] nx1 array of residuals
- **e_filtered** [array] nx1 array of spatially filtered residuals
- **e_pred** [array] nx1 array of residuals (using reduced form)
- **predy** [array] nx1 array of predicted y values
- **predy_e** [array] nx1 array of predicted y values (using reduced form)
- **n** [integer] Number of observations
- **k** [integer] Number of variables for which coefficients are estimated (including the constant)
- **y** [array] nx1 array for dependent variable
- **x** [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
- **yend** [array] Two dimensional array with n rows and one column for each endogenous variable
z [array] nxk array of variables (combination of x and yend)
mean_y [float] Mean of dependent variable
std_y [float] Standard deviation of dependent variable
vm [array] Variance covariance matrix (kxk)
pr2 [float] Pseudo R squared (squared correlation between y and ypred)
pr2_e [float] Pseudo R squared (squared correlation between y and ypred_e (using reduced form))
sig2 [float] Sigma squared used in computations (based on filtered residuals)
std_err [array] 1xk array of standard errors of the betas
z_stat [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float
name_y [string] Name of dependent variable for use in output
name_x [list of strings] Names of independent variables for use in output
name_yend [list of strings] Names of endogenous variables for use in output
name_z [list of strings] Names of exogenous and endogenous variables for use in output
name_q [list of strings] Names of external instruments
name_h [list of strings] Names of all instruments used in output
name_w [string] Name of weights matrix for use in output
title [string] Name of the regression method used

__init__(y, x, yend=None, q=None, w=None, w_lags=1, lag_q=True, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, x[, yend, q, w, w_lags, lag_q, ...]) Initialize self.

Attributes

mean_y

std_y

pysal.model.spreg.GM_Combo_Het

class pysal.model.spreg.GM_Combo_Het (y, x, yend=None, q=None, w=None, w_lags=1, lag_q=True, max_iter=1, epsilon=1e-05, step1c=False, inv_method='power_exp', vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None)
GMM method for a spatial lag and error model with hetroskedasticity and endogenous variables, with results
and diagnostics; based on Arraiz et al [Arraiz2010], following Anselin [Anselin2011].

**Parameters**

- `y` [array] nx1 array for dependent variable
- `x` [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
- `yend` [array] Two dimensional array with n rows and one column for each endogenous variable
- `q` [array] Two dimensional array with n rows and one column for each external exogenous variable to use as instruments (note: this should not contain any variables from x)
- `w` [pysal W object] Spatial weights object (always needed)
- `w_lags` [integer] Orders of W to include as instruments for the spatially lagged dependent variable. For example, w_lags=1, then instruments are WX; if w_lags=2, then WX, WWX; and so on.
- `lag_q` [boolean] If True, then include spatial lags of the additional instruments (q).
- `max_iter` [int] Maximum number of iterations of steps 2a and 2b from Arraiz et al. Note: epsilon provides an additional stop condition.
- `epsilon` [float] Minimum change in lambda required to stop iterations of steps 2a and 2b from Arraiz et al. Note: max_iter provides an additional stop condition.
- `step1c` [boolean] If True, then include Step 1c from Arraiz et al.
- `inv_method` [string] If “power_exp”, then compute inverse using the power expansion. If “true_inv”, then compute the true inverse. Note that true_inv will fail for large n.
- `vm` [boolean] If True, include variance-covariance matrix in summary results
- `name_y` [string] Name of dependent variable for use in output
- `name_x` [list of strings] Names of independent variables for use in output
- `name_yend` [list of strings] Names of endogenous variables for use in output
- `name_q` [list of strings] Names of instruments for use in output
- `name_w` [string] Name of weights matrix for use in output
- `name_ds` [string] Name of dataset for use in output

**Examples**

We first need to import the needed modules, namely numpy to convert the data we read into arrays that spreg understands and pysal to perform all the analysis.

```python
>>> import numpy as np
>>> import pysal.lib
```

Open data on Columbus neighborhood crime (49 areas) using pysal.lib.io.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.lib.io.open() also reads data in CSV format; since the actual class requires data to be passed in as numpy arrays, the user can read their data in using any method.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path('columbus.dbf'), 'r')
```
Extract the HOVL column (home values) from the DBF file and make it the dependent variable for the regression. Note that PySAL requires this to be a numpy array of shape (n, 1) as opposed to the also common shape of (n, ) that other packages accept.

```python
>>> y = np.array(db.by_col("HOVAL"))
>>> y = np.reshape(y, (49,1))
```

Extract INC (income) vector from the DBF to be used as independent variables in the regression. Note that PySAL requires this to be an nxj numpy array, where j is the number of independent variables (not including a constant). By default this class adds a vector of ones to the independent variables passed in.

```python
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X = np.array(X).T
```

Since we want to run a spatial error model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations into the error component of the model. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from columbus.shp.

```python
>>> w = pysal.lib.weights.Rook.from_shapefile(pysal.lib.examples.get_path("columbus.shp"))
```

Unless there is a good reason not to do it, the weights have to be row-standardized so every row of the matrix sums to one. Among other things, his allows to interpret the spatial lag of a variable as the average value of the neighboring observations. In PySAL, this can be easily performed in the following way:

```python
>>> w.transform = 'r'
```

The Combo class runs an SARAR model, that is a spatial lag+error model. In this case we will run a simple version of that, where we have the spatial effects as well as exogenous variables. Since it is a spatial model, we have to pass in the weights matrix. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional.

```python
>>> reg = GM_Combo_Het(y, X, w=w, step1c=True, name_y='hoval', name_x=['income'], name_ds='columbus')
```

Once we have run the model, we can explore a little bit the output. The regression object we have created has many attributes so take your time to discover them. This class offers an error model that explicitly accounts for heteroskedasticity and that unlike the models from spreg.error_sp, it allows for inference on the spatial parameter. Hence, we find the same number of betas as of standard errors, which we calculate taking the square root of the diagonal of the variance-covariance matrix:

```python
>>> print reg.name_z
['CONSTANT', 'income', 'W_hoval', 'lambda']
>>> print np.around(np.hstack((reg.betas,np.sqrt(reg.vm.diagonal()).reshape(4,1))),4)
[[ 9.9753  14.1435]
 [ 1.5742   0.374 ]
 [ 0.1535   0.3978]
 [ 0.2103   0.3924]]
```

This class also allows the user to run a spatial lag+error model with the extra feature of including non-spatial endogenous regressors. This means that, in addition to the spatial lag and error, we consider some of the variables on the right-hand side of the equation as endogenous and we instrument for this. As an example, we will include CRIME (crime rates) as endogenous and will instrument with DISCBD (distance to the CSB). We first need to read in the variables:
And then we can run and explore the model analogously to the previous combo:

```python
>>> reg = GM_Combo_Het(y, X, yd, q, w=w, step1c=True, name_x=['inc'], name_y='hoval', name_yend=['crime'], name_q=['discbd'], name_ds='columbus')
>>> print reg.name_z
['CONSTANT', 'inc', 'crime', 'W_hoval', 'lambda']
>>> print np.round(reg.betas, 4)
[[ 113.9129]
 [ -0.3482]
 [ -1.3566]
 [ -0.5766]
 [  0.6561]]
```

Attributes

- **summary** [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
- **betas** [array] kx1 array of estimated coefficients
- **u** [array] nx1 array of residuals
- **e_filtered** [array] nx1 array of spatially filtered residuals
- **e_pred** [array] nx1 array of residuals (using reduced form)
- **predy** [array] nx1 array of predicted y values
- **predy_e** [array] nx1 array of predicted y values (using reduced form)
- **n** [integer] Number of observations
- **k** [integer] Number of variables for which coefficients are estimated (including the constant)
- **y** [array] nx1 array for dependent variable
- **x** [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
- **yend** [array] Two dimensional array with n rows and one column for each endogenous variable
- **q** [array] Two dimensional array with n rows and one column for each external exogenous variable used as instruments
- **z** [array] nxk array of variables (combination of x and yend)
- **h** [array] nx1 array of instruments (combination of x and q)
- **iter_stop** [string] Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al.
- **iteration** [integer] Number of iterations of steps 2a and 2b from Arraiz et al.
- **mean_y** [float] Mean of dependent variable
- **std_y** [float] Standard deviation of dependent variable
- **vm** [array] Variance covariance matrix (kxk)
pr2 [float] Pseudo R squared (squared correlation between y and ypred)

pr2_e [float] Pseudo R squared (squared correlation between y and ypred_e (using reduced form))

std_err [array] 1xk array of standard errors of the betas

z_stat [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float

name_y [string] Name of dependent variable for use in output

name_x [list of strings] Names of independent variables for use in output

name_yend [list of strings] Names of endogenous variables for use in output

name_z [list of strings] Names of exogenous and endogenous variables for use in output

name_q [list of strings] Names of external instruments

name_h [list of strings] Names of all instruments used in output

name_w [string] Name of weights matrix for use in output

name_ds [string] Name of dataset for use in output

title [string] Name of the regression method used

hth [float] H’H

__init__(y, x, yend=None, q=None, w=None, w_lags=1, lag_q=True, max_iter=1, epsilon=1e-05, step1c=False, inv_method='power_exp', vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, x, yend, q, w, w_lags, lag_q, ...)) Initialize self.

Attributes

mean_y

std_y

pysal.model.spreg.GM_Combo_Hom

class pysal.model.spreg.GM_Combo_Hom(y, x, yend=None, q=None, w=None, w_lags=1, lag_q=True, max_iter=1, epsilon=1e-05, A1='hom_sc', vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None)

GMM method for a spatial lag and error model with homoskedasticity and endogenous variables, with results and diagnostics; based on Drukker et al. (2013) [Drukker2013], following Anselin (2011) [Anselin2011].

Parameters

y [array] nx1 array for dependent variable

x [array] Two dimensional array with n rows and one column for each independent (exogenous)
variable, excluding the constant

**yend**  [array] Two dimensional array with n rows and one column for each endogenous variable

**q**  [array] Two dimensional array with n rows and one column for each external exogenous variable to use as instruments (note: this should not contain any variables from x)

**w**  [pysal W object] Spatial weights object (always necessary)

**w_lags**  [integer] Orders of W to include as instruments for the spatially lagged dependent variable. For example, w_lags=1, then instruments are WX; if w_lags=2, then WX, WWX; and so on.

**lag_q**  [boolean] If True, then include spatial lags of the additional instruments (q).

**max_iter**  [int] Maximum number of iterations of steps 2a and 2b from Arraiz et al. Note: epsilon provides an additional stop condition.

**epsilon**  [float] Minimum change in lambda required to stop iterations of steps 2a and 2b from Arraiz et al. Note: max_iter provides an additional stop condition.

**A1**  [string] If A1='het', then the matrix A1 is defined as in Arraiz et al. If A1='hom', then as in Anselin (2011). If A1='hom_sc' (default), then as in Drukker, Egger and Prucha (2010) and Drukker, Prucha and Raciborski (2010).

**vm**  [boolean] If True, include variance-covariance matrix in summary results

**name_y**  [string] Name of dependent variable for use in output

**name_x**  [list of strings] Names of independent variables for use in output

**name_yend**  [list of strings] Names of endogenous variables for use in output

**name_q**  [list of strings] Names of instruments for use in output

**name_w**  [string] Name of weights matrix for use in output

**name_ds**  [string] Name of dataset for use in output

**Examples**

We first need to import the needed modules, namely numpy to convert the data we read into arrays that *spreg* understands and *pysal* to perform all the analysis.

```
>>> import numpy as np
>>> import pysal.lib

Open data on Columbus neighborhood crime (49 areas) using pysal.lib.io.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.lib.io.open() also reads data in CSV format; since the actual class requires data to be passed in as numpy arrays, the user can read their data in using any method.

```

```
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path('columbus.dbf'), 'r')
```

Extract the HOVAL column (home values) from the DBF file and make it the dependent variable for the regression. Note that PySAL requires this to be an numpyp array of shape (n, 1) as opposed to the also common shape of (n,) that other packages accept.

```
>>> y = np.array(db.by_col("HOVAL"))
>>> y = np.reshape(y, (49,1))
```
Extract INC (income) vector from the DBF to be used as independent variables in the regression. Note that PySAL requires this to be an nxj numpy array, where j is the number of independent variables (not including a constant). By default this class adds a vector of ones to the independent variables passed in.

```python
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X = np.array(X).T
```

Since we want to run a spatial error model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations into the error component of the model. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `columbus.shp`.

```python
>>> w = pysal.lib.weights.Rook.from_shapefile(pysal.lib.examples.get_path("columbus.shp"))
```

Unless there is a good reason not to do it, the weights have to be row-standardized so every row of the matrix sums to one. Among other things, this allows to interpret the spatial lag of a variable as the average value of the neighboring observations. In PySAL, this can be easily performed in the following way:

```python
>>> w.transform = 'r'
```

Example only with spatial lag

The Combo class runs an SARAR model, that is a spatial lag+error model. In this case we will run a simple version of that, where we have the spatial effects as well as exogenous variables. Since it is a spatial model, we have to pass in the weights matrix. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional.

```python
>>> reg = GM_Combo_Hom(y, X, w=w, A1='hom_sc', name_x=['inc'], name_y='hoval', name_yend=['crime'], name_q=['discbd'], name_ds='columbus')
>>> print np.around(np.hstack((reg.betas,np.sqrt(reg.vm.diagonal()).reshape(4,1))),4)
[[ 10.1254  15.2871]
 [  1.5683   0.4407]
 [  0.1513   0.4048]
 [  0.2103   0.4226]]
```

This class also allows the user to run a spatial lag+error model with the extra feature of including non-spatial endogenous regressors. This means that, in addition to the spatial lag and error, we consider some of the variables on the right-hand side of the equation as endogenous and we instrument for this. As an example, we will include CRIME (crime rates) as endogenous and will instrument with DISCBD (distance to the CSB). We first need to read in the variables:

```python
>>> yd = []
>>> yd.append(db.by_col("CRIME"))
>>> yd = np.array(yd).T
>>> q = []
>>> q.append(db.by_col("DISCBD"))
>>> q = np.array(q).T
```

And then we can run and explore the model analogously to the previous combo:

```python
>>> reg = GM_Combo_Hom(y, X, yd, q, w=w, A1='hom_sc', name_ds='columbus')
>>> betas = np.array([[['CONSTANT'],['inc'],['crime'],['W_hoval'],['lambda']]])
>>> print np.hstack((betas, np.around(np.hstack((reg.betas, np.sqrt(reg.vm.diagonal()).reshape(5,1))),5)))
(continues on next page)
```
Attributes

summary  [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
betas  [array] kx1 array of estimated coefficients
u  [array] nx1 array of residuals
e_filtered  [array] nx1 array of spatially filtered residuals
e_pred  [array] nx1 array of residuals (using reduced form)
predy  [array] nx1 array of predicted y values
predy_e  [array] nx1 array of predicted y values (using reduced form)
n  [integer] Number of observations
k  [integer] Number of variables for which coefficients are estimated (including the constant)
y  [array] nx1 array for dependent variable
x  [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
yend  [array] Two dimensional array with n rows and one column for each endogenous variable
q  [array] Two dimensional array with n rows and one column for each external exogenous variable used as instruments
z  [array] nxk array of variables (combination of x and yend)
h  [array] nx1 array of instruments (combination of x and q)
iter_stop  [string] Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al.
iteration  [integer] Number of iterations of steps 2a and 2b from Arraiz et al.
mean_y  [float] Mean of dependent variable
std_y  [float] Standard deviation of dependent variable
vm  [array] Variance covariance matrix (kxk)
pr2  [float] Pseudo R squared (squared correlation between y and ypred)
pr2_e  [float] Pseudo R squared (squared correlation between y and ypred_e (using reduced form))
sig2  [float] Sigma squared used in computations (based on filtered residuals)
std_err  [array] 1xk array of standard errors of the betas
z_stat  [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float
name_y  [string] Name of dependent variable for use in output
name_x  [list of strings] Names of independent variables for use in output
name_yend [list of strings] Names of endogenous variables for use in output
name_z [list of strings] Names of exogenous and endogenous variables for use in output
name_q [list of strings] Names of external instruments
name_h [list of strings] Names of all instruments used in output
name_w [string] Name of weights matrix for use in output
name_ds [string] Name of dataset for use in output
title [string] Name of the regression method used
hth [float] H’H

__init__(y, x, yend=None, q=None, w=None, w_lags=1, lag_q=True, max_iter=1, epsilon=1e-05, A1='hom_sc', vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, x, yend, q, w, w_lags, lag_q, ...)) Initialize self.

Attributes

mean_y
std_y

pysal.model.spreg.GM_Endog_Error
class pysal.model.spreg.GM_Endog_Error(y, x, yend, q, w, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None)
GMM method for a spatial error model with endogenous variables, with results and diagnostics; based on Kelejian and Prucha (1998, 1999) [Kelejian1998] [Kelejian1999].

Parameters

y [array] nx1 array for dependent variable
x [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
yend [array] Two dimensional array with n rows and one column for each endogenous variable
q [array] Two dimensional array with n rows and one column for each external exogenous variable to use as instruments (note: this should not contain any variables from x)
w [pysal W object] Spatial weights object (always needed)
vm [boolean] If True, include variance-covariance matrix in summary results
name_y [string] Name of dependent variable for use in output
name_x [list of strings] Names of independent variables for use in output
name_yend [list of strings] Names of endogenous variables for use in output
name_q  [list of strings] Names of instruments for use in output
name_w  [string] Name of weights matrix for use in output
name_ds  [string] Name of dataset for use in output

Examples

We first need to import the needed modules, namely numpy to convert the data we read into arrays that spreg understands and pysal to perform all the analysis.

