pysal Documentation

Release 1.13.0

PySAL Developers

Mar 15, 2017

Contents

Releases

- Stable 1.13.0 (Released 2016-12-9)
- Development

PySAL is an open source library of spatial analysis functions written in Python intended to support the development of high level applications. PySAL is open source under the BSD License.

CHAPTER 1

User Guide

Introduction

Contents

History

PySAL grew out of a collaborative effort between Luc Anselin's group previously located at the University of Illinois, Champaign-Urbana, and Serge Rey who was at San Diego State University. It was born out of a recognition that the respective projects at the two institutions, PySpace (now GeoDaSpace) and STARS - Space Time Analysis of Regional Systems, could benefit from a shared analytical core, since this would limit code duplication and free up additional developer time to focus on enhancements of the respective applications.

This recognition also came at a time when Python was starting to make major inroads in geographic information systems as represented by projects such as the Python Cartographic Library, Shapely and ESRI's adoption of Python as a scripting language, among others. At the same time there was a dearth of Python modules for spatial statistics, spatial econometrics, location modeling and other areas of spatial analysis, and the role for PySAL was then expanded beyond its support of STARS and GeoDaSpace to provide a library of core spatial analytical functions that could support the next generation of spatial analysis applications.

In 2008 the home for PySAL moved to the GeoDa Center for Geospatial Analysis and Computation at Arizona State University.

Scope

It is important to underscore what PySAL is, and is not, designed to do. First and foremost, PySAL is a **library** in the fullest sense of the word. Developers looking for a suite of spatial analytical methods that they can incorporate into application development should feel at home using PySAL. Spatial analysts who may be carrying out research projects requiring customized scripting, extensive simulation analysis, or those seeking to advance the state of the art in spatial analysis should also find PySAL to be a useful foundation for their work.

End users looking for a user friendly graphical user interface for spatial analysis should not turn to PySAL directly. Instead, we would direct them to projects like STARS and the GeoDaX suite of software products which wrap PySAL functionality in GUIs. At the same time, we expect that with developments such as the Python based plug-in architectures for QGIS, GRASS, and the toolbox extensions for ArcGIS, that end user access to PySAL functionality will be widening in the near future.

Research Papers and Presentations

- Rey, Sergio J. (2012) PySAL: A Python Library for Exploratory Spatial Data Analysis and Geocomputation (Movie) SciPy 2012.
- Rey, Sergio J. and Luc Anselin. (2010) PySAL: A Python Library of Spatial Analytical Methods. In M. Fischer and A. Getis (eds.) Handbook of Applied Spatial Analysis: Software Tools, Methods and Applications. Springer, Berlin.
- Rey, Sergio J. and Luc Anselin. (2009) PySAL: A Python Library for Spatial Analysis and Geocomputation. (Movie) Python for Scientific Computing. Caltech, Pasadena, CA August 2009.
- Rey, Sergio J. (2009). Show Me the Code: Spatial Analysis and Open Source. *Journal of Geographical Systems* 11: 191-2007.
- Rey, S.J., Anselin, L., & M. Hwang. (2008). Dynamic Manipulation of Spatial Weights Using Web Services. GeoDa Center Working Paper 2008-12.

Install PySAL

Windows users can download an .exe installer here on Sourceforge.

PySAL is built upon the Python scientific stack including numpy and scipy. While these libraries are packaged for several platforms, the Anaconda and Enthought Python distributions include them along with the core Python library.

- Anaconda Python distribution
- Enthought Canopy

Note that while both Anaconda and Enthought Canopy will satisfy the dependencies for PySAL, the version of PySAL included in these distributions might be behind the latest stable release of PySAL. You can update to the latest stable version of PySAL with either of these distributions as follows:

1. In a terminal start the python version associated with the distribution. Make sure you are not using a different (system) version of Python. To check this use *which python* from a terminal to see if Anaconda or Enthought appear in the output.
2. *pip install -U pysal*

If you do not wish to use either Anaconda or Enthought, ensure the following software packages are available on your machine:

- Python 2.6, 2.7 or 3.4

- numpy 1.3 or later
- scipy 0.11 or later

Getting your feet wet

You can start using PySAL right away on the web with Wakari, PythonAnywhere, or SageMathCloud.

wakari http://continuum.io/wakari

PythonAnywhere https://www.pythonanywhere.com/

SageMathCloud https://cloud.sagemath.com/

Download and install

PySAL is available on the Python Package Index, which means it can be downloaded and installed manually or from the command line using *pip*, as follows:

```
pip install pysal
```

Alternatively, grab the source distribution (.tar.gz) and decompress it to your selected destination. Open a command shell and navigate to the decompressed pysal folder. Type:

```
pip install .
```

Development version on GitHub

Developers can checkout PySAL using **git**:

```
git clone https://github.com/pysal/pysal.git
```

Open a command shell and navigate to the cloned pysal directory. Type:

```
pip install -e .[dev]
```

The '-e' builds the modules in place and symlinks from the python site packages directory to the pysal folder. The advantage of this method is that you get the latest code but don't have to fuss with editing system environment variables.

To test your setup, start a Python session and type:

```
>>> import pysal
```

Keep up to date with pysal development by 'pulling' the latest changes:

```
git pull
```

Windows

To keep up to date with PySAL development, you will need a Git client that allows you to access and update the code from our repository. We recommend GitHub Windows for a more graphical client, or Git Bash for a command line client. This one gives you a nice Unix-like shell with familiar commands. Here is a nice tutorial on getting going with Open Source software on Windows.

After cloning pysal, install it in develop mode so Python knows where to find it.

Open a command shell and navigate to the cloned pysal directory. Type:

```
pip install -e .[dev]
```

To test your setup, start a Python session and type:

```
>>> import pysal
```

Keep up to date with pysal development by 'pulling' the latest changes:

```
git pull
```

Troubleshooting

If you experience problems when building, installing, or testing pysal, ask for help on the OpenSpace list or browse the archives of the pysal-dev google group.

Please include the output of the following commands in your message:

1. Platform information:

   ```
   python -c 'import os,sys;print os.name, sys.platform'
   uname -a
   ```

2. Python version:

   ```
   python -c 'import sys; print sys.version'
   ```

3. SciPy version:

   ```
   python -c 'import scipy; print scipy.__version__'
   ```

3. NumPy version:

   ```
   python -c 'import numpy; print numpy.__version__'
   ```

4. Feel free to add any other relevant information. For example, the full output (both stdout and stderr) of the pysal installation command can be very helpful. Since this output can be rather large, ask before sending it into the mailing list (or better yet, to one of the developers, if asked).

Getting Started with PySAL

Introduction to the Tutorials

Assumptions

The tutorials presented here are designed to illustrate a selection of the functionality in PySAL. Further details on PySAL functionality not covered in these tutorials can be found in the *API*. The reader is **assumed to have working knowledge of the particular spatial analytical methods** illustrated. Background on spatial analysis can be found in the references cited in the tutorials.

It is also assumed that the reader has already *installed PySAL*.

Examples

The examples use several sample data sets that are included in the pysal/examples directory. In the examples that follow, we refer to those using the path:

```
../pysal/examples/filename_of_example
```

You may need to adjust this path to match the location of the sample files on your system.

Getting Help

Help for PySAL is available from a number of sources.

email lists

The main channel for user support is the openspace mailing list.

Questions regarding the development of PySAL should be directed to pysal-dev.

Documentation

Documentation is available on-line at pysal.org.

You can also obtain help at the interpreter:

```
>>> import pysal
>>> help(pysal)
```

which would bring up help on PySAL:

```
Help on package pysal:

NAME
    pysal

FILE
    /Users/serge/Dropbox/pysal/src/trunk/pysal/__init__.py

DESCRIPTION
    Python Spatial Analysis Library
    ===============================


    Documentation
    -------------
    PySAL documentation is available in two forms: python docstrings and a html␣
↪webpage at http://pysal.org/

    Available sub-packages
    ----------------------

    cg
:
```

Note that you can use this on any option within PySAL:

```
>>> w=pysal.lat2W()
>>> help(w)
```

which brings up:

```
Help on W in module pysal.weights object:

class W(__builtin__.object)
 |  Spatial weights
 |
 |  Parameters
 |  ----------
 |  neighbors       : dictionary
 |                    key is region ID, value is a list of neighbor IDS
 |                    Example:  {'a':['b'],'b':['a','c'],'c':['b']}
 |  weights = None  : dictionary
 |                    key is region ID, value is a list of edge weights
 |                    If not supplied all edge wegiths are assumed to have a weight␣
↪of 1.
 |                    Example: {'a':[0.5],'b':[0.5,1.5],'c':[1.5]}
 |  id_order = None : 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.
 |
```

Note that the help is truncated at the bottom of the terminal window and more of the contents can be seen by scrolling (hit any key).

An Overview of the FileIO system in PySAL.

Contents

Introduction

PySAL contains a file input-output API that stands as a reference pure python implementation for spatial IO. The design goal for this API is to abstract file handling and return native PySAL data types when reading from known file types. A list of known extensions can be found by issuing the following command:

```
pysal.open.check()
```

Note that in some cases the FileIO module will peek inside your file to determine its type. For example "geoda_txt" is just a unique scheme for ".txt" files, so when opening a ".txt" pysal will peek inside the file to determine it if has the necessary header information and dispatch accordingly. In the event that pysal does not understand your file IO operations will be dispatched to python's internal open.

PySAL can also fully leverage Geopandas in analyses. It also provides an alternative, tabular data IO system, `pdio`.

Examples: Reading files

Shapefiles

```
>>> import pysal
>>> shp = pysal.open('../pysal/examples/10740.shp')
>>> poly = shp.next()
>>> type(poly)
<class 'pysal.cg.shapes.Polygon'>
>>> len(shp)
195
>>> shp.get(len(shp)-1).id
195
>>> polys = list(shp)
>>> len(polys)
195
```

DBF Files

```
>>> import pysal
>>> db = pysal.open('../pysal/examples/10740.dbf','r')
>>> db.header
['GIST_ID', 'FIPSSTCO', 'TRT2000', 'STFID', 'TRACTID']
>>> db.field_spec
[('N', 8, 0), ('C', 5, 0), ('C', 6, 0), ('C', 11, 0), ('C', 10, 0)]
>>> db.next()
[1, '35001', '000107', '35001000107', '1.07']
>>> db[0]
[[1, '35001', '000107', '35001000107', '1.07']]
>>> db[0:3]
[[1, '35001', '000107', '35001000107', '1.07'], [2, '35001', '000108', '35001000108',
↪'1.08'], [3, '35001', '000109', '35001000109', '1.09']]
>>> db[0:5,1]
['35001', '35001', '35001', '35001', '35001']
>>> db[0:5,0:2]
[[1, '35001'], [2, '35001'], [3, '35001'], [4, '35001'], [5, '35001']]
>>> db[-1,-1]
['9712']
```

CSV Files

```
>>> import pysal
>>> db = pysal.open('../pysal/examples/stl_hom.csv')
>>> db.header
['WKT', 'NAME', 'STATE_NAME', 'STATE_FIPS', 'CNTY_FIPS', 'FIPS', 'FIPSNO', 'HR7984',
↪'HR8488', 'HR8893', 'HC7984', 'HC8488', 'HC8893', 'PO7984', 'PO8488', 'PO8893',
↪'PE77', 'PE82', 'PE87', 'RDAC80', 'RDAC85', 'RDAC90']
>>> db[0]
[['POLYGON ((-89.585220336914062 39.978794097900391,-89.581146240234375 40.
↪094867706298828,-89.603988647460938 40.095306396484375,-89.60589599609375 40.
↪136119842529297,-89.6103515625 40.3251953125,-89.269027709960938 40.329566955566406,
↪-89.268562316894531 40.285579681396484,-89.154655456542969 40.285774230957031,-89.
↪152763366699219 40.054969787597656,-89.151618957519531 39.919403076171875,-89.
↪224777221679688 39.918678283691406,-89.411857604980469 39.918041229248047,-89.
↪412437438964844 39.931644439697266,-89.495201110839844 39.933486938476562,-89.
↪4927978515625 39.980186462402344,-89.585220336914062 39.978794097900391))', 'Logan',
↪ 'Illinois', 17, 107, 17107, 17107, 2.115428, 1.290722, 1.624458, 4, 2, 3, 189087,␣
↪154952, 184677, 5.10432, 6.59578, 5.832951, -0.991256, -0.940265, -0.845005]]
```

```
>>> fromWKT = pysal.core.util.wkt.WKTParser()
>>> db.cast('WKT',fromWKT)
>>> type(db[0][0][0])
<class 'pysal.cg.shapes.Polygon'>
>>> db[0][0][1:]
['Logan', 'Illinois', 17, 107, 17107, 17107, 2.115428, 1.290722, 1.624458, 4, 2, 3,␣
↪189087, 154952, 184677, 5.10432, 6.59578, 5.832951, -0.991256, -0.940265, -0.845005]
>>> polys = db.by_col('WKT')
>>> from pysal.cg import standalone
>>> standalone.get_bounding_box(polys)[:]
[-92.70067596435547, 36.88180923461914, -87.91657257080078, 40.329566955566406]
```

WKT Files

```
>>> import pysal
>>> wkt = pysal.open('../pysal/examples/stl_hom.wkt', 'r')
>>> polys = wkt.read()
>>> wkt.close()
>>> print len(polys)
78
>>> print polys[1].centroid
(-91.19578469430738, 39.990883050220845)
```

GeoDa Text Files

```
>>> import pysal
>>> geoda_txt = pysal.open('../pysal/examples/stl_hom.txt', 'r')
>>> geoda_txt.header
['FIPSNO', 'HR8488', 'HR8893', 'HC8488']
>>> print len(geoda_txt)
78
>>> geoda_txt.dat[0]
['17107', '1.290722', '1.624458', '2']
>>> geoda_txt._spec
[<type 'int'>, <type 'float'>, <type 'float'>, <type 'int'>]
>>> geoda_txt.close()
```

GAL Binary Weights Files

```
>>> import pysal
>>> gal = pysal.open('../pysal/examples/sids2.gal','r')
>>> w = gal.read()
>>> gal.close()
>>> w.n
100
```

GWT Weights Files

```
>>> import pysal
>>> gwt = pysal.open('../pysal/examples/juvenile.gwt', 'r')
>>> w = gwt.read()
>>> gwt.close()
>>> w.n
168
```

ArcGIS Text Weights Files

```
>>> import pysal
>>> arcgis_txt = pysal.open('../pysal/examples/arcgis_txt.txt','r','arcgis_text')
>>> w = arcgis_txt.read()
>>> arcgis_txt.close()
>>> w.n
3
```

ArcGIS DBF Weights Files

```
>>> import pysal
>>> arcgis_dbf = pysal.open('../pysal/examples/arcgis_ohio.dbf','r','arcgis_dbf')
>>> w = arcgis_dbf.read()
>>> arcgis_dbf.close()
>>> w.n
88
```

ArcGIS SWM Weights Files

```
>>> import pysal
>>> arcgis_swm = pysal.open('../pysal/examples/ohio.swm','r')
>>> w = arcgis_swm.read()
>>> arcgis_swm.close()
>>> w.n
88
```

DAT Weights Files

```
>>> import pysal
>>> dat = pysal.open('../pysal/examples/wmat.dat','r')
>>> w = dat.read()
>>> dat.close()
>>> w.n
49
```

MATLAB MAT Weights Files

```
>>> import pysal
>>> mat = pysal.open('../pysal/examples/spat-sym-us.mat','r')
>>> w = mat.read()
>>> mat.close()
>>> w.n
46
```

LOTUS WK1 Weights Files

```
>>> import pysal
>>> wk1 = pysal.open('../pysal/examples/spat-sym-us.wk1','r')
>>> w = wk1.read()
>>> wk1.close()
>>> w.n
46
```

GeoBUGS Text Weights Files

```
>>> import pysal
>>> geobugs_txt = pysal.open('../pysal/examples/geobugs_scot','r','geobugs_text')
>>> w = geobugs_txt.read()
WARNING: there are 3 disconnected observations
Island ids:  [6, 8, 11]
>>> geobugs_txt.close()
>>> w.n
56
```

STATA Text Weights Files

```
>>> import pysal
>>> stata_txt = pysal.open('../pysal/examples/stata_sparse.txt','r','stata_text')
>>> w = stata_txt.read()
WARNING: there are 7 disconnected observations
Island ids:  [5, 9, 10, 11, 12, 14, 15]
>>> stata_txt.close()
>>> w.n
56
```

MatrixMarket MTX Weights Files

This file format or its variant is currently under consideration of the PySAL team to store general spatial weights in a sparse matrix form.

```
>>> import pysal
>>> mtx = pysal.open('../pysal/examples/wmat.mtx','r')
>>> w = mtx.read()
>>> mtx.close()
>>> w.n
49
```

Examples: Writing files

GAL Binary Weights Files

```
>>> import pysal
>>> w = pysal.queen_from_shapefile('../pysal/examples/virginia.shp',idVariable='FIPS')
>>> w.n
136
>>> gal = pysal.open('../pysal/examples/virginia_queen.gal','w')
>>> gal.write(w)
>>> gal.close()
```

GWT Weights Files

Currently, it is not allowed to write a GWT file.

ArcGIS Text Weights Files

```
>>> import pysal
>>> w = pysal.queen_from_shapefile('../pysal/examples/virginia.shp',idVariable='FIPS')
>>> w.n
136
>>> arcgis_txt = pysal.open('../pysal/examples/virginia_queen.txt','w','arcgis_text')
>>> arcgis_txt.write(w, useIdIndex=True)
>>> arcgis_txt.close()
```

ArcGIS DBF Weights Files

```
>>> import pysal
>>> w = pysal.queen_from_shapefile('../pysal/examples/virginia.shp',idVariable='FIPS')
>>> w.n
136
>>> arcgis_dbf = pysal.open('../pysal/examples/virginia_queen.dbf','w','arcgis_dbf')
>>> arcgis_dbf.write(w, useIdIndex=True)
>>> arcgis_dbf.close()
```

ArcGIS SWM Weights Files

```
>>> import pysal
>>> w = pysal.queen_from_shapefile('../pysal/examples/virginia.shp',idVariable='FIPS')
>>> w.n
136
>>> arcgis_swm = pysal.open('../pysal/examples/virginia_queen.swm','w')
>>> arcgis_swm.write(w, useIdIndex=True)
>>> arcgis_swm.close()
```

DAT Weights Files

```
>>> import pysal
>>> w = pysal.queen_from_shapefile('../pysal/examples/virginia.shp',idVariable='FIPS')
>>> w.n
136
>>> dat = pysal.open('../pysal/examples/virginia_queen.dat','w')
>>> dat.write(w)
>>> dat.close()
```

MATLAB MAT Weights Files

```
>>> import pysal
>>> w = pysal.queen_from_shapefile('../pysal/examples/virginia.shp',idVariable='FIPS')
>>> w.n
136
>>> mat = pysal.open('../pysal/examples/virginia_queen.mat','w')
>>> mat.write(w)
>>> mat.close()
```

LOTUS WK1 Weights Files

```
>>> import pysal
>>> w = pysal.queen_from_shapefile('../pysal/examples/virginia.shp',idVariable='FIPS')
>>> w.n
136
>>> wk1 = pysal.open('../pysal/examples/virginia_queen.wk1','w')
>>> wk1.write(w)
>>> wk1.close()
```

GeoBUGS Text Weights Files

```
>>> import pysal
>>> w = pysal.queen_from_shapefile('../pysal/examples/virginia.shp',idVariable='FIPS')
>>> w.n
136
>>> geobugs_txt = pysal.open('../pysal/examples/virginia_queen','w','geobugs_text')
>>> geobugs_txt.write(w)
>>> geobugs_txt.close()
```

STATA Text Weights Files

```
>>> import pysal
>>> w = pysal.queen_from_shapefile('../pysal/examples/virginia.shp',idVariable='FIPS')
>>> w.n
136
>>> stata_txt = pysal.open('../pysal/examples/virginia_queen.txt','w','stata_text')
>>> stata_txt.write(w,matrix_form=True)
>>> stata_txt.close()
```

MatrixMarket MTX Weights Files

```
>>> import pysal
>>> w = pysal.queen_from_shapefile('../pysal/examples/virginia.shp',idVariable='FIPS')
>>> w.n
136
>>> mtx = pysal.open('../pysal/examples/virginia_queen.mtx','w')
>>> mtx.write(w)
>>> mtx.close()
```

Examples: Converting the format of spatial weights files

PySAL provides a utility tool to convert a weights file from one format to another.

From GAL to ArcGIS SWM format

```
>>> from pysal.core.util.weight_converter import weight_convert
>>> gal_file = '../pysal/examples/sids2.gal'
>>> swm_file = '../pysal/examples/sids2.swm'
>>> weight_convert(gal_file, swm_file, useIdIndex=True)
>>> wold = pysal.open(gal_file, 'r').read()
>>> wnew = pysal.open(swm_file, 'r').read()
>>> wold.n == wnew.n
True
```

For further details see the *FileIO API*.

Alternative Tabular API

For shapefile input and output, the dataframe API constructs a dataframe similar to that constructed by Geopandas, but populated by PySAL's own shape classes. This is provided as a convenience method for users who have shapefile-heavy workflows and would like to get Geopandas-style interaction. This API is only a frontend to the existing PySAL api documented above, and users who have heavier spatial data needs may find Geopandas useful.

```
```

Spatial Weights

Contents

Introduction

Spatial weights are central components of many areas of spatial analysis. In general terms, for a spatial data set composed of n locations (points, areal units, network edges, etc.), the spatial weights matrix expresses the potential for interaction between observations at each pair i,j of locations. There is a rich variety of ways to specify the structure of these weights, and PySAL supports the creation, manipulation and analysis of spatial weights matrices across three different general types:

- Contiguity Based Weights
- Distance Based Weights
- Kernel Weights

These different types of weights are implemented as instances or subclasses of the PySAL weights class `W`.

In what follows, we provide a high level overview of spatial weights in PySAL, starting with the three different types of weights, followed by a closer look at the properties of the W class and some related functions.[1]

PySAL Spatial Weight Types

PySAL weights are handled in objects of the `pysal.weights.W`. The conceptual idea of spatial weights is that of a nxn matrix in which the diagonal elements (w_{ii}) are set to zero by definition and the rest of the cells (w_{ij}) capture

[1] Although this tutorial provides an introduction to the functionality of the PySAL weights class, it is not exhaustive. Complete documentation for the class and associated functions can be found by accessing the help from within a Python interpreter.

the potential of interaction. However, these matrices tend to be fairly sparse (i.e. many cells contain zeros) and hence a full nxn array would not be an efficient representation. PySAL employs a different way of storing that is structured in two main dictionaries[2] : neighbors which, for each observation (key) contains a list of the other ones (value) with potential for interaction ($w_{ij} \neq 0$); and weights, which contains the weight values for each of those observations (in the same order). This way, large datasets can be stored when keeping the full matrix would not be possible because of memory constraints. In addition to the sparse representation via the weights and neighbors dictionaries, a PySAL W object also has an attribute called sparse, which is a scipy.sparse CSR representation of the spatial weights. (See *WSP* for an alternative PySAL weights object.)

Contiguity Based Weights

To illustrate the general weights object, we start with a simple contiguity matrix constructed for a 5 by 5 lattice (composed of 25 spatial units):

```
>>> import pysal
>>> w = pysal.lat2W(5, 5)
```

The w object has a number of attributes:

```
>>> w.n
25
>>> w.pct_nonzero
12.8
>>> w.weights[0]
[1.0, 1.0]
>>> w.neighbors[0]
[5, 1]
>>> w.neighbors[5]
[0, 10, 6]
>>> w.histogram
[(2, 4), (3, 12), (4, 9)]
```

n is the number of spatial units, so conceptually we could be thinking that the weights are stored in a 25x25 matrix. The second attribute (pct_nonzero) shows the sparseness of the matrix. The key attributes used to store contiguity relations in W are the neighbors and weights attributes. In the example above we see that the observation with id 0 (Python is zero-offset) has two neighbors with ids [5, 1] each of which have equal weights of 1.0.

The histogram attribute is a set of tuples indicating the cardinality of the neighbor relations. In this case we have a regular lattice, so there are 4 units that have 2 neighbors (corner cells), 12 units with 3 neighbors (edge cells), and 9 units with 4 neighbors (internal cells).

In the above example, the default criterion for contiguity on the lattice was that of the rook which takes as neighbors any pair of cells that share an edge. Alternatively, we could have used the queen criterion to include the vertices of the lattice to define contiguities:

```
>>> wq = pysal.lat2W(rook = False)
>>> wq.neighbors[0]
[5, 1, 6]
```

The bishop criterion, which designates pairs of cells as neighbors if they share only a vertex, is yet a third alternative for contiguity weights. A bishop matrix can be computed as the *Difference* between the rook and queen cases.

The lat2W function is particularly useful in setting up simulation experiments requiring a regular grid. For empirical research, a common use case is to have a shapefile, which is a nontopological vector data structure, and a need to carry

[2] The dictionaries for the weights and value attributes in W are read-only.

out some form of spatial analysis that requires spatial weights. Since topology is not stored in the underlying file there is a need to construct the spatial weights prior to carrying out the analysis.

In PySAL, weights are constructed by default from any contiguity graph representation. Most users will find the `.from_shapefile` methods most useful:

```
>>> w = pysal.weights.Rook.from_shapefile("../pysal/examples/columbus.shp")
>>> w.n
49
>>> print "%.4f"%w.pct_nonzero
0.0833
>>> w.histogram
[(2, 7), (3, 10), (4, 17), (5, 8), (6, 3), (7, 3), (8, 0), (9, 1)]
```

If queen, rather than rook, contiguity is required then the following would work:

```
>>> w = pysal.weights.Queen.from_shapefile("../pysal/examples/columbus.shp")
>>> print "%.4f"%w.pct_nonzero
0.0983
>>> w.histogram
[(2, 5), (3, 9), (4, 12), (5, 5), (6, 9), (7, 3), (8, 4), (9, 1), (10, 1)]
```

In addition to these methods, contiguity weights can be built from dataframes with a geometry column. This includes dataframes built from geopandas or from the PySAL pandas IO extension, pdio. For instance:

```
>>> import geopandas as gpd
>>> test = gpd.read_file(pysal.examples.get_path('south.shp'))
>>> W = pysal.weights.Queen.from_dataframe(test)
>>> Wrook = pysal.weights.Rook.from_dataframe(test, idVariable='CNTY_FIPS')
>>> pdiodf = pysal.pdio.read_files(pysal.examples.get_path('south.shp'))
>>> W = pysal.weights.Rook.from_dataframe(pdiodf)
```

Or, weights can be constructed directly from an interable of shapely objects:

```
>>> import geopandas as gpd
>>> shapelist = gpd.read_file(pysal.examples.get_path('columbus.shp')).geometry.
↪tolist()
>>> W = pysal.weights.Queen.from_iterable(shapelist)
```

The `.from_file` method on contigutiy weights simply passes down to the parent class's `.from_file` method, so the returned object is of instance `W`, not `Queen` or `Rook`. This occurs because the weights cannot be verified *as* contiguity weights without the original shapes.

```
>>> W = pysal.weights.Rook.from_file(pysal.examples.get_path('columbus.gal')
>>> type(W)
pysal.weights.weights.W
```

Distance Based Weights

In addition to using contiguity to define neighbor relations, more general functions of the distance separating observations can be used to specify the weights.

Please note that distance calculations are coded for a flat surface, so you will need to have your shapefile projected in advance for the output to be correct.

k-nearest neighbor weights

The neighbors for a given observations can be defined using a k-nearest neighbor criterion. For example we could use the the centroids of our 5x5 lattice as point locations to measure the distances. First, we import numpy to create the coordinates as a 25x2 numpy array named `data`:

```
>>> import numpy as np
>>> x,y=np.indices((5,5))
>>> x.shape=(25,1)
>>> y.shape=(25,1)
>>> data=np.hstack([x,y])
```

then define the KNN weight as:

```
>>> wknn3 = pysal.weights.KNN(data, k = 3)
>>> wknn3.neighbors[0]
[1, 5, 6]
>>> wknn3.s0
75.0
```

For efficiency, a KDTree is constructed to compute efficient nearest neighbor queries. To construct many K-Nearest neighbor weights from the same data, a convenience method is provided that prevents re-constructing the KDTree while letting the user change aspects of the weight object. By default, the reweight method operates in place:

```
>>> w4 = wknn3.reweight(k=4, inplace=False)
>>> w4.neighbors[0]
[1,5,6,2]
>>> l1norm = wknn3.reweight(p=1, inplace=False)
>>> l1norm.neighbors
[1,5,2]
>>> set(w4.neighbors[0]) == set([1, 5, 6, 2])
True
>>> w4.s0
100.0
>>> w4.weights[0]
[1.0, 1.0, 1.0, 1.0]
```

Alternatively, we can use a utility function to build a knn W straight from a shapefile:

```
>>> wknn5 = pysal.weights.KNN.from_shapefile(pysal.examples.get_path('columbus.shp'),␣
↪k=5)
>>> wknn5.neighbors[0]
[2, 1, 3, 7, 4]
```

Or from a dataframe:

```
>>> import geopandas as gpd
>>> df = gpd.read_file(ps.examples.get_path('baltim.shp'))
>>> k5 = pysal.weights.KNN.from_dataframe(df, k=5)
```

Distance band weights

Knn weights ensure that all observations have the same number of neighbors.[3] An alternative distance based set of weights relies on distance bands or thresholds to define the neighbor set for each spatial unit as those other units falling

[3] Ties at the k-nn distance band are randomly broken to ensure each observation has exactly k neighbors.

within a threshold distance of the focal unit:

```
>>> wthresh = pysal.weights.DistanceBand.from_array(data, 2)
>>> set(wthresh.neighbors[0]) == set([1, 2, 5, 6, 10])
True
>>> set(wthresh.neighbors[1]) == set( [0, 2, 5, 6, 7, 11, 3])
True
>>> wthresh.weights[0]
[1, 1, 1, 1, 1]
>>> wthresh.weights[1]
[1, 1, 1, 1, 1, 1, 1]
>>>
```

As can be seen in the above example, the number of neighbors is likely to vary across observations with distance band weights in contrast to what holds for knn weights.

In addition to constructing these from the helper function, Distance Band weights. For example, a threshold binary W can be constructed from a dataframe:

```
>>> import geopandas as gpd
>>> df = gpd.read_file(ps.examples.get_path('baltim.shp'))
>>> ps.weights.DistanceBand.from_dataframe(df, threshold=6, binary=True)
```

Distance band weights can be generated for shapefiles as well as arrays of points.[4] First, the minimum nearest neighbor distance should be determined so that each unit is assured of at least one neighbor:

```
>>> thresh = pysal.min_threshold_dist_from_shapefile("../pysal/examples/columbus.shp")
>>> thresh
0.61886415807685413
```

with this threshold in hand, the distance band weights are obtained as:

```
>>> wt = pysal.weights.DistanceBand.from_shapefile("../pysal/examples/columbus.shp",␣
↪threshold=thresh, binary=True)
>>> wt.min_neighbors
1
>>> wt.histogram
[(1, 4), (2, 8), (3, 6), (4, 2), (5, 5), (6, 8), (7, 6), (8, 2), (9, 6), (10, 1), (11,
↪ 1)]
>>> set(wt.neighbors[0]) == set([1,2])
True
>>> set(wt.neighbors[1]) == set([3,0])
True
```

Distance band weights can also be specified to take on continuous values rather than binary, with the values set to the inverse distance separating each pair within a given threshold distance. We illustrate this with a small set of 6 points:

```
>>> points = [(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
>>> wid = pysal.weights.DistanceBand.from_array(points,14.2,binary=False)
>>> wid.weights[0]
[0.10000000000000001, 0.089442719099991588]
```

If we change the distance decay exponent to -2.0 the result is so called gravity weights:

[4] If the shapefile contains geographical coordinates these distance calculations will be misleading and the user should first project their coordinates using a GIS.

```
>>> wid2 = pysal.weights.DistanceBand.from_array(points,14.2,alpha = -2.0,␣
↪binary=False)
>>> wid2.weights[0]
[0.01, 0.0079999999999999984]
```

Kernel Weights

A combination of distance based thresholds together with continuously valued weights is supported through kernel weights:

```
>>> points = [(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
>>> kw = pysal.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],
       [ 20.000002]])
```

The bandwidth attribute plays the role of the distance threshold with kernel weights, while the form of the kernel function determines the distance decay in the derived continuous weights (the following are available: 'triangular','uniform','quadratic','epanechnikov','quartic','bisquare','gaussian'). In the above example, the bandwidth is set to the default value and fixed across the observations. The user could specify a different value for a fixed bandwidth:

```
>>> kw15 = pysal.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.],
       [ 15.]])
```

which results in fewer neighbors for the first unit. Adaptive bandwidths (i.e., different bandwidths for each unit) can also be user specified:

```
>>> bw = [25.0,15.0,25.0,16.0,14.5,25.0]
>>> kwa = pysal.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. ]])
```

Alternatively the adaptive bandwidths could be defined endogenously:

```
>>> kwea = pysal.Kernel(points,fixed = False)
>>> kwea.weights[0]
[1.0, 0.10557289844279438, 9.99999900663795e-08]
>>> kwea.neighbors[0]
[0, 1, 3]
>>> kwea.bandwidth
array([[ 11.18034101],
       [ 11.18034101],
       [ 20.000002  ],
       [ 11.18034101],
       [ 14.14213704],
       [ 18.02775818]])
```

Finally, the kernel function could be changed (with endogenous adaptive bandwidths):

```
>>> kweag = pysal.Kernel(points,fixed = False,function = 'gaussian')
>>> kweag.weights[0]
[0.3989422804014327, 0.2674190291577696, 0.2419707487162134]
>>> kweag.bandwidth
array([[ 11.18034101],
       [ 11.18034101],
       [ 20.000002  ],
       [ 11.18034101],
       [ 14.14213704],
       [ 18.02775818]])
```

More details on kernel weights can be found in *Kernel*. All kernel methods also support construction from shapefiles with `Kernel.from_shapefile` and from dataframes with `Kernel.from_dataframe`.

Weights from other python objects

PySAL weights can also be constructed easily from many other objects. Most importantly, all weight types can be constructed directly from `geopandas` geodataframes using the `.from_dataframe` method. For distance and kernel weights, underlying features should typically be points. But, if polygons are supplied, the centroids of the polygons will be used by default:

```
>>> import geopandas as gpd
>>> df = gpd.read_file(pysal.examples.get_path('columbus.shp'))
>>> kw = pysal.weights.Kernel.from_dataframe(df)
>>> dbb = pysal.weights.DistanceBand.from_dataframe(df, threshold=.9, binary=False)
>>> dbc = pysal.weights.DistanceBand.from_dataframe(df, threshold=.9, binary=True)
>>> q = pysal.weights.Queen.from_dataframe(df)
>>> r = pysal.weights.Rook.from_dataframe(df)
```

This also applies to dynamic views of the dataframe:

```
>>> q2 = pysal.weights.Queen.from_dataframe(df.query('DISCBD < 2'))
```

Weights can also be constructed from NetworkX objects. This is easiest to construct using a sparse weight, but that can be converted to a full dense PySAL weight easily:

```
>>> import networkx as nx
>>> G = nx.random_lobster(50,.2,.5)
>>> sparse_lobster = ps.weights.WSP(nx.adj_matrix(G))
>>> dense_lobster = sparse_lobster.to_W()
```

A Closer look at W

Although the three different types of spatial weights illustrated above cover a wide array of approaches towards specifying spatial relations, they all share common attributes from the base W class in PySAL. Here we take a closer look at some of the more useful properties of this class.

Attributes of W

W objects come with a whole bunch of useful attributes that may help you when dealing with spatial weights matrices. To see a list of all of them, same as with any other Python object, type:

```
>>> import pysal
>>> help(pysal.W)
```

If you want to be more specific and learn, for example, about the attribute *s0*, then type:

```
>>> help(pysal.W.s0)
Help on property:
```

float

$$s0 = \sum_i \sum_j w_{i,j}$$

Weight Transformations

Often there is a need to apply a transformation to the spatial weights, such as in the case of row standardization. Here each value in the row of the spatial weights matrix is rescaled to sum to one:

$$ws_{i,j} = w_{i,j} / \sum_j w_{i,j}$$

This and other weights transformations in PySAL are supported by the transform property of the W class. To see this let's build a simple contiguity object for the Columbus data set:

```
>>> w = pysal.rook_from_shapefile("../pysal/examples/columbus.shp")
>>> w.weights[0]
[1.0, 1.0]
```

We can row standardize this by setting the transform property:

```
>>> w.transform = 'r'
>>> w.weights[0]
[0.5, 0.5]
```

Supported transformations are the following:

- '*b*': binary.

- '*r*': row standardization.
- '*v*': variance stabilizing.

If the original weights (unstandardized) are required, the transform property can be reset:

```
>>> w.transform = 'o'
>>> w.weights[0]
[1.0, 1.0]
```

Behind the scenes the transform property is updating all other characteristics of the spatial weights that are a function of the values and these standardization operations, freeing the user from having to keep these other attributes updated. To determine the current value of the transformation, simply query this attribute:

```
>>> w.transform
'O'
```

More details on other transformations that are supported in W can be found in `pysal.weights.W`.

W related functions

Generating a full array

As the underlying data structure of the weights in W is based on a sparse representation, there may be a need to work with the full numpy array. This is supported through the full method of W:

```
>>> wf = w.full()
>>> len(wf)
2
```

The first element of the return from w.full is the numpy array:

```
>>> wf[0].shape
(49, 49)
```

while the second element contains the ids for the row (column) ordering of the array:

```
>>> wf[1][0:5]
[0, 1, 2, 3, 4]
```

If only the array is required, a simple Python slice can be used:

```
>>> wf = w.full()[0]
```

Shimbel Matrices

The Shimbel matrix for a set of n objects contains the shortest path distance separating each pair of units. This has wide use in spatial analysis for solving different types of clustering and optimization problems. Using the function *shimbel* with a *W* instance as an argument generates this information:

```
>>> w = pysal.lat2W(3,3)
>>> ws = pysal.shimbel(w)
>>> ws[0]
[-1, 1, 2, 1, 2, 3, 2, 3, 4]
```

Thus we see that observation 0 (the northwest cell of our 3x3 lattice) is a first order neighbor to observations 1 and 3, second order neighbor to observations 2, 4, and 6, a third order neighbor to 5, and 7, and a fourth order neighbor to observation 8 (the extreme southeast cell in the lattice). The position of the -1 simply denotes the focal unit.

Higher Order Contiguity Weights

Closely related to the shortest path distances is the concept of a spatial weight based on a particular order of contiguity. For example, we could define the second order contiguity relations using:

```
>>> w2 = pysal.higher_order(w, 2)
>>> w2.neighbors[0]
[4, 6, 2]
```

or a fourth order set of weights:

```
>>> w4 = pysal.higher_order(w, 4)
WARNING: there are 5 disconnected observations
Island ids:  [1, 3, 4, 5, 7]
>>> w4.neighbors[0]
[8]
```

In both cases a new instance of the W class is returned with the weights and neighbors defined using the particular order of contiguity.

Spatial Lag

The final function related to spatial weights that we illustrate here is used to construct a new variable called the spatial lag. The spatial lag is a function of the attribute values observed at neighboring locations. For example, if we continue with our regular 3x3 lattice and create an attribute variable y:

```
>>> import numpy as np
>>> y = np.arange(w.n)
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

then the spatial lag can be constructed with:

```
>>> yl = pysal.lag_spatial(w,y)
>>> yl
array([  4.,   6.,   6.,  10.,  16.,  14.,  10.,  18.,  12.])
```

Mathematically, the spatial lag is a weighted sum of neighboring attribute values

$$yl_i = \sum_j w_{i,j} y_j$$

In the example above, the weights were binary, based on the rook criterion. If we row standardize our W object first and then recalculate the lag, it takes the form of a weighted average of the neighboring attribute values:

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

One important consideration in calculating the spatial lag is that the ordering of the values in y aligns with the underlying order in W. In cases where the source for your attribute data is different from the one to construct your weights you may need to reorder your y values accordingly. To check if this is the case you can find the order in W as follows:

```
>>> w.id_order
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

In this case the lag_spatial function assumes that the first value in the y attribute corresponds to unit 0 in the lattice (northwest cell), while the last value in y would correspond to unit 8 (southeast cell). In other words, for the value of the spatial lag to be valid the number of elements in y must match w.n and the orderings must be aligned.

Fortunately, for the common use case where both the attribute and weights information come from a shapefile (and its dbf), PySAL handles the alignment automatically:[5]

```
>>> w = pysal.rook_from_shapefile("../pysal/examples/columbus.shp")
>>> f = pysal.open("../pysal/examples/columbus.dbf")
>>> f.header
['AREA', 'PERIMETER', 'COLUMBUS_', 'COLUMBUS_I', 'POLYID', 'NEIG', 'HOVAL', 'INC',
↪'CRIME', 'OPEN', 'PLUMB', 'DISCBD', 'X', 'Y', 'NSA', 'NSB', 'EW', 'CP', 'THOUS',
↪'NEIGNO']
>>> y = np.array(f.by_col['INC'])
>>> w.transform = 'r'
>>> y
array([ 19.531   ,  21.232   ,  15.956   ,   4.477   ,  11.252   ,
        16.028999,   8.438   ,  11.337   ,  17.586   ,  13.598   ,
         7.467   ,  10.048   ,   9.549   ,   9.963   ,   9.873   ,
         7.625   ,   9.798   ,  13.185   ,  11.618   ,  31.07    ,
        10.655   ,  11.709   ,  21.155001,  14.236   ,   8.461   ,
         8.085   ,  10.822   ,   7.856   ,   8.681   ,  13.906   ,
        16.940001,  18.941999,   9.918   ,  14.948   ,  12.814   ,
        18.739   ,  17.017   ,  11.107   ,  18.476999,  29.833   ,
        22.207001,  25.872999,  13.38    ,  16.961   ,  14.135   ,
        18.323999,  18.950001,  11.813   ,  18.796   ])
>>> yl = pysal.lag_spatial(w,y)
>>> yl
array([ 18.594     ,  13.32133333,  14.123     ,  14.94425   ,
        11.817857  ,  14.419     ,  10.283     ,   8.3364    ,
        11.7576665 ,  19.48466667,  10.0655    ,   9.1882    ,
         9.483     ,  10.07716667,  11.231     ,  10.46185714,
        21.94100033,  10.8605    ,  12.46133333,  15.39877778,
        14.36333333,  15.0838    ,  19.93666633,  10.90833333,
         9.7       ,  11.403     ,  15.13825   ,  10.448     ,
        11.81      ,  12.64725   ,  16.8435    ,  26.0662505 ,
        15.6405    ,  18.05175   ,  15.3824    ,  18.9123996 ,
        12.2418    ,  12.76675   ,  18.5314995 ,  22.79225025,
        22.575     ,  16.8435    ,  14.2066    ,  14.20075   ,
        15.2515    ,  18.6079995 ,  26.0200005 ,  15.818     ,  14.303     ])

>>> w.id_order
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
↪ 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
↪ 45, 46, 47, 48]
```

[5] The ordering exploits the one-to-one relation between a record in the DBF file and the shape in the shapefile.

Non-Zero Diagonal

The typical weights matrix has zeros along the main diagonal. This has the practical result of excluding the self from any computation. However, this is not always the desired situation, and so PySAL offers a function that adds values to the main diagonal of a W object.

As an example, we can build a basic rook weights matrix, which has zeros on the diagonal, then insert ones along the diagonal:

```
>>> w = pysal.lat2W(5, 5, id_type='string')
>>> w['id0']
{'id5': 1.0, 'id1': 1.0}
>>> w_const = pysal.weights.insert_diagonal(w)
>>> w_const['id0']
{'id5': 1.0, 'id0': 1.0, 'id1': 1.0}
```

The default is to add ones to the diagonal, but the function allows any values to be added.

WSets

PySAL offers set-like manipulation of spatial weights matrices. While a W is more complex than a set, the two objects have a number of commonalities allowing for traditional set operations to have similar functionality on a W. Conceptually, we treat each neighbor pair as an element of a set, and then return the appropriate pairs based on the operation invoked (e.g. union, intersection, etc.). A key distinction between a set and a W is that a W must keep track of the universe of possible pairs, even those that do not result in a neighbor relationship.

PySAL follows the naming conventions for Python sets, but adds optional flags allowing the user to control the shape of the weights object returned. At this time, all the functions discussed in this section return a binary W no matter the weights objects passed in.

Union

The union of two weights objects returns a binary weights object, W, that includes all neighbor pairs that exist in either weights object. This function can be used to simply join together two weights objects, say one for Arizona counties and another for California counties. It can also be used to join two weights objects that overlap as in the example below.

```
>>> w1 = pysal.lat2W(4,4)
>>> w2 = pysal.lat2W(6,4)
>>> w = pysal.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]
```

Intersection

The intersection of two weights objects returns a binary weights object, W, that includes only those neighbor pairs that exist in both weights objects. Unlike the union case, where all pairs in either matrix are returned, the intersection only returns a subset of the pairs. This leaves open the question of the shape of the weights matrix to return. For example, you have one weights matrix of census tracts for City A and second matrix of tracts for Utility Company B's service area, and want to find the W for the tracts that overlap. Depending on the research question, you may want the returned W to have the same dimensions as City A's weights matrix, the same as the utility company's weights matrix, a new dimensionality based on all the census tracts in either matrix or with the dimensionality of just those tracts in the overlapping area. All of these options are available via the w_shape parameter and the order that the matrices are passed to the function. The following example uses the all case:

```
>>> w1 = pysal.lat2W(4,4)
>>> w2 = pysal.lat2W(6,4)
>>> w = pysal.w_intersection(w1, w2, 'all')
WARNING: there are 8 disconnected observations
Island ids:  [16, 17, 18, 19, 20, 21, 22, 23]
>>> w1[0] == w[0]
True
>>> w1.neighbors[15]
[11, 14]
>>> w2.neighbors[15]
[11, 14, 19]
>>> w.neighbors[15]
[11, 14]
>>> w2.neighbors[16]
[12, 20, 17]
>>> w.neighbors[16]
[]
```

Difference

The difference of two weights objects returns a binary weights object, W, that includes only neighbor pairs from the first object that are not in the second. Similar to the intersection function, the user must select the shape of the weights object returned using the w_shape parameter. The user must also consider the constrained parameter which controls whether the observations and the neighbor pairs are differenced or just the neighbor pairs are differenced. If you were to apply the difference function to our city and utility company example from the intersection section above, you must decide whether or not pairs that exist along the border of the regions should be considered different or not. It boils down to whether the tracts should be differenced first and then the differenced pairs identified (constrained=True), or if the differenced pairs should be identified based on the sets of pairs in the original weights matrices (constrained=False). In the example below we difference weights matrices from regions with partial overlap.

```
>>> w1 = pysal.lat2W(6,4)
>>> w2 = pysal.lat2W(4,4)
>>> w1.neighbors[15]
[11, 14, 19]
>>> w2.neighbors[15]
[11, 14]
>>> w = pysal.w_difference(w1, w2, w_shape = 'w1', constrained = False)
WARNING: there are 12 disconnected observations
Island ids:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> w.neighbors[15]
[19]
>>> w.neighbors[19]
[15, 18, 23]
>>> w = pysal.w_difference(w1, w2, w_shape = 'min', constrained = False)
>>> 15 in w.neighbors
False
>>> w.neighbors[19]
[18, 23]
```

```
>>> w = pysal.w_difference(w1, w2, w_shape = 'w1', constrained = True)
WARNING: there are 16 disconnected observations
Island ids:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> w.neighbors[15]
[]
>>> w.neighbors[19]
[18, 23]
>>> w = pysal.w_difference(w1, w2, w_shape = 'min', constrained = True)
>>> 15 in w.neighbors
False
>>> w.neighbors[19]
[18, 23]
```

The difference function can be used to construct a bishop *contiguity weights matrix* by differencing a queen and rook matrix.

```
>>> wr = pysal.lat2W(5,5)
>>> wq = pysal.lat2W(5,5,rook = False)
>>> wb = pysal.w_difference(wq, wr,constrained = False)
>>> wb.neighbors[0]
[6]
```

Symmetric Difference

Symmetric difference of two weights objects returns a binary weights object, W, that includes only neighbor pairs that are not shared by either matrix. This function offers options similar to those in the difference function described above.

```
>>> w1 = pysal.lat2W(6, 4)
>>> w2 = pysal.lat2W(2, 4)
>>> w_lower = pysal.w_difference(w1, w2, w_shape = 'min', constrained = True)
>>> w_upper = pysal.lat2W(4, 4)
>>> w = pysal.w_symmetric_difference(w_lower, w_upper, 'all', False)
>>> w_lower.id_order
[8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
>>> w_upper.id_order
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> w.id_order
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
>>> w.neighbors[11]
[7]
>>> w = pysal.w_symmetric_difference(w_lower, w_upper, 'min', False)
WARNING: there are 8 disconnected observations
Island ids:  [0, 1, 2, 3, 4, 5, 6, 7]
>>> 11 in w.neighbors
False
>>> w.id_order
[0, 1, 2, 3, 4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23]
>>> w = pysal.w_symmetric_difference(w_lower, w_upper, 'all', True)
WARNING: there are 16 disconnected observations
Island ids:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> w.neighbors[11]
[]
>>> w = pysal.w_symmetric_difference(w_lower, w_upper, 'min', True)
WARNING: there are 8 disconnected observations
Island ids:  [0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> 11 in w.neighbors
False
```

Subset

Subset of a weights object returns a binary weights object, W, that includes only those observations provided by the user. It also can be used to add islands to a previously existing weights object.

```
>>> w1 = pysal.lat2W(6, 4)
>>> w1.id_order
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
>>> ids = range(16)
>>> w = pysal.w_subset(w1, ids)
>>> w.id_order
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> w1[0] == w[0]
True
>>> w1.neighbors[15]
[11, 14, 19]
>>> w.neighbors[15]
[11, 14]
```

WSP

A thin PySAL weights object is available to users with extremely large weights matrices, on the order of 2 million or more observations, or users interested in holding many large weights matrices in RAM simultaneously. The `pysal.weights.WSP` is a thin weights object that does not include the neighbors and weights dictionaries, but does contain the scipy.sparse form of the weights. For many PySAL functions the W and WSP objects can be used interchangeably.

A WSP object can be constructed from a Matrix Market file (see *MatrixMarket MTX Weights Files* for more info on reading and writing mtx files in PySAL):

```
>>> mtx = pysal.open("../pysal/examples/wmat.mtx", 'r')
>>> wsp = mtx.read(sparse=True)
```

or built directly from a scipy.sparse object:

```
>>> import scipy.sparse
>>> 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 = pysal.weights.WSP(sparse)
```

The WSP object has subset of the attributes of a W object; for example:

```
>>> w.n
4
>>> w.s0
4.0
>>> w.trcWtW_WW
6.3949999999999996
```

The following functionality is available to convert from a W to a WSP:

```
>>> w = pysal.weights.lat2W(5,5)
>>> w.s0
80.0
>>> wsp = pysal.weights.WSP(w.sparse)
>>> wsp.s0
80.0
```

and from a WSP to W:

```
>>> sp = pysal.weights.lat2SW(5, 5)
>>> wsp = pysal.weights.WSP(sp)
>>> wsp.s0
80
>>> w = pysal.weights.WSP2W(wsp)
>>> w.s0
80
```

Further Information

For further details see the *Weights API*.

Spatial Autocorrelation

Contents

Introduction

Spatial autocorrelation pertains to the non-random pattern of attribute values over a set of spatial units. This can take two general forms: positive autocorrelation which reflects value similarity in space, and negative autocorrelation or value dissimilarity in space. In either case the autocorrelation arises when the observed spatial pattern is different from what would be expected under a random process operating in space.

Spatial autocorrelation can be analyzed from two different perspectives. Global autocorrelation analysis involves the study of the entire map pattern and generally asks the question as to whether the pattern displays clustering or not. Local autocorrelation, on the other hand, shifts the focus to explore within the global pattern to identify clusters or so called hot spots that may be either driving the overall clustering pattern, or that reflect heterogeneities that depart from global pattern.

In what follows, we first highlight the global spatial autocorrelation classes in PySAL. This is followed by an illustration of the analysis of local spatial autocorrelation.

Global Autocorrelation

PySAL implements five different tests for global spatial autocorrelation: the Gamma index of spatial autocorrelation, join count statistics, Moran's I, Geary's C, and Getis and Ord's G.

Gamma Index of Spatial Autocorrelation

The Gamma Index of spatial autocorrelation consists of the application of the principle behind a general cross-product statistic to measuring spatial autocorrelation.[1] The idea is to assess whether two similarity matrices for n objects, i.e., n by n matrices A and B measure the same type of similarity. This is reflected in a so-called Gamma Index $\Gamma = \sum_i \sum_j a_{ij}.b_{ij}$. In other words, the statistic consists of the sum over all cross-products of matching elements (i,j) in the two matrices.

The application of this principle to spatial autocorrelation consists of turning the first similarity matrix into a measure of attribute similarity and the second matrix into a measure of locational similarity. Naturally, the second matrix is the a spatial *weight* matrix. The first matrix can be any reasonable measure of attribute similarity or dissimilarity, such as a cross-product, squared difference or absolute difference.

Formally, then, the Gamma index is:

$$\Gamma = \sum_i \sum_j a_{ij}.w_{ij}$$

where the w_{ij} are the elements of the weights matrix and a_{ij} are corresponding measures of attribute similarity.

Inference for this statistic is based on a permutation approach in which the values are shuffled around among the locations and the statistic is recomputed each time. This creates a reference distribution for the statistic under the null hypothesis of spatial randomness. The observed statistic is then compared to this reference distribution and a pseudo-significance computed as

$$p = (m + 1)/(n + 1)$$

where m is the number of values from the reference distribution that are equal to or greater than the observed join count and n is the number of permutations.

The Gamma test is a two-sided test in the sense that both extremely high values (e.g., larger than any value in the reference distribution) and extremely low values (e.g., smaller than any value in the reference distribution) can be considered to be significant. Depending on how the measure of attribute similarity is defined, a high value will indicate positive or negative spatial autocorrelation, and vice versa. For example, for a cross-product measure of attribute similarity, high values indicate positive spatial autocorrelation and low values negative spatial autocorrelation. For a squared difference measure, it is the reverse. This is similar to the interpretation of the *Moran's I* statistic and *Geary's C* statistic respectively.

Many spatial autocorrelation test statistics can be shown to be special cases of the Gamma index. In most instances, the Gamma index is an unstandardized version of the commonly used statistics. As such, the Gamma index is scale

[1] Hubert, L., R. Golledge and C.M. Costanzo (1981). Generalized procedures for evaluating spatial autocorrelation. Geographical Analysis 13, 224-233.

dependent, since no normalization is carried out (such as deviations from the mean or rescaling by the variance). Also, since the sum is over all the elements, the value of a Gamma statistic will grow with the sample size, everything else being the same.

PySAL implements four forms of the Gamma index. Three of these are pre-specified and one allows the user to pass any function that computes a measure of attribute similarity. This function should take three parameters: the vector of observations, an index i and an index j.

We will illustrate the Gamma index using the same small artificial example as we use for the *Join Count Statistics* in order to illustrate the similarities and differences between them. The data consist of a regular 4 by 4 lattice with values of 0 in the top half and values of 1 in the bottom half. We start with the usual imports, and set the random seed to 12345 in order to be able to replicate the results of the permutation approach.

```
>>> import pysal
>>> import numpy as np
>>> np.random.seed(12345)
```

We create the binary weights matrix for the 4 x 4 lattice and generate the observation vector y:

```
>>> w=pysal.lat2W(4,4)
>>> y=np.ones(16)
>>> y[0:8]=0
```

The Gamma index function has five arguments, three of which are optional. The first two arguments are the vector of observations (y) and the spatial weights object (w). Next are `operation`, the measure of attribute similarity, the default of which is `operation = 'c'` for cross-product similarity, $a_{ij} = y_i.y_j$. The other two built-in options are `operation = 's'` for squared difference, $a_{ij} = (y_i - y_j)^2$ and `operation = 'a'` for absolute difference, $a_{ij} = |y_i - y_j|$. The fourth option is to pass an arbitrary attribute similarity function, as in `operation = func`, where `func` is a function with three arguments, `def func(y,i,j)` with y as the vector of observations, and i and j as indices. This function should return a single value for attribute similarity.

The fourth argument allows the observed values to be standardized before the calculation of the Gamma index. To some extent, this addresses the scale dependence of the index, but not its dependence on the number of observations. The default is no standardization, `standardize = 'no'`. To force standardization, set `standardize = 'yes'` or `'y'`. The final argument is the number of permutations, `permutations` with the default set to 999.

As a first illustration, we invoke the Gamma index using all the default values, i.e. cross-product similarity, no standardization, and permutations set to 999. The interesting statistics are the magnitude of the Gamma index `g`, the standardized Gamma index using the mean and standard deviation from the reference distribution, `g_z` and the pseudo-p value obtained from the permutation, `g_sim_p`. In addition, the minimum (`min_g`), maximum (`max_g`) and mean (`mean_g`) of the reference distribution are available as well.

```
>>> g = pysal.Gamma(y,w)
>>> g.g
20.0
>>> "%.3f"%g.g_z
'3.188'
>>> g.p_sim_g
0.0030000000000000001
>>> g.min_g
0.0
>>> g.max_g
20.0
>>> g.mean_g
11.093093093093094
```

Note that the value for Gamma is exactly twice the BB statistic obtained in the example below, since the attribute similarity criterion is identical, but Gamma is not divided by 2.0. The observed value is very extreme, with only two replications from the permutation equalling the value of 20.0. This indicates significant positive spatial autocorrelation.

As a second illustration, we use the squared difference criterion, which corresponds to the BW Join Count statistic. We reset the random seed to keep comparability of the results.

```
>>> np.random.seed(12345)
>>> g1 = pysal.Gamma(y,w,operation='s')
>>> g1.g
8.0
>>> "%.3f"%g1.g_z
'-3.706'
>>> g1.p_sim_g
0.001
>>> g1.min_g
14.0
>>> g1.max_g
48.0
>>> g1.mean_g
25.623623623623622
```

The Gamma index value of 8.0 is exactly twice the value of the BW statistic for this example. However, since the Gamma index is used for a two-sided test, this value is highly significant, and with a negative z-value, this suggests positive spatial autocorrelation (similar to Geary's C). In other words, this result is consistent with the finding for the Gamma index that used cross-product similarity.

As a third example, we use the absolute difference for attribute similarity. The results are identical to those for squared difference since these two criteria are equivalent for 0-1 values.

```
>>> np.random.seed(12345)
>>> g2 = pysal.Gamma(y,w,operation='a')
>>> g2.g
8.0
>>> "%.3f"%g2.g_z
'-3.706'
>>> g2.p_sim_g
0.001
>>> g2.min_g
14.0
>>> g2.max_g
48.0
>>> g2.mean_g
25.623623623623622
```

We next illustrate the effect of standardization, using the default operation. As shown, the value of the statistic is quite different from the unstandardized form, but the inference is equivalent.

```
>>> np.random.seed(12345)
>>> g3 = pysal.Gamma(y,w,standardize='y')
>>> g3.g
32.0
>>> "%.3f"%g3.g_z
'3.706'
>>> g3.p_sim_g
0.001
>>> g3.min_g
-48.0
>>> g3.max_g
20.0
```

```
>>> "%.3f"%g3.mean_g
'-3.247'
```

Note that all the tests shown here have used the weights matrix in binary form. However, since the Gamma index is perfectly general, any standardization can be applied to the weights.

Finally, we illustrate the use of an arbitrary attribute similarity function. In order to compare to the results above, we will define a function that produces a cross product similarity measure. We will then pass this function to the `operation` argument of the Gamma index.

```
>>> np.random.seed(12345)
>>> def func(z,i,j):
...     q = z[i]*z[j]
...     return q
...
>>> g4 = pysal.Gamma(y,w,operation=func)
>>> g4.g
20.0
>>> "%.3f"%g4.g_z
'3.188'
>>> g4.p_sim_g
0.0030000000000000001
```

As expected, the results are identical to those obtained with the default operation.

Join Count Statistics

The join count statistics measure global spatial autocorrelation for binary data, i.e., with observations coded as 1 or B (for Black) and 0 or W (for White). They follow the very simple principle of counting joins, i.e., the arrangement of values between pairs of observations where the pairs correspond to neighbors. The three resulting join count statistics are BB, WW and BW. Both BB and WW are measures of positive spatial autocorrelation, whereas BW is an indicator of negative spatial autocorrelation.

To implement the join count statistics, we need the spatial weights matrix in binary (not row-standardized) form. With y as the vector of observations and the spatial *weight* as $w_{i,j}$, the three statistics can be expressed as:

$$BB = (1/2)\sum_i \sum_j y_i y_j w_{ij}$$

$$WW = (1/2)\sum_i \sum_j (1-y_i)(1-y_j) w_{ij}$$

$$BW = (1/2)\sum_i \sum_j (y_i - y_j)^2 w_{ij}$$

By convention, the join counts are divided by 2 to avoid double counting. Also, since the three joins exhaust all the possibilities, they sum to one half (because of the division by 2) of the total sum of weights $J = (1/2)S_0 = (1/2)\sum_i \sum_j w_{ij}$.

Inference for the join count statistics can be based on either an analytical approach or a computational approach. The analytical approach starts from the binomial distribution and derives the moments of the statistics under the assumption of free sampling and non-free sampling. The resulting mean and variance are used to construct a standardized z-variable which can be approximated as a standard normal variate.[2] However, the approximation is often poor in practice. We therefore only implement the computational approach.

[2] Technical details and derivations can be found in A.D. Cliff and J.K. Ord (1981). Spatial Processes, Models and Applications. London, Pion, pp. 34-41.

Computational inference is based on a permutation approach in which the values of y are randomly reshuffled many times to obtain a reference distribution of the statistics under the null hypothesis of spatial randomness. The observed join count is then compared to this reference distribution and a pseudo-significance computed as

$$p = (m + 1)/(n + 1)$$

where m is the number of values from the reference distribution that are equal to or greater than the observed join count and n is the number of permutations. Note that the join counts are a one sided-test. If the counts are extremely smaller than the reference distribution, this is not an indication of significance. For example, if the BW counts are extremely small, this is not an indication of *negative* BW autocorrelation, but instead points to the presence of BB or WW autocorrelation.

We will illustrate the join count statistics with a simple artificial example of a 4 by 4 square lattice with values of 0 in the top half and values of 1 in the bottom half.

We start with the usual imports, and set the random seed to 12345 in order to be able to replicate the results of the permutation approach.

```
>>> import pysal
>>> import numpy as np
>>> np.random.seed(12345)
```

We create the binary weights matrix for the 4 x 4 lattice and generate the observation vector y:

```
>>> w=pysal.lat2W(4,4)
>>> y=np.ones(16)
>>> y[0:8]=0
```

We obtain an instance of the joint count statistics BB, BW and WW as (J is half the sum of all the weights and should equal the sum of BB, WW and BW):

```
>>> jc=pysal.Join_Counts(y,w)
>>> jc.bb
10.0
>>> jc.bw
4.0
>>> jc.ww
10.0
>>> jc.J
24.0
```

The number of permutations is set to 999 by default. For other values, this parameter needs to be passed explicitly, as in:

```
>>> jc=pysal.Join_Counts(y,w,permutations=99)
```

The results in our simple example show that the BB counts are 10. There are in fact 3 horizontal joins in each of the bottom rows of the lattice as well as 4 vertical joins, which makes for bb = 3 + 3 + 4 = 10. The BW joins are 4, matching the separation between the bottom and top part.