```python
>>> import pysal.lib
>>> import numpy as np
```

Open data on Columbus neighborhood crime (49 areas) using pysal.lib.io.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.lib.io.open() also reads data in CSV format; since the actual class requires data to be passed in as numpy arrays, the user can read their data in using any method.

```python
>>> dbf = pysal.lib.io.open(pysal.lib.examples.get_path("columbus.dbf"),'r')
```

Extract the CRIME column (crime rates) from the DBF file and make it the dependent variable for the regression. Note that PySAL requires this to be an n x 1 numpy array as opposed to the also common shape of (n, ) that other packages accept.

```python
>>> y = np.array([dbf.by_col('CRIME')]).T
```

Extract INC (income) vector from the DBF to be used as independent variables in the regression. Note that PySAL requires this to be an nxj numpy array, where j is the number of independent variables (not including a constant). By default this model adds a vector of ones to the independent variables passed in.

```python
>>> x = np.array([dbf.by_col('INC')]).T
```

In this case we consider HOVAL (home value) is an endogenous regressor. We tell the model that this is so by passing it in a different parameter from the exogenous variables (x).

```python
>>> yend = np.array([dbf.by_col('HOVAL')]).T
```

Because we have endogenous variables, to obtain a correct estimate of the model, we need to instrument for HOVAL. We use DISCBD (distance to the CBD) for this and hence put it in the instruments parameter, ‘q’.

```python
>>> q = np.array([dbf.by_col('DISCBD')]).T
```

Since we want to run a spatial error model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations into the error component of the model. To do that, we can open an already existing gal file or create a new one. In this case, we will use columbus.gal, which contains contiguity relationships between the observations in the Columbus dataset we are using throughout this example. Note that, in order to read the file, not only to open it, we need to append `.read()` at the end of the command.

```python
>>> w = pysal.lib.io.open(pysal.lib.examples.get_path("columbus.gal"), 'r').read()
```

Unless there is a good reason not to do it, the weights have to be row-standardized so every row of the matrix sums to one. Among other things, this allows to interpret the spatial lag of a variable as the average value of the neighboring observations. In PySAL, this can be easily performed in the following way:

```python
>>> w.transform='r'
```
We are all set with the preliminars, we are good to run the model. In this case, we will need the variables (exogenous and endogenous), the instruments and the weights matrix. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional.

```python
>>> model = GM_Endog_Error(y, x, yend, q, w=w, name_x=['inc'], name_y='crime',
                           name_yend=['hoval'], name_q=['discbd'], name_ds='columbus')
```

Once we have run the model, we can explore a little bit the output. The regression object we have created has many attributes so take your time to discover them. Note that because we are running the classical GMM error model from 1998/99, the spatial parameter is obtained as a point estimate, so although you get a value for it (there are for coefficients under model.betas), you cannot perform inference on it (there are only three values in model.se_betas). Also, this regression uses a two stage least squares estimation method that accounts for the endogeneity created by the endogenous variables included.

```python
>>> print model.name_z
['CONSTANT', 'inc', 'hoval', 'lambda']
>>> np.around(model.betas, decimals=4)
array([[ 82.573 ],
       [ 0.581 ],
       [ -1.4481],
       [ 0.3499]])
>>> np.around(model.std_err, decimals=4)
array([ 16.1381,  1.3545,  0.7862])
```

Attributes

- **summary** [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
- **betas** [array] kx1 array of estimated coefficients
- **u** [array] nx1 array of residuals
- **e_filtered** [array] nx1 array of spatially filtered residuals
- **predy** [array] nx1 array of predicted y values
- **n** [integer] Number of observations
- **k** [integer] Number of variables for which coefficients are estimated (including the constant)
- **y** [array] nx1 array for dependent variable
- **x** [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
- **yend** [array] Two dimensional array with n rows and one column for each endogenous variable
- **z** [array] nxk array of variables (combination of x and yend)
- **mean_y** [float] Mean of dependent variable
- **std_y** [float] Standard deviation of dependent variable
- **vm** [array] Variance covariance matrix (kxk)
- **pr2** [float] Pseudo R squared (squared correlation between y and ypred)
- **sig2** [float] Sigma squared used in computations
- **std_err** [array] 1xk array of standard errors of the betas
- **z_stat** [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float
name_y [string] Name of dependent variable for use in output
name_x [list of strings] Names of independent variables for use in output
name_yend [list of strings] Names of endogenous variables for use in output
name_z [list of strings] Names of exogenous and endogenous variables for use in output
name_q [list of strings] Names of external instruments
name_h [list of strings] Names of all instruments used in output
name_w [string] Name of weights matrix for use in output
name_ds [string] Name of dataset for use in output
title [string] Name of the regression method used

__init__(y, x, yend, q, w[, vm, name_y, ...]) Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, x, yend, q, w[, vm, name_y, ...]) Initialize self.

Attributes

mean_y
std_y

pysal.model.spreg.GM_Endog_Error_Het
class pysal.model.spreg.GM_Endog_Error_Het (y, x, yend, q, w, max_iter=1, epsilon=1e-05, step1c=False, inv_method='power_exp', vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None)
GMM method for a spatial error model with heteroskedasticity and endogenous variables, with results and diagnostics; based on Arraiz et al [Arraiz2010], following Anselin [Anselin2011].

Parameters

y [array] nx1 array for dependent variable
x [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
yend [array] Two dimensional array with n rows and one column for each endogenous variable
q [array] Two dimensional array with n rows and one column for each external exogenous variable to use as instruments (note: this should not contain any variables from x)
w [pysal W object] Spatial weights object
max_iter [int] Maximum number of iterations of steps 2a and 2b from Arraiz et al. Note: epsilon provides an additional stop condition.
epsilon  [float] Minimum change in lambda required to stop iterations of steps 2a and 2b from
Arraiz et al. Note: max_iter provides an additional stop condition.
step1c  [boolean] If True, then include Step 1c from Arraiz et al.
inv_method  [string] If “power_exp”, then compute inverse using the power expansion. If
“true_inv”, then compute the true inverse. Note that true_inv will fail for large n.
vm  [boolean] If True, include variance-covariance matrix in summary results
name_y  [string] Name of dependent variable for use in output
name_x  [list of strings] Names of independent variables for use in output
name_yend  [list of strings] Names of endogenous variables for use in output
name_q  [list of strings] Names of instruments for use in output
name_w  [string] Name of weights matrix for use in output
name_ds  [string] Name of dataset for use in output

Examples

We first need to import the needed modules, namely numpy to convert the data we read into arrays that spreg
understands and pysal to perform all the analysis.

```python
>>> import numpy as np
>>> import pysal.lib
```

Open data on Columbus neighborhood crime (49 areas) using pysal.lib.io.open(). This is the DBF associated
with the Columbus shapefile. Note that pysal.lib.io.open() also reads data in CSV format; since the actual class
requires data to be passed in as numpy arrays, the user can read their data in using any method.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path('columbus.dbf'),'r')
```

Extract the HOVAL column (home values) from the DBF file and make it the dependent variable for the regres-
sion. Note that PySAL requires this to be an n x 1 numpy array as opposed to the also common shape
of (n, ) that other packages accept.

```python
>>> y = np.array(db.by_col("HOVAL"))
>>> y = np.reshape(y, (49,1))
```

Extract INC (income) vector from the DBF to be used as independent variables in the regression. Note that
PySAL requires this to be an nxj numpy array, where j is the number of independent variables (not including a
constant). By default this class adds a vector of ones to the independent variables passed in.

```python
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X = np.array(X).T
```

In this case we consider CRIME (crime rates) is an endogenous regressor. We tell the model that this is so by
passing it in a different parameter from the exogenous variables (x).

```python
>>> yd = []
>>> yd.append(db.by_col("CRIME"))
>>> yd = np.array(yd).T
```

Because we have endogenous variables, to obtain a correct estimate of the model, we need to instrument for
CRIME. We use DISCBD (distance to the CBD) for this and hence put it in the instruments parameter, ‘q’.
```python
>>> q = []
>>> q.append(db.by_col("DISCBD"))
>>> q = np.array(q).T
```

Since we want to run a spatial error model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations into the error component of the model. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `columbus.shp`.

```python
>>> w = pysal.lib.weights.Rook.from_shapefile(pysal.lib.examples.get_path("columbus.shp"))
```

Unless there is a good reason not to do it, the weights have to be row-standardized so every row of the matrix sums to one. Among other things, his allows to interpret the spatial lag of a variable as the average value of the neighboring observations. In PySAL, this can be easily performed in the following way:

```python
>>> w.transform = 'r'
```

We are all set with the preliminaries, we are good to run the model. In this case, we will need the variables (exogenous and endogenous), the instruments and the weights matrix. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional.

```python
>>> reg = GM_Endog_Error_Het(y, X, yd, q, w=w, step1c=True, name_x=['inc'], name_y="hoval", name_yend=['crime'], name_q=['discbd'], name_ds='columbus')
```

Once we have run the model, we can explore a little bit the output. The regression object we have created has many attributes so take your time to discover them. This class offers an error model that explicitly accounts for heteroskedasticity and that unlike the models from `spreg.error_sp`, it allows for inference on the spatial parameter. Hence, we find the same number of betas as of standard errors, which we calculate taking the square root of the diagonal of the variance-covariance matrix:

```python
>>> print reg.name_z
['CONSTANT', 'inc', 'crime', 'lambda']
>>> print np.around(np.hstack((reg.betas, np.sqrt(reg.vm.diagonal()).reshape(4, 1))), 4)
[[ 55.3971  28.8901]
 [ 0.4656  0.7731]
 [-0.6704  0.468 ]
 [ 0.4114  0.1777]]
```

**Attributes**

- `summary` [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
- `betas` [array] kx1 array of estimated coefficients
- `u` [array] nx1 array of residuals
- `e_filtered` [array] nx1 array of spatially filtered residuals
- `predy` [array] nx1 array of predicted y values
- `n` [integer] Number of observations
- `k` [integer] Number of variables for which coefficients are estimated (including the constant)
- `y` [array] nx1 array for dependent variable
- `x` [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
yend [array] Two dimensional array with n rows and one column for each endogenous variable
q [array] Two dimensional array with n rows and one column for each external exogenous variable used as instruments
z [array] nxk array of variables (combination of x and yend)
h [array] nxl array of instruments (combination of x and q)
iter_stop [string] Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al.
iteration [integer] Number of iterations of steps 2a and 2b from Arraiz et al.
mean_y [float] Mean of dependent variable
std_y [float] Standard deviation of dependent variable
vm [array] Variance covariance matrix (kxk)
pr2 [float] Pseudo R squared (squared correlation between y and ypred)
std_err [array] 1xk array of standard errors of the betas
z_stat [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float
name_y [string] Name of dependent variable for use in output
name_x [list of strings] Names of independent variables for use in output
name_yend [list of strings] Names of endogenous variables for use in output
name_z [list of strings] Names of exogenous and endogenous variables for use in output
name_q [list of strings] Names of external instruments
name_h [list of strings] Names of all instruments used in ouput
name_w [string] Name of weights matrix for use in output
name_ds [string] Name of dataset for use in output
title [string] Name of the regression method used
hth [float] H’H

__init__(y, x, yend, q, w[, max_iter, ...]) Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(y, x, yend, q, w[, max_iter, ...]) Initialize self.

Attributes

mean_y
std_y
pysal.model.spreg.GM_Endog_Error_Hom

**class** pysal.model.spreg.GM_Endog_Error_Hom(y, x, yend, w, max_iter=1, epsilon=1e-05, A1='hom_sc', vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None)

GMM method for a spatial error model with homoskedasticity and endogenous variables, with results and diagnostics; based on Drukker et al. (2013) [Drukker2013], following Anselin (2011) [Anselin2011].

**Parameters**

- **y** [array] nx1 array for dependent variable
- **x** [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
- **yend** [array] Two dimensional array with n rows and one column for each endogenous variable
- **q** [array] Two dimensional array with n rows and one column for each external exogenous variable to use as instruments (note: this should not contain any variables from x)
- **w** [pysal W object] Spatial weights object
- **max_iter** [int] Maximum number of iterations of steps 2a and 2b from Arraiz et al. Note: epsilon provides an additional stop condition.
- **epsilon** [float] Minimum change in lambda required to stop iterations of steps 2a and 2b from Arraiz et al. Note: max_iter provides an additional stop condition.
- **A1** [string] If A1='het', then the matrix A1 is defined as in Arraiz et al. If A1='hom', then as in Anselin (2011). If A1='hom_sc' (default), then as in Drukker, Egger and Prucha (2010) and Drukker, Prucha and Raciborski (2010).
- **vm** [boolean] If True, include variance-covariance matrix in summary results
- **name_y** [string] Name of dependent variable for use in output
- **name_x** [list of strings] Names of independent variables for use in output
- **name_yend** [list of strings] Names of endogenous variables for use in output
- **name_q** [list of strings] Names of instruments for use in output
- **name_w** [string] Name of weights matrix for use in output
- **name_ds** [string] Name of dataset for use in output

**Examples**

We first need to import the needed modules, namely numpy to convert the data we read into arrays that spreg understands and pysal to perform all the analysis.

```python
>>> import numpy as np
>>> import pysal.lib
```

Open data on Columbus neighborhood crime (49 areas) using pysal.lib.io.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.lib.io.open() also reads data in CSV format; since the actual class requires data to be passed in as numpy arrays, the user can read their data in using any method.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path('columbus.dbf'), 'r')
```
Extract the HOV AL column (home values) from the DBF file and make it the dependent variable for the regression. Note that PySAL requires this to be an numpy array of shape (n, 1) as opposed to the also common shape of (n, ) that other packages accept.

```python
gm = np.array(db.by_col("HOVAL"))
gm = np.reshape(gm, (49, 1))
```

Extract INC (income) vector from the DBF to be used as independent variables in the regression. Note that PySAL requires this to be an nxj numpy array, where j is the number of independent variables (not including a constant). By default this class adds a vector of ones to the independent variables passed in.

```python
gm = []
gm.append(db.by_col("INC"))
gm = np.array(gm).T
```

In this case we consider CRIME (crime rates) is an endogenous regressor. We tell the model that this is so by passing it in a different parameter from the exogenous variables (x).

```python
gm = []
gm.append(db.by_col("CRIME"))
gm = np.array(gm).T
```

Because we have endogenous variables, to obtain a correct estimate of the model, we need to instrument for CRIME. We use DISCBD (distance to the CBD) for this and hence put it in the instruments parameter, ‘q’.

```python
gm = []
gm.append(db.by_col("DISCBD"))
gm = np.array(gm).T
```

Since we want to run a spatial error model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations into the error component of the model. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `columbus.shp`.

```python
gm = pysal.lib.weights.rook.from_shapefile(pysal.lib.examples.get_path("columbus.shp"))
```

Unless there is a good reason not to do it, the weights have to be row-standardized so every row of the matrix sums to one. Among other things, this allows to interpret the spatial lag of a variable as the average value of the neighboring observations. In PySAL, this can be easily performed in the following way:

```python
gm.transform = 'r'
```

We are all set with the preliminars, we are good to run the model. In this case, we will need the variables (exogenous and endogenous), the instruments and the weights matrix. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional.

```python
gm = GM_Endog_Error_Hom(y, X, yd, q, w=w, A1='hom_sc', name_x=['inc'], name_y=['hoval'], name_yend=['crime'], name_q=['discbd'], name_ds='columbus')
```

Once we have run the model, we can explore a little bit the output. The regression object we have created has many attributes so take your time to discover them. This class offers an error model that assumes homoskedasticity but that unlike the models from `spreg.error_sp`, it allows for inference on the spatial parameter. Hence, we find the same number of betas as of standard errors, which we calculate taking the square root of the diagonal of the variance-covariance matrix:
```python
>>> print reg.name_z
['CONSTANT', 'inc', 'crime', 'lambda']
>>> print np.around(np.hstack((reg.betas, np.sqrt(reg.vm.diagonal()).reshape(4, -1))), 4)
[[ 55.3658  23.4960 ]
 [  0.4643   0.7382 ]
[-0.6690   0.3943 ]
 [  0.4321   0.1927 ]]
```

**Attributes**

- `summary` [string] Summary of regression results and diagnostics (note: use in conjunction with the `print` command)
- `betas` [array] kx1 array of estimated coefficients
- `u` [array] nx1 array of residuals
- `e_filtered` [array] nx1 array of spatially filtered residuals
- `predy` [array] nx1 array of predicted y values
- `n` [integer] Number of observations
- `k` [integer] Number of variables for which coefficients are estimated (including the constant)
- `y` [array] nx1 array for dependent variable
- `x` [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
- `yend` [array] Two dimensional array with n rows and one column for each endogenous variable
- `q` [array] Two dimensional array with n rows and one column for each external exogenous variable used as instruments
- `z` [array] nxk array of variables (combination of x and yend)
- `h` [array] nx1 array of instruments (combination of x and q)
- `iter_stop` [string] Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al.
- `iteration` [integer] Number of iterations of steps 2a and 2b from Arraiz et al.
- `mean_y` [float] Mean of dependent variable
- `std_y` [float] Standard deviation of dependent variable
- `vm` [array] Variance covariance matrix (kxk)
- `pr2` [float] Pseudo R squared (squared correlation between y and ypred)
- `sig2` [float] Sigma squared used in computations
- `std_err` [array] 1xk array of standard errors of the betas
- `z_stat` [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float
- `name_y` [string] Name of dependent variable for use in output
- `name_x` [list of strings] Names of independent variables for use in output
- `name_yend` [list of strings] Names of endogenous variables for use in output
- `name_z` [list of strings] Names of exogenous and endogenous variables for use in output
name_q  [list of strings] Names of external instruments
name_h  [list of strings] Names of all instruments used in output
name_w  [string] Name of weights matrix for use in output
name_ds  [string] Name of dataset for use in output
title  [string] Name of the regression method used
hth  [float] H’H

__init__(y, x, yend, q, w[, max_iter, . . .]) Initialize self.

Methods

__init__(y, x, yend, q, w[, max_iter, . . .]) Initialize self.

Attributes

mean_y
std_y

pysal.model.spreg.TSLS

class pysal.model.spreg.TSLS(y, x, yend, q, w=None, robust=None, gwk=None, sig2n_k=False, spat_diag=False, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_gwk=None, name_ds=None)

Two stage least squares with results and diagnostics.

Parameters

y  [array] nx1 array for dependent variable
x  [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
yend  [array] Two dimensional array with n rows and one column for each endogenous variable
q  [array] Two dimensional array with n rows and one column for each external exogenous variable to use as instruments (note: this should not contain any variables from x)
w  [pysal W object] Spatial weights object (required if running spatial diagnostics)
robust  [string] If ‘white’, then a White consistent estimator of the variance-covariance matrix is given. If ‘hac’, then a HAC consistent estimator of the variance-covariance matrix is given. Default set to None.
gwk  [pysal W object] Kernel spatial weights needed for HAC estimation. Note: matrix must have ones along the main diagonal.
sig2n_k  [boolean] If True, then use n-k to estimate sigma^2. If False, use n.
spat_diag  [boolean] If True, then compute Anselin-Kelejian test (requires w)
vm  [boolean] If True, include variance-covariance matrix in summary results
name_y  [string] Name of dependent variable for use in output
name_x  [list of strings] Names of independent variables for use in output
name_yend  [list of strings] Names of endogenous variables for use in output
name_q  [list of strings] Names of instruments for use in output
name_w  [string] Name of weights matrix for use in output
name_gwk  [string] Name of kernel weights matrix for use in output
name_ds  [string] Name of dataset for use in output

Examples

We first need to import the needed modules, namely numpy to convert the data we read into arrays that `spreg` understands and `pysal` to perform all the analysis.

```python
>>> import numpy as np
>>> import pysal.lib
```

Open data on Columbus neighborhood crime (49 areas) using `pysal.lib.io.open()`. This is the DBF associated with the Columbus shapefile. Note that `pysal.lib.io.open()` also reads data in CSV format; since the actual class requires data to be passed in as numpy arrays, the user can read their data in using any method.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path("columbus.dbf"),'r')
```

Extract the CRIME column (crime rates) from the DBF file and make it the dependent variable for the regression. Note that PySAL requires this to be an numpy array of shape (n, 1) as opposed to the also common shape of (n, ) that other packages accept.

```python
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Extract INC (income) vector from the DBF to be used as independent variables in the regression. Note that PySAL requires this to be an nxj numpy array, where j is the number of independent variables (not including a constant). By default this model adds a vector of ones to the independent variables passed in, but this can be overridden by passing constant=False.

```python
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X = np.array(X).T
```

In this case we consider HOVAL (home value) is an endogenous regressor. We tell the model that this is so by passing it in a different parameter from the exogenous variables (x).

```python
>>> yd = []
>>> yd.append(db.by_col("HOVAL"))
>>> yd = np.array(yd).T
```

Because we have endogenous variables, to obtain a correct estimate of the model, we need to instrument for HOVAL. We use DISCBD (distance to the CBD) for this and hence put it in the instruments parameter, ‘q’.

```python
>>> q = []
>>> q.append(db.by_col("DISCBD"))
>>> q = np.array(q).T
```
We are all set with the preliminaries, we are good to run the model. In this case, we will need the variables (exogenous and endogenous) and the instruments. If we want to have the names of the variables printed in the output summary, we will have to pass them in as well, although this is optional.

```python
>>> reg = TSLS(y, X, yd, q, name_x=['inc'], name_y='crime', name_yend=['hoval'],
            name_q=['discbd'], name_ds='columbus')
>>> print reg.betas
[[ 88.46579584]
 [ 0.5200379 ]
[-1.58216593]]
```

Attributes

- `summary` [string] Summary of regression results and diagnostics (note: use in conjunction with the print command)
- `betas` [array] kx1 array of estimated coefficients
- `u` [array] nx1 array of residuals
- `predy` [array] nx1 array of predicted y values
- `n` [integer] Number of observations
- `k` [integer] Number of variables for which coefficients are estimated (including the constant)
- `kstar` [integer] Number of endogenous variables.
- `y` [array] nx1 array for dependent variable
- `x` [array] Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant
- `yend` [array] Two dimensional array with n rows and one column for each endogenous variable
- `q` [array] Two dimensional array with n rows and one column for each external exogenous variable used as instruments
- `z` [array] nxk array of variables (combination of x and yend)
- `h` [array] nxl array of instruments (combination of x and q)
- `robust` [string] Adjustment for robust standard errors
- `mean_y` [float] Mean of dependent variable
- `std_y` [float] Standard deviation of dependent variable
- `vm` [array] Variance covariance matrix (kxk)
- `pr2` [float] Pseudo R squared (squared correlation between y and ypred)
- `utu` [float] Sum of squared residuals
- `sig2` [float] Sigma squared used in computations
- `std_err` [array] 1xk array of standard errors of the betas
- `z_stat` [list of tuples] z statistic; each tuple contains the pair (statistic, p-value), where each is a float
- `ak_test` [tuple] Anselin-Kelejian test; tuple contains the pair (statistic, p-value)
- `name_y` [string] Name of dependent variable for use in output
- `name_x` [list of strings] Names of independent variables for use in output
**name_yend** [list of strings] Names of endogenous variables for use in output

**name_z** [list of strings] Names of exogenous and endogenous variables for use in output

**name_q** [list of strings] Names of external instruments

**name_h** [list of strings] Names of all instruments used in output

**name_w** [string] Name of weights matrix for use in output

**name_gwk** [string] Name of kernel weights matrix for use in output

**name_ds** [string] Name of dataset for use in output

**title** [string] Name of the regression method used

**sig2n** [float] Sigma squared (computed with n in the denominator)

**sig2n_k** [float] Sigma squared (computed with n-k in the denominator)

**hth** [float] \(H^TH\)

**hthi** [float] \((H^TH)^{-1}\)

**varb** [array] \((Z^TH (H^TH)^{-1} H^TH)^{-1}\)

**zthhthi** [array] \(Z^TH(H^TH)^{-1}\)

**pfora1a2** [array] \(n(zthhthi)^{T}varb\)

```python
__init__(y, x, yend, q[, w, robust, gwk, ...])
```
Initialize self. See help(type(self)) for accurate signature.

### Methods

- `__init__(y, x, yend, q[, w, robust, gwk, ...])` Initialize self.

### Attributes

- `mean_y`
- `pfora1a2`
- `sig2n`
- `sig2n_k`
- `std_y`
- `utu`
- `vm`

```python
class pysal.model.spreg.ThreeSLS
```
User class for 3SLS estimation

### Parameters
**Examples**

First import pysal.lib to load the spatial analysis tools.

```python
>>> import pysal.lib
```

Open data on NCOVR US County Homicides (3085 areas) using pysal.lib.io.open(). This is the DBF associated with the NAT shapefile. Note that pysal.lib.io.open() also reads data in CSV format.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path("NAT.dbf"),'r')
```

The specification of the model to be estimated can be provided as lists. Each equation should be listed separately. In this example, equation 1 has HR80 as dependent variable, PS80 and UE80 as exogenous regressors, RD80 as endogenous regressor and FP79 as additional instrument. For equation 2, HR90 is the dependent variable, PS90 and UE90 the exogenous regressors, RD90 as endogenous regressor and FP99 as additional instrument.

```python
>>> y_var = ['HR80','HR90']
>>> x_var = [['PS80','UE80'],['PS90','UE90']]
>>> yend_var = [['RD80'],['RD90']]
>>> q_var = [['FP79'],['FP89']]
```

The SUR method requires data to be provided as dictionaries. PySAL provides two tools to create these dictionaries from the list of variables: sur_dictxy and sur_dictZ. The tool sur_dictxy can be used to create the dictionaries for Y and X, and sur_dictZ for endogenous variables (yend) and additional instruments (q).

```python
>>> bigy,bigX,bigYvars,bigXvars = pysal.model.spreg.sur_utils.sur_dictxy(db,y_var,
          x_var)
>>> bigyend,bigyendvars = pysal.model.spreg.sur_utils.sur_dictZ(db,yend_var)
>>> bigq,bigqvars = pysal.model.spreg.sur_utils.sur_dictZ(db,q_var)
```
We can now run the regression and then have a summary of the output by typing: `print(reg.summary)`

Alternatively, we can just check the betas and standard errors, asymptotic t and p-value of the parameters:

```python
>>> reg = ThreeSLS(bigy, bigX, bigyend, bigq, name_bigy=bigyvars, name_bigX=bigXvars,
                   name_bigyend=bigyendvars, name_bigq=bigqvars, name_ds="NAT")
>>> reg.b3SLS
{0: array([[ 6.92426353],
           [ 1.42921826],
           [ 0.00049435],
           [ 3.5829275 ]]), 1: array([[ 7.62385875],
           [ 1.65031181],
           [-0.21682974],
           [ 3.91250428]])}
```

```python
>>> reg.tsls_inf
{0: array([[ 0.23220853, 29.81916157, 0.          ],
           [ 0.10373417, 13.77770036, 0.          ],
           [ 0.03086193, 17.19606554, 0.          ],
           [ 0.11131999, 32.18584124, 0.          ]]), 1: array([[ 0.28739415, 26.52735638, 0.          ],
           [ 0.09597031, 17.19606554, 0.          ],
           [ 0.04089547, -5.30204786, 0.00000011],
           [ 0.13586789, 28.79638723, 0.          ]])}
```

Attributes

- `bigy` [dictionary with y values]
- `bigZ` [dictionary with matrix of exogenous and endogenous variables] for each equation
- `bigZHZH` [dictionary with matrix of cross products $Z_{hat_r}'Z_{hat_s}$]
- `bigZHy` [dictionary with matrix of cross products $Z_{hat_r}'y_{end_s}$]
- `n_eq` [number of equations]
- `n` [number of observations in each cross-section]
- `bigK` [vector with number of explanatory variables (including constant,) exogenous and endogenous] for each equation
- `b2SLS` [dictionary with 2SLS regression coefficients for each equation]
- `tslsE` [N x n_eq array with OLS residuals for each equation]
- `b3SLS` [dictionary with 3SLS regression coefficients for each equation]
- `varb` [variance-covariance matrix]
- `sig` [Sigma matrix of inter-equation error covariances]
- `bigE` [n by n_eq array of residuals]
- `corr` [inter-equation 3SLS error correlation matrix]
- `tsls_inf` [dictionary with standard error, asymptotic t and p-value,] one for each equation
- `surchow` [list with tuples for Chow test on regression coefficients] each tuple contains test value, degrees of freedom, p-value
- `name_ds` [string; name for the data set]
- `name_bigy` [dictionary with name of dependent variable for each equation]
name__bigX [dictionary with names of explanatory variables for each equation]

name__bigyend [dictionary with names of endogenous variables for each equation]

name__bigq [dictionary with names of instrumental variables for each equation]

name__regimes [string; name of regime variable for use in the output]

__init__(bigy, bigX, bigyend, bigq, regimes=None, nonspat_diag=True, name__bigy=None, name__bigX=None, name__bigyend=None, name__bigq=None, name_ds=None, name__regimes=None)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(bigy, bigX, bigyend, bigq[,...]) Initialize self.