The permutation results give a pseudo-p value for BB of 0.003, suggesting highly significant positive spatial autocorrelation. The average BB count for the sample of 999 replications is 5.5, quite a bit lower than the count of 10 we obtain. Only two instances of the replicated samples yield a value equal to 10, none is greater (the randomly permuted samples yield bb values between 0 and 10).

```
>>> len(jc.sim_bb)
999
>>> jc.p_sim_bb
```

```
0.0030000000000000001
>>> np.mean(jc.sim_bb)
5.5465465465465469
>>> np.max(jc.sim_bb)
10.0
>>> np.min(jc.sim_bb)
0.0
```

The results for BW (negative spatial autocorrelation) show a probability of 1.0 under the null hypothesis. This means that all the values of BW from the randomly permuted data sets were larger than the observed value of 4. In fact the range of these values is between 7 and 24. In other words, this again strongly points towards the presence of positive spatial autocorrelation. The observed number of BB and WW joins (10 each) is so high that there are hardly any BW joins (4).

```
>>> 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)
7.0
```

Moran's I

Moran's I measures the global spatial autocorrelation in an attribute y measured over n spatial units and is given as:

$$I = n/S_0 \sum_i \sum_j z_i w_{i,j} z_j / \sum_i z_i z_i$$

where $w_{i,j}$ is a spatial *weight*, $z_i = y_i - \bar{y}$, and $S_0 = \sum_i \sum_j w_{i,j}$. We illustrate the use of Moran's I with a case study of homicide rates for a group of 78 counties surrounding St. Louis over the period 1988-93.[3] We start with the usual imports:

```
>>> import pysal
>>> import numpy as np
```

Next, we read in the homicide rates:

```
>>> f = pysal.open(pysal.examples.get_path("stl_hom.txt"))
>>> y = np.array(f.by_col['HR8893'])
```

To calculate Moran's I we first need to read in a GAL file for a rook weights matrix and create an instance of W:

```
>>> w = pysal.open(pysal.examples.get_path("stl.gal")).read()
```

The instance of Moran's I can then be obtained with:

```
>>> mi = pysal.Moran(y, w, two_tailed=False)
>>> "%.3f"%mi.I
'0.244'
```

[3] Messner, S., L. Anselin, D. Hawkins, G. Deane, S. Tolnay, R. Baller (2000). An Atlas of the Spatial Patterning of County-Level Homicide, 1960-1990. Pittsburgh, PA, National Consortium on Violence Research (NCOVR)

```
>>> mi.EI
-0.012987012987012988
>>> "%.5f"%mi.p_norm
'0.00014'
```

From these results, we see that the observed value for I is significantly above its expected value, under the assumption of normality for the homicide rates.

If we peek inside the mi object to learn more:

```
>>> help(mi)
```

which generates:

```
Help on instance of Moran in module pysal.esda.moran:

class Moran
 |  Moran's I Global Autocorrelation Statistic
 |
 |  Parameters
 |  ----------
 |
 |  y                : array
 |                     variable measured across n spatial units
 |  w                : W
 |                     spatial weights instance
 |  permutations     : int
 |                     number of random permutations for calculation of pseudo-p_values
 |
 |
 |  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
 |  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 (one-sided)
 |                 for two-sided tests, this value should be multiplied by 2
 |  VI_rand      : float
 |                 variance of I under randomization assumption
 |  seI_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 (1-tailed)
 |  sim           : array (if permutations>0)
```

we see that we can base the inference not only on the normality assumption, but also on random permutations of the values on the spatial units to generate a reference distribution for I under the null:

```
>>> np.random.seed(10)
>>> mir = pysal.Moran(y, w, permutations = 9999)
```

The pseudo p value based on these permutations is:

```
>>> print mir.p_sim
0.0022
```

in other words there were 14 permutations that generated values for I that were as extreme as the original value, so the p value becomes (14+1)/(9999+1).[4] Alternatively, we could use the realized values for I from the permutations and compare the original I using a z-transformation to get:

```
>>> print mir.EI_sim
-0.0118217511619
>>> print mir.z_sim
4.55451777821
>>> print mir.p_z_sim
2.62529422013e-06
```

When the variable of interest (y) is rates based on populations with different sizes, the Moran's I value for y needs to be adjusted to account for the differences among populations.[5] To apply this adjustment, we can create an instance of the Moran_Rate class rather than the Moran class. For example, let's assume that we want to estimate the Moran's I for the rates of newborn infants who died of Sudden Infant Death Syndrome (SIDS). We start this estimation by reading in the total number of newborn infants (BIR79) and the total number of newborn infants who died of SIDS (SID79):

```
>>> f = pysal.open(pysal.examples.get_path("sids2.dbf"))
>>> b = np.array(f.by_col('BIR79'))
>>> e = np.array(f.by_col('SID79'))
```

Next, we create an instance of W:

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

Now, we create an instance of Moran_Rate:

```
>>> mi = pysal.esda.moran.Moran_Rate(e, b, w, two_tailed=False)
>>> "%6.4f" % mi.I
'0.1662'
>>> "%6.4f" % mi.EI
'-0.0101'
>>> "%6.4f" % mi.p_norm
'0.0042'
```

From these results, we see that the observed value for I is significantly higher than its expected value, after the adjustment for the differences in population.

[4] Because the permutations are random, results from those presented here may vary if you replicate this example.

[5] Assuncao, R. E. and Reis, E. A. 1999. A new proposal to adjust Moran's I for population density. Statistics in Medicine. 18, 2147-2162.

Geary's C

The fourth statistic for global spatial autocorrelation implemented in PySAL is Geary's C:

$$C = \frac{(n-1)}{2S_0} \sum_i \sum_j w_{i,j} (y_i - y_j)^2 / \sum_i z_i^2$$

with all the terms defined as above. Applying this to the St. Louis data:

```
>>> np.random.seed(12345)
>>> f = pysal.open(pysal.examples.get_path("stl_hom.txt"))
>>> y = np.array(f.by_col['HR8893'])
>>> w = pysal.open(pysal.examples.get_path("stl.gal")).read()
>>> gc = pysal.Geary(y, w)
>>> "%.3f"%gc.C
'0.597'
>>> gc.EC
1.0
>>> "%.3f"%gc.z_norm
'-5.449'
```

we see that the statistic C is significantly lower than its expected value EC. Although the sign of the standardized statistic is negative (in contrast to what held for I, the interpretation is the same, namely evidence of strong positive spatial autocorrelation in the homicide rates.

Similar to what we saw for Moran's I, we can base inference on Geary's C using random spatial permutations, which are actually run as a default with the number of permutations=999 (this is why we set the seed of the random number generator to 12345 to replicate the result):

```
>>> gc.p_sim
0.001
```

which indicates that none of the C values from the permuted samples was as extreme as our observed value.

Getis and Ord's G

The last statistic for global spatial autocorrelation implemented in PySAL is Getis and Ord's G:

$$G(d) = \frac{\sum_i \sum_j w_{i,j}(d) y_i y_j}{\sum_i \sum_j y_i y_j}$$

where d is a threshold distance used to define a spatial *weight*. Only `pysal.weights.Distance.DistanceBand` weights objects are applicable to Getis and Ord's G. Applying this to the St. Louis data:

```
>>> dist_w = pysal.threshold_binaryW_from_shapefile('../pysal/examples/stl_hom.shp',0.
↪6)
>>> dist_w.transform = "B"
>>> from pysal.esda.getisord import G
>>> g = G(y, dist_w)
>>> print g.G
0.103483215873
>>> print g.EG
0.0752580752581
>>> print g.z_norm
3.28090342959
>>> print g.p_norm
0.000517375830488
```

Although we switched the contiguity-based weights object into another distance-based one, we see that the statistic G is significantly higher than its expected value EG under the assumption of normality for the homicide rates.

Similar to what we saw for Moran's I and Geary's C, we can base inference on Getis and Ord's G using random spatial permutations:

```
>>> np.random.seed(12345)
>>> g = G(y, dist_w, permutations=9999)
>>> print g.p_z_sim
0.000564384586974
>>> print g.p_sim
0.0065
```

with the first p-value based on a z-transform of the observed G relative to the distribution of values obtained in the permutations, and the second based on the cumulative probability of the observed value in the empirical distribution.

Local Autocorrelation

To measure local autocorrelation quantitatively, PySAL implements Local Indicators of Spatial Association (LISAs) for Moran's I and Getis and Ord's G.

Local Moran's I

PySAL implements local Moran's I as follows:

$$I_i = \frac{(n-1)z_i \sum_j w_{i,j} z_j}{\sum_j z_j^2}$$

which results in n values of local spatial autocorrelation, 1 for each spatial unit. Continuing on with the St. Louis example, the LISA statistics are obtained with:

```
>>> f = pysal.open(pysal.examples.get_path("stl_hom.txt"))
>>> y = np.array(f.by_col['HR8893'])
>>> w = pysal.open(pysal.examples.get_path("stl.gal")).read()
>>> np.random.seed(12345)
>>> lm = pysal.Moran_Local(y,w)
>>> lm.n
78
>>> len(lm.Is)
78
```

thus we see 78 LISAs are stored in the vector lm.Is. Inference about these values is obtained through conditional randomization[6] which leads to pseudo p-values for each LISA:

```
>>> lm.p_sim
array([ 0.176,  0.073,  0.405,  0.267,  0.332,  0.057,  0.296,  0.242,
        0.055,  0.062,  0.273,  0.488,  0.44 ,  0.354,  0.415,  0.478,
        0.473,  0.374,  0.415,  0.21 ,  0.161,  0.025,  0.338,  0.375,
        0.285,  0.374,  0.208,  0.3  ,  0.373,  0.411,  0.478,  0.414,
        0.009,  0.429,  0.269,  0.015,  0.005,  0.002,  0.077,  0.001,
        0.088,  0.459,  0.435,  0.365,  0.231,  0.017,  0.033,  0.04 ,
        0.068,  0.101,  0.284,  0.309,  0.113,  0.457,  0.045,  0.269,
        0.118,  0.346,  0.328,  0.379,  0.342,  0.39 ,  0.376,  0.467,
        0.357,  0.241,  0.26 ,  0.401,  0.185,  0.172,  0.248,  0.4  ,
        0.482,  0.159,  0.373,  0.455,  0.083,  0.128])
```

[6] The n-1 spatial units other than i are used to generate the empirical distribution of the LISA statistics for each i.

To identify the significant[7] LISA values we can use numpy indexing:

```
>>> sig = lm.p_sim<0.05
>>> lm.p_sim[sig]
array([ 0.025,  0.009,  0.015,  0.005,  0.002,  0.001,  0.017,  0.033,
        0.04 ,  0.045])
```

and then use this indexing on the q attribute to find out which quadrant of the Moran scatter plot each of the significant values is contained in:

```
>>> lm.q[sig]
array([4, 1, 3, 1, 3, 1, 1, 3, 3, 3])
```

As in the case of global Moran's I, when the variable of interest is rates based on populations with different sizes, we need to account for the differences among population to estimate local Moran's Is. Continuing on with the SIDS example above, the adjusted local Moran's Is are obtained with:

```
    >>> f = pysal.open(pysal.examples.get_path("sids2.dbf"))
    >>> b = np.array(f.by_col('BIR79'))
    >>> e = np.array(f.by_col('SID79'))
    >>> w = pysal.open(pysal.examples.get_path("sids2.gal")).read()
>>> np.random.seed(12345)
>>> lm = pysal.esda.moran.Moran_Local_Rate(e, b, w)
>>> lm.Is[:10]
array([-0.13452366, -1.21133985,  0.05019761,  0.06127125, -0.12627466,
        0.23497679,  0.26345855, -0.00951288, -0.01517879, -0.34513514])
```

As demonstrated above, significant Moran's Is can be identified by using numpy indexing:

```
>>> sig = lm.p_sim<0.05
>>> lm.p_sim[sig]
array([ 0.021,  0.04 ,  0.047,  0.015,  0.001,  0.017,  0.032,  0.031,
        0.019,  0.014,  0.004,  0.048,  0.003])
```

Local G and G*

Getis and Ord's G can be localized in two forms: G_i and G^*_i.

$$G_i(d) = \frac{\sum_j w_{i,j}(d) y_j - W_i\bar{y}(i)}{s(i)\{[(n-1)S_{1i} - W^2_i]/(n-2)\}^(1/2)}, j \neq i$$

$$G^*_i(d) = \frac{\sum_j w_{i,j}(d) y_j - W^*_i\bar{y}}{s\{[(nS^*_{1i}) - (W^*_i)^2]/(n-1)\}^(1/2)}, j = i$$

where we have $W_i = \sum_{j \neq i} w_{i,j}(d)$, $\bar{y}(i) = \frac{\sum_j y_j}{(n-1)}$, $s^2(i) = \frac{\sum_j y^2_j}{(n-1)} - [\bar{y}(i)]^2$, $W^*_i = W_i + wi, i$, $S_{1i} = \sum_j w^2_{i,j}(j \neq i)$, and $S^*_{1i} = \sum_j w^2_{i,j}(\forall j)$, $\bar{y}$ and s^2 denote the usual sample mean and variance of y.

Continuing on with the St. Louis example, the G_i and G^*_i statistics are obtained with:

```
>>> from pysal.esda.getisord import G_Local
>>> np.random.seed(12345)
>>> lg = G_Local(y, dist_w)
>>> lg.n
```

[7] Caution is required in interpreting the significance of the LISA statistics due to difficulties with multiple comparisons and a lack of independence across the individual tests. For further discussion see Anselin, L. (1995). "Local indicators of spatial association – LISA". Geographical Analysis, 27, 93-115.

```
78
>>> len(lg.Gs)
78
>>> lgstar = G_Local(y, dist_w, star=True)
>>> lgstar.n
78
>>> len(lgstar.Gs)
78
```

thus we see 78 G_i and G_i^* are stored in the vector lg.Gs and lgstar.Gs, respectively. Inference about these values is obtained through conditional randomization as in the case of local Moran's I:

```
>>> lg.p_sim
array([ 0.301,  0.037,  0.457,  0.011,  0.062,  0.006,  0.094,  0.163,
        0.075,  0.078,  0.419,  0.286,  0.138,  0.443,  0.36 ,  0.484,
        0.434,  0.251,  0.415,  0.21 ,  0.177,  0.001,  0.304,  0.042,
        0.285,  0.394,  0.208,  0.089,  0.244,  0.493,  0.478,  0.433,
        0.006,  0.429,  0.037,  0.105,  0.005,  0.216,  0.23 ,  0.023,
        0.105,  0.343,  0.395,  0.305,  0.264,  0.017,  0.033,  0.01 ,
        0.001,  0.115,  0.034,  0.225,  0.043,  0.312,  0.045,  0.092,
        0.118,  0.428,  0.258,  0.379,  0.408,  0.39 ,  0.475,  0.493,
        0.357,  0.298,  0.232,  0.454,  0.149,  0.161,  0.226,  0.4  ,
        0.482,  0.159,  0.27 ,  0.423,  0.083,  0.128])
```

To identify the significant G_i values we can use numpy indexing:

```
>>> sig = lg.p_sim<0.05
>>> lg.p_sim[sig]
array([ 0.037,  0.011,  0.006,  0.001,  0.042,  0.006,  0.037,  0.005,
        0.023,  0.017,  0.033,  0.01 ,  0.001,  0.034,  0.043,  0.045])
```

Further Information

For further details see the *ESDA API*.

Spatial Econometrics

Comprehensive user documentation on spreg can be found in Anselin, L. and S.J. Rey (2014) Modern Spatial Econometrics in Practice: A Guide to GeoDa, GeoDaSpace and PySAL. GeoDa Press, Chicago.

spreg API

For further details see the *spreg API*.

Spatial Smoothing

Contents

Introduction

In the spatial analysis of attributes measured for areal units, it is often necessary to transform an extensive variable, such as number of disease cases per census tract, into an intensive variable that takes into account the underlying population at risk. Raw rates, counts divided by population values, are a common standardization in the literature, yet these tend to have unequal reliability due to different population sizes across the spatial units. This problem becomes severe for areas with small population values, since the raw rates for those areas tend to have higher variance.

A variety of spatial smoothing methods have been suggested to address this problem by aggregating the counts and population values for the areas neighboring an observation and using these new measurements for its rate computation. PySAL provides a range of smoothing techniques that exploit different types of moving windows and non-parametric weighting schemes as well as the Empirical Bayesian principle. In addition, PySAL offers several methods for calculating age-standardized rates, since age standardization is critical in estimating rates of some events where the probability of an event occurrence is different across different age groups.

In what follows, we overview the methods for age standardization and spatial smoothing and describe their implementations in PySAL.[1]

Age Standardization in PySAL

Raw rates, counts divided by populations values, are based on an implicit assumption that the risk of an event is constant over all age/sex categories in the population. For many phenomena, however, the risk is not uniform and often highly correlated with age. To take this into account explicitly, the risks for individual age categories can be estimated separately and averaged to produce a representative value for an area.

PySAL supports three approaches to this age standardization: crude, direct, and indirect standardization.

Crude Age Standardization

In this approach, the rate for an area is simply the sum of age-specific rates weighted by the ratios of each age group in the total population.

To obtain the rates based on this approach, we first need to create two variables that correspond to event counts and population values, respectively.

[1] Although this tutorial provides an introduction to the PySAL implementations for spatial smoothing, it is not exhaustive. Complete documentation for the implementations can be found by accessing the help from within a Python interpreter.

```
>>> import numpy as np
>>> e = np.array([30, 25, 25, 15, 33, 21, 30, 20])
>>> b = np.array([100, 100, 110, 90, 100, 90, 110, 90])
```

Each set of numbers should include n by h elements where n and h are the number of areal units and the number of age groups. In the above example there are two regions with 4 age groups. Age groups are identical across regions. The first four elements in b represent the populations of 4 age groups in the first region, and the last four elements the populations of the same age groups in the second region.

To apply the crude age standardization, we need to make the following function call:

```
>>> from pysal.esda import smoothing as sm
>>> sm.crude_age_standardization(e, b, 2)
array([ 0.2375    ,  0.26666667])
```

In the function call above, the last argument indicates the number of area units. The outcome in the second line shows that the age-standardized rates for two areas are about 0.24 and 0.27, respectively.

Direct Age Standardization

Direct age standardization is a variation of the crude age standardization. While crude age standardization uses the ratios of each age group in the observed population, direct age standardization weights age-specific rates by the ratios of each age group in a reference population. This reference population, the so-called standard million, is another required argument in the PySAL implementation of direct age standardization:

```
>>> s = np.array([100, 90, 100, 90, 100, 90, 100, 90])
>>> rate = sm.direct_age_standardization(e, b, s, 2, alpha=0.05)
>>> np.array(rate).round(6)
array([[ 0.23744 ,  0.192049,  0.290485],
       [ 0.266507,  0.217714,  0.323051]])
```

The outcome of direct age standardization includes a set of standardized rates and their confidence intervals. The confidence intervals can vary according to the value for the last argument, alpha.

Indirect Age Standardization

While direct age standardization effectively addresses the variety in the risks across age groups, its indirect counterpart is better suited to handle the potential imprecision of age-specific rates due to the small population size. This method uses age-specific rates from the standard million instead of the observed population. It then weights the rates by the ratios of each age group in the observed population. To compute the age-specific rates from the standard million, the PySAL implementation of indirect age standardization requires another argument that contains the counts of the events occurred in the standard million.

```
>>> s_e = np.array([10, 15, 12, 10, 5, 3, 20, 8])
>>> rate = sm.indirect_age_standardization(e, b, s_e, s, 2, alpha=0.05)
>>> np.array(rate).round(6)
array([[ 0.208055,  0.170156,  0.254395],
       [ 0.298892,  0.246631,  0.362228]])
```

The outcome of indirect age standardization is the same as that of its direct counterpart.

Spatial Smoothing in PySAL

Mean and Median Based Smoothing

A simple approach to rate smoothing is to find a local average or median from the rates of each observation and its neighbors. The first method adopting this approach is the so-called locally weighted averages or disk smoother. In this method a rate for each observation is replaced by an average of rates for its neighbors. A *spatial weights object* is used to specify the neighborhood relationships among observations. To obtain locally weighted averages of the homicide rates in the counties surrounding St. Louis during 1979-84, we first read the corresponding data table and extract data values for the homicide counts (the 11th column) and total population (the 13th column):

```
>>> import pysal
>>> stl = pysal.open('../pysal/examples/stl_hom.csv', 'r')
>>> e, b = np.array(stl[:,10]), np.array(stl[:,13])
```

We then read the spatial weights file defining neighborhood relationships among the counties and ensure that the *order* of observations in the weights object is the same as that in the data table.

```
>>> w = pysal.open('../pysal/examples/stl.gal', 'r').read()
>>> if not w.id_order_set: w.id_order = range(1,len(stl) + 1)
```

Now we calculate locally weighted averages of the homicide rates.

```
>>> rate = sm.Disk_Smoother(e, b, w)
>>> rate.r
array([  4.56502262e-05,   3.44027685e-05,   3.38280487e-05,
         4.78530468e-05,   3.12278573e-05,   2.22596997e-05,
         ...
         5.29577710e-05,   5.51034691e-05,   4.65160450e-05,
         5.32513363e-05,   3.86199097e-05,   1.92952422e-05])
```

A variation of locally weighted averages is to use median instead of mean. In other words, the rate for an observation can be replaced by the median of the rates of its neighbors. This method is called locally weighted median and can be applied in the following way:

```
>>> rate = sm.Spatial_Median_Rate(e, b, w)
>>> rate.r
array([  3.96047383e-05,   3.55386859e-05,   3.28308921e-05,
         4.30731238e-05,   3.12453969e-05,   1.97300409e-05,
         ...
         6.10668237e-05,   5.86355507e-05,   3.67396656e-05,
         4.82535850e-05,   5.51831429e-05,   2.99877050e-05])
```

In this method the procedure to find local medians can be iterated until no further change occurs. The resulting local medians are called iteratively resmoothed medians.

```
>>> rate = sm.Spatial_Median_Rate(e, b, w, iteration=10)
>>> rate.r
array([  3.10194715e-05,   2.98419439e-05,   3.10194715e-05,
         3.10159267e-05,   2.99214885e-05,   2.80530524e-05,
         ...
         3.81364519e-05,   4.72176972e-05,   3.75320135e-05,
         3.76863269e-05,   4.72176972e-05,   3.75320135e-05])
```

The pure local medians can also be replaced by a weighted median. To obtain weighted medians, we need to create an array of weights. For example, we can use the total population of the counties as auxiliary weights:

```
>>> rate = sm.Spatial_Median_Rate(e, b, w, aw=b)
>>> rate.r
```

```
array([  5.77412020e-05,   4.46449551e-05,   5.77412020e-05,
         5.77412020e-05,   4.46449551e-05,   3.61363528e-05,
         ...
         5.49703305e-05,   5.86355507e-05,   3.67396656e-05,
         3.67396656e-05,   4.72176972e-05,   2.99877050e-05])
```

When obtaining locally weighted medians, we can consider only a specific subset of neighbors rather than all of them. A representative method following this approach is the headbanging smoother. In this method all areal units are represented by their geometric centroids. Among the neighbors of each observation, only near collinear points are considered for median search. Then, triples of points are selected from the near collinear points, and local medians are computed from the triples' rates.[2] We apply this headbanging smoother to the rates of the deaths from Sudden Infant Death Syndrome (SIDS) for North Carolina counties during 1974-78. We first need to read the source data and extract the event counts (the 9th column) and population values (the 9th column). In this example the population values correspond to the numbers of live births during 1974-78.

```
>>> sids_db = pysal.open('../pysal/examples/sids2.dbf', 'r')
>>> e, b = np.array(sids_db[:,9]), np.array(sids_db[:,8])
```

Now we need to find triples for each observation. To support the search of triples, PySAL provides a class called Headbanging_Triples. This class requires an array of point observations, a spatial weights object, and the number of triples as its arguments:

```
>>> from pysal import knnW
>>> sids = pysal.open('../pysal/examples/sids2.shp', 'r')
>>> sids_d = np.array([i.centroid for i in sids])
>>> sids_w = knnW(sids_d,k=5)
>>> if not sids_w.id_order_set: sids_w.id_order = sids_w.id_order
>>> triples = sm.Headbanging_Triples(sids_d,sids_w,k=5)
```

The second line in the above example shows how to extract centroids of polygons. In this example we define 5 neighbors for each observation by using nearest neighbors criteria. In the last line we define the maximum number of triples to be found as 5.

Now we use the triples to compute the headbanging median rates:

```
>>> rate = sm.Headbanging_Median_Rate(e,b,triples)
>>> rate.r
array([ 0.00075586,  0.        ,  0.0008285 ,  0.0018315 ,  0.00498891,
        0.00482094,  0.00133156,  0.0018315 ,  0.00413223,  0.00142116,
        ...
        0.00221541,  0.00354767,  0.00259903,  0.00392952,  0.00207125,
        0.00392952,  0.00229253,  0.00392952,  0.00229253,  0.00229253])
```

As in the locally weighted medians, we can use a set of auxiliary weights and resmooth the medians iteratively.

Non-parametric Smoothing

Non-parametric smoothing methods compute rates without making any assumptions of distributional properties of rate estimates. A representative method in this approach is spatial filtering. PySAL provides the most simplistic form of spatial filtering where a user-specified grid is imposed on the data set and a moving window withi a fixed or adaptive radius visits each vertex of the grid to compute the rate at the vertex. Using the previous SIDS example, we can use Spatial_Filtering class:

[2] For the details of triple selection and headbanging smoothing please refer to Anselin, L., Lozano, N., and Koschinsky, J. (2006). "Rate Transformations and Smoothing". GeoDa Center Research Report.

```
>>> bbox = [sids.bbox[:2], sids.bbox[2:]]
>>> rate = sm.Spatial_Filtering(bbox, sids_d, e, b, 10, 10, r=1.5)
>>> rate.r
array([ 0.00152555,  0.00079271,  0.00161253,  0.00161253,  0.00139513,
        0.00139513,  0.00139513,  0.00139513,  0.00139513,  0.00156348,
        ...
        0.00240216,  0.00237389,  0.00240641,  0.00242211,  0.0024854 ,
        0.00255477,  0.00266573,  0.00288918,  0.0028991 ,  0.00293492])
```

The first and second arguments of the Spatial_Filtering class are a minimum bounding box containing the observations and a set of centroids representing the observations. Be careful that the bounding box is NOT the bounding box of the centroids. The fifth and sixth arguments are to specify the numbers of grid cells along x and y axes. The last argument, r, is to define the radius of the moving window. When this parameter is set, a fixed radius is applied to all grid vertices. To make the size of moving window variable, we can specify the minimum number of population in the moving window without specifying r:

```
>>> rate = sm.Spatial_Filtering(bbox, sids_d, e, b, 10, 10, pop=10000)
>>> rate.r
array([ 0.00157398,  0.00157398,  0.00157398,  0.00157398,  0.00166885,
        0.00166885,  0.00166885,  0.00166885,  0.00166885,  0.00166885,
        ...
        0.00202977,  0.00215322,  0.00207378,  0.00207378,  0.00217173,
        0.00232408,  0.00222717,  0.00245399,  0.00267857,  0.00267857])
```

The spatial rate smoother is another non-parametric smoothing method that PySAL supports. This smoother is very similar to the locally weighted averages. In this method, however, the weighted sum is applied to event counts and population values separately. The resulting weighted sum of event counts is then divided by the counterpart of population values. To obtain neighbor information, we need to use a spatial weights matrix as before.

```
>>> rate = sm.Spatial_Rate(e, b, sids_w)
>>> rate.r
array([ 0.00114976,  0.00104622,  0.00110001,  0.00153257,  0.00399662,
        0.00361428,  0.00146807,  0.00238521,  0.00288871,  0.00145228,
        ...
        0.00240839,  0.00376101,  0.00244941,  0.0028813 ,  0.00240839,
        0.00261705,  0.00226554,  0.0031575 ,  0.00254536,  0.0029003 ])
```

Another variation of spatial rate smoother is kernel smoother. PySAL supports kernel smoothing by using a kernel spatial weights instance in place of a general spatial weights object.

```
>>> from pysal import Kernel
>>> kw = Kernel(sids_d)
>>> if not kw.id_order_set: kw.id_order = range(0,len(sids_d))
>>> rate = sm.Kernel_Smoother(e, b, kw)
>>> rate.r
array([ 0.0009831 ,  0.00104298,  0.00137113,  0.00166406,  0.00556741,
        0.00442273,  0.00158202,  0.00243354,  0.00282158,  0.00099243,
        ...
        0.00221017,  0.00328485,  0.00257988,  0.00370461,  0.0020566 ,
        0.00378135,  0.00240358,  0.00432019,  0.00227857,  0.00251648])
```

Age-adjusted rate smoother is another non-parametric smoother that PySAL provides. This smoother applies direct age standardization while computing spatial rates. To illustrate the age-adjusted rate smoother, we create a new set of event counts and population values as well as a new kernel weights object.

```
>>> e = np.array([10, 8, 1, 4, 3, 5, 4, 3, 2, 1, 5, 3])
>>> b = np.array([100, 90, 15, 30, 25, 20, 30, 20, 80, 80, 90, 60])
>>> s = np.array([98, 88, 15, 29, 20, 23, 33, 25, 76, 80, 89, 66])
>>> points=[(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
>>> kw=Kernel(points)
>>> if not kw.id_order_set: kw.id_order = range(0,len(points))
```

In the above example we created 6 observations each of which has two age groups. To apply age-adjusted rate smoothing, we use the Age_Adjusted_Smoother class as follows:

```
>>> rate = sm.Age_Adjusted_Smoother(e, b, kw, s)
>>> rate.r
array([ 0.10519625,  0.08494318,  0.06440072,  0.06898604,  0.06952076,
        0.05020968])
```

Empirical Bayes Smoothers

The last group of smoothing methods that PySAL supports is based upon the Bayesian principle. These methods adjust a raw rate by taking into account information in the other raw rates. As a reference PySAL provides a method for a-spatial Empirical Bayes smoothing:

```
>>> e, b = sm.sum_by_n(e, np.ones(12), 6), sm.sum_by_n(b, np.ones(12), 6)
>>> rate = sm.Empirical_Bayes(e, b)
>>> rate.r
array([ 0.09080775,  0.09252352,  0.12332267,  0.10753624,  0.03301368,
        0.05934766])
```

In the first line of the above example we aggregate the event counts and population values by observation. Next we applied the Empirical_Bayes class to the aggregated counts and population values.

A spatial Empirical Bayes smoother is also implemented in PySAL. This method requires an additional argument, i.e., a spatial weights object. We continue to reuse the kernel spatial weights object we built before.

```
>>> rate = sm.Spatial_Empirical_Bayes(e, b, kw)
>>> rate.r
array([ 0.10105263,  0.10165261,  0.16104362,  0.11642038,  0.0226908 ,
        0.05270639])
```

Excess Risk

Besides a variety of spatial smoothing methods, PySAL provides a class for estimating excess risk from event counts and population values. Excess risks are the ratios of observed event counts over expected event counts. An example for the class usage is as follows:

```
>>> risk = sm.Excess_Risk(e, b)
>>> risk.r
array([ 1.23737916,  1.45124717,  2.32199546,  1.82857143,  0.24489796,
        0.69659864])
```

Further Information

For further details see the *Smoothing API*.

Regionalization

Introduction

PySAL offers a number of tools for the construction of regions. For the purposes of this section, a “region” is a group of “areas,” and there are generally multiple regions in a particular dataset. At this time, PySAL offers the max-p regionalization algorithm and tools for constructing random regions.

max-p

Most regionalization algorithms require the user to define a priori the number of regions to be built (e.g. k-means clustering). The max-p algorithm[1] determines the number of regions (p) endogenously based on a set of areas, a matrix of attributes on each area and a floor constraint. The floor constraint defines the minimum bound that a variable must reach for each region; for example, a constraint might be the minimum population each region must have. max-p further enforces a contiguity constraint on the areas within regions.

To illustrate this we will use data on per capita income from the lower 48 US states over the period 1929-2010. The goal is to form contiguous regions of states displaying similar levels of income throughout this period:

```
>>> import pysal
>>> import numpy as np
>>> import random
>>> f = pysal.open("../pysal/examples/usjoin.csv")
>>> pci = np.array([f.by_col[str(y)] for y in range(1929, 2010)])
>>> pci = pci.transpose()
>>> pci.shape
(48, 81)
```

We also require set of binary contiguity *weights* for the Maxp class:

```
>>> w = pysal.open("../pysal/examples/states48.gal").read()
```

Once we have the attribute data and our weights object we can create an instance of Maxp:

```
>>> np.random.seed(100)
>>> random.seed(10)
>>> r = pysal.Maxp(w, pci, floor = 5, floor_variable = np.ones((48, 1)), initial = 99)
```

Here we are forming regions with a minimum of 5 states in each region, so we set the floor_variable to a simple unit vector to ensure this floor constraint is satisfied. We also specify the initial number of feasible solutions to 99 - which are then searched over to pick the optimal feasible solution to then commence with the more expensive swapping component of the algorithm.[2]

The Maxp instance s has a number of attributes regarding the solution. First is the definition of the regions:

```
>>> r.regions
[['44', '34', '3', '25', '1', '4', '47'], ['12', '46', '20', '24', '13'], ['14', '45',
↪ '35', '30', '39'], ['6', '27', '17', '29', '5', '43'], ['33', '40', '28', '15', '41
↪', '9', '23', '31', '38'], ['37', '8', '0', '7', '21', '2'], ['32', '19', '11', '10
↪', '22'], ['16', '26', '42', '18', '36']]
```

[1] Duque, J. C., L. Anselin and S. J. Rey. 2011. “The max-p-regions problem.” *Journal of Regional Science* DOI: 10.1111/j.1467-9787.2011.00743.x

[2] Because this is a randomized algorithm, results may vary when replicating this example. To reproduce a regionalization solution, you should first set the random seed generator. See http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.seed.html for more information.

which is a list of eight lists of region ids. For example, the first nested list indicates there are seven states in the first region, while the last region has five states. To determine which states these are we can read in the names from the original csv file:

```
>>> f.header
['Name', 'STATE_FIPS', '1929', '1930', '1931', '1932', '1933', '1934', '1935', '1936',
↪ '1937', '1938', '1939', '1940', '1941', '1942', '1943', '1944', '1945', '1946',
↪'1947', '1948', '1949', '1950', '1951', '1952', '1953', '1954', '1955', '1956',
↪'1957', '1958', '1959', '1960', '1961', '1962', '1963', '1964', '1965', '1966',
↪'1967', '1968', '1969', '1970', '1971', '1972', '1973', '1974', '1975', '1976',
↪'1977', '1978', '1979', '1980', '1981', '1982', '1983', '1984', '1985', '1986',
↪'1987', '1988', '1989', '1990', '1991', '1992', '1993', '1994', '1995', '1996',
↪'1997', '1998', '1999', '2000', '2001', '2002', '2003', '2004', '2005', '2006',
↪'2007', '2008', '2009']
>>> names = f.by_col('Name')
>>> names = np.array(names)
>>> print names
['Alabama' 'Arizona' 'Arkansas' 'California' 'Colorado' 'Connecticut'
 'Delaware' 'Florida' 'Georgia' 'Idaho' 'Illinois' 'Indiana' 'Iowa'
 'Kansas' 'Kentucky' 'Louisiana' 'Maine' 'Maryland' 'Massachusetts'
 'Michigan' 'Minnesota' 'Mississippi' 'Missouri' 'Montana' 'Nebraska'
 'Nevada' 'New Hampshire' 'New Jersey' 'New Mexico' 'New York'
 'North Carolina' 'North Dakota' 'Ohio' 'Oklahoma' 'Oregon' 'Pennsylvania'
 'Rhode Island' 'South Carolina' 'South Dakota' 'Tennessee' 'Texas' 'Utah'
 'Vermont' 'Virginia' 'Washington' 'West Virginia' 'Wisconsin' 'Wyoming']
```

and then loop over the region definitions to identify the specific states comprising each of the regions:

```
>>> for region in r.regions:
...     ids = map(int,region)
...     print names[ids]
...
['Washington' 'Oregon' 'California' 'Nevada' 'Arizona' 'Colorado' 'Wyoming']
['Iowa' 'Wisconsin' 'Minnesota' 'Nebraska' 'Kansas']
['Kentucky' 'West Virginia' 'Pennsylvania' 'North Carolina' 'Tennessee']
['Delaware' 'New Jersey' 'Maryland' 'New York' 'Connecticut' 'Virginia']
['Oklahoma' 'Texas' 'New Mexico' 'Louisiana' 'Utah' 'Idaho' 'Montana'
 'North Dakota' 'South Dakota']
['South Carolina' 'Georgia' 'Alabama' 'Florida' 'Mississippi' 'Arkansas']
['Ohio' 'Michigan' 'Indiana' 'Illinois' 'Missouri']
['Maine' 'New Hampshire' 'Vermont' 'Massachusetts' 'Rhode Island']
```

We can evaluate our solution by developing a pseudo pvalue for the regionalization. This is done by comparing the within region sum of squares for the solution against simulated solutions where areas are randomly assigned to regions that maintain the cardinality of the original solution. This method must be explicitly called once the Maxp instance has been created:

```
>>> r.inference()
>>> r.pvalue
0.01
```

so we see we have a regionalization that is significantly different than a chance partitioning.

Random Regions

PySAL offers functionality to generate random regions based on user-defined constraints. There are three optional parameters to constrain the regionalization: number of regions, cardinality and contiguity. The default case simply takes a list of area IDs and randomly selects the number of regions and then allocates areas to each region. The user can also pass a vector of integers to the cardinality parameter to designate the number of areas to randomly assign to each region. The contiguity parameter takes a *spatial weights object* and uses that to ensure that each region is made up of spatially contiguous areas. When the contiguity constraint is enforced, it is possible to arrive at infeasible solutions; the maxiter parameter can be set to make multiple attempts to find a feasible solution. The following examples show some of the possible combinations of constraints.

```
>>> import random
>>> import numpy as np
>>> import pysal
>>> from pysal.region import Random_Region
>>> nregs = 13
>>> cards = range(2,14) + [10]
>>> w = pysal.lat2W(10,10,rook = False)
>>> ids = w.id_order
>>>
>>> # unconstrained
>>> random.seed(10)
>>> np.random.seed(10)
>>> t0 = Random_Region(ids)
>>> t0.regions[0]
[19, 14, 43, 37, 66, 3, 79, 41, 38, 68, 2, 1, 60]
>>> # cardinality and contiguity constrained (num_regions implied)
>>> random.seed(60)
>>> np.random.seed(60)
>>> t1 = pysal.region.Random_Region(ids, num_regions = nregs, cardinality = cards,␣
↪contiguity = w)
>>> t1.regions[0]
[88, 97, 98, 89, 99, 86, 78, 59, 49, 69, 68, 79, 77]
>>> # cardinality constrained (num_regions implied)
>>> random.seed(100)
>>> np.random.seed(100)
>>> t2 = Random_Region(ids, num_regions = nregs, cardinality = cards)
>>> t2.regions[0]
[37, 62]
>>> # number of regions and contiguity constrained
>>> random.seed(100)
>>> np.random.seed(100)
>>> t3 = Random_Region(ids, num_regions = nregs, contiguity = w)
>>> t3.regions[1]
[71, 72, 70, 93, 51, 91, 85, 74, 63, 73, 61, 62, 82]
>>> # cardinality and contiguity constrained
>>> random.seed(60)
>>> np.random.seed(60)
>>> t4 = Random_Region(ids, cardinality = cards, contiguity = w)
>>> t4.regions[0]
[88, 97, 98, 89, 99, 86, 78, 59, 49, 69, 68, 79, 77]
>>> # number of regions constrained
>>> random.seed(100)
>>> np.random.seed(100)
>>> t5 = Random_Region(ids, num_regions = nregs)
>>> t5.regions[0]
[37, 62, 26, 41, 35, 25, 36]
>>> # cardinality constrained
>>> random.seed(100)
>>> np.random.seed(100)
>>> t6 = Random_Region(ids, cardinality = cards)
>>> t6.regions[0]
```

```
[37, 62]
>>> # contiguity constrained
>>> random.seed(100)
>>> np.random.seed(100)
>>> t7 = Random_Region(ids, contiguity = w)
>>> t7.regions[0]
[37, 27, 36, 17]
>>>
```

Further Information

For further details see the *Regionalization API*.

Spatial Dynamics

Contents

Introduction

PySAL implements a number of exploratory approaches to analyze the dynamics of longitudinal spatial data, or observations on fixed areal units over multiple time periods. Examples could include time series of voting patterns in US Presidential elections, time series of remote sensing images, labor market dynamics, regional business cycles, among many others. Two broad sets of spatial dynamics methods are implemented to analyze these data types. The first are Markov based methods, while the second are based on Rank dynamics.

Additionally, methods are included in this module to analyze patterns of individual events which have spatial and temporal coordinates associated with them. Examples include locations and times of individual cases of disease or crimes. Methods are included here to determine if these event patterns exhibit space-time interaction.

Markov Based Methods

The Markov based methods include classic Markov chains and extensions of these approaches to deal with spatially referenced data. In what follows we illustrate the functionality of these Markov methods. Readers interested in the methodological foundations of these approaches are directed to[1].

Classic Markov

We start with a look at a simple example of classic Markov methods implemented in PySAL. A Markov chain may be in one of k different states at any point in time. These states are exhaustive and mutually exclusive. For example, if one had a time series of remote sensing images used to develop land use classifications, then the states could be defined as the specific land use classes and interest would center on the transitions in and out of different classes for each pixel.

For example, let's construct a small artificial chain consisting of 3 states (a,b,c) and 5 different pixels at three different points in time:

```
>>> import pysal
>>> import numpy as np
>>> c = np.array([['b','a','c'],['c','c','a'],['c','b','c'],['a','a','b'],['a','b
↪','c']])
>>> c
array([['b', 'a', 'c'],
       ['c', 'c', 'a'],
       ['c', 'b', 'c'],
       ['a', 'a', 'b'],
       ['a', 'b', 'c']],
      dtype='|S1')
```

So the first pixel was in class 'b' in period 1, class 'a' in period 2, and class 'c' in period 3. We can summarize the overall transition dynamics for the set of pixels by treating it as a Markov chain:

```
>>> m = pysal.Markov(c)
>>> m.classes
array(['a', 'b', 'c'],
      dtype='|S1')
```

The Markov instance m has an attribute class extracted from the chain - the assumption is that the observations are on the rows of the input and the different points in time on the columns. In addition to extracting the classes as an attribute, our Markov instance will also have a transitions matrix:

```
>>> m.transitions
array([[ 1.,  2.,  1.],
       [ 1.,  0.,  2.],
       [ 1.,  1.,  1.]])
```

indicating that of the four pixels that began a transition interval in class 'a', 1 remained in that class, 2 transitioned to class 'b' and 1 transitioned to class 'c'.

This simple example illustrates the basic creation of a Markov instance, but the small sample size makes it unrealistic for the more advanced features of this approach. For a larger example, we will look at an application of Markov

[1] Rey, S.J. 2001. "Spatial empirics for economic growth and convergence", 34 Geographical Analysis, 33, 195-214.

methods to understanding regional income dynamics in the US. Here we will load in data on per capita income observed annually from 1929 to 2010 for the lower 48 US states:

```
>>> f = pysal.open("../pysal/examples/usjoin.csv")
>>> pci = np.array([f.by_col[str(y)] for y in range(1929,2010)])
>>> pci.shape
(81, 48)
```

The first row of the array is the per capita income for the first year:

```
>>> pci[0, :]
array([ 323,  600,  310,  991,  634, 1024, 1032,  518,  347,  507,  948,
        607,  581,  532,  393,  414,  601,  768,  906,  790,  599,  286,
        621,  592,  596,  868,  686,  918,  410, 1152,  332,  382,  771,
        455,  668,  772,  874,  271,  426,  378,  479,  551,  634,  434,
        741,  460,  673,  675])
```

In order to apply the classic Markov approach to this series, we first have to discretize the distribution by defining our classes. There are many ways to do this, but here we will use the quintiles for each annual income distribution to define the classes:

```
>>> q5 = np.array([pysal.Quantiles(y).yb for y in pci]).transpose()
>>> q5.shape
(48, 81)
>>> q5[:, 0]
array([0, 2, 0, 4, 2, 4, 4, 1, 0, 1, 4, 2, 2, 1, 0, 1, 2, 3, 4, 4, 2, 0, 2,
       2, 2, 4, 3, 4, 0, 4, 0, 0, 3, 1, 3, 3, 4, 0, 1, 0, 1, 2, 2, 1, 3, 1,
       3, 3])
```

A number of things need to be noted here. First, we are relying on the classification methods in PySAL for defining our quintiles. The class Quantiles uses quintiles as the default and will create an instance of this class that has multiple attributes, the one we are extracting in the first line is yb - the class id for each observation. The second thing to note is the transpose operator which gets our resulting array q5 in the proper structure required for use of Markov. Thus we see that the first spatial unit (Alabama with an income of 323) fell in the first quintile in 1929, while the last unit (Wyoming with an income of 675) fell in the fourth quintile[2].

So now we have a time series for each state of its quintile membership. For example, Colorado's quintile time series is:

```
>>> q5[4, :]
array([2, 3, 2, 2, 3, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 3, 2, 3, 2, 3, 2, 3,
       3, 3, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4,
       4, 4, 4, 4, 4, 3, 3, 3, 4, 3, 3, 3])
```

indicating that it has occupied the 3rd, 4th and 5th quintiles in the distribution at different points in time. To summarize the transition dynamics for all units, we instantiate a Markov object:

```
>>> m5 = pysal.Markov(q5)
>>> m5.transitions
array([[ 729.,   71.,    1.,    0.,    0.],
       [  72.,  567.,   80.,    3.,    0.],
       [   0.,   81.,  631.,   86.,    2.],
       [   0.,    3.,   86.,  573.,   56.],
       [   0.,    0.,    1.,   57.,  741.]])
```

[2] The states are ordered alphabetically.

Assuming we can treat these transitions as a first order Markov chain, we can estimate the transition probabilities:

```
>>> m5.p
matrix([[ 0.91011236,  0.0886392 ,  0.00124844,  0.        ,  0.        ],
        [ 0.09972299,  0.78531856,  0.11080332,  0.00415512,  0.        ],
        [ 0.        ,  0.10125   ,  0.78875   ,  0.1075    ,  0.0025    ],
        [ 0.        ,  0.00417827,  0.11977716,  0.79805014,  0.07799443],
        [ 0.        ,  0.        ,  0.00125156,  0.07133917,  0.92740926]])
```

as well as the long run steady state distribution:

```
>>> m5.steady_state
matrix([[ 0.20774716],
        [ 0.18725774],
        [ 0.20740537],
        [ 0.18821787],
        [ 0.20937187]])
```

With the transition probability matrix in hand, we can estimate the first mean passage time:

```
>>> pysal.ergodic.fmpt(m5.p)
matrix([[   4.81354357,   11.50292712,   29.60921231,   53.38594954,
          103.59816743],
        [  42.04774505,    5.34023324,   18.74455332,   42.50023268,
           92.71316899],
        [  69.25849753,   27.21075248,    4.82147603,   25.27184624,
           75.43305672],
        [  84.90689329,   42.85914824,   17.18082642,    5.31299186,
           51.60953369],
        [  98.41295543,   56.36521038,   30.66046735,   14.21158356,
            4.77619083]])
```

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

Spatial Markov

Thus far we have treated all the spatial units as independent to estimate the transition probabilities. This hides a number of implicit assumptions. First, the transition dynamics are assumed to hold for all units and for all time periods. Second, interactions between the transitions of individual units are ignored. In other words regional context may be important to understand regional income dynamics, but the classic Markov approach is silent on this issue.

PySAL includes a number of spatially explicit extensions to the Markov framework. The first is the spatial Markov class that we illustrate here. We first are going to transform the income series to relative incomes (by standardizing by each period by the mean):

```
>>> import pysal
>>> f = pysal.open("../pysal/examples/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))
```

Next, we require a spatial weights object, and here we will create one from an external GAL file:

```
>>> w = pysal.open("../pysal/examples/states48.gal").read()
>>> w.transform = 'r'
```

Finally, we create an instance of the Spatial Markov class using 5 states for the chain:

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

Here we are keeping the quintiles fixed, meaning the data are pooled over space and time and the quintiles calculated for the pooled data. This is why we first transformed the data to relative incomes. We can next examine the global transition probability matrix for relative incomes:

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

The Spatial Markov allows us to compare the global transition dynamics to those conditioned on regional context. More specifically, the transition dynamics are split across economies who have spatial lags in different quintiles at the beginning of the year. In our example we have 5 classes, so 5 different conditioned transition probability matrices are estimated:

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

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 0.977 if their neighbors are in the 5th quintile, but if their neighbors are in the 4th quintile this drops to 0.903.

We can also explore the different steady state distributions implied by these different transition probabilities:

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

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. And, finally the first mean passage times:

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

States with incomes in the first quintile with neighbors in the first quintile return to the first quintile after 2.298 years, after leaving the first quintile. They enter the fourth quintile 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.

LISA Markov

The Spatial Markov conditions the transitions on the value of the spatial lag for an observation at the beginning of the transition period. An alternative approach to spatial dynamics is to consider the joint transitions of an observation and its spatial lag in the distribution. By exploiting the form of the static *LISA* and embedding it in a dynamic context we develop the LISA Markov in which the states of the chain are defined as the four quadrants in the Moran scatter plot. Continuing on with our US example:

```
>>> import numpy as np
>>> f = pysal.open("../pysal/examples/usjoin.csv")
>>> pci = np.array([f.by_col[str(y)] for y in range(1929, 2010)]).transpose()
>>> w = pysal.open("../pysal/examples/states48.gal").read()
>>> lm = pysal.LISA_Markov(pci, w)
>>> lm.classes
array([1, 2, 3, 4])
```

The LISA transitions are:

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

and the estimated transition probability matrix is:

```
>>> lm.p
matrix([[ 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]])
```

The diagonal elements indicate the staying probabilities and we see that there is greater mobility for observations in quadrants 1 and 3 than 2 and 4.

The implied long run steady state distribution of the chain is

```
>>> lm.steady_state
matrix([[ 0.28561505],
        [ 0.14190226],
        [ 0.40493672],
        [ 0.16754598]])
```

again reflecting the dominance of quadrants 1 and 3 (positive autocorrelation).[3] Finally the first mean passage time for the LISAs is:

```
>>> pysal.ergodic.fmpt(lm.p)
matrix([[  3.50121609,  37.93025465,  40.55772829,  43.17412009],
        [ 31.72800152,   7.04710419,  28.68182751,  49.91485137],
        [ 52.44489385,  47.42097495,   2.46952168,  43.75609676],
        [ 38.76794022,  51.51755827,  26.31568558,   5.96851095]])
```

Rank Based Methods

The second set of spatial dynamic methods in PySAL are based on rank correlations and spatial extensions of the classic rank statistics.

Spatial Rank Correlation

Kendall's τ is based on a comparison of the number of pairs of n observations that have concordant ranks between two variables. For spatial dynamics in PySAL, the two variables in question are the values of an attribute measured at

[3] The complex values of the steady state distribution arise from complex eigenvalues in the transition probability matrix which may indicate cyclicality in the chain.

two points in time over n spatial units. This classic measure of rank correlation indicates how much relative stability there has been in the map pattern over the two periods.

The spatial τ 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.[4] To illustrate this we turn to the case of regional incomes in Mexico over the 1940 to 2010 period:

```
>>> import pysal
>>> f = pysal.open("../pysal/examples/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]))
```

We also introduce the concept of regime weights that defines the neighbor set as those spatial units belonging to the same region. In this example the variable "esquivel99" represents a categorical classification of Mexican states into regions:

```
>>> regime = np.array(f.by_col['esquivel99'])
>>> w = pysal.weights.block_weights(regime)
>>> np.random.seed(12345)
```

Now we will calculate the spatial tau for decade transitions from 1940 through 2000 and report the observed spatial tau against that expected if the rank changes were randomly distributed in space by using 99 permutations:

```
>>> res=[pysal.SpatialTau(y[:,i],y[:,i+1],w,99) for i in range(6)]
>>> for r in res:
...     ev = r.taus.mean()
...     "%8.3f %8.3f %8.3f"%(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'
```

The observed level of spatial concordance during the 1940-50 transition was 0.397 which is significantly lower (p=0.010) than the average level of spatial concordance (0.659) from randomly permuted incomes in Mexico. Similar patterns are found for the next two transition periods as well. In other words the amount of rank concordance is significantly distinct between pairs of observations that are geographical neighbors and those that are not in these first three transition periods. This reflects the greater degree of spatial similarity within rather than between the regimes making the discordant pairs dominated by neighboring pairs.

Rank Decomposition

For a sequence of time periods, θ 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.[4]

```
>>> import pysal
>>> f = pysal.open("../pysal/examples/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]))
```

[4] Rey, S.J. (2004) "Spatial dependence in the evolution of regional income distributions," in A. Getis, J. Mur and H.Zoeller (eds). Spatial Econometrics and Spatial Statistics. Palgrave, London, pp. 194-213.

```
>>> regime = np.array(f.by_col['esquivel99'])
>>> np.random.seed(10)
>>> t = pysal.Theta(y, regime, 999)
>>> t.theta
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])
```

Space-Time Interaction Tests

The third set of spatial dynamic methods in PySAL are global tests of space-time interaction. The purpose of these tests is to detect clustering within space-time event patterns. These patterns are composed of unique events that are labeled with spatial and temporal coordinates. The tests are designed to detect clustering of events in both space and time beyond "any purely spatial or purely temporal clustering"[5], that is, to determine if the events are "interacting." Essentially, the tests examine the dataset to determine if pairs of events closest to each other in space are also those closest to each other in time. The null hypothesis of these tests is that the examined events are distributed randomly in space and time, i.e. the distance between pairs of events in space is independent of the distance in time. Three tests are currently implemented in PySAL: the Knox test, the Mantel test and the Jacquez k Nearest Neighbors test. These tests have been widely applied in epidemiology, criminology and biology. A more in-depth technical review of these methods is available in[6].

Knox Test

The Knox test for space-time interaction employs user-defined critical thresholds in space and time to define proximity between events. All pairs of events are examined to determine if the distance between them in space and time is within the respective thresholds. The Knox statistic is calculated as the total number of event pairs where the spatial and temporal distances separating the pair are within the specified thresholds[7]. If interaction is present, the test statistic will be large. Significance is traditionally established using a Monte Carlo permuation method where event timestamps are permuted and the statistic is recalculated. This procedure is repeated to generate a distribution of statistics which is used to establish the pseudo-significance of the observed test statistic. This approach assumes a static underlying population from which events are drawn. If this is not the case the results may be biased[8].

Formally, the specification of the Knox test is given as:

$$X = \sum_{i}^{n} \sum_{j}^{n} a_{ij}^{s} a_{ij}^{t}$$

$$a_{ij}^{s} = \begin{cases} 1, & \text{if } d_{ij}^{s} < \delta \\ 0, & \text{otherwise} \end{cases}$$

$$a_{ij}^{t} = \begin{cases} 1, & \text{if } d_{ij}^{t} < \tau \\ 0, & \text{otherwise} \end{cases}$$

Where n = number of events, a^s = adjacency in space, a^t = adjacency in time, d^s = distance in space, and d^t = distance in time. Critical space and time distance thresholds are defined as δ and τ, respectively.

[5] Kulldorff, M. (1998). Statistical methods for spatial epidemiology: tests for randomness. In Gatrell, A. and Loytonen, M., editors, GIS and Health, pages 49–62. Taylor & Francis, London.

[6] Tango, T. (2010). Statistical Methods for Disease Clustering. Springer, New York.

[7] Knox, E. (1964). The detection of space-time interactions. Journal of the Royal Statistical Society. Series C (Applied Statistics), 13(1):25–30.

[8] R.D. Baker. (2004). Identifying space-time disease clusters. Acta Tropica, 91(3):291-299.

We illustrate the use of the Knox test using data from a study of Burkitt's Lymphoma in Uganda during the period 1961-75[9]. We start by importing Numpy, PySAL and the interaction module:

```
>>> import numpy as np
>>> import pysal
>>> import pysal.spatial_dynamics.interaction as interaction
>>> np.random.seed(100)
```

The example data are then read in and used to create an instance of SpaceTimeEvents. This reformats the data so the test can be run by PySAL. This class requires the input of a point shapefile. The shapefile must contain a column that includes a timestamp for each point in the dataset. The class requires that the user input a path to an appropriate shapefile and the name of the column containing the timestamp. In this example, the appropriate column name is 'T'.

```
>>> path = "../pysal/examples/burkitt"
>>> events = interaction.SpaceTimeEvents(path,'T')
```

Next, we run the Knox test with distance and time thresholds of 20 and 5,respectively. This counts the events that are closer than 20 units in space, and 5 units in time.

```
>>> result = interaction.knox(events.space, events.t ,delta=20,tau=5,permutations=99)
```

Finally we examine the results. We call the statistic from the results dictionary. This reports that there are 13 events close in both space and time, based on our threshold definitions.

```
>>> print(result['stat'])
13
```

Then we look at the pseudo-significance of this value, calculated by permuting the timestamps and rerunning the statistics. Here, 99 permutations were used, but an alternative number can be specified by the user. In this case, the results indicate that we fail to reject the null hypothesis of no space-time interaction using an alpha value of 0.05.

```
>>> print("%2.2f"%result['pvalue'])
0.17
```

Modified Knox Test

A modification to the Knox test was proposed by Baker[10]. Baker's modification measures the difference between the original observed Knox statistic and its expected value. This difference serves as the test statistic. Again, the significance of this statistic is assessed using a Monte Carlo permutation procedure.

$$T = \frac{1}{2}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij}g_{ij} - \frac{1}{n-1}\sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{j=1}^{n} f_{kj}g_{lj}\right)$$

Where n = number of events, f = adjacency in space, g = adjacency in time (calculated in a manner equivalent to a^s and a^t above in the Knox test). The first part of this statistic is equivalent to the original Knox test, while the second part is the expected value under spatio-temporal randomness.

Here we illustrate the use of the modified Knox test using the data on Burkitt's Lymphoma cases in Uganda from above. We start by importing Numpy, PySAL and the interaction module. Next the example data are then read in and used to create an instance of SpaceTimeEvents.

[9] Kulldorff, M. and Hjalmars, U. (1999). The Knox method and other tests for space- time interaction. Biometrics, 55(2):544–552.

[10] Williams, E., Smith, P., Day, N., Geser, A., Ellice, J., and Tukei, P. (1978). Space-time clustering of Burkitt's lymphoma in the West Nile district of Uganda: 1961-1975. British Journal of Cancer, 37(1):109.

```
>>> import numpy as np
>>> import pysal
>>> import pysal.spatial_dynamics.interaction as interaction
>>> np.random.seed(100)
>>> path = "../pysal/examples/burkitt"
>>> events = interaction.SpaceTimeEvents(path,'T')
```

Next, we run the modified Knox test with distance and time thresholds of 20 and 5,respectively. This counts the events that are closer than 20 units in space, and 5 units in time.

```
>>> result = interaction.modified_knox(events.space, events.t,delta=20,tau=5,
↪permutations=99)
```

Finally we examine the results. We call the statistic from the results dictionary. This reports a statistic value of 2.810160.

```
>>> print("%2.8f"%result['stat'])
2.81016043
```

Next we look at the pseudo-significance of this value, calculated by permuting the timestamps and rerunning the statistics. Here, 99 permutations were used, but an alternative number can be specified by the user. In this case, the results indicate that we fail to reject the null hypothesis of no space-time interaction using an alpha value of 0.05.

```
>>> print("%2.2f"%result['pvalue'])
0.11
```

Mantel Test

Akin to the Knox test in its simplicity, the Mantel test keeps the distance information discarded by the Knox test. The unstandardized Mantel statistic is calculated by summing the product of the spatial and temporal distances between all event pairs[11]. To prevent multiplication by 0 in instances of colocated or simultaneous events, Mantel proposed adding a constant to the distance measurements. Additionally, he suggested a reciprocal transform of the resulting distance measurement to lessen the effect of the larger distances on the product sum. The test is defined formally below:

$$Z = \sum_i^n \sum_j^n (d_{ij}^s + c)^p (d_{ij}^t + c)^p$$

Where, again, d^s and d^t denote distance in space and time, respectively. The constant, c, and the power, p, are parameters set by the user. The default values are 0 and 1, respectively. A standardized version of the Mantel test is implemented here in PySAL, however. The standardized statistic (r) is a measure of correlation between the spatial and temporal distance matrices. This is expressed formally as:

$$r = \frac{1}{n^2 - n - 1} \sum_i^n \sum_j^n \left[\frac{d_{ij}^s - \bar{d}^s}{\sigma_{d^s}} \right] \left[\frac{d_{ij}^t - \bar{d}^t}{\sigma_{d^t}} \right]$$

Where $\bar{d}^s$ refers to the average distance in space, and $\bar{d}^t$ the average distance in time. For notational convenience σ_{d^t} and σ_{d^t} refer to the sample (not population) standard deviations, for distance in space and time, respectively. The same constant and power transformations may also be applied to the spatial and temporal distance matrices employed by the standardized Mantel. Significance is determined through a Monte Carlo permuation approach similar to that employed in the Knox test.

Again, we use the Burkitt's Lymphoma data to illustrate the test. We start with the usual imports and read in the example data.

[11] Mantel, N. (1967). The detection of disease clustering and a generalized regression approach. Cancer Research, 27(2):209–220.

```
>>> import numpy as np
>>> import pysal
>>> import pysal.spatial_dynamics.interaction as interaction
>>> np.random.seed(100)
>>> path = "../pysal/examples/burkitt"
>>> events = interaction.SpaceTimeEvents(path,'T')
```

The following example runs the standardized Mantel test with constants of 0 and transformations of 1, meaning the distance matrices will remain unchanged; however, as recommended by Mantel, a small constant should be added and an inverse transformation (i.e. -1) specified.

```
>>> result = interaction.mantel(events.space, events.t,99,scon=0.0,spow=1.0,tcon=0.0,
↪tpow=1.0)
```

Next, we examine the result of the test.

```
>>> print("%6.6f"%result['stat'])
0.014154
```

Finally, we look at the pseudo-significance of this value, calculated by permuting the timestamps and rerunning the statistic for each of the 99 permuatations. Again, note, the number of permutations can be changed by the user. According to these parameters, the results fail to reject the null hypothesis of no space-time interaction between the events.

```
>>> print("%2.2f"%result['pvalue'])
0.27
```

Jacquez Test

Instead of using a set distance in space and time to determine proximity (like the Knox test) the Jacquez test employs a nearest neighbor distance approach. This allows the test to account for changes in underlying population density. The statistic is calculated as the number of event pairs that are within the set of k nearest neighbors for each other in both space and time[12]. Significance of this count is established using a Monte Carlo permutation method. The test is expressed formally as:

$$J_k = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ijk}^s a_{ijk}^t$$

$$a_{ijk}^s = \begin{cases} 1, & \text{if event } j \text{ is a } k \text{ nearest neighbor of event } i \text{ in space} \\ 0, & \text{otherwise} \end{cases}$$

$$a_{ijk}^t = \begin{cases} 1, & \text{if event } j \text{ is a } k \text{ nearest neighbor of event } i \text{ in time} \\ 0, & \text{otherwise} \end{cases}$$

Where n = number of cases; a^s = adjacency in space; a^t = adjacency in time. To illustrate the test, the Burkitt's Lymphoma data are employed again. We start with the usual imports and read in the example data.

```
>>> import numpy as np
>>> import pysal
>>> import pysal.spatial_dynamics.interaction as interaction
>>> np.random.seed(100)
>>> path = "../pysal/examples/burkitt"
>>> events = interaction.SpaceTimeEvents(path,'T')
```

[12] Jacquez, G. (1996). A k nearest neighbour test for space-time interaction. Statistics in Medicine, 15(18):1935–1949.