Regimes Models

Regimes models are variants of spatial regression models which allow for structural instability in parameters. That means that these models allow different coefficient values in distinct subsets of the data.

spreg.OLS_Regimes(y, x, regimes[, w,...]) Ordinary least squares with results and diagnostics.

spreg.ML_Lag_Regimes(y, x, regimes[, w,...]) ML estimation of the spatial lag model with regimes (note no consistency checks, diagnostics or constants added; Anselin (1988) [Anselin1988]

spreg.ML_Error_Regimes(y, x, regimes[, w,...]) ML estimation of the spatial error model with regimes (note no consistency checks, diagnostics or constants added; Anselin (1988) [Anselin1988]

spreg.GM_Lag_Regimes(y, x, regimes[, yend,...]) Spatial two stage least squares (S2SLS) with regimes; Anselin (1988) [Anselin1988]

spreg.GM_Error_Regimes(y, x, regimes, w[, ...]) GMM method for a spatial error model with regimes, with results and diagnostics; based on Kelejian and Prucha (1998, 1999) [Kelejian1998] [Kelejian1999].

spreg.GM_Error_Het_Regimes(y, x, regimes, w) GMM method for a spatial error model with heteroskedasticity and regimes; based on Arraiz et al [Arraiz2010], following Anselin [Anselin2011].

spreg.GM_Error_Hom_Regimes(y, x, regimes, w) GMM method for a spatial error model with homoskedasticity, with regimes, results and diagnostics; based on Drukker et al.

spreg.GM_Combi_Regimes(y, x, regimes[,...]) GMM method for a spatial lag and error model with regimes and endogenous variables, with results and diagnostics; based on Kelejian and Prucha (1998, 1999) [Kelejian1998] [Kelejian1999].

spreg.GM_Combi_Hom_Regimes(y, x, regimes[,...]) GMM method for a spatial lag and error model with homoskedasticity, regimes and endogenous variables, with results and diagnostics; based on Drukker et al.

spreg.GM_Combi_Het_Regimes(y, x, regimes[,...]) GMM method for a spatial lag and error model with heteroskedasticity, regimes and endogenous variables, with results and diagnostics; based on Arraiz et al [Arraiz2010], following Anselin [Anselin2011].

Continued on next page
Table 206 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spreg.GM_Endog_Error_Regimes(y, x, yend, q, ... )</td>
<td>GMM method for a spatial error model with regimes and endogenous variables, with results and diagnostics; based on Kelejian and Prucha (1998, 1999) [Kelejian1998] [Kelejian1999].</td>
</tr>
<tr>
<td>spreg.GM_Endog_Error_Hom_Regimes(y, x, yend, ...)</td>
<td>GMM method for a spatial error model with homoskedasticity, regimes and endogenous variables.</td>
</tr>
<tr>
<td>spreg.GM_Endog_Error_Het_Regimes(y, x, yend, ...)</td>
<td>GMM method for a spatial error model with heteroskedasticity, regimes and endogenous variables, with results and diagnostics; based on Arraiz et al [Arraiz2010], following Anselin [Anselin2011].</td>
</tr>
</tbody>
</table>

### Seemingly-Unrelated Regressions

Seemingly-unrelated regression models are a generalization of linear regression. These models (and their spatial generalizations) allow for correlation in the residual terms between groups that use the same model. In spatial Seemingly-Unrelated Regressions, the error terms across groups are allowed to exhibit a structured type of correlation: spatial correlation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spreg.SUR(bigy, bigX[, w, regimes, ... ])</td>
<td>User class for SUR estimation, both two step as well as iterated</td>
</tr>
<tr>
<td>spreg.SURerrorGM(bigy, bigX, w[, regimes, ... ])</td>
<td>User class for SUR Error estimation by Maximum Likelihood</td>
</tr>
<tr>
<td>spreg.SURerrorML(bigy, bigX, w[, regimes, ... ])</td>
<td>User class for SUR Error estimation by Maximum Likelihood</td>
</tr>
<tr>
<td>spreg.SURLagIV(bigy, bigX[, bigyend, bigq, ... ])</td>
<td>User class for spatial lag estimation using IV</td>
</tr>
<tr>
<td>spreg.ThreeSLS(bigy, bigX, bigyend, bigq[, ... ])</td>
<td>User class for 3SLS estimation</td>
</tr>
</tbody>
</table>

### pysal.model.spreg.SUR

**class pysal.model.spreg.SUR** (bigy, bigX, w=None, regimes=None, nonspat_diag=True, spat_diag=False, vm=False, iter=False, maxiter=5, epsilon=1e-05, verbose=False, name_bigy=None, name_bigX=None, name_ds=None, name_w=None, name_regimes=None)

User class for SUR estimation, both two step as well as iterated

**Parameters**

- **bigy**  [dictionary with vector for dependent variable by equation]
- **bigX**  [dictionary with matrix of explanatory variables by equation] (note, already includes constant term)
- **w**  [spatial weights object, default = None]
- **regimes**  [list; default = None] List of n values with the mapping of each observation to a regime. Assumed to be aligned with ‘x’.
- **nonspat_diag**  [boolean; flag for non-spatial diagnostics, default = True]
- **spat_diag**  [boolean; flag for spatial diagnostics, default = False]
- **iter**  [boolean; whether or not to use iterated estimation] default = False
- **maxiter**  [integer; maximum iterations; default = 5]
**Examples**

First import pysal.lib to load the spatial analysis tools.

```python
>>> import pysal.lib
```

Open data on NCOVR US County Homicides (3085 areas) using pysal.lib.io.open(). This is the DBF associated with the NAT shapefile. Note that pysal.lib.io.open() also reads data in CSV format.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path("NAT.dbf"),'r')
```

The specification of the model to be estimated can be provided as lists. Each equation should be listed separately. In this example, equation 1 has HR80 as dependent variable and PS80 and UE80 as exogenous regressors. For equation 2, HR90 is the dependent variable, and PS90 and UE90 the exogenous regressors.

```python
>>> y_var = ['HR80','HR90']
>>> x_var = [['PS80','UE80'],['PS90','UE90']]
```

Although not required for this method, we can load a weights matrix file to allow for spatial diagnostics.

```python
>>> w = pysal.lib.weights.Queen.from_shapefile(pysal.lib.examples.get_path("NAT.shp"))
>>> w.transform='r'
```

The SUR method requires data to be provided as dictionaries. PySAL provides the tool sur_dictxy to create these dictionaries from the list of variables. The line below will create four dictionaries containing respectively the dependent variables (bigy), the regressors (bigX), the dependent variables’ names (bigyvars) and regressors’ names (bigXvars). All these will be created from the database (db) and lists of variables (y_var and x_var) created above.

```python
>>> bigy,bigX,bigyvars,bigXvars = pysal.model.spreg.sur_utils.sur_dictxy(db,y_var,x_var)
```

We can now run the regression and then have a summary of the output by typing: ‘print(reg.summary)’

```python
>>> reg = SUR(bigy,bigX,w=w,name_bigy=bigyvars,name_bigX=bigXvars,spat_diag=True,
name_ds="nat")
>>> print (reg.summary)
REGRESSION
---------
SUMMARY OF OUTPUT: SEEMINGLY UNRELATED_REGRESSIONS (SUR)
```

(continues on next page)
<table>
<thead>
<tr>
<th>Data set</th>
<th>nat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weights matrix</td>
<td>unknown</td>
</tr>
<tr>
<td>Number of Equations</td>
<td>2</td>
</tr>
<tr>
<td>Number of Observations</td>
<td>3085</td>
</tr>
<tr>
<td>Log likelihood (SUR):</td>
<td>-19902.966</td>
</tr>
<tr>
<td>Number of Iterations</td>
<td>1</td>
</tr>
</tbody>
</table>

**SUMMARY OF EQUATION 1**

<table>
<thead>
<tr>
<th>Dependent Variable</th>
<th>HR80</th>
<th>Number of Variables</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean dependent var</td>
<td>6.9276</td>
<td>Degrees of Freedom</td>
<td>3082</td>
</tr>
<tr>
<td>S.D. dependent var</td>
<td>6.8251</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Coefficient</th>
<th>Std.Error</th>
<th>z-Statistic</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant_1</td>
<td>5.1390718</td>
<td>0.2624673</td>
<td>19.5798587</td>
<td>0.</td>
</tr>
<tr>
<td>PS80</td>
<td>0.6776481</td>
<td>0.1219578</td>
<td>5.5564132</td>
<td>0.</td>
</tr>
<tr>
<td>UE80</td>
<td>0.2637240</td>
<td>0.0343184</td>
<td>7.6846277</td>
<td>0.</td>
</tr>
</tbody>
</table>

**SUMMARY OF EQUATION 2**

<table>
<thead>
<tr>
<th>Dependent Variable</th>
<th>HR90</th>
<th>Number of Variables</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean dependent var</td>
<td>6.1829</td>
<td>Degrees of Freedom</td>
<td>3082</td>
</tr>
<tr>
<td>S.D. dependent var</td>
<td>6.6403</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Coefficient</th>
<th>Std.Error</th>
<th>z-Statistic</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant_2</td>
<td>3.6139403</td>
<td>0.2534996</td>
<td>14.2561949</td>
<td>0.</td>
</tr>
<tr>
<td>PS90</td>
<td>1.0260715</td>
<td>0.1121662</td>
<td>9.1477755</td>
<td>0.</td>
</tr>
<tr>
<td>UE90</td>
<td>0.3865499</td>
<td>0.0341996</td>
<td>11.3027760</td>
<td>0.</td>
</tr>
</tbody>
</table>

**REGRESSION DIAGNOSTICS**

(continues on next page)
<table>
<thead>
<tr>
<th>TEST</th>
<th>DF</th>
<th>VALUE</th>
<th>PROB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM test on Sigma</td>
<td>1</td>
<td>680.168</td>
<td>0.0000</td>
</tr>
<tr>
<td>LR test on Sigma</td>
<td>1</td>
<td>765.385</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

OTHER DIAGNOSTICS – CHOW TEST BETWEEN EQUATIONS

<table>
<thead>
<tr>
<th>VARIABLES</th>
<th>DF</th>
<th>VALUE</th>
<th>PROB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant_1, Constant_2</td>
<td>1</td>
<td>26.729</td>
<td>0.0000</td>
</tr>
<tr>
<td>PS80, PS90</td>
<td>1</td>
<td>8.241</td>
<td>0.0041</td>
</tr>
<tr>
<td>UE80, UE90</td>
<td>1</td>
<td>9.384</td>
<td>0.0022</td>
</tr>
</tbody>
</table>

DIAGNOSTICS FOR SPATIAL DEPENDENCE

<table>
<thead>
<tr>
<th>TEST</th>
<th>DF</th>
<th>VALUE</th>
<th>PROB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lagrange Multiplier (error)</td>
<td>2</td>
<td>1333.586</td>
<td>0.0000</td>
</tr>
<tr>
<td>Lagrange Multiplier (lag)</td>
<td>2</td>
<td>1275.821</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

ERROR CORRELATION MATRIX

<table>
<thead>
<tr>
<th>EQUATION 1</th>
<th>EQUATION 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000000</td>
<td>0.469548</td>
</tr>
<tr>
<td>0.469548</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

Attributes

- bigy [dictionary with y values]
- bigX [dictionary with X values]
- bigXX [dictionary with $X_t'X_r$ cross-products]
- bigXy [dictionary with $X_t'y_r$ cross-products]
- n_eq [number of equations]
- n [number of observations in each cross-section]
- bigK [vector with number of explanatory variables (including constant)] for each equation
- bOLS [dictionary with OLS regression coefficients for each equation]
- olsE [N x n_eq array with OLS residuals for each equation]
- bSUR [dictionary with SUR regression coefficients for each equation]
- varb [variance-covariance matrix]
- sig [Sigma matrix of inter-equation error covariances]
- ldetS1 [log det(Sigma) for SUR model]
- bigE [n by n_eq array of residuals]
- sig_ols [Sigma matrix for OLS residuals (diagonal)]
- ldetS0 [log det(Sigma) for null model (OLS by equation)]
- niter [number of iterations (=0 for iter=False)]
- corr [inter-equation error correlation matrix]
- llik [log-likelihood (including the constant pi)]
- sur_inf [dictionary with standard error, asymptotic t and p-value,] one for each equation
lrtest  [Likelihood Ratio test on off-diagonal elements of sigma] (tuple with test,df,p-value)
lmtest  [Lagrange Multiplier test on off-diagonal elements of sigma] (tuple with test,df,p-value)
lmEtest  [Lagrange Multiplier test on error spatial autocorrelation in SUR] (tuple with test, df, p-value)
lmlagtest  [Lagrange Multiplier test on spatial lag autocorrelation in SUR] (tuple with test, df, p-value)
surchow  [list with tuples for Chow test on regression coefficients] each tuple contains test value, degrees of freedom, p-value
name_bigy  [dictionary with name of dependent variable for each equation]
name_bigX  [dictionary with names of explanatory variables for each equation]
name_ds  [string; name for the data set]
name_w  [string; name for the weights file]
name_regimes  [string; name of regime variable for use in the output]

__init__ (bigy, bigX, w=None, regimes=None, nonspat_diag=True, spat_diag=False, vm=False, iter=False, maxiter=5, epsilon=1e-05, verbose=False, name_bigy=None, name_bigX=None, name_ds=None, name_w=None, name_regimes=None)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (bigy, bigX[, w, regimes, . . . ]) Initialize self.

pysal.model.spreg.SURerrorGM

class pysal.model.spreg.SURerrorGM (bigy, bigX, w, regimes=None, nonspat_diag=True, spat_diag=False, vm=False, name_bigy=None, name_bigX=None, name_ds=None, name_w=None, name_regimes=None)
User class for SUR Error estimation by Maximum Likelihood

Parameters

bigy  [dictionary with vectors of dependent variable, one for] each equation
bigX  [dictionary with matrices of explanatory variables,] one for each equation
w  [spatial weights object]
regimes  [list] List of n values with the mapping of each observation to a regime. Assumed to be aligned with ‘x’.
nonspat_diag  [boolean; flag for non-spatial diagnostics, default = False]
spat_diag  [boolean; flag for spatial diagnostics, default = False (to be implemented)]
vm  [boolean; flag for asymptotic variance for lambda and Sigma,] default = False (to be imple-mented)
name_bigy  [dictionary with name of dependent variable for each equation] default = None, but should be specified is done when sur_stackxy is used
name_bigX: [dictionary with names of explanatory variables for each] equation default = None, but should be specified is done when sur_stackxy is used
name_ds: [string; name for the data set]
name_w: [string; name for the weights file]
name_regimes: [string; name of regime variable for use in the output]

Examples

First import pysal to load the spatial analysis tools.

```python
>>> import pysal
```

Open data on NCOVR US County Homicides (3085 areas) using pysal.open(). This is the DBF associated with the NAT shapefile. Note that pysal.open() also reads data in CSV format.

```python
>>> db = pysal.open(pysal.examples.get_path("NAT.dbf"), 'r')
```

The specification of the model to be estimated can be provided as lists. Each equation should be listed separately. Equation 1 has HR80 as dependent variable, and PS80 and UE80 as exogenous regressors. For equation 2, HR90 is the dependent variable, and PS90 and UE90 the exogenous regressors.