The following runs the Jacquez test on the example data for a value of $k = 3$ and reports the resulting statistic. In this case, there are 13 instances where events are nearest neighbors in both space and time. The significance of this can be assessed by calling the p-value from the results dictionary. Again, there is not enough evidence to reject the null hypothesis of no space-time interaction.

```
>>> result = interaction.jacquez(events.space, events.t ,k=3,permutations=99)
>>> print result['stat']
13
>>> print "%3.1f"%result['pvalue']
0.2
```

Spatial Dynamics API

For further details see the *Spatial Dynamics API*.

Using PySAL with Shapely for GIS Operations

New in version 1.3.

Introduction

The Shapely project is a BSD-licensed Python package for manipulation and analysis of planar geometric objects, and depends on the widely used GEOS library.

PySAL supports interoperation with the Shapely library through Shapely's Python Geo Interface. All PySAL geometries provide a __geo_interface__ property which models the geometries as a GeoJSON object. Shapely geometry objects also export the __geo_interface__ property and can be adapted to PySAL geometries using the `pysal.cg.asShape` function.

Additionally, PySAL provides an optional contrib module that handles the conversion between pysal and shapely data strucutures for you. The module can be found in at, `pysal.contrib.shapely_ext`.

Installation

Please refer to the Shapely website for instructions on installing Shapely and its dependencies, *without which PySAL's Shapely extension will not work.*

Usage

Using the Python Geo Interface...

```
>>> import pysal
>>> import shapely.geometry
>>> # The get_path function returns the absolute system path to pysal's
>>> # included example files no matter where they are installed on the system.
>>> fpath = pysal.examples.get_path('stl_hom.shp')
>>> # Now, open the shapefile using pysal's FileIO
>>> shps = pysal.open(fpath , 'r')
>>> # We can read a polygon...
>>> polygon = shps.next()
>>> # To use this polygon with shapely we simply convert it with
>>> # Shapely's asShape method.
>>> polygon = shapely.geometry.asShape(polygon)
>>> # now we can operate on our polygons like normal shapely objects...
>>> print "%.4f"%polygon.area
0.1701
>>> # We can do things like buffering...
>>> eroded_polygon = polygon.buffer(-0.01)
>>> print "%.4f"%eroded_polygon.area
0.1533
>>> # and containment testing...
>>> polygon.contains(eroded_polygon)
True
>>> eroded_polygon.contains(polygon)
False
>>> # To go back to pysal shapes we call pysal.cg.asShape...
>>> eroded_polygon = pysal.cg.asShape(eroded_polygon)
>>> type(eroded_polygon)
<class 'pysal.cg.shapes.Polygon'>
```

Using The PySAL shapely_ext module...

```
>>> import pysal
>>> from pysal.contrib import shapely_ext
>>> fpath = pysal.examples.get_path('stl_hom.shp')
>>> shps = pysal.open(fpath , 'r')
>>> polygon = shps.next()
>>> eroded_polygon = shapely_ext.buffer(polygon, -0.01)
>>> print "%0.4f"%eroded_polygon.area
0.1533
>>> shapely_ext.contains(polygon,eroded_polygon)
True
>>> shapely_ext.contains(eroded_polygon,polygon)
False
>>> type(eroded_polygon)
<class 'pysal.cg.shapes.Polygon'>
```

PySAL: Example Data Sets

PySAL comes with a number of example data sets that are used in some of the documentation strings in the source code. All the example data sets can be found in the **examples** directory.

10740

Polygon shapefile for Albuquerque New Mexico.

- 10740.dbf: attribute database file
- 10740.shp: shapefile
- 10740.shx: spatial index
- 10740_queen.gal: queen contiguity GAL format
- 10740_rook.gal: rook contiguity GAL format

book

Synthetic data to illustrate spatial weights. Source: Anselin, L. and S.J. Rey (in progress) Spatial Econometrics: Foundations.

- book.gal: rook contiguity for regular lattice
- book.txt: attribute data for regular lattice

calempdensity

Employment density for California counties. Source: Anselin, L. and S.J. Rey (in progress) Spatial Econometrics: Foundations.

- calempdensity.csv: data on employment and employment density in California counties.

chicago77

Chicago Community Areas (n=77). Source: Anselin, L. and S.J. Rey (in progress) Spatial Econometrics: Foundations.

- Chicago77.dbf: attribute data
- Chicago77.shp: shapefile
- Chicago77.shx: spatial index

desmith

Example data for autocorrelation analysis. Source: de Smith et al (2009) Geospatial Analysis (Used with permission)

- desmith.txt: attribute data for 10 spatial units
- desmith.gal: spatial weights in GAL format

juvenile

Cardiff juvenile delinquent residences.

- juvenile.dbf: attribute data
- juvenile.html: documentation
- juvenile.shp: shapefile
- juvenile.shx: spatial index
- juvenile.gwt: spatial weights in GWT format

mexico

State regional income Mexican states 1940-2000. Source: Rey, S.J. and M.L. Sastre Gutierrez. “Interregional inequality dynamics in Mexico.” Spatial Economic Analysis. Forthcoming.

- mexico.csv: attribute data
- mexico.gal: spatial weights in GAL format

rook31

Small test shapefile

- rook31.dbf: attribute data
- rook31.gal: spatia weights in GAL format
- rook31.shp: shapefile
- rook31.shx: spatial index

sacramento2

1998 and 2001 Zip Code Business Patterns (Census Bureau) for Sacramento MSA

- sacramento2.dbf
- sacramento2.sbn
- sacramento2.sbx
- sacramento2.shp
- sacramento2.shx

shp_test

Sample Shapefiles used only for testing purposes. Each example include a ".shp" Shapefile, ".shx" Shapefile Index, ".dbf" DBase file, and a ".prj" ESRI Projection file.

Examples include:

- Point: Example of an ESRI Shapefile of Type 1 (Point).
- Line: Example of an ESRI Shapefile of Type 3 (Line).
- Polygon: Example of an ESRI Shapefile of Type 5 (Polygon).

sids2

North Carolina county SIDS death counts and rates

- sids2.dbf: attribute data
- sids2.html: documentation
- sids2.shp: shapefile
- sids2.shx: spatial index
- sids2.gal: GAL file for spatial weights

stl_hom

Homicides and selected socio-economic characteristics for counties surrounding St Louis, MO. Data aggregated for three time periods: 1979-84 (steady decline in homicides), 1984-88 (stable period), and 1988-93 (steady increase in homicides). Source: S. Messner, L. Anselin, D. Hawkins, G. Deane, S. Tolnay, R. Baller (2000). An Atlas of the Spatial Patterning of County-Level Homicide, 1960-1990. Pittsburgh, PA, National Consortium on Violence Research (NCOVR).

- stl_hom.html: Metadata
- stl_hom.txt: txt file with attribute data
- stl_hom.wkt: A Well-Known-Text representation of the geometry.
- stl_hom.csv: attribute data and WKT geometry.
- stl.hom.gal: GAL file for spatial weights

US Regional Incomes

Per capita income for the lower 48 US states, 1929-2010

- us48.shp: shapefile
- us48.dbf: dbf for shapefile
- us48.shx: index for shapefile
- usjoin.csv: attribute data (comma delimited file)

Virginia

Virginia Counties Shapefile.

- virginia.shp: Shapefile
- virginia.shx: shapefile index
- virginia.dbf: attributes
- virginia.prj: shapefile projection

Next Steps with PySAL

The tutorials you have (hopefully) just gone through should be enough to get you going with PySAL. They covered some, but not all, of the modules in PySAL, and at that, only a selection of the functionality of particular classes that were included in the tutorials. To learn more about PySAL you should consult the documentation.

PySAL is an open source, community-based project and we highly value contributions from individuals to the project. There are many ways to contribute, from filing bug reports, suggesting feature requests, helping with documentation, to becoming a developer. Individuals interested in joining the team should send an email to pysal-dev@googlegroups.com or contact one of the developers directly.

CHAPTER 2

Developer Guide

Go to our issues queue on GitHub NOW!

Guidelines

Contents

PySAL is adopting many of the conventions in the larger scientific computing in Python community and we ask that anyone interested in joining the project please review the following documents:

- Documentation standards
- Coding guidelines
- *Testing guidelines*

Open Source Development

PySAL is an open source project and we invite any interested user who wants to contribute to the project to contact one of the team members. For users who are new to open source development you may want to consult the following documents for background information:

- Contributing to Open Source Projects HOWTO

Source Code

PySAL uses git and github for our code repository.

Please see our procedures and policies for development on GitHub as well as how to configure your local git for development.

You can setup PySAL for local development following the *installation instructions*.

Development Mailing List

Development discussions take place on pysal-dev and the gitter room.

Release Schedule

As of version 1.11, PySAL has moved to a rolling release model. Discussions about releases are carried out during the monthly developer meetings and in the gitter room.

Governance

PySAL is organized around the Benevolent Dictator for Life (BDFL) model of project management. The BDFL is responsible for overall project management and direction. Developers have a critical role in shaping that direction. Specific roles and rights are as follows:

Title	Role	Rights
BDFL	Project Director	Commit, Voting, Veto, Developer Approval/Management
Developer	Development	Commit, Voting

Voting and PEPs

During the initial phase of a release cycle, new functionality for PySAL should be described in a PySAL Enhancment Proposal (PEP). These should follow the standard format used by the Python project. For PySAL, the PEP process is as follows

1. Developer prepares a plain text PEP following the guidelines
2. Developer sends PEP to the BDFL
3. Developer posts PEP to the PEP index
4. All developers consider the PEP and vote
5. PEPs receiving a majority approval become priorities for the release cycle

PySAL Testing Procedures

Contents

As of PySAL release 1.6, continuous integration testing was ported to the Travis-CI hosted testing framework (http://travis-ci.org). There is integration within GitHub that provides Travis-CI test results included in a pending Pull Request page, so developers can know before merging a Pull Request that the changes will or will not induce breakage.

Take a moment to read about the Pull Request development model on our wiki at https://github.com/pysal/pysal/wiki/GitHub-Standard-Operating-Procedures

PySAL relies on two different modes of testing [1] integration (regression) testing and [2] doctests. All developers responsible for given packages shall utilize both modes.

Integration Testing

Each package shall have a directory *tests* in which unit test scripts for each module in the package directory are required. For example, in the directory *pysal/esda* the module *moran.py* requires a unittest script named *test_moran.py*. This path for this script needs to be *pysal/esda/tests/test_moran.py*.

To ensure that any changes made to one package/module do not introduce breakage in the wider project, developers should run the package wide test suite using nose before making any commits. As of release version 1.5, all tests must pass using a 64-bit version of Python. To run the new test suite, install nose, nose-progressive, and nose-exclude into your working python installation. If you're using EPD, nose is already available:

```
pip install -U nose
pip install nose-progressive
pip install nose-exclude
```

Then:

```
cd trunk/
nosetests pysal/
```

You can also run the test suite from within a Python session. At the conclusion of the test, Python will, however, exit:

```
import pysal
import nose
nose.runmodule('pysal')
```

The file setup.cfg (added in revision 1050) in trunk holds nose configuration variables. When nosetests is run from trunk, nose reads those configuration parameters into its operation, so developers do not need to specify the optional flags on the command line as shown below.

To specify running just a subset of the tests, you can also run:

```
nosetests pysal/esda/
```

or any other directory, for instance, to run just those tests. To run the entire unittest test suite plus all of the doctests, run:

```
nosetests --with-doctest pysal/
```

To exclude a specific directory or directories, install nose-exclude from PyPi (pip install nose-exclude). Then run it like this:

```
nosetests -v --exclude-dir=pysal/contrib --with-doctest  pysal/
```

Note that you'll probably run into an IOError complaining about too many open files. To fix that, pass this via the command line:

```
ulimit -S -n 1024
```

That changes the machine's open file limit for just the current terminal session.

The trunk should most always be in a state where all tests are passed.

Generating Unit Tests

A useful development companion is the package pythoscope. It scans package folders and produces test script stubs for your modules that fail until you write the tests – a pesky but useful trait. Using pythoscope in the most basic way requires just two simple command line calls:

```
pythoscope --init

pythoscope <my_module>.py
```

One caveat: pythoscope does not name your test classes in a PySAL-friendly way so you'll have to rename each test class after the test scripts are generated. Nose finds tests!

Docstrings and Doctests

All public classes and functions should include examples in their docstrings. Those examples serve two purposes:

1. Documentation for users
2. Tests to ensure code behavior is aligned with the documentation

Doctests will be executed when building PySAL documentation with Sphinx.

Developers *should* run tests manually before committing any changes that may potentially effect usability. Developers can run doctests (docstring tests) manually from the command line using nosetests

```
nosetests --with-doctest pysal/
```

Tutorial Doctests

All of the tutorials are tested along with the overall test suite. Developers can test their changes against the tutorial docstrings by cd'ing into /doc/ and running:

```
make doctest
```

PySAL Enhancement Proposals (PEP)

PEP 0001 Spatial Dynamics Module

Author	Serge Rey <sjsrey@gmail.com>, Xinyue Ye <xinyue.ye@gmail.com>
Status	Approved 1.0
Created	18-Jan-2010
Updated	09-Feb-2010

Abstract

With the increasing availability of spatial longitudinal data sets there is an growing demand for exploratory methods that integrate both the spatial and temporal dimensions of the data. The spatial dynamics module combines a number of previously developed and to-be-developed classes for the analysis of spatial dynamics. It will include classes for the following statistics for spatial dynamics, Markov, spatial Markov, rank mobility, spatial rank mobility, space-time LISA.

Motivation

Rather than having each of the spatial dynamics as separate modules in PySAL, it makes sense to move them all within the same module. This would facilitate common signatures for constructors and similar forms of data structures for space-time analysis (and generation of results).

The module would implement some of the ideas for extending LISA statistics to a dynamic context (*[Anselin2000] [ReyJanikas2006]*), and recent work developing empirics and summary measures for comparative space time analysis (*[ReyYe2010]*).

Reference Implementation

We suggest adding the module `pysal.spatialdynamics` which in turn would encompass the following modules:

- rank mobility rank concordance (relative mobility or internal mixing) Kendall's index
- spatial rank mobility add a spatial dimension into rank mobility investigate the extent to which the relative mobility is spatially dependent use various types of spatial weight matrix
- Markov empirical transition probability matrix (mobility across class) Shorrock's index
- Spatial Markov adds a spatial dimension (regional conditioning) into classic Markov models a trace statistic from a modified Markov transition matrix investigate the extent to which the inter-class mobility are spatially dependent
- Space-Time LISA extends LISA measures to integrate the time dimension combined with cg (computational geometry) module to develop comparative measurements

References

PEP 0002 Residential Segregation Module

Author	David C. Folch <david.folch@asu.edu> Serge Rey <srey@asu.edu>
Status	Draft
Created	10-Feb-2010
Updated	

Abstract

The segregation module combines a number of previously developed and to-be-developed measures for the analysis of residential segregation. It will include classes for two-group and multi-group aspatial (classic) segregation indices along with their spatialized counterparts. Local segregation indices will also be included.

Motivation

The study of residential segregation continues to be a popular field in empirical social science and public policy development. While some of the classic measures are relatively simple to implement, the spatial versions are not nearly as straightforward for the average user. Furthermore, there does not appear to be a Python implementation of residential segregation measures currently available. There is a standalone C#.Net GUI implementation (http://www.ucs.inrs.ca/inc/Groupes/LASER/Segregation.zip) containing many of the measures to be implanted via this PEP but this is Windows only and I could not get it to run easily (it is not open source but the author sent me the code).

It has been noted that there is no one-size-fits-all segregation index; however, some are clearly more popular than others. This module would bring together a wide variety of measures to allow users to easily compare the results from different indices.

Reference Implementation

We suggest adding the module `pysal.segregation` which in turn would encompass the following modules:

- globalSeg
- localSeg

References

PEP 0003 Spatial Smoothing Module

Author	Myunghwa Hwang <mhwang4@gmail.com> Luc Anselin <luc.anselin@asu.edu> Serge Rey <srey@asu.edu>
Status	Approved 1.0
Created	11-Feb-2010
Updated	

Abstract

Spatial smoothing techniques aim to adjust problems with applying simple normalization to rate computation. Geographic studies of disease widely adopt these techniques to better summarize spatial patterns of disease occurrences. The smoothing module combines a number of previously developed and to-be-developed classes for carrying out spatial smoothing. It will include classes for the following techniques: mean and median based smoothing, nonparametric smoothing, and empirical Bayes smoothing.

Motivation

Despite wide usage of spatial smoothing techniques in epidemiology, there are only few software libraries that include a range of different smoothing techniques at one place. Since spatial smoothing is a subtype of exploratory data analysis method, PySAL is the best place that host multiple smoothing techniques.

The smoothing module will mainly implement the techniques reported in [Anselin2006].

Reference Implementation

We suggest adding the module `pysal.esda.smoothing` which in turn would encompass the following modules:

- locally weighted averages, locally weighted median, headbanging
- spatial rate smoothing
- excess risk, empricial Bayes smoothing, spatial empirical Bayes smoothing
- headbanging

References

[Anselin2006] Anselin, L., N. Lozano, and J. Koschinsky (2006) Rate Transformations and Smoothing, GeoDa Center Research Report.

PEP 0004 Geographically Nested Inequality based on the Geary Statistic

Author	Boris Dev <boris.dev@gmail.com> Charles Schmidt <schmidtc@gmail.com>
Status	Draft
Created	9-Aug-2010
Updated	

Abstract

I propose to extend the Geary statistic to describe inequality patterns between people in the same geographic zones. Geographically nested associations can be represented with a spatial weights matrix defined jointly using both geographic and social positions. The key class in the proposed geographically nested inequality module would sub-class from class `pysal.esda.geary` with 2 extensions: 1) as an additional argument, an array of regimes to represent social space; and 2) for the output, spatially nested randomizations will be performed for pseudo-significance tests.

Motivation

Geographically nested measures may reveal inequality patterns that are masked by conventional aggregate approaches. Aggregate human inequality statistics summarize the size of the gaps in variables such as mortality rate or income level between different different groups of people. A geographically nested measure is computed using only a pairwise subset of the values defined by common location in the same geographic zone. For example, this type of measure was proposed in my dissertation to assess changes in income inequality between nearby blocks of different school attendance zones or different racial neighborhoods within the same cities. Since there are no standard statistical packages to do this sort of analysis, currently such a pairwise approach to inequality analysis across many geographic zones is tedious for researchers who are non-hackers. Since it will take advantage of the currently existing `pysal.esda.geary` and `pysal.weights.regime_weights()`, the proposed module should be readable for hackers.

Reference Implementation

I suggest adding the module `pysal.inequality.nested`.

References

[Dev2010] Dev, B. (2010) “Assessing Inequality using Geographic Income Distributions: Spatial Data Analysis of States, Neighborhoods, and School Attendance Zones” http://dl.dropbox.com/u/408103/dissertation.pdf.

PEP 0005 Space Time Event Clustering Module

Author	Nicholas Malizia <nmalizia@gmail.com>, Serge Rey <sjsrey@gmail.com>
Status	Approved 1.1
Created	13-Jul-2010
Updated	06-Oct-2010

Abstract

The space-time event clustering module will be an addition (in the form of a sub-module) to the spatial dynamics module. The purpose of this module will be to house all methods concerned with identifying clusters within spatio-temporal event data. The module will include classes for the major methods for spatio-temporal event clustering, including: the Knox, Mantel, Jacquez k Nearest Neighbors, and the Space-Time K Function. Although these methods are tests of global spatio-temporal clustering, it is our aim to eventually extend this module to include to-be-developed methods for local spatio-temporal clustering.

Motivation

While the methods of the parent module are concerned with the dynamics of aggregate lattice-based data, the methods encompassed in this sub-module will focus on exploring the dynamics of individual events. The methods suggested here have historically been utilized by researchers looking for clusters of events in the fields of epidemiology and criminology. Currently, the methods presented here are not widely implemented in an open source context. Although the Knox, Mantel, and Jacquez methods are available in the commercial, GUI-based software ClusterSeer, they do not appear to be implemented in an open-source context. Also, as they are implemented in ClusterSeer, the methods are not scriptable[1]. The Space-Time K function, however, is available in an open-source context in the `splancs`

[1]

7. Jacquez, D. Greiling, H. Durbeck, L. Estberg, E. Do, A. Long, and B. Rommel. ClusterSeer User Guide 2: Software for Identifying Disease Clusters. Ann Arbor, MI: TerraSeer Press, 2002.

package for R[2]. The combination of these methods in this module would be a unique, scriptable, open-source resource for researchers interested in spatio-temporal interaction of event-based data.

Reference Implementation

We suggest adding the module `pysal.spatialdynamics.events` which in turn would encompass the following modules:

Knox The Knox test for space-time interaction sets critical distances in space and time; if the data are clustered, numerous pairs of events will be located within both of these critical distances and the test statistic will be large[3]. Significance will be established using a Monte Carlo method. This means that either the time stamp or location of the events is scrambled and the statistic is calculated again. This procedure is permuted to generate a distribution of statistics (for the null hypothesis of spatio-temporal randomness) which is used to establish the pseudo-significance of the observed test statistic. Options will be given to specify a range of critical distances for the space and time scales.

Mantel Akin to the Knox test in its simplicity, the Mantel test keeps the distance information discarded by the Knox test. The Mantel statistic is calculated by summing the product of the distances between all the pairs of events[4]. Again, significance will be determined through a Monte Carlo approach.

Jacquez This test tallies the number of event pairs that are within k-nearest neighbors of each other in *both* space and time. Significance of this count is established using a Monte Carlo permutation method[5]. Again, the permutation is done by randomizing either the time or location of the events and then running the statistic again. The test should be implemented with the additional descriptives as suggested by[6].

SpaceTimeK The space-time K function takes the K function which has been used to detect clustering in spatial point patterns and expands it to the realm of spatio-temporal data. Essentially, the method calculates K functions in space and time independently and then compares the product of these functions with a K function which takes both dimensions of space and time into account from the start[7]. Significance is established through Monte Carlo methods and the construction of confidence envelopes.

2

2. Rowlingson and P. Diggle. splancs: Spatial and Space-Time Point Pattern Analysis. R Package. Version 2.01-25, 2009.

3

5. Knox. The detection of space-time interactions. Journal of the Royal Statistical Society. Series C (Applied Statistics), 13(1):25–30, 1964.

4

14. Mantel. The detection of disease clustering and a generalized regression approach. Cancer Research, 27(2):209–220, 1967.

5

7. Jacquez. A k nearest neighbour test for space-time interaction. Statistics in Medicine, 15(18):1935– 1949, 1996.

6

5. Mack and N. Malizia. Enhancing the results of the Jacquez *k* Nearest Neighbor test for space-time interaction. *In Preparation*

7

16. Diggle, A. Chetwynd, R. Haggkvist, and S. Morris. Second-order analysis of space-time clustering. Statistical Methods in Medical Research, 4(2):124, 1995.

References

PEP 0006 Kernel Density Estimation

Author	Serge Rey <sjsrey@gmail.com> Charles Schmidt <schmidtc@gmail.com>
Status	Draft
Created	11-Oct-2010
Updated	11-Oct-2010

Abstract

The kernel density estimation module will provide a uniform interface to a set of kernel density estimation (KDE) methods. Currently KDE is used in various places within PySAL (e.g., `Kernel`, `Kernel_Smoother`) as well as in STARS and various projects within the GeoDA Center, but these implementations were done separately. This module would centralize KDE within PySAL as well as extend the suite of KDE methods and related measures available in PySAL.

Motivation

KDE is widely used throughout spatial analysis, from estimation of process intensity in point pattern analysis, deriving spatial weights, geographically weighted regression, rate smoothing, to hot spot detection, among others.

Reference Implementation

Since KDE would be used throughout existing (and likely future) modules in PySAL, it makes sense to implement it as a top level module in PySAL.

Core KDE methods that would be implemented include:

- triangular
- uniform
- quadratic
- quartic
- gaussian

Additional classes and methods to deal with KDE on restricted spaces would also be implemented.

A unified KDE api would be developed for use of the module.

Computational optimization would form a significant component of the effort for this PEP.

References

in progress

PEP 0007 Spatial Econometrics

Author	Luc Anselin <luc.anselin@asu.edu> Serge Rey <sjsrey@gmail.com>,David Folch <dfolch@asu.edu>,Daniel Arribas-Bel <daniel.arribas.bel@gmail.com>,Pedro Amaral <pvmda2@cam.ac.uk>,Nicholas Malizia <nmalizia@gmail.com>,Ran Wei <rwei5@asu.edu>,Jing Yao <jyao13@asu.edu>,Elizabeth Mack <Elizabeth.A.Mack@asu.edu>
Status	Approved 1.1
Created	12-Oct-2010
Updated	12-Oct-2010

Abstract

The spatial econometrics module will provide a uniform interface to the spatial econometric functionality contained in the former PySpace and current GeoDaSpace efforts. This module would centralize all specification, estimation, diagnostic testing and prediction/simulation for spatial econometric models.

Motivation

Spatial econometric methodology is at the core of GeoDa and GeoDaSpace. This module would allow access to state of the art methods at the source code level.

Reference Implementation

We suggest adding the module `pysal.spreg`. As development progresses, there may be a need for submodules dealing with pure cross sectional regression, spatial panel models and spatial probit.

Core methods to be implemented include:

- OLS estimation with diagnostics for spatial effects
- 2SLS estimation with diagnostics for spatial effects
- spatial 2SLS for spatial lag model (with endogeneity)
- GM and GMM estimation for spatial error model
- GMM spatial error with heteroskedasticity
- spatial HAC estimation

A significant component of the effort for this PEP would consist of implementing methods with good performance on very large data sets, exploiting sparse matrix operations in scipy.

References

[1] Anselin, L. (1988). Spatial Econometrics, Methods and Models. Kluwer, Dordrecht.

[2] Anselin, L. (2006). Spatial econometrics. In Mills, T. and Patterson, K., editors, Palgrave Handbook of Econometrics, Volume I, Econometric Theory, pp. 901-969. Palgrave Macmillan, Basingstoke.

[3] Arraiz, I., Drukker, D., Kelejian H.H., and Prucha, I.R. (2010). A spatial Cliff-Ord-type model with heteroskedastic innovations: small and large sample results. Journal of Regional Science 50: 592-614.

[4] Kelejian, H.H. and Prucha, I.R. (1998). A generalized spatial two stage least squares procedure for estimationg a spatial autoregressive model with autoregressive disturbances. Journal of Real Estate Finance and Economics 17: 99-121.

[5] Kelejian, H.H. and Prucha, I.R. (1999). A generalized moments estimator for the autoregressive parameter in a spatial model. International Economic Review 40: 509-533.

[6] Kelejian, H.H. and Prucha, I.R. (2007). HAC estimation in a spatial framework. Journal of Econometrics 140: 131-154.

[7] Kelejian, H.H. and Prucha, I.R. (2010). Specification and estimation of spatial autoregressive models with autoregressive and heteroskedastic disturbances. Journal of Econometrics (forthcoming).

PEP 0008 Spatial Database Module

Author	Phil Stephens <phil.stphns@gmail.com>, Serge Rey <sjsrey@gmail.com>
Status	Draft
Created	09-Sep-2010
Updated	31-Aug-2012

Abstract

A spatial database module will extend PySAL file I/O capabilities to spatial database software, allowing PySAL users to connect to and perform geographic lookups and queries on spatial databases.

Motivation

PySAL currently reads and writes geometry in only the Shapefile data structure. Spatially-indexed databases permit queries on the geometric relations between objects[1].

Reference Implementation

We propose to add the module `pysal.contrib.spatialdb`, hereafter referred to simply as spatialdb. spatialdb will leverage the Python Object Relational Mapper (ORM) libraries SQLAlchemy[2] and GeoAlchemy[3], MIT-licensed software that provides a database-agnostic SQL layer for several different databases and spatial database extensions including PostgreSQL/PostGIS, Oracle Spatial, Spatialite, MS SQL Server, MySQL Spatial, and others. These lightweight libraries manage database connections, transactions, and SQL expression translation.

Another option to research is the GeoDjango package. It provides a large number of spatial lookups[5] and geo queries for PostGIS databases, and a smaller set of lookups / queries for Oracle, MySQL, and SpatiaLite.

[1] OpenGeo (2010) Spatial Database Tips and Tricks. Accessed September 9, 2010.
[2] SQLAlchemy (2010) SQLAlchemy 0.6.5 Documentation. Accessed October 4, 2010.
[3] GeoAlchemy (2010) GeoAlchemy 0.4.1 Documentation. Accessed October 4, 2010.
[5] GeoDjango (2012) GeoDjango Compatibility Tables. Accessed August 31, 2012.

References

PEP 0009 Add Python 3.x Support

Author	Charles Schmidt <schmidtc@gmail.com>
Status	Approved 1.2
Created	02-Feb-2011
Updated	02-Feb-2011

Abstract

Python 2.x is being phased out in favor of the backwards incompatible Python 3 line. In order to stay relevant to the python community as a whole PySAL needs to support the latest production releases of Python. With the release of Numpy 1.5 and the pending release of SciPy 0.9, all PySAL dependencies support Python 3. This PEP proposes porting the code base to support both the 2.x and 3.x lines of Python.

Motivation

Python 2.7 is the final major release in the 2.x line. The Python 2.x line will continue to receive bug fixes, however only the 3.x line will receive new features (*[Python271]*). Python 3.x introduces many backward incompatible changes to Python (*[PythonNewIn3]*). Numpy added support for Python 3.0 in version 1.5 (*[NumpyANN150]*). Scipy 0.9.0 is currently in the release candidate stage and supports Python 3.0 (*[SciPyRoadmap]*, *[SciPyANN090rc2]*). Many of the new features in Python 2.7 were back ported from 3.0, allowing us to start using some of the new feature of the language without abandoning our 2.x users.

Reference Implementation

Since python 2.6 the interpreter has included a '-3' command line switch to "warn about Python 3.x incompatibilities that 2to3 cannot trivially fix" (*[Python2to3]*). Running PySAL tests with this switch produces no warnings internal to PySAL. This suggests porting to 3.x will require only trivial changes to the code. A porting strategy is provided by *[PythonNewIn3]*.

References

PEP 0010 Add pure Python rtree

Author	Serge Rey <sjsrey@gmail.com>
Status	Approved 1.2
Created	12-Feb-2011
Updated	12-Feb-2011

Abstract

A pure Python implementation of an Rtree will be developed for use in the construction of spatial weights matrices based on contiguity relations in shapefiles as well as supporting a spatial index that can be used by GUI based applications built with PySAL requiring brushing and linking.

Motivation

As of 1.1 PySAL checks if the external library (*[Rtree]*) is installed. If it is not, then an internal binning algorithm is used to determine contiguity relations in shapefiles for the construction of certain spatial weights. A pure Python implementation of Rtrees may provide for improved cross-platform efficiency when the external Rtree library is not present. At the same time, such an implementation can be relied on by application developers using PySAL who wish to build visualization applications supporting brushing, linking and other interactions requiring spatial indices for object selection.

Reference Implementation

A pure Python implementation of Rtrees has recently been implemented (*[pyrtree]*) and is undergoing testing for possible inclusion in PySAL. It appears that this module can be integrated into PySAL with modest effort.

References

PEP 0011 Move from Google Code to Github

Author	Serge Rey <sjsrey@gmail.com>
Status	Draft
Created	04-Aug-2012
Updated	04-Aug-2012

Abstract

This proposal is to move the PySAL code repository from Google Code to Github.

Motivation

Git is a decentralized version control system that brings a number of benefits:

- distributed development
- off-line development
- elegant and lightweight branching
- fast operations
- flexible workflows

among many others.

The two main PySAL dependencies, SciPy and NumPy, made the switch to GitHub roughly two years ago. In discussions with members of those development teams and related projects (pandas, statsmodels) it is clear that git is gaining widespread adoption in the Python scientific computing community. By moving to git and GitHub, PySAL would benefit by facilitating interaction with developers in this community. Discussions with developers at SciPy 2012 indicated that all projects experienced significant growth in community involvement after the move to Github. Other projects considering such a move have been discussing similar issues.

Moving to GitHub would also streamline the administration of project updates, documentation and related tasks. The Google Code infrastructure requires updates in multiple locations which results in either additional work, or neglected changes during releases. GitHub understands markdown and reStructured text formats, the latter is heavily used in PySAL documentation and the former is clearly preferred to wiki markup on Google Code.

Although there is a learning curve to Git, it is relatively minor for developers familiar with Subversion, as all PySAL developers are. Moreover, several of the developers have been using Git and GitHub for other projects and have expressed interest in such a move. There are excellent on-line resources for learning more about git, such as this book.

Reference Implementation

Moving code and history

There are utilities, such as svn2git that can be used to convert an SVN repo to a git repo.

The converted git repo would then be pushed to a GitHub account.

Setting up post-(commit|push|pull) hooks

Migration of the current integration testing will be required. Github has support for Post-Receive Hooks that can be used for this aspect of the migration.

Moving issues tracking over

A decision about whether to move the issue tracking over to Github will have to be considered. This has been handled in different ways:

- keep using Google Code for issue tracking
- move all issues (even closed ones) over to Github
- freeze tickets at Google Code and have a breadcrumb for active tickets pointing to issue tracker at Github

If we decide to move the issues over we may look at tratihubus as well as other possibilities.

Continuous integration with travis-ci

Travis-CI is a hosted Continuous Integration (CI) service that is integrated with GitHub. This sponsored service provides:

- testing with multiple versions of Python
- testing with multiple versions of project dependencies (numpy and scipy)
- build history
- integrated GitHub commit hooks
- testing against multiple database services

Configuration is achieved with a single YAML file, reducing development overhead, maintenance, and monitoring.

Code Sprint for GitHub migration

The proposal is to organize a future sprint to focus on this migration.

PySAL Documentation

Contents

Writing Documentation

The PySAL project contains two distinct forms of documentation: inline and non-inline. Inline docs are contained in the source code itself, in what are known as *docstrings*. Non-inline documentation is in the doc folder in the trunk.

Inline documentation is processed with an extension to Sphinx called napoleon. We have adopted the community standard outlined here.

PySAL makes use of the built-in Sphinx extension *viewcode*, which allows the reader to quicky toggle between docs and source code. To use it, the source code module requires at least one properly formatted docstring.

Non-inline documentation editors can opt to strike-through older documentation rather than delete it with the custom "role" directive as follows. Near the top of the document, add the role directive. Then, to strike through old text, add the :strike: directive and offset the text with back-ticks. This strikethrough is produced like this:

```
.. role:: strike

...
...

This :strike:`strikethrough` is produced like this:
```

Compiling Documentation

PySAL documentation is built using Sphinx and the Sphinx extension napoleon, which formats PySAL's docstrings.

Note

If you're using Sphinx version 1.3 or newer, napoleon is included and should be called in the main conf.py as sphinx.ext.napoleon rather than installing it as we show below.

If you're using a version of Sphinx that does not ship with napoleon (Sphinx < 1.3), you'll need napoleon version 0.2.4 or later and Sphinx version 1.0 or later to compile the documentation. Both modules are available at the Python Package Index, and can be downloaded and installed from the command line using *pip* or *easy_install*.:

```
$ easy_install sphinx
$ easy_install sphinxcontrib-napoleon
```

or

> $ pip sphinx $ pip sphinxcontrib-napoleon

If you get a permission error, trying using 'sudo'.

The source for the docs is in *doc*. Building the documentation is done as follows (assuming sphinx and napoleon are already installed):

```
$ cd doc; ls
build  Makefile  source

$ make clean
$ make html
```

To see the results in a browser open *build/html/index.html*. To make changes, edit (or add) the relevant files in *source* and rebuild the docs using the 'make html' (or 'make clean' if you're adding new documents) command. Consult the Sphinx markup guide for details on the syntax and structure of the files in *source*.

Once you're happy with your changes, check-in the *source* files. Do not add or check-in files under *build* since they are dynamically built.

Changes checked in to Github will be propogated to readthedocs within a few minutes.

Lightweight Editing with rst2html.py

Because the doc build process can sometimes be lengthy, you may want to avoid having to do a full build until after you are done with your major edits on one particular document. As part of the docutils package, the file *rs2html.py* can take an *rst* document and generate the html file. This will get most of the work done that you need to get a sense if your edits are good, *without* having to rebuild all the PySAL docs. As of version 0.8 it also understands LaTeX. It will cough on some sphinx directives, but those can be dealt with in the final build.

To use this download the doctutils tarball and put *rst2html.py* somewhere in your path. In vim (on Mac OS X) you can then add something like:

```
map ;r ^[:!rst2html.py % > ~/tmp/tmp.html; open ~/tmp/tmp.html^M^M
```

which will render the html in your default browser.

Things to watch out for

If you encounter a failing tutorial doctest that does not seem to be in error, it could be a difference in whitespace between the expected and received output. In that case, add an 'options' line as follows:

```
.. doctest::
   :options: +NORMALIZE_WHITESPACE

   >>> print 'a   b   c'
   abc
```

Adding a new package and modules

To include the docstrings of a new module in the *API docs* the following steps are required:

1. In the directory */doc/source/library* add a directory with the name of the new package. You can skip to step 3 if the package exists and you are just adding new modules to this package.
2. Within */doc/source/library/packageName* add a file *index.rst*
3. For each new module in this package, add a file *moduleName.rst* and update the *index.rst* file to include *modulename*.

Adding a new tutorial: spreg

While the *API docs* are automatically generated when compiling with Sphinx, tutorials that demonstrate use cases for new modules need to be crafted by the developer. Below we use the case of one particular module that currently does not have a tutorial as a guide for how to add tutorials for new modules.

As of PySAL 1.3 there are API docs for *spreg* but no *tutorial* currently exists for this module.

We will fix this and add a tutorial for *spreg*.

Requirements

- sphinx
- napoleon
- pysal sources

You can install *sphinx* or *napoleon* using *easy_install* as described above in *Writing Documentation*.

Where to add the tutorial content

Within the PySAL source the docs live in:

```
pysal/doc/source
```

This directory has the source reStructuredText files used to render the html pages. The tutorial pages live under:

```
pysal/doc/source/users/tutorials
```

As of PySAL 1.3, the content of this directory is:

```
autocorrelation.rst  fileio.rst  next.rst     smoothing.rst
dynamics.rst         index.rst   region.rst   weights.rst
examples.rst         intro.rst   shapely.rst
```

The body of the *index.rst* file lists the sections for the tutorials:

```
Introduction to the Tutorials <intro>
File Input and Output <fileio>
Spatial Weights <weights>
Spatial Autocorrelation <autocorrelation>
Spatial Smoothing <smoothing>
Regionalization <region>
Spatial Dynamics <dynamics>
```

```
Shapely Extension <shapely>
Next Steps <next>
Sample Datasets <examples>
```

In order to add a tutorial for *spreg* we need the to change this to read:

```
Introduction to the Tutorials <intro>
File Input and Output <fileio>
Spatial Weights <weights>
Spatial Autocorrelation <autocorrelation>
Spatial Smoothing <smoothing>
Spatial Regression <spreg>
Regionalization <region>
Spatial Dynamics <dynamics>
Shapely Extension <shapely>
Next Steps <next>
Sample Datasets <examples>
```

So we are adding a new section that will show up as *Spatial Regression* and its contents will be found in the file *spreg.rst*. To create the latter file simpy copy say *dynamics.rst* to *spreg.rst* and then modify *spreg.rst* to have the correct content.

Once this is done, move back up to the top level doc directory:

```
pysal/doc
```

Then:

```
$ make clean
$ make html
```

Point your browser to *pysal/doc/build/html/index.html*

and check your work. You can then make changes to the *spreg.rst* file and recompile until you are set with the content.

Proper Reference Formatting

For proper hypertext linking of reference material, each unique reference in a single python module can only be explicitly named once. Take the following example for instance:

```
References
----------

.. [1] Kelejian, H.R., Prucha, I.R. (1998) "A generalized spatial
two-stage least squares procedure for estimating a spatial autoregressive
model with autoregressive disturbances". The Journal of Real State
Finance and Economics, 17, 1.
```

It is "named" as "1". Any other references (even the same paper) with the same "name" will cause a Duplicate Reference error when Sphinx compiles the document. Several work-arounds are available but no concensus has emerged.

One possible solution is to use an anonymous reference on any subsequent duplicates, signified by a single underscore with no brackets. Another solution is to put all document references together at the bottom of the document, rather than listing them at the bottom of each class, as has been done in some modules.

PySAL Release Management

Contents

Prepare the release

- Create a branch out of the *dev* branch
- Check all tests pass. See *PySAL Testing Procedures*.
- Update CHANGELOG:

```
$ python tools/github_stats.py days >> chglog
```

- where *days* is the number of days to start the logs at
- Prepend *chglog* to *CHANGELOG* and edit
- Edit THANKS and README and README.md if needed.
- Edit the file *version.py* to update MAJOR, MINOR, MICRO
- Bump:

```
$ cd tools; python bump.py
```

- Commit all changes.
- Push your branch up to your GitHub repos
- On github issue a pull request, with a target of **upstream master**. Add a comment that this is for release.

Tag

With version 1.10 we changed the way the tags are created to reflect our policy of not pushing directly to upstream.

After you have issued a pull request, the project maintainer can create the release on GitHub.

Make docs

As of version 1.6, docs are automatically compiled and hosted.

Make and Upload and Test Distributions

On each build machine, clone and checkout the newly created tag (assuming that is *v1.10* in what follows):

```
$ git clone http://github.com/pysal/pysal.git
$ cd pysal
$ git fetch --tags
$ git checkout v1.10
```

- Make and upload to the **Testing** Python Package Index:

  ```
  $ python setup.py sdist upload -r https://testpypi.python.org/pypi
  ```

- Test that your package can install correctly:

  ```
  $ pip install --extra-index-url https://testpypi.python.org/pypi pysal
  ```

If all is good, proceed, otherwise fix, and repeat.

- Make and upload to the Python Package Index in one shot!:

  ```
  $ python setup.py sdist  (to test it)
  $ python setup.py sdist upload
  ```

  – **if not registered, do so. Follow the prompts. You can save the** login credentials in a dot-file, .pypirc

- Make and upload the Windows installer to SourceForge. - On a Windows box, build the installer as so:

  ```
  $ python setup.py bdist_wininst
  ```

Announce

- Draft and distribute press release on geodacenter.asu.edu, openspace-list, and pysal.org
  - On GeoDa center website, do this:
    - Login and expand the wrench icon to reveal the Admin menu
    - Click “Administer”, “Content Management”, “Content”
    - Next, click “List”, filter by type, and select “Featured Project”.
    - Click “Filter”

    Now you will see the list of Featured Projects. Find “PySAL”.

    - Choose to ‘edit’ PySAL and modify the short text there. This changes the text users see on the homepage slider.
    - Clicking on the name “PySAL” allows you to edit the content of the PySAL project page, which is also the “About PySAL” page linked to from the homepage slider.

Put master back to dev

- Change MAJOR, MINOR version in setup script.
- Change pysal/version.py to dev number
- Change the docs version from X.x to X.xdev by editing doc/source/conf.py in two places.

- Update the release schedule in doc/source/developers/guidelines.rst

Update the github.io news page to announce the release.

PySAL and Python3

Contents

Starting with version 1.11, PySAL supports both the Python 2.x series (version 2.6 and 2.7) as well as Python 3.4 and newer.

Projects Using PySAL

This page lists other software projects making use of PySAL. If your project is not listed here, contact one of the team members and we'll add it.

GeoDa Center Projects

- GeoDaNet
- GeoDaSpace
- GeoDaWeights
- STARS

Related Projects

- Anaconda
- StatsModels
- PythonAnywhere includes latest PySAL release
- CartoDB includes PySAL through a custom PostgreSQL extension

Known Issues

1.5 install fails with scipy 11.0 on Mac OS X

Running *python setup.py install* results in:

```
from _cephes import *
ImportError:
dlopen(/Users/serge/Documents/p/pysal/virtualenvs/python1.5/lib/python2.7/site-
↪packages/scipy/special/_cephes.so,
2): Symbol not found: _aswfa_
```

```
Referenced from:
/Users/serge/Documents/p/pysal/virtualenvs/python1.5/lib/python2.7/site-packages/
↪scipy/special/_cephes.so
  Expected in: dynamic lookup
```

This occurs when your scipy on Mac OS X was complied with gnu95 and not gfortran. See this thread for possible solutions.

weights.DistanceBand failing

This occurs due to a bug in scipy.sparse prior to version 0.8. If you are running such a version see Issue 73 for a fix.

doc tests and unit tests under Linux

Some Linux machines return different results for the unit and doc tests. We suspect this has to do with the way random seeds are set. See Issue 52

LISA Markov missing a transpose

In versions of PySAL < 1.1 there is a bug in the LISA Markov, resulting in incorrect values. For a fix and more details see Issue 115.

PIP Install Fails

Having numpy and scipy specified in pip requiretments.txt causes PIP install of pysal to fail. For discussion and suggested fixes see Issue 207.

CHAPTER 3

Library Reference

Release 1.13.0

Date Mar 15, 2017

Python Spatial Analysis Library

The Python Spatial Analysis Library consists of several sub-packages each addressing a different area of spatial analysis. In addition to these sub-packages PySAL includes some general utilities used across all modules.

Sub-packages

`pysal.cg` – Computational Geometry

`cg.locators` — Locators

The `cg.locators` module provides ....

New in version 1.0. Computational geometry code for PySAL: Python Spatial Analysis Library.

class `pysal.cg.locators.`**`IntervalTree`**(*intervals*)

Representation of an interval tree. An interval tree is a data structure which is used to quickly determine which intervals in a set contain a value or overlap with a query interval. [DeBerg2008]

`query`(*q*)

Returns the intervals intersected by a value or interval.

query((number, number) or number) -> x list

Parameters **q** (*a value or interval to find intervals intersecting*) –

Examples

```
>>> intervals = [(-1, 2, 'A'), (5, 9, 'B'), (3, 6, 'C')]
>>> it = IntervalTree(intervals)
>>> it.query((7, 14))
['B']
>>> it.query(1)
['A']
```

class `pysal.cg.locators.`**`Grid`**(*bounds*, *resolution*)

Representation of a binning data structure.

`add`(*item*, *pt*)

Adds an item to the grid at a specified location.

add(x, Point) -> x

Parameters

- **item** (*the item to insert into the grid*) –
- **pt** (*the location to insert the item at*) –

Examples

```
>>> g = Grid(Rectangle(0, 0, 10, 10), 1)
>>> g.add('A', Point((4.2, 8.7)))
'A'
```

`bounds`(*bounds*)

Returns a list of items found in the grid within the bounds specified.

bounds(Rectangle) -> x list

Parameters

- **item** (*the item to remove from the grid*) –
- **pt** (*the location the item was added at*) –

Examples

```
>>> g = Grid(Rectangle(0, 0, 10, 10), 1)
>>> g.add('A', Point((1.0, 1.0)))
'A'
>>> g.add('B', Point((4.0, 4.0)))
'B'
>>> g.bounds(Rectangle(0, 0, 3, 3))
['A']
>>> g.bounds(Rectangle(2, 2, 5, 5))
['B']
>>> sorted(g.bounds(Rectangle(0, 0, 5, 5)))
['A', 'B']
```

`in_grid`(*loc*)

Returns whether a 2-tuple location _loc_ lies inside the grid bounds.

Test tag: <tc>#is#Grid.in_grid</tc>

nearest (*pt*)
Returns the nearest item to a point.

nearest(Point) -> x

Parameters pt (*the location to search near*) –

Examples

```
>>> g = Grid(Rectangle(0, 0, 10, 10), 1)
>>> g.add('A', Point((1.0, 1.0)))
'A'
>>> g.add('B', Point((4.0, 4.0)))
'B'
>>> g.nearest(Point((2.0, 1.0)))
'A'
>>> g.nearest(Point((7.0, 5.0)))
'B'
```

proximity (*pt*, *r*)
Returns a list of items found in the grid within a specified distance of a point.

proximity(Point, number) -> x list

Parameters

- **pt** (*the location to search around*) –
- **r** (*the distance to search around the point*) –

Examples

```
>>> g = Grid(Rectangle(0, 0, 10, 10), 1)
>>> g.add('A', Point((1.0, 1.0)))
'A'
>>> g.add('B', Point((4.0, 4.0)))
'B'
>>> g.proximity(Point((2.0, 1.0)), 2)
['A']
>>> g.proximity(Point((6.0, 5.0)), 3.0)
['B']
>>> sorted(g.proximity(Point((4.0, 1.0)), 4.0))
['A', 'B']
```

remove (*item*, *pt*)
Removes an item from the grid at a specified location.

remove(x, Point) -> x

Parameters

- **item** (*the item to remove from the grid*) –
- **pt** (*the location the item was added at*) –

Examples

```
>>> g = Grid(Rectangle(0, 0, 10, 10), 1)
>>> g.add('A', Point((4.2, 8.7)))
'A'
>>> g.remove('A', Point((4.2, 8.7)))
'A'
```

class `pysal.cg.locators.`**`BruteForcePointLocator`**(*points*)

A class which does naive linear search on a set of Point objects.

`nearest`(*query_point*)

Returns the nearest point indexed to a query point.

nearest(Point) -> Point

Parameters **`query_point`** (*`a point to find the nearest indexed point to`*) –

Examples

```
>>> points = [Point((0, 0)), Point((1, 6)), Point((5.4, 1.4))]
>>> pl = BruteForcePointLocator(points)
>>> n = pl.nearest(Point((1, 1)))
>>> str(n)
'(0.0, 0.0)'
```

`proximity`(*origin, r*)

Returns the indexed points located within some distance of an origin point.

proximity(Point, number) -> Point list

Parameters

- **`origin`** (*`the point to find indexed points near`*) –
- **`r`** (*`the maximum distance to find indexed point from the origin point`*) –

Examples

```
>>> points = [Point((0, 0)), Point((1, 6)), Point((5.4, 1.4))]
>>> pl = BruteForcePointLocator(points)
>>> neighs = pl.proximity(Point((1, 0)), 2)
>>> len(neighs)
1
>>> p = neighs[0]
>>> isinstance(p, Point)
True
>>> str(p)
'(0.0, 0.0)'
```

`region`(*region_rect*)

Returns the indexed points located inside a rectangular query region.

region(Rectangle) -> Point list

Parameters **region_rect** (*the rectangular range to find indexed points in*) –

Examples

```
>>> points = [Point((0, 0)), Point((1, 6)), Point((5.4, 1.4))]
>>> pl = BruteForcePointLocator(points)
>>> pts = pl.region(Rectangle(-1, -1, 10, 10))
>>> len(pts)
3
```

class `pysal.cg.locators.`**`PointLocator`**(*points*)

An abstract representation of a point indexing data structure.

`nearest`(*query_point*)

Returns the nearest point indexed to a query point.

nearest(Point) -> Point

Parameters **query_point** (*a point to find the nearest indexed point to*) –

Examples

```
>>> points = [Point((0, 0)), Point((1, 6)), Point((5.4, 1.4))]
>>> pl = PointLocator(points)
>>> n = pl.nearest(Point((1, 1)))
>>> str(n)
'(0.0, 0.0)'
```

`overlapping`(*region_rect*)

Returns the indexed points located inside a rectangular query region.

region(Rectangle) -> Point list

Parameters **region_rect** (*the rectangular range to find indexed points in*) –

Examples

```
>>> points = [Point((0, 0)), Point((1, 6)), Point((5.4, 1.4))]
>>> pl = PointLocator(points)
>>> pts = pl.region(Rectangle(-1, -1, 10, 10))
>>> len(pts)
3
```

`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.

proximity(Point, number) -> Point list

Parameters

- **origin** (*the point to find indexed points near*) –
- **r** (*the maximum distance to find indexed point from the origin point*) –

Examples

```
>>> points = [Point((0, 0)), Point((1, 6)), Point((5.4, 1.4))]
>>> pl = PointLocator(points)
>>> len(pl.proximity(Point((1, 0)), 2))
1
```

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

region(Rectangle) -> Point list

Parameters region_rect (*the rectangular range to find indexed points in*) –

Examples

```
>>> points = [Point((0, 0)), Point((1, 6)), Point((5.4, 1.4))]
>>> pl = PointLocator(points)
>>> pts = pl.region(Rectangle(-1, -1, 10, 10))
>>> len(pts)
3
```

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

contains_point (*point*)
Returns polygons that contain point

Parameters point (*point (x,y)*) –

Returns

Return type list of polygons containing point

Examples

```
>>> p1 = Polygon([Point((0,0)), Point((6,0)), Point((4,4))])
>>> p2 = Polygon([Point((1,2)), Point((4,0)), Point((4,4))])
>>> p1.contains_point((2,2))
1
>>> p2.contains_point((2,2))
1
>>> pl = PolygonLocator([p1, p2])
>>> len(pl.contains_point((2,2)))
2
>>> p2.contains_point((1,1))
0
>>> p1.contains_point((1,1))
1
```

```
>>> len(pl.contains_point((1,1)))
1
>>> p1.centroid
(3.3333333333333335, 1.3333333333333333)
>>> pl.contains_point((1,1))[0].centroid
(3.3333333333333335, 1.3333333333333333)
```

inside (*query_rectangle*)
Returns polygons that are inside query_rectangle

Examples

```
>>> p1 = Polygon([Point((0, 1)), Point((4, 5)), Point((5, 1))])
>>> p2 = Polygon([Point((3, 9)), Point((6, 7)), Point((1, 1))])
>>> p3 = Polygon([Point((7, 1)), Point((8, 7)), Point((9, 1))])
>>> pl = PolygonLocator([p1, p2, p3])
>>> qr = Rectangle(0, 0, 5, 5)
>>> res = pl.inside( qr )
>>> len(res)
1
>>> qr = Rectangle(3, 7, 5, 8)
>>> res = pl.inside( qr )
>>> len(res)
0
>>> qr = Rectangle(10, 10, 12, 12)
>>> res = pl.inside( qr )
>>> len(res)
0
>>> qr = Rectangle(0, 0, 12, 12)
>>> res = pl.inside( qr )
>>> len(res)
3
```

Notes

inside means the intersection of the query rectangle and a polygon is not empty and is equal to the area of the polygon

nearest (*query_point*, *rule='vertex'*)
Returns the nearest polygon indexed to a query point based on various rules.

nearest(Polygon) -> Polygon

Parameters

- **query_point** (*a point to find the nearest indexed polygon to*) –
- **rule** (*representative point for polygon in nearest query.*) – vertex – measures distance between vertices and query_point centroid – measures distance between centroid and query_point edge – measures the distance between edges and query_point

Examples

```
>>> 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])
>>> try: n = pl.nearest(Point((-1, 1)))
... except NotImplementedError: print "future test: str(min(n.vertices())) ==␣
↪(0.0, 1.0)"
future test: str(min(n.vertices())) == (0.0, 1.0)
```

overlapping(*query_rectangle*)
Returns list of polygons that overlap query_rectangle

Examples

```
>>> p1 = Polygon([Point((0, 1)), Point((4, 5)), Point((5, 1))])
>>> p2 = Polygon([Point((3, 9)), Point((6, 7)), Point((1, 1))])
>>> p3 = Polygon([Point((7, 1)), Point((8, 7)), Point((9, 1))])
>>> pl = PolygonLocator([p1, p2, p3])
>>> qr = Rectangle(0, 0, 5, 5)
>>> res = pl.overlapping( qr )
>>> len(res)
2
>>> qr = Rectangle(3, 7, 5, 8)
>>> res = pl.overlapping( qr )
>>> len(res)
1
>>> qr = Rectangle(10, 10, 12, 12)
>>> res = pl.overlapping( qr )
>>> len(res)
0
>>> qr = Rectangle(0, 0, 12, 12)
>>> res = pl.overlapping( qr )
>>> len(res)
3
>>> qr = Rectangle(8, 3, 9, 4)
>>> p1 = Polygon([Point((2, 1)), Point((2, 3)), Point((4, 3)), Point((4,1))])
>>> p2 = Polygon([Point((7, 1)), Point((7, 5)), Point((10, 5)), Point((10,␣
↪1))])
>>> pl = PolygonLocator([p1, p2])
>>> res = pl.overlapping(qr)
>>> len(res)
1
```

Notes

overlapping means the intersection of the query rectangle and a polygon is not empty and is no larger than the area of the polygon

proximity(*origin*, *r*, *rule='vertex'*)
Returns the indexed polygons located within some distance of an origin point based on various rules.

proximity(Polygon, number) -> Polygon list

Parameters

- **origin** (*the point to find indexed polygons near*) –
- **r** (*the maximum distance to find indexed polygon from the origin point*) –
- **rule** (*representative point for polygon in nearest query.*) – vertex – measures distance between vertices and query_point centroid – measures distance between centroid and query_point edge – measures the distance between edges and query_point

Examples

```
>>> 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])
>>> try:
...     len(pl.proximity(Point((0, 0)), 2))
... except NotImplementedError:
...     print "future test: len(pl.proximity(Point((0, 0)), 2)) == 2"
future test: len(pl.proximity(Point((0, 0)), 2)) == 2
```

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

region(Rectangle) -> Polygon list

Parameters **region_rect** (*the rectangular range to find indexed polygons in*) –

Examples

```
>>> 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])
>>> n = pl.region(Rectangle(0, 0, 4, 10))
>>> len(n)
2
```

cg.shapes — Shapes

The cg.shapes module provides basic data structures.

New in version 1.0. Computational geometry code for PySAL: Python Spatial Analysis Library.

class pysal.cg.shapes.**Point** (*loc*)
Geometric class for point objects.

None

class pysal.cg.shapes.**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

p1
Point – Starting point

p2
Point – Ending point

bounding_box
tuple – The bounding box of the segment (number 4-tuple)

len
float – The length of the segment

line
Line – The line on which the segment lies

bounding_box
Returns the minimum bounding box of a LineSegment object.

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

bounding_box -> Rectangle

Examples

```
>>> ls = LineSegment(Point((1, 2)), Point((5, 6)))
>>> ls.bounding_box.left
1.0
>>> ls.bounding_box.lower
2.0
>>> ls.bounding_box.right
5.0
>>> ls.bounding_box.upper
6.0
```

get_swap()
Returns a LineSegment object which has its endpoints swapped.

get_swap() -> LineSegment

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

Examples

```
>>> ls = LineSegment(Point((1, 2)), Point((5, 6)))
>>> swap = ls.get_swap()
>>> swap.p1[0]
5.0
>>> swap.p1[1]
6.0
>>> swap.p2[0]
1.0
>>> swap.p2[1]
2.0
```

intersect(*other*)
Test whether segment intersects with other segment

Handles endpoints of segments being on other segment

Examples

```
>>> ls = LineSegment(Point((5,0)), Point((10,0)))
>>> ls1 = LineSegment(Point((5,0)), Point((10,1)))
>>> ls.intersect(ls1)
True
>>> ls2 = LineSegment(Point((5,1)), Point((10,1)))
>>> ls.intersect(ls2)
False
>>> ls2 = LineSegment(Point((7,-1)), Point((7,2)))
>>> ls.intersect(ls2)
True
>>>
```

is_ccw (*pt*)

Returns whether a point is counterclockwise of the segment. Exclusive.

is_ccw(Point) -> bool

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

Parameters **pt** (*point lying ccw or cw of a segment*) –

Examples

```
>>> ls = LineSegment(Point((0, 0)), Point((5, 0)))
>>> ls.is_ccw(Point((2, 2)))
True
>>> ls.is_ccw(Point((2, -2)))
False
```

is_cw (*pt*)

Returns whether a point is clockwise of the segment. Exclusive.

is_cw(Point) -> bool

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

Parameters **pt** (*point lying ccw or cw of a segment*) –

Examples

```
>>> ls = LineSegment(Point((0, 0)), Point((5, 0)))
>>> ls.is_cw(Point((2, 2)))
False
>>> ls.is_cw(Point((2, -2)))
True
```

len

Returns the length of a LineSegment object.

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

len() -> number

Examples

```
>>> ls = LineSegment(Point((2, 2)), Point((5, 2)))
>>> ls.len
3.0
```

line

Returns a Line object of the line which the segment lies on.

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

line() -> Line

Examples

```
>>> ls = LineSegment(Point((2, 2)), Point((3, 3)))
>>> l = ls.line
>>> l.m
1.0
>>> l.b
0.0
```

p1

HELPER METHOD. DO NOT CALL.

Returns the p1 attribute of the line segment.

_get_p1() -> Point

Examples

```
>>> ls = LineSegment(Point((1, 2)), Point((5, 6)))
>>> r = ls._get_p1()
>>> r == Point((1, 2))
True
```

p2

HELPER METHOD. DO NOT CALL.

Returns the p2 attribute of the line segment.

_get_p2() -> Point

Examples

```
>>> ls = LineSegment(Point((1, 2)), Point((5, 6)))
>>> r = ls._get_p2()
>>> r == Point((5, 6))
True
```

sw_ccw(*pt*)

Sedgewick test for pt being ccw of segment

Returns

- *1 if turn from self.p1 to self.p2 to pt is ccw*
- *-1 if turn from self.p1 to self.p2 to pt is cw*
- *-1 if the points are collinear and self.p1 is in the middle*
- *1 if the points are collinear and self.p2 is in the middle*
- *0 if the points are collinear and pt is in the middle*

class `pysal.cg.shapes.`**`Line`**(*m*, *b*)

Geometric representation of line objects.

`m`

float – slope

`b`

float – y-intercept

`x`(*y*)

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

x(number) -> number

Parameters **y** (*the y-value to compute x at*) –

Examples

```
>>> l = Line(0.5, 0)
>>> l.x(0.25)
0.5
```

`y`(*x*)

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

y(number) -> number

Parameters **x** (*the x-value to compute y at*) –

Examples

```
>>> l = Line(1, 0)
>>> l.y(1)
1.0
```

class `pysal.cg.shapes.`**`Ray`**(*origin*, *second_p*)

Geometric representation of ray objects.

`o`

Point – Origin (point where ray originates)

`p`

Point – Second point on the ray (not point where ray originates)

class `pysal.cg.shapes.`**`Chain`**(*vertices*)

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

`vertices`

list – List of Points of the vertices of the chain in order.

len
float – The geometric length of the chain.

arclen
Returns the geometric length of the chain computed using arcdistance (meters).

len -> number

Examples

bounding_box
Returns the bounding box of the chain.

bounding_box -> Rectangle

Examples

```
>>> c = Chain([Point((0, 0)), Point((2, 0)), Point((2, 1)), Point((0, 1))])
>>> c.bounding_box.left
0.0
>>> c.bounding_box.lower
0.0
>>> c.bounding_box.right
2.0
>>> c.bounding_box.upper
1.0
```

len
Returns the geometric length of the chain.

len -> number

Examples

```
>>> c = Chain([Point((0, 0)), Point((1, 0)), Point((1, 1)), Point((2, 1))])
>>> c.len
3.0
>>> c = Chain([[Point((0, 0)), Point((1, 0)), Point((1, 1))],[Point((10,10)),
↪Point((11,10)),Point((11,11))]])
>>> c.len
4.0
```

parts
Returns the parts of the chain.

parts -> Point list

Examples

```
>>> c = Chain([[Point((0, 0)), Point((1, 0)), Point((1, 1)), Point((0, 1))],
↪[Point((2,1)),Point((2,2)),Point((1,2)),Point((1,1))]])
>>> len(c.parts)
2
```

segments
Returns the segments that compose the Chain

vertices
Returns the vertices of the chain in clockwise order.

vertices -> Point list

Examples

```
>>> c = Chain([Point((0, 0)), Point((1, 0)), Point((1, 1)), Point((2, 1))])
>>> verts = c.vertices
>>> len(verts)
4
```

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

vertices
list – List of Points with the vertices of the Polygon in clockwise order

len
int – Number of vertices including holes

perimeter
float – Geometric length of the perimeter of the Polygon

bounding_box
Rectangle – Bounding box of the polygon

bbox
List – [left, lower, right, upper]

area
float – Area enclosed by the polygon

centroid
tuple – The 'center of gravity', i.e. the mean point of the polygon.

area
Returns the area of the polygon.

area -> number

Examples

```
>>> p = Polygon([Point((0, 0)), Point((1, 0)), Point((1, 1)), Point((0, 1))])
>>> p.area
1.0
>>> p = Polygon([Point((0, 0)), Point((10, 0)), Point((10, 10)), Point((0,␣
↪10))],[Point((2,1)),Point((2,2)),Point((1,2)),Point((1,1))])
>>> p.area
99.0
```

bbox
Returns the bounding box of the polygon as a list

See also bounding_box

bounding_box

Returns the bounding box of the polygon.

bounding_box -> Rectangle

Examples

```
>>> p = Polygon([Point((0, 0)), Point((2, 0)), Point((2, 1)), Point((0, 1))])
>>> p.bounding_box.left
0.0
>>> p.bounding_box.lower
0.0
>>> p.bounding_box.right
2.0
>>> p.bounding_box.upper
1.0
```

centroid

Returns the centroid of the polygon

centroid -> Point

Notes

The centroid returned by this method is the geometric centroid and respects multipart polygons with holes. Also known as the 'center of gravity' or 'center of mass'.