```python
>>> y_var = ["HR80", "HR90"]
>>> x_var = [["PS80", "UE80"], ["PS90", "UE90"]]
>>> yend_var = [["RD80"], ["RD90"]]
>>> q_var = [["FP79"], ["FP89"]]
```

The SUR method requires data to be provided as dictionaries. PySAL provides the tool sur_dictxy to create these dictionaries from the list of variables. The line below will create four dictionaries containing respectively the dependent variables (bigy), the regressors (bigX), the dependent variables’ names (bigyvars) and regressors’ names (bigXvars). All these will be created from the database (db) and lists of variables (y_var and x_var) created above.

```python
>>> bigy, bigX, bigyvars, bigXvars = pysal.model.spreg.sur_utils.sur_dictxy(db, y_var, x_var)
```

To run a spatial error model, we need to specify the spatial weights matrix. To do that, we can open an already existing gal file or create a new one. In this example, we will create a new one from NAT.shp and transform it to row-standardized.

```python
>>> w = pysal.queen_from_shapefile(pysal.examples.get_path("NAT.shp"))
>>> w.transform = 'r'
```

We can now run the regression and then have a summary of the output by typing: print(reg.summary)
Alternatively, we can just check the betas and standard errors, asymptotic t and p-value of the parameters:

```python
>>> reg = SURerrorGM(bigy, bigX, w, name_bigy=bigyvars, name_bigX=bigXvars, name_ds="NAT", name_w="nat_queen")
>>> reg.bSUR
```

(continues on next page)
Attributes

- **n** [number of observations in each cross-section]
- **n_eq** [number of equations]
- **bigy** [dictionary with vectors of dependent variable, one for each equation]
- **bigX** [dictionary with matrices of explanatory variables, one for each equation]
- **bigK** [n_eq x 1 array with number of explanatory variables by equation]
- **bigylag** [spatially lagged dependent variable]
- **bigXlag** [spatially lagged explanatory variable]
- **lamsur** [spatial autoregressive coefficient in ML SUR Error]
- **bSUR** [beta coefficients in ML SUR Error]
- **varb** [variance of beta coefficients in ML SUR Error]
- **sig** [error variance-covariance matrix in ML SUR Error]
- **bigE** [n by n_eq matrix of vectors of residuals for each equation]
- **sur_inf** [inference for regression coefficients, stand. error, t, p]
- **surchow** [list with tuples for Chow test on regression coefficients] each tuple contains test value, degrees of freedom, p-value
- **name_bigy** [dictionary with name of dependent variable for each equation]
- **name_bigX** [dictionary with names of explanatory variables for each equation]
- **name_ds** [string; name for the data set]
- **name_w** [string; name for the weights file]
- **name_regimes** [string; name of regime variable for use in the output]

**__init__**(bigy, bigX, w[, regimes, ...]) Initialize self.

Methods

**__init__**(bigy, bigX, w[, regimes, ...]) Initialize self.
**pysal.model.spreg.SURerrorML**

**class** `pysal.model.spreg.SURerrorML` *(bigy, bigX, w, regimes=None, nonspat_diag=True, spat_diag=False, vm=False, epsilon=1e-07, name_bigy=None, name_bigX=None, name_ds=None, name_w=None, name_regimes=None)*

User class for SUR Error estimation by Maximum Likelihood

**Parameters**

- **bigy** [dictionary with vectors of dependent variable, one for each equation]
- **bigX** [dictionary with matrices of explanatory variables, one for each equation]
- **w** [spatial weights object]
- **regimes** [list; default = None] List of n values with the mapping of each observation to a regime. Assumed to be aligned with ‘x’.
- **epsilon** [convergence criterion for ML iterations] default 0.0000001
- **nonspat_diag** [boolean; flag for non-spatial diagnostics, default = True]
- **spat_diag** [boolean; flag for spatial diagnostics, default = False]
- **vm** [boolean; flag for asymptotic variance for lambda and Sigma, default = False]
- **name_bigy** [dictionary with name of dependent variable for each equation] default = None, but should be specified is done when sur_stackxy is used
- **name_bigX** [dictionary with names of explanatory variables for each equation] default = None, but should be specified is done when sur_stackxy is used
- **name_ds** [string; name for the data set]
- **name_w** [string; name for the weights file]
- **name_regimes** [string; name of regime variable for use in the output]

**Examples**

First import pysal.lib to load the spatial analysis tools.

```python
>>> import pysal.lib
```

Open data on NCOVR US County Homicides (3085 areas) using pysal.lib.io.open(). This is the DBF associated with the NAT shapefile. Note that pysal.lib.io.open() also reads data in CSV format.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path("NAT.dbf"),'r')
```

The specification of the model to be estimated can be provided as lists. Each equation should be listed separately. Equation 1 has HR80 as dependent variable, and PS80 and UE80 as exogenous regressors. For equation 2, HR90 is the dependent variable, and PS90 and UE90 the exogenous regressors.

```python
>>> y_var = ['HR80','HR90']
>>> x_var = [['PS80','UE80'],['PS90','UE90']]
>>> yend_var = [['RD80'],['RD90']]
>>> q_var = [['FP79'],['FP89']]
```
The SUR method requires data to be provided as dictionaries. PySAL provides the tool sur_dictxy to create these dictionaries from the list of variables. The line below will create four dictionaries containing respectively the dependent variables (bigy), the regressors (bigX), the dependent variables’ names (bigyvars) and regressors’ names (bigXvars). All these will be created from the database (db) and lists of variables (y_var and x_var) created above.

```python
>>> bigy, bigX, bigyvars, bigXvars = pysal.model.spreg.sur_utils.sur_dictxy(db, y_var, x_var)
```

To run a spatial error model, we need to specify the spatial weights matrix. To do that, we can open an already existing gal file or create a new one. In this example, we will create a new one from NAT.shp and transform it to row-standardized.

```python
>>> w = pysal.lib.weights.Queen.from_shapefile(pysal.lib.examples.get_path("NAT.shp"))
>>> w.transform='r'
```

We can now run the regression and then have a summary of the output by typing: print(reg.summary)

Alternatively, we can just check the betas and standard errors, asymptotic t and p-value of the parameters:

```python
>>> reg = SURerrorML(bigy, bigX, w=w, name_bigy=bigyvars, name_bigX=bigXvars, name_ds="NAT", name_w="nat_queen")
>>> reg.bSUR
{0: array([[ 4.0222855 ],
             [ 0.88489646],
             [ 0.42402853]],
            dtype=float64),
  1: array([[ 3.04923009],
             [ 1.10972634],
             [ 0.47075682]],
            dtype=float64))
```

```python
>>> reg.sur_inf
{0: array([[ 0.36692181, 10.96224141, 0. ],
             [ 0.14129077,  6.26294579, 0. ],
             [ 0.04267954,  9.93517021, 0. ]], dtype=float64),
  1: array([[ 0.39139969, 20.200106497, 0. ],
             [ 0.13352591,  8.31094371, 0. ],
             [ 0.04004097, 11.756878 , 0. ]], dtype=float64))
```

Attributes:

- **n** [number of observations in each cross-section]
- **n2** [n/2]
- **n_eq** [number of equations]
- **bigy** [dictionary with vectors of dependent variable, one for each equation]
- **bigX** [dictionary with matrices of explanatory variables, one for each equation]
- **bigK** [n_eq x 1 array with number of explanatory variables] by equation
- **bigylag** [spatially lagged dependent variable]
- **bigXlag** [spatially lagged explanatory variable]
- **lamols** [spatial autoregressive coefficients from equation by] equation ML-Error estimation
- **clikerr** [concentrated log-likelihood from equation by equation] ML-Error estimation (no constant)
**bSUR0** [SUR estimation for betas without spatial autocorrelation]

**llik** [log-likelihood for classic SUR estimation (includes constant)]

**lamsur** [spatial autoregressive coefficient in ML SUR Error]

**bSUR** [beta coefficients in ML SUR Error]

**varb** [variance of beta coefficients in ML SUR Error]

**sig** [error variance-covariance matrix in ML SUR Error]

**bigE** [n by n_eq matrix of vectors of residuals for each equation]

**cliksurerr** [concentrated log-likelihood from ML SUR Error (no constant)]

**sur_inf** [inference for regression coefficients, stand. error, t, p]

**errllik** [log-likelihood for error model without SUR (with constant)]

**surerrllik** [log-likelihood for SUR error model (with constant)]

**lrtest** [likelihood ratio test for off-diagonal Sigma elements]

**likrlambda** [likelihood ratio test on spatial autoregressive coefficients]

**vm** [asymptotic variance matrix for lambda and Sigma (only for vm=True)]

**lamsetp** [inference for lambda, stand. error, t, p (only for vm=True)]

**lamtest** [tuple with test for constancy of lambda across equations] (test value, degrees of freedom, p-value)

**joinlam** [tuple with test for joint significance of lambda across equations] (test value, degrees of freedom, p-value)

**surchow** [list with tuples for Chow test on regression coefficients] each tuple contains test value, degrees of freedom, p-value

**name_bigy** [dictionary with name of dependent variable for each equation]

**name_bigX** [dictionary with names of explanatory variables for each equation]

**name_ds** [string: name for the data set]

**name_w** [string: name for the weights file]

**name_regimes** [string: name of regime variable for use in the output]

```python
__init__(bigy, bigX, w[, regimes, . . . ])
```

Initialize self: See help(type(self)) for accurate signature.

### Methods

```python
__init__(bigy, bigX, w[, regimes, . . . ])  Initialize self.
```
User class for spatial lag estimation using IV

**Parameters**

- **bigy** [dictionary with vector for dependent variable by equation]
- **bigX** [dictionary with matrix of explanatory variables by equation] (note, already includes constant term)
- **bigyend** [dictionary with matrix of endogenous variables by equation] (optional)
- **bigq** [dictionary with matrix of instruments by equation] (optional)
- **w** [spatial weights object, required]
- **vm** [boolean] listing of full variance-covariance matrix, default = False
- **w_lags** [integer] order of spatial lags for WX instruments, default = 1
- **lag_q** [boolean] flag to apply spatial lag to other instruments, default = True
- **nonspat_diag** [boolean; flag for non-spatial diagnostics, default = True]
- **spat_diag** [boolean; flag for spatial diagnostics, default = False]
- **name_bigy** [dictionary with name of dependent variable for each equation] default = None, but should be specified is done when sur_stackxy is used
- **name_bigX** [dictionary with names of explanatory variables for each equation] default = None, but should be specified is done when sur_stackxy is used
- **name_bigyend** [dictionary with names of endogenous variables for each equation] default = None, but should be specified is done when sur_stackZ is used
- **name_bigq** [dictionary with names of instrumental variables for each equation] default = None, but should be specified is done when sur_stackZ is used
- **name_ds** [string; name for the data set]
- **name_w** [string; name for the spatial weights]

**Examples**

First import pysal.lib to load the spatial analysis tools.

```python
>>> import pysal.lib
```

Open data on NCOVR US County Homicides (3085 areas) using pysal.lib.io.open(). This is the DBF associated with the NAT shapefile. Note that pysal.lib.io.open() also reads data in CSV format.

```python
>>> db = pysal.lib.io.open(pysal.lib.examples.get_path("NAT.dbf"), 'r')
```
The specification of the model to be estimated can be provided as lists. Each equation should be listed separately. Although not required, in this example we will specify additional endogenous regressors. Equation 1 has HR80 as dependent variable, PS80 and UE80 as exogenous regressors, RD80 as endogenous regressor and FP79 as additional instrument. For equation 2, HR90 is the dependent variable, PS90 and UE90 the exogenous regressors, RD90 as endogenous regressor and FP99 as additional instrument.

```python
>>> y_var = ['HR80', 'HR90']
>>> x_var = [['PS80', 'UE80'], ['PS90', 'UE90']]
>>> yend_var = [['RD80'], ['RD90']]
>>> q_var = [['FP79'], ['FP99']]
```

The SUR method requires data to be provided as dictionaries. PySAL provides two tools to create these dictionaries from the list of variables: `sur_dictxy` and `sur_dictZ`. The tool `sur_dictxy` can be used to create the dictionaries for Y and X, and `sur_dictZ` for endogenous variables (yend) and additional instruments (q).

```python
>>> bigy, bigX, bigyvars, bigXvars = pysal.model.spreg.sur_utils.sur_dictxy(db, y_var, x_var)
>>> bigyend, bigyendvars = pysal.model.spreg.sur_utils.sur_dictZ(db, yend_var)
>>> bigq, bigqvars = pysal.model.spreg.sur_utils.sur_dictZ(db, q_var)
```

To run a spatial lag model, we need to specify the spatial weights matrix. To do that, we can open an already existing gal file or create a new one. In this example, we will create a new one from NAT.shp and transform it to row-standardized.

```python
>>> w = pysal.lib.weights.Queen.from_shapefile(pysal.lib.examples.get_path("NAT.shp"))
>>> w.transform = 'r'
```

We can now run the regression and then have a summary of the output by typing: `print(reg.summary)`

Alternatively, we can just check the betas and standard errors, asymptotic t and p-value of the parameters:

```python
>>> reg = SURlagIV(bigy, bigX, bigyend, bigq, w=w, name_bigy=bigyvars, name_bigX=bigXvars, name_bigyend=bigyendvars, name_bigq=bigqvars, name_ds="NAT", name_w="nat_queen")
>>> reg.b3SLS
{0: array([[6.95472387],
  [1.44044301],
  [-0.00771893],
  [3.65051153],
  [0.00362663]]), 1: array([[5.61101925],
  [1.38716801],
  [-0.15512029],
  [3.1884457],
  [0.25832185]])}
```

```python
>>> reg.tsls_inf
{0: array([[0.49128435, 14.15620899, 0.0],
  [0.11516292, 12.50787151, 0.0],
  [0.03204088, -0.2409087, 0.80962588],
  [0.1876025, 19.45875745, 0.0],
  [0.05450628, 0.06653605, 0.94695106]],
  1: array([[0.44969956, 12.47726211, 0.0],
  [0.10440241, 13.28674277, 0.0],
  [0.04150243, -3.73761961, 0.00018577],
  [0.19133145, 16.66451427, 0.0],
  [0.04394024, 5.87893596, 0.0]]))}
```
Attributes

\[w\] [spatial weights object]

\[\text{bigy}\] [dictionary with y values]

\[\text{bigZ}\] [dictionary with matrix of exogenous and endogenous variables for each equation]

\[\text{bigyend}\] [dictionary with matrix of endogenous variables for each equation; contains Wy only if no other endogenous specified]

\[\text{bigq}\] [dictionary with matrix of instrumental variables for each equation; contains WX only if no other endogenous specified]

\[\text{bigZHZH}\] [dictionary with matrix of cross products Zhat_r’Zhat_s]

\[\text{bigZHy}\] [dictionary with matrix of cross products Zhat_r’y_end_s]

\[\text{n_eq}\] [number of equations]

\[\text{n}\] [number of observations in each cross-section]

\[\text{bigK}\] [vector with number of explanatory variables (including constant,) exogenous and endogenous for each equation]

\[\text{b2SLS}\] [dictionary with 2SLS regression coefficients for each equation]

\[\text{tssE}\] [N x n_eq array with OLS residuals for each equation]

\[\text{b3SLS}\] [dictionary with 3SLS regression coefficients for each equation]

\[\text{varb}\] [variance-covariance matrix]

\[\text{sig}\] [Sigma matrix of inter-equation error covariances]

\[\text{resids}\] [n by n_eq array of residuals]

\[\text{corr}\] [inter-equation 3SLS error correlation matrix]

\[\text{tss_inf}\] [dictionary with standard error, asymptotic t and p-value, one for each equation]

\[\text{joinrho}\] [test on joint significance of spatial autoregressive coefficient] tuple with test statistic, degrees of freedom, p-value

\[\text{surchow}\] [list with tuples for Chow test on regression coefficients] each tuple contains test value, degrees of freedom, p-value

\[\text{name_w}\] [string; name for the spatial weights]

\[\text{name_ds}\] [string; name for the data set]

\[\text{name_bigy}\] [dictionary with name of dependent variable for each equation]

\[\text{name_bigX}\] [dictionary with names of explanatory variables for each equation]

\[\text{name_bigyend}\] [dictionary with names of endogenous variables for each equation]

\[\text{name_bigq}\] [dictionary with names of instrumental variables for each equation]

\[\text{__init__}\] (bigy, bigX, bigyend=None, bigq=None, w=None, regimes=None, vm=False, regime_lag_sep=False, w_lags=1, lag_q=True, nonspat_diag=True, spat_diag=False, name_bigy=None, name_bigX=None, name_bigyend=None, name_bigq=None, name_ds=None, name_w=None, name_regimes=None)