Examples

```
>>> p = Polygon([Point((0, 0)), Point((10, 0)), Point((10, 10)), Point((0,␣
↪10))], [Point((1, 1)), Point((1, 2)), Point((2, 2)), Point((2, 1))])
>>> p.centroid
(5.0353535353535355, 5.0353535353535355)
```

contains_point (*point*)

Test if polygon contains point

Examples

```
>>> p = Polygon([Point((0,0)), Point((4,0)), Point((4,5)), Point((2,3)),␣
↪Point((0,5))])
>>> p.contains_point((3,3))
1
>>> p.contains_point((0,6))
0
>>> p.contains_point((2,2.9))
1
>>> p.contains_point((4,5))
0
>>> p.contains_point((4,0))
0
>>>
```

Handles holes

```
>>> p = Polygon([Point((0, 0)), Point((0, 10)), Point((10, 10)), Point((10,␣
↪0))], [Point((2, 2)), Point((4, 2)), Point((4, 4)), Point((2, 4))])
>>> p.contains_point((3.0,3.0))
False
>>> p.contains_point((1.0,1.0))
True
>>>
```

Notes

Points falling exactly on polygon edges may yield unpredictable results

holes
Returns the holes of the polygon in clockwise order.

holes -> Point list

Examples

```
>>> p = Polygon([Point((0, 0)), Point((10, 0)), Point((10, 10)), Point((0,␣
↪10))], [Point((1, 2)), Point((2, 2)), Point((2, 1)), Point((1, 1))])
>>> len(p.holes)
1
```

len
Returns the number of vertices in the polygon.

len -> int

Examples

```
>>> p1 = Polygon([Point((0, 0)), Point((0, 1)), Point((1, 1)), Point((1, 0))])
>>> p1.len
4
>>> len(p1)
4
```

parts
Returns the parts of the polygon in clockwise order.

parts -> Point list

Examples

```
>>> p = Polygon([[Point((0, 0)), Point((1, 0)), Point((1, 1)), Point((0, 1))],
↪ [Point((2,1)),Point((2,2)),Point((1,2)),Point((1,1))]])
>>> len(p.parts)
2
```

perimeter
Returns the perimeter of the polygon.

perimeter() -> number

Examples

```
>>> p = Polygon([Point((0, 0)), Point((1, 0)), Point((1, 1)), Point((0, 1))])
>>> p.perimeter
4.0
```

vertices
Returns the vertices of the polygon in clockwise order.

vertices -> Point list

Examples

```
>>> p1 = Polygon([Point((0, 0)), Point((0, 1)), Point((1, 1)), Point((1, 0))])
>>> len(p1.vertices)
4
```

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

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

area
Returns the area of the Rectangle.

area -> number

Examples

```
>>> r = Rectangle(0, 0, 4, 4)
>>> r.area
16.0
```

height
Returns the height of the Rectangle.

height -> number

Examples

```
>>> r = Rectangle(0, 0, 4, 4)
>>> r.height
4.0
```

set_centroid(*new_center*)
Moves the rectangle center to a new specified point.

set_centroid(Point) -> Point

Parameters **new_center** (*the new location of the centroid of the polygon*) –

Examples

```
>>> r = Rectangle(0, 0, 4, 4)
>>> r.set_centroid(Point((4, 4)))
>>> r.left
2.0
>>> r.right
6.0
>>> r.lower
2.0
>>> r.upper
6.0
```

set_scale(*scale*)
Rescales the rectangle around its center.

set_scale(number) -> number

Parameters **scale** (*the ratio of the new scale to the old scale (e.g. 1.0 is current size)*) –

Examples

```
>>> r = Rectangle(0, 0, 4, 4)
>>> r.set_scale(2)
>>> r.left
-2.0
>>> r.right
6.0
>>> r.lower
-2.0
>>> r.upper
6.0
```

width
Returns the width of the Rectangle.

width -> number

Examples

```
>>> r = Rectangle(0, 0, 4, 4)
>>> r.width
4.0
```

`pysal.cg.shapes.`**`asShape`**(*obj*)
Returns a pysal shape object from obj. obj must support the __geo_interface__.

`cg.standalone` — Standalone

The `cg.standalone` module provides ...

New in version 1.0. Helper functions for computational geometry in PySAL

`pysal.cg.standalone.`**`bbcommon`**(*bb*, *bbother*)
Old Stars method for bounding box overlap testing Also defined in pysal.weights._cont_binning

Examples

```
>>> b0 = [0,0,10,10]
>>> b1 = [10,0,20,10]
>>> bbcommon(b0,b1)
1
```

`pysal.cg.standalone.`**`get_bounding_box`**(*items*)

Examples

```
>>> 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.cg.standalone.`**`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

```
>>> get_angle_between(Ray(Point((0, 0)), Point((1, 0))), Ray(Point((0, 0)),␣
↪Point((1, 0))))
0.0
```

pysal.cg.standalone.**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

```
>>> 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.cg.standalone.**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

```
>>> 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.cg.standalone.**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

```
>>> 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.cg.standalone.`**`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

```
>>> 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.cg.standalone.`**`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

```
>>> 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.cg.standalone.`**`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

- **ray** (*a ray to check intersection for*) –
- **seg** (*a line segment to check intersection for*) –

Examples

```
>>> 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.cg.standalone.**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

```
>>> 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.cg.standalone.**get_polygon_point_dist**(*poly*, *pt*)

Returns the distance between a polygon and point.

get_polygon_point_dist(Polygon, Point) -> number

Parameters

- **poly** (*a polygon to compute distance from*) –
- **pt** (*a point to compute distance from*) –

Examples

```
>>> 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.cg.standalone.`**`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

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

`pysal.cg.standalone.`**`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

```
>>> 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.cg.standalone.`**`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

```
>>> 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.cg.standalone.**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

```
>>> 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.cg.standalone.**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.
```

```
>>> is_clockwise(v)
True
```

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

point_touches_rectangle(point, rect) -> bool

Parameters

- **point** (Point *or Tuple*) –
- **rect** (Rectangle) –

Examples

```
>>> 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.cg.standalone.**get_shared_segments**(*poly1*, *poly2*, *bool_ret=False*)
Returns the line segments in common to both polygons.

get_shared_segments(poly1, poly2) -> list

Parameters

- **poly1** (*a Polygon*) –
- **poly2** (*a Polygon*) –

Examples

```
>>> x = [0, 0, 1, 1]
>>> y = [0, 1, 1, 0]
>>> poly1 = Polygon( map(Point,zip(x,y)) )
>>> x = [a+1 for a in x]
>>> poly2 = Polygon( map(Point,zip(x,y)) )
>>> get_shared_segments(poly1, poly2, bool_ret=True)
True
```

pysal.cg.standalone.**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 ammount of ram (in bytes) that will be used.

Examples

```
>>> 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.        ,
         1.41421356,  2.23606798,  2.        ,  2.23606798],
       [ 2.        ,  1.        ,  0.        ,  2.23606798,  1.41421356,
         1.        ,  2.82842712,  2.23606798,  2.        ],
       [ 1.        ,  1.41421356,  2.23606798,  0.        ,  1.        ,
         2.        ,  1.        ,  1.41421356,  2.23606798],
       [ 1.41421356,  1.        ,  1.41421356,  1.        ,  0.        ,
         1.        ,  1.41421356,  1.        ,  1.41421356],
       [ 2.23606798,  1.41421356,  1.        ,  2.        ,  1.        ,
         0.        ,  2.23606798,  1.41421356,  1.        ],
       [ 2.        ,  2.23606798,  2.82842712,  1.        ,  1.41421356,
         2.23606798,  0.        ,  1.        ,  2.        ],
       [ 2.23606798,  2.        ,  2.23606798,  1.41421356,  1.        ,
         1.41421356,  1.        ,  0.        ,  1.        ],
       [ 2.82842712,  2.23606798,  2.        ,  2.23606798,  1.41421356,
         1.        ,  2.        ,  1.        ,  0.        ]])
>>>
```

cg.rtree — rtree

The cg.rtree module provides a pure python rtree.

New in version 1.2. Pure Python implementation of RTree spatial index

Adaptation of http://code.google.com/p/pyrtree/

R-tree. see doc/ref/r-tree-clustering-split-algo.pdf

class pysal.cg.rtree.**Rect** (*minx*, *miny*, *maxx*, *maxy*)

A rectangle class that stores: an axis aligned rectangle, and: two flags (swapped_x and swapped_y). (The flags are stored implicitly via swaps in the order of minx/y and maxx/y.)

cg.kdtree — KDTree

The cg.kdtree module provides kdtree data structures for PySAL.

New in version 1.3. KDTree for PySAL: Python Spatial Analysis Library.

Adds support for Arc Distance to scipy.spatial.KDTree.

`pysal.cg.kdtree.`**`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.cg.RADIUS_EARTH_KM (default) pysal.cg.RADIUS_EARTH_MILES

`cg.sphere` — Sphere

The `cg.sphere` module provides tools for working with spherical distances.

New in version 1.3. sphere: Tools for working with spherical geometry.

Author(s): Charles R Schmidt schmidtc@gmail.com Luc Anselin luc.anselin@asu.edu Xun Li xun.li@asu.edu

`pysal.cg.sphere.`**`arcdist`**(*pt0*, *pt1*, *radius=6371.0*)

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

  Source: http://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html

Returns

Return type The arc distance between pt0 and pt1 using supplied radius

Examples

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

`pysal.cg.sphere.`**`arcdist2linear`**(*arc_dist*, *radius=6371.0*)

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

Examples

```
>>> 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.cg.sphere.`**`brute_knn`**(*pts*, *k*, *mode='arc'*)
valid modes are ['arc','xrz']

`pysal.cg.sphere.`**`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.cg.sphere.`**`linear2arcdist`**(*linear_dist*, *radius=6371.0*)
Convert a linear distance in the unit sphere (R3) to an arc distance based on supplied radius

Examples

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

`pysal.cg.sphere.`**`toXYZ`**(*pt*)

Parameters

- **pt0** (*point*) – assumed to be in form (lng,lat)
- **pt1** (*point*) – assumed to be in form (lng,lat)

Returns

Return type *x, y, z*

`pysal.cg.sphere.`**`lonlat`**(*pointslist*)
Converts 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**

Return type list with tuples of points in lon-lat order

Example

```
>>> points = [(41.981417, -87.893517), (41.980396, -87.776787), (41.980906, -87.
↪696450)]
>>> newpoints = lonlat(points)
>>> newpoints
[(-87.893517, 41.981417), (-87.776787, 41.980396), (-87.69645, 41.980906)]
```

`pysal.cg.sphere.`**`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)

  None (for result in radians)

Returns **d**

Return type distance in units specified, km, miles or radians (for None)

Example

```
>>> p0 = (-87.893517, 41.981417)
>>> p1 = (-87.519295, 41.657498)
>>> harcdist(p0,p1)
47.52873002976876
>>> harcdist(p0,p1,radius=None)
0.007460167953189258
```

Note: Uses radangle function to compute radian angle

`pysal.cg.sphere.`**`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** – depending on setting of lonx; in other words, the same order is used as for the input

Return type tuple in decimal degrees of lon-lat (default) or lat-lon,

Example

```
>>> p0 = (-87.893517, 41.981417)
>>> p1 = (-87.519295, 41.657498)
>>> geointerpolate(p0,p1,0.1)           # using lon-lat
(-87.85592403438788, 41.949079912574796)
>>> p3 = (41.981417, -87.893517)
>>> p4 = (41.657498, -87.519295)
>>> geointerpolate(p3,p4,0.1,lonx=False)    # using lat-lon
(41.949079912574796, -87.85592403438788)
```

`pysal.cg.sphere.`**`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** – starting with the top row and moving to the bottom; coordinate tuples are returned in same order as input

Return type list of tuples with lat-lon or lon-lat for grid points, row by row,

Example

```
>>> pup = (42.023768,-87.946389)     # Arlington Heights IL
>>> pdown = (41.644415,-87.524102)   # Hammond, IN
>>> geogrid(pup,pdown,3,lonx=False)
[(42.023768, -87.946389), (42.02393997819538, -87.80562679358316), (42.
↪02393997819538, -87.66486420641684), (42.023768, -87.524102), (41.897317, -87.
↪94638900000001), (41.8974888973743, -87.80562679296166), (41.8974888973743, -87.
↪66486420703835), (41.897317, -87.524102), (41.770866000000005, -87.
↪94638900000001), (41.77103781320412, -87.80562679234043), (41.77103781320412, -
↪87.66486420765956), (41.770866000000005, -87.524102), (41.644415, -87.946389),␣
↪(41.64458672568646, -87.80562679171955), (41.64458672568646, -87.
↪66486420828045), (41.644415, -87.524102)]
```

`pysal.core` — Core Data Structures and IO

`Tables` – DataTable Extension

New in version 1.0.

class `pysal.core.Tables.`**`DataTable`**(**args*, ***kwargs*)

DataTable provides additional functionality to FileIO for data table file tables FileIO Handlers that provide data tables should subclass this instead of FileIO

`by_col`

by_col_array(**args*)

Return columns of table as a numpy array.

Parameters ***args** (*any number of strings of length k*) – names of variables to extract

Returns **implicit**

Return type numpy array of shape (*n*,*k*)

Notes

If the variables are not all of the same data type, then numpy rules for casting will result in a uniform type applied to all variables.

If only strings are passed to the function, then an array with those columns will be constructed.

If only one list of strings is passed, the output is identical to those strings being passed.

If at least one list is passed and other strings or lists are passed, this returns a tuple containing arrays constructed from each positional argument.

Examples

```
>>> import pysal as ps
>>> dbf = ps.open(ps.examples.get_path('NAT.dbf'))
>>> hr = dbf.by_col_array('HR70', 'HR80')
>>> hr[0:5]
array([[  0.        ,   8.85582713],
       [  0.        ,  17.20874204],
       [  1.91515848,   3.4507747 ],
       [  1.28864319,   3.26381409],
       [  0.        ,   7.77000777]])
>>> hr = dbf.by_col_array(['HR80', 'HR70'])
>>> hr[0:5]
array([[  8.85582713,   0.        ],
       [ 17.20874204,   0.        ],
       [  3.4507747 ,   1.91515848],
       [  3.26381409,   1.28864319],
       [  7.77000777,   0.        ]])
>>> hr = dbf.by_col_array(['HR80'])
>>> hr[0:5]
array([[  8.85582713],
       [ 17.20874204],
       [  3.4507747 ],
       [  3.26381409],
       [  7.77000777]])
```

Numpy only supports homogeneous arrays. See Notes above.

```
>>> hr = dbf.by_col_array('STATE_NAME', 'HR80')
>>> hr[0:5]
array([['Minnesota', '8.8558271343'],
       ['Washington', '17.208742041'],
       ['Washington', '3.4507746989'],
       ['Washington', '3.2638140931'],
       ['Washington', '7.77000777']],
      dtype='|S20')
```

```
>>> y, X = dbf.by_col_array('STATE_NAME', ['HR80', 'HR70'])
>>> y[0:5]
array([['Minnesota'],
       ['Washington'],
       ['Washington'],
       ['Washington'],
       ['Washington']],
      dtype='|S20')
>>> X[0:5]
array([[  8.85582713,   0.        ],
       [ 17.20874204,   0.        ],
       [  3.4507747 ,   1.91515848],
       [  3.26381409,   1.28864319],
       [  7.77000777,   0.        ]])
```

to_df(*n=-1*, *read_shp=None*, ***df_kws*)

FileIO – File Input/Output System

New in version 1.0. FileIO: Module for reading and writing various file types in a Pythonic way. This module should not be used directly, instead... import pysal.core.FileIO as FileIO Readers and Writers will mimic python file objects. .seek(n) seeks to the n'th object .read(n) reads n objects, default == all .next() reads the next object

class pysal.core.FileIO.**FileIO**(*dataPath=''*, *mode='r'*, *dataFormat=None*)

How this works: FileIO.open(*args) == FileIO(*args) When creating a new instance of FileIO the .__new__ method intercepts .__new__ 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

by_row

cast(*key*, *typ*)
cast key as typ

classmethod check()
Prints the contents of the registry

close()
subclasses should clean themselves up and then call this method

flush()

get(*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

static getType(*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next()
A FileIO object is its own iterator, see StringIO

classmethod open(**args*, ***kwargs*)
Alias for FileIO()

rIds

read(*n=-1*)
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.

seek(*n*)
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

tell()
Return id (or offset) of next object

truncate(*size=None*)
Should be implemented by subclasses and redefine this doc string

write(*obj*)
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

`pysal.core.IOHandlers` — Input Output Handlers

`IOHandlers.arcgis_dbf` – ArcGIS DBF plugin

New in version 1.2.

class `pysal.core.IOHandlers.arcgis_dbf.`**ArcGISDbfIO**(**args*, ***kwargs*)
Opens, reads, and writes weights file objects in ArcGIS dbf format.

Spatial weights objects in the ArcGIS dbf format are used in ArcGIS Spatial Statistics tools. This format is the same as the general dbf format, but the structure of the weights dbf file is fixed unlike other dbf files. This dbf format can be used with the "Generate Spatial Weights Matrix" tool, but not with the tools under the "Mapping Clusters" category.

The ArcGIS dbf file is assumed to have three or four data columns. When the file has four columns, the first column is meaningless and will be ignored in PySAL during both file reading and file writing. The next three columns hold origin IDs, destinations IDs, and weight values. When the file has three columns, it is assumed that only these data columns exist in the stated order. The name for the orgin IDs column should be the name of ID variable in the original source data table. The names for the destination IDs and weight values columns are NID and WEIGHT, respectively. ArcGIS Spatial Statistics tools support only unique integer IDs. Therefore, the values for origin and destination ID columns should be integer. For the case where the IDs of a weights object are not integers, ArcGISDbfIO allows users to use internal id values corresponding to record numbers, instead of original ids.

An exemplary structure of an ArcGIS dbf file is as follows: [Line 1] Field1 RECORD_ID NID WEIGHT [Line 2] 0 72 76 1 [Line 3] 0 72 79 1 [Line 4] 0 72 78 1 ...

Unlike the ArcGIS text format, this format does not seem to include self-neighbors.

References

http://webhelp.esri.com/arcgisdesktop/9.3/index.cfm?TopicName=Convert_Spatial_Weights_Matrix_to_Table_(Spatial_Statistics)

`FORMATS` = ['arcgis_dbf']

`MODES` = ['r', 'w']

`by_row`

`cast`(*key*, *typ*)
cast key as typ

`check`()
Prints the contents of the registry

`close`()

`flush`()

`get`(*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

`getType`(*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

`ids`

`next`()
A FileIO object is its own iterator, see StringIO

`open`(**args*, ***kwargs*)
Alias for FileIO()

`rIds`

`read`(*n=-1*)

`seek`(*pos*)

`tell`()
Return id (or offset) of next object

`truncate`(*size=None*)
Should be implemented by subclasses and redefine this doc string

`varName`

`write`(*obj*, *useIdIndex=False*)

Parameters

- **`write(weightsObject)`** –
- **`a weights object`** (*`accepts`*) –

Returns

- *an ArcGIS dbf file*
- *write a weights object to the opened dbf file.*

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('arcgis_ohio.dbf'),'r',
↪'arcgis_dbf')
>>> w = testfile.read()
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='.dbf')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w','arcgis_dbf')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created text file

```
>>> wnew =  pysal.open(fname,'r','arcgis_dbf').read()
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

`IOHandlers.arcgis_swm` — ArcGIS SWM plugin

New in version 1.2.

class `pysal.core.IOHandlers.arcgis_swm.`**`ArcGISSwmIO`**(**args*, ***kwargs*)

Opens, reads, and writes weights file objects in ArcGIS swm format.

Spatial weights objects in the ArcGIS swm format are used in ArcGIS Spatial Statistics tools. Particularly, this format can be directly used with the tools under the category of Mapping Clusters.

The values for [ORG_i] and [DST_i] should be integers, as ArcGIS Spatial Statistics tools support only unique integer IDs. For the case where a weights object uses non-integer IDs, ArcGISSwmIO allows users to use internal ids corresponding to record numbers, instead of original ids.

The specifics of each part of the above structure is as follows.

Table 3.1: ArcGIS SWM Components

Part	Data type	Description	Length
ID_VAR_NAME	ASCII TEXT	ID variable name	Flexible (Up to the 1st ;)
ESRI_SRS	ASCII TEXT	ESRI spatial reference system	Flexible (Btw the 1st ; and n)
NO_OBS	l.e. int	Number of observations	4
ROW_STD	l.e. int	Whether or not row-standardized	4
WGT_i			
ORG_i	l.e. int	ID of observaiton i	4
NO_NGH_i	l.e. int	Number of neighbors for obs. i (m)	4
NGHS_i			
DSTS_i	l.e. int	IDs of all neighbors of obs. i	4*m
WS_i	l.e. float	Weights for obs. i and its neighbors	8*m
W_SUM_i	l.e. float	Sum of weights for “	8

FORMATS = [’swm’]

MODES = [’r’, ‘w’]

by_row

cast (*key*, *typ*)
cast key as typ

check ()
Prints the contents of the registry

close ()

flush ()

get (*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType (*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next ()
A FileIO object is its own iterator, see StringIO

open (**args*, ***kwargs*)
Alias for FileIO()

rIds

read (*n=-1*)

seek (*pos*)

tell ()
Return id (or offset) of next object

truncate (*size=None*)
Should be implemented by subclasses and redefine this doc string

varName

write (*obj*, *useIdIndex=False*)
Writes a spatial weights matrix data file in swm format.

Parameters

- **write(weightsObject)** –
- **a weights object** (*accepts*) –

Returns

- *an ArcGIS swm file*
- *write a weights object to the opened swm file.*

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('ohio.swm'),'r')
>>> w = testfile.read()
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='.swm')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created text file

```
>>> wnew = pysal.open(fname,'r').read()
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

`IOHandlers.arcgis_txt` – ArcGIS ASCII plugin

New in version 1.2.

class `pysal.core.IOHandlers.arcgis_txt.`**`ArcGISTextIO`**(**args*, ***kwargs*)

Opens, reads, and writes weights file objects in ArcGIS ASCII text format.

Spatial weights objects in the ArcGIS text format are used in ArcGIS Spatial Statistics tools. This format is a simple text file with ASCII encoding. This format can be directly used with the tools under the category of "Mapping Clusters." But, it cannot be used with the "Generate Spatial Weights Matrix" tool.

The first line of the ArcGIS text file is a header including the name of a data column that holded the ID variable in the original source data table. After this header line, it includes three data columns for origin id, destination id, and weight values. ArcGIS Spatial Statistics tools support only unique integer ids. Thus, the values in the first two columns should be integers. For the case where a weights object uses non-integer IDs, ArcGISTextIO allows users to use internal ids corresponding to record numbers, instead of original ids.

An exemplary structure of an ArcGIS text file is as follows: [Line 1] StationID [Line 2] 1 1 0.0 [Line 3] 1 2 0.1 [Line 4] 1 3 0.14286 [Line 5] 2 1 0.1 [Line 6] 2 3 0.05 [Line 7] 3 1 0.16667 [Line 8] 3 2 0.06667 [Line 9] 3 3 0.0 ...

As shown in the above example, this file format allows explicit specification of weights for self-neighbors. When no entry is available for self-neighbors, ArcGIS spatial statistics tools consider they have zero weights. PySAL ArcGISTextIO class ignores self-neighbors if their weights are zero.

References

http://webhelp.esri.com/arcgisdesktop/9.3/index.cfm?TopicName=Modeling_spatial_relationships

Notes

When there are an dbf file whose name is identical to the name of the source text file, ArcGISTextIO checks the data type of the ID data column and uses it for reading and writing the text file. Otherwise, it considers IDs are strings.

`FORMATS` **= ['arcgis_text']**

`MODES` **= ['r', 'w']**

`by_row`

`cast`(*key*, *typ*)
cast key as typ

`check`()
Prints the contents of the registry

`close`()

`flush`()

`get`(*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

`getType`(*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

`ids`

`next`()
A FileIO object is its own iterator, see StringIO

`open`(**args*, ***kwargs*)
Alias for FileIO()

rIds

read(*n=-1*)

seek(*pos*)

shpName

tell()
Return id (or offset) of next object

truncate(*size=None*)
Should be implemented by subclasses and redefine this doc string

varName

write(*obj, useIdIndex=False*)

Parameters

- **write(weightsObject)** –
- **a weights object** (*accepts*) –

Returns

- *an ArcGIS text file*
- *write a weights object to the opened text file.*

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('arcgis_txt.txt'),'r',
↪'arcgis_text')
>>> w = testfile.read()
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='.txt')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w','arcgis_text')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created text file

```
>>> wnew =  pysal.open(fname,'r','arcgis_text').read()
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

`IOHandlers.csvWrapper` — CSV plugin

New in version 1.0.

class `pysal.core.IOHandlers.csvWrapper.`**`csvWrapper`**(**args*, ***kwargs*)

DataTable provides additional functionality to FileIO for data table file tables FileIO Handlers that provide data tables should subclass this instead of FileIO

`FORMATS` = ['csv']

`MODES` = ['r', 'Ur', 'rU', 'U']

`READ_MODES` = ['r', 'Ur', 'rU', 'U']

`by_col`

`by_col_array`(**args*)

Return columns of table as a numpy array.

Parameters ***args** (*any number of strings of length k*) – names of variables to extract

Returns **implicit**

Return type numpy array of shape (n,k)

Notes

If the variables are not all of the same data type, then numpy rules for casting will result in a uniform type applied to all variables.

If only strings are passed to the function, then an array with those columns will be constructed.

If only one list of strings is passed, the output is identical to those strings being passed.

If at least one list is passed and other strings or lists are passed, this returns a tuple containing arrays constructed from each positional argument.

Examples

```
>>> import pysal as ps
>>> dbf = ps.open(ps.examples.get_path('NAT.dbf'))
>>> hr = dbf.by_col_array('HR70', 'HR80')
>>> hr[0:5]
array([[  0.        ,   8.85582713],
       [  0.        ,  17.20874204],
       [  1.91515848,   3.4507747 ],
       [  1.28864319,   3.26381409],
       [  0.        ,   7.77000777]])
>>> hr = dbf.by_col_array(['HR80', 'HR70'])
>>> hr[0:5]
array([[  8.85582713,   0.        ],
       [ 17.20874204,   0.        ],
       [  3.4507747 ,   1.91515848],
       [  3.26381409,   1.28864319],
       [  7.77000777,   0.        ]])
>>> hr = dbf.by_col_array(['HR80'])
>>> hr[0:5]
array([[  8.85582713],
       [ 17.20874204],
       [  3.4507747 ],
       [  3.26381409],
       [  7.77000777]])
```

Numpy only supports homogeneous arrays. See Notes above.

```
>>> hr = dbf.by_col_array('STATE_NAME', 'HR80')
>>> hr[0:5]
array([['Minnesota', '8.8558271343'],
       ['Washington', '17.208742041'],
       ['Washington', '3.4507746989'],
       ['Washington', '3.2638140931'],
       ['Washington', '7.77000777']],
      dtype='|S20')
```

```
>>> y, X = dbf.by_col_array('STATE_NAME', ['HR80', 'HR70'])
>>> y[0:5]
array([['Minnesota'],
       ['Washington'],
       ['Washington'],
       ['Washington'],
       ['Washington']],
      dtype='|S20')
>>> X[0:5]
array([[  8.85582713,   0.        ],
       [ 17.20874204,   0.        ],
       [  3.4507747 ,   1.91515848],
       [  3.26381409,   1.28864319],
       [  7.77000777,   0.        ]])
```

by_row

cast (*key*, *typ*)
cast key as typ

check ()
Prints the contents of the registry

close ()
subclasses should clean themselves up and then call this method

flush ()

get (*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType (*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next ()
A FileIO object is its own iterator, see StringIO

open (**args*, ***kwargs*)
Alias for FileIO()

rIds

read (*n=-1*)
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.

seek (*n*)
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

tell ()
Return id (or offset) of next object

to_df (*n=-1*, *read_shp=None*, ***df_kws*)

truncate (*size=None*)
Should be implemented by subclasses and redefine this doc string

write (*obj*)
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

`IOHandlers.dat` — DAT plugin

New in version 1.2.

class `pysal.core.IOHandlers.dat.`**DatIO** (**args*, ***kwargs*)
Opens, reads, and writes file objects in DAT format.

Spatial weights objects in DAT format are used in Dr. LeSage's MatLab Econ library. This DAT format is a simple text file with DAT or dat extension. Without header line, it includes three data columns for origin id, destination id, and weight values as follows:

[Line 1] 2 1 0.25 [Line 2] 5 1 0.50 ...

Origin/destination IDs in this file format are simply record numbers starting with 1. IDs are not necessarily integers. Data values for all columns should be numeric.

FORMATS = ['dat']

MODES = ['r', 'w']

by_row

cast (*key*, *typ*)
cast key as typ

check ()
Prints the contents of the registry

close ()

flush()

get(*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType(*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next()
A FileIO object is its own iterator, see StringIO

open(**args*, ***kwargs*)
Alias for FileIO()

rIds

read(*n=-1*)

seek(*pos*)

shpName

tell()
Return id (or offset) of next object

truncate(*size=None*)
Should be implemented by subclasses and redefine this doc string

varName

write(*obj*)

Parameters

- **write(weightsObject)** –
- **a weights object** (*accepts*) –

Returns

- *a DAT file*
- *write a weights object to the opened DAT file.*

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('wmat.dat'),'r')
>>> w = testfile.read()
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='.dat')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created dat file

```
>>> wnew =  pysal.open(fname,'r').read()
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

`IOHandlers.gal` — GAL plugin

New in version 1.0.

class `pysal.core.IOHandlers.gal.`**`GalIO`**(**args*, ***kwargs*)
Opens, reads, and writes file objects in GAL format.

`FORMATS` **= ['gal']**

`MODES` **= ['r', 'w']**

`by_row`

`cast`(*key*, *typ*)
cast key as typ

`check`()
Prints the contents of the registry

`close`()

`data_type`

`flush`()

`get`(*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

`getType`(*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

`ids`

`next`()
A FileIO object is its own iterator, see StringIO

open(**args, **kwargs*)
Alias for FileIO()

rIds

read(*n=-1, sparse=False*)

sparse: boolean If true return scipy sparse object If false return pysal w object

seek(*pos*)

tell()
Return id (or offset) of next object

truncate(*size=None*)
Should be implemented by subclasses and redefine this doc string

write(*obj*)

Parameters

- **write(weightsObject)** –
- **a weights object** (*accepts*) –

Returns

- *a GAL file*
- *write a weights object to the opened GAL file.*

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('sids2.gal'),'r')
>>> w = testfile.read()
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='.gal')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created gal file

```
>>> wnew =  pysal.open(fname,'r').read()
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

`IOHandlers.geobugs_txt` — GeoBUGS plugin

New in version 1.2.

class `pysal.core.IOHandlers.geobugs_txt.`**`GeoBUGSTextIO`**(**args*, ***kwargs*)

Opens, reads, and writes weights file objects in the text format used in GeoBUGS. GeoBUGS generates a spatial weights matrix as an R object and writes it out as an ASCII text representation of the R object.

An exemplary GeoBUGS text file is as follows. list([CARD],[ADJ],[WGT],[SUMNUMNEIGH]) where [CARD] and [ADJ] are required but the others are optional. PySAL assumes [CARD] and [ADJ] always exist in an input text file. It can read a GeoBUGS text file, even when its content is not written in the order of [CARD], [ADJ], [WGT], and [SUMNUMNEIGH]. It always writes all of [CARD], [ADJ], [WGT], and [SUMNUMNEIGH]. PySAL does not apply text wrapping during file writing.

In the above example,

[CARD]: num=c([a list of comma-splitted neighbor cardinalities])

[ADJ]: adj=c([a list of comma-splitted neighbor IDs]) if caridnality is zero, neighbor IDs are skipped. The ordering of observations is the same in both [CARD] and [ADJ]. Neighbor IDs are record numbers starting from one.

[WGT]: weights=c([a list of comma-splitted weights]) The restrictions for [ADJ] also apply to [WGT].

[SUMNUMNEIGH]: sumNumNeigh=[The total number of neighbor pairs] the total number of neighbor pairs is an integer value and the same as the sum of neighbor cardinalities.

Notes

For the files generated from R spdep nb2WB and dput function, it is assumed that the value for the control parameter of dput function is NULL. Please refer to R spdep nb2WB function help file.

References

Thomas, A., Best, N., Lunn, D., Arnold, R., and Spiegelhalter, D.

2004.GeoBUGS User Manual.

R spdep nb2WB function help file.

`FORMATS` = ['geobugs_text']

`MODES` = ['r', 'w']

`by_row`

cast (*key*, *typ*)
cast key as typ

check ()
Prints the contents of the registry

close ()

flush ()

get (*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType (*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next ()
A FileIO object is its own iterator, see StringIO

open (**args*, ***kwargs*)
Alias for FileIO()

rIds

read (*n=-1*)
Reads GeoBUGS text file

Returns

Return type a pysal.weights.weights.W object

Examples

Type 'dir(w)' at the interpreter to see what methods are supported. Open a GeoBUGS text file and read it into a pysal weights object

```
>>> w = pysal.open(pysal.examples.get_path('geobugs_scot'),'r','geobugs_text
↪').read()
WARNING: there are 3 disconnected observations
Island ids:  [6, 8, 11]
```

Get the number of observations from the header

```
>>> w.n
56
```

Get the mean number of neighbors

```
>>> w.mean_neighbors
4.1785714285714288
```

Get neighbor distances for a single observation

```
>>> w[1]
{9: 1.0, 19: 1.0, 5: 1.0}
```

seek (*pos*)

tell()
Return id (or offset) of next object

truncate(*size=None*)
Should be implemented by subclasses and redefine this doc string

write(*obj*)
Writes a weights object to the opened text file.

Parameters

- **write(weightsObject)** –
- **a weights object** (*accepts*) –

Returns

Return type a GeoBUGS text file

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('geobugs_scot'),'r',
↪'geobugs_text')
>>> w = testfile.read()
WARNING: there are 3 disconnected observations
Island ids:  [6, 8, 11]
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w','geobugs_text')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created text file

```
>>> wnew =  pysal.open(fname,'r','geobugs_text').read()
WARNING: there are 3 disconnected observations
Island ids:  [6, 8, 11]
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

IOHandlers.geoda_txt – Geoda text plugin

New in version 1.0.

class pysal.core.IOHandlers.geoda_txt.**GeoDaTxtReader**(**args*, ***kwargs*)

DataTable provides additional functionality to FileIO for data table file tables FileIO Handlers that provide data tables should subclass this instead of FileIO

FORMATS = ['geoda_txt']

MODES = ['r']

by_col

by_col_array(**args*)

Return columns of table as a numpy array.

Parameters ***args** (*any number of strings of length k*) – names of variables to extract

Returns **implicit**

Return type numpy array of shape (*n*,*k*)

Notes

If the variables are not all of the same data type, then numpy rules for casting will result in a uniform type applied to all variables.

If only strings are passed to the function, then an array with those columns will be constructed.

If only one list of strings is passed, the output is identical to those strings being passed.

If at least one list is passed and other strings or lists are passed, this returns a tuple containing arrays constructed from each positional argument.

Examples

```
>>> import pysal as ps
>>> dbf = ps.open(ps.examples.get_path('NAT.dbf'))
>>> hr = dbf.by_col_array('HR70', 'HR80')
>>> hr[0:5]
array([[  0.        ,   8.85582713],
       [  0.        ,  17.20874204],
       [  1.91515848,   3.4507747 ],
       [  1.28864319,   3.26381409],
       [  0.        ,   7.77000777]])
>>> hr = dbf.by_col_array(['HR80', 'HR70'])
>>> hr[0:5]
array([[  8.85582713,   0.        ],
       [ 17.20874204,   0.        ],
       [  3.4507747 ,   1.91515848],
       [  3.26381409,   1.28864319],
       [  7.77000777,   0.        ]])
>>> hr = dbf.by_col_array(['HR80'])
>>> hr[0:5]
array([[  8.85582713],
       [ 17.20874204],
       [  3.4507747 ],
       [  3.26381409],
       [  7.77000777]])
```

Numpy only supports homogeneous arrays. See Notes above.

```
>>> hr = dbf.by_col_array('STATE_NAME', 'HR80')
>>> hr[0:5]
array([['Minnesota', '8.8558271343'],
       ['Washington', '17.208742041'],
       ['Washington', '3.4507746989'],
       ['Washington', '3.2638140931'],
       ['Washington', '7.77000777']],
      dtype='|S20')
```

```
>>> y, X = dbf.by_col_array('STATE_NAME', ['HR80', 'HR70'])
>>> y[0:5]
array([['Minnesota'],
       ['Washington'],
       ['Washington'],
       ['Washington'],
       ['Washington']],
      dtype='|S20')
>>> X[0:5]
array([[  8.85582713,   0.        ],
       [ 17.20874204,   0.        ],
       [  3.4507747 ,   1.91515848],
       [  3.26381409,   1.28864319],
       [  7.77000777,   0.        ]])
```

by_row

cast (*key*, *typ*)
cast key as typ

check ()
Prints the contents of the registry

close ()

flush ()

get (*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType (*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next ()
A FileIO object is its own iterator, see StringIO

open (**args*, ***kwargs*)
Alias for FileIO()

rIds

read(*n=-1*)
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.

seek(*n*)
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

tell()
Return id (or offset) of next object

to_df(*n=-1*, *read_shp=None*, ***df_kws*)

truncate(*size=None*)
Should be implemented by subclasses and redefine this doc string

write(*obj*)
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

IOHandlers.gwt — GWT plugin

New in version 1.0.

class pysal.core.IOHandlers.gwt.**GwtIO**(**args*, ***kwargs*)

FORMATS = ['kwt', 'gwt']

MODES = ['r', 'w']

by_row

cast(*key*, *typ*)
cast key as typ

check()
Prints the contents of the registry

close()

flush()

get(*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType(*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next()
A FileIO object is its own iterator, see StringIO

open(**args*, ***kwargs*)
Alias for FileIO()

rIds

read(*n=-1*)

seek(*pos*)

shpName

tell()
Return id (or offset) of next object

truncate(*size=None*)
Should be implemented by subclasses and redefine this doc string

varName

write(*obj*)

Parameters

- **write(weightsObject)** –
- **a weights object** (*accepts*) –

Returns

- *a GWT file*
- *write a weights object to the opened GWT file.*

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('juvenile.gwt'),'r')
>>> w = testfile.read()
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='.gwt')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created gwt file

```
>>> wnew =  pysal.open(fname,'r').read()
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

`IOHandlers.mat` — MATLAB Level 4-5 plugin

New in version 1.2.

class `pysal.core.IOHandlers.mat.`**`MatIO`**`(`**args*, ***kwargs*`)`

Opens, reads, and writes weights file objects in MATLAB Level 4-5 MAT format.

MAT files are used in Dr. LeSage's MATLAB Econometrics library. The MAT file format can handle both full and sparse matrices, and it allows for a matrix dimension greater than 256. In PySAL, row and column headers of a MATLAB array are ignored.

PySAL uses matlab io tools in scipy. Thus, it is subject to all limits that loadmat and savemat in scipy have.

Notes

If a given weights object contains too many observations to write it out as a full matrix, PySAL writes out the object as a sparse matrix.

References

MathWorks (2011) "MATLAB 7 MAT-File Format" at http://www.mathworks.com/help/pdf_doc/matlab/matfile_format.pdf.

scipy matlab io http://docs.scipy.org/doc/scipy/reference/tutorial/io.html

`FORMATS` **= ['mat']**

`MODES` **= ['r', 'w']**

`by_row`

`cast``(`*key*, *typ*`)`
cast key as typ

`check``()`
Prints the contents of the registry

`close``()`

`flush``()`

`get``(`*n*`)`
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

`getType``(`*dataPath*, *mode*, *dataFormat=None*`)`
Parse the dataPath and return the data type

`ids`

`next``()`
A FileIO object is its own iterator, see StringIO

`open``(`**args*, ***kwargs*`)`
Alias for FileIO()

rIds

read(*n=-1*)

seek(*pos*)

tell()
Return id (or offset) of next object

truncate(*size=None*)
Should be implemented by subclasses and redefine this doc string

varName

write(*obj*)

Parameters

- **write(weightsObject)** –
- **a weights object** (*accepts*) –

Returns

- *a MATLAB mat file*
- *write a weights object to the opened mat file.*

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('spat-sym-us.mat'),'r')
>>> w = testfile.read()
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='.mat')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created mat file

```
>>> wnew =  pysal.open(fname,'r').read()
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

`IOHandlers.mtx` — Matrix Market MTX plugin

New in version 1.2.

class `pysal.core.IOHandlers.mtx.`**`MtxIO`**(**args*, ***kwargs*)

Opens, reads, and writes weights file objects in Matrix Market MTX format.

The Matrix Market MTX format is used to facilitate the exchange of matrix data. In PySAL, it is being tested as a new file format for delivering the weights information of a spatial weights matrix. Although the MTX format supports both full and sparse matrices with different data types, it is assumed that spatial weights files in the mtx format always use the sparse (or coordinate) format with real data values. For now, no additional assumption (e.g., symmetry) is made of the structure of a weights matrix.

With the above assumptions, the structure of a MTX file containing a spatial weights matrix can be defined as follows: %%MatrixMarket matrix coordinate real general <— header 1 (constant) % Comments starts <— % .... | 0 or more comment lines % Comments ends <— M N L <— header 2, rows, columns, entries I1 J1 A(I1,J1) <— ... | L entry lines IL JL A(IL,JL) <—

In the MTX foramt, the index for rows or columns starts with 1.

PySAL uses mtx io tools in scipy. Thus, it is subject to all limits that scipy currently has. Reengineering might be required, since scipy currently reads in the entire entry into memory.

References

MTX format specification http://math.nist.gov/MatrixMarket/formats.html

scipy matlab io http://docs.scipy.org/doc/scipy/reference/tutorial/io.html

`FORMATS` = ['mtx']

`MODES` = ['r', 'w']

`by_row`

`cast`(*key*, *typ*)
cast key as typ

`check`()
Prints the contents of the registry

`close`()

`flush`()

`get`(*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

`getType`(*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

`ids`

next()
A FileIO object is its own iterator, see StringIO

open(**args*, ***kwargs*)
Alias for FileIO()

rIds

read(*n=-1*, *sparse=False*)

sparse: boolean if true, return pysal WSP object if false, return pysal W object

seek(*pos*)

tell()
Return id (or offset) of next object

truncate(*size=None*)
Should be implemented by subclasses and redefine this doc string

write(*obj*)

Parameters

- **write(weightsObject)** –
- **a weights object** (*accepts*) –

Returns

- *a MatrixMarket mtx file*
- *write a weights object to the opened mtx file.*

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('wmat.mtx'),'r')
>>> w = testfile.read()
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='.mtx')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created mtx file

```
>>> wnew =  pysal.open(fname,'r').read()
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

Go to the beginning of the test file

```
>>> testfile.seek(0)
```

Create a sparse weights instance from the test file

```
>>> wsp = testfile.read(sparse=True)
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w')
```

Write the sparse weights object into the open file

```
>>> o.write(wsp)
>>> o.close()
```

Read in the newly created mtx file

```
>>> wsp_new =  pysal.open(fname,'r').read(sparse=True)
```

Compare values from old to new

```
>>> wsp_new.s0 == wsp.s0
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

`IOHandlers.pyDbfIO` – PySAL DBF plugin

New in version 1.0.

class `pysal.core.IOHandlers.pyDbfIO.`**`DBF`**(**args*, ***kwargs*)

PySAL DBF Reader/Writer

This DBF handler implements the PySAL DataTable interface.

`header`

list – A list of field names. The header is a python list of strings. Each string is a field name and field name must not be longer than 10 characters.

field_spec

list – A list describing the data types of each field. It is comprised of a list of tuples, each tuple describing a field. The format for the tuples is ("Type",len,precision). Valid Types are 'C' for characters, 'L' for bool, 'D' for data, 'N' or 'F' for number.

Examples

```
>>> import pysal
>>> dbf = pysal.open(pysal.examples.get_path('juvenile.dbf'), 'r')
>>> dbf.header
['ID', 'X', 'Y']
>>> dbf.field_spec
[('N', 9, 0), ('N', 9, 0), ('N', 9, 0)]
```

FORMATS = ['dbf']

MODES = ['r', 'w']

by_col

by_col_array (**args*)

Return columns of table as a numpy array.

Parameters ***args** (*any number of strings of length k*) – names of variables to extract

Returns **implicit**

Return type numpy array of shape (n,k)

Notes

If the variables are not all of the same data type, then numpy rules for casting will result in a uniform type applied to all variables.

If only strings are passed to the function, then an array with those columns will be constructed.

If only one list of strings is passed, the output is identical to those strings being passed.

If at least one list is passed and other strings or lists are passed, this returns a tuple containing arrays constructed from each positional argument.

Examples

```
>>> import pysal as ps
>>> dbf = ps.open(ps.examples.get_path('NAT.dbf'))
>>> hr = dbf.by_col_array('HR70', 'HR80')
>>> hr[0:5]
array([[  0.        ,   8.85582713],
       [  0.        ,  17.20874204],
       [  1.91515848,   3.4507747 ],
       [  1.28864319,   3.26381409],
       [  0.        ,   7.77000777]])
>>> hr = dbf.by_col_array(['HR80', 'HR70'])
>>> hr[0:5]
array([[  8.85582713,   0.        ],
       [ 17.20874204,   0.        ],
       [  3.4507747 ,   1.91515848],
       [  3.26381409,   1.28864319],
       [  7.77000777,   0.        ]])
>>> hr = dbf.by_col_array(['HR80'])
>>> hr[0:5]
array([[  8.85582713],
       [ 17.20874204],
       [  3.4507747 ],
       [  3.26381409],
       [  7.77000777]])
```

Numpy only supports homogeneous arrays. See Notes above.

```
>>> hr = dbf.by_col_array('STATE_NAME', 'HR80')
>>> hr[0:5]
array([['Minnesota', '8.8558271343'],
       ['Washington', '17.208742041'],
       ['Washington', '3.4507746989'],
       ['Washington', '3.2638140931'],
       ['Washington', '7.77000777']],
      dtype='|S20')
```

```
>>> y, X = dbf.by_col_array('STATE_NAME', ['HR80', 'HR70'])
>>> y[0:5]
array([['Minnesota'],
       ['Washington'],
       ['Washington'],
       ['Washington'],
       ['Washington']],
      dtype='|S20')
>>> X[0:5]
array([[  8.85582713,   0.        ],
       [ 17.20874204,   0.        ],
       [  3.4507747 ,   1.91515848],
       [  3.26381409,   1.28864319],
       [  7.77000777,   0.        ]])
```

by_row

cast (*key*, *typ*)
cast key as typ

check ()
Prints the contents of the registry

close ()

flush ()

get (*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType (*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next ()
A FileIO object is its own iterator, see StringIO

open(**args*, ***kwargs*)
Alias for FileIO()

rIds

read(*n=-1*)
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.

read_record(*i*)

seek(*i*)

tell()
Return id (or offset) of next object

to_df(*n=-1*, *read_shp=None*, ***df_kws*)

truncate(*size=None*)
Should be implemented by subclasses and redefine this doc string

write(*obj*)

`IOHandlers.pyShpIO` – Shapefile plugin

The `IOHandlers.pyShpIO` Shapefile Plugin for PySAL's FileIO System

New in version 1.0. PySAL ShapeFile Reader and Writer based on pure python shapefile module.

class `pysal.core.IOHandlers.pyShpIO.`**PurePyShpWrapper**(**args*, ***kwargs*)
FileIO handler for ESRI ShapeFiles.

Notes

This class wraps _pyShpIO's shp_file class with the PySAL FileIO API. shp_file can be used without PySAL.

Formats
list – A list of support file extensions

Modes
list – A list of support file modes

Examples

```
>>> import tempfile
>>> f = tempfile.NamedTemporaryFile(suffix='.shp'); fname = f.name; f.close()
>>> import pysal
>>> i = pysal.open(pysal.examples.get_path('10740.shp'),'r')
>>> o = pysal.open(fname,'w')
>>> for shp in i:
...     o.write(shp)
>>> o.close()
>>> open(pysal.examples.get_path('10740.shp'),'rb').read() == open(fname,'rb').
↪read()
True
>>> open(pysal.examples.get_path('10740.shx'),'rb').read() == open(fname[:-1]+'x',
↪'rb').read()
True
```

```
>>> import os
>>> os.remove(fname); os.remove(fname.replace('.shp','.shx'))
```

FORMATS = ['shp', 'shx']

MODES = ['w', 'r', 'wb', 'rb']

by_row

cast (*key*, *typ*)
cast key as typ

check ()
Prints the contents of the registry

close ()

flush ()

get (*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType (*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next ()
A FileIO object is its own iterator, see StringIO

open (**args*, ***kwargs*)
Alias for FileIO()

rIds

read (*n=-1*)
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.

seek (*n*)
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

tell ()
Return id (or offset) of next object

truncate (*size=None*)
Should be implemented by subclasses and redefine this doc string

write (*obj*)
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

IOHandlers.stata_txt — STATA plugin

New in version 1.2.

class pysal.core.IOHandlers.stata_txt.**StataTextIO** (**args*, ***kwargs*)
Opens, reads, and writes weights file objects in STATA text format.

Spatial weights objects in the STATA text format are used in STATA sppack library through the spmat command. This format is a simple text file delimited by a whitespace. The spmat command does not specify which file extension to use. But, txt seems the default file extension, which is assumed in PySAL.

The first line of the STATA text file is a header including the number of observations. After this header line, it includes at least one data column that contains unique ids or record numbers of observations. When an id variable is not specified for the original spatial weights matrix in STATA, record numbers are used to identify individual observations, and the record numbers start with one. The spmat command seems to allow only integer IDs, which is also assumed in PySAL.

A STATA text file can have one of the following structures according to its export options in STATA.

Structure 1: encoding using the list of neighbor ids [Line 1] [Number_of_Observations] [Line 2] [ID_of_Obs_1] [ID_of_Neighbor_1_of_Obs_1] [ID_of_Neighbor_2_of_Obs_1] .... [ID_of_Neighbor_m_of_Obs_1] [Line 3] [ID_of_Obs_2] [Line 4] [ID_of_Obs_3] [ID_of_Neighbor_1_of_Obs_3] [ID_of_Neighbor_2_of_Obs_3] ... Note that for island observations their IDs are still recorded.

Structure 2: encoding using a full matrix format [Line 1] [Number_of_Observations] [Line 2] [ID_of_Obs_1] [w_11] [w_12] ... [w_1n] [Line 3] [ID_of_Obs_2] [w_21] [w_22] ... [w_2n] [Line 4] [ID_of_Obs_3] [w_31] [w_32] ... [w_3n] ... [Line n+1] [ID_of_Obs_n] [w_n1] [w_n2] ... [w_nn] where w_ij can be a form of general weight. That is, w_ij can be both a binary value or a general numeric value. If an observation is an island, all of its w columns contains 0.

References

Drukker D.M., Peng H., Prucha I.R., and Raciborski R. (2011) “Creating and managing spatial-weighting matrices using the spmat command”

Notes

The spmat command allows users to add any note to a spatial weights matrix object in STATA. However, all those notes are lost when the matrix is exported. PySAL also does not take care of those notes.

FORMATS = ['stata_text']

MODES = ['r', 'w']

by_row

cast (*key*, *typ*)
cast key as typ

check ()
Prints the contents of the registry

close ()

flush ()

get (*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType (*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next ()
A FileIO object is its own iterator, see StringIO

open (**args*, ***kwargs*)
Alias for FileIO()

rIds

read (*n=-1*)

seek (*pos*)

tell ()
Return id (or offset) of next object

truncate (*size=None*)
Should be implemented by subclasses and redefine this doc string

write (*obj*, *matrix_form=False*)

Parameters

- **write(weightsObject)** –
- **a weights object** (*accepts*) –

Returns

- *a STATA text file*
- *write a weights object to the opened text file.*

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('stata_sparse.txt'),'r',
↪'stata_text')
>>> w = testfile.read()
WARNING: there are 7 disconnected observations
Island ids:  [5, 9, 10, 11, 12, 14, 15]
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='.txt')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w','stata_text')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created text file

```
>>> wnew =  pysal.open(fname,'r','stata_text').read()
WARNING: there are 7 disconnected observations
Island ids:  [5, 9, 10, 11, 12, 14, 15]
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

`IOHandlers.wk1` — Lotus WK1 plugin

New in version 1.2.

class `pysal.core.IOHandlers.wk1.`**`Wk1IO`**`(`**args*, ***kwargs*`)`

MATLAB wk1read.m and wk1write.m that were written by Brian M. Bourgault in 10/22/93

Opens, reads, and writes weights file objects in Lotus Wk1 format.

Lotus Wk1 file is used in Dr. LeSage's MATLAB Econometrics library.

A Wk1 file holds a spatial weights object in a full matrix form without any row and column headers. The maximum number of columns supported in a Wk1 file is 256. Wk1 starts the row (column) number from 0 and uses little endian binary endcoding. In PySAL, when the number of observations is n, it is assumed that each cell of a n*n(=m) matrix either is a blank or have a number.

The internal structure of a Wk1 file written by PySAL is as follows: [BOF][DIM][CPI][CAL][CMODE][CORD][SPLIT][SYNC][CURS][WIN][HCOL][MRG][LBL][CELL_1]...[CELL_m][EOF] where [CELL_k] equals to [DTYPE][DLEN][DFORMAT][CINDEX][CVALUE]. The parts between [BOF] and [CELL_1] are variable according to the software program used to write a wk1 file. While reading a wk1 file, PySAL ignores them. Each part of this structure is detailed below.

Table 3.2: Lotus WK1 fields

Part	Description	Data Type	Length	Value
[BOF]	Begining of field	unsigned character	6	0,0,2,0,6,4
[DIM]	Matrix dimension			
[DIMDTYPE] [DIMLEN] [DIMVAL]	Type of dim. rec Length of dim. rec Value of dim. rec	unsigned short unsigned short unsigned short	2 2 8	6 8 0,0,n,n
[CPI]	CPI			
[CPITYPE] [CPILEN] [CPIVAL]	Type of cpi rec Length of cpi rec Value of cpi rec	unsigned short unsigned short unsigned char	2 2 6	150 6 0,0,0,0,0,0
[CAL]	calcount			
[CALTYPE] [CALLEN] [CALVAL]	Type of calcount rec Length calcount rec Value of calcount rec	unsigned short unsigned short unsigned char	2 2 1	47 1 0
[CMODE]	calmode			
[CMODETYP] [CMODELEN] [CMODEVAL]	Type of calmode rec Length of calmode rec Value of calmode rec	unsigned short unsigned short signed char	2 2 1	2 1 0
[CORD]	calorder			
[CORDTYPE] [CORDLEN] [CORDVAL]	Type of calorder rec Length calorder rec Value of calorder rec	unsigned short unsigned short signed char	2 2 1	3 1 0
[SPLIT]	split			
[SPLTYPE] [SPLLEN] [SPLVAL]	Type of split rec Length of split rec Value of split rec	unsigned short unsigned short signed char	2 2 1	4 1 0
[SYNC]	sync			
[SYNCTYP] [SYNCLEN] [SYNCVAL]	Type of sync rec Length of sync rec Value of sync rec	unsigned short unsigned short singed char	2 2 1	5 1 0
[CURS]	cursor			
[CURSTYP] [CURSLEN] [CURSVAL]	Type of cursor rec Length of cursor rec Value of cursor rec	unsigned short unsigned short signed char	2 2 1	49 1 1
[WIN]	window			
[WINTYPE] [WINLEN] [WINVAL1] [WINVAL2] [WINVAL3]	Type of window rec Length of window rec Value 1 of window rec Value 2 of window rec Value 3 of window rec	unsigned short unsigned short unsigned short signed char unsigned short	2 2 4 2 26	7 32 0,0 113,0 10,n,n,0,0,0,0,0,0,0,0,72,0
[HCOL]	hidcol			
[HCOLTYP] [HCOLLEN] [HCOLVAL]	Type of hidcol rec Length of hidcol rec Value of hidcol rec	unsigned short unsigned short signed char	2 2 32	100 32 0*32
[MRG]	margins			
[MRGTYPE] [MRGLEN] [MRGVAL]	Type of margins rec Length of margins rec Value of margins rec	unsigned short unsigned short unsigned short	2 2 10	40 10 4,76,66,2,2

FORMATS = ['wk1']

MODES = ['r', 'w']

by_row

cast (*key*, *typ*)
cast key as typ

check ()
Prints the contents of the registry

close ()

flush ()

get (*n*)
Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType (*dataPath*, *mode*, *dataFormat=None*)
Parse the dataPath and return the data type

ids

next ()
A FileIO object is its own iterator, see StringIO

open (**args*, ***kwargs*)
Alias for FileIO()

rIds

read (*n=-1*)

seek (*pos*)

tell ()
Return id (or offset) of next object

truncate (*size=None*)
Should be implemented by subclasses and redefine this doc string

varName

write (*obj*)

Parameters

- **write(weightsObject)** –
- **a weights object** (*accepts*) –

Returns

- *a Lotus wk1 file*
- *write a weights object to the opened wk1 file.*

Examples

```
>>> import tempfile, pysal, os
>>> testfile = pysal.open(pysal.examples.get_path('spat-sym-us.wk1'),'r')
>>> w = testfile.read()
```

Create a temporary file for this example

```
>>> f = tempfile.NamedTemporaryFile(suffix='.wk1')
```

Reassign to new var

```
>>> fname = f.name
```

Close the temporary named file

```
>>> f.close()
```

Open the new file in write mode

```
>>> o = pysal.open(fname,'w')
```

Write the Weights object into the open file

```
>>> o.write(w)
>>> o.close()
```

Read in the newly created text file

```
>>> wnew =  pysal.open(fname,'r').read()
```

Compare values from old to new

```
>>> wnew.pct_nonzero == w.pct_nonzero
True
```

Clean up temporary file created for this example

```
>>> os.remove(fname)
```

`IOHandlers.wkt` – Well Known Text (geometry) plugin

New in version 1.0.

PySAL plugin for Well Known Text (geometry)

class `pysal.core.IOHandlers.wkt.`**`WKTReader`**(**args*, ***kwargs*)

Parameters

- **`Well-Known Text`** (*Reads*) –
- **`a list of PySAL Polygon objects`** (*Returns*) –

Examples

Read in WKT-formatted file

```
>>> import pysal
>>> f = pysal.open(pysal.examples.get_path('stl_hom.wkt'), 'r')
```

Convert wkt to pysal polygons

```
>>> polys = f.read()
```

Check length

```
>>> len(polys)
78
```

Return centroid of polygon at index 1

```
>>> polys[1].centroid
(-91.19578469430738, 39.990883050220845)
```

Type dir(polys[1]) at the python interpreter to get a list of supported methods

FORMATS = ['wkt']

MODES = ['r']

by_row

cast (*key*, *typ*)
    cast key as typ

check ()
    Prints the contents of the registry

close ()

flush ()

get (*n*)
    Seeks the file to n and returns n If .ids is set n should be an id, else, n should be an offset

getType (*dataPath*, *mode*, *dataFormat=None*)
    Parse the dataPath and return the data type

ids

next ()
    A FileIO object is its own iterator, see StringIO

open ()

rIds

read (*n=-1*)
    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.

seek (*n*)

tell ()
    Return id (or offset) of next object

truncate (*size=None*)
    Should be implemented by subclasses and redefine this doc string

write (*obj*)
    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

`pysal.esda` — Exploratory Spatial Data Analysis

`esda.gamma` — Gamma statistics for spatial autocorrelation

New in version 1.4. Gamma index for spatial autocorrelation

class `pysal.esda.gamma.`**`Gamma`**(*y*, *w*, *operation='c'*, *standardize='no'*, *permutations=999*)

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

`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

Examples

use same example as for join counts to show similarity

```
>>> import pysal, numpy as np
>>> w=pysal.lat2W(4,4)
>>> y=np.ones(16)
>>> y[0:8]=0
>>> np.random.seed(12345)
>>> g = pysal.Gamma(y,w)
>>> g.g
20.0
>>> round(g.g_z, 3)
3.188
>>> g.p_sim_g
0.0030000000000000001
>>> g.min_g
0.0
>>> g.max_g
20.0
>>> g.mean_g
11.093093093093094
>>> np.random.seed(12345)
>>> g1 = pysal.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 = pysal.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 = pysal.Gamma(y,w,standardize='y')
>>> g3.g
32.0
>>> round(g3.g_z, 3)
3.706
>>> g3.p_sim_g
0.001
>>> g3.min_g
-48.0
>>> g3.max_g
20.0
```

```
>>> g3.mean_g
-3.2472472472472473
>>> np.random.seed(12345)
>>> def func(z,i,j):
...     q = z[i]*z[j]
...     return q
...
>>> g4 = pysal.Gamma(y,w,operation=func)
>>> g4.g
20.0
>>> round(g4.g_z, 3)
3.188
>>> g4.p_sim_g
0.0030000000000000001
```

p_sim
new name to fit with Moran module

esda.geary — Geary's C statistics for spatial autocorrelation

New in version 1.0. Geary's C statistic for spatial autocorrelation

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

Parameters

- **y** (*array*) – (n, 1) attribute vector
- **w** (W) – spatial weights
- **transformation** (*{'B', 'R', 'D', 'U', 'V'}*) – weights transformation, default is binary. Other options include "R": row-standardized, "D": doubly-standardized, "U": untransformed (general weights), "V": variance-stabilizing.
- **permutations** (*int*) – number of random permutations for calculation of pseudo-p_values

y
array – original variable

w
W – 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: sptial 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 (one-tailed)

Examples

```
>>> import pysal
>>> w = pysal.open(pysal.examples.get_path("book.gal")).read()
>>> f = pysal.open(pysal.examples.get_path("book.txt"))
>>> y = np.array(f.by_col['y'])
>>> c = Geary(y,w,permutations=0)
>>> print round(c.C,7)
0.3330108
>>> print round(c.p_norm,7)
9.2e-05
>>>
```

classmethod by_col (*df*, *cols*, *w=None*, *inplace=False*, *pvalue='sim'*, *outvals=None*, ***stat_kws*)
Function to compute a Geary statistic on a dataframe

Parameters

- **df** (*pandas.DataFrame*) – a pandas dataframe with a geometry column
- **cols** (*string or list of string*) – name or list of names of columns to use to compute the statistic
- **w** (*pysal weights object*) – a weights object aligned with the dataframe. If not provided, this is searched for in the dataframe's metadata
- **inplace** (*bool*) – a boolean denoting whether to operate on the dataframe inplace or to return a series contaning the results of the computation. If operating inplace, with default configurations, the derived columns will be named like 'column_geary' and 'column_p_sim'

- **pvalue** (*string*) – a string denoting which pvalue should be returned. Refer to the the Geary statistic's documentation for available p-values
- **outvals** (*list of strings*) – list of arbitrary attributes to return as columns from the Geary statistic
- ****stat_kws** (*keyword arguments*) – options to pass to the underlying statistic. For this, see the documentation for the Geary statistic.