Initialize self. See help(type(self)) for accurate signature.
Methods

```
__init__(bigy, bigX[, bigyend, bigq, w, . . .])    Initialize self.
```

**pysal.model.mgwr: Multiscale Geographically Weighted Regression**

**GWR Model Estimation and Inference**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mgwr.gwr.GWR(coords, y, X, bw[, family,...])</code></td>
<td>Geographically weighted regression.</td>
</tr>
<tr>
<td><code>mgwr.gwr.GWRResults(model, params, predy, S, CCT)</code></td>
<td>Basic class including common properties for all GWR regression models</td>
</tr>
<tr>
<td><code>mgwr.gwr.GWRResultsLite(model, resid, influ)</code></td>
<td>Lightweight GWR that computes the minimum diagnostics needed for bandwidth selection</td>
</tr>
</tbody>
</table>

**pysal.model.mgwr.gwr.GWR**

```python
class pysal.model.mgwr.gwr.GWR(coords, y, X, bw, family=<pysal.model.spglm.family.Gaussian object>, offset=None, sigma2_v1=True, kernel='bisquare', fixed=False, constant=True, dmat=None, sorted_dmat=None, spherical=False)
```

Geographically weighted regression. Can currently estimate Gaussian, Poisson, and logistic models (built on a GLM framework). GWR object prepares model input. Fit method performs estimation and returns a GWRResults object.

**Parameters**

- **coords** [array-like] n*2, collection of n sets of (x,y) coordinates of observations; also used as calibration locations is ‘points’ is set to None
- **y** [array] n*1, dependent variable
- **X** [array] n*k, independent variable, excluding the constant
- **bw** [scalar] bandwidth value consisting of either a distance or N nearest neighbors; user specified or obtained using Sel_BW
- **family** [family object] underlying probability model; provides distribution-specific calculations
- **offset** [array] n*1, the offset variable at the ith location. For Poisson model this term is often the size of the population at risk or the expected size of the outcome in spatial epidemiology. Default is None where Ni becomes 1.0 for all locations; only for Poisson models
- **sigma2_v1** [boolean] specify form of corrected denominator of sigma squared to use for model diagnostics; Acceptable options are:
  - ‘True’: n-tr(S) (default) ‘False’: n-2(tr(S)+tr(S’S))
- **kernel** [string] type of kernel function used to weight observations; available options: ‘gaussian’ ‘bisquare’ ‘exponential’
- **fixed** [boolean] True for distance based kernel function and False for adaptive (nearest neighbor) kernel function (default)
- **constant** [boolean] True to include intercept (default) in model and False to exclude intercept.
- **dmat** [array] n*n, distance matrix between calibration locations used to compute weight matrix. Defaults to None and is primarily for avoiding duplicate computation during bandwidth selection.
sorted_dmat [array] n*n, sorted distance matrix between calibration locations used to compute
weight matrix. Defaults to None and is primarily for avoiding duplicate computation during
bandwidth selection.

spherical [boolean] True for spherical coordinates (long-lat), False for projected coordinates
default).

Examples

#basic model calibration

```python
>>> import pysal.lib as ps
>>> from mgwr.gwr import GWR
>>> data = ps.io.open(ps.examples.get_path('GData_utm.csv'))
>>> coords = list(zip(data.by_col('X'), data.by_col('Y')))
>>> y = np.array(data.by_col('PctBach')).reshape((-1,1))
>>> rural = np.array(data.by_col('PctRural')).reshape((-1,1))
>>> pov = np.array(data.by_col('PctPov')).reshape((-1,1))
>>> african_amer = np.array(data.by_col('PctBlack')).reshape((-1,1))
>>> X = np.hstack([rural, pov, african_amer])
>>> model = GWR(coords, y, X, bw=90.000, fixed=False, kernel='bisquare')
>>> results = model.fit()
>>> print(results.params.shape)
(159, 4)
```

#predict at unsampled locations

```python
>>> index = np.arange(len(y))
>>> test = index[-10:]
>>> X_test = X[test]
>>> coords_test = np.array(coords)[test]
>>> model = GWR(coords, y, X, bw=94, fixed=False, kernel='bisquare')
>>> results = model.predict(coords_test, X_test)
>>> print(results.params.shape)
(10, 4)
```

Attributes

- **coords** [array-like] n*2, collection of n sets of (x,y) coordinates used for calibration locations
- **y** [array] n*1, dependent variable
- **X** [array] n*k, independent variable, excluding the constant
- **bw** [scalar] bandwidth value consisting of either a distance or N nearest neighbors; user specified or obtained using Sel_BW
- **family** [family object] underlying probability model; provides distribution-specific calculations
- **offset** [array] n*1, the offset variable at the ith location. For Poisson model this term is often the size of the population at risk or the expected size of the outcome in spatial epidemiology. Default is None where Ni becomes 1.0 for all locations
- **sigma2_v1** [boolean] specify form of corrected denominator of sigma squared to use for model diagnostics; Acceptable options are:
  - ‘True’: n-tr(S) (default) ‘False’: n-2(tr(S)+tr(S’S))
- **kernel** [string] type of kernel function used to weight observations; available options: ‘gaussian’ ‘bisquare’ ‘exponential’
fixed [boolean] True for distance based kernel function and False for adaptive (nearest neighbor) kernel function (default)

constant [boolean] True to include intercept (default) in model and False to exclude intercept

dmat [array] n*n, distance matrix between calibration locations used to compute weight matrix. Defaults to None and is primarily for avoiding duplicate computation during bandwidth selection.

sorted_dmat [array] n*n, sorted distance matrix between calibration locations used to compute weight matrix. Defaults to None and is primarily for avoiding duplicate computation during bandwidth selection.

spherical [boolean] True for spherical coordinates (long-lat), False for projected coordinates (default).

n [integer] number of observations

k [integer] number of independent variables

mean_y [float] mean of y

std_y [float] standard deviation of y

fit_params [dict] parameters passed into fit method to define estimation routine

W [array] n*n, spatial weights matrix for weighting all observations from each calibration point

points [array-like] n*2, collection of n sets of (x,y) coordinates used for calibration locations instead of all observations; defaults to None unless specified in predict method

P [array] n*k, independent variables used to make prediction; excluding the constant; default to None unless specified in predict method

exog_scale [scalar] estimated scale using sampled locations; default is None unless specified in predict method

exog_resid [array-like] estimated residuals using sampled locations; default is None unless specified in predict method

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit( )</td>
<td>Method that fits a model with a particular estimation routine.</td>
</tr>
<tr>
<td>predict( )</td>
<td>Method that predicts values of the dependent variable at un-sampled locations</td>
</tr>
</tbody>
</table>

__init__ (coords, y, X, bw, family=<pysal.model.spglm.family.Gaussian object>, offset=None, sigma2_v1=True, kernel='bisquare', fixed=False, constant=True, dmat=None, sorted_dmat=None, spherical=False)

Initialize class

Methods
pysal Documentation, Release 2.0.0

__init__(coords, y, X, bw[, family, offset, ...]) Initialize class

df_model()

df_resid()

fit([ini_params, tol, max_iter, solve, ...]) Method that fits a model with a particular estimation routine.

predict(points, P[, exog_scale, exog_resid, ...]) Method that predicts values of the dependent variable at un-sampled locations

Attributes

mean_y
std_y

pysal.model.mgwr.gwr.GWRResults

class pysal.model.mgwr.gwr.GWRResults (model, params, predy, S, CCT, w=None)
Basic class including common properties for all GWR regression models

Parameters

model [GWR object] pointer to GWR object with estimation parameters
params [array] n*k, estimated coefficients
predy [array] n*1, predicted y values
S [array] n*n, hat matrix
CCT [array] n*k, scaled variance-covariance matrix
w [array] n*1, final weight used for iteratively re-weighted least squares; default is None

Attributes

model [GWR Object] points to GWR object for which parameters have been estimated
params [array] n*k, parameter estimates
predy [array] n*1, predicted value of y
y [array] n*1, dependent variable
X [array] n*k, independent variable, including constant
family [family object] underlying probability model; provides distribution-specific calculations
n [integer] number of observations
k [integer] number of independent variables
df_model [integer] model degrees of freedom
df_resid [integer] residual degrees of freedom
offset [array] n*1, the offset variable at the ith location. For Poisson model this term is often the size of the population at risk or the expected size of the outcome in spatial epidemiology; Default is None where Ni becomes 1.0 for all locations
scale [float] sigma squared used for subsequent computations
w [array] n*1, final weights from iteratively re-weighted least squares routine
resid_response [array] n*1, residuals of the response
resid_ss [scalar] residual sum of squares
W [array] n*n; spatial weights for each observation from each calibration point
S [array] n*n, hat matrix
CCT [array] n*k, scaled variance-covariance matrix
ENP [scalar] effective number of parameters
tr_S [float] trace of S (hat) matrix
tr_STS [float] trace of STS matrix
y_bar [array] weighted mean of y
TSS [array] geographically weighted total sum of squares
RSS [array] geographically weighted residual sum of squares
R2 [float] R-squared for the entire model (1- RSS/TSS)
aic [float] Akaike information criterion
aicc [float] corrected Akaike information criterion to account for model complexity (smaller bandwidths)
bic [float] Bayesian information criterion
localR2 [array] local R square
sigma2 [float] residual variance
std_res [array] standardized residuals
bse [array] standard errors of Betas
influ [array] Influence: leading diagonal of S Matrix
CooksD [array] n*1, Cook’s D
tvalues [array] Return the t-statistic for a given parameter estimate.
adj_alpha [array] Corrected alpha (critical) values to account for multiple testing during hypothesis testing.
deviance [array] n*1, local model deviance for each calibration point
resid_deviance [array] n*1, local sum of residual deviance for each calibration point
llf [scalar] log-likelihood of the full model; see pysal.contrib glm.family for family-specific log-likelihoods
pDev [float] Local percentage of deviance accounted for.
mu [array] n*, flat one dimensional array of predicted mean response value from estimator
fit_params [dict] parameters passed into fit method to define estimation routine
predictions [array] p*1, predicted values generated by calling the GWR predict method to predict dependent variable at unsampled points ()
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENP()</td>
<td>effective number of parameters</td>
</tr>
<tr>
<td>RSS()</td>
<td>geographically weighted residual sum of squares</td>
</tr>
<tr>
<td>TSS()</td>
<td>geographically weighted total sum of squares</td>
</tr>
<tr>
<td>adj_alpha()</td>
<td>Corrected alpha (critical) values to account for multiple testing</td>
</tr>
<tr>
<td>bse()</td>
<td>standard errors of Betas</td>
</tr>
<tr>
<td>conf_int()</td>
<td>Returns the confidence interval of the fitted parameters</td>
</tr>
<tr>
<td>cooksD()</td>
<td>Influence: leading diagonal of S Matrix</td>
</tr>
<tr>
<td>cov_params()</td>
<td>Returns scaled covariance parameters</td>
</tr>
<tr>
<td>critical_tval()</td>
<td>Utility function to derive the critical t-value based on given alpha</td>
</tr>
<tr>
<td>filter_tvals()</td>
<td>Utility function to set t-values with an absolute value smaller than the</td>
</tr>
<tr>
<td></td>
<td>absolute value of the alpha (critical) value to 0.</td>
</tr>
<tr>
<td>influ()</td>
<td>Influence: leading diagonal of S Matrix</td>
</tr>
<tr>
<td>localR2()</td>
<td>local R square</td>
</tr>
<tr>
<td>local_collinearity()</td>
<td>Computes several indicators of multicollinearity</td>
</tr>
<tr>
<td>pDev()</td>
<td>Local percentage of deviance accounted for</td>
</tr>
<tr>
<td>sigma2()</td>
<td>residual variance</td>
</tr>
<tr>
<td>spatial_variability()</td>
<td>Method to compute a Monte Carlo test of spatial variability for</td>
</tr>
<tr>
<td></td>
<td>each estimated coefficient surface.</td>
</tr>
<tr>
<td>std_res()</td>
<td>standardized residuals</td>
</tr>
<tr>
<td>summary()</td>
<td>Print out GWR summary</td>
</tr>
<tr>
<td>tr_S()</td>
<td>trace of S (hat) matrix</td>
</tr>
<tr>
<td>tr_STS()</td>
<td>trace of STS matrix</td>
</tr>
<tr>
<td>tvalues()</td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td>use_t()</td>
<td>bool(x) -&gt; bool</td>
</tr>
<tr>
<td>y_bar()</td>
<td>weighted mean of y</td>
</tr>
</tbody>
</table>
D2
R2
adj_D2
adj_pseudoR2
aic
aicc
bic
deviance
df_model
df_resid
initialize
llf
llnull
normalized_cov_params
null
null_deviance
pearson_chi2
predictions
pseudoR2
pvalues
resid_anscombe
resid_deviance
resid_pearson
resid_response
resid_ss
resid_working
scale

```
__init__(model, params, predy, S, CCT, w= None)
Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

D2() effective number of parameters
ENP() geographically weighted residual sum of squares
R2() geographically weighted total sum of squares
__init__(model, params, predy, S, CCT, w) Initialize self.
adj_D2()
adj_alpha() Corrected alpha (critical) values to account for multiple testing during hypothesis testing.
adj_pseudoR2()
aic()
aicc()
bic()
bse() standard errors of Betas
conf_int() Returns the confidence interval of the fitted parameters.

Continued on next page
Table 217 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cooksD()</td>
<td>Influence: leading diagonal of S Matrix</td>
</tr>
<tr>
<td>cov_params(cov[, exog_scale])</td>
<td>Returns scaled covariance parameters</td>
</tr>
<tr>
<td>critical_tvals([alpha])</td>
<td>Utility function to derive the critical t-value based on given alpha that are needed for hypothesis testing</td>
</tr>
<tr>
<td>deviance()</td>
<td></td>
</tr>
<tr>
<td>df_model()</td>
<td></td>
</tr>
<tr>
<td>df_resid()</td>
<td></td>
</tr>
<tr>
<td>filter_tvals([critical_t, alpha])</td>
<td>Utility function to set tvalues with an absolute value smaller than the absolute value of the alpha (critical) value to 0.</td>
</tr>
<tr>
<td>influ()</td>
<td>Influence: leading diagonal of S Matrix</td>
</tr>
<tr>
<td>initialize(model, params, **kwd)</td>
<td></td>
</tr>
<tr>
<td>llf()</td>
<td></td>
</tr>
<tr>
<td>linull()</td>
<td>local R square</td>
</tr>
<tr>
<td>localR2()</td>
<td>local R square</td>
</tr>
<tr>
<td>local_collinearity()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>normalized_cov_params()</td>
<td></td>
</tr>
<tr>
<td>null()</td>
<td></td>
</tr>
<tr>
<td>null_deviance()</td>
<td></td>
</tr>
<tr>
<td>pDev()</td>
<td>Local percentage of deviance accounted for.</td>
</tr>
<tr>
<td>pearson_chi2()</td>
<td></td>
</tr>
<tr>
<td>predictions()</td>
<td></td>
</tr>
<tr>
<td>pseudoR2()</td>
<td></td>
</tr>
<tr>
<td>pvalues()</td>
<td></td>
</tr>
<tr>
<td>resid_anscombe()</td>
<td></td>
</tr>
<tr>
<td>resid_deviance()</td>
<td></td>
</tr>
<tr>
<td>resid_pearson()</td>
<td></td>
</tr>
<tr>
<td>resid_response()</td>
<td></td>
</tr>
<tr>
<td>resid_ss()</td>
<td></td>
</tr>
<tr>
<td>resid_working()</td>
<td></td>
</tr>
<tr>
<td>scale(scale)</td>
<td></td>
</tr>
<tr>
<td>sigma2()</td>
<td>residual variance</td>
</tr>
<tr>
<td>spatial_variability(selector[, seed])</td>
<td>Method to compute a Monte Carlo test of spatial variability for each estimated coefficient surface.</td>
</tr>
<tr>
<td>std_res()</td>
<td>standardized residuals</td>
</tr>
<tr>
<td>summary()</td>
<td>Print out GWR summary</td>
</tr>
<tr>
<td>tr_S()</td>
<td>trace of S (hat) matrix</td>
</tr>
<tr>
<td>tr_STS()</td>
<td>trace of STS matrix</td>
</tr>
<tr>
<td>tvalues()</td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td>use_t()</td>
<td>bool(x) -&gt; bool</td>
</tr>
<tr>
<td>y_bar()</td>
<td>weighted mean of y</td>
</tr>
</tbody>
</table>

**pysal.model.mgwr.gwr.GWRResultsLite**

**class** pysal.model.mgwr.gwr.GWRResultsLite *(model, resid, infl)*

Lightweight GWR that computes the minimum diagnostics needed for bandwidth selection

**Parameters**

- **model** [GWR object] pointer to GWR object with estimation parameters
resid  [array] n*1, residuals of the response
influ  [array] n*1, leading diagonal of S matrix

Attributes

tr_S  [float] trace of S (hat) matrix
llf  [scalar] log-likelihood of the full model; see pysal.contrib.glm.family for family-specific
     log-likelihoods
mu  [array] n*, flat one dimensional array of predicted mean response value from estimator
resid_ss  [scalar] residual sum of squares

Methods

__init__ (model, resid, influ)
   Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (model, resid, influ)          Initialize self.
llf()
mu()
resid_ss()                            
tr_S()

MGWR Estimation and Inference

mgwr.gwr.MGWR(coords, y, X, selector[, ...])  Multiscale GWR estimation and inference.
mgwr.gwr.MGWRResults(model, params, predy, ...
                       ... )                        Class including common properties for a MGWR model.

pysal.model.mgwr.gwr.MGWR

class pysal.model.mgwr.gwr.MGWR(coords, y, X, selector, sigma2_yI=True, kernel='bisquare',
fixed=False, constant=True, dmat=None, sorted_dmat=None, spherical=True)

Multiscale GWR estimation and inference.

Parameters

coords  [array-like] n*2, collection of n sets of (x,y) coordinates of observations; also used as
calibration locations if ‘points’ is set to None
y  [array] n*1, dependent variable
X  [array] n*k, independent variable, excluding the constant
selector [sel_bw object] valid sel_bw object that has successfully called the “search” method. This parameter passes on information from GAM model estimation including optimal bandwidths.

family [family object] underlying probability model; provides distribution-specific calculations

sigma2_v1 [boolean] specify form of corrected denominator of sigma squared to use for model diagnostics; Acceptable options are:
- ‘True’: n-tr(S) (default)
- ‘False’: n-2(tr(S)+tr(S’))

kernel [string] type of kernel function used to weight observations; available options: ‘gaussian’ ‘bisquare’ ‘exponential’

fixed [boolean] True for distance based kernel function and False for adaptive (nearest neighbor) kernel function (default)

constant [boolean] True to include intercept (default) in model and False to exclude intercept.

dmat [array] n*n, distance matrix between calibration locations used to compute weight matrix. Defaults to None and is primarily for avoiding duplicate computation during bandwidth selection.

sorted_dmat [array] n*n, sorted distance matrix between calibration locations used to compute weight matrix. Defaults to None and is primarily for avoiding duplicate computation during bandwidth selection.

spherical [boolean] True for spherical coordinates (long-lat), False for projected coordinates (default).

Examples

#basic model calibration