Returns

- *If inplace, None, and operation is conducted on dataframe in memory. Otherwise,*
- *returns a copy of the dataframe with the relevant columns attached.*

See also:

`For, refer`

`esda.getisord` — Getis-Ord statistics for spatial association

New in version 1.0. Getis and Ord G statistic for spatial autocorrelation

class `pysal.esda.getisord.`**`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

y
array – original variable

w
W – DistanceBand W spatial weights based on distance band

permutation
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

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

Notes

Moments are based on normality assumption.

Examples

```
>>> from pysal.weights.Distance import DistanceBand
>>> 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 = 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 >>> g = G(y,w)

Examining the results >>> print "%.8f" % g.G 0.55709779

```
>>> print "%.4f" % g.p_norm
0.1729
```

classmethod by_col (*df*, *cols*, *w=None*, *inplace=False*, *pvalue='sim'*, *outvals=None*, ***stat_kws*)
Function to compute a G statistic on a dataframe

Parameters

- **df** (*pandas.DataFrame*) – a pandas dataframe with a geometry column
- **cols** (*string or list of string*) – name or list of names of columns to use to compute the statistic
- **w** (*pysal weights object*) – a weights object aligned with the dataframe. If not provided, this is searched for in the dataframe's metadata
- **inplace** (*bool*) – a boolean denoting whether to operate on the dataframe inplace or to return a series contaning the results of the computation. If operating inplace, the derived columns will be named 'column_g'

- **pvalue** (*string*) – a string denoting which pvalue should be returned. Refer to the the G statistic's documentation for available p-values
- **outvals** (*list of strings*) – list of arbitrary attributes to return as columns from the G statistic
- ****stat_kws** (*keyword arguments*) – options to pass to the underlying statistic. For this, see the documentation for the G statistic.

Returns

- *If inplace, None, and operation is conducted on dataframe in memory. Otherwise,*
- *returns a copy of the dataframe with the relevant columns attached.*

See also:

`For, refer`

class `pysal.esda.getisord.`**`G_Local`**(*y*, *w*, *transform='R'*, *permutations=999*, *star=False*)

Generalized Local G Autocorrelation Statistic [Getis1992], [Ord1995], [Getis1996] .

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)

y
array – original variable

w
DistanceBand W – original weights object

permutations
int – the number of permutations

Gs
array – of floats, the value of the orginal 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 permutated 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)

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).

Examples

```
>>> from pysal.weights.Distance import DistanceBand
>>> 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 = DistanceBand(points,threshold=15)
```

Prepareing a variable

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

Applying Getis and Ord local G test using a binary weights object >>> lg = G_Local(y,w,transform='B')

Examining the results >>> lg.Zs array([-1.0136729 , -0.04361589, 1.31558703, -0.31412676, 1.15373986,

1.77833941])

```
>>> lg.p_sim[0]
0.10100000000000001
```

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

Applying Getis and Ord local G* test using a binary weights object >>> lg_star = G_Local(y,w,transform='B',star=True)

Examining the results >>> lg_star.Zs array([-1.39727626, -0.28917762, 0.65064964, -0.28917762, 1.23452088,

2.02424331])

```
>>> lg_star.p_sim[0]
0.10100000000000001
```

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

Applying Getis and Ord local G test using a row-standardized weights object >>> lg = G_Local(y,w,transform='R')

Examining the results >>> lg.Zs array([-0.62074534, -0.01780611, 1.31558703, -0.12824171, 0.28843496,

1.77833941])

```
>>> lg.p_sim[0]
0.10100000000000001
```

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

Applying Getis and Ord local G* test using a row-standardized weights object >>> lg_star = G_Local(y,w,transform='R',star=True)

Examining the results >>> lg_star.Zs array([-0.62488094, -0.09144599, 0.41150696, -0.09144599, 0.24690418,

1.28024388])

```
>>> lg_star.p_sim[0]
0.10100000000000001
```

classmethod by_col (*df*, *cols*, *w=None*, *inplace=False*, *pvalue='sim'*, *outvals=None*, ***stat_kws*)
Function to compute a G_Local statistic on a dataframe

Parameters

- **df** (*pandas.DataFrame*) – a pandas dataframe with a geometry column
- **cols** (*string or list of string*) – name or list of names of columns to use to compute the statistic
- **w** (*pysal weights object*) – a weights object aligned with the dataframe. If not provided, this is searched for in the dataframe's metadata
- **inplace** (*bool*) – a boolean denoting whether to operate on the dataframe inplace or to return a series contaning the results of the computation. If operating inplace, the derived columns will be named 'column_g_local'
- **pvalue** (*string*) – a string denoting which pvalue should be returned. Refer to the the G_Local statistic's documentation for available p-values
- **outvals** (*list of strings*) – list of arbitrary attributes to return as columns from the G_Local statistic
- ****stat_kws** (*keyword arguments*) – options to pass to the underlying statistic. For this, see the documentation for the G_Local statistic.

Returns

- *If inplace, None, and operation is conducted on dataframe in memory. Otherwise,*
- *returns a copy of the dataframe with the relevant columns attached.*

See also:

`For, refer`

`esda.join_counts` — Spatial autocorrelation statistics for binary attributes

New in version 1.0. Spatial autocorrelation for binary attributes

class `pysal.esda.join_counts.`**`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

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

Examples

Replicate example from anselin and rey

```
>>> import numpy as np
>>> w = pysal.lat2W(4, 4)
>>> y = np.ones(16)
>>> y[0:8] = 0
>>> np.random.seed(12345)
>>> jc = pysal.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
>>> jc.p_sim_bb
0.0030000000000000001
>>> np.mean(jc.sim_bb)
5.5465465465465469
>>> 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)
7.0
>>>
```

classmethod by_col (*df*, *cols*, *w=None*, *inplace=False*, *pvalue='sim'*, *outvals=None*, ***stat_kws*)
Function to compute a Join_Count statistic on a dataframe

Parameters

- **df** (*pandas.DataFrame*) – a pandas dataframe with a geometry column

- **cols** (*string or list of string*) – name or list of names of columns to use to compute the statistic
- **w** (*pysal weights object*) – a weights object aligned with the dataframe. If not provided, this is searched for in the dataframe's metadata
- **inplace** (*bool*) – a boolean denoting whether to operate on the dataframe inplace or to return a series contaning the results of the computation. If operating inplace, the derived columns will be named 'column_join_count'
- **pvalue** (*string*) – a string denoting which pvalue should be returned. Refer to the the Join_Count statistic's documentation for available p-values
- **outvals** (*list of strings*) – list of arbitrary attributes to return as columns from the Join_Count statistic
- ****stat_kws** (*keyword arguments*) – options to pass to the underlying statistic. For this, see the documentation for the Join_Count statistic.

Returns

- *If inplace, None, and operation is conducted on dataframe in memory. Otherwise,*
- *returns a copy of the dataframe with the relevant columns attached.*

See also:

`For, refer`

esda.mapclassify — Choropleth map classification

New in version 1.0. A module of classification schemes for choropleth mapping.

class pysal.esda.mapclassify.**Map_Classifier**(*y*)

Abstract class for all map classifications [Slocum2008]

For an array y of n values, a map classifier places each value y_i into one of k mutually exclusive and exhaustive classes. Each classifer defines the classes based on different criteria, but in all cases the following hold for the classifiers in PySAL:

$$C_j^l < y_i \le C_j^u \ \ \forall i \in C_j$$

where C_j denotes class j which has lower bound C_j^l and upper bound C_j^u.

Map Classifiers Supported

- *Box_Plot*
- *Equal_Interval*
- *Fisher_Jenks*
- *Fisher_Jenks_Sampled*
- *HeadTail_Breaks*
- *Jenks_Caspall*
- *Jenks_Caspall_Forced*
- *Jenks_Caspall_Sampled*
- *Max_P_Classifier*
- *Maximum_Breaks*

- *Natural_Breaks*
- *Quantiles*
- *Percentiles*
- *Std_Mean*
- *User_Defined*

Utilities:

In addition to the classifiers, there are several utility functions that can be used to evaluate the properties of a specific classifier for different parameter values, or for automatic selection of a classifier and number of classes.

- *gadf*
- *K_classifiers*

find_bin(*x*)
Sort input or inputs according to the current bin estimate

Parameters **x** (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()
Goodness of absolute deviation of fit

get_tss()
Total sum of squares around class means

Returns sum of squares over all class means

classmethod make(**args*, ***kwargs*)
Configure and create a classifier that will consume data and produce classifications, given the configuration options specified by this function.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.

- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
```

```
6    3    0    4
7    3    0    4
8    4    0    4
9    4    0    4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update(*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

pysal.esda.mapclassify.**quantile**(*y*, *k=4*)
Calculates the quantiles for an array

Parameters

- **y** (*array*) – (n,1), values to classify
- **k** (*int*) – number of quantiles

Returns **implicit** – (n,1), quantile values

Return type array

Examples

```
>>> x = np.arange(1000)
>>> quantile(x)
array([ 249.75,  499.5 ,  749.25,  999.  ])
>>> quantile(x, k = 3)
array([ 333.,  666.,  999.])
>>>
```

Note that if there are enough ties that the quantile values repeat, we collapse to pseudo quantiles in which case the number of classes will be less than k

```
>>> x = [1.0] * 100
>>> x.extend([3.0] * 40)
>>> len(x)
140
```

```
>>> y = np.array(x)
>>> quantile(y)
array([ 1.,  3.])
```

class `pysal.esda.mapclassify.`**`Box_Plot`**(*y*, *hinge=1.5*)

Box_Plot Map Classification

Parameters

- **`y`** (*`array`*) – attribute to classify
- **`hinge`** (*`float`*) – multiplier for IQR

`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

Notes

The bins are set as follows:

```
bins[0] = q[0]-hinge*IQR
bins[1] = q[0]
bins[2] = q[1]
bins[3] = q[2]
bins[4] = q[2]+hinge*IQR
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

```
>>> cal = load_example()
>>> bp = Box_Plot(cal)
>>> bp.bins
array([ -5.28762500e+01,   2.56750000e+00,   9.36500000e+00,
         3.95300000e+01,   9.49737500e+01,   4.11145000e+03])
>>> bp.counts
array([ 0, 15, 14, 14,  6,  9])
>>> bp.high_outlier_ids
array([ 0,  6, 18, 29, 33, 36, 37, 40, 42])
```

```
>>> cal[bp.high_outlier_ids]
array([  329.92,   181.27,   370.5 ,   722.85,   192.05,   110.74,
        4111.45,   317.11,   264.93])
>>> bx = Box_Plot(np.arange(100))
>>> bx.bins
array([ -49.5 ,   24.75,   49.5 ,   74.25,  148.5 ])
```

find_bin(*x*)
Sort input or inputs according to the current bin estimate

Parameters **x** (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()
Goodness of absolute deviation of fit

get_tss()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
         0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
```

```
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update(*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.mapclassify.**Equal_Interval**(*y*, *k=5*)
Equal Interval Classification

Parameters

- **y** (*array*) – (n,1), values to classify
- **k** (*int*) – number of classes required

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

Examples

```
>>> cal = load_example()
>>> ei = 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 ])
>>>
```

Notes

Intervals defined to have equal width:

$$bins_j = min(y) + w * (j + 1)$$

with $w = \frac{max(y) - min(j)}{k}$

find_bin (*x*)
Sort input or inputs according to the current bin estimate

Parameters **x** (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm ()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf ()
Goodness of absolute deviation of fit

get_tss ()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
```

```
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update(*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (`array`) – (n,1) array of data to classify
- **inplace** (`bool`) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (`Additional`) –
- **of the class. For documentation, check the class constructor.** (`function`) –

class `pysal.esda.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

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

Examples

```
>>> cal = load_example()
>>> fj = Fisher_Jenks(cal)
>>> fj.adcm
799.24000000000001
>>> fj.bins
array([   75.29,   192.05,   370.5 ,   722.85,  4111.45])
>>> fj.counts
array([49,  3,  4,  1,  1])
>>>
```

find_bin(*x*)
Sort input or inputs according to the current bin estimate

Parameters **x** (`array or numeric`) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()

Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()

Goodness of absolute deviation of fit

get_tss()

Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update (*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class `pysal.esda.mapclassify.`**`Fisher_Jenks_Sampled`**(*y*, *k=5*, *pct=0.1*, *truncate=True*)

Fisher Jenks optimal classifier - mean based using 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, unless truncate is False
- **truncate** (*boolean*) – truncate pct in cases where pct * n > 1000., (Default True)

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

Examples

(Turned off due to timing being different across hardware)

find_bin(*x*)

Sort input or inputs according to the current bin estimate

Parameters **x** (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()

Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()

Goodness of absolute deviation of fit

get_tss()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update (*y=None*, *inplace=False*, ***kwargs*)

Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.mapclassify.**Jenks_Caspall** (*y*, *k=5*)

Jenks Caspall Map Classification

Parameters

- **y** (*array*) – (n,1), values to classify
- **k** (*int*) – number of classes required

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

Examples

```
>>> cal = load_example()
>>> jc = Jenks_Caspall(cal, k = 5)
>>> jc.bins
array([  1.81000000e+00,   7.60000000e+00,   2.98200000e+01,
         1.81270000e+02,   4.11145000e+03])
>>> jc.counts
array([14, 13, 14, 10,  7])
```

find_bin(*x*)
Sort input or inputs according to the current bin estimate

Parameters **x** (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()
Goodness of absolute deviation of fit

get_tss()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- **`*args`** (*`required positional arguments`*) – all positional arguments required by the classifier, excluding the input data.
- **`rolling`** (*`bool`*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **`return_object`** (*`bool`*) – a boolean configuring the outputted classifier to return the classifier object or not
- **`return_bins`** (*`bool`*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **`return_counts`** (*`bool`*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0  3.000000  10.000000  -5.000000
1  3.555556   8.888889  -2.777778
2  4.111111   7.777778  -0.555556
3  4.666667   6.666667   1.666667
4  5.222222   5.555556   3.888889
5  5.777778   4.444444   6.111111
6  6.333333   3.333333   8.333333
7  6.888888   2.222222  10.555556
8  7.444444   1.111111  12.777778
9  8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
```

```
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update(*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.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

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

Examples

```
>>> cal = load_example()
>>> jcf = Jenks_Caspall_Forced(cal, k = 5)
>>> jcf.k
5
>>> jcf.bins
array([[  1.34000000e+00],
       [  5.90000000e+00],
       [  1.67000000e+01],
       [  5.06500000e+01],
       [  4.11145000e+03]])
>>> jcf.counts
array([12, 12, 13,  9, 12])
>>> jcf4 = Jenks_Caspall_Forced(cal, k = 4)
>>> jcf4.k
4
>>> jcf4.bins
array([[  2.51000000e+00],
       [  8.70000000e+00],
       [  3.66800000e+01],
       [  4.11145000e+03]])
>>> jcf4.counts
array([15, 14, 14, 15])
>>>
```

find_bin(*x*)
Sort input or inputs according to the current bin estimate

Parameters **x** (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()
Goodness of absolute deviation of fit

get_tss()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
         0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
```

```
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update(*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.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

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

Examples

```
>>> cal = load_example()
>>> x = np.random.random(100000)
>>> jc = Jenks_Caspall(x)
>>> jcs = Jenks_Caspall_Sampled(x)
>>> jc.bins
array([ 0.19770952,  0.39695769,  0.59588617,  0.79716865,  0.99999425])
>>> jcs.bins
array([ 0.18877882,  0.39341638,  0.6028286 ,  0.80070925,  0.99999425])
>>> jc.counts
array([19804, 20005, 19925, 20178, 20088])
>>> jcs.counts
array([18922, 20521, 20980, 19826, 19751])
>>>
```

# 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

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.

find_bin(*x*)
Sort input or inputs according to the current bin estimate

Parameters **x** (`array or numeric`) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()
Goodness of absolute deviation of fit

get_tss()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0           1           2
```

```
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update(*y=None*, *inplace=False*, ***kwargs*)

Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.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

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

Examples

```
>>> import pysal
>>> cal = pysal.esda.mapclassify.load_example()
>>> mp = pysal.Max_P_Classifier(cal)
>>> mp.bins
array([    8.7 ,    16.7 ,    20.47,    66.26,  4111.45])
>>> mp.counts
array([29,  8,  1, 10, 10])
```

find_bin (*x*)

Sort input or inputs according to the current bin estimate

Parameters **x** (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm ()

Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf ()

Goodness of absolute deviation of fit

get_tss ()

Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0  3.000000  10.000000  -5.000000
1  3.555556   8.888889  -2.777778
2  4.111111   7.777778  -0.555556
3  4.666667   6.666667   1.666667
4  5.222222   5.555556   3.888889
5  5.777778   4.444444   6.111111
6  6.333333   3.333333   8.333333
7  6.888888   2.222222  10.555556
8  7.444444   1.111111  12.777778
9  8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
```

```
0    0    4    0
1    0    4    0
2    1    4    0
3    1    3    0
4    2    2    1
5    2    1    2
6    3    0    4
7    3    0    4
8    4    0    4
9    4    0    4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update (*y=None*, *inplace=False*, ***kwargs*)

Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.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

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)

Examples

```
>>> cal = load_example()
>>> mb = 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])
>>>
```

find_bin (*x*)
Sort input or inputs according to the current bin estimate

Parameters **x** (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm ()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf ()
Goodness of absolute deviation of fit

get_tss ()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not

- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
```

```
9    4    0    4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update (*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.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)

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

Examples

```
>>> import numpy
>>> import pysal
>>> numpy.random.seed(123456)
>>> cal = pysal.esda.mapclassify.load_example()
>>> nb = pysal.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 = numpy.array([1] * 50)
>>> x[-1] = 20
>>> nb = pysal.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])
```

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.

find_bin(*x*)
Sort input or inputs according to the current bin estimate

Parameters x (`array or numeric`) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()
Goodness of absolute deviation of fit

get_tss()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
```

```
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update(*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.mapclassify.**Quantiles**(*y*, *k=5*)
Quantile Map Classification

Parameters

- **y** (*array*) – (n,1), values to classify
- **k** (*int*) – number of classes required

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

Examples

```
>>> cal = load_example()
>>> q = Quantiles(cal, k = 5)
>>> q.bins
array([  1.46400000e+00,   5.79800000e+00,   1.32780000e+01,
         5.46160000e+01,   4.11145000e+03])
>>> q.counts
array([12, 11, 12, 11, 12])
>>>
```

find_bin(*x*)
Sort input or inputs according to the current bin estimate

Parameters x (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()
Goodness of absolute deviation of fit

get_tss()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not

- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
```

```
9    4    0    4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update (*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.mapclassify.**Percentiles** (*y*, *pct=[1, 10, 50, 90, 99, 100]*)
Percentiles Map Classification

Parameters

- **y** (*array*) – attribute to classify
- **pct** (*array*) – percentiles default=[1,10,50,90,99,100]

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)

Examples

```
>>> cal = load_example()
>>> p = Percentiles(cal)
>>> p.bins
array([  1.35700000e-01,   5.53000000e-01,   9.36500000e+00,
         2.13914000e+02,   2.17994800e+03,   4.11145000e+03])
>>> p.counts
array([ 1,  5, 23, 23,  5,  1])
>>> p2 = Percentiles(cal, pct = [50, 100])
>>> p2.bins
```

```
array([    9.365,   4111.45 ])
>>> p2.counts
array([29, 29])
>>> p2.k
2
```

find_bin (*x*)
Sort input or inputs according to the current bin estimate

Parameters x (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm ()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf ()
Goodness of absolute deviation of fit

get_tss ()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
         0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
```

```
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update (*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.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]

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

Examples

```
>>> cal = load_example()
>>> st = 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 = 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])
>>>
```

find_bin(*x*)
Sort input or inputs according to the current bin estimate

Parameters x (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()
Goodness of absolute deviation of fit

get_tss()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update(*y=None, inplace=False, **kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

class pysal.esda.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 monotically increasing)

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

Examples

```
>>> cal = load_example()
>>> bins = [20, max(cal)]
>>> bins
[20, 4111.4499999999998]
>>> ud = User_Defined(cal, bins)
>>> ud.bins
array([   20.  ,  4111.45])
>>> ud.counts
array([37, 21])
>>> bins = [20, 30]
>>> ud = User_Defined(cal, bins)
>>> ud.bins
array([   20.  ,    30.  ,  4111.45])
>>> ud.counts
array([37,  4, 17])
>>>
```

Notes

If upper bound of user bins does not exceed max(y) we append an additional bin.

find_bin(*x*)

Sort input or inputs according to the current bin estimate

Parameters **x** (*array or numeric*) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()

Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()

Goodness of absolute deviation of fit

get_tss()

Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
         0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update(*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

pysal.esda.mapclassify.**gadf**(*y*, *method='Quantiles'*, *maxk=15*, *pct=0.8*)
Evaluate the Goodness of Absolute Deviation Fit of a Classifier Finds the minimum value of k for which gadf>pct

Parameters

- **y** (*array*) – (n, 1) values to be classified
- **method** (*{'Quantiles, 'Fisher_Jenks', 'Maximum_Breaks', 'Natrual_Breaks'}*) –
- **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

Examples

```
>>> cal = load_example()
>>> qgadf = gadf(cal)
>>> qgadf[0]
15
>>> qgadf[-1]
0.37402575909092828
```

Quantiles fail to exceed 0.80 before 15 classes. If we lower the bar to 0.2 we see quintiles as a result

```
>>> qgadf2 = gadf(cal, pct = 0.2)
>>> qgadf2[0]
5
>>> qgadf2[-1]
0.21710231966462412
>>>
```

Notes

The GADF is defined as:

$$GADF = 1 - \sum_c \sum_{i \in c} |y_i - y_{c,med}| / \sum_i |y_i - y_{med}|$$

where y_{med} is the global median and $y_{c,med}$ is the median for class c.

See also:

K_classifiers

class pysal.esda.mapclassify.**K_classifiers**(*y*, *pct=0.8*)

Evaluate all k-classifers 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

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 classifer

Examples

```
>>> cal = load_example()
>>> ks = K_classifiers(cal)
>>> ks.best.name
'Fisher_Jenks'
>>> ks.best.k
4
>>> ks.best.gadf
0.84810327199081048
>>>
```

Notes

This can be used to suggest a classification scheme.

See also:

gadf

class pysal.esda.mapclassify.**HeadTail_Breaks**(*y*)

Head/tail Breaks Map Classification for Heavy-tailed Distributions

Parameters **y** (*array*) – (n,1), values to classify

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

Examples

```
>>> import numpy as np
>>> np.random.seed(10)
>>> cal = load_example()
>>> htb = 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 = 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])
```

Notes

Head/tail Breaks is a relatively new classification method developed and introduced by [Jiang2013] for data with a heavy-tailed distribution.

Based on contributions by Alessandra Sozzi <alessandra.sozzi@gmail.com>.

find_bin(*x*)
Sort input or inputs according to the current bin estimate

Parameters x (`array or numeric`) – a value or array of values to fit within the estimated bins

Returns

- *a bin index or array of bin indices that classify the input into one of*
- *the classifiers' bins*

get_adcm()
Absolute deviation around class median (ADCM).

Calculates the absolute deviations of each observation about its class median as a measure of fit for the classification method.

Returns sum of ADCM over all classes

get_gadf()
Goodness of absolute deviation of fit

get_tss()
Total sum of squares around class means

Returns sum of squares over all 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.

Note that this like a *partial application* of the relevant class constructor. *make* creates a function that returns classifications; it does not actually do the classification.

If you want to classify data directly, use the appropriate class constructor, like Quantiles, Max_Breaks, etc.

If you *have* a classifier object, but want to find which bins new data falls into, use find_bin.

Parameters

- ***args** (*required positional arguments*) – all positional arguments required by the classifier, excluding the input data.
- **rolling** (*bool*) – a boolean configuring the outputted classifier to use a rolling classifier rather than a new classifier for each input. If rolling, this adds the current data to all of the previous data in the classifier, and rebalances the bins, like a running median computation.
- **return_object** (*bool*) – a boolean configuring the outputted classifier to return the classifier object or not
- **return_bins** (*bool*) – a boolean configuring the outputted classifier to return the bins/breaks or not
- **return_counts** (*bool*) – a boolean configuring the outputted classifier to return the histogram of objects falling into each bin or not

Returns

- *A function that consumes data and returns their bins (and object,*
- *bins/breaks, or counts, if requested).*

Note: This is most useful when you want to run a classifier many times with a given configuration, such as when classifying many columns of an array or dataframe using the same configuration.

Examples

```
>>> import pysal as ps
>>> df = ps.pdio.read_files(ps.examples.get_path('columbus.dbf'))
>>> classifier = ps.Quantiles.make(k=9)
>>> classifier
>>> classifications = df[['HOVAL', 'CRIME', 'INC']].apply(ps.Quantiles.
↪make(k=9))
>>> classifications.head()
    HOVAL  CRIME   INC
0       8      0     7
1       7      1     8
2       2      3     5
3       4      4     0
4       1      6     3
>>> import pandas as pd; from numpy import linspace as lsp
>>> data = [lsp(3,8,num=10), lsp(10, 0, num=10), lsp(-5, 15, num=10)]
>>> data = pd.DataFrame(data).T
>>> data
          0          1          2
0 3.000000  10.000000  -5.000000
1 3.555556   8.888889  -2.777778
2 4.111111   7.777778  -0.555556
3 4.666667   6.666667   1.666667
4 5.222222   5.555556   3.888889
5 5.777778   4.444444   6.111111
6 6.333333   3.333333   8.333333
7 6.888888   2.222222  10.555556
8 7.444444   1.111111  12.777778
9 8.000000   0.000000  15.000000
>>> data.apply(ps.Quantiles.make(rolling=True))
    0   1   3
0   0   4   0
1   0   4   0
2   1   4   0
3   1   3   0
4   2   2   1
5   2   1   2
6   3   0   4
7   3   0   4
8   4   0   4
9   4   0   4
>>> dbf = ps.open(ps.examples.get_path('baltim.dbf'))
>>> data = dbf.by_col_array('PRICE', 'LOTSZ', 'SQFT')
>>> my_bins = [1, 10, 20, 40, 80]
>>> classifications = [ps.User_Defined.make(bins=my_bins)(a) for a in data.T]
>>> len(classifications)
3
>>> print(classifications)
[array([4, 5, 5, 5, 4, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 1, 2, 2, 3, 4, 4, 3, 3,
    ...
    2, 2, 2, 2])]
```

update(*y=None*, *inplace=False*, ***kwargs*)
Add data or change classification parameters.

Parameters

- **y** (*array*) – (n,1) array of data to classify
- **inplace** (*bool*) – whether to conduct the update in place or to return a copy estimated from the additional specifications.
- **parameters provided in **kwargs are passed to the init** (*Additional*) –
- **of the class. For documentation, check the class constructor.** (*function*) –

esda.moran — Moran's I measures of spatial autocorrelation

New in version 1.0.

Moran's I global and local measures of spatial autocorrelation Moran's I Spatial Autocorrelation Statistics

class `pysal.esda.moran.`**`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 include "B": binary, "D": doubly-standardized, "U": untransformed (general weights), "V": variance-stabilizing.
- **`permutations`** (*`int`*) – number of random permutations for calculation of pseudo-p_values
- **`two_tailed`** (*`boolean`*) – If True (default) analytical p-values for Moran are two tailed, otherwise if False, they are one-tailed.

`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

`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

`seI_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

seI_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

Examples

```
>>> import pysal
>>> w = pysal.open(pysal.examples.get_path("stl.gal")).read()
>>> f = pysal.open(pysal.examples.get_path("stl_hom.txt"))
>>> y = np.array(f.by_col['HR8893'])
>>> mi = Moran(y,  w)
>>> "%7.5f" % mi.I
'0.24366'
>>> mi.EI
-0.012987012987012988
>>> mi.p_norm
0.00027147862770937614
```

SIDS example replicating OpenGeoda

```
>>> w = pysal.open(pysal.examples.get_path("sids2.gal")).read()
>>> f = pysal.open(pysal.examples.get_path("sids2.dbf"))
>>> SIDR = np.array(f.by_col("SIDR74"))
>>> mi = pysal.Moran(SIDR,  w)
>>> "%6.4f" % mi.I
'0.2477'
>>> mi.p_norm
0.0001158330781489969
```

One-tailed

```
>>> mi_1 = pysal.Moran(SIDR,  w, two_tailed=False)
>>> "%6.4f" % mi_1.I
'0.2477'
>>> mi_1.p_norm
5.7916539074498452e-05
```

classmethod by_col (*df, cols, w=None, inplace=False, pvalue='sim', outvals=None, **stat_kws*)
Function to compute a Moran statistic on a dataframe

Parameters

- **df** (*pandas.DataFrame*) – a pandas dataframe with a geometry column

- **cols** (*string or list of string*) – name or list of names of columns to use to compute the statistic
- **w** (*pysal weights object*) – a weights object aligned with the dataframe. If not provided, this is searched for in the dataframe's metadata
- **inplace** (*bool*) – a boolean denoting whether to operate on the dataframe inplace or to return a series contaning the results of the computation. If operating inplace, the derived columns will be named 'column_moran'
- **pvalue** (*string*) – a string denoting which pvalue should be returned. Refer to the the Moran statistic's documentation for available p-values
- **outvals** (*list of strings*) – list of arbitrary attributes to return as columns from the Moran statistic
- ****stat_kws** (*keyword arguments*) – options to pass to the underlying statistic. For this, see the documentation for the Moran statistic.

Returns

- *If inplace, None, and operation is conducted on dataframe in memory. Otherwise,*
- *returns a copy of the dataframe with the relevant columns attached.*

See also:

`For, refer`

class `pysal.esda.moran.`**`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

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 quandrant 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 Ii is further away or extreme from the median of simulated values. It is either extremelyi 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

seI_sim
array – (if permutations>0) standard deviations of Is under permutations.

z_sim
arrray – (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

Examples

```
>>> import pysal as ps
>>> import numpy as np
>>> np.random.seed(10)
>>> w = ps.open(ps.examples.get_path("desmith.gal")).read()
>>> f = ps.open(ps.examples.get_path("desmith.txt"))
>>> y = np.array(f.by_col['z'])
>>> lm = ps.Moran_Local(y, w, transformation = "r", permutations = 99)
>>> lm.q
array([4, 4, 4, 2, 3, 3, 1, 4, 3, 3])
>>> lm.p_z_sim[0]
0.24669152541631179
>>> lm = ps.Moran_Local(y, w, transformation = "r", permutations = 99,
↪                    geoda_quads=True)
>>> lm.q
array([4, 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 unittests that are conditional on architectures

classmethod by_col (*df*, *cols*, *w=None*, *inplace=False*, *pvalue='sim'*, *outvals=None*, ***stat_kws*)
Function to compute a Moran_Local statistic on a dataframe

Parameters

- **df** (*pandas.DataFrame*) – a pandas dataframe with a geometry column
- **cols** (*string or list of string*) – name or list of names of columns to use to compute the statistic
- **w** (*pysal weights object*) – a weights object aligned with the dataframe. If not provided, this is searched for in the dataframe's metadata
- **inplace** (*bool*) – a boolean denoting whether to operate on the dataframe inplace or to return a series contaning the results of the computation. If operating inplace, the derived columns will be named 'column_moran_local'

- **pvalue** (*string*) – a string denoting which pvalue should be returned. Refer to the the Moran_Local statistic's documentation for available p-values
- **outvals** (*list of strings*) – list of arbitrary attributes to return as columns from the Moran_Local statistic
- ****stat_kws** (*keyword arguments*) – options to pass to the underlying statistic. For this, see the documentation for the Moran_Local statistic.

Returns

- *If inplace, None, and operation is conducted on dataframe in memory. Otherwise,*
- *returns a copy of the dataframe with the relevant columns attached.*

See also:

`For, refer`

class `pysal.esda.moran.`**`Moran_BV`**(*x*, *y*, *w*, *transformation='r'*, *permutations=999*)

Bivariate Moran's I

Parameters

- **x** (*array*) – x-axis variable
- **y** (*array*) – 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

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

Notes

Inference is only based on permutations as analytical results are none too reliable.

Examples

```
>>> import pysal
>>> import numpy as np
```

Set random number generator seed so we can replicate the example

```
>>> 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

```
>>> f = pysal.open(pysal.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

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

Create an instance of Moran_BV

```
>>> mbi = Moran_BV(SIDR79,  SIDR74,  w)
```

What is the bivariate Moran's I value

```
>>> print mbi.I
0.156131961696
```

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

```
>>> mbi.p_z_sim
0.0014186617421765302
```

classmethod by_col(*df*, *x*, *y=None*, *w=None*, *inplace=False*, *pvalue='sim'*, *outvals=None*, ***stat_kws*)
Function to compute a Moran_BV statistic on a dataframe

Parameters

- **df** (*pandas.DataFrame*) – a pandas dataframe with a geometry column
- **X** (*list of strings*) – column name or list of column names to use as X values to compute the bivariate statistic. If no Y is provided, pairwise comparisons among these variates are used instead.

- **Y** (*list of strings*) – column name or list of column names to use as Y values to compute the bivariate statistic. if no Y is provided, pariwise comparisons among the X variates are used instead.
- **w** (*pysal weights object*) – a weights object aligned with the dataframe. If not provided, this is searched for in the dataframe's metadata
- **inplace** (*bool*) – a boolean denoting whether to operate on the dataframe inplace or to return a series contaning the results of the computation. If operating inplace, the derived columns will be named 'column_moran_local'
- **pvalue** (*string*) – a string denoting which pvalue should be returned. Refer to the the Moran_BV statistic's documentation for available p-values
- **outvals** (*list of strings*) – list of arbitrary attributes to return as columns from the Moran_BV statistic
- ****stat_kws** (*keyword arguments*) – options to pass to the underlying statistic. For this, see the documentation for the Moran_BV statistic.

Returns

- *If inplace, None, and operation is conducted on dataframe in memory. Otherwise,*
- *returns a copy of the dataframe with the relevant columns attached.*

See also:

`For`, `refer`

`pysal.esda.moran.`**`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** (*list*) – sequence of variables
- **w** (*W*) – a spatial weights object
- **permutations** (*int*) – number of permutations
- **varnames** (*list*) – strings for variable names. If specified runtime summary is printed

Returns **results** – (i, j) is the key for the pair of variables, values are the Moran_BV objects.

Return type dictionary

Examples

```
>>> import pysal
```

open dbf

```
>>> f = pysal.open(pysal.examples.get_path("sids2.dbf"))
```

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

```
>>> 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

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

create an instance of Moran_BV_matrix

```
>>> res = Moran_BV_matrix(vars,  w,  varnames = varnames)
```

check values

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

class `pysal.esda.moran.`**`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

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 quandrant 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 values. It is either extremelyi 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

seI_sim
array – (if permutations>0) standard deviations of Is under permutations.

z_sim
arrray – (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

Examples

```
>>> import pysal as ps
>>> import numpy as np
>>> np.random.seed(10)
>>> w = ps.open(ps.examples.get_path("sids2.gal")).read()
>>> f = ps.open(ps.examples.get_path("sids2.dbf"))
>>> x = np.array(f.by_col['SIDR79'])
>>> y = np.array(f.by_col['SIDR74'])
>>> lm = ps.Moran_Local_BV(x, y, w, transformation = "r",                    ␣
↪     permutations = 99)
>>> lm.q[:10]
array([3, 4, 3, 4, 2, 1, 4, 4, 2, 4])
>>> lm.p_z_sim[0]
0.0017240031348827456
>>> lm = ps.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

classmethod by_col (*df*, *x*, *y=None*, *w=None*, *inplace=False*, *pvalue='sim'*, *outvals=None*, ***stat_kws*)
Function to compute a Moran_Local_BV statistic on a dataframe

Parameters

- **df** (*pandas.DataFrame*) – a pandas dataframe with a geometry column
- **X** (*list of strings*) – column name or list of column names to use as X values to compute the bivariate statistic. If no Y is provided, pairwise comparisons among these variates are used instead.
- **Y** (*list of strings*) – column name or list of column names to use as Y values to compute the bivariate statistic. if no Y is provided, pariwise comparisons among the X variates are used instead.
- **w** (*pysal weights object*) – a weights object aligned with the dataframe. If not provided, this is searched for in the dataframe's metadata

- **inplace** (*bool*) – a boolean denoting whether to operate on the dataframe inplace or to return a series contaning the results of the computation. If operating inplace, the derived columns will be named 'column_moran_local_bv'
- **pvalue** (*string*) – a string denoting which pvalue should be returned. Refer to the the Moran_Local_BV statistic's documentation for available p-values
- **outvals** (*list of strings*) – list of arbitrary attributes to return as columns from the Moran_Local_BV statistic
- ****stat_kws** (*keyword arguments*) – options to pass to the underlying statistic. For this, see the documentation for the Moran_Local_BV statistic.

Returns

- *If inplace, None, and operation is conducted on dataframe in memory. Otherwise,*
- *returns a copy of the dataframe with the relevant columns attached.*

See also:

`For`, `refer`

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

Adjusted Moran's I Global Autocorrelation Statistic for Rate Variables [Assuncao1999]

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
- **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.
- **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

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

seI_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 permutaitons

EI_sim
: *float* – (if permutations>0) average value of I from permutations

VI_sim
: *float* – (if permutations>0) variance of I from permutations

seI_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

Examples

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

classmethod **`by_col`**(*df*, *events*, *populations*, *w=None*, *inplace=False*, *pvalue='sim'*, *outvals=None*, *swapname='‘*, ***stat_kws*)

Function to compute a Moran_Rate statistic on a dataframe

Parameters

- **df** (*pandas.DataFrame*) – a pandas dataframe with a geometry column
- **events** (*string or list of strings*) – one or more names where events are stored
- **populations** (*string or list of strings*) – one or more names where the populations corresponding to the events are stored. If one population column is provided, it is used for all event columns. If more than one population column is provided but there is not a population for every event column, an exception will be raised.
- **w** (*pysal weights object*) – a weights object aligned with the dataframe. If not provided, this is searched for in the dataframe's metadata
- **inplace** (*bool*) – a boolean denoting whether to operate on the dataframe inplace or to return a series contaning the results of the computation. If operating inplace, the derived columns will be named 'column_moran_rate'
- **pvalue** (*string*) – a string denoting which pvalue should be returned. Refer to the the Moran_Rate statistic's documentation for available p-values
- **outvals** (*list of strings*) – list of arbitrary attributes to return as columns from the Moran_Rate statistic
- ****stat_kws** (*keyword arguments*) – options to pass to the underlying statistic. For this, see the documentation for the Moran_Rate statistic.

Returns

- *If inplace, None, and operation is conducted on dataframe in memory. Otherwise,*
- *returns a copy of the dataframe with the relevant columns attached.*

See also:

`For`, `refer`

class `pysal.esda.moran.`**`Moran_Local_Rate`**(*e*, *b*, *w*, *adjusted=True*, *transformation='r'*, *permutations=999*, *geoda_quads=False*)

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
- **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

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 quandrant 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 Iis. 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

seI_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 as ps
>>> import numpy as np
>>> np.random.seed(10)
>>> w = ps.open(ps.examples.get_path("sids2.gal")).read()
>>> f = ps.open(ps.examples.get_path("sids2.dbf"))
>>> e = np.array(f.by_col('SID79'))
>>> b = np.array(f.by_col('BIR79'))
>>> lm = ps.esda.moran.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.p_z_sim[0]
0.39319552026912641
>>> lm = ps.esda.moran.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 unittests that are conditional on architectures

classmethod by_col (*df*, *events*, *populations*, *w=None*, *inplace=False*, *pvalue='sim'*, *outvals=None*, *swapname=''*, ***stat_kws*)

Function to compute a Moran_Local_Rate statistic on a dataframe

Parameters

- **df** (*pandas.DataFrame*) – a pandas dataframe with a geometry column
- **events** (*string or list of strings*) – one or more names where events are stored
- **populations** (*string or list of strings*) – one or more names where the populations corresponding to the events are stored. If one population column is provided, it is used for all event columns. If more than one population column is provided but there is not a population for every event column, an exception will be raised.
- **w** (*pysal weights object*) – a weights object aligned with the dataframe. If not provided, this is searched for in the dataframe's metadata
- **inplace** (*bool*) – a boolean denoting whether to operate on the dataframe inplace or to return a series contaning the results of the computation. If operating inplace, the derived columns will be named 'column_moran_local_rate'
- **pvalue** (*string*) – a string denoting which pvalue should be returned. Refer to the the Moran_Local_Rate statistic's documentation for available p-values
- **outvals** (*list of strings*) – list of arbitrary attributes to return as columns from the Moran_Local_Rate statistic
- ****stat_kws** (*keyword arguments*) – options to pass to the underlying statistic. For this, see the documentation for the Moran_Local_Rate statistic.

Returns

- *If inplace, None, and operation is conducted on dataframe in memory. Otherwise,*
- *returns a copy of the dataframe with the relevant columns attached.*

See also:

`For`, `refer`

esda.smoothing — Smoothing of spatial rates

New in version 1.0.

class pysal.esda.smoothing.**Excess_Risk** (*e*, *b*)

Excess Risk

Parameters

- **e** (*array (n, 1)*) – event variable measured across n spatial units
- **b** (*array (n, 1)*) – population at risk variable measured across n spatial units

r

array (n, 1) – execess risk values

Examples

Reading data in stl_hom.csv into stl to extract values for event and population-at-risk variables

```
>>> stl = pysal.open(pysal.examples.get_path('stl_hom.csv'), 'r')
```

The 11th and 14th columns in stl_hom.csv includes the number of homocides and population. Creating two arrays from these columns.

```
>>> stl_e, stl_b = np.array(stl[:,10]), np.array(stl[:,13])
```

Creating an instance of Excess_Risk class using stl_e and stl_b

```
>>> er = Excess_Risk(stl_e, stl_b)
```

Extracting the excess risk values through the property r of the Excess_Risk instance, er

```
>>> er.r[:10]
array([ 0.20665681,  0.43613787,  0.42078261,  0.22066928,  0.57981596,
        0.35301709,  0.56407549,  0.17020994,  0.3052372 ,  0.25821905])
```

class `pysal.esda.smoothing.`**`Empirical_Bayes`**(*e*, *b*)

Aspatial Empirical Bayes Smoothing

Parameters

- **e** (*array (n, 1)*) – event variable measured across n spatial units
- **b** (*array (n, 1)*) – population at risk variable measured across n spatial units

r

array (n, 1) – rate values from Empirical Bayes Smoothing

Examples

Reading data in stl_hom.csv into stl to extract values for event and population-at-risk variables

```
>>> stl = pysal.open(pysal.examples.get_path('stl_hom.csv'), 'r')
```

The 11th and 14th columns in stl_hom.csv includes the number of homocides and population. Creating two arrays from these columns.

```
>>> stl_e, stl_b = np.array(stl[:,10]), np.array(stl[:,13])
```

Creating an instance of Empirical_Bayes class using stl_e and stl_b

```
>>> eb = Empirical_Bayes(stl_e, stl_b)
```

Extracting the risk values through the property r of the Empirical_Bayes instance, eb

```
>>> eb.r[:10]
array([  2.36718950e-05,   4.54539167e-05,   4.78114019e-05,
         2.76907146e-05,   6.58989323e-05,   3.66494122e-05,
         5.79952721e-05,   2.03064590e-05,   3.31152999e-05,
         3.02748380e-05])
```

class `pysal.esda.smoothing.`**`Spatial_Empirical_Bayes`**(*e*, *b*, *w*)

Spatial Empirical Bayes Smoothing

Parameters

- **e** (*array (n, 1)*) – event variable measured across n spatial units
- **b** (*array (n, 1)*) – population at risk variable measured across n spatial units
- **w** (*spatial weights instance*) –

r

array (n, 1) – rate values from Empirical Bayes Smoothing

Examples

Reading data in stl_hom.csv into stl to extract values for event and population-at-risk variables

```
>>> stl = pysal.open(pysal.examples.get_path('stl_hom.csv'), 'r')
```

The 11th and 14th columns in stl_hom.csv includes the number of homocides and population. Creating two arrays from these columns.

```
>>> stl_e, stl_b = np.array(stl[:,10]), np.array(stl[:,13])
```

Creating a spatial weights instance by reading in stl.gal file.

```
>>> stl_w = pysal.open(pysal.examples.get_path('stl.gal'), 'r').read()
```

Ensuring that the elements in the spatial weights instance are ordered by the given sequential numbers from 1 to the number of observations in stl_hom.csv

```
>>> if not stl_w.id_order_set: stl_w.id_order = range(1,len(stl) + 1)
```

Creating an instance of Spatial_Empirical_Bayes class using stl_e, stl_b, and stl_w

```
>>> s_eb = Spatial_Empirical_Bayes(stl_e, stl_b, stl_w)
```

Extracting the risk values through the property r of s_eb

```
>>> s_eb.r[:10]
array([  4.01485749e-05,   3.62437513e-05,   4.93034844e-05,
         5.09387329e-05,   3.72735210e-05,   3.69333797e-05,
         5.40245456e-05,   2.99806055e-05,   3.73034109e-05,
         3.47270722e-05])
```

class pysal.esda.smoothing.**Spatial_Rate**(*e*, *b*, *w*)

Spatial Rate Smoothing

Parameters

- **e** (*array (n, 1)*) – event variable measured across n spatial units
- **b** (*array (n, 1)*) – population at risk variable measured across n spatial units
- **w** (*spatial weights instance*) –

r

array (n, 1) – rate values from spatial rate smoothing

Examples

Reading data in stl_hom.csv into stl to extract values for event and population-at-risk variables

```
>>> stl = pysal.open(pysal.examples.get_path('stl_hom.csv'), 'r')
```

The 11th and 14th columns in stl_hom.csv includes the number of homocides and population. Creating two arrays from these columns.

```
>>> stl_e, stl_b = np.array(stl[:,10]), np.array(stl[:,13])
```

Creating a spatial weights instance by reading in stl.gal file.

```
>>> stl_w = pysal.open(pysal.examples.get_path('stl.gal'), 'r').read()
```

Ensuring that the elements in the spatial weights instance are ordered by the given sequential numbers from 1 to the number of observations in stl_hom.csv

```
>>> if not stl_w.id_order_set: stl_w.id_order = range(1,len(stl) + 1)
```

Creating an instance of Spatial_Rate class using stl_e, stl_b, and stl_w

```
>>> sr = Spatial_Rate(stl_e,stl_b,stl_w)
```

Extracting the risk values through the property r of sr

```
>>> sr.r[:10]
array([  4.59326407e-05,   3.62437513e-05,   4.98677081e-05,
         5.09387329e-05,   3.72735210e-05,   4.01073093e-05,
         3.79372794e-05,   3.27019246e-05,   4.26204928e-05,
         3.47270722e-05])
```

class `pysal.esda.smoothing.`**`Kernel_Smoother`**(*e*, *b*, *w*)

Kernal smoothing

Parameters

- **e** (*array (n, 1)*) – event variable measured across n spatial units
- **b** (*array (n, 1)*) – population at risk variable measured across n spatial units
- **w** (*Kernel weights instance*) –

r

array (n, 1) – rate values from spatial rate smoothing

Examples

Creating an array including event values for 6 regions

```
>>> e = np.array([10, 1, 3, 4, 2, 5])
```

Creating another array including population-at-risk values for the 6 regions

```
>>> b = np.array([100, 15, 20, 20, 80, 90])
```

Creating a list containing geographic coordinates of the 6 regions' centroids

```
>>> points=[(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
```

Creating a kernel-based spatial weights instance by using the above points

```
>>> kw=Kernel(points)
```

Ensuring that the elements in the kernel-based weights are ordered by the given sequential numbers from 0 to 5

```
>>> if not kw.id_order_set: kw.id_order = range(0,len(points))
```

Applying kernel smoothing to e and b

```
>>> kr = Kernel_Smoother(e, b, kw)
```

Extracting the smoothed rates through the property r of the Kernel_Smoother instance

```
>>> kr.r
array([ 0.10543301,  0.0858573 ,  0.08256196,  0.09884584,  0.04756872,
        0.04845298])
```

class `pysal.esda.smoothing.`**`Age_Adjusted_Smoother`**(*e*, *b*, *w*, *s*, *alpha=0.05*)

Age-adjusted rate smoothing

Parameters

- **e** (*array (n*h, 1)*) – event variable measured for each age group across n spatial units
- **b** (*array (n*h, 1)*) – population at risk variable measured for each age group across n spatial units
- **w** (*spatial weights instance*) –
- **s** (*array (n*h, 1)*) – standard population for each age group across n spatial units

r

array (n, 1) – rate values from spatial rate smoothing

Notes

Weights used to smooth age-specific events and populations are simple binary weights

Examples

Creating an array including 12 values for the 6 regions with 2 age groups

```
>>> e = np.array([10, 8, 1, 4, 3, 5, 4, 3, 2, 1, 5, 3])
```

Creating another array including 12 population-at-risk values for the 6 regions

```
>>> b = np.array([100, 90, 15, 30, 25, 20, 30, 20, 80, 80, 90, 60])
```

For age adjustment, we need another array of values containing standard population s includes standard population data for the 6 regions

```
>>> s = np.array([98, 88, 15, 29, 20, 23, 33, 25, 76, 80, 89, 66])
```

Creating a list containing geographic coordinates of the 6 regions' centroids

```
>>> points=[(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
```

Creating a kernel-based spatial weights instance by using the above points

```
>>> kw=Kernel(points)
```

Ensuring that the elements in the kernel-based weights are ordered by the given sequential numbers from 0 to 5

```
>>> if not kw.id_order_set: kw.id_order = range(0,len(points))
```

Applying age-adjusted smoothing to e and b

```
>>> ar = Age_Adjusted_Smoother(e, b, kw, s)
```

Extracting the smoothed rates through the property r of the Age_Adjusted_Smoother instance

```
>>> ar.r
array([ 0.10519625,  0.08494318,  0.06440072,  0.06898604,  0.06952076,
        0.05020968])
```

classmethod by_col (*df*, *e*, *b*, *w=None*, *s=None*, ***kwargs*)
Compute smoothing by columns in a dataframe.

Parameters

- **df** (*pandas.DataFrame*) – a dataframe containing the data to be smoothed
- **e** (*string or list of strings*) – the name or names of columns containing event variables to be smoothed
- **b** (*string or list of strings*) – the name or names of columns containing the population variables to be smoothed
- **w** (*pysal.weights.W or list of pysal.weights.W*) – the spatial weights object or objects to use with the event-population pairs. If not provided and a weights object is in the dataframe's metadata, that weights object will be used.
- **s** (*string or list of strings*) – the name or names of columns to use as a standard population variable for the events *e* and at-risk populations *b*.
- **inplace** (*bool*) – a flag denoting whether to output a copy of *df* with the relevant smoothed columns appended, or to append the columns directly to *df* itself.
- ****kwargs** (*optional keyword arguments*) – optional keyword options that are passed directly to the smoother.

Returns

- a copy of *df* containing the columns. Or, if *inplace*, this returns
- None, but implicitly adds columns to *df*.

class pysal.esda.smoothing.**Disk_Smoother** (*e*, *b*, *w*)
Locally weighted averages or disk smoothing

Parameters

- **e** (*array (n, 1)*) – event variable measured across n spatial units
- **b** (*array (n, 1)*) – population at risk variable measured across n spatial units
- **w** (*spatial weights matrix*) –

r

array (n, 1) – rate values from disk smoothing

Examples

Reading data in stl_hom.csv into stl to extract values for event and population-at-risk variables

```
>>> stl = pysal.open(pysal.examples.get_path('stl_hom.csv'), 'r')
```

The 11th and 14th columns in stl_hom.csv includes the number of homocides and population. Creating two arrays from these columns.

```
>>> stl_e, stl_b = np.array(stl[:,10]), np.array(stl[:,13])
```

Creating a spatial weights instance by reading in stl.gal file.

```
>>> stl_w = pysal.open(pysal.examples.get_path('stl.gal'), 'r').read()
```

Ensuring that the elements in the spatial weights instance are ordered by the given sequential numbers from 1 to the number of observations in stl_hom.csv

```
>>> if not stl_w.id_order_set: stl_w.id_order = range(1,len(stl) + 1)
```

Applying disk smoothing to stl_e and stl_b

```
>>> sr = Disk_Smoother(stl_e,stl_b,stl_w)
```

Extracting the risk values through the property r of s_eb

```
>>> sr.r[:10]
array([  4.56502262e-05,   3.44027685e-05,   3.38280487e-05,
         4.78530468e-05,   3.12278573e-05,   2.22596997e-05,
         2.67074856e-05,   2.36924573e-05,   3.48801587e-05,
         3.09511832e-05])
```

class `pysal.esda.smoothing.`**`Spatial_Median_Rate`**(*e*, *b*, *w*, *aw=None*, *iteration=1*)

Spatial Median Rate Smoothing

Parameters

- **e** (*array (n, 1)*) – event variable measured across n spatial units
- **b** (*array (n, 1)*) – population at risk variable measured across n spatial units
- **w** (*spatial weights instance*) –
- **aw** (*array (n, 1)*) – auxiliary weight variable measured across n spatial units
- **iteration** (*integer*) – the number of interations

r

array (n, 1) – rate values from spatial median rate smoothing

w

spatial weights instance

aw

array (n, 1) – auxiliary weight variable measured across n spatial units

Examples

Reading data in stl_hom.csv into stl to extract values for event and population-at-risk variables

```
>>> stl = pysal.open(pysal.examples.get_path('stl_hom.csv'), 'r')
```

The 11th and 14th columns in stl_hom.csv includes the number of homocides and population. Creating two arrays from these columns.

```
>>> stl_e, stl_b = np.array(stl[:,10]), np.array(stl[:,13])
```

Creating a spatial weights instance by reading in stl.gal file.

```
>>> stl_w = pysal.open(pysal.examples.get_path('stl.gal'), 'r').read()
```

Ensuring that the elements in the spatial weights instance are ordered by the given sequential numbers from 1 to the number of observations in stl_hom.csv

```
>>> if not stl_w.id_order_set: stl_w.id_order = range(1,len(stl) + 1)
```

Computing spatial median rates without iteration

```
>>> smr0 = Spatial_Median_Rate(stl_e,stl_b,stl_w)
```

Extracting the computed rates through the property r of the Spatial_Median_Rate instance

```
>>> smr0.r[:10]
array([  3.96047383e-05,   3.55386859e-05,   3.28308921e-05,
         4.30731238e-05,   3.12453969e-05,   1.97300409e-05,
         3.10159267e-05,   2.19279204e-05,   2.93763432e-05,
         2.93763432e-05])
```

Recomputing spatial median rates with 5 iterations

```
>>> smr1 = Spatial_Median_Rate(stl_e,stl_b,stl_w,iteration=5)
```

Extracting the computed rates through the property r of the Spatial_Median_Rate instance

```
>>> smr1.r[:10]
array([  3.11293620e-05,   2.95956330e-05,   3.11293620e-05,
         3.10159267e-05,   2.98436066e-05,   2.76406686e-05,
         3.10159267e-05,   2.94788171e-05,   2.99460806e-05,
         2.96981070e-05])
```

Computing spatial median rates by using the base variable as auxilliary weights without iteration

```
>>> smr2 = Spatial_Median_Rate(stl_e,stl_b,stl_w,aw=stl_b)
```

Extracting the computed rates through the property r of the Spatial_Median_Rate instance

```
>>> smr2.r[:10]
array([  5.77412020e-05,   4.46449551e-05,   5.77412020e-05,
         5.77412020e-05,   4.46449551e-05,   3.61363528e-05,
         3.61363528e-05,   4.46449551e-05,   5.77412020e-05,
         4.03987355e-05])
```

Recomputing spatial median rates by using the base variable as auxilliary weights with 5 iterations

```
>>> smr3 = Spatial_Median_Rate(stl_e,stl_b,stl_w,aw=stl_b,iteration=5)
```

Extracting the computed rates through the property r of the Spatial_Median_Rate instance

```
>>> smr3.r[:10]
array([  3.61363528e-05,   4.46449551e-05,   3.61363528e-05,
         3.61363528e-05,   4.46449551e-05,   3.61363528e-05,
         3.61363528e-05,   4.46449551e-05,   3.61363528e-05,
         4.46449551e-05])
>>>
```

class `pysal.esda.smoothing.`**`Spatial_Filtering`**(*bbox*, *data*, *e*, *b*, *x_grid*, *y_grid*, *r=None*, *pop=None*)

Spatial Filtering

Parameters

- **`bbox`** (*`a list of two lists where each list is a pair of coordinates`*) – a bounding box for the entire n spatial units
- **`data`** (*`array (n, 2)`*) – x, y coordinates
- **`e`** (*`array (n, 1)`*) – event variable measured across n spatial units
- **`b`** (*`array (n, 1)`*) – population at risk variable measured across n spatial units
- **`x_grid`** (*`integer`*) – the number of cells on x axis
- **`y_grid`** (*`integer`*) – the number of cells on y axis
- **`r`** (*`float`*) – fixed radius of a moving window
- **`pop`** (*`integer`*) – population threshold to create adaptive moving windows

`grid`

*array (x_grid*y_grid, 2)* – x, y coordinates for grid points

`r`

*array (x_grid*y_grid, 1)* – rate values for grid points

Notes

No tool is provided to find an optimal value for r or pop.

Examples

Reading data in stl_hom.csv into stl to extract values for event and population-at-risk variables

```
>>> stl = pysal.open(pysal.examples.get_path('stl_hom.csv'), 'r')
```

Reading the stl data in the WKT format so that we can easily extract polygon centroids

```
>>> fromWKT = pysal.core.util.WKTParser()
>>> stl.cast('WKT',fromWKT)
```

Extracting polygon centroids through iteration

```
>>> d = np.array([i.centroid for i in stl[:,0]])
```

Specifying the bounding box for the stl_hom data. The bbox should includes two points for the left-bottom and the right-top corners

```
>>> bbox = [[-92.700676, 36.881809], [-87.916573, 40.3295669]]
```

The 11th and 14th columns in stl_hom.csv includes the number of homocides and population. Creating two arrays from these columns.

```
>>> stl_e, stl_b = np.array(stl[:,10]), np.array(stl[:,13])
```

Applying spatial filtering by using a 10*10 mesh grid and a moving window with 2 radius

```
>>> sf_0 = Spatial_Filtering(bbox,d,stl_e,stl_b,10,10,r=2)
```

Extracting the resulting rates through the property r of the Spatial_Filtering instance

```
>>> sf_0.r[:10]
array([  4.23561763e-05,   4.45290850e-05,   4.56456221e-05,
         4.49133384e-05,   4.39671835e-05,   4.44903042e-05,
         4.19845497e-05,   4.11936548e-05,   3.93463504e-05,
         4.04376345e-05])
```

Applying another spatial filtering by allowing the moving window to grow until 600000 people are found in the window

```
>>> sf = Spatial_Filtering(bbox,d,stl_e,stl_b,10,10,pop=600000)
```

Checking the size of the reulting array including the rates

```
>>> sf.r.shape
(100,)
```

Extracting the resulting rates through the property r of the Spatial_Filtering instance

```
>>> sf.r[:10]
array([  3.73728738e-05,   4.04456300e-05,   4.04456300e-05,
         3.81035327e-05,   4.54831940e-05,   4.54831940e-05,
         3.75658628e-05,   3.75658628e-05,   3.75658628e-05,
         3.75658628e-05])
```

classmethod **`by_col`**(*df, e, b, x_grid, y_grid, geom_col='geometry', **kwargs*)

Compute smoothing by columns in a dataframe. The bounding box and point information is computed from the geometry column.

Parameters

- **`df`** (*`pandas.DataFrame`*) – a dataframe containing the data to be smoothed
- **`e`** (*`string or list of strings`*) – the name or names of columns containing event variables to be smoothed
- **`b`** (*`string or list of strings`*) – the name or names of columns containing the population variables to be smoothed
- **`x_grid`** (*`integer`*) – number of grid cells to use along the x-axis
- **`y_grid`** (*`integer`*) – number of grid cells to use along the y-axis
- **`geom_col`** (*`string`*) – the name of the column in the dataframe containing the geometry information.

- ****kwargs** (*optional keyword arguments*) – optional keyword options that are passed directly to the smoother.

Returns

- *a new dataframe of dimension (x_grid*y_grid, 3), containing the*
- *coordinates of the grid cells and the rates associated with those grid*
- *cells.*

class pysal.esda.smoothing.**Headbanging_Triples**(*data*, *w*, *k=5*, *t=3*, *angle=135.0*, *edgecor=False*)

Generate a pseudo spatial weights instance that contains headbanging triples

Parameters

- **data** (*array (n, 2)*) – numpy array of x, y coordinates
- **w** (*spatial weights instance*) –
- **k** (*integer number of nearest neighbors*) –
- **t** (*integer*) – the number of triples
- **angle** (*integer between 0 and 180*) – the angle criterium for a set of triples
- **edgecorr** (*boolean*) – whether or not correction for edge points is made

triples

dictionary – key is observation record id, value is a list of lists of triple ids

extra

dictionary – key is observation record id, value is a list of the following: tuple of original triple observations distance between original triple observations distance between an original triple observation and its extrapolated point

Examples

importing k-nearest neighbor weights creator

```
>>> from pysal import knnW_from_array
```

Reading data in stl_hom.csv into stl_db to extract values for event and population-at-risk variables

```
>>> stl_db = pysal.open(pysal.examples.get_path('stl_hom.csv'),'r')
```

Reading the stl data in the WKT format so that we can easily extract polygon centroids

```
>>> fromWKT = pysal.core.util.WKTParser()
>>> stl_db.cast('WKT',fromWKT)
```

Extracting polygon centroids through iteration

```
>>> d = np.array([i.centroid for i in stl_db[:,0]])
```

Using the centroids, we create a 5-nearst neighbor weights

```
>>> w = knnW_from_array(d,k=5)
```

Ensuring that the elements in the spatial weights instance are ordered by the order of stl_db's IDs

```
>>> if not w.id_order_set: w.id_order = w.id_order
```

Finding headbaning triples by using 5 nearest neighbors

```
>>> ht = Headbanging_Triples(d,w,k=5)
```

Checking the members of triples

```
>>> for k, item in ht.triples.items()[:5]: print k, item
0 [(5, 6), (10, 6)]
1 [(4, 7), (4, 14), (9, 7)]
2 [(0, 8), (10, 3), (0, 6)]
3 [(4, 2), (2, 12), (8, 4)]
4 [(8, 1), (12, 1), (8, 9)]
```

Opening sids2.shp file

```
>>> sids = pysal.open(pysal.examples.get_path('sids2.shp'),'r')
```

Extracting the centroids of polygons in the sids data

```
>>> sids_d = np.array([i.centroid for i in sids])
```

Creating a 5-nearest neighbors weights from the sids centroids

```
>>> sids_w = knnW_from_array(sids_d,k=5)
```

Ensuring that the members in sids_w are ordered by the order of sids_d's ID

```
>>> if not sids_w.id_order_set: sids_w.id_order = sids_w.id_order
```

Finding headbaning triples by using 5 nearest neighbors

```
>>> s_ht = Headbanging_Triples(sids_d,sids_w,k=5)
```

Checking the members of the found triples

```
>>> for k, item in s_ht.triples.items()[:5]: print k, item
0 [(1, 18), (1, 21), (1, 33)]
1 [(2, 40), (2, 22), (22, 40)]
2 [(39, 22), (1, 9), (39, 17)]
3 [(16, 6), (19, 6), (20, 6)]
4 [(5, 15), (27, 15), (35, 15)]
```

Finding headbanging triples by using 5 nearest neighbors with edge correction

```
>>> s_ht2 = Headbanging_Triples(sids_d,sids_w,k=5,edgecor=True)
```

Checking the members of the found triples

```
>>> for k, item in s_ht2.triples.items()[:5]: print k, item
0 [(1, 18), (1, 21), (1, 33)]
1 [(2, 40), (2, 22), (22, 40)]
2 [(39, 22), (1, 9), (39, 17)]
3 [(16, 6), (19, 6), (20, 6)]
4 [(5, 15), (27, 15), (35, 15)]
```

Checking the extrapolated point that is introduced into the triples during edge correction

```
>>> extrapolated = s_ht2.extra[72]
```

Checking the observation IDs constituting the extrapolated triple

```
>>> extrapolated[0]
(89, 77)
```

Checking the distances between the extraploated point and the observation 89 and 77

```
>>> round(extrapolated[1],5), round(extrapolated[2],6)
(0.33753, 0.302707)
```

class `pysal.esda.smoothing.`**`Headbanging_Median_Rate`**(*e*, *b*, *t*, *aw=None*, *iteration=1*)
Headbaning Median Rate Smoothing

Parameters

- **e** (*array (n, 1)*) – event variable measured across n spatial units
- **b** (*array (n, 1)*) – population at risk variable measured across n spatial units
- **t** (*Headbanging_Triples instance*) –
- **aw** (*array (n, 1)*) – auxilliary weight variable measured across n spatial units
- **iteration** (*integer*) – the number of iterations

r

array (n, 1) – rate values from headbanging median smoothing

Examples

importing k-nearest neighbor weights creator

```
>>> from pysal import knnW_from_array
```

opening the sids2 shapefile

```
>>> sids = pysal.open(pysal.examples.get_path('sids2.shp'), 'r')
```

extracting the centroids of polygons in the sids2 data

```
>>> sids_d = np.array([i.centroid for i in sids])
```

creating a 5-nearest neighbors weights from the centroids

```
>>> sids_w = knnW_from_array(sids_d,k=5)
```

ensuring that the members in sids_w are ordered

```
>>> if not sids_w.id_order_set: sids_w.id_order = sids_w.id_order
```

finding headbanging triples by using 5 neighbors return outdf

```
>>> s_ht = Headbanging_Triples(sids_d,sids_w,k=5)
```

reading in the sids2 data table

```
>>> sids_db = pysal.open(pysal.examples.get_path('sids2.dbf'), 'r')
```

extracting the 10th and 9th columns in the sids2.dbf and using data values as event and population-at-risk variables

```
>>> s_e, s_b = np.array(sids_db[:,9]), np.array(sids_db[:,8])
```

computing headbanging median rates from s_e, s_b, and s_ht

```
>>> sids_hb_r = Headbanging_Median_Rate(s_e,s_b,s_ht)
```

extracting the computed rates through the property r of the Headbanging_Median_Rate instance

```
>>> sids_hb_r.r[:5]
array([ 0.00075586,  0.        ,  0.0008285 ,  0.0018315 ,  0.00498891])
```

recomputing headbanging median rates with 5 iterations

```
>>> sids_hb_r2 = Headbanging_Median_Rate(s_e,s_b,s_ht,iteration=5)
```

extracting the computed rates through the property r of the Headbanging_Median_Rate instance

```
>>> sids_hb_r2.r[:5]
array([ 0.0008285 ,  0.00084331,  0.00086896,  0.0018315 ,  0.00498891])
```

recomputing headbanging median rates by considring a set of auxilliary weights

```
>>> sids_hb_r3 = Headbanging_Median_Rate(s_e,s_b,s_ht,aw=s_b)
```

extracting the computed rates through the property r of the Headbanging_Median_Rate instance

```
>>> sids_hb_r3.r[:5]
array([ 0.00091659,  0.        ,  0.00156838,  0.0018315 ,  0.00498891])
```

classmethod `by_col`(*df*, *e*, *b*, *t=None*, *geom_col='geometry'*, *inplace=False*, ***kwargs*)

Compute smoothing by columns in a dataframe. The bounding box and point information is computed from the geometry column.

Parameters

- **`df`** (*`pandas.DataFrame`*) – a dataframe containing the data to be smoothed
- **`e`** (*`string or list of strings`*) – the name or names of columns containing event variables to be smoothed
- **`b`** (*`string or list of strings`*) – the name or names of columns containing the population variables to be smoothed
- **`t`** (*`Headbanging_Triples instance or list of Headbanging_Triples`*) – list of headbanging triples instances. If not provided, this is computed from the geometry column of the dataframe.
- **`geom_col`** (*`string`*) – the name of the column in the dataframe containing the geometry information.
- **`inplace`** (*`bool`*) – a flag denoting whether to output a copy of *df* with the relevant smoothed columns appended, or to append the columns directly to *df* itself.

- ****kwargs** (*optional keyword arguments*) – optional keyword options that are passed directly to the smoother.

Returns

- *a new dataframe containing the smoothed Headbanging Median Rates for the*
- *event/population pairs. If done inplace, there is no return value and*
- *df* is modified in place.

`pysal.esda.smoothing.`**`flatten`**(*l*, *unique=True*)

flatten a list of lists

Parameters

- **l** (*list*) – of lists
- **unique** (*boolean*) – whether or not only unique items are wanted (default=True)

Returns of single items

Return type list

Examples

Creating a sample list whose elements are lists of integers

```
>>> l = [[1, 2], [3, 4, ], [5, 6]]
```

Applying flatten function

```
>>> flatten(l)
[1, 2, 3, 4, 5, 6]
```

`pysal.esda.smoothing.`**`weighted_median`**(*d*, *w*)

A utility function to find a median of d based on w

Parameters

- **d** (*array*) – (n, 1), variable for which median will be found
- **w** (*array*) – (n, 1), variable on which d's median will be decided

Notes

d and w are arranged in the same order

Returns median of d

Return type float

Examples

Creating an array including five integers. We will get the median of these integers.

```
>>> d = np.array([5,4,3,1,2])
```

Creating another array including weight values for the above integers. The median of d will be decided with a consideration to these weight values.

```
>>> w = np.array([10, 22, 9, 2, 5])
```

Applying weighted_median function

```
>>> weighted_median(d, w)
4
```

`pysal.esda.smoothing.`**`sum_by_n`**`(`*d*, *w*, *n*`)`

A utility function to summarize a data array into n values after weighting the array with another weight array w

Parameters

- **d** (*array*) – (t, 1), numerical values
- **w** (*array*) – (t, 1), numerical values for weighting
- **n** (*integer*) – the number of groups t = c*n (c is a constant)

Returns (n, 1), an array with summarized values

Return type array

Examples

Creating an array including four integers. We will compute weighted means for every two elements.

```
>>> d = np.array([10, 9, 20, 30])
```

Here is another array with the weight values for d's elements.

```
>>> w = np.array([0.5, 0.1, 0.3, 0.8])
```

We specify the number of groups for which the weighted mean is computed.

```
>>> n = 2
```

Applying sum_by_n function

```
>>> sum_by_n(d, w, n)
array([  5.9,   30. ])
```

`pysal.esda.smoothing.`**`crude_age_standardization`**`(`*e*, *b*, *n*`)`

A utility function to compute rate through crude age standardization

Parameters

- **e** (*array*) – (n*h, 1), event variable measured for each age group across n spatial units
- **b** (*array*) – (n*h, 1), population at risk variable measured for each age group across n spatial units
- **n** (*integer*) – the number of spatial units

Notes

e and b are arranged in the same order

Returns (n, 1), age standardized rate

Return type array

Examples

Creating an array of an event variable (e.g., the number of cancer patients) for 2 regions in each of which 4 age groups are available. The first 4 values are event values for 4 age groups in the region 1, and the next 4 values are for 4 age groups in the region 2.

```
>>> e = np.array([30, 25, 25, 15, 33, 21, 30, 20])
```

Creating another array of a population-at-risk variable (e.g., total population) for the same two regions. The order for entering values is the same as the case of e.

```
>>> b = np.array([100, 100, 110, 90, 100, 90, 110, 90])
```

Specifying the number of regions.

```
>>> n = 2
```

Applying crude_age_standardization function to e and b

```
>>> crude_age_standardization(e, b, n)
array([ 0.2375    ,  0.26666667])
```

pysal.esda.smoothing.**direct_age_standardization**(*e*, *b*, *s*, *n*, *alpha=0.05*)

A utility function to compute rate through direct age standardization

Parameters

- **e** (*array*) – (n*h, 1), event variable measured for each age group across n spatial units
- **b** (*array*) – (n*h, 1), population at risk variable measured for each age group across n spatial units
- **s** (*array*) – (n*h, 1), standard population for each age group across n spatial units
- **n** (*integer*) – the number of spatial units
- **alpha** (*float*) – significance level for confidence interval

Notes

e, b, and s are arranged in the same order

Returns a list of n tuples; a tuple has a rate and its lower and upper limits age standardized rates and confidence intervals

Return type list

Examples

Creating an array of an event variable (e.g., the number of cancer patients) for 2 regions in each of which 4 age groups are available. The first 4 values are event values for 4 age groups in the region 1, and the next 4 values are for 4 age groups in the region 2.

```
>>> e = np.array([30, 25, 25, 15, 33, 21, 30, 20])
```

Creating another array of a population-at-risk variable (e.g., total population) for the same two regions. The order for entering values is the same as the case of e.

```
>>> b = np.array([1000, 1000, 1100, 900, 1000, 900, 1100, 900])
```

For direct age standardization, we also need the data for standard population. Standard population is a reference population-at-risk (e.g., population distribution for the U.S.) whose age distribution can be used as a benchmarking point for comparing age distributions across regions (e.g., population distribution for Arizona and California). Another array including standard population is created.

```
>>> s = np.array([1000, 900, 1000, 900, 1000, 900, 1000, 900])
```

Specifying the number of regions.

```
>>> n = 2
```

Applying direct_age_standardization function to e and b

```
>>> [i[0] for i in direct_age_standardization(e, b, s, n)]
[0.023744019138755977, 0.026650717703349279]
```

pysal.esda.smoothing.**indirect_age_standardization**(*e, b, s_e, s_b, n, alpha=0.05*)

A utility function to compute rate through indirect age standardization

Parameters

- **e** (*array*) – (n*h, 1), event variable measured for each age group across n spatial units
- **b** (*array*) – (n*h, 1), population at risk variable measured for each age group across n spatial units
- **s_e** (*array*) – (n*h, 1), event variable measured for each age group across n spatial units in a standard population
- **s_b** (*array*) – (n*h, 1), population variable measured for each age group across n spatial units in a standard population
- **n** (*integer*) – the number of spatial units
- **alpha** (*float*) – significance level for confidence interval

Notes

e, b, s_e, and s_b are arranged in the same order

Returns a list of n tuples; a tuple has a rate and its lower and upper limits age standardized rate

Return type list

Examples

Creating an array of an event variable (e.g., the number of cancer patients) for 2 regions in each of which 4 age groups are available. The first 4 values are event values for 4 age groups in the region 1, and the next 4 values are for 4 age groups in the region 2.

```
>>> e = np.array([30, 25, 25, 15, 33, 21, 30, 20])
```

Creating another array of a population-at-risk variable (e.g., total population) for the same two regions. The order for entering values is the same as the case of e.

```
>>> b = np.array([100, 100, 110, 90, 100, 90, 110, 90])
```

For indirect age standardization, we also need the data for standard population and event. Standard population is a reference population-at-risk (e.g., population distribution for the U.S.) whose age distribution can be used as a benchmarking point for comparing age distributions across regions (e.g., popoulation distribution for Arizona and California). When the same concept is applied to the event variable, we call it standard event (e.g., the number of cancer patients in the U.S.). Two additional arrays including standard population and event are created.

```
>>> s_e = np.array([100, 45, 120, 100, 50, 30, 200, 80])
>>> s_b = np.array([1000, 900, 1000, 900, 1000, 900, 1000, 900])
```

Specifying the number of regions.

```
>>> n = 2
```

Applying indirect_age_standardization function to e and b

```
>>> [i[0] for i in indirect_age_standardization(e, b, s_e, s_b, n)]
[0.23723821989528798, 0.2610803324099723]
```

`pysal.esda.smoothing.`**`standardized_mortality_ratio`**(*e*, *b*, *s_e*, *s_b*, *n*)

A utility function to compute standardized mortality ratio (SMR).

Parameters

- **e** (*array*) – (n*h, 1), event variable measured for each age group across n spatial units
- **b** (*array*) – (n*h, 1), population at risk variable measured for each age group across n spatial units
- **s_e** (*array*) – (n*h, 1), event variable measured for each age group across n spatial units in a standard population
- **s_b** (*array*) – (n*h, 1), population variable measured for each age group across n spatial units in a standard population
- **n** (*integer*) – the number of spatial units

Notes

e, b, s_e, and s_b are arranged in the same order

Returns (nx1)

Return type array

Examples

Creating an array of an event variable (e.g., the number of cancer patients) for 2 regions in each of which 4 age groups are available. The first 4 values are event values for 4 age groups in the region 1, and the next 4 values are for 4 age groups in the region 2.

```
>>> e = np.array([30, 25, 25, 15, 33, 21, 30, 20])
```

Creating another array of a population-at-risk variable (e.g., total population) for the same two regions. The order for entering values is the same as the case of e.

```
>>> b = np.array([100, 100, 110, 90, 100, 90, 110, 90])
```

To compute standardized mortality ratio (SMR), we need two additional arrays for standard population and event. Creating s_e and s_b for standard event and population, respectively.

```
>>> s_e = np.array([100, 45, 120, 100, 50, 30, 200, 80])
>>> s_b = np.array([1000, 900, 1000, 900, 1000, 900, 1000, 900])
```

Specifying the number of regions.

```
>>> n = 2
```

Applying indirect_age_standardization function to e and b

```
>>> standardized_mortality_ratio(e, b, s_e, s_b, n)
array([ 2.48691099,  2.73684211])
```

`pysal.esda.smoothing.`**`choynowski`**`(`*e*, *b*, *n*, *threshold=None*`)`

Choynowski map probabilities [Choynowski1959] .

Parameters

- **e** (*array (n*h, 1)*) – event variable measured for each age group across n spatial units
- **b** (*array (n*h, 1)*) – population at risk variable measured for each age group across n spatial units
- **n** (*integer*) – the number of spatial units
- **threshold** (*float*) – Returns zero for any p-value greater than threshold

Notes

e and b are arranged in the same order

Returns

Return type array (nx1)

Examples

Creating an array of an event variable (e.g., the number of cancer patients) for 2 regions in each of which 4 age groups are available. The first 4 values are event values for 4 age groups in the region 1, and the next 4 values are for 4 age groups in the region 2.

```
>>> e = np.array([30, 25, 25, 15, 33, 21, 30, 20])
```

Creating another array of a population-at-risk variable (e.g., total population) for the same two regions. The order for entering values is the same as the case of e.

```
>>> b = np.array([100, 100, 110, 90, 100, 90, 110, 90])
```

Specifying the number of regions.

```
>>> n = 2
```

Applying indirect_age_standardization function to e and b

```
>>> print choynowski(e, b, n)
[ 0.30437751  0.29367033]
```

pysal.esda.smoothing.**assuncao_rate**(*e*, *b*)

The standardized rates where the mean and stadard deviation used for the standardization are those of Empirical Bayes rate estimates The standardized rates resulting from this function are used to compute Moran's I corrected for rate variables [Choynowski1959] .

Parameters

- **e** (*array (n, 1)*) – event variable measured at n spatial units
- **b** (*array (n, 1)*) – population at risk variable measured at n spatial units

Notes

e and b are arranged in the same order

Returns

Return type array (nx1)

Examples

Creating an array of an event variable (e.g., the number of cancer patients) for 8 regions.

```
>>> e = np.array([30, 25, 25, 15, 33, 21, 30, 20])
```

Creating another array of a population-at-risk variable (e.g., total population) for the same 8 regions. The order for entering values is the same as the case of e.

```
>>> b = np.array([100, 100, 110, 90, 100, 90, 110, 90])
```

Computing the rates

```
>>> print assuncao_rate(e, b)[:4]
[ 1.04319254 -0.04117865 -0.56539054 -1.73762547]
```

`pysal.inequality` — Spatial Inequality Analysis

`inequality.gini` – Gini inequality and decomposition measures

The `inequality.gini` module provides Gini inequality based measures

New in version 1.6. Gini based Inequality Metrics

class `pysal.inequality.gini.`**`Gini`**(*x*)

Classic Gini coefficient in absolute deviation form

Parameters **`y`** (*`array (n,1)`*) – attribute

`g`

float – Gini coefficient

class `pysal.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. [Rey2013]

Parameters

- **`y`** (*`array (n,1)`*) – attribute
- **`w`** (*`binary spatial weights object`*) –
- **`permutations`** (*`int (default = 99)`*) – number of permutations for inference

`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 Permuations > 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

`z_wcg`

float – z-value non-neighbor inequality component (level) from permutations

`p_z_sim`

float – pseudo p-value based on standard normal approximation of permutation based values

Examples

```
>>> import pysal
>>> import numpy as np
```

Use data from the 32 Mexican States, Decade frequency 1940-2010

```
>>> f=pysal.open(pysal.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

```
>>> regimes=np.array(f.by_col('hanson98'))
>>> w = pysal.block_weights(regimes)
>>> np.random.seed(12345)
>>> gs = pysal.inequality.gini.Gini_Spatial(y[:,0],w)
>>> gs.p_sim
0.040000000000000001
>>> 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.

`inequality.theil` – Theil inequality and decomposition measures

The `inequality.theil` module provides Theil inequality based measures

New in version 1.0. Theil Inequality metrics

class `pysal.inequality.theil.`**`Theil`**(*y*)

Classic Theil measure of inequality

$$T = \sum_{i=1}^n \left(\frac{y_i}{\sum_{i=1}^n y_i} \ln \left[N \frac{y_i}{\sum_{i=1}^n y_i} \right] \right)$$

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.

T

array (t,) or (1,) – Theil's T for each column of 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

```
>>> import pysal
>>> f=pysal.open(pysal.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]))
>>> theil_y=Theil(y)
>>> theil_y.T
array([ 0.20894344,  0.15222451,  0.10472941,  0.10194725,  0.09560113,
        0.10511256,  0.10660832])
```

class `pysal.inequality.theil.`**`TheilD`**(*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.

`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

Examples

```
>>> import pysal
>>> f=pysal.open(pysal.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]))
>>> regimes=np.array(f.by_col('hanson98'))
>>> theil_d=TheilD(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 ])
```

class `pysal.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 permutations. [Rey2004b]

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.

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

Examples

```
>>> import pysal
>>> f=pysal.open(pysal.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]))
>>> 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])
```

`pysal.region` — Spatially Constrained Clustering

`region.maxp` – maxp regionalization

New in version 1.0. Max p regionalization

Heuristically form the maximum number (p) of regions given a set of n areas and a floor constraint.

class `pysal.region.maxp.`**`Maxp`**(*w*, *z*, *floor*, *floor_variable*, *verbose=False*, *initial=100*, *seeds=[]*)
Try to find the maximum number of regions for a set of areas such that each region combines contiguous areas that satisfy a given threshold constraint.

Parameters

- **w** (`W`) – spatial weights object
- **z** (*array*) – n*m array of observations on m attributes across n areas. This is used to calculate intra-regional homogeneity
- **floor** (*int*) – a minimum bound for a variable that has to be obtained in each region
- **floor_variable** (*array*) – n*1 vector of observations on variable for the floor
- **initial** (*int*) – number of initial solutions to generate
- **verbose** (*binary*) – if true debugging information is printed
- **seeds** (*list*) – ids of observations to form initial seeds. If len(ids) is less than the number of observations, the complementary ids are added to the end of seeds. Thus the specified seeds get priority in the solution

area2region
dict – mapping of areas to region. key is area id, value is region id

regions
list – list of lists of regions (each list has the ids of areas in that region)

p
int – number of regions

swap_iterations
int – number of swap iterations

total_moves
int – number of moves into internal regions

Examples

Setup imports and set seeds for random number generators to insure the results are identical for each run.

```
>>> import numpy as np
>>> import pysal
>>> np.random.seed(100)
```

Setup a spatial weights matrix describing the connectivity of a square community with 100 areas. Generate two random data attributes for each area in the community (a 100x2 array) called z. p is the data vector used to compute the floor for a region, and floor is the floor value; in this case p is simply a vector of ones and the floor is set to three. This means that each region will contain at least three areas. In other cases the floor may be computed based on a minimum population count for example.

```
>>> import numpy as np
>>> import pysal
>>> np.random.seed(100)
>>> w = pysal.lat2W(10,10)
>>> z = np.random.random_sample((w.n,2))
>>> p = np.ones((w.n,1), float)
>>> floor = 3
>>> solution = pysal.region.Maxp(w, z, floor, floor_variable=p, initial=100)
>>> solution.p
29
>>> min([len(region) for region in solution.regions])
3
>>> solution.regions[0]
[76, 66, 56]
>>>
```

cinference(*nperm=99*, *maxiter=1000*)
Compare the within sum of squares for the solution against conditional simulated solutions where areas are randomly assigned to regions that maintain the cardinality of the original solution and respect contiguity relationships.

Parameters

- **nperm** (*int*) – number of random permutations for calculation of pseudo-p_values
- **maxiter** (*int*) – maximum number of attempts to find each permutation

pvalue
float – pseudo p_value

feas_sols
int – number of feasible solutions found

Notes

it is possible for the number of feasible solutions (feas_sols) to be less than the number of permutations requested (nperm); an exception is raised if this occurs.

Examples

Setup is the same as shown above except using a 5x5 community.

```
>>> import numpy as np
>>> import pysal
>>> np.random.seed(100)
>>> w=pysal.weights.lat2W(5,5)
>>> z=np.random.random_sample((w.n,2))
>>> p=np.ones((w.n,1),float)
>>> floor=3
>>> solution=pysal.region.Maxp(w,z,floor,floor_variable=p,initial=100)
```

Set nperm to 9 meaning that 9 random regions are computed and used for the computation of a pseudo-p-value for the actual Max-p solution. In empirical work this would typically be set much higher, e.g. 999 or 9999.

```
>>> solution.cinference(nperm=9, maxiter=100)
>>> solution.cpvalue
0.1
```

inference(*nperm=99*)

Compare the within sum of squares for the solution against simulated solutions where areas are randomly assigned to regions that maintain the cardinality of the original solution.

Parameters **nperm** (*int*) – number of random permutations for calculation of pseudo-p_values

pvalue

float – pseudo p_value

Examples

Setup is the same as shown above except using a 5x5 community.

```
>>> import numpy as np
>>> import pysal
>>> np.random.seed(100)
>>> w=pysal.weights.lat2W(5,5)
>>> z=np.random.random_sample((w.n,2))
>>> p=np.ones((w.n,1),float)
>>> floor=3
>>> solution=pysal.region.Maxp(w,z,floor,floor_variable=p,initial=100)
```

Set nperm to 9 meaning that 9 random regions are computed and used for the computation of a pseudo-p-value for the actual Max-p solution. In empirical work this would typically be set much higher, e.g. 999 or 9999.

```
>>> solution.inference(nperm=9)
>>> solution.pvalue
0.1
```

class `pysal.region.maxp.`**`Maxp_LISA`**(*w*, *z*, *y*, *floor*, *floor_variable*, *initial=100*)

Max-p regionalization using LISA seeds

Parameters

- **w** (`W`) – spatial weights object
- **z** (*array*) – nxk array of n observations on k variables used to measure similarity between areas within the regions.
- **y** (*array*) – nx1 array used to calculate the LISA statistics and to set the intial seed order
- **floor** (*float*) – value that each region must obtain on floor_variable
- **floor_variable** (*array*) – nx1 array of values for regional floor threshold
- **initial** (*int*) – number of initial feasible solutions to generate prior to swapping

area2region

dict – mapping of areas to region. key is area id, value is region id

regions

list – list of lists of regions (each list has the ids of areas in that region)

swap_iterations

int – number of swap iterations

total_moves

int – number of moves into internal regions

Notes

We sort the observations based on the value of the LISAs. This ordering then gives the priority for seeds forming the p regions. The initial priority seeds are not guaranteed to be separated in the final solution.

Examples

Setup imports and set seeds for random number generators to insure the results are identical for each run.

```
>>> import numpy as np
>>> import pysal
>>> np.random.seed(100)
```

Setup a spatial weights matrix describing the connectivity of a square community with 100 areas. Generate two random data attributes for each area in the community (a 100x2 array) called z. p is the data vector used to compute the floor for a region, and floor is the floor value; in this case p is simply a vector of ones and the floor is set to three. This means that each region will contain at least three areas. In other cases the floor may be computed based on a minimum population count for example.

```
>>> w=pysal.lat2W(10,10)
>>> z=np.random.random_sample((w.n,2))
>>> p=np.ones(w.n)
>>> mpl=pysal.region.Maxp_LISA(w,z,p,floor=3,floor_variable=p)
>>> mpl.p
```

```
30
>>> mpl.regions[0]
[99, 98, 89]
```

`region.randomregion` – Random region creation

New in version 1.0. Generate random regions

Randomly form regions given various types of constraints on cardinality and composition.

class `pysal.region.randomregion.`**`Random_Regions`**(*area_ids*, *num_regions=None*, *cardinality=None*, *contiguity=None*, *maxiter=100*, *compact=False*, *max_swaps=1000000*, *permutations=99*)

Generate a list of Random_Region instances.

Parameters

- **`area_ids`** (*`list`*) – IDs indexing the areas to be grouped into regions (must be in the same order as spatial weights matrix if this is provided)
- **`num_regions`** (*`integer`*) – number of regions to generate (if None then this is chosen randomly from 2 to n where n is the number of areas)
- **`cardinality`** (*`list`*) – list containing the number of areas to assign to regions (if num_regions is also provided then len(cardinality) must equal num_regions; if cardinality=None then a list of length num_regions will be generated randomly)
- **`contiguity`** (`W`) – spatial weights object (if None then contiguity will be ignored)
- **`maxiter`** (*`int`*) – maximum number attempts (for each permutation) at finding a feasible solution (only affects contiguity constrained regions)
- **`compact`** (*`boolean`*) – attempt to build compact regions, note (only affects contiguity constrained regions)
- **`max_swaps`** (*`int`*) – maximum number of swaps to find a feasible solution (only affects contiguity constrained regions)
- **`permutations`** (*`int`*) – number of Random_Region instances to generate

`solutions`

list – list of length permutations containing all Random_Region instances generated

`solutions_feas`

list – list of the Random_Region instances that resulted in feasible solutions

Examples

Setup the data

```
>>> import random
>>> import numpy as np
>>> import pysal
>>> nregs = 13
>>> cards = range(2,14) + [10]
>>> w = pysal.lat2W(10,10,rook=True)
>>> ids = w.id_order
```

Unconstrained

```
>>> random.seed(10)
>>> np.random.seed(10)
>>> t0 = pysal.region.Random_Regions(ids, permutations=2)
>>> t0.solutions[0].regions[0]
[19, 14, 43, 37, 66, 3, 79, 41, 38, 68, 2, 1, 60]
```

Cardinality and contiguity constrained (num_regions implied)

```
>>> random.seed(60)
>>> np.random.seed(60)
>>> t1 = pysal.region.Random_Regions(ids, num_regions=nregs, cardinality=cards,␣
↪contiguity=w, permutations=2)
>>> t1.solutions[0].regions[0]
[62, 61, 81, 71, 64, 90, 72, 51, 80, 63, 50, 73, 52]
```

Cardinality constrained (num_regions implied)

```
>>> random.seed(100)
>>> np.random.seed(100)
>>> t2 = pysal.region.Random_Regions(ids, num_regions=nregs, cardinality=cards,␣
↪permutations=2)
>>> t2.solutions[0].regions[0]
[37, 62]
```

Number of regions and contiguity constrained

```
>>> random.seed(100)
>>> np.random.seed(100)
>>> t3 = pysal.region.Random_Regions(ids, num_regions=nregs, contiguity=w,␣
↪permutations=2)
>>> t3.solutions[0].regions[1]
[62, 52, 51, 63, 61, 73, 41, 53, 60, 83, 42, 31, 32]
```

Cardinality and contiguity constrained

```
>>> random.seed(60)
>>> np.random.seed(60)
>>> t4 = pysal.region.Random_Regions(ids, cardinality=cards, contiguity=w,␣
↪permutations=2)
>>> t4.solutions[0].regions[0]
[62, 61, 81, 71, 64, 90, 72, 51, 80, 63, 50, 73, 52]
```

Number of regions constrained

```
>>> random.seed(100)
>>> np.random.seed(100)
>>> t5 = pysal.region.Random_Regions(ids, num_regions=nregs, permutations=2)
>>> t5.solutions[0].regions[0]
[37, 62, 26, 41, 35, 25, 36]
```

Cardinality constrained

```
>>> random.seed(100)
>>> np.random.seed(100)
>>> t6 = pysal.region.Random_Regions(ids, cardinality=cards, permutations=2)
>>> t6.solutions[0].regions[0]
[37, 62]
```

```
```

Contiguity constrained

```
>>> random.seed(100)
>>> np.random.seed(100)
>>> t7 = pysal.region.Random_Regions(ids, contiguity=w, permutations=2)
>>> t7.solutions[0].regions[1]
[62, 61, 71, 63]
```

class `pysal.region.randomregion.`**`Random_Region`**(*area_ids*, *num_regions=None*, *cardinality=None*, *contiguity=None*, *maxiter=1000*, *compact=False*, *max_swaps=1000000*)

Randomly combine a given set of areas into two or more regions based on various constraints.

Parameters

- **`area_ids`** (*`list`*) – IDs indexing the areas to be grouped into regions (must be in the same order as spatial weights matrix if this is provided)
- **`num_regions`** (*`integer`*) – number of regions to generate (if None then this is chosen randomly from 2 to n where n is the number of areas)
- **`cardinality`** (*`list`*) – list containing the number of areas to assign to regions (if num_regions is also provided then len(cardinality) must equal num_regions; if cardinality=None then a list of length num_regions will be generated randomly)
- **`contiguity`** (`W`) – spatial weights object (if None then contiguity will be ignored)
- **`maxiter`** (*`int`*) – maximum number attempts at finding a feasible solution (only affects contiguity constrained regions)
- **`compact`** (*`boolean`*) – attempt to build compact regions (only affects contiguity constrained regions)
- **`max_swaps`** (*`int`*) – maximum number of swaps to find a feasible solution (only affects contiguity constrained regions)

`feasible`
boolean – if True then solution was found

`regions`
list – list of lists of regions (each list has the ids of areas in that region)

Examples

Setup the data

```
>>> import random
>>> import numpy as np
>>> import pysal
>>> nregs = 13
>>> cards = range(2,14) + [10]
>>> w = pysal.weights.lat2W(10,10,rook=True)
>>> ids = w.id_order
```

Unconstrained

```
>>> random.seed(10)
>>> np.random.seed(10)
>>> t0 = pysal.region.Random_Region(ids)
>>> t0.regions[0]
[19, 14, 43, 37, 66, 3, 79, 41, 38, 68, 2, 1, 60]
```

Cardinality and contiguity constrained (num_regions implied)

```
>>> random.seed(60)
>>> np.random.seed(60)
>>> t1 = pysal.region.Random_Region(ids, num_regions=nregs, cardinality=cards,␣
↪contiguity=w)
>>> t1.regions[0]
[62, 61, 81, 71, 64, 90, 72, 51, 80, 63, 50, 73, 52]
```

Cardinality constrained (num_regions implied)

```
>>> random.seed(100)
>>> np.random.seed(100)
>>> t2 = pysal.region.Random_Region(ids, num_regions=nregs, cardinality=cards)
>>> t2.regions[0]
[37, 62]
```

Number of regions and contiguity constrained

```
>>> random.seed(100)
>>> np.random.seed(100)
>>> t3 = pysal.region.Random_Region(ids, num_regions=nregs, contiguity=w)
>>> t3.regions[1]
[62, 52, 51, 63, 61, 73, 41, 53, 60, 83, 42, 31, 32]
```

Cardinality and contiguity constrained

```
>>> random.seed(60)
>>> np.random.seed(60)
>>> t4 = pysal.region.Random_Region(ids, cardinality=cards, contiguity=w)
>>> t4.regions[0]
[62, 61, 81, 71, 64, 90, 72, 51, 80, 63, 50, 73, 52]
```

Number of regions constrained

```
>>> random.seed(100)
>>> np.random.seed(100)
>>> t5 = pysal.region.Random_Region(ids, num_regions=nregs)
>>> t5.regions[0]
[37, 62, 26, 41, 35, 25, 36]
```

Cardinality constrained

```
>>> random.seed(100)
>>> np.random.seed(100)
>>> t6 = pysal.region.Random_Region(ids, cardinality=cards)
>>> t6.regions[0]
[37, 62]
```

Contiguity constrained

```
>>> random.seed(100)
>>> np.random.seed(100)
>>> t7 = pysal.region.Random_Region(ids, contiguity=w)
>>> t7.regions[0]
[37, 36, 38, 39]
```

`pysal.spatial_dynamics` — Spatial Dynamics

`spatial_dynamics.directional` – Directional LISA Analytics

New in version 1.0. Directional Analysis of Dynamic LISAs

`pysal.spatial_dynamics.directional.`**`rose`**(*Y*, *w*, *k=8*, *permutations=0*)
Calculation of rose diagram for local indicators of spatial association.

Parameters

- **`Y`** (*`array`*) – (n, 2), variable observed on n spatial units over 2 time. periods
- **`w`** (`W`) – spatial weights object.
- **`k`** (*`int, optional`*) – number of circular sectors in rose diagram (the default is 8).
- **`permutations`** (*`int, optional`*) – number of random spatial permutations for calculation of pseudo p-values (the default is 0).

Returns

- **results** (*dictionary*) – (keys defined below)
- **counts** (*array*) – (k, 1), number of vectors with angular movement falling in each sector.
- **cuts** (*array*) – (k, 1), intervals defining circular sectors (in radians).
- **random_counts** (*array*) – (permutations, k), counts from random permutations.
- **pvalues** (*array*) – (k, 1), one sided (upper tail) pvalues for observed counts.

Notes

Based on [Rey2011] .

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

```
>>> f=open(pysal.examples.get_path("spi_download.csv"),'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([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

```
>>> sids=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/"']
>>> snames=[name for name in names if name not in out]
>>> sids=[names.index(name) for name in snames]
>>> states=data[sids,:]
>>> us=data[0]
>>> years=np.arange(1969,2009)
```

Now we convert state incomes to express them relative to the national average

```
>>> rel=states/(us*1.)
```

Create our contiguity matrix from an external GAL file and row standardize the resulting weights

```
>>> gal=pysal.open(pysal.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

```
>>> 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

```
>>> 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

```
>>> r4=rose(Y,w,k=4,permutations=999)
```

What are the cut-offs for our histogram - in radians

```
>>> r4['cuts']
array([ 0.        ,  1.57079633,  3.14159265,  4.71238898,  6.28318531])
```

How many vectors fell in each sector

```
>>> r4['counts']
array([32,  5,  9,  2])
```

What are the pseudo-pvalues for these counts based on 999 random spatial permutations of the state income data

```
>>> r4['pvalues']
array([ 0.02 ,  0.001,  0.001,  0.001])
```

Repeat the exercise but now for 8 rather than 4 sectors

```
>>> r8=rose(Y,w,permutations=999)
>>> r8['counts']
array([19, 13,  3,  2,  7,  2,  1,  1])
>>> r8['pvalues']
array([ 0.445,  0.042,  0.079,  0.003,  0.005,  0.1  ,  0.269,  0.002])
```

`spatial_dynamics.ergodic` – Summary measures for ergodic Markov chains

New in version 1.0. Summary measures for ergodic Markov chains

`pysal.spatial_dynamics.ergodic.`**`steady_state`**`(`*P*`)`

Calculates the steady state probability vector for a regular Markov transition matrix P.

Parameters **P** (*matrix*) – (k, k), an ergodic Markov transition probability matrix.

Returns (k, 1), steady state distribution.

Return type matrix

Examples

Taken from Kemeny and Snell. 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).

```
>>> import numpy as np
>>> p=np.matrix([[.5, .25, .25],[.5,0,.5],[.25,.25,.5]])
>>> steady_state(p)
matrix([[ 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.spatial_dynamics.ergodic.`**`fmpt`**`(`*P*`)`

Calculates the matrix of first mean passage times for an ergodic transition probability matrix.

Parameters **P** (*matrix*) – (k, k), an ergodic Markov transition probability matrix.

Returns **M** – (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.

Return type matrix

Examples

```
>>> import numpy as np
>>> p=np.matrix([[.5, .25, .25],[.5,0,.5],[.25,.25,.5]])
>>> fm=fmpt(p)
>>> fm
matrix([[ 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).

Notes

Uses formulation (and examples on p. 218) in [Kemeny1967].

pysal.spatial_dynamics.ergodic.**var_fmpt**(*P*)

Variances of first mean passage times for an ergodic transition probability matrix.

Parameters **P** (*matrix*) – (k, k), an ergodic Markov transition probability matrix.

Returns (k, k), elements are the variances for the number of intervals required for a chain starting in state i to first enter state j.

Return type matrix

Examples

```
>>> import numpy as np
>>> p=np.matrix([[.5, .25, .25],[.5,0,.5],[.25,.25,.5]])
>>> vfm=var_fmpt(p)
>>> vfm
matrix([[  5.58333333,  12.        ,   6.88888889],
        [  6.22222222,  12.        ,   6.22222222],
        [  6.88888889,  12.        ,   5.58333333]])
```

Notes

Uses formulation (and examples on p. 83) in [Kemeny1967].

spatial_dynamics.interaction – Space-time interaction tests

New in version 1.1. Methods for identifying space-time interaction in spatio-temporal event data.

class pysal.spatial_dynamics.interaction.**SpaceTimeEvents**(*path*, *time_col*, *infer_timestamp=False*)

Method for reformatting event data stored in a shapefile for use in calculating metrics of spatio-temporal interaction.

Parameters

- **path** (*string*) – the path to the appropriate shapefile, including the file name and extension

- **time** (*string*) – column header in the DBF file indicating the column containing the time stamp.
- **infer_timestamp** (*bool, optional*) – if the column containing the timestamp is formatted as calendar dates, try to coerce them into Python datetime objects (the default is False).

n
int – number of events.

x
array – (n, 1), array of the x coordinates for the events.

y
array – (n, 1), array of the y coordinates for the events.

t
array – (n, 1), array of the temporal coordinates for the events.

space
array – (n, 2), array of the spatial coordinates (x,y) for the events.

time
array – (n, 2), array of the temporal coordinates (t,1) for the events, the second column is a vector of ones.

Examples

Read in the example shapefile data, ensuring to omit the file extension. In order to successfully create the event data the .dbf file associated with the shapefile should have a column of values that are a timestamp for the events. This timestamp may be a numerical value or a date. Date inference was added in version 1.6.

```
>>> path = pysal.examples.get_path("burkitt.shp")
```

Create an instance of SpaceTimeEvents from a shapefile, where the temporal information is stored in a column named "T".

```
>>> events = SpaceTimeEvents(path,'T')
```

See how many events are in the instance.

```
>>> events.n
188
```

Check the spatial coordinates of the first event.

```
>>> events.space[0]
array([ 300.,  302.])
```

Check the time of the first event.

```
>>> events.t[0]
array([ 413.])
```

Calculate the time difference between the first two events.

```
>>> events.t[1] - events.t[0]
array([ 59.])
```

New, in 1.6, date support:

Now, create an instance of SpaceTimeEvents from a shapefile, where the temporal information is stored in a column named "DATE".

```
>>> events = SpaceTimeEvents(path,'DATE')
```

See how many events are in the instance.

```
>>> events.n
188
```

Check the spatial coordinates of the first event.

```
>>> events.space[0]
array([ 300.,  302.])
```

Check the time of the first event. Note that this value is equivalent to 413 days after January 1, 1900.

```
>>> events.t[0][0]
datetime.date(1901, 2, 16)
```

Calculate the time difference between the first two events.

```
>>> (events.t[1][0] - events.t[0][0]).days
59
```

`pysal.spatial_dynamics.interaction.`**`knox`**(*s_coords*, *t_coords*, *delta*, *tau*, *permutations=99*, *debug=False*)

Knox test for spatio-temporal interaction. [Knox1964]

Parameters

- **s_coords** (*array*) – (n, 2), spatial coordinates.
- **t_coords** (*array*) – (n, 1), temporal coordinates.
- **delta** (*float*) – threshold for proximity in space.
- **tau** (*float*) – threshold for proximity in time.
- **permutations** (*int, optional*) – the number of permutations used to establish pseudo- significance (the default is 99).
- **debug** (*bool, optional*) – if true, debugging information is printed (the default is False).

Returns

- **knox_result** (*dictionary*) – contains the statistic (stat) for the test and the associated p-value (pvalue).
- **stat** (*float*) – value of the knox test for the dataset.
- **pvalue** (*float*) – pseudo p-value associated with the statistic.
- **counts** (*int*) – count of space time neighbors.

Examples

```
>>> import numpy as np
>>> import pysal
```

Read in the example data and create an instance of SpaceTimeEvents.

```
>>> path = pysal.examples.get_path("burkitt.shp")
>>> events = SpaceTimeEvents(path,'T')
```

Set the random seed generator. This is used by the permutation based inference to replicate the pseudo-significance of our example results - the end-user will normally omit this step.

```
>>> np.random.seed(100)
```

Run the Knox test with distance and time thresholds of 20 and 5, respectively. This counts the events that are closer than 20 units in space, and 5 units in time.

```
>>> result = knox(events.space, events.t, delta=20, tau=5, permutations=99)
```

Next, we examine the results. First, we call the statistic from the results dictionary. This reports that there are 13 events close in both space and time, according to our threshold definitions.

```
>>> result['stat'] == 13
True
```

Next, we look at the pseudo-significance of this value, calculated by permuting the timestamps and rerunning the statistics. In this case, the results indicate there is likely no space-time interaction between the events.

```
>>> print("%2.2f"%result['pvalue'])
0.17
```

`pysal.spatial_dynamics.interaction.`**`mantel`**(*s_coords*, *t_coords*, *permutations=99*, *scon=1.0*, *spow=-1.0*, *tcon=1.0*, *tpow=-1.0*)

Standardized Mantel test for spatio-temporal interaction. [Mantel1967]

Parameters

- **s_coords** (*array*) – (n, 2), spatial coordinates.
- **t_coords** (*array*) – (n, 1), temporal coordinates.
- **permutations** (*int, optional*) – the number of permutations used to establish pseudo- significance (the default is 99).
- **scon** (*float, optional*) – constant added to spatial distances (the default is 1.0).
- **spow** (*float, optional*) – value for power transformation for spatial distances (the default is -1.0).
- **tcon** (*float, optional*) – constant added to temporal distances (the default is 1.0).
- **tpow** (*float, optional*) – value for power transformation for temporal distances (the default is -1.0).

Returns

- **mantel_result** (*dictionary*) – contains the statistic (stat) for the test and the associated p-value (pvalue).
- **stat** (*float*) – value of the knox test for the dataset.

- **pvalue** (*float*) – pseudo p-value associated with the statistic.

Examples

```
>>> import numpy as np
>>> import pysal
```

Read in the example data and create an instance of SpaceTimeEvents.

```
>>> path = pysal.examples.get_path("burkitt.shp")
>>> events = SpaceTimeEvents(path,'T')
```

Set the random seed generator. This is used by the permutation based inference to replicate the pseudo-significance of our example results - the end-user will normally omit this step.

```
>>> np.random.seed(100)
```

The standardized Mantel test is a measure of matrix correlation between the spatial and temporal distance matrices of the event dataset. The following example runs the standardized Mantel test without a constant or transformation; however, as recommended by Mantel (1967) [2]_, these should be added by the user. This can be done by adjusting the constant and power parameters.

```
>>> result = mantel(events.space, events.t, 99, scon=1.0, spow=-1.0, tcon=1.0,␣
↪tpow=-1.0)
```

Next, we examine the result of the test.

```
>>> print("%6.6f"%result['stat'])
0.048368
```

Finally, we look at the pseudo-significance of this value, calculated by permuting the timestamps and rerunning the statistic for each of the 99 permutations. According to these parameters, the results indicate space-time interaction between the events.

```
>>> print("%2.2f"%result['pvalue'])
0.01
```

`pysal.spatial_dynamics.interaction.`**`jacquez`**(*s_coords*, *t_coords*, *k*, *permutations=99*)

Jacquez k nearest neighbors test for spatio-temporal interaction. [Jacquez1996]

Parameters

- **`s_coords`** (`array`) – (n, 2), spatial coordinates.
- **`t_coords`** (`array`) – (n, 1), temporal coordinates.
- **`k`** (`int`) – the number of nearest neighbors to be searched.
- **`permutations`** (`int, optional`) – the number of permutations used to establish pseudo- significance (the default is 99).

Returns

- **jacquez_result** (*dictionary*) – contains the statistic (stat) for the test and the associated p-value (pvalue).
- **stat** (*float*) – value of the Jacquez k nearest neighbors test for the dataset.
- **pvalue** (*float*) – p-value associated with the statistic (normally distributed with k-1 df).

Examples

```
>>> import numpy as np
>>> import pysal
```

Read in the example data and create an instance of SpaceTimeEvents.

```
>>> path = pysal.examples.get_path("burkitt.shp")
>>> events = SpaceTimeEvents(path,'T')
```

The Jacquez test counts the number of events that are k nearest neighbors in both time and space. The following runs the Jacquez test on the example data and reports the resulting statistic. In this case, there are 13 instances where events are nearest neighbors in both space and time.

# turning off as kdtree changes from scipy < 0.12 return 13 #>>> np.random.seed(100) #>>> result = jacquez(events.space, events.t ,k=3,permutations=99) #>>> print result['stat'] #12

The significance of this can be assessed by calling the p- value from the results dictionary, as shown below. Again, no space-time interaction is observed.

#>>> result['pvalue'] < 0.01 #False

`pysal.spatial_dynamics.interaction.`**`modified_knox`**`(`*s_coords*, *t_coords*, *delta*, *tau*, *permutations=99*`)`

Baker's modified Knox test for spatio-temporal interaction. [Baker2004]

Parameters

- **`s_coords`** (*`array`*) – (n, 2), spatial coordinates.
- **`t_coords`** (*`array`*) – (n, 1), temporal coordinates.
- **`delta`** (*`float`*) – threshold for proximity in space.
- **`tau`** (*`float`*) – threshold for proximity in time.
- **`permutations`** (*`int, optional`*) – the number of permutations used to establish pseudo- significance (the default is 99).

Returns

- **modknox_result** (*dictionary*) – contains the statistic (stat) for the test and the associated p-value (pvalue).
- **stat** (*float*) – value of the modified knox test for the dataset.
- **pvalue** (*float*) – pseudo p-value associated with the statistic.

Examples

```
>>> import numpy as np
>>> import pysal
```

Read in the example data and create an instance of SpaceTimeEvents.

```
>>> path = pysal.examples.get_path("burkitt.shp")
>>> events = SpaceTimeEvents(path, 'T')
```

Set the random seed generator. This is used by the permutation based inference to replicate the pseudo-significance of our example results - the end-user will normally omit this step.

```
>>> np.random.seed(100)
```

Run the modified Knox test with distance and time thresholds of 20 and 5, respectively. This counts the events that are closer than 20 units in space, and 5 units in time.

```
>>> result = modified_knox(events.space, events.t, delta=20, tau=5,␣
↪permutations=99)
```

Next, we examine the results. First, we call the statistic from the results dictionary. This reports the difference between the observed and expected Knox statistic.

```
>>> print("%2.8f" % result['stat'])
2.81016043
```

Next, we look at the pseudo-significance of this value, calculated by permuting the timestamps and rerunning the statistics. In this case, the results indicate there is likely no space-time interaction.

```
>>> print("%2.2f" % result['pvalue'])
0.11
```

`spatial_dynamics.markov` – Markov based methods

New in version 1.0.

class `pysal.spatial_dynamics.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.

`p`
matrix – (k, k), transition probability matrix.

`steady_state`
matrix – (k, 1), ergodic distribution.

`transitions`
matrix – (k, k), count of transitions between each state i and j.

Examples

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

```
matrix([[ 0.30769231],
        [ 0.28846154],
        [ 0.40384615]])
```

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

```
>>> import pysal
>>> f = pysal.open(pysal.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

```
>>> q5 = np.array([pysal.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
matrix([[ 0.91011236,  0.0886392 ,  0.00124844,  0.        ,  0.        ],
        [ 0.09972299,  0.78531856,  0.11080332,  0.00415512,  0.        ],
        [ 0.        ,  0.10125   ,  0.78875   ,  0.1075    ,  0.0025    ],
        [ 0.        ,  0.00417827,  0.11977716,  0.79805014,  0.07799443],
        [ 0.        ,  0.        ,  0.00125156,  0.07133917,  0.92740926]])
>>> m.steady_state
matrix([[ 0.20774716],
        [ 0.18725774],
        [ 0.20740537],
        [ 0.18821787],
        [ 0.20937187]])
```

Relative incomes

```
>>> pci = pci.transpose()
>>> rpci = pci/(pci.mean(axis=0))
>>> rq = pysal.Quantiles(rpci.flatten()).yb
>>> rq.shape = (48,81)
>>> 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
matrix([[ 0.17957376],
        [ 0.21631443],
        [ 0.21499942],
        [ 0.21134662],
        [ 0.17776576]])
```

class `pysal.spatial_dynamics.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).

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:: Move Types

q1	q2	move_type
1	1	1
1	2	2
1	3	3
1	4	4
2	1	5
2	2	6
2	3	7
2	4	8
3	1	9
3	2	10
3	3	11
3	4	12
4	1	13
4	2	14
4	3	15
4	4	16

p
matrix – (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:: Significant Moves
```

(s1,s2)	move_type
(1,1)	[1, 16]
(1,0)	[17, 32]
(0,1)	[33, 48]
(0,0)	[49, 64]

q1	q2	s1	s2	move_type
1	1	1	1	1
1	2	1	1	2
1	3	1	1	3
1	4	1	1	4
2	1	1	1	5
2	2	1	1	6
2	3	1	1	7
2	4	1	1	8
3	1	1	1	9
3	2	1	1	10
3	3	1	1	11
3	4	1	1	12
4	1	1	1	13
4	2	1	1	14
4	3	1	1	15
4	4	1	1	16
1	1	1	0	17
1	2	1	0	18
.	.	.	.	.
.	.	.	.	.
4	3	1	0	31
4	4	1	0	32
1	1	0	1	33
1	2	0	1	34
.	.	.	.	.
.	.	.	.	.
4	3	0	1	47
4	4	0	1	48
1	1	0	0	49
1	2	0	0	50
.	.	.	.	.
.	.	.	.	.
4	3	0	0	63
4	4	0	0	64

steady_state [matrix] (k, 1), ergodic distribution.

transitions [matrix] (4, 4), count of transitions between each state i and j.

spillover [array] (n, 1) binary array, locations that were not part of a cluster in period 1 but joined a prexisting cluster in period 2.

Examples

```
>>> import pysal as ps
>>> import numpy as np
>>> f = ps.open(ps.examples.get_path("usjoin.csv"))
>>> years = range(1929, 2010)
>>> pci = np.array([f.by_col[str(y)] for y in years]).transpose()
>>> w = ps.open(ps.examples.get_path("states48.gal")).read()
>>> lm = ps.LISA_Markov(pci,w)
>>> lm.classes
array([1, 2, 3, 4])
>>> lm.steady_state
matrix([[ 0.28561505],
        [ 0.14190226],
        [ 0.40493672],
        [ 0.16754598]])
>>> lm.transitions
array([[  1.08700000e+03,   4.40000000e+01,   4.00000000e+00,
          3.40000000e+01],
       [  4.10000000e+01,   4.70000000e+02,   3.60000000e+01,
          1.00000000e+00],
       [  5.00000000e+00,   3.40000000e+01,   1.42200000e+03,
          3.90000000e+01],
       [  3.00000000e+01,   1.00000000e+00,   4.00000000e+01,
          5.52000000e+02]])
>>> lm.p
matrix([[ 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

```
>>> np.random.seed(10)
>>> lm_random = pysal.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

```
>>> "Chi2: %8.3f, p: %5.2f, dof: %d" % lm.chi_2
'Chi2: 1058.208, p:  0.00, dof: 9'
```

Actual transitions of LISAs

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

Expected transitions of LISAs under the null y and wy are moving independently of one another

```
>>> lm.expected_t
array([[  1.12328098e+03,   1.15377356e+01,   3.47522158e-01,
          3.38337644e+01],
       [  3.50272664e+00,   5.28473882e+02,   1.59178880e+01,
          1.05503814e-01],
       [  1.53878082e-01,   2.32163556e+01,   1.46690710e+03,
          9.72266513e+00],
       [  9.60775143e+00,   9.86856346e-02,   6.23537392e+00,
          6.07058189e+02]])
```

If the LISA classes are to be defined according to GeoDa, the *geoda_quad* option has to be set to true

```
>>> lm.q[0:5,0]
array([3, 2, 3, 1, 4])
>>> lm = ps.LISA_Markov(pci,w, geoda_quads=True)
>>> lm.q[0:5,0]
array([2, 3, 2, 1, 4])
```

spillover(*quadrant=1*, *neighbors_on=False*)
Detect spillover locations for diffusion in LISA Markov.

Parameters

- **quadrant** (*int*) – which quadrant in the scatterplot should form the core of a cluster.
- **neighbors_on** (*binary*) – If false, then only the 1st order neighbors of a core location are included in the cluster. If true, neighbors of cluster core 1st order neighbors are included in the cluster.

Returns **results** – two keys - values pairs: ‘components’ - array (n, t) values are integer ids (starting at 1) indicating which component/cluster observation i in period t belonged to. ‘spillover’ - array (n, t-1) binary values indicating if the location was a spill-over location that became a new member of a previously existing cluster.

Return type dictionary

Examples

```
>>> f = pysal.open(pysal.examples.get_path("usjoin.csv"))
>>> years = range(1929, 2010)
>>> pci = np.array([f.by_col[str(y)] for y in years]).transpose()
>>> w = pysal.open(pysal.examples.get_path("states48.gal")).read()
>>> np.random.seed(10)
>>> lm_random = pysal.LISA_Markov(pci, w, permutations=99)
>>> r = lm_random.spillover()
>>> r['components'][:,12]
array([ 0.,  0.,  0.,  2.,  0.,  1.,  1.,  0.,  0.,  2.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  2.,
        1.,  1.,  0.,  1.,  0.,  0.,  1.,  0.,  2.,  1.,  1.,  0.,  0.,
        0.,  0.,  0.,  1.,  0.,  2.,  1.,  0.,  0.])
>>> r['components'][:,14]
array([ 0.,  2.,  0.,  2.,  0.,  1.,  1.,  0.,  0.,  2.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  2.,
        0.,  1.,  0.,  1.,  0.,  0.,  1.,  0.,  2.,  1.,  1.,  0.,  0.,
        0.,  0.,  0.,  1.,  0.,  2.,  1.,  0.,  0.])
>>> r['spill_over'][:,12]
array([ 0.,  1.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,
        0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  1.])
```

Including neighbors of core neighbors

```
>>> rn = lm_random.spillover(neighbors_on=True)
>>> rn['components'][:,12]
array([ 0.,  2.,  0.,  2.,  0.,  1.,  1.,  0.,  0.,  2.,  0.,  1.,  0.,
        0.,  1.,  0.,  1.,  1.,  1.,  1.,  0.,  0.,  0.,  2.,  0.,  2.,
        1.,  1.,  0.,  1.,  0.,  0.,  1.,  0.,  2.,  1.,  1.,  0.,  0.,
        0.,  0.,  2.,  1.,  1.,  2.,  1.,  0.,  2.])
>>> rn["components"][:,13]
array([ 0.,  2.,  0.,  2.,  2.,  1.,  1.,  0.,  0.,  2.,  0.,  1.,  0.,
        2.,  1.,  0.,  1.,  1.,  1.,  1.,  0.,  0.,  0.,  2.,  2.,  2.,
        1.,  1.,  2.,  1.,  0.,  2.,  1.,  2.,  2.,  1.,  1.,  0.,  2.,
        0.,  2.,  2.,  1.,  1.,  2.,  1.,  0.,  2.])
>>> rn["spill_over"][:,12]
array([ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,
        0.,  0.,  1.,  0.,  0.,  1.,  0.,  1.,  0.,  0.,  0.,  0.,  1.,
        0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

class `pysal.spatial_dynamics.markov.`**`Spatial_Markov`**(*y*, *w*, *k=4*, *permutations=0*, *fixed=False*, *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*) – number of classes (quantiles).
- **permutations** (*int, optional*) – number of permutations for use in randomization based inference (the default is 0).
- **fixed** (*bool*) – If true, quantiles are taken over the entire n*t pooled series. If false, quantiles are taken each time period over n.
- **variable_name** (*string*) – name of variable.

p

matrix – (k, k), transition probability matrix for a-spatial Markov.

s

matrix – (k, 1), ergodic distribution for a-spatial Markov.

transitions
matrix – (k, k), counts of transitions between each state i and j for a-spatial Markov.

T
matrix – (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-1th.

P
matrix – (k, k, k), transition probability matrix for spatial Markov first dimension is the conditioned on the lag.

S
matrix – (k, k), steady state distributions for spatial Markov. Each row is a conditional steady_state.

F
matrix – (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 – pvalue 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 Bickenbach and Bode (2003) [Bickenbach2003].

Q_p_value
float – p-value for Q.

LR
float – Likelihood ratio statistic for homogeneity across lag classes based on Bickenback and Bode (2003) [Bickenbach2003].

LR_p_value
float – p-value for LR.

dof_hom
int – degrees of freedom for LR and Q, corrected for 0 cells.

Notes

Based on Rey (2001) [Rey2001].

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.

This is new

Examples

```
>>> import pysal as ps
>>> f = ps.open(ps.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 = ps.open(ps.examples.get_path("states48.gal")).read()
>>> w.transform = 'r'
>>> sm = ps.Spatial_Markov(rpci, w, fixed=True, k=5, variable_name='rpci')
>>> for p in sm.P:
...     print(p)
...
[[ 0.96341463  0.0304878   0.00609756  0.          0.        ]
 [ 0.06040268  0.83221477  0.10738255  0.          0.        ]
 [ 0.          0.14        0.74        0.12        0.        ]
 [ 0.          0.03571429  0.32142857  0.57142857  0.07142857]
 [ 0.          0.          0.          0.16666667  0.83333333]]
[[ 0.79831933  0.16806723  0.03361345  0.          0.        ]
 [ 0.0754717   0.88207547  0.04245283  0.          0.        ]
 [ 0.00537634  0.06989247  0.8655914   0.05913978  0.        ]
 [ 0.          0.          0.06372549  0.90196078  0.03431373]
 [ 0.          0.          0.          0.19444444  0.80555556]]
[[ 0.84693878  0.15306122  0.          0.          0.        ]
 [ 0.08133971  0.78947368  0.1291866   0.          0.        ]
 [ 0.00518135  0.0984456   0.79274611  0.0984456   0.00518135]
 [ 0.          0.          0.09411765  0.87058824  0.03529412]
 [ 0.          0.          0.          0.10204082  0.89795918]]
[[ 0.8852459   0.09836066  0.          0.01639344  0.        ]
 [ 0.03875969  0.81395349  0.13953488  0.          0.00775194]
 [ 0.0049505   0.09405941  0.77722772  0.11881188  0.0049505 ]
 [ 0.          0.02339181  0.12865497  0.75438596  0.09356725]
 [ 0.          0.          0.          0.09661836  0.90338164]]
[[ 0.33333333  0.66666667  0.          0.          0.        ]
 [ 0.0483871   0.77419355  0.16129032  0.01612903  0.        ]
 [ 0.01149425  0.16091954  0.74712644  0.08045977  0.        ]
 [ 0.          0.01036269  0.06217617  0.89637306  0.03108808]
 [ 0.          0.          0.          0.02352941  0.97647059]]
```

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 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,  0.2635101 ,  0.29013417,  0.1137326 ],
       [ 0.0776413 ,  0.19748806,  0.25352636,  0.22480415,  0.24654013],
       [ 0.01776781,  0.19964349,  0.19009833,  0.25524697,  0.3372434 ]])
```

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

`pysal.spatial_dynamics.markov.`**`kullback`**`(`*F*`)`

Kullback information based test of Markov Homogeneity.

Parameters **F** (*array*) – (s, r, r), values are transitions (not probabilities) for s strata, r initial states, r terminal states.

Returns

Results – (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).

Return type dictionary

Notes

Based on Kullback, Kupperman and Ku (1962) [Kullback1962]. Example below is taken from Table 9.2 .

Examples

```
>>> 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]
...     ])
>>>
>>> F = np.array([s1, s2])
>>> res = kullback(F)
>>> "%8.3f"%res['Conditional homogeneity']
' 160.961'
>>> "%d"%res['Conditional homogeneity dof']
'30'
>>> "%3.1f"%res['Conditional homogeneity pvalue']
'0.0'
```

`pysal.spatial_dynamics.markov.`**`prais`**(*pmat*)

Prais conditional mobility measure.

Parameters **pmat** (*matrix*) – (k, k), Markov probability transition matrix.

Returns **pr** – (1, k), conditional mobility measures for each of the k classes.

Return type matrix

Notes

Prais' conditional mobility measure for a class is defined as:

$$pr_i = 1 - p_{i,i}$$

Examples

```
>>> import numpy as np
>>> import pysal
>>> f = pysal.open(pysal.examples.get_path("usjoin.csv"))
>>> pci = np.array([f.by_col[str(y)] for y in range(1929,2010)])
>>> q5 = np.array([pysal.Quantiles(y).yb for y in pci]).transpose()
>>> m = pysal.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
matrix([[ 0.91011236,  0.0886392 ,  0.00124844,  0.        ,  0.        ],
        [ 0.09972299,  0.78531856,  0.11080332,  0.00415512,  0.        ],
        [ 0.        ,  0.10125   ,  0.78875   ,  0.1075    ,  0.0025    ],
        [ 0.        ,  0.00417827,  0.11977716,  0.79805014,  0.07799443],
        [ 0.        ,  0.        ,  0.00125156,  0.07133917,  0.92740926]])
>>> pysal.spatial_dynamics.markov.prais(m.p)
matrix([[ 0.08988764,  0.21468144,  0.21125   ,  0.20194986,  0.07259074]])
```

pysal.spatial_dynamics.markov.**shorrock**(*pmat*)

Shorrock's mobility measure.

Parameters **pmat** (*matrix*) – (k, k), Markov probability transition matrix.

Returns **sh** – Shorrock mobility measure.

Return type float

Notes

Shorock's mobility measure is defined as

$$sh = (k - \sum_{j=1}^{k} p_{j,j})/(k - 1)$$

Examples

```
>>> import numpy as np
>>> import pysal
>>> f = pysal.open(pysal.examples.get_path("usjoin.csv"))
>>> pci = np.array([f.by_col[str(y)] for y in range(1929,2010)])
>>> q5 = np.array([pysal.Quantiles(y).yb for y in pci]).transpose()
>>> m = pysal.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
matrix([[ 0.91011236,  0.0886392 ,  0.00124844,  0.        ,  0.        ],
        [ 0.09972299,  0.78531856,  0.11080332,  0.00415512,  0.        ],
        [ 0.        ,  0.10125   ,  0.78875   ,  0.1075    ,  0.0025    ],
        [ 0.        ,  0.00417827,  0.11977716,  0.79805014,  0.07799443],
        [ 0.        ,  0.        ,  0.00125156,  0.07133917,  0.92740926]])
>>> pysal.spatial_dynamics.markov.shorrock(m.p)
0.19758992000997844
```

`pysal.spatial_dynamics.markov.`**`homogeneity`**(*transition_matrices*, *regime_names=[]*, *class_names=[]*, *title='Markov Homogeneity Test'*)

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 an instance of Homogeneity_Results.

Return type implicit

`spatial_dynamics.rank` – Rank and spatial rank mobility measures

New in version 1.0. Rank and spatial rank mobility measures.

class `pysal.spatial_dynamics.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.

`tau`

float – The classic Tau statistic.

tau_spatial
float – Value of Tau for pairs that are spatial neighbors.

taus
array – (permtuations, 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 – pseudo p-value for observed tau_spatial under the null of spatial randomness (if permutations>0).

Notes

Algorithm has two stages. The first calculates classic Tau using a list based implementation of the algorithm from Christensen (2005) [Christensen2005]. Second stage calculates concordance measures for neighboring pairs of locations using a modification of the algorithm from Press et al (2007) [Press2007]. See Rey (2014) [Rey2014] for details.

Examples

```
>>> import pysal
>>> import numpy as np
>>> f=pysal.open(pysal.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=pysal.weights.block_weights(regime)
>>> np.random.seed(12345)
>>> res=[pysal.SpatialTau(y[:,i],y[:,i+1],w,99) for i in range(6)]
>>> for r in res:
...     ev = r.taus.mean()
...     "%8.3f %8.3f %8.3f"%(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'
```

class `pysal.spatial_dynamics.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.

`tau`

float – The classic Tau statistic.

`tau_p`

float – asymptotic p-value.

Notes

Modification of algorithm suggested by Christensen (2005). [Christensen2005] PySAL 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

```
>>> from scipy.stats import kendalltau
>>> 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
>>> skt = kendalltau(x1,x2)
>>> skt
(-0.47140452079103173, 0.24821309157521476)
```

class `pysal.spatial_dynamics.rank.`**`Theta`**(*y*, *regime*, *permutations=999*)

Regime mobility measure. [Rey2004a]

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.

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.

Examples

```
>>> import pysal
>>> f=pysal.open(pysal.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)
>>> t.theta
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
```

pysal.spreg — Regression and Diagnostics

`spreg.ols` — Ordinary Least Squares

The `spreg.ols` module provides OLS regression estimation.

New in version 1.1. Ordinary Least Squares regression classes.

class `pysal.spreg.ols.`**`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

`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)

f_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)

xtx
float – X'X

xtxi
float – (X'X)^-1

Examples

```
>>> import numpy as np
>>> import pysal
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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.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.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.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)
-2.654
>>> print round(ols.t_stat[2][1],3)
0.011
>>> print round(ols.r2,3)
0.35
```

Or we can easily obtain a full summary of all the results nicely formatted and ready to be printed:

```
>>> 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

------------------------------------------------------------------------------------
↪--
            Variable     Coefficient       Std.Error     t-Statistic     ␣
↪Probability
------------------------------------------------------------------------------------
↪--
            CONSTANT      46.4281827      13.1917570       3.5194844       0.
↪0009867
               crime      -0.4848885       0.1826729      -2.6544086       0.
↪0108745
              income       0.6289840       0.5359104       1.1736736       0.
↪2465669
------------------------------------------------------------------------------------
↪--

REGRESSION DIAGNOSTICS
MULTICOLLINEARITY CONDITION NUMBER           12.538

TEST ON NORMALITY OF ERRORS
TEST                             DF        VALUE           PROB
Jarque-Bera                       2          39.706           0.0000

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
================================ END OF REPORT␣
↪=====================================
```

If the optional parameters w and spat_diag are passed to pysal.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.