```python
>>> import pysal.lib as ps
>>> from mgwr.gwr import MGWR
>>> from mgwr.sel_bw import Sel_BW

# import data
>>> data = ps.io.open(ps.examples.get_path('GData_utm.csv'))

# get coordinates and response variables
>>> coords = list(zip(data.by_col('X'), data.by_col('Y')))
>>> y = np.array(data.by_col('PctBach')).reshape((-1,1))
>>> rural = np.array(data.by_col('PctRural')).reshape((-1,1))
>>> fb = np.array(data.by_col('PctFB')).reshape((-1,1))
>>> african_amer = np.array(data.by_col('PctBlack')).reshape((-1,1))

# center data
>>> X = (X - X.mean(axis=0)) / X.std(axis=0)

# scale data
>>> y = (y - y.mean(axis=0)) / y.std(axis=0)

# select optimal bandwidth
>>> selector = Sel_BW(coords, y, X, multi=True)
>>> selector.search(multi_bw_min=[2])
[92.0, 101.0, 136.0, 158.0]

# fit model
>>> model = MGWR(coords, y, X, selector, fixed=False, kernel='bisquare', sigma2_v1=True)
>>> results = model.fit()
>>> print(results.params.shape)
(159, 4)
```

Attributes

coords [array-like] n*2, collection of n sets of (x,y) coordinates of observations; also used as calibration locations is ‘points’ is set to None.
y [array] n*1, dependent variable
X [array] n*k, independent variable, excluding the constant
selector [sel_bw object] valid sel_bw object that has successfully called the “search” method. This parameter passes on information from GAM model estimation including optimal bandwidths.
bw [array-like] collection of bandwidth values consisting of either a distance or N nearest neighbors; user specified or obtained using Sel_BW with fb=True. Order of values should the same as the order of columns associated with X
family [family object] underlying probability model; provides distribution-specific calculations
sigma2_v1 [boolean] specify form of corrected denominator of sigma squared to use for model diagnostics; Acceptable options are:
   ‘True’: n-tr(S) (default) ‘False’: n-2(tr(S)+tr(S’S))
kernal [string] type of kernel function used to weight observations; available options: ‘gaussian’ ‘bisquare’ ‘exponential’
fixed [boolean] True for distance based kernel function and False for adaptive (nearest neighbor) kernel function (default)
constant [boolean] True to include intercept (default) in model and False to exclude intercept.
dmat [array] n*n, distance matrix between calibration locations used to compute weight matrix. Defaults to None and is primarily for avoiding duplicate computation during bandwidth selection.
sorted_dmat [array] n*n, sorted distance matrix between calibration locations used to compute weight matrix. Defaults to None and is primarily for avoiding duplicate computation during bandwidth selection.
spherical [boolean] True for spherical coordinates (long-lat), False for projected coordinates (default).
n [integer] number of observations
k [integer] number of independent variables
mean_y [float] mean of y
std_y [float] standard deviation of y
fit_params [dict] parameters passed into fit method to define estimation routine
W [array-like] list of n*n arrays, spatial weights matrices for weighting all observations from each calibration point: one for each covariate (k)

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit()</td>
<td>Method that extracts information from Sel_BW (selector) object and prepares GAM estimation results for MGWRResults object.</td>
</tr>
<tr>
<td>predict()</td>
<td>Not implemented.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DataFrame</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>df_model</td>
<td></td>
</tr>
<tr>
<td>df_resid</td>
<td></td>
</tr>
</tbody>
</table>
__init__(coords, y, X, selector, sigma2_v1=True, kernel='bisquare', fixed=False, constant=True, dmat=None, sorted_dmat=None, spherical=False)

Initialize class

Methods

__init__(coords, y, X, selector[, ...]) Initialize class
df_model()
df_resid()

fit() Method that extracts information from Sel_BW (selector) object and prepares GAM estimation results for MGWRResults object.
predict() Not implemented.

Attributes

mean_y
std_y

pysal.model.mgwr.gwr.MGWRResults

class pysal.model.mgwr.gwr.MGWRResults (model, params, predy, S, CCT, R, w)
Class including common properties for a MGWR model.

Parameters

model [MGWR object] pointer to MGWR object with estimation parameters
params [array] n*k, estimated coefficients
predy [array] n*1, predicted y values
S [array] n*n, hat matrix
R [array] n*n*k, partial hat matrices for each covariate
CCT [array] n*k, scaled variance-covariance matrix
w [array] n*1, final weight used for iteratively re-weighted least squares; default is None

Attributes

model [GWR Object] points to GWR object for which parameters have been estimated
params [array] n*k, parameter estimates
predy [array] n*1, predicted value of y
y [array] n*1, dependent variable
X [array] n*k, independent variable, including constant
family [family object] underlying probability model; provides distribution-specific calculations
n [integer] number of observations
k [integer] number of independent variables
df_model [integer] model degrees of freedom
df_resid  [integer] residual degrees of freedom
scale  [float] sigma squared used for subsequent computations
w  [array] n*1, final weights from iteratively re-weighted least squares routine
resid_response  [array] n*1, residuals of the response
resid_ss  [scalar] residual sum of squares
W  [array-like] list of n*n arrays, spatial weights matrices for weighting all observations from each calibration point: one for each covariate (k)
S  [array] n*n, hat matrix
R  [array] n*n*k, partial hat matrices for each covariate
CCT  [array] n*k, scaled variance-covariance matrix
ENP  [scalar] effective number of parameters
ENP_j  [array-like] effective number of parameters, which depends on sigma2, for each covariate in the model
adj_alpha  [array] Corrected alpha (critical) values to account for multiple testing during hypothesis testing.
adj_alpha_j  [array] Corrected alpha (critical) values to account for multiple testing during hypothesis testing.
tr_S  [float] trace of S (hat) matrix
tr_STS  [float] trace of STS matrix
R2  [float] R-squared for the entire model (1- RSS/TSS)
aic  [float] Akaike information criterion
aicc  [float] corrected Akaike information criterion to account for model complexity (smaller bandwidths)
bic  [float] Bayesian information criterion
sigma2  [float] residual variance
std_res  [array] standardized residuals
bse  [array] standard errors of Betas
influ  [array] Influence: leading diagonal of S Matrix
CooksD  [array] n*1, Cook's D
tvalues  [array] Return the t-statistic for a given parameter estimate.
llf  [scalar] log-likelihood of the full model; see pysal.contrib.glm.family for family-specific
log-likelihoods
mu  [array] n*, flat one dimensional array of predicted mean response value from estimator

Methods

<table>
<thead>
<tr>
<th>ENP()</th>
<th>effective number of parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSS()</td>
<td>geographically weighted residual sum of squares</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSS()</td>
<td>Geographically weighted total sum of squares</td>
</tr>
<tr>
<td>adj_alpha()</td>
<td>Corrected alpha (critical) values to account for multiple testing during hypothesis testing.</td>
</tr>
<tr>
<td>adj_alpha_j()</td>
<td>Corrected alpha (critical) values to account for multiple testing during hypothesis testing.</td>
</tr>
<tr>
<td>bse()</td>
<td>Standard errors of Betas</td>
</tr>
<tr>
<td>conf_int()</td>
<td>Returns the confidence interval of the fitted parameters</td>
</tr>
<tr>
<td>cooksD()</td>
<td>Influence: leading diagonal of S Matrix</td>
</tr>
<tr>
<td>cov_params(cov[, exog_scale])</td>
<td>Returns scaled covariance parameters</td>
</tr>
<tr>
<td>critical_tval([alpha])</td>
<td>Utility function to derive the critical t-value based on given alpha that are needed for hypothesis testing</td>
</tr>
<tr>
<td>filter_tvals([critical_t, alpha])</td>
<td>Utility function to set t-values with an absolute value smaller than the absolute value of the alpha (critical) value to 0.</td>
</tr>
<tr>
<td>influ()</td>
<td>Influence: leading diagonal of S Matrix</td>
</tr>
<tr>
<td>localR2()</td>
<td>Local R square</td>
</tr>
<tr>
<td>local_collinearity()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>pDev()</td>
<td>Local percentage of deviance accounted for.</td>
</tr>
<tr>
<td>sigma2()</td>
<td>Residual variance</td>
</tr>
<tr>
<td>spatial_variability(selector[, n_iters, seed])</td>
<td>Method to compute a Monte Carlo test of spatial variability for each estimated coefficient surface.</td>
</tr>
<tr>
<td>std_res()</td>
<td>Standardized residuals</td>
</tr>
<tr>
<td>summary()</td>
<td>Print out MGWR summary</td>
</tr>
<tr>
<td>tr_S()</td>
<td>Trace of S (hat) matrix</td>
</tr>
<tr>
<td>tr_STS()</td>
<td>Trace of STS matrix</td>
</tr>
<tr>
<td>tvalues()</td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td>use_t()</td>
<td>Bool(x) -&gt; bool</td>
</tr>
<tr>
<td>y_bar()</td>
<td>Weighted mean of y</td>
</tr>
</tbody>
</table>
Methods

__init__(model, params, predy, S, CCT, R, w)
   Initialize class

D2()                             effective number of parameters
ENP_j()                          effective number of parameters
R2()                             geographically weighted residual sum of squares
__init__(model, params, predy, S, CCT, R, w) Initialize class
adj_D2()                         Corrected alpha (critical) values to account for multiple testing during hypothesis testing.
adj_alpha()                      Corrected alpha (critical) values to account for multiple testing during hypothesis testing.
adj_alpha_j()                    Corrected alpha (critical) values to account for multiple testing during hypothesis testing.
adj_pseudoR2()                   geographically weighted total sum of squares
aic()                            geographically weighted total sum of squares
aicc()                           geographically weighted total sum of squares
bic()                            geographically weighted total sum of squares
deviance()                       geographically weighted total sum of squares
df_model()                       geographically weighted total sum of squares
df_resid()                       geographically weighted total sum of squares
initialize()                     geographically weighted total sum of squares
llf()                            geographically weighted total sum of squares
lnull()                          geographically weighted total sum of squares
null_deviance()                  geographically weighted total sum of squares
pearson_chi2()                   geographically weighted total sum of squares
predictions()                    geographically weighted total sum of squares
pseudoR2()                       geographically weighted total sum of squares
pvalues()                        geographically weighted total sum of squares
resid_anscombe()                 geographically weighted total sum of squares
resid_deviance()                 geographically weighted total sum of squares
resid_pearson()                  geographically weighted total sum of squares
resid_response()                 geographically weighted total sum of squares
resid_ss()                       geographically weighted total sum of squares
resid_working()                  geographically weighted total sum of squares
scale()                          geographically weighted total sum of squares

Continued on next page
### Table 224 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bic()</td>
<td>Standard errors of Betas</td>
</tr>
<tr>
<td>bse()</td>
<td>Returns the confidence interval of the fitted parameters.</td>
</tr>
<tr>
<td>conf_int()</td>
<td>Influence: leading diagonal of S Matrix</td>
</tr>
<tr>
<td>cooksD()</td>
<td>Returns scaled covariance parameters</td>
</tr>
<tr>
<td>critical_tval(alpha)</td>
<td>Utility function to derive the critical t-value based on given alpha that are needed for hypothesis testing</td>
</tr>
<tr>
<td>deviance()</td>
<td>Returns scaled covariance parameters</td>
</tr>
<tr>
<td>df_model()</td>
<td>Influence: leading diagonal of S Matrix</td>
</tr>
<tr>
<td>df_resid()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>filter_tvals([critical_t, alpha])</td>
<td>Utility function to set tvalues with an absolute value smaller than the absolute value of the alpha (critical) value to 0.</td>
</tr>
<tr>
<td>influ()</td>
<td>Local R square</td>
</tr>
<tr>
<td>initialize(model, params, **kwd)</td>
<td>Local percentage of deviance accounted for.</td>
</tr>
<tr>
<td>llf()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>llnull()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>localR2()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>local_collinearity()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>normalized_cov_params()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>null()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>null_deviance()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>pDev()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>pearson_chl2()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>predictions()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>pseudoR2()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>pvalues()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>resid_anscombe()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>resid_deviance()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>resid_pearson()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>resid_response()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>resid_ss()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>resid_working()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>scale([scale])</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>sigma2()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>spatial_variability(selector[, seed])</td>
<td>Method to compute a Monte Carlo test of spatial variability for each estimated coefficient surface.</td>
</tr>
<tr>
<td>std_res()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>summary()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>tr_S()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>tr_STS()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>tvalues()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>use_t()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
<tr>
<td>y_bar()</td>
<td>Computes several indicators of multicollinearity within a geographically weighted design matrix, including:</td>
</tr>
</tbody>
</table>

**Kernel Specification**

1.2. API Reference 315
mgwr.kernels.fix_gauss(coords, bw[, points, ...])  Fixed Gaussian kernel.

mgwr.kernels.adapt_gauss(coords, nn[,...])  Spatially adaptive Gaussian kernel.

mgwr.kernels.fix_bisquare(coords, bw[, ...])  Fixed bisquare kernel.

mgwr.kernels.adapt_bisquare(coords, nn[, ...])  Spatially adaptive bisquare kernel.

mgwr.kernels.fix_exp(coords, bw[, points, ...])  Fixed exponential kernel.

mgwr.kernels.adapt_exp(coords, nn[, points, ...])  Spatially adaptive exponential kernel.

pysal.model.mgwr.kernels.fix_gauss

pysal.model.mgwr.kernels.fix_gauss(coords, bw, points=None, dmat=None, sorted_dmat=None, spherical=False)  Fixed Gaussian kernel.

pysal.model.mgwr.kernels.adapt_gauss

pysal.model.mgwr.kernels.adapt_gauss(coords, nn, points=None, dmat=None, sorted_dmat=None, spherical=False)  Spatially adaptive Gaussian kernel.

pysal.model.mgwr.kernels.fix_bisquare

pysal.model.mgwr.kernels.fix_bisquare(coords, bw, points=None, dmat=None, sorted_dmat=None, spherical=False)  Fixed bisquare kernel.

pysal.model.mgwr.kernels.adapt_bisquare

pysal.model.mgwr.kernels.adapt_bisquare(coords, nn, points=None, dmat=None, sorted_dmat=None, spherical=False)  Spatially adaptive bisquare kernel.

pysal.model.mgwr.kernels.fix_exp

pysal.model.mgwr.kernels.fix_exp(coords, bw, points=None, dmat=None, sorted_dmat=None, spherical=False)  Fixed exponential kernel.

pysal.model.mgwr.kernels.adapt_exp

pysal.model.mgwr.kernels.adapt_exp(coords, nn, points=None, dmat=None, sorted_dmat=None, spherical=False)  Spatially adaptive exponential kernel.
Bandwidth Selection