```
>>> w = pysal.weights.rook_from_shapefile(pysal.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
```

`spreg.ols_regimes` — Ordinary Least Squares with Regimes

The `spreg.ols_regimes` module provides OLS with regimes regression estimation.

New in version 1.5. Ordinary Least Squares regression with regimes.

class `pysal.spreg.ols_regimes.`**`OLS_Regimes`**(*y*, *x*, *regimes*, *w=None*, *robust=None*, *gwk=None*, *sig2n_k=True*, *nonspat_diag=True*, *spat_diag=False*, *moran=False*, *white_test=False*, *vm=False*, *constant_regi='many'*, *cols2regi='all'*, *regime_err_sep=True*, *cores=False*, *name_y=None*, *name_x=None*, *name_regimes=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
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with '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.

- **`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
- **`constant_regi`** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **`cols2regi`** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **`regime_err_sep`** (*boolean*) – If True, a separate regression is run for each regime.
- **`cores`** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **`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
- **`name_regimes`** (*string*) – Name of regime variable for use in the output

`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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

robust
string – Adjustment for robust standard errors Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

ar2
float – Adjusted R squared Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

utu
float – Sum of squared residuals

sig2
float – Sigma squared used in computations Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

sig2ML
float – Sigma squared (maximum likelihood) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

f_stat
tuple – Statistic (float), p-value (float) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

logll
float – Log likelihood Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

aic
float – Akaike information criterion Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

schwarz
float – Schwarz information criterion Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
array – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

t_stat
list of tuples – t statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`mulColli`
float – Multicollinearity condition number Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`jarque_bera`
dictionary – 'jb': Jarque-Bera statistic (float); 'pvalue': p-value (float); 'df': degrees of freedom (int) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`breusch_pagan`
dictionary – 'bp': Breusch-Pagan statistic (float); 'pvalue': p-value (float); 'df': degrees of freedom (int) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`koenker_bassett`
dictionary – 'kb': Koenker-Bassett statistic (float); 'pvalue': p-value (float); 'df': degrees of freedom (int) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`white`
dictionary – 'wh': White statistic (float); 'pvalue': p-value (float); 'df': degrees of freedom (int) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`lm_error`

***tuple* – Lagrange multiplier test for spatial error model; tuple** contains the pair (statistic, p-value), where each is a float

Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`lm_lag`

***tuple* – Lagrange multiplier test for spatial lag model; tuple** contains the pair (statistic, p-value), where each is a float

Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`rlm_error`

***tuple* – Robust lagrange multiplier test for spatial error model;** tuple contains the pair (statistic, p-value), where each is a float

Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`rlm_lag`

***tuple* – Robust lagrange multiplier test for spatial lag model;** tuple contains the pair (statistic, p-value), where each is a float

Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`lm_sarma`

***tuple* – Lagrange multiplier test for spatial SARMA model; tuple** contains the pair (statistic, p-value), where each is a float

Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`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

`name_regimes`
: *string* – Name of regime variable for use in the output

`title`
: *string* – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`sig2n`
: *float* – Sigma squared (computed with n in the denominator)

`sig2n_k`
: *float* – Sigma squared (computed with n-k in the denominator)

`xtx`
: *float* – X'X Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`xtxi`
: *float* – (X'X)^-1 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`regimes`
: *list* – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

`constant_regi`
: *['one', 'many']* – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime

`cols2regi`
: *list, 'all'* – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

`regime_err_sep`
: *boolean* – If True, a separate regression is run for each regime.

`kr`
: *int* – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

`kf`
: *int* – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

`nr`
: *int* – Number of different regimes in the 'regimes' list

`multi`
: *dictionary* – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

Examples

```
>>> import numpy as np
>>> 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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = db.by_col(y_var)
>>> y = np.array(y).reshape(len(y), 1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2','...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

We can now run the regression and then have a summary of the output by typing: olsr.summary Alternatively, we can just check the betas and standard errors of the parameters:

```
>>> olsr = OLS_Regimes(y, x, regimes, nonspat_diag=False, name_y=y_var, name_x=[
↪'PS90','UE90'], name_regimes=r_var, name_ds='NAT')
>>> olsr.betas
array([[ 0.39642899],
       [ 0.65583299],
       [ 0.48703937],
       [ 5.59835   ],
       [ 1.16210453],
       [ 0.53163886]])
>>> np.sqrt(olsr.vm.diagonal())
array([ 0.24816345,  0.09662678,  0.03628629,  0.46894564,  0.21667395,
        0.05945651])
>>> olsr.cols2regi
'all'
```

spreg.probit — Probit

The spreg.probit module provides probit regression estimation.

New in version 1.4. Probit regression class and diagnostics.

class `pysal.spreg.probit.`**`Probit`**(*y*, *x*, *w=None*, *optim='newton'*, *scalem='phimean'*, *maxiter=100*, *vm=False*, *name_y=None*, *name_x=None*, *name_w=None*, *name_ds=None*, *spat_diag=False*)

Classic non-spatial Probit and spatial diagnostics. The class includes a printout that formats all the results and tests in a nice format.

The diagnostics for spatial dependence currently implemented are:

- Pinkse Error [Pinkse2004]
- Kelejian and Prucha Moran's I [Kelejian2001]
- Pinkse & Slade Error [Pinkse1998]

Parameters

- **`x`** (*`array`*) – nxk array of independent variables (assumed to be aligned with y)
- **`y`** (*`array`*) – nx1 array of dependent binary variable
- **`w`** (`W`) – PySAL weights instance aligned with y
- **`optim`** (*`string`*) – Optimization method. Default: 'newton' (Newton-Raphson). Alternatives: 'ncg' (Newton-CG), 'bfgs' (BFGS algorithm)
- **`scalem`** (*`string`*) – Method to calculate the scale of the marginal effects. Default: 'phimean' (Mean of individual marginal effects) Alternative: 'xmean' (Marginal effects at variables mean)
- **`maxiter`** (*`int`*) – Maximum number of iterations until optimizer stops
- **`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

`x`

array – Two dimensional array with n rows and one column for each independent (exogenous) variable, including the constant

`y`

array – nx1 array of dependent variable

`betas`

array – kx1 array with estimated coefficients

`predy`

array – nx1 array of predicted y values

`n`

int – Number of observations

`k`

int – Number of variables

`vm`

array – Variance-covariance matrix (kxk)

`z_stat`

list of tuples – z statistic; each tuple contains the pair (statistic, p-value), where each is a float

xmean
array – Mean of the independent variables (kx1)

predpc
float – Percent of y correctly predicted

logl
float – Log-Likelihhod of the estimation

scalem
string – Method to calculate the scale of the marginal effects.

scale
float – Scale of the marginal effects.

slopes
array – Marginal effects of the independent variables (k-1x1)

slopes_vm
array – Variance-covariance matrix of the slopes (k-1xk-1)

LR
tuple – Likelihood Ratio test of all coefficients = 0 (test statistics, p-value)

Pinkse_error
float – Lagrange Multiplier test against spatial error correlation. Implemented as presented in [Pinkse2004]

KP_error
float – Moran's I type test against spatial error correlation. Implemented as presented in [Kelejian2001]

PS_error
float – Lagrange Multiplier test against spatial error correlation. Implemented as presented in [Pinkse1998]

warning
boolean – if True Maximum number of iterations exceeded or gradient and/or function calls not changing.

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

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
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.

```
>>> dbf = pysal.open(pysal.examples.get_path('columbus.dbf'),'r')
```

Extract the CRIME column (crime) 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. Since we want to run a probit model and for this example we use the Columbus data, we also need to transform the continuous CRIME variable into a binary variable. As in [McMillen1992], we define y = 1 if CRIME > 40.

```
>>> y = np.array([dbf.by_col('CRIME')]).T
>>> y = (y>40).astype(float)
```

Extract HOVAL (home values) 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.

```
>>> names_to_extract = ['INC', 'HOVAL']
>>> x = np.array([dbf.by_col(name) for name in names_to_extract]).T
```

Since we want to the test the probit model for spatial dependence, 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.

```
>>> w = pysal.open(pysal.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. 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.

```
>>> model = Probit(y, x, w=w, name_y='crime', name_x=['income','home value'],␣
↪name_ds='columbus', name_w='columbus.gal')
```

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.

```
>>> np.around(model.betas, decimals=6)
array([[ 3.353811],
       [-0.199653],
       [-0.029514]])
```

```
>>> np.around(model.vm, decimals=6)
array([[ 0.852814, -0.043627, -0.008052],
       [-0.043627,  0.004114, -0.000193],
       [-0.008052, -0.000193,  0.00031 ]])
```

Since we have provided a spatial weigths matrix, the diagnostics for spatial dependence have also been computed. We can access them and their p-values individually:

```
>>> tests = np.array([['Pinkse_error','KP_error','PS_error']])
>>> stats = np.array([[model.Pinkse_error[0],model.KP_error[0],model.PS_
↪error[0]]])
>>> pvalue = np.array([[model.Pinkse_error[1],model.KP_error[1],model.PS_
↪error[1]]])
>>> print np.hstack((tests.T,np.around(np.hstack((stats.T,pvalue.T)),6)))
[['Pinkse_error' '3.131719' '0.076783']
 ['KP_error' '1.721312' '0.085194']
 ['PS_error' '2.558166' '0.109726']]
```

Or we can easily obtain a full summary of all the results nicely formatted and ready to be printed simply by typing 'print model.summary'

`spreg.twosls` — Two Stage Least Squares

The `spreg.twosls` module provides 2SLS regression estimation.

New in version 1.3.

class `pysal.spreg.twosls.`**`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

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 ouput

`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

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
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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.

```
>>> 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.

```
>>> 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).

```
>>> 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’.

```
>>> q = []
>>> q.append(db.by_col("DISCBD"))
>>> q = np.array(q).T
```

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) 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.

```
>>> 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]]
```

spreg.twosls_regimes — Two Stage Least Squares with Regimes

The `spreg.twosls_regimes` module provides 2SLS with regimes regression estimation.

New in version 1.5.

class `pysal.spreg.twosls_regimes.`**`TSLS_Regimes`** (*y*, *x*, *yend*, *q*, *regimes*, *w=None*, *robust=None*, *gwk=None*, *sig2n_k=True*, *spat_diag=False*, *vm=False*, *constant_regi='many'*, *cols2regi='all'*, *regime_err_sep=True*, *name_y=None*, *name_x=None*, *cores=False*, *name_yend=None*, *name_q=None*, *name_regimes=None*, *name_w=None*, *name_gwk=None*, *name_ds=None*, *summ=True*)

Two stage least squares (2SLS) with regimes

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)
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **constant_regi** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **regime_err_sep** (*boolean*) – If True, a separate regression is run for each regime.
- **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. If 'ogmm', then Optimal GMM is used to estimate betas and the variance-covariance matrix. 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.
- **vm** (*boolean*) – If True, include variance-covariance matrix in summary
- **cores** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **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_regimes** (*string*) – Name of regimes variable 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

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

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

yend
array – Two dimensional array with n rows and one column for each endogenous variable Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

q
array – Two dimensional array with n rows and one column for each external exogenous variable used as instruments Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

vm
array – Variance covariance matrix (kxk)

regimes
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
[False, 'one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_err_sep
boolean – If True, a separate regression is run for each regime.

kr
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

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_regimes
string – Name of regimes variable 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

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2','...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

In this case we consider RD90 (resource deprivation) as an endogenous regressor. We tell the model that this is so by passing it in a different parameter from the exogenous variables (x).

```
>>> yd_var = ['RD90']
>>> yd = np.array([db.by_col(name) for name in yd_var]).T
```

Because we have endogenous variables, to obtain a correct estimate of the model, we need to instrument for RD90. We use FP89 (families below poverty) for this and hence put it in the instruments parameter, 'q'.

```
>>> q_var = ['FP89']
>>> q = np.array([db.by_col(name) for name in q_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

Since we want to perform tests for spatial dependence, 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 `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> w.transform = 'r'
```

We can now run the regression and then have a summary of the output by typing: model.summary Alternatively, we can just check the betas and standard errors of the parameters:

```
>>> tslsr = TSLS_Regimes(y, x, yd, q, regimes, w=w, constant_regi='many', spat_
↪diag=False, name_y=y_var, name_x=x_var, name_yend=yd_var, name_q=q_var, name_
↪regimes=r_var, name_ds='NAT', name_w='NAT.shp')
```

```
>>> tslsr.betas
array([[ 3.66973562],
       [ 1.06950466],
       [ 0.14680946],
       [ 2.45864196],
       [ 9.55873243],
       [ 1.94666348],
       [-0.30810214],
       [ 3.68718119]])
```

```
>>> np.sqrt(tslsr.vm.diagonal())
array([ 0.38389901,  0.09963973,  0.04672091,  0.22725012,  0.49181223,
        0.19630774,  0.07784587,  0.25529011])
```

`spreg.twosls_sp` — Spatial Two Stage Least Squares

The `spreg.twosls_sp` module provides S2SLS regression estimation.

New in version 1.3. Spatial Two Stages Least Squares

class `pysal.spreg.twosls_sp.`**`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

- **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

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 ouput

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

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.

```
>>> import numpy as np
>>> import pysal
>>> import pysal.spreg.diagnostics as D
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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.

```
>>> 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.

```
>>> 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.rook_from_shapefile(pysal.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:

```
>>> 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:

```
>>> 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],
       [ -0.48072345],
       [  0.02836221]])
```

Once the model is run, we can obtain the standard error of the coefficient estimates by calling the diagnostics module:

```
>>> D.se_betas(reg)
array([ 17.91278862,   0.52486082,   0.1822815 ,   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'`:

```
>>> 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],
       [ -0.48072345],
       [  0.02836221]])
```

And we can access the standard errors from the model object:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> D.se_betas(reg)
array([ 53.0829123 ,   1.02511494,   0.57589064,   0.59891744])
```

spreg.twosls_sp_regimes — Spatial Two Stage Least Squares with Regimes

The `spreg.twosls_sp_regimes` module provides S2SLS with regimes regression estimation.

New in version 1.5. Spatial Two Stages Least Squares with Regimes

class `pysal.spreg.twosls_sp_regimes.`**`GM_Lag_Regimes`**(*y, x, regimes, yend=None, q=None, w=None, w_lags=1, lag_q=True, robust=None, gwk=None, sig2n_k=False, spat_diag=False, constant_regi='many', cols2regi='all', regime_lag_sep=False, regime_err_sep=True, cores=False, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_regimes=None, name_w=None, name_gwk=None, name_ds=None*)

Spatial two stage least squares (S2SLS) with regimes; 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
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **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
- **constant_regi** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **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).
- **regime_lag_sep** (*boolean*) – If True (default), the spatial parameter for spatial lag is also computed according to different regimes. If False, the spatial parameter is fixed accross regimes. Option valid only when regime_err_sep=True
- **regime_err_sep** (*boolean*) – If True, a separate regression is run for each regime.
- **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. If 'ogmm', then Optimal GMM is used to estimate betas and the variance-covariance matrix. 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
- **cores** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **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
- **name_regimes** (*string*) – Name of regimes variable for use in output

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

kstar
: *integer* – Number of endogenous variables. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

yend
: *array* – Two dimensional array with n rows and one column for each endogenous variable Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

q
: *array* – Two dimensional array with n rows and one column for each external exogenous variable used as instruments Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z
: *array* – nxk array of variables (combination of x and yend) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

h
: *array* – nxl array of instruments (combination of x and q) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

robust
: *string* – Adjustment for robust standard errors Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

pr2_e
: *float* – Pseudo R squared (squared correlation between y and ypred_e (using reduced form)) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

utu
: *float* – Sum of squared residuals

sig2
: *float* – Sigma squared used in computations Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
array – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z_stat
list of tuples – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

ak_test
tuple – Anselin-Kelejian test; tuple contains the pair (statistic, p-value) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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_gwk
string – Name of kernel weights matrix for use in output

name_ds
string – Name of dataset for use in output

name_regimes
string – Name of regimes variable for use in output

title
string – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

hthi
float – (H'H)^-1 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

varb
array – (Z'H (H'H)^-1 H'Z)^-1 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

zthhthi
array – Z'H(H'H)^-1 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

pfora1a2
array – n(zthhthi)'varb Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

regimes
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
['one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_lag_sep
boolean – If True, the spatial parameter for spatial lag is also computed according to different regimes. If False (default), the spatial parameter is fixed accross regimes.

regime_err_sep
boolean – If True, a separate regression is run for each regime.

kr
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2',...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

Since we want to run a spatial lag model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> 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.

```
>>> model=GM_Lag_Regimes(y, x, regimes, w=w, regime_lag_sep=False, regime_err_
↪sep=False, name_y=y_var, name_x=x_var, name_regimes=r_var, name_ds='NAT', name_
↪w='NAT.shp')
>>> model.betas
array([[ 1.28897623],
       [ 0.79777722],
       [ 0.56366891],
       [ 8.73327838],
       [ 1.30433406],
       [ 0.62418643],
       [-0.39993716]])
```

Once the model is run, we can have a summary of the output by typing: model.summary . Alternatively, we can obtain the standard error of the coefficient estimates by calling:

```
>>> model.std_err
array([ 0.44682888,  0.14358192,  0.05655124,  1.06044865,  0.20184548,
        0.06118262,  0.12387232])
```

In the example above, all coefficients but the spatial lag vary according to the regime. It is also possible to have the spatial lag varying according to the regime, which effective will result in an independent spatial lag model estimated for each regime. To run these models, the argument regime_lag_sep must be set to True:

```
>>> model=GM_Lag_Regimes(y, x, regimes, w=w, regime_lag_sep=True, name_y=y_var,␣
↪name_x=x_var, name_regimes=r_var, name_ds='NAT', name_w='NAT.shp')
>>> print np.hstack((np.array(model.name_z).reshape(8,1),model.betas,np.
↪sqrt(model.vm.diagonal().reshape(8,1))))
[['0_CONSTANT' '1.36584769' '0.39854720']
 ['0_PS90' '0.80875730' '0.11324884']
 ['0_UE90' '0.56946813' '0.04625087']
 ['0_W_HR90' '-0.4342438' '0.13350159']
 ['1_CONSTANT' '7.90731073' '1.63601874']
 ['1_PS90' '1.27465703' '0.24709870']
 ['1_UE90' '0.60167693' '0.07993322']
 ['1_W_HR90' '-0.2960338' '0.19934459']]
```

Alternatively, we can type: 'model.summary' to see the organized results output. 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 add the endogenous variable RD90 (resource deprivation) and we decide to instrument for it with FP89 (families below poverty):

```
>>> yd_var = ['RD90']
>>> yd = np.array([db.by_col(name) for name in yd_var]).T
>>> q_var = ['FP89']
>>> q = np.array([db.by_col(name) for name in q_var]).T
```

And we can run the model again:

```
>>> model = GM_Lag_Regimes(y, x, regimes, yend=yd, q=q, w=w, regime_lag_sep=False,
↪ regime_err_sep=False, name_y=y_var, name_x=x_var, name_yend=yd_var, name_q=q_
↪var, name_regimes=r_var, name_ds='NAT', name_w='NAT.shp')
>>> model.betas
array([[ 3.42195202],
       [ 1.03311878],
       [ 0.14308741],
       [ 8.99740066],
       [ 1.91877758],
       [-0.32084816],
       [ 2.38918212],
       [ 3.67243761],
       [ 0.06959139]])
```

Once the model is run, we can obtain the standard error of the coefficient estimates. Alternatively, we can have a summary of the output by typing: model.summary

```
>>> model.std_err
array([ 0.49163311,  0.12237382,  0.05633464,  0.72555909,  0.17250521,
        0.06749131,  0.27370369,  0.25106224,  0.05804213])
```

spreg.diagnostics- Diagnostics

The `spreg.diagnostics` module provides a set of standard non-spatial diagnostic tests.

New in version 1.1. Diagnostics for regression estimations.

pysal.spreg.diagnostics.**f_stat**(*reg*)

Calculates the f-statistic and associated p-value of the regression. [Greene2003] (For two stage least squares see f_stat_tsls)

Parameters **reg** (*regression object*) – output instance from a regression model

Returns **fs_result** – includes value of F statistic and associated p-value

Return type tuple

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the F-statistic for the regression.

```
>>> testresult = diagnostics.f_stat(reg)
```

Print the results tuple, including the statistic and its significance.

```
>>> print("%12.12f"%testresult[0],"%12.12f"%testresult[1])
('28.385629224695', '0.000000009341')
```

pysal.spreg.diagnostics.**t_stat**(*reg*, *z_stat=False*)

Calculates the t-statistics (or z-statistics) and associated p-values. [Greene2003]

Parameters

- **reg** (*regression object*) – output instance from a regression model
- **z_stat** (*boolean*) – If True run z-stat instead of t-stat

Returns **ts_result** – each tuple includes value of t statistic (or z statistic) and associated p-value

Return type list of tuples

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate t-statistics for the regression coefficients.

```
>>> testresult = diagnostics.t_stat(reg)
```

Print the tuples that contain the t-statistics and their significances.

```
>>> print("%12.12f"%testresult[0][0], "%12.12f"%testresult[0][1], "%12.12f"
↪%testresult[1][0], "%12.12f"%testresult[1][1], "%12.12f"%testresult[2][0], "%12.
↪12f"%testresult[2][1])
('14.490373143689', '0.000000000000', '-4.780496191297', '0.000018289595', '-2.
↪654408642718', '0.010874504910')
```

`pysal.spreg.diagnostics.`**`r2`**(*reg*)

Calculates the R^2 value for the regression. [Greene2003]

Parameters **`reg`** (*`regression object`*) – output instance from a regression model

Returns **r2_result** – value of the coefficient of determination for the regression

Return type float

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the R^2 value for the regression.

```
>>> testresult = diagnostics.r2(reg)
```

Print the result.

```
>>> print("%1.8f"%testresult)
0.55240404
```

`pysal.spreg.diagnostics.`**`ar2`**(*reg*)

Calculates the adjusted R^2 value for the regression. [Greene2003]

Parameters **`reg`** (*`regression object`*) – output instance from a regression model

Returns **ar2_result** – value of R^2 adjusted for the number of explanatory variables.

Return type float

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the adjusted R^2 value for the regression. >>> testresult = diagnostics.ar2(reg)

Print the result.

```
>>> print("%1.8f"%testresult)
0.53294335
```

`pysal.spreg.diagnostics.se_betas(reg)`

Calculates the standard error of the regression coefficients. [Greene2003]

Parameters **reg** (*regression object*) – output instance from a regression model

Returns **se_result** – includes standard errors of each coefficient (1 x k)

Return type array

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the standard errors of the regression coefficients.

```
>>> testresult = diagnostics.se_betas(reg)
```

Print the vector of standard errors.

```
>>> testresult
array([ 4.73548613,  0.33413076,  0.10319868])
```

pysal.spreg.diagnostics.**log_likelihood**(*reg*)
Calculates the log-likelihood value for the regression. [Greene2003]

Parameters **reg** (*regression object*) – output instance from a regression model

Returns **ll_result** – value for the log-likelihood of the regression.

Return type float

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the log-likelihood for the regression.

```
>>> testresult = diagnostics.log_likelihood(reg)
```

Print the result.

```
>>> testresult
-187.3772388121491
```

pysal.spreg.diagnostics.**akaike**(*reg*)
Calculates the Akaike Information Criterion. [Akaike1974]

Parameters **reg** (*regression object*) – output instance from a regression model

Returns **aic_result** – value for Akaike Information Criterion of the regression.

Return type scalar

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the Akaike Information Criterion (AIC).

```
>>> testresult = diagnostics.akaike(reg)
```

Print the result.

```
>>> testresult
380.7544776242982
```

`pysal.spreg.diagnostics.`**`schwarz`**`(`*reg*`)`

Calculates the Schwarz Information Criterion. [Schwarz1978]

Parameters **reg** (*regression object*) – output instance from a regression model

Returns **bic_result** – value for Schwarz (Bayesian) Information Criterion of the regression.

Return type scalar

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the Schwarz Information Criterion.

```
>>> testresult = diagnostics.schwarz(reg)
```

Print the results.

```
>>> testresult
386.42993851863008
```

`pysal.spreg.diagnostics.`**`condition_index`**(*reg*)

Calculates the multicollinearity condition index according to Belsey, Kuh and Welsh (1980) [Belsley1980].

Parameters **reg** (*regression object*) – output instance from a regression model

Returns **ci_result** – scalar value for the multicollinearity condition index.

Return type float

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the condition index to check for multicollinearity.

```
>>> testresult = diagnostics.condition_index(reg)
```

Print the result.

```
>>> print("%1.3f"%testresult)
6.542
```

`pysal.spreg.diagnostics.`**`jarque_bera`**(*reg*)

Jarque-Bera test for normality in the residuals. [Jarque1980]

Parameters **`reg`** (*`regression object`*) – output instance from a regression model

Returns

- **jb_result** (*dictionary*) – contains the statistic (jb) for the Jarque-Bera test and the associated p-value (p-value)
- **df** (*integer*) – degrees of freedom for the test (always 2)
- **jb** (*float*) – value of the test statistic
- **pvalue** (*float*) – p-value associated with the statistic (chi^2 distributed with 2 df)

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"), "r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the Jarque-Bera test for normality of residuals.

```
>>> testresult = diagnostics.jarque_bera(reg)
```

Print the degrees of freedom for the test.

```
>>> testresult['df']
2
```

Print the test statistic.

```
>>> print("%1.3f"%testresult['jb'])
1.836
```

Print the associated p-value.

```
>>> print("%1.4f"%testresult['pvalue'])
0.3994
```

`pysal.spreg.diagnostics.`**`breusch_pagan`**(*reg*, *z=None*)

Calculates the Breusch-Pagan test statistic to check for heteroscedasticity. [Breusch1979]

Parameters

- **reg** (*regression object*) – output instance from a regression model
- **z** (*array*) – optional input for specifying an alternative set of variables (Z) to explain the observed variance. By default this is a matrix of the squared explanatory variables (X**2) with a constant added to the first column if not already present. In the default case, the explanatory variables are squared to eliminate negative values.

Returns

- **bp_result** (*dictionary*) – contains the statistic (bp) for the test and the associated p-value (p-value)
- **bp** (*float*) – scalar value for the Breusch-Pagan test statistic
- **df** (*integer*) – degrees of freedom associated with the test (k)
- **pvalue** (*float*) – p-value associated with the statistic (chi^2 distributed with k df)

Notes

x attribute in the reg object must have a constant term included. This is standard for spreg.OLS so no testing done to confirm constant.

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"), "r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the Breusch-Pagan test for heteroscedasticity.

```
>>> testresult = diagnostics.breusch_pagan(reg)
```

Print the degrees of freedom for the test.

```
>>> testresult['df']
2
```

Print the test statistic.

```
>>> print("%1.3f"%testresult['bp'])
7.900
```

Print the associated p-value.

```
>>> print("%1.4f"%testresult['pvalue'])
0.0193
```

`pysal.spreg.diagnostics.`**`white`**(*reg*)

Calculates the White test to check for heteroscedasticity. [White1980]

Parameters **`reg`** (*`regression object`*) – output instance from a regression model

Returns

- **white_result** (*dictionary*) – contains the statistic (white), degrees of freedom (df) and the associated p-value (pvalue) for the White test.
- **white** (*float*) – scalar value for the White test statistic.
- **df** (*integer*) – degrees of freedom associated with the test
- **pvalue** (*float*) – p-value associated with the statistic (chi^2 distributed with k df)

Notes

x attribute in the reg object must have a constant term included. This is standard for spreg.OLS so no testing done to confirm constant.

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the White test for heteroscedasticity.

```
>>> testresult = diagnostics.white(reg)
```

Print the degrees of freedom for the test.

```
>>> print testresult['df']
5
```

Print the test statistic.

```
>>> print("%1.3f"%testresult['wh'])
19.946
```

Print the associated p-value.

```
>>> print("%1.4f"%testresult['pvalue'])
0.0013
```

`pysal.spreg.diagnostics.`**`koenker_bassett`**(*reg*, *z=None*)

Calculates the Koenker-Bassett test statistic to check for heteroscedasticity. [Koenker1982] [Greene2003]

Parameters

- **reg** (*regression output*) – output from an instance of a regression class
- **z** (*array*) – optional input for specifying an alternative set of variables (Z) to explain the observed variance. By default this is a matrix of the squared explanatory variables (X**2) with a constant added to the first column if not already present. In the default case, the explanatory variables are squared to eliminate negative values.

Returns

- **kb_result** (*dictionary*) – contains the statistic (kb), degrees of freedom (df) and the associated p-value (pvalue) for the test.
- **kb** (*float*) – scalar value for the Koenker-Bassett test statistic.
- **df** (*integer*) – degrees of freedom associated with the test
- **pvalue** (*float*) – p-value associated with the statistic (chi^2 distributed)

Notes

x attribute in the reg object must have a constant term included. This is standard for spreg.OLS so no testing done to confirm constant.

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the Koenker-Bassett test for heteroscedasticity.

```
>>> testresult = diagnostics.koenker_bassett(reg)
```

Print the degrees of freedom for the test.

```
>>> testresult['df']
2
```

Print the test statistic.

```
>>> print("%1.3f"%testresult['kb'])
5.694
```

Print the associated p-value.

```
>>> print("%1.4f"%testresult['pvalue'])
0.0580
```

pysal.spreg.diagnostics.**vif**(*reg*)

Calculates the variance inflation factor for each independent variable. For the ease of indexing the results, the constant is currently included. This should be omitted when reporting the results to the output text. [Greene2003]

Parameters **reg** (*regression object*) – output instance from a regression model

Returns **vif_result** – each tuple includes the vif and the tolerance, the order of the variables corresponds to their order in the reg.x matrix

Return type list of tuples

Examples

```
>>> import numpy as np
>>> import pysal
>>> import diagnostics
>>> from ols import OLS
```

Read the DBF associated with the Columbus data.

```
>>> db = pysal.open(pysal.examples.get_path("columbus.dbf"),"r")
```

Create the dependent variable vector.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Create the matrix of independent variables.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression.

```
>>> reg = OLS(y,X)
```

Calculate the variance inflation factor (VIF). >>> testresult = diagnostics.vif(reg)

Select the tuple for the income variable.

```
>>> incvif = testresult[1]
```

Print the VIF for income.

```
>>> print("%12.12f"%incvif[0])
1.333117497189
```

Print the tolerance for income.

```
>>> print("%12.12f"%incvif[1])
0.750121427487
```

Repeat for the home value variable.

```
>>> hovalvif = testresult[2]
>>> print("%12.12f"%hovalvif[0])
1.333117497189
>>> print("%12.12f"%hovalvif[1])
0.750121427487
```

`pysal.spreg.diagnostics.`**`likratiotest`**`(`*reg0*, *reg1*`)`

Likelihood ratio test statistic [Greene2003]

Parameters

- **reg0** (*regression object for constrained model (H0)*) –
- **reg1** (*regression object for unconstrained model (H1)*) –

Returns

- **likratio** (*dictionary*) – contains the statistic (likr), the degrees of freedom (df) and the p-value (pvalue)
- **likr** (*float*) – likelihood ratio statistic
- **df** (*integer*) – degrees of freedom
- **p-value** (*float*) – p-value

Examples

```
>>> import numpy as np
>>> import pysal as ps
>>> import scipy.stats as stats
>>> import pysal.spreg.ml_lag as lag
```

Use the baltim sample data set

```
>>> db =  ps.open(ps.examples.get_path("baltim.dbf"),'r')
>>> 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.transform = 'r'
```

OLS regression

```
>>> ols1 = ps.spreg.OLS(y,x)
```

ML Lag regression

```
>>> mllag1 = lag.ML_Lag(y,x,w)
```

```
>>> lr = likratiotest(ols1,mllag1)
>>> print "Likelihood Ratio Test: {0:.4f}       df: {1}        p-value: {2:.4f}".
↪format(lr["likr"],lr["df"],lr["p-value"])
Likelihood Ratio Test: 44.5721       df: 1        p-value: 0.0000
```

`spreg.diagnostics_sp` — Spatial Diagnostics

The `spreg.diagnostics_sp` module provides spatial diagnostic tests.

New in version 1.1. Spatial diagnostics module

class `pysal.spreg.diagnostics_sp.`**`LMtests`**(*ols, w, tests=['all']*)

Lagrange Multiplier tests. Implemented as presented in Anselin et al. (1996) [Anselin1996a]

...

`ols`

OLS – OLS regression object

`w`

W – Spatial weights instance

`tests`

list – Lists of strings with the tests desired to be performed. Values may be:

- 'all': runs all the options (default)
- 'lme': LM error test
- 'rlme': Robust LM error test
- 'lml' : LM lag test
- 'rlml': Robust LM lag test

Parameters

- **`lme`** (*`tuple`*) – (Only if 'lme' or 'all' was in tests). Pair of statistic and p-value for the LM error test.
- **`lml`** (*`tuple`*) – (Only if 'lml' or 'all' was in tests). Pair of statistic and p-value for the LM lag test.
- **`rlme`** (*`tuple`*) – (Only if 'rlme' or 'all' was in tests). Pair of statistic and p-value for the Robust LM error test.
- **`rlml`** (*`tuple`*) – (Only if 'rlml' or 'all' was in tests). Pair of statistic and p-value for the Robust LM lag test.
- **`sarma`** (*`tuple`*) – (Only if 'rlml' or 'all' was in tests). Pair of statistic and p-value for the SARMA test.

Examples

```
>>> import numpy as np
>>> import pysal
>>> from ols import OLS
```

Open the csv file to access the data for analysis

```
>>> csv = pysal.open(pysal.examples.get_path('columbus.dbf'),'r')
```

Pull out from the csv the files we need ('HOVAL' as dependent as well as 'INC' and 'CRIME' as independent) and directly transform them into nx1 and nx2 arrays, respectively

```
>>> y = np.array([csv.by_col('HOVAL')]).T
>>> x = np.array([csv.by_col('INC'), csv.by_col('CRIME')]).T
```

Create the weights object from existing .gal file

```
>>> w = pysal.open(pysal.examples.get_path('columbus.gal'), 'r').read()
```

Row-standardize the weight object (not required although desirable in some cases)

```
>>> w.transform='r'
```

Run an OLS regression

```
>>> ols = OLS(y, x)
```

Run all the LM tests in the residuals. These diagnostics test for the presence of remaining spatial autocorrelation in the residuals of an OLS model and give indication about the type of spatial model. There are five types: presence of a spatial lag model (simple and robust version), presence of a spatial error model (simple and robust version) and joint presence of both a spatial lag as well as a spatial error model.

```
>>> lms = pysal.spreg.diagnostics_sp.LMtests(ols, w)
```

LM error test:

```
>>> print round(lms.lme[0],4), round(lms.lme[1],4)
3.0971 0.0784
```

LM lag test:

```
>>> print round(lms.lml[0],4), round(lms.lml[1],4)
0.9816 0.3218
```

Robust LM error test:

```
>>> print round(lms.rlme[0],4), round(lms.rlme[1],4)
3.2092 0.0732
```

Robust LM lag test:

```
>>> print round(lms.rlml[0],4), round(lms.rlml[1],4)
1.0936 0.2957
```

LM SARMA test:

```
>>> print round(lms.sarma[0],4), round(lms.sarma[1],4)
4.1907 0.123
```

class `pysal.spreg.diagnostics_sp.`**`MoranRes`**(*ols*, *w*, *z=False*)

Moran's I for spatial autocorrelation in residuals from OLS regression

...

Parameters

- **ols** (OLS) – OLS regression object
- **w** (W) – Spatial weights instance
- **z** (*boolean*) – If set to True computes attributes eI, vI and zI. Due to computational burden of vI, defaults to False.

I
float – Moran's I statistic

eI
float – Moran's I expectation

vI
float – Moran's I variance

zI
float – Moran's I standardized value

Examples

```
>>> import numpy as np
>>> import pysal
>>> from ols import OLS
```

Open the csv file to access the data for analysis

```
>>> csv = pysal.open(pysal.examples.get_path('columbus.dbf'),'r')
```

Pull out from the csv the files we need ('HOVAL' as dependent as well as 'INC' and 'CRIME' as independent) and directly transform them into nx1 and nx2 arrays, respectively

```
>>> y = np.array([csv.by_col('HOVAL')]).T
>>> x = np.array([csv.by_col('INC'), csv.by_col('CRIME')]).T
```

Create the weights object from existing .gal file

```
>>> w = pysal.open(pysal.examples.get_path('columbus.gal'), 'r').read()
```

Row-standardize the weight object (not required although desirable in some cases)

```
>>> w.transform='r'
```

Run an OLS regression

```
>>> ols = OLS(y, x)
```

Run Moran's I test for residual spatial autocorrelation in an OLS model. This computes the traditional statistic applying a correction in the expectation and variance to account for the fact it comes from residuals instead of an independent variable

```
>>> m = pysal.spreg.diagnostics_sp.MoranRes(ols, w, z=True)
```

Value of the Moran's I statistic:

```
>>> print round(m.I,4)
0.1713
```

Value of the Moran's I expectation:

```
>>> print round(m.eI,4)
-0.0345
```

Value of the Moran's I variance:

```
>>> print round(m.vI,4)
0.0081
```

Value of the Moran's I standardized value. This is distributed as a standard Normal(0, 1)

```
>>> print round(m.zI,4)
2.2827
```

P-value of the standardized Moran's I value (z):

```
>>> print round(m.p_norm,4)
0.0224
```

class `pysal.spreg.diagnostics_sp.`**`AKtest`** (*iv*, *w*, *case='nosp'*)

Moran's I test of spatial autocorrelation for IV estimation. Implemented following the original reference Anselin and Kelejian (1997) [Anselin1997] ...

Parameters

- **`iv`** (`TSLS`) – Regression object from TSLS class
- **`w`** (`W`) – Spatial weights instance
- **`case`** (*`string`*) – Flag for special cases (default to 'nosp'):
  - 'nosp': Only NO spatial end. reg.
  - 'gen': General case (spatial lag + end. reg.)

`mi`

float – Moran's I statistic for IV residuals

`ak`

float –

Square of corrected Moran's I for residuals:

```
.. math::

      ak = \dfrac{N   imes I^*}{\phi^2}

Note: if case='nosp' then it simplifies to the LMerror
```

`p`

float – P-value of the test

Examples

We first need to import the needed modules. Numpy is needed to convert the data we read into arrays that `spreg` understands and `pysal` to perform all the analysis. The TSLS is required to run the model on which we will perform the tests.

```
>>> import numpy as np
>>> import pysal
>>> from twosls import TSLS
>>> from twosls_sp import GM_Lag
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.examples.get_path("columbus.dbf"),'r')
```

Before being able to apply the diagnostics, we have to run a model and, for that, we need the input variables. 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.

```
>>> 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.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X = np.array(X).T
```

In this case, we consider HOVAL (home value) as an endogenous regressor, so we acknowledge that by reading it in a different category.

```
>>> yd = []
>>> yd.append(db.by_col("HOVAL"))
>>> yd = np.array(yd).T
```

In order to properly account for the endogeneity, we have to pass in the instruments. Let us consider DISCBD (distance to the CBD) is a good one:

```
>>> q = []
>>> q.append(db.by_col("DISCBD"))
>>> q = np.array(q).T
```

Now we are good to run the model. It is an easy one line task.

```
>>> reg = TSLS(y, X, yd, q=q)
```

Now we are concerned with whether our non-spatial model presents spatial autocorrelation in the residuals. To assess this possibility, we can run the Anselin-Kelejian test, which is a version of the classical LM error test adapted for the case of residuals from an instrumental variables (IV) regression. First we need an extra object, the weights matrix, which 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.rook_from_shapefile(pysal.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:

```
>>> w.transform = 'r'
```

We are good to run the test. It is a very simple task:

```
>>> ak = AKtest(reg, w)
```

And explore the information obtained:

```
>>> print('AK test: %f       P-value: %f'%(ak.ak, ak.p))
AK test: 4.642895      P-value: 0.031182
```

The test also accomodates the case when the residuals come from an IV regression that includes a spatial lag of the dependent variable. The only requirement needed is to modify the `case` parameter when we call `AKtest`. First, let us run a spatial lag model:

```
>>> reg_lag = GM_Lag(y, X, yd, q=q, w=w)
```

And now we can run the AK test and obtain similar information as in the non-spatial model.

```
>>> ak_sp = AKtest(reg, w, case='gen')
>>> print('AK test: %f       P-value: %f'%(ak_sp.ak, ak_sp.p))
AK test: 1.157593      P-value: 0.281965
```

`spreg.diagnostics_tsls` — Diagnostics for 2SLS

The `spreg.diagnostics_tsls` module provides diagnostic tests for two stage least squares based models.

New in version 1.3. Diagnostics for two stage least squares regression estimations.

`pysal.spreg.diagnostics_tsls.`**`t_stat`**(*reg*, *z_stat=False*)

Calculates the t-statistics (or z-statistics) and associated p-values. [Greene2003]

Parameters

- **`reg`** (*`regression object`*) – output instance from a regression model
- **`z_stat`** (*`boolean`*) – If True run z-stat instead of t-stat

Returns **ts_result** – each tuple includes value of t statistic (or z statistic) and associated p-value

Return type list of tuples

Examples

We first need to import the needed modules. Numpy is needed to convert the data we read into arrays that `spreg` understands and `pysal` to perform all the analysis. The `diagnostics` module is used for the tests we will show here and the OLS and TSLS are required to run the models on which we will perform the tests.

```
>>> import numpy as np
>>> import pysal
>>> import pysal.spreg.diagnostics as diagnostics
>>> from pysal.spreg.ols import OLS
>>> from twosls import TSLS
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.examples.get_path("columbus.dbf"),'r')
```

Before being able to apply the diagnostics, we have to run a model and, for that, we need the input variables. 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.

```
>>> y = np.array(db.by_col("CRIME"))
>>> y = np.reshape(y, (49,1))
```

Extract INC (income) and HOVAL (home value) 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.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X.append(db.by_col("HOVAL"))
>>> X = np.array(X).T
```

Run an OLS regression. Since it is a non-spatial model, all we need is the dependent and the independent variable.

```
>>> reg = OLS(y,X)
```

Now we can perform a t-statistic on the model:

```
>>> testresult = diagnostics.t_stat(reg)
>>> print("%12.12f"%testresult[0][0], "%12.12f"%testresult[0][1], "%12.12f"
↪%testresult[1][0], "%12.12f"%testresult[1][1], "%12.12f"%testresult[2][0], "%12.
↪12f"%testresult[2][1])
('14.490373143689', '0.000000000000', '-4.780496191297', '0.000018289595', '-2.
↪654408642718', '0.010874504910')
```

We can also use the z-stat. For that, we re-build the model so we consider HOVAL as endogenous, instrument for it using DISCBD and carry out two stage least squares (TSLS) estimation.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X = np.array(X).T
>>> yd = []
>>> yd.append(db.by_col("HOVAL"))
>>> yd = np.array(yd).T
>>> q = []
>>> q.append(db.by_col("DISCBD"))
>>> q = np.array(q).T
```

Once the variables are read as different objects, we are good to run the model.

```
>>> reg = TSLS(y, X, yd, q)
```

With the output of the TSLS regression, we can perform a z-statistic:

```
>>> testresult = diagnostics.t_stat(reg, z_stat=True)
>>> print("%12.10f"%testresult[0][0], "%12.10f"%testresult[0][1], "%12.10f"
↪%testresult[1][0], "%12.10f"%testresult[1][1], "%12.10f"%testresult[2][0], "%12.
↪10f"%testresult[2][1])
('5.8452644705', '0.0000000051', '0.3676015668', '0.7131703463', '-1.9946891308',
↪'0.0460767956')
```

`pysal.spreg.diagnostics_tsls.`**`pr2_aspatial`**(*tslsreg*)

Calculates the pseudo r^2 for the two stage least squares regression.

Parameters **`tslsreg`** (*`two stage least squares regression object`*) – output instance from a two stage least squares regression model

Returns **pr2_result** – value of the squared pearson correlation between the y and tsls-predicted y vectors

Return type float

Examples

We first need to import the needed modules. Numpy is needed to convert the data we read into arrays that `spreg` understands and `pysal` to perform all the analysis. The TSLS is required to run the model on which we will perform the tests.

```
>>> import numpy as np
>>> import pysal
>>> from twosls import TSLS
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.examples.get_path("columbus.dbf"),'r')
```

Before being able to apply the diagnostics, we have to run a model and, for that, we need the input variables. 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.

```
>>> 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.

```
>>> X = []
>>> X.append(db.by_col("INC"))
>>> X = np.array(X).T
```

In this case, we consider HOVAL (home value) as an endogenous regressor, so we acknowledge that by reading it in a different category.

```
>>> yd = []
>>> yd.append(db.by_col("HOVAL"))
>>> yd = np.array(yd).T
```

In order to properly account for the endogeneity, we have to pass in the instruments. Let us consider DISCBD (distance to the CBD) is a good one:

```
>>> q = []
>>> q.append(db.by_col("DISCBD"))
>>> q = np.array(q).T
```

Now we are good to run the model. It is an easy one line task.

```
>>> reg = TSLS(y, X, yd, q=q)
```

In order to perform the pseudo R^2, we pass the regression object to the function and we are done!

```
>>> result = pr2_aspatial(reg)
>>> print("%1.6f"%result)
0.279361
```

`pysal.spreg.diagnostics_tsls.`**`pr2_spatial`**(*tslsreg*)

Calculates the pseudo r^2 for the spatial two stage least squares regression.

Parameters **stslsreg** (*spatial two stage least squares regression object*) – output instance from a spatial two stage least squares regression model

Returns **pr2_result** – value of the squared pearson correlation between the y and stsls-predicted y vectors

Return type float

Examples

We first need to import the needed modules. Numpy is needed to convert the data we read into arrays that `spreg` understands and `pysal` to perform all the analysis. The GM_Lag is required to run the model on which we will perform the tests and the `pysal.spreg.diagnostics` module contains the function with the test.

```
>>> import numpy as np
>>> import pysal
>>> import pysal.spreg.diagnostics as D
>>> from twosls_sp import GM_Lag
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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.

```
>>> y = np.array(db.by_col("HOVAL"))
>>> y = np.reshape(y, (49,1))
```

Extract 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 model adds a vector of ones to the independent variables passed in, but this can be overridden by passing constant=False.

```
>>> X = np.array(db.by_col("INC"))
>>> X = np.reshape(X, (49,1))
```

In this case, we consider CRIME (crime rates) as an endogenous regressor, so we acknowledge that by reading it in a different category.

```
>>> yd = np.array(db.by_col("CRIME"))
>>> yd = np.reshape(yd, (49,1))
```

In order to properly account for the endogeneity, we have to pass in the instruments. Let us consider DISCBD (distance to the CBD) is a good one:

```
>>> q = np.array(db.by_col("DISCBD"))
>>> q = np.reshape(q, (49,1))
```

Since this test has a spatial component, 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.rook_from_shapefile(pysal.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:

```
>>> w.transform = 'r'
```

Now we are good to run the spatial lag model. Make sure you pass all the parameters correctly and, if desired, pass the names of the variables as well so when you print the summary (reg.summary) they are included:

```
>>> 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')
```

Once we have a regression object, we can perform the spatial version of the pesudo R^2. It is as simple as one line!

```
>>> result = pr2_spatial(reg)
>>> print("%1.6f"%result)
0.299649
```

`spreg.error_sp` — GM/GMM Estimation of Spatial Error and Spatial Combo Models

The `spreg.error_sp` module provides spatial error and spatial combo (spatial lag with spatial error) regression estimation with and without endogenous variables; based on Kelejian and Prucha (1998 and 1999).

New in version 1.3. Spatial Error Models module

class `pysal.spreg.error_sp.`**`GM_Error`**(*y*, *x*, *w*, *vm=False*, *name_y=None*, *name_x=None*, *name_w=None*, *name_ds=None*)

GMM method for a spatial error model, 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
- **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

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

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 pysal
>>> import numpy as np
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.

```
>>> dbf = pysal.open(pysal.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 numpy array of shape (n, 1) as opposed to the also common shape of (n,) that other packages accept.

```
>>> 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.

```
>>> 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.

```
>>> w = pysal.open(pysal.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, 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 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([ 12.412 ,   0.5044,   0.1785])
>>> np.around(model.z_stat, decimals=6)
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
```

class `pysal.spreg.error_sp.`**`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

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 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

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 pysal
>>> import numpy as np
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.

```
>>> dbf = pysal.open(pysal.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.

```
>>> 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.

```
>>> 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).

```
>>> 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’.

```
>>> 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.

```
>>> w = pysal.open(pysal.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:

```
>>> 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.

```
>>> 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.

```
>>> 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])
```

class `pysal.spreg.error_sp.`**`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, name_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

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 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

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
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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.

```
>>> 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.

```
>>> 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`.

```
>>> w = pysal.rook_from_shapefile(pysal.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:

```
>>> 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.

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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']]
```

```
>>> print 'lambda: ', np.around(reg.betas[-1], 3)
lambda:  [ 0.254]
```

`spreg.error_sp_regimes` — GM/GMM Estimation of Spatial Error and Spatial Combo Models with Regimes

The `spreg.error_sp_regimes` module provides spatial error and spatial combo (spatial lag with spatial error) regression estimation with regimes and with and without endogenous variables; based on Kelejian and Prucha (1998 and 1999).

New in version 1.5. Spatial Error Models with regimes module

class `pysal.spreg.error_sp_regimes.`**`GM_Combo_Regimes`**(*y, x, regimes, yend=None, q=None, w=None, w_lags=1, lag_q=True, cores=False, constant_regi='many', cols2regi='all', regime_err_sep=False, regime_lag_sep=False, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None, name_regimes=None*)

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].

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
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **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)
- **constant_regi** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **regime_err_sep** (*boolean*) – If True, a separate regression is run for each regime.
- **regime_lag_sep** (*boolean*) – If True, the spatial parameter for spatial lag is also computed according to different regimes. If False (default), the spatial parameter is fixed accross regimes.
- **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
- **cores** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **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
- **name_regimes** (*string*) – Name of regime variable for use in the output

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

yend
: *array* – Two dimensional array with n rows and one column for each endogenous variable Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z
: *array* – nxk array of variables (combination of x and yend) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

pr2_e
: *float* – Pseudo R squared (squared correlation between y and ypred_e (using reduced form)) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

sig2
: *float* – Sigma squared used in computations (based on filtered residuals) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
: *array* – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z_stat
list of tuples – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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

name_regimes
string – Name of regimes variable for use in output

title
string – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

regimes
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
['one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_err_sep
boolean – If True, a separate regression is run for each regime.

regime_lag_sep
boolean – If True, the spatial parameter for spatial lag is also computed according to different regimes. If False (default), the spatial parameter is fixed accross regimes.

kr
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2','...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

Since we want to run a spatial lag model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> 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.

```
>>> model = GM_Combo_Regimes(y, x, regimes, w=w, name_y=y_var, name_x=x_var, name_
↪regimes=r_var, name_ds='NAT')
```

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 have a summary of the output by typing: model.summary Alternatively, we can check the betas:

```
>>> print model.name_z
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', '_Global_W_
↪HR90', 'lambda']
>>> print np.around(model.betas,4)
[[ 1.4607]
 [ 0.958 ]
 [ 0.5658]
 [ 9.113 ]
 [ 1.1338]
 [ 0.6517]
 [-0.4583]
 [ 0.6136]]
```

And lambda:

```
>>> print 'lambda: ', np.around(model.betas[-1], 4)
lambda:  [ 0.6136]
```

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. In this case we consider RD90 (resource deprivation) as an endogenous regressor. We use FP89 (families below poverty) for this and hence put it in the instruments parameter, ‘q’.

```
>>> yd_var = ['RD90']
>>> yd = np.array([db.by_col(name) for name in yd_var]).T
>>> q_var = ['FP89']
>>> q = np.array([db.by_col(name) for name in q_var]).T
```

And then we can run and explore the model analogously to the previous combo:

```
>>> model = GM_Combo_Regimes(y, x, regimes, yd, q, w=w, name_y=y_var, name_x=x_
↪var, name_yend=yd_var, name_q=q_var, name_regimes=r_var, name_ds='NAT')
>>> print model.name_z
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', '0_RD90', '1_
↪RD90', '_Global_W_HR90', 'lambda']
>>> print model.betas
[[ 3.41963782]
 [ 1.04065841]
 [ 0.16634393]
 [ 8.86544628]
 [ 1.85120528]
 [-0.24908469]
 [ 2.43014046]
 [ 3.61645481]
 [ 0.03308671]
 [ 0.18684992]]
>>> print np.sqrt(model.vm.diagonal())
[ 0.53067577  0.13271426  0.06058025  0.76406411  0.17969783  0.07167421
  0.28943121  0.25308326  0.06126529]
>>> print 'lambda: ', np.around(model.betas[-1], 4)
lambda:  [ 0.1868]
```

class `pysal.spreg.error_sp_regimes.`**`GM_Endog_Error_Regimes`**(*y, x, yend, q, regimes, w, cores=False, vm=False, constant_regi='many', cols2regi='all', regime_err_sep=False, regime_lag_sep=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None, name_regimes=None, summ=True, add_lag=False*)

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].

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)
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **w** (*pysal W object*) – Spatial weights object
- **constant_regi** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)

- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **regime_err_sep** (*boolean*) – If True, a separate regression is run for each regime.
- **regime_lag_sep** (*boolean*) – Always False, kept for consistency, ignored.
- **vm** (*boolean*) – If True, include variance-covariance matrix in summary results
- **cores** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **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
- **name_regimes** (*string*) – Name of regime variable for use in the output

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

yend
array – Two dimensional array with n rows and one column for each endogenous variable Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z
array – nxk array of variables (combination of x and yend) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

sig2
: *float* – Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details) Sigma squared used in computations

std_err
: *array* – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z_stat
: *list of tuples* – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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

name_regimes
: *string* – Name of regimes variable for use in output

title
: *string* – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

regimes
: *list* – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
: *['one', 'many']* – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
: *list, 'all'* – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_err_sep
: *boolean* – If True, a separate regression is run for each regime.

kr
: *int* – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
: *int* – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
: *int* – Number of different regimes in the 'regimes' list

multi
: *dictionary* – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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 pysal
>>> import numpy as np
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2',...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

For the endogenous models, we add the endogenous variable RD90 (resource deprivation) and we decide to instrument for it with FP89 (families below poverty):

```
>>> yd_var = ['RD90']
>>> yend = np.array([db.by_col(name) for name in yd_var]).T
>>> q_var = ['FP89']
>>> q = np.array([db.by_col(name) for name in q_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

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 `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> 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.

```
>>> model = GM_Endog_Error_Regimes(y, x, yend, q, regimes, w=w, name_y=y_var,␣
↪name_x=x_var, name_yend=yd_var, name_q=q_var, name_regimes=r_var, name_ds='NAT.
↪dbf')
```

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. Alternatively, we can have a summary of the output by typing: model.summary

```
>>> print model.name_z
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', '0_RD90', '1_
↪RD90', 'lambda']
>>> np.around(model.betas, decimals=5)
array([[ 3.59718],
       [ 1.0652 ],
       [ 0.15822],
       [ 9.19754],
       [ 1.88082],
       [-0.24878],
       [ 2.46161],
       [ 3.57943],
       [ 0.25564]])
>>> np.around(model.std_err, decimals=6)
array([ 0.522633,  0.137555,  0.063054,  0.473654,  0.18335 ,  0.072786,
        0.300711,  0.240413])
```

class `pysal.spreg.error_sp_regimes.`**`GM_Error_Regimes`**(*y, x, regimes, w, vm=False, name_y=None, name_x=None, name_w=None, constant_regi='many', cols2regi='all', regime_err_sep=False, regime_lag_sep=False, cores=False, name_ds=None, name_regimes=None*)

GMM method for a spatial error model with regimes, 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
- **`regimes`** (*`list`*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **`w`** (*`pysal W object`*) – Spatial weights object
- **`constant_regi`** (*`['one', 'many']`*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **`cols2regi`** (*`list, 'all'`*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **`regime_err_sep`** (*`boolean`*) – If True, a separate regression is run for each regime.
- **`regime_lag_sep`** (*`boolean`*) – Always False, kept for consistency, ignored.
- **`vm`** (*`boolean`*) – If True, include variance-covariance matrix in summary results
- **`cores`** (*`boolean`*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **`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
- **`name_regimes`** (*`string`*) – Name of regime variable for use in the output

`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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

vm
: *array* – Variance covariance matrix (kxk)

sig2
: *float* – Sigma squared used in computations Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
: *array* – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z_stat
: *list of tuples* – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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

name_regimes
: *string* – Name of regime variable for use in the output

title
string – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

regimes
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
['one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_err_sep
boolean – If True, a separate regression is run for each regime.

kr
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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 pysal
>>> import numpy as np
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = [’Var1’,’Var2’,’...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

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. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> 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.

```
>>> model = GM_Error_Regimes(y, x, regimes, w=w, name_y=y_var, name_x=x_var, name_
↪regimes=r_var, name_ds='NAT.dbf')
```

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). Alternatively, we can have a summary of the output by typing: model.summary

```
>>> print model.name_x
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', 'lambda']
>>> np.around(model.betas, decimals=6)
array([[ 0.074807],
       [ 0.786107],
       [ 0.538849],
       [ 5.103756],
       [ 1.196009],
       [ 0.600533],
       [ 0.364103]])
>>> np.around(model.std_err, decimals=6)
array([ 0.379864,  0.152316,  0.051942,  0.471285,  0.19867 ,  0.057252])
>>> np.around(model.z_stat, decimals=6)
array([[  0.196932,   0.843881],
       [  5.161042,   0.      ],
       [ 10.37397 ,   0.      ],
       [ 10.829455,   0.      ],
       [  6.02007 ,   0.      ],
       [ 10.489215,    0.       ]])
>>> np.around(model.sig2, decimals=6)
28.172732
```

`spreg.error_sp_het` — GM/GMM Estimation of Spatial Error and Spatial Combo Models with Heteroskedasticity

The `spreg.error_sp_het` module provides spatial error and spatial combo (spatial lag with spatial error) regression estimation with and without endogenous variables, and allowing for heteroskedasticity; based on Arraiz et al (2010) and Anselin (2011).

New in version 1.3. Spatial Error with Heteroskedasticity family of models

class `pysal.spreg.error_sp_het.`**`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 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
- **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.
- **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

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

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
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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 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.rook_from_shapefile(pysal.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_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 `pysal.spreg.error_sp`, it allows for inference on the spatial parameter.

```
>>> print reg.name_x
['CONSTANT', 'income', 'crime', 'lambda']
```

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:

```
>>> 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 ]]
```

class `pysal.spreg.error_sp_het.`**`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

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

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
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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 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) 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.

```
>>> 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).

```
>>> 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'.

```
>>> 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`.

```
>>> w = pysal.rook_from_shapefile(pysal.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 (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.

```
>>> 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 `pysal.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:

```
>>> 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]]
```

class `pysal.spreg.error_sp_het.`**`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 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 (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

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* – 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)

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 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

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
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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 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) 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.

```
>>> 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`.

```
>>> w = pysal.rook_from_shapefile(pysal.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'
```

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.

```
>>> 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 `pysal.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:

```
>>> 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:

```
>>> 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:

```
>>> 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]]
```

`spreg.error_sp_het_regimes` — GM/GMM Estimation of Spatial Error and Spatial Combo Models with Heteroskedasticity with Regimes

The `spreg.error_sp_het_regimes` module provides spatial error and spatial combo (spatial lag with spatial error) regression estimation with regimes and with and without endogenous variables, and allowing for heteroskedasticity; based on Arraiz et al (2010) and Anselin (2011).

New in version 1.5. Spatial Error with Heteroskedasticity and Regimes family of models

class `pysal.spreg.error_sp_het_regimes.`**`GM_Combo_Het_Regimes`**(*y, x, regimes, yend=None, q=None, w=None, w_lags=1, lag_q=True, max_iter=1, epsilon=1e-05, step1c=False, cores=False, inv_method='power_exp', constant_regi='many', cols2regi='all', regime_err_sep=False, regime_lag_sep=False, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None, name_regimes=None*)

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].

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)
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **w** (*pysal W object*) – Spatial weights object (always needed)
- **constant_regi** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)

- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **regime_err_sep** (*boolean*) – If True, a separate regression is run for each regime.
- **regime_lag_sep** (*boolean*) – If True, the spatial parameter for spatial lag is also computed according to different regimes. If False (default), the spatial parameter is fixed accross regimes.
- **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
- **cores** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **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
- **name_regimes** (*string*) – Name of regime variable for use in the output

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

yend
array – Two dimensional array with n rows and one column for each endogenous variable Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

q
array – Two dimensional array with n rows and one column for each external exogenous variable used as instruments Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z
array – nxk array of variables (combination of x and yend) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

h
array – nxl array of instruments (combination of x and q) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

iter_stop
string – Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

iteration
integer – Number of iterations of steps 2a and 2b from Arraiz et al. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

pr2_e
float – Pseudo R squared (squared correlation between y and ypred_e (using reduced form)) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
array – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`z_stat`
list of tuples – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`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

`name_regimes`
string – Name of regimes variable for use in output

`title`
string – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`regimes`
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

`constant_regi`
['one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

`cols2regi`
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

`regime_err_sep`
boolean – If True, a separate regression is run for each regime.

`regime_lag_sep`
boolean – If True, the spatial parameter for spatial lag is also computed according to different regimes. If False (default), the spatial parameter is fixed accross regimes.

`kr`
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2','...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

Since we want to run a spatial combo model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> 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.

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. We can have a summary of the output by typing: model.summary Alternatively, we can check the betas:

```
>>> reg = GM_Combo_Het_Regimes(y, x, regimes, w=w, step1c=True, name_y=y_var,␣
↪name_x=x_var, name_regimes=r_var, name_ds='NAT')
>>> print reg.name_z
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', '_Global_W_
↪HR90', 'lambda']
>>> print np.around(reg.betas,4)
[[ 1.4613]
 [ 0.9587]
 [ 0.5658]
 [ 9.1157]
 [ 1.1324]
 [ 0.6518]
 [-0.4587]
 [ 0.7174]]
```

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. In this case we consider RD90 (resource deprivation) as an endogenous regressor. We use FP89 (families below poverty) for this and hence put it in the instruments parameter, 'q'.

```
>>> yd_var = ['RD90']
>>> yd = np.array([db.by_col(name) for name in yd_var]).T
>>> q_var = ['FP89']
>>> q = np.array([db.by_col(name) for name in q_var]).T
```

And then we can run and explore the model analogously to the previous combo:

```
>>> reg = GM_Combo_Het_Regimes(y, x, regimes, yd, q, w=w, step1c=True, name_y=y_
↪var, name_x=x_var, name_yend=yd_var, name_q=q_var, name_regimes=r_var, name_ds=
↪'NAT')
>>> print reg.name_z
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', '0_RD90', '1_
↪RD90', '_Global_W_HR90', 'lambda']
>>> print reg.betas
[[ 3.41936197]
 [ 1.04071048]
 [ 0.16747219]
 [ 8.85820215]
 [ 1.847382  ]
 [-0.24545394]
 [ 2.43189808]
 [ 3.61328423]
 [ 0.03132164]
 [ 0.29544224]]
>>> print np.sqrt(reg.vm.diagonal())
[ 0.53103804  0.20835827  0.05755679  1.00496234  0.34332131  0.10259525
  0.3454436   0.37932794  0.07611667  0.07067059]
>>> print 'lambda: ', np.around(reg.betas[-1], 4)
lambda:  [ 0.2954]
```

class `pysal.spreg.error_sp_het_regimes.`**`GM_Endog_Error_Het_Regimes`**(*y, x, yend, q, regimes, w, max_iter=1, epsilon=1e-05, step1c=False, constant_regi='many', cols2regi='all', regime_err_sep=False, regime_lag_sep=False, inv_method='power_exp', cores=False, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None, name_regimes=None, summ=True, add_lag=False*)

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].

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)
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **w** (*pysal W object*) – Spatial weights object
- **constant_regi** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)

- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **regime_err_sep** (*boolean*) – If True, a separate regression is run for each regime.
- **regime_lag_sep** (*boolean*) – Always False, kept for consistency, ignored.
- **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
- **cores** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **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
- **name_regimes** (*string*) – Name of regime variable for use in the output

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

yend
array – Two dimensional array with n rows and one column for each endogenous variable Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

q
array – Two dimensional array with n rows and one column for each external exogenous variable used as instruments Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z
array – nxk array of variables (combination of x and yend) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

h
array – nxl array of instruments (combination of x and q) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

iter_stop
string – Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

iteration
integer – Number of iterations of steps 2a and 2b from Arraiz et al. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
array – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z_stat
list of tuples – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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

name_regimes
string – Name of regimes variable for use in output

title
string – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

regimes
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
['one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_err_sep
boolean – If True, a separate regression is run for each regime.

kr
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2',...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

For the endogenous models, we add the endogenous variable RD90 (resource deprivation) and we decide to instrument for it with FP89 (families below poverty):

```
>>> yd_var = ['RD90']
>>> yend = np.array([db.by_col(name) for name in yd_var]).T
>>> q_var = ['FP89']
>>> q = np.array([db.by_col(name) for name in q_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

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 `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> 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.

```
>>> reg = GM_Endog_Error_Het_Regimes(y, x, yend, q, regimes, w=w, step1c=True,␣
↪name_y=y_var, name_x=x_var, name_yend=yd_var, name_q=q_var, name_regimes=r_var,␣
↪name_ds='NAT.dbf')
```

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 `pysal.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 Alternatively, we can have a summary of the output by typing: model.summary

```
>>> print reg.name_z
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', '0_RD90', '1_
↪RD90', 'lambda']
```

```
>>> print np.around(reg.betas,4)
[[ 3.5944]
 [ 1.065 ]
 [ 0.1587]
 [ 9.184 ]
 [ 1.8784]
 [-0.2466]
 [ 2.4617]
 [ 3.5756]
 [ 0.2908]]
```

```
>>> print np.around(np.sqrt(reg.vm.diagonal()),4)
[ 0.5043  0.2132  0.0581  0.6681  0.3504  0.0999  0.3686  0.3402  0.028 ]
```

class `pysal.spreg.error_sp_het_regimes.`**`GM_Error_Het_Regimes`**(*y, x, regimes, w, max_iter=1, epsilon=1e-05, step1c=False, constant_regi='many', cols2regi='all', regime_err_sep=False, regime_lag_sep=False, cores=False, vm=False, name_y=None, name_x=None, name_w=None, name_ds=None, name_regimes=None*)

GMM method for a spatial error model with heteroskedasticity and regimes; 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
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **w** (*pysal W object*) – Spatial weights object
- **constant_regi** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes

  - **'many': a vector of ones is appended to x and considered** different per regime (default)

- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **regime_err_sep** (*boolean*) – If True, a separate regression is run for each regime.
- **regime_lag_sep** (*boolean*) – Always False, kept for consistency, ignored.
- **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.
- **vm** (*boolean*) – If True, include variance-covariance matrix in summary results
- **cores** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **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
- **name_regimes** (*string*) – Name of regime variable for use in the output

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

iter_stop
: *string* – Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

iteration
: *integer* – Number of iterations of steps 2a and 2b from Arraiz et al. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

vm
: *array* – Variance covariance matrix (kxk)

sig2
: *float* – Sigma squared used in computations Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
: *array* – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z_stat
: *list of tuples* – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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

name_regimes
: *string* – Name of regime variable for use in the output

title
: *string* – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

regimes
: *list* – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
: *['one', 'many']* – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_err_sep
boolean – If True, a separate regression is run for each regime.

kr
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2','...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

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. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> 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_Het_Regimes(y, x, regimes, w=w, step1c=True, name_y=y_var,␣
↪name_x=x_var, name_regimes=r_var, name_ds='NAT.dbf')
```

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 `pysal.spreg.error_sp`, it allows for inference on the spatial parameter. Alternatively, we can have a summary of the output by typing: model.summary

```
>>> print reg.name_x
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', 'lambda']
>>> np.around(reg.betas, decimals=6)
array([[ 0.009121],
       [ 0.812973],
       [ 0.549355],
       [ 5.00279 ],
       [ 1.200929],
       [ 0.614681],
       [ 0.429277]])
>>> np.around(reg.std_err, decimals=6)
array([ 0.355844,  0.221743,  0.059276,  0.686764,  0.35843 ,  0.092788,
        0.02524 ])
```

`spreg.error_sp_hom` — GM/GMM Estimation of Spatial Error and Spatial Combo Models

The `spreg.error_sp_hom` module provides spatial error and spatial combo (spatial lag with spatial error) regression estimation with and without endogenous variables, and includes inference on the spatial error parameter (lambda); based on Drukker et al. (2010) and Anselin (2011).

New in version 1.3. Hom family of models based on: [Drukker2013] Following: [Anselin2011]

class `pysal.spreg.error_sp_hom.`**`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

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

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
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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 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.rook_from_shapefile(pysal.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 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.

```
>>> 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 `pysal.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]
 [  0.7063   0.4967]
 [ -0.556    0.179 ]
 [  0.4129   0.1835]]
```

class `pysal.spreg.error_sp_hom.`**`GM_Endog_Error_Hom`**(*y, x, yend, q, 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

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)

`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 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

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
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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 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) 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.

```
>>> 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).

```
>>> 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'.

```
>>> 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`.

```
>>> w = pysal.rook_from_shapefile(pysal.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 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.

```
>>> reg = 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 `pysal.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:

```
>>> 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  23.496 ]
 [  0.4643   0.7382]
 [ -0.669    0.3943]
 [  0.4321   0.1927]]
```

class `pysal.spreg.error_sp_hom.`**`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

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 – 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)

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 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

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
```

Open data on Columbus neighborhood crime (49 areas) using pysal.open(). This is the DBF associated with the Columbus shapefile. Note that pysal.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.open(pysal.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 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) 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.

```
>>> 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`.

```
>>> w = pysal.rook_from_shapefile(pysal.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'
```

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.

```
>>> 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:

```
>>> 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:

```
>>> 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)))
[['CONSTANT' '111.7705' '67.75191']
 ['inc' '-0.30974' '1.16656']
 ['crime' '-1.36043' '0.6841']
 ['W_hoval' '-0.52908' '0.84428']
 ['lambda' '0.60116' '0.18605']]
```

`spreg.error_sp_hom_regimes` — GM/GMM Estimation of Spatial Error and Spatial Combo Models with Regimes

The `spreg.error_sp_hom_regimes` module provides spatial error and spatial combo (spatial lag with spatial error) regression estimation with regimes and with and without endogenous variables, and includes inference on the spatial error parameter (lambda); based on Drukker et al. (2010) and Anselin (2011).

New in version 1.5. Hom family of models with regimes.

class `pysal.spreg.error_sp_hom_regimes.`**`GM_Combo_Hom_Regimes`**(*y, x, regimes, yend=None, q=None, w=None, w_lags=1, lag_q=True, cores=False, max_iter=1, epsilon=1e-05, A1='het', constant_regi='many', cols2regi='all', regime_err_sep=False, regime_lag_sep=False, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None, name_regimes=None*)

GMM method for a spatial lag and error model with homoskedasticity, regimes 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)
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **w** (*pysal W object*) – Spatial weights object (always needed)
- **constant_regi** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **regime_err_sep** (*boolean*) – If True, a separate regression is run for each regime.
- **regime_lag_sep** (*boolean*) – If True, the spatial parameter for spatial lag is also computed according to different regimes. If False (default), the spatial parameter is fixed accross regimes.
- **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', 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
- **cores** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **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
- **name_regimes** (*string*) – Name of regime variable for use in the output

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

yend
array – Two dimensional array with n rows and one column for each endogenous variable Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

q
array – Two dimensional array with n rows and one column for each external exogenous variable used as instruments Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z
array – nxk array of variables (combination of x and yend) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

h
array – nxl array of instruments (combination of x and q) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

iter_stop
string – Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

iteration
integer – Number of iterations of steps 2a and 2b from Arraiz et al. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

pr2_e
float – Pseudo R squared (squared correlation between y and ypred_e (using reduced form)) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

sig2
float – Sigma squared used in computations (based on filtered residuals) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
array – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z_stat
list of tuples – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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

name_regimes
string – Name of regimes variable for use in output

title
string – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

regimes
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
['one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_err_sep
boolean – If True, a separate regression is run for each regime.

regime_lag_sep
boolean – If True, the spatial parameter for spatial lag is also computed according to different regimes. If False (default), the spatial parameter is fixed accross regimes.

kr
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2','...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

Since we want to run a spatial combo model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> 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.

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. We can have a summary of the output by typing: model.summary Alternatively, we can check the betas:

```
>>> reg = GM_Combo_Hom_Regimes(y, x, regimes, w=w, A1='hom_sc', name_y=y_var,␣
↪name_x=x_var, name_regimes=r_var, name_ds='NAT')
>>> print reg.name_z
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', '_Global_W_
↪HR90', 'lambda']
>>> print np.around(reg.betas,4)
[[ 1.4607]
 [ 0.9579]
 [ 0.5658]
 [ 9.1129]
 [ 1.1339]
 [ 0.6517]
 [-0.4583]
 [ 0.6634]]
```

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. In this case we consider RD90 (resource deprivation) as an endogenous regressor. We use FP89 (families below poverty) for this and hence put it in the instruments parameter, 'q'.

```
>>> yd_var = ['RD90']
>>> yd = np.array([db.by_col(name) for name in yd_var]).T
>>> q_var = ['FP89']
>>> q = np.array([db.by_col(name) for name in q_var]).T
```

And then we can run and explore the model analogously to the previous combo:

```
>>> reg = GM_Combo_Hom_Regimes(y, x, regimes, yd, q, w=w, A1='hom_sc', name_y=y_
↪var, name_x=x_var, name_yend=yd_var, name_q=q_var, name_regimes=r_var, name_ds=
↪'NAT')
>>> print reg.name_z
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', '0_RD90', '1_
↪RD90', '_Global_W_HR90', 'lambda']
>>> print reg.betas
[[ 3.4196478 ]
 [ 1.04065595]
 [ 0.16630304]
 [ 8.86570777]
 [ 1.85134286]
 [-0.24921597]
 [ 2.43007651]
 [ 3.61656899]
 [ 0.03315061]
 [ 0.22636055]]
>>> print np.sqrt(reg.vm.diagonal())
[ 0.53989913  0.13506086  0.06143434  0.77049956  0.18089997  0.07246848
  0.29218837  0.25378655  0.06184801  0.06323236]
>>> print 'lambda: ', np.around(reg.betas[-1], 4)
lambda:  [ 0.2264]
```

class `pysal.spreg.error_sp_hom_regimes.`**`GM_Endog_Error_Hom_Regimes`**(*y, x, yend, q, regimes, w, constant_regi='many', cols2regi='all', regime_err_sep=False, regime_lag_sep=False, max_iter=1, epsilon=1e-05, A1='het', cores=False, vm=False, name_y=None, name_x=None, name_yend=None, name_q=None, name_w=None, name_ds=None, name_regimes=None, summ=True, add_lag=False*)

GMM method for a spatial error model with homoskedasticity, regimes and endogenous variables. 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)
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **w** (*pysal W object*) – Spatial weights object
- **constant_regi** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **regime_err_sep** (*boolean*) – If True, a separate regression is run for each regime.
- **regime_lag_sep** (*boolean*) – Always False, kept for consistency, ignored.
- **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', then as in Drukker, Egger and Prucha (2010) and Drukker, Prucha and Raciborski (2010).
- **cores** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **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
- **name_regimes** (*string*) – Name of regime variable for use in the output

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) Only available in dictionary ‘multi’ when multiple regressions (see ‘multi’ below for details)

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 Only available in dictionary ‘multi’ when multiple regressions (see ‘multi’ below for details)

yend
array – Two dimensional array with n rows and one column for each endogenous variable Only available in dictionary ‘multi’ when multiple regressions (see ‘multi’ below for details)

q
array – Two dimensional array with n rows and one column for each external exogenous variable used as instruments Only available in dictionary ‘multi’ when multiple regressions (see ‘multi’ below for details)

z
array – nxk array of variables (combination of x and yend) Only available in dictionary ‘multi’ when multiple regressions (see ‘multi’ below for details)

h
array – nxl array of instruments (combination of x and q) Only available in dictionary ‘multi’ when multiple regressions (see ‘multi’ below for details)

iter_stop
string – Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al. Only available in dictionary ‘multi’ when multiple regressions (see ‘multi’ below for details)

iteration
integer – Number of iterations of steps 2a and 2b from Arraiz et al. Only available in dictionary ‘multi’ when multiple regressions (see ‘multi’ below for details)

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) Only available in dictionary ‘multi’ when multiple regressions (see ‘multi’ below for details)

sig2
float – Sigma squared used in computations Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
array – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z_stat
list of tuples – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

hth
float – H'H Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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

name_regimes
string – Name of regimes variable for use in output

title
string – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

regimes
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
['one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_err_sep
boolean – If True, a separate regression is run for each regime.

kr
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2','...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

For the endogenous models, we add the endogenous variable RD90 (resource deprivation) and we decide to instrument for it with FP89 (families below poverty):

```
>>> yd_var = ['RD90']
>>> yend = np.array([db.by_col(name) for name in yd_var]).T
>>> q_var = ['FP89']
>>> q = np.array([db.by_col(name) for name in q_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

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 `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> 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.

```
>>> reg = GM_Endog_Error_Hom_Regimes(y, x, yend, q, regimes, w=w, A1='hom_sc',␣
↪name_y=y_var, name_x=x_var, name_yend=yd_var, name_q=q_var, name_regimes=r_var,␣
↪name_ds='NAT.dbf')
```

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 `pysal.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. Alternatively, we can have a summary of the output by typing: model.summary

```
>>> print reg.name_z
['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', '0_RD90', '1_
↪RD90', 'lambda']
```

```
>>> print np.around(reg.betas,4)
[[ 3.5973]
 [ 1.0652]
 [ 0.1582]
 [ 9.198 ]
 [ 1.8809]
 [-0.2489]
 [ 2.4616]
 [ 3.5796]
 [ 0.2541]]
```

```
>>> print np.around(np.sqrt(reg.vm.diagonal()),4)
[ 0.5204  0.1371  0.0629  0.4721  0.1824  0.0725  0.2992  0.2395  0.024 ]
```

class `pysal.spreg.error_sp_hom_regimes.`**`GM_Error_Hom_Regimes`**(*y*, *x*, *regimes*, *w*, *max_iter=1*, *epsilon=1e-05*, *A1='het'*, *cores=False*, *constant_regi='many'*, *cols2regi='all'*, *regime_err_sep=False*, *regime_lag_sep=False*, *vm=False*, *name_y=None*, *name_x=None*, *name_w=None*, *name_ds=None*, *name_regimes=None*)

GMM method for a spatial error model with homoskedasticity, with regimes, 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
- **`regimes`** (*`list`*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **`w`** (*`pysal W object`*) – Spatial weights object
- **`constant_regi`** (*`['one', 'many']`*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **`cols2regi`** (*`list, 'all'`*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **`regime_err_sep`** (*`boolean`*) – If True, a separate regression is run for each regime.
- **`regime_lag_sep`** (*`boolean`*) – Always False, kept for consistency, ignored.
- **`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', 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
- **`cores`** (*`boolean`*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **`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
- **name_regimes** (*string*) – Name of regime variable for use in the output

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

iter_stop
string – Stop criterion reached during iteration of steps 2a and 2b from Arraiz et al. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

iteration
integer – Number of iterations of steps 2a and 2b from Arraiz et al. Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

vm
array – Variance covariance matrix (kxk)

sig2
float – Sigma squared used in computations Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`std_err`
array – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`z_stat`
list of tuples – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`xtx`
float – X'X Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`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

`name_regimes`
string – Name of regime variable for use in the output

`title`
string – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

`regimes`
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

`constant_regi`
['one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

`cols2regi`
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

`regime_err_sep`
boolean – If True, a separate regression is run for each regime.

`kr`
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

`kf`
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

`nr`
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

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
```

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; 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.open(pysal.examples.get_path("NAT.dbf"),'r')
```

Extract the HR90 column (homicide rates in 1990) 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_var = 'HR90'
>>> y = np.array([db.by_col(y_var)]).reshape(3085,1)
```

Extract UE90 (unemployment rate) and PS90 (population structure) vectors from the DBF to be used as independent variables in the regression. Other variables can be inserted by adding their names to x_var, such as x_var = ['Var1','Var2','...] 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.

```
>>> x_var = ['PS90','UE90']
>>> x = np.array([db.by_col(name) for name in x_var]).T
```

The different regimes in this data are given according to the North and South dummy (SOUTH).

```
>>> r_var = 'SOUTH'
>>> regimes = db.by_col(r_var)
```

Since we want to run a spatial lag model, we need to specify the spatial weights matrix that includes the spatial configuration of the observations. To do that, we can open an already existing gal file or create a new one. In this case, we will create one from `NAT.shp`.

```
>>> w = pysal.rook_from_shapefile(pysal.examples.get_path("NAT.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:

```
>>> 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_Regimes(y, x, regimes, w=w, name_y=y_var, name_x=x_var,␣
↪name_ds='NAT')
```

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 `pysal.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. Alternatively, we can have a summary of the output by typing: model.summary >>> print reg.name_x ['0_CONSTANT', '0_PS90', '0_UE90', '1_CONSTANT', '1_PS90', '1_UE90', 'lambda']

```
>>> print np.around(reg.betas,4)
[[ 0.069 ]
 [ 0.7885]
 [ 0.5398]
 [ 5.0948]
 [ 1.1965]
 [ 0.6018]
 [ 0.4104]]
```

```
>>> print np.sqrt(reg.vm.diagonal())
[ 0.39105854  0.15664624  0.05254328  0.48379958  0.20018799  0.05834139
  0.01882401]
```

`spreg.regimes` — Spatial Regimes

The `spreg.regimes` module provides different spatial regime estimation procedures.

New in version 1.5.

class `pysal.spreg.regimes.`**`Chow`**(*reg*)

Chow test of coefficient stability across regimes. The test is a particular case of the Wald statistic in which the constraint are setup according to the spatial or other type of regime structure

...

Parameters **`reg`** (*`regression object`*) – Regression object from PySAL.spreg which is assumed to have the following attributes:

- betas : coefficient estimates
- vm : variance covariance matrix of betas
- kr : Number of variables varying across regimes
- kryd : Number of endogenous variables varying across regimes
- kf : Number of variables fixed (global) across regimes
- nr : Number of regimes

`joint`

tuple – Pair of Wald statistic and p-value for the setup of global regime stability, that is all betas are the same across regimes.

`regi`

array – kr x 2 array with Wald statistic (col 0) and its p-value (col 1) for each beta that varies across regimes. The restrictions are setup to test for the global stability (all regimes have the same parameter) of the beta.

Examples

```
>>> import numpy as np
>>> import pysal
>>> from ols_regimes import OLS_Regimes
>>> db = pysal.open(pysal.examples.get_path('columbus.dbf'),'r')
>>> y_var = 'CRIME'
>>> y = np.array([db.by_col(y_var)]).reshape(49,1)
>>> x_var = ['INC','HOVAL']
>>> x = np.array([db.by_col(name) for name in x_var]).T
>>> r_var = 'NSA'
>>> regimes = db.by_col(r_var)
>>> olsr = OLS_Regimes(y, x, regimes, constant_regi='many', nonspat_diag=False,␣
↪spat_diag=False, name_y=y_var, name_x=x_var, name_ds='columbus', name_regimes=r_
↪var, regime_err_sep=False)
>>> print olsr.name_x_r #x_var
['CONSTANT', 'INC', 'HOVAL']
>>> print olsr.chow.regi
[[ 0.01020844  0.91952121]
 [ 0.46024939  0.49750745]
 [ 0.55477371  0.45637369]]
>>> print 'Joint test:'
Joint test:
>>> print olsr.chow.joint
(0.6339319928978806, 0.8886223520178802)
```

class pysal.spreg.regimes.**Regimes_Frame**(*x*, *regimes*, *constant_regi*, *cols2regi*, *names=None*, *yend=False*)

Setup framework to work with regimes. Basically it involves:

- Dealing with the constant in a regimes world
- Creating a sparse representation of X
- Generating a list of names of X taking into account regimes

...

Parameters

- **x** (*array*) – Two dimensional array with n rows and one column for each independent (exogenous) variable, excluding the constant
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **constant_regi** (*[False, 'one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - False: no constant term is appended in any way
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.

- **names** (None, *list of strings*) – Names of independent variables for use in output

Returns

- **x** (*csr sparse matrix*) – Sparse matrix containing X variables properly aligned for regimes regression. 'xsp' is of dimension (n, k*r) where 'r' is the number of different regimes The structure of the alignent is X1r1 X2r1 ... X1r2 X2r2 ...
- **names** (*None, list of strings*) – Names of independent variables for use in output conveniently arranged by regimes. The structure of the name is "**regimeName_**-_varName"
- **kr** (*int*) – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)
- **kf** (*int*) – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate
- **nr** (*int*) – Number of different regimes in the 'regimes' list

class `pysal.spreg.regimes.`**`Wald`**(*reg*, *r*, *q=None*)

Chi sq. Wald statistic to test for restriction of coefficients. Implementation following Greene [Greene2003] eq. (17-24), p. 488

...

Parameters

- **reg** (*regression object*) – Regression object from PySAL.spreg
- **r** (*array*) – Array of dimension Rxk (R being number of restrictions) with constrain setup.
- **q** (*array*) – Rx1 array with constants in the constraint setup. See Greene **[1]_** for reference.

w

float – Wald statistic

pvalue

float – P value for Wald statistic calculated as a Chi sq. distribution with R degrees of freedom

`pysal.spreg.regimes.`**`buildR`**(*kr*, *kf*, *nr*)

Build R matrix to globally test for spatial heterogeneity across regimes. The constraint setup reflects the null every beta is the same across regimes

Note: needs a placeholder for kryd in builR1var, set to 0

...

Parameters

- **kr** (*int*) – Number of variables that vary across regimes ("regimized")
- **kf** (*int*) – Number of variables that do not vary across regimes ("fixed" or global)
- **nr** (*int*) – Number of regimes

Returns **R** – Array with constrain setup to test stability across regimes of one variable

Return type array

`pysal.spreg.regimes.`**`buildR1var`**(*vari*, *kr*, *kf*, *kryd*, *nr*)

Build R matrix to test for spatial heterogeneity across regimes in one variable. The constraint setup reflects the null betas for variable 'vari' are the same across regimes

...

Parameters

- **vari** (*int*) – Position of the variable to be tested (order in the sequence of variables per regime)
- **kr** (*int*) – Number of variables that vary across regimes ("regimized")
- **kf** (*int*) – Number of variables that do not vary across regimes ("fixed" or global)
- **kryd** (*Number of endogenous variables varying across regimes*) –
- **nr** (*int*) – Number of regimes

Returns **R** – Array with constrain setup to test stability across regimes of one variable

Return type array

`pysal.spreg.regimes.`**`check_cols2regi`**(*constant_regi*, *cols2regi*, *x*, *yend=None*, *add_cons=True*)

Checks if dimensions of list cols2regi match number of variables.

`pysal.spreg.regimes.`**`regimeX_setup`**(*x*, *regimes*, *cols2regi*, *regimes_set*, *constant=False*)

Flexible full setup of a regime structure

NOTE: constant term, if desired in the model, should be included in the x already

...

Parameters

- **x** (*np.array*) – Dense array of dimension (n, k) with values for all observations IMPORTANT: constant term (if desired in the model) should be included
- **regimes** (*list*) – list of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **cols2regi** (*list*) – List of k booleans indicating whether each column should be considered as different per regime (True) or held constant across regimes (False)
- **regimes_set** (*list*) – List of ordered regimes tags
- **constant** (*[False, 'one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - False: no constant term is appended in any way
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime

Returns

xsp – Sparse matrix containing the full setup for a regimes model as specified in the arguments passed NOTE: columns are reordered so first are all the regime columns then all the global columns (this makes it much more efficient) Structure of the output matrix (assuming X1, X2 to vary across regimes and constant term, X3 and X4 to be global):

X1r1, X2r1, ... , X1r2, X2r2, ... , constant, X3, X4

Return type csr sparse matrix

`pysal.spreg.regimes.`**`set_name_x_regimes`**(*name_x*, *regimes*, *constant_regi*, *cols2regi*, *regimes_set*)

Generate the set of variable names in a regimes setup, according to the order of the betas

NOTE: constant term, if desired in the model, should be included in the x already

...

Parameters

- **name_x** (*list/None*) – If passed, list of strings with the names of the variables aligned with the original dense array x IMPORTANT: constant term (if desired in the model) should be included
- **regimes** (*list*) – list of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **constant_regi** (*[False, 'one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - False: no constant term is appended in any way
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime
- **cols2regi** (*list*) – List of k booleans indicating whether each column should be considered as different per regime (True) or held constant across regimes (False)
- **regimes_set** (*list*) – List of ordered regimes tags

Returns

Return type name_x_regi

pysal.spreg.regimes.**w_regime**(*w*, *regi_ids*, *regi_i*, *transform=True*, *min_n=None*)

Returns the subset of W matrix according to a given regime ID

...

pysal.spreg.regimes.**w**

pysal W object – Spatial weights object

pysal.spreg.regimes.**regi_ids**

list – Contains the location of observations in y that are assigned to regime regi_i

pysal.spreg.regimes.**regi_i**

string or float – The regime for which W will be subset

Returns **w_regi_i** – Subset of W for regime regi_i

Return type pysal W object

pysal.spreg.regimes.**w_regimes**(*w*, *regimes*, *regimes_set*, *transform=True*, *get_ids=None*, *min_n=None*)

######### DEPRECATED ########## Subsets W matrix according to regimes

...

pysal.spreg.regimes.**w**

pysal W object – Spatial weights object

pysal.spreg.regimes.**regimes**

list – list of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

pysal.spreg.regimes.**regimes_set**

list – List of ordered regimes tags

Returns **w_regi** – Dictionary containing the subsets of W according to regimes: [r1:w1, r2:w2, ..., rR:wR]

Return type dictionary

`pysal.spreg.regimes.`**`w_regimes_union`**(*w*, *w_regi_i*, *regimes_set*)

Combines the subsets of the W matrix according to regimes

...

`pysal.spreg.regimes.`**`w`**

pysal W object – Spatial weights object

`pysal.spreg.regimes.`**`w_regi_i`**

dictionary – Dictionary containing the subsets of W according to regimes: [r1:w1, r2:w2, ..., rR:wR]

`pysal.spreg.regimes.`**`regimes_set`**

list – List of ordered regimes tags

Returns **w_regi** – Spatial weights object containing the union of the subsets of W

Return type pysal W object

`pysal.spreg.regimes.`**`wald_test`**(*betas*, *r*, *q*, *vm*)

Chi sq. Wald statistic to test for restriction of coefficients. Implementation following Greene [Greene2003] eq. (17-24), p. 488

...

Parameters

- **`betas`** (*`array`*) – kx1 array with coefficient estimates
- **`r`** (*`array`*) – Array of dimension Rxk (R being number of restrictions) with constrain setup.
- **`q`** (*`array`*) – Rx1 array with constants in the constraint setup. See Greene [1]_ for reference.
- **`vm`** (*`array`*) – kxk variance-covariance matrix of coefficient estimates

Returns

- **w** (*float*) – Wald statistic
- **pvalue** (*float*) – P value for Wald statistic calculated as a Chi sq. distribution with R degrees of freedom

`pysal.spreg.regimes.`**`x2xsp`**(*x*, *regimes*, *regimes_set*)

Convert X matrix with regimes into a sparse X matrix that accounts for the regimes

...

`pysal.spreg.regimes.`**`x`**

np.array – Dense array of dimension (n, k) with values for all observations

`pysal.spreg.regimes.`**`regimes`**

list – list of n values with the mapping of each observation to a regime. Assumed to be aligned with ‘x’.

`pysal.spreg.regimes.`**`regimes_set`**

list – List of ordered regimes tags

Returns **xsp** – Sparse matrix containing X variables properly aligned for regimes regression. ‘xsp’ is of dimension (n, k*r) where ‘r’ is the number of different regimes The structure of the alignent is X1r1 X2r1 ... X1r2 X2r2 ...

Return type csr sparse matrix

`spreg.ml_error` — ML Estimation of Spatial Error Model

The `spreg.ml_error` module provides spatial error model estimation with maximum likelihood following Anselin (1988).

New in version 1.7. ML Estimation of Spatial Error Model

class `pysal.spreg.ml_error.`**`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
- **`method`** (*`string`*) – if 'full', brute force calculation (full matrix expressions) if 'ord', Ord eigenvalue method if 'LU', LU sparse matrix decomposition
- **`epsilon`** (*`float`*) – tolerance criterion in mimimize_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

`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

Examples

```
>>> import numpy as np
>>> import pysal as ps
>>> np.set_printoptions(suppress=True)  #prevent scientific format
>>> db = ps.open(ps.examples.get_path("south.dbf"),'r')
>>> ds_name = "south.dbf"
>>> y_name = "HR90"
>>> y = np.array(db.by_col(y_name))
>>> y.shape = (len(y),1)
>>> x_names = ["RD90","PS90","UE90","DV90"]
>>> x = np.array([db.by_col(var) for var in x_names]).T
>>> ww = ps.open(ps.examples.get_path("south_q.gal"))
>>> w = ww.read()
>>> ww.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)
>>> np.around(mlerr.betas, decimals=4)
array([[ 6.1492],
       [ 4.4024],
       [ 1.7784],
       [-0.3781],
       [ 0.4858],
       [ 0.2991]])
>>> "{0:.4f}".format(mlerr.lam)
'0.2991'
>>> "{0:.4f}".format(mlerr.mean_y)
'9.5493'
>>> "{0:.4f}".format(mlerr.std_y)
'7.0389'
>>> np.around(np.diag(mlerr.vm), decimals=4)
array([ 1.0648,  0.0555,  0.0454,  0.0061,  0.0148,  0.0014])
>>> np.around(mlerr.sig2, decimals=4)
array([[ 32.4069]])
>>> "{0:.4f}".format(mlerr.logll)
'-4471.4071'
>>> "{0:.4f}".format(mlerr.aic)
'8952.8141'
>>> "{0:.4f}".format(mlerr.schwarz)
'8979.0779'
>>> "{0:.4f}".format(mlerr.pr2)
'0.3058'
>>> "{0:.4f}".format(mlerr.utu)
'48534.9148'
>>> np.around(mlerr.std_err, decimals=4)
array([ 1.0319,  0.2355,  0.2132,  0.0784,  0.1217,  0.0378])
>>> np.around(mlerr.z_stat, decimals=4)
array([[  5.9593,   0.    ],
       [ 18.6902,   0.    ],
       [  8.3422,   0.    ],
       [ -4.8233,   0.    ],
       [  3.9913,   0.0001],
       [  7.9089,   0.    ]])
>>> mlerr.name_y
'HR90'
>>> mlerr.name_x
['CONSTANT', 'RD90', 'PS90', 'UE90', 'DV90', 'lambda']
>>> mlerr.name_w
'south_q.gal'
>>> mlerr.name_ds
'south.dbf'
>>> mlerr.title
```

```
'MAXIMUM LIKELIHOOD SPATIAL ERROR (METHOD = FULL)'
```

spreg.ml_error_regimes — ML Estimation of Spatial Error Model with Regimes

The spreg.ml_error_regimes module provides spatial error model with regimes estimation with maximum likelihood following Anselin (1988).

New in version 1.7. ML Estimation of Spatial Error Model

class pysal.spreg.ml_error_regimes.**ML_Error_Regimes**(*y, x, regimes, w=None, constant_regi='many', cols2regi='all', method='full', epsilon=1e-07, regime_err_sep=False, regime_lag_sep=False, cores=False, spat_diag=False, vm=False, name_y=None, name_x=None, name_w=None, name_ds=None, name_regimes=None*)

ML estimation of the spatial error model with regimes (note no consistency checks, diagnostics or constants added); 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
- **regimes** (*list*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **constant_regi** (*['one', 'many']*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **cols2regi** (*list, 'all'*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **w** (*Sparse matrix*) – Spatial weights sparse matrix
- **method** (*string*) – if 'full', brute force calculation (full matrix expressions) if 'ord', Ord eigenvalue computation if 'LU', LU sparse matrix decomposition
- **epsilon** (*float*) – tolerance criterion in mimimize_scalar function and inverse_product
- **regime_err_sep** (*boolean*) – If True, a separate regression is run for each regime.
- **regime_lag_sep** (*boolean*) – Always False, kept for consistency in function call, ignored.
- **cores** (*boolean*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **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
- **name_regimes** (*string*) – Name of regimes variable for use in output

summary
string – Summary of regression results and diagnostics (note: use in conjunction with the print command)

betas
array – (k+1)x1 array of estimated coefficients (lambda last)

lam
float – estimate of spatial autoregressive coefficient Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 the rho) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

method
string – log Jacobian method if 'full': brute force (full matrix computations) if 'ord', Ord eigenvalue computation if 'LU', LU sparse matrix decomposition

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 for lambda, sigma (2 x 2) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

sig2
float – Sigma squared used in computations Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

logll
float – maximized log-likelihood (including constant terms) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

pr2
float – Pseudo R squared (squared correlation between y and ypred) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
array – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z_stat
list of tuples – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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

name_regimes
string – Name of regimes variable for use in output

title
string – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

regimes
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
['one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_lag_sep
boolean – If True, the spatial parameter for spatial lag is also computed according to different regimes. If False (default), the spatial parameter is fixed accross regimes.

kr
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

Examples

Open data baltim.dbf using pysal and create the variables matrices and weights matrix.

```
>>> import numpy as np
>>> import pysal as ps
>>> db =  ps.open(ps.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","AGE","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'
```

Since in this example we are interested in checking whether the results vary by regimes, we use CITCOU to define whether the location is in the city or outside the city (in the county):

```
>>> regimes = db.by_col("CITCOU")
```

Now we can run the regression with all parameters:

```
>>> mlerr = ML_Error_Regimes(y,x,regimes,w=w,name_y=y_name,name_x=x_names,
↪     name_w=w_name,name_ds=ds_name,name_regimes="CITCOU")
>>> np.around(mlerr.betas, decimals=4)
array([[ -2.3949],
       [  4.8738],
       [ -0.0291],
       [  0.3328],
       [ 31.7962],
       [  2.981 ],
       [ -0.2371],
       [  0.8058],
       [  0.6177]])
>>> "{0:.6f}".format(mlerr.lam)
```

```
'0.617707'
>>> "{0:.6f}".format(mlerr.mean_y)
'44.307180'
>>> "{0:.6f}".format(mlerr.std_y)
'23.606077'
>>> np.around(mlerr.vm1, decimals=4)
array([[   0.005 ,   -0.3535],
       [  -0.3535,  441.3039]])
>>> np.around(np.diag(mlerr.vm), decimals=4)
array([ 58.5055,    2.4295,    0.0072,    0.0639,   80.5925,    3.161 ,
         0.012 ,    0.0499,    0.005 ])
>>> np.around(mlerr.sig2, decimals=4)
array([[ 209.6064]])
>>> "{0:.6f}".format(mlerr.logll)
'-870.333106'
>>> "{0:.6f}".format(mlerr.aic)
'1756.666212'
>>> "{0:.6f}".format(mlerr.schwarz)
'1783.481077'
>>> mlerr.title
'MAXIMUM LIKELIHOOD SPATIAL ERROR - REGIMES (METHOD = full)'
```

`spreg.ml_lag` — ML Estimation of Spatial Lag Model

The `spreg.ml_lag` module provides spatial lag model estimation with maximum likelihood following Anselin (1988).

New in version 1.7. ML Estimation of Spatial Lag Model

class `pysal.spreg.ml_lag.`**`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_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 eigenvalue method
- **epsilon** (*float*) – tolerance criterion in mimimize_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

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

```
>>> import numpy as np
>>> import pysal as ps
>>> db =  ps.open(ps.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)
>>> np.around(mllag.betas, decimals=4)
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)
'0.425885'
```

```
>>> "{0:.6f}".format(mllag.mean_y)
'44.307180'
>>> "{0:.6f}".format(mllag.std_y)
'23.606077'
>>> np.around(np.diag(mllag.vm1), decimals=4)
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])
>>> np.around(np.diag(mllag.vm), decimals=4)
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)
'151.458698'
>>> "{0:.6f}".format(mllag.logll)
'-832.937174'
>>> "{0:.6f}".format(mllag.aic)
'1687.874348'
>>> "{0:.6f}".format(mllag.schwarz)
'1724.744787'
>>> "{0:.6f}".format(mllag.pr2)
'0.727081'
>>> "{0:.4f}".format(mllag.pr2_e)
'0.7062'
>>> "{0:.4f}".format(mllag.utu)
'31957.7853'
>>> np.around(mllag.std_err, decimals=4)
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)
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],
       [-2.178 ,  0.0294],
       [ 4.6487,  0.    ],
       [ 0.4866,  0.6266],
       [ 7.4775,  0.    ]])
>>> mllag.name_y
'PRICE'
>>> mllag.name_x
['CONSTANT', 'NROOM', 'NBATH', 'PATIO', 'FIREPL', 'AC', 'GAR', 'AGE', 'LOTSZ',
↪'SQFT', 'W_PRICE']
>>> mllag.name_w
'baltim_q.gal'
>>> mllag.name_ds
'baltim.dbf'
>>> mllag.title
'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)
>>> np.around(mllag.betas, decimals=4)
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)
'0.425885'
>>> "{0:.6f}".format(mllag.mean_y)
'44.307180'
>>> "{0:.6f}".format(mllag.std_y)
'23.606077'
>>> np.around(np.diag(mllag.vm1), decimals=4)
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])
>>> np.around(np.diag(mllag.vm), decimals=4)
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)
'151.458698'
>>> "{0:.6f}".format(mllag.logll)
'-832.937174'
>>> "{0:.6f}".format(mllag.aic)
'1687.874348'
>>> "{0:.6f}".format(mllag.schwarz)
'1724.744787'
>>> "{0:.6f}".format(mllag.pr2)
'0.727081'
>>> "{0:.6f}".format(mllag.pr2_e)
'0.706198'
>>> "{0:.4f}".format(mllag.utu)
'31957.7853'
>>> np.around(mllag.std_err, decimals=4)
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)
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],
       [-2.178 ,  0.0294],
       [ 4.6487,  0.    ],
       [ 0.4866,  0.6266],
       [ 7.4775,  0.    ]])
>>> mllag.name_y
'PRICE'
>>> mllag.name_x
['CONSTANT', 'NROOM', 'NBATH', 'PATIO', 'FIREPL', 'AC', 'GAR', 'AGE', 'LOTSZ',
↪'SQFT', 'W_PRICE']
>>> mllag.name_w
'baltim_q.gal'
>>> mllag.name_ds
'baltim.dbf'
>>> mllag.title
'MAXIMUM LIKELIHOOD SPATIAL LAG (METHOD = ORD)'
```

`spreg.ml_lag_regimes` — ML Estimation of Spatial Lag Model with Regimes

The `spreg.ml_lag_regimes` module provides spatial lag model with regimes estimation with maximum likelihood following Anselin (1988).

New in version 1.7. ML Estimation of Spatial Lag Model with Regimes

class `pysal.spreg.ml_lag_regimes.`**`ML_Lag_Regimes`**(*y, x, regimes, w=None, constant_regi='many', cols2regi='all', method='full', epsilon=1e-07, regime_lag_sep=False, regime_err_sep=False, cores=False, spat_diag=False, vm=False, name_y=None, name_x=None, name_w=None, name_ds=None, name_regimes=None*)

ML estimation of the spatial lag model with regimes (note no consistency checks, diagnostics or constants added); 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
- **`regimes`** (*`list`*) – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.
- **`constant_regi`** (*`['one', 'many']`*) – Switcher controlling the constant term setup. It may take the following values:
  - **'one': a vector of ones is appended to x and held** constant across regimes
  - **'many': a vector of ones is appended to x and considered** different per regime (default)
- **`cols2regi`** (*`list, 'all'`*) – Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all' (default), all the variables vary by regime.
- **`w`** (*`Sparse matrix`*) – Spatial weights sparse matrix
- **`method`** (*`string`*) – if 'full', brute force calculation (full matrix expressions) if 'ord', Ord eigenvalue method if 'LU', LU sparse matrix decomposition
- **`epsilon`** (*`float`*) – tolerance criterion in mimimize_scalar function and inverse_product
- **`regime_lag_sep`** (*`boolean`*) – If True, the spatial parameter for spatial lag is also computed according to different regimes. If False (default), the spatial parameter is fixed accross regimes.
- **`cores`** (*`boolean`*) – Specifies if multiprocessing is to be used Default: no multiprocessing, cores = False Note: Multiprocessing may not work on all platforms.
- **`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
- **name_regimes** (*string*) – Name of regimes variable for use in output

summary
: *string* – Summary of regression results and diagnostics (note: use in conjunction with the print command)

betas
: *array* – (k+1)x1 array of estimated coefficients (rho first)

rho
: *float* – estimate of spatial autoregressive coefficient Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

method
: *string* – log Jacobian method if 'full': brute force (full matrix computations) if 'ord', Ord eigenvalue method if 'LU', LU sparse matrix decomposition

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 Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

sig2
: *float* – Sigma squared used in computations Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

logll
float – maximized log-likelihood (including constant terms) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

aic
float – Akaike information criterion Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

schwarz
float – Schwarz criterion Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

pr2_e
float – Pseudo R squared (squared correlation between y and ypred_e (using reduced form)) Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

std_err
array – 1xk array of standard errors of the betas Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

z_stat
list of tuples – z statistic; each tuple contains the pair (statistic, p-value), where each is a float Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

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

name_regimes
string – Name of regimes variable for use in output

title
string – Name of the regression method used Only available in dictionary 'multi' when multiple regressions (see 'multi' below for details)

regimes
list – List of n values with the mapping of each observation to a regime. Assumed to be aligned with 'x'.

constant_regi
['one', 'many'] – Ignored if regimes=False. Constant option for regimes. Switcher controlling the constant term setup. It may take the following values:

- **'one': a vector of ones is appended to x and held** constant across regimes
- **'many': a vector of ones is appended to x and considered** different per regime

cols2regi
list, 'all' – Ignored if regimes=False. Argument indicating whether each column of x should be considered as different per regime or held constant across regimes (False). If a list, k booleans indicating for each variable the option (True if one per regime, False to be held constant). If 'all', all the variables vary by regime.

regime_lag_sep
boolean – If True, the spatial parameter for spatial lag is also computed according to different regimes. If False (default), the spatial parameter is fixed accross regimes.

regime_err_sep
boolean – always set to False - kept for compatibility with other regime models

kr
int – Number of variables/columns to be "regimized" or subject to change by regime. These will result in one parameter estimate by regime for each variable (i.e. nr parameters per variable)

kf
int – Number of variables/columns to be considered fixed or global across regimes and hence only obtain one parameter estimate

nr
int – Number of different regimes in the 'regimes' list

multi
dictionary – Only available when multiple regressions are estimated, i.e. when regime_err_sep=True and no variable is fixed across regimes. Contains all attributes of each individual regression

Examples

Open data baltim.dbf using pysal and create the variables matrices and weights matrix.

```
>>> import numpy as np
>>> import pysal as ps
>>> db =  ps.open(ps.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","AGE","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'
```

Since in this example we are interested in checking whether the results vary by regimes, we use CITCOU to define whether the location is in the city or outside the city (in the county):

```
>>> regimes = db.by_col("CITCOU")
```

Now we can run the regression with all parameters:

```
>>> mllag = ML_Lag_Regimes(y,x,regimes,w=w,name_y=y_name,name_x=x_names,           ␣
↪    name_w=w_name,name_ds=ds_name,name_regimes="CITCOU")
>>> np.around(mllag.betas, decimals=4)
array([[-15.0059],
        [  4.496 ],
        [ -0.0318],
        [  0.35  ],
        [ -4.5404],
        [  3.9219],
        [ -0.1702],
        [  0.8194],
        [  0.5385]])
>>> "{0:.6f}".format(mllag.rho)
'0.538503'
>>> "{0:.6f}".format(mllag.mean_y)
'44.307180'
>>> "{0:.6f}".format(mllag.std_y)
'23.606077'
>>> np.around(np.diag(mllag.vm1), decimals=4)
array([  47.42   ,    2.3953,    0.0051,    0.0648,   69.6765,    3.2066,
          0.0116,    0.0486,    0.004 ,  390.7274])
>>> np.around(np.diag(mllag.vm), decimals=4)
array([ 47.42   ,   2.3953,   0.0051,   0.0648,  69.6765,   3.2066,
         0.0116,   0.0486,   0.004 ])
>>> "{0:.6f}".format(mllag.sig2)
'200.044334'
>>> "{0:.6f}".format(mllag.logll)
'-864.985056'
>>> "{0:.6f}".format(mllag.aic)
'1747.970112'
>>> "{0:.6f}".format(mllag.schwarz)
'1778.136835'
>>> mllag.title
'MAXIMUM LIKELIHOOD SPATIAL LAG - REGIMES (METHOD = full)'
```

`spreg.sur` — Seeming Unrelated Regression

The `spreg.sur` module provides SUR estimation.

New in version 1.11. SUR and 3SLS estimation

class `pysal.spreg.sur.`**`SUR`**(*bigy*, *bigX*, *w=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*)

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*) –
- **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*) –
- **epsilon** (*float; precision criterion to end iterations*) – default = 0.00001
- **verbose** (*boolean; flag to print out iteration number and value*) – of log det(sig) at the beginning and the end of the iteration
- **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*) –

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 Multipler 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)

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

Examples

First import pysal to load the spatial analysis tools.

```
>>> 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.

```
>>> 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. 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.

```
>>> 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.

```
>>> w = pysal.queen_from_shapefile(pysal.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 th database (db) and lists of variables (y_var and x_var) created above.

```
>>> bigy,bigX,bigyvars,bigXvars = pysal.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)'

```
>>> 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)
--------------------------------------------------------
Data set             :         nat
Weights matrix       :     unknown
Number of Equations  :           2                Number of Observations:       ␣
↪3085
Log likelihood (SUR):  -19902.966                Number of Iterations  :         ␣
↪ 1
----------

SUMMARY OF EQUATION 1
---------------------
Dependent Variable   :        HR80                Number of Variables   :         ␣
↪ 3
Mean dependent var   :      6.9276                Degrees of Freedom    :       ␣
↪3082
S.D. dependent var   :      6.8251

------------------------------------------------------------------------------------
↪--
            Variable     Coefficient       Std.Error     z-Statistic     ␣
↪Probability
------------------------------------------------------------------------------------
↪--
          Constant_1       5.1390718       0.2624673      19.5798587       0.
↪0000000
                PS80       0.6776481       0.1219578       5.5564132       0.
↪0000000
                UE80       0.2637240       0.0343184       7.6846277       0.
↪0000000
------------------------------------------------------------------------------------
↪--

SUMMARY OF EQUATION 2
---------------------
Dependent Variable   :        HR90                Number of Variables   :         ␣
↪ 3
Mean dependent var   :      6.1829                Degrees of Freedom    :       ␣
↪3082
```

```
S.D. dependent var  :      6.6403

-----------------------------------------------------------------------------------
↪--
            Variable     Coefficient       Std.Error     z-Statistic     ␣
↪Probability
-----------------------------------------------------------------------------------
↪--
          Constant_2       3.6139403       0.2534996      14.2561949       0.
↪0000000
                PS90       1.0260715       0.1121662       9.1477755       0.
↪0000000
                UE90       0.3865499       0.0341996      11.3027760       0.
↪0000000
-----------------------------------------------------------------------------------
↪--


REGRESSION DIAGNOSTICS
                                     TEST         DF       VALUE           PROB
                         LM test on Sigma         1      680.168           0.0000
                         LR test on Sigma         1      768.385           0.0000

OTHER DIAGNOSTICS - CHOW TEST
                                VARIABLES         DF       VALUE           PROB
                   Constant_1, Constant_2         1       26.729           0.0000
                               PS80, PS90         1        8.241           0.0041
                               UE80, UE90         1        9.384           0.0022

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                              DF       VALUE           PROB
Lagrange Multiplier (error)       2        1333.625        0.0000

ERROR CORRELATION MATRIX
  EQUATION 1  EQUATION 2
    1.000000    0.469548
    0.469548    1.000000
================================ END OF REPORT␣
↪=====================================
```

class `pysal.spreg.sur.`**`ThreeSLS`**(*bigy*, *bigX*, *bigyend*, *bigq*, *nonspat_diag=True*, *name_bigy=None*, *name_bigX=None*, *name_bigyend=None*, *name_bigq=None*, *name_ds=None*)

User class for 3SLS estimation

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*) –
- **bigq** (*dictionary with matrix of instruments by equation*) –
- **nonspat_diag** (*boolean; flag for non-spatial diagnostics, default = True*) –

- **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*) – equations default = None, but should be specified is done when sur_stackZ is used
- **name_ds** (*string; name for the data set*) –

bigy
dictionary with y values

bigZ
dictionary with matrix of exogenous and endogenous variables – for each equation

bigZHZH
dictionary with matrix of cross products Zhat_r'Zhat_s

bigZHy
dictionary with matrix of cross products Zhat_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 – equations

Examples

First import pysal to load the spatial analysis tools.

```
>>> 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.

```
>>> 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. 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

```
>>> 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).

```
>>> bigy,bigX,bigyvars,bigXvars = pysal.spreg.sur_utils.sur_dictxy(db,y_var,x_var)
>>> bigyend,bigyendvars = pysal.spreg.sur_utils.sur_dictZ(db,yend_var)
>>> bigq,bigqvars = pysal.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:

```
>>> 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]])}
```

```
>>> reg.tsls_inf
{0: array([[  0.23220853,  29.81916157,   0.        ],
       [  0.10373417,  13.77770036,   0.        ],
       [  0.03086193,   0.01601807,   0.98721998],
       [  0.11131999,  32.18584124,   0.        ]]), 1: array([[  0.28739415,  26.
↪52753638,   0.        ],
       [  0.09597031,  17.19606554,   0.        ],
       [  0.04089547,  -5.30204786,   0.00000011],
       [  0.13586789,  28.79638723,   0.        ]])}
```

`spreg.sur_error` — Spatial Error Seeming Unrelated Regression

The `spreg.sur_error` module provides SUR estimation for spatial error model

New in version 1.11. Spatial Error SUR estimation

class `pysal.spreg.sur_error.`**`BaseSURerrorML`**(*bigy*, *bigX*, *w*, *epsilon=1e-07*)
Base class for SUR Error estimation by Maximum Likelihood

requires: scipy.optimize.minimize_scalar and scipy.optimize.minimize

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*) –
- **epsilon** (*convergence criterion for ML iterations*) – default 0.0000001

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

corr
error correlation matrix

bigE
n by n_eq matrix of vectors of residuals for each equation

cliksurerr
concentrated log-likelihood from ML SUR Error (no constant)

class `pysal.spreg.sur_error.`**`SURerrorML`**(*bigy*, *bigX*, *w*, *nonspat_diag=True*, *spat_diag=False*, *vm=False*, *epsilon=1e-07*, *name_bigy=None*, *name_bigX=None*, *name_ds=None*, *name_w=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*) –
- **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*) –

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*

Examples

First import pysal to load the spatial analysis tools.

```
>>> 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.

```
>>> 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.

```
>>> 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 th database (db) and lists of variables (y_var and x_var) created above.

```
>>> bigy,bigX,bigyvars,bigXvars = pysal.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.

```
>>> 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:

```
>>> 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]]), 1: array([[ 3.04923009],
       [ 1.10972634],
       [ 0.47075682]])}
```

```
>>> reg.sur_inf
{0: array([[  0.36692181,  10.96224141,   0.        ],
       [  0.14129077,   6.26294579,   0.        ],
       [  0.04267954,   9.93517021,   0.        ]]), 1: array([[  0.33139969,   9.
↪20106497,   0.        ],
       [  0.13352591,   8.31094371,   0.        ],
       [  0.04004097,  11.756878  ,   0.        ]])}
```

spreg.sur_lag — Spatial Lag Seeming Unrelated Regression

The `spreg.sur_lag` module provides SUR estimation for spatial lag model

New in version 1.11. Spatial Lag SUR estimation

class `pysal.spreg.sur_lag.`**`SURlagIV`**(*bigy, bigX, bigyend=None, bigq=None, w=None, vm=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*)

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 spedified is done when sur_stackZ is used
- **name_bigq** (*dictionary with names of instrumental variables for each*) – equations 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*) –

w
spatial weights object

bigy
dictionary with y values

bigZ
dictionary with matrix of exogenous and endogenous variables – for each equation

bigyend
dictionary with matrix of endogenous variables for each – equation; contains Wy only if no other endogenous specified

bigq
dictionary with matrix of instrumental variables for each – equation; contains WX only if no other endogenous specified

bigZHZH
dictionary with matrix of cross products Zhat_r'Zhat_s

bigZHy
dictionary with matrix of cross products Zhat_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

resids
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

joinrho
test on joint significance of spatial autoregressive coefficient – tuple with test statistic, 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_w
string; name for the spatial weights

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 – equations

Examples

First import pysal to load the spatial analysis tools.

```
>>> 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.

```
>>> 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. 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

```
>>> 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).

```
>>> bigy,bigX,bigyvars,bigXvars = pysal.spreg.sur_utils.sur_dictxy(db,y_var,x_var)
>>> bigyend,bigyendvars = pysal.spreg.sur_utils.sur_dictZ(db,yend_var)
>>> bigq,bigqvars = pysal.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.

```
>>> 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:

```
>>> 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]])}
```

```
```

```
>>> reg.tsls_inf
{0: array([[  0.49128435,  14.15620899,   0.        ],
       [  0.11516292,  12.50787151,   0.        ],
       [  0.03204088,  -0.2409087 ,   0.80962588],
       [  0.1876025 ,  19.45875745,   0.        ],
       [  0.05450628,   0.06653605,   0.94695106]]), 1: array([[  0.44969956,  12.
↪47726211,   0.        ],
       [  0.10440241,  13.28674277,   0.        ],
       [  0.04150243,  -3.73761961,   0.00018577],
       [  0.19133145,  16.66451427,   0.        ],
       [  0.04394024,   5.87893596,   0.        ]])}
```

`pysal.weights` — Spatial Weights

`pysal.weights` — Spatial weights matrices

The `weights` Spatial weights for PySAL

New in version 1.0. Weights.

class `pysal.weights.weights.`**`W`**(*neighbors*, *weights=None*, *id_order=None*, *silent_island_warning=False*, *ids=None*)

Spatial weights.

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 will print a warning if the dataset contains any disconnected observations or islands. To silence this warning set this parameter to True.
- **`ids`** (*`list`*) – values to use for keys of the neighbors and weights dicts

`asymmetries`
list – of

`cardinalities`
dictionary – of

`diagW2`
array – of

`diagWtW`
array – of

`diagWtW_WW`
array – of

histogram
dictionary – of

id2i
dictionary – of

id_order
list – of

id_order_set
boolean – True if

islands
list – of

max_neighbors
int – maximum number of neighbors

mean_neighbors
int – mean number of neighbors

min_neighbors
int – minimum neighbor count

n
int – of

neighbor_offsets
list – ids of neighbors to a region in id_order

nonzero
int – Number of non-zero entries

pct_nonzero
float – Percentage of nonzero neighbor counts

s0
float – of

s1
float – of

s2
float – of

s2array
array – of

sd
float – of

sparse
sparse_matrix – SciPy sparse matrix instance

trcW2
float – of

trcWtW
float – of

trcWtW_WW
float – of

transform
string – of

Examples

```
>>> from pysal import W, lat2W
>>> 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], 3: [1, 1, 1], 4: [1, 1, 1, 1],␣
↪5: [1, 1, 1], 6: [1, 1], 7: [1, 1, 1], 8: [1, 1]}
>>> w = W(neighbors, weights)
>>> "%.3f"%w.pct_nonzero
'29.630'
```

Read from external gal file

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

Set weights implicitly

```
>>> 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
>>> w = lat2W(100, 100)
>>> w.trcW2
39600.0
>>> w.trcWtW
39600.0
>>> w.transform='r'
>>> round(w.trcW2, 3)
2530.722
>>> round(w.trcWtW, 3)
2533.667
```

Cardinality Histogram

```
>>> w=pysal.rook_from_shapefile(pysal.examples.get_path("sacramentot2.shp"))
>>> w.histogram
[(1, 1), (2, 6), (3, 33), (4, 103), (5, 114), (6, 73), (7, 35), (8, 17), (9, 9),␣
↪(10, 4), (11, 4), (12, 3), (13, 0), (14, 1)]
```

Disconnected observations (islands)

```
>>> w = pysal.W({1:[0],0:[1],2:[], 3:[]})
WARNING: there are 2 disconnected observations
Island ids:  [2, 3]
```

asymmetries
List of id pairs with asymmetric weights.

asymmetry(*intrinsic=True*)
Asymmetry check.

Parameters **intrinsic** (*boolean*) – default=True

intrinsic symmetry: $w_{i,j} == w_{j,i}$

if intrisic is False: symmetry is defined as $i \in N_j \ AND \ j \in N_i$ where N_j is the set of neighbors for j.

Returns **asymmetries** – empty if no asymmetries are found if asymmetries, then a list of (i,j) tuples is returned

Return type list

Examples

```
>>> from pysal import lat2W
>>> w=lat2W(3,3)
>>> w.asymmetry()
[]
>>> w.transform='r'
>>> w.asymmetry()
[(0, 1), (0, 3), (1, 0), (1, 2), (1, 4), (2, 1), (2, 5), (3, 0), (3, 4), (3,␣
↪6), (4, 1), (4, 3), (4, 5), (4, 7), (5, 2), (5, 4), (5, 8), (6, 3), (6, 7),␣
↪(7, 4), (7, 6), (7, 8), (8, 5), (8, 7)]
>>> result = w.asymmetry(intrinsic=False)
>>> result
[]
>>> neighbors={0:[1,2,3], 1:[1,2,3], 2:[0,1], 3:[0,1]}
>>> weights={0:[1,1,1], 1:[1,1,1], 2:[1,1], 3:[1,1]}
>>> w=W(neighbors,weights)
>>> w.asymmetry()
[(0, 1), (1, 0)]
```

cardinalities
Number of neighbors for each observation.

diagW2
Diagonal of WW.

See also:

trcW2

diagWtW
Diagonal of $W'W$.

See also:

trcWtW

diagWtW_WW
Diagonal of $W'W + WW$.

full()
Generate a full numpy array.

Returns **implicit** – first element being the full numpy array and second element keys being the ids associated with each row in the array.

Return type tuple

Examples

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

See also:

full

get_transform()
Getter for transform property.

Returns transformation

Return type string (or none)

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.weights[0]
[1.0, 1.0]
>>> w.transform
'O'
>>> w.transform='r'
>>> w.weights[0]
[0.5, 0.5]
>>> w.transform='b'
>>> w.weights[0]
[1.0, 1.0]
>>>
```

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.

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.id_order_set
True
```

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.

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

Returns offsets of the id's neighbors in id_order

Return type list

Examples

```
>>> from pysal import W
>>> neighbors={'c': ['b'], 'b': ['c', 'a'], 'a': ['b']}
>>> weights ={'c': [1.0], 'b': [1.0, 1.0], 'a': [1.0]}
>>> w=W(neighbors,weights)
>>> w.id_order = ['a','b','c']
>>> w.neighbor_offsets['b']
[2, 0]
>>> w.id_order = ['b','a','c']
>>> w.neighbor_offsets['b']
[2, 1]
```

nonzero
Number of nonzero weights.

pct_nonzero
Percentage of nonzero weights.

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

...

Parameters **new_ids** (*list*) – /ndarray Aligned list of new ids to be inserted. Note that first element of new_ids will replace first element of w.id_order, second element of new_ids replaces second element of w.id_order and so on.

Example

```
>>> import pysal as ps
>>> w = ps.lat2W(3, 3)
>>> w.id_order
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> w.neighbors[0]
[3, 1]
>>> new_ids = ['id%i'%id for id in w.id_order]
>>> _ = w.remap_ids(new_ids)
>>> w.id_order
['id0', 'id1', 'id2', 'id3', 'id4', 'id5', 'id6', 'id7', 'id8']
>>> w.neighbors['id0']
['id3', 'id1']
```

s0

s0 is defined as

$$s0 = \sum_i \sum_j w_{i,j}$$

s1

s1 is defined as

$$s1 = 1/2 \sum_i \sum_j (w_{i,j} + w_{j,i})^2$$

s2

s2 is defined as

$$s2 = \sum_j (\sum_i w_{i,j} + \sum