```python
mgwr.sel_bw.Sel_BW(coords, y, X_loc[, ...]) Select bandwidth for kernel
```

**pysal.model.mgwr.sel_bw.Sel_BW**

```python
class pysal.model.mgwr.sel_bw.Sel_BW(coords, y, X_loc, X_glob=None, family=<pysal.model.spglm.family.Gaussian object>, offset=None, kernel='bisquare', fixed=False, multi=False, constant=True, spherical=False)
```

Select bandwidth for kernel

Methods: p211 - p213, bandwidth selection


**Parameters**

- `y` [array] n*1, dependent variable.
- `X_glob` [array] n*k1, fixed independent variable.
- `X_loc` [array] n*k2, local independent variable, including constant.
- `coords` [list of tuples] (x,y) of points used in bandwidth selection
- `offset` [array] n*1, the offset variable at the ith location. For Poisson model this term is often the size of the population at risk or the expected size of the outcome in spatial epidemiology. Default is None where Ni becomes 1.0 for all locations
- `fixed` [boolean] True for fixed bandwidth and False for adaptive (NN)
- `multi` [True for multiple (covariate-specific) bandwidths] False for a traditional (same for all covariates) bandwidth; default is False.
- `constant` [boolean] True to include intercept (default) in model and False to exclude intercept.
- `spherical` [boolean] True for spherical coordinates (long-lat), False for projected coordinates (default).

**Examples**

```python
>>> import pysal.lib as ps
>>> from mgwr.sel_bw import Sel_BW
>>> data = ps.io.open(ps.examples.get_path('GData_utm.csv'))
>>> coords = list(zip(data.by_col('X'), data.by_col('Y')))
>>> y = np.array(data.by_col('PctBach')).reshape((-1,1))
>>> rural = np.array(data.by_col('PctRural')).reshape((-1,1))
>>> pov = np.array(data.by_col('PctPov')).reshape((-1,1))
>>> african_amer = np.array(data.by_col('PctBlack')).reshape((-1,1))
>>> X = np.hstack([rural, pov, african_amer])
```

Golden section search AICc - adaptive bisquare
Golden section search AIC - adaptive Gaussian

```python
>>> bw = Sel_BW(coords, y, X).search(criterion='AICc')
>>> print(bw)
93.0
```

Golden section search BIC - adaptive Gaussian

```python
>>> bw = Sel_BW(coords, y, X, kernel='gaussian').search(criterion='BIC')
>>> print(bw)
62.0
```

Golden section search CV - adaptive Gaussian

```python
>>> bw = Sel_BW(coords, y, X, kernel='gaussian').search(criterion='CV')
>>> print(bw)
68.0
```

Interval AICc - fixed bisquare

```python
>>> sel = Sel_BW(coords, y, X, fixed=True)
>>> bw = sel.search(search_method='interval', bw_min=211001.0, bw_max=211035.0, interval=2)
>>> print(bw)
211025.0
```

Attributes

- `y` [array] n*1, dependent variable.
- `X_glob` [array] n*k1, fixed independent variable.
- `X_loc` [array] n*k2, local independent variable, including constant.
- `coords` [list of tuples] (x,y) of points used in bandwidth selection
- `kernel` [string] type of kernel used and wether fixed or adaptive
- `fixed` [boolean] True for fixed bandwidth and False for adaptive (NN)
- `bw_min` [float] min value used in bandwidth search
- `bw_max` [float] max value used in bandwidth search
- `interval` [float] interval increment used in interval search
- `tol` [float] tolerance used to determine convergence
- `max_iter` [integer] max interations if no convergence to tol
- `multi` [True for multiple (covariate-specific) bandwidths] False for a traditional (same for all covariates) bandwidth; defualt is False.
constant [boolean] True to include intercept (default) in model and False to exclude intercept.

offset [array] n*1, the offset variable at the ith location. For Poisson model this term is often the size of the population at risk or the expected size of the outcome in spatial epidemiology. Default is None where Ni becomes 1.0 for all locations.

dmat [array] n*n, distance matrix between calibration locations used to compute weight matrix.

sorted_dmat [array] n*n, sorted distance matrix between calibration locations used to compute weight matrix. Will be None for fixed bandwidths.

spherical [boolean] True for spherical coordinates (long-lat), False for projected coordinates (default).

search_params [dict] stores search arguments.

int_score [boolean] True if adaptive bandwidth is being used and bandwidth selection should be discrete. False if fixed bandwidth is being used and bandwidth does not have to be discrete.

bw [scalar or array-like] Derived optimal bandwidth(s). Will be a scalar for GWR (multi=False) and a list of scalars for MGWR (multi=True) with one bandwidth for each covariate.

S [array] n*n, hat matrix derived from the iterative backfitting algorithm for MGWR during bandwidth selection.

R [array] n*n*k, partial hat matrices derived from the iterative backfitting algorithm for MGWR during bandwidth selection. There is one n*n matrix for each of the k covariates.

params [array] n*k, calibrated parameter estimates for MGWR based on the iterative backfitting algorithm - computed and saved here to avoid having to do it again in the MGWR object.

Methods

search([search_method, criterion, bw_min, ...]) Method to select one unique bandwidth for a gwr model or a bandwidth vector for a mgwr model.

__init__(coords, y, X_loc[, X_glob, family, ..., fixed=False, multi=False, constant=True, spherical=False]) Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(coords, y, X_loc[, X_glob, family, ...]) Initialize self.

search([search_method, criterion, bw_min, ...]) Method to select one unique bandwidth for a gwr model or a bandwidth vector for a mgwr model.

Visualization

mgwr.utils.shift_colormap(cmap[, start, ...]) Function to offset the “center” of a colormap.

mgwr.utils.truncate_colormap(cmap[, min-val, ...]) Function to truncate a colormap by selecting a subset of the original colormap’s values.

Continued on next page
Table 229 – continued from previous page

```python
mgwr.utils.compare_surfaces(data, var1, ...)  # Function that creates comparative visualization of GWR and MGWR surfaces.
```

**pysal.model.mgwr.utils.shift_colormap**

```python
pysal.model.mgwr.utils.shift_colormap(cmap, start=0, midpoint=0.5, stop=1.0, name='shiftedcmap')
```

Function to offset the “center” of a colormap. Useful for data with a negative min and positive max and you want the middle of the colormap’s dynamic range to be at zero

**Parameters**

- `cmap` [The matplotlib colormap to be altered]
- `start` [Offset from lowest point in the colormap’s range.] Defaults to 0.0 (no lower offset). Should be between 0.0 and `midpoint`.
- `midpoint` [The new center of the colormap. Defaults to] 0.5 (no shift). Should be between 0.0 and 1.0. In general, this should be `1 - vmax/(vmax + abs(vmin))` For example if your data range from -15.0 to +5.0 and you want the center of the colormap at 0.0, `midpoint` should be set to `1 - 5/(5 + 15)) or 0.75
- `stop` [Offset from highest point in the colormap’s range.] Defaults to 1.0 (no upper offset). Should be between `midpoint` and 1.0.

**Returns**

- `new_cmap` [A new colormap that has been shifted.]  

**pysal.model.mgwr.utils.truncate_colormap**

```python
pysal.model.mgwr.utils.truncate_colormap(cmap, minval=0.0, maxval=1.0, n=100)
```

Function to truncate a colormap by selecting a subset of the original colormap’s values

**Parameters**

- `cmap` [Matplotlib colormap to be altered]
- `minval` [Minimum value of the original colormap to include in the truncated colormap]
- `maxval` [Maximum value of the original colormap to include in the truncated colormap]
- `n` [Number of intervals between the min and max values for the gradient of the truncated colormap]

**Returns**

- `new_cmap` [A new colormap that has been shifted.]  

**pysal.model.mgwr.utils.compare_surfaces**

```python
pysal.model.mgwr.utils.compare_surfaces(data, var1, var2, gwr_f, gwr_bw, mgwr_f, mgwr_bw, name, kwargs1, kwargs2, save_fig=None)
```

Function that creates comparative visualization of GWR and MGWR surfaces.

**Parameters**

- `data` [pandas or geopandas Dataframe] gwr/mgwr results
var1 : [string] name of gwr parameter estimate column in frame
var2 : [string] name of mgwr parameter estimate column in frame
gwr_t : [string] name of gwr t-values column in frame associated with var1
gwr_bw : [float] bandwidth for gwr model for var1
mgwr_t : [string] name of mgwr t-values column in frame associated with var2
mgwr_bw : float bandwidth for mgwr model for var2
name : [string] common variable name to use for title
kwargs1 : additional plotting arguments for gwr surface
kwargs2 : additional plotting arguments for mgwr surface
savefig : string, optional path to save the figure. Default is None. Not to save figure.

1.3 References

Details are available in the PySAL api.
For background information see [RA07].

1.4 Development

As of version 2.0.0, PySAL is now a collection of affiliated geographic data science packages. Changes to the code for any of the subpackages should be directed at the respective upstream repositories and not made here. Infrastructural changes for the meta-package, like those for tooling, building the package, and code standards, will be considered.
PySAL development is hosted on github.
Discussions of development occurs on the developer list as well as gitter.

1.5 Getting Involved

If you are interested in contributing to PySAL please see our development guidelines.

1.6 Bug reports

To search for or report bugs, please see PySAL’s issues.

1.7 Citing PySAL

If you use PySAL in a scientific publication, we would appreciate citations to the following paper:


Bibtex entry:
1.8 License information

See the file “LICENSE.txt” for information on the history of this software, terms & conditions for usage, and a DISCLAIMER OF ALL WARRANTIES.
<table>
<thead>
<tr>
<th>Ref</th>
<th>Author(s)</th>
<th>Title</th>
<th>Journal</th>
<th>Year</th>
<th>URL</th>
<th>DOI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chr05</td>
<td>David Christensen</td>
<td>Fast algorithms for the calculation of kendall’s $\tau$</td>
<td>Computational Statistics</td>
<td>2005</td>
<td><a href="https://doi.org/10.1007/BF02736122">https://doi.org/10.1007/BF02736122</a></td>
<td>10.1007/BF02736122</td>
</tr>
<tr>
<td>Dij59</td>
<td>E. W. Dijkstra</td>
<td>A Note on Two Problen in Connexion with Graphs</td>
<td>Numerische Mathematik</td>
<td>1959</td>
<td>10.1007/BF01386390</td>
<td></td>
</tr>
<tr>
<td>KS67</td>
<td>John G. Kemeny and James Laurie Snell</td>
<td>Finite markov chains</td>
<td>Van Nostrand</td>
<td>1967</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Index

A

adapt_bisquare() (in module pysal.model.mgwr.kernels), 316
adapt_exp() (in module pysal.model.mgwr.kernels), 316
adapt_gauss() (in module pysal.model.mgwr.kernels), 316
allneighbordistances() (in module pysal.explore.spaghetti.Network), 149
alpha_shape() (in module pysal.lib.cg), 58
alpha_shape_auto() (in module pysal.lib.cg), 58
arcdist() (in module pysal.lib.cg), 60
arcdist2linear() (in module pysal.lib.cg), 61
asShape() (in module pysal.lib.cg), 70
attach_islands() (in module pysal.lib.weights), 44
available() (in module pysal.lib.examples), 86

B

bbcommon() (in module pysal.lib.cg), 71
block_weights() (in module pysal.lib.weights), 33
Box_Plot (class in module pysal.viz.mapclassify), 170
brute_knn() (in module pysal.lib.cg), 61
build_lattice_shapefile() (in module pysal.lib.weights), 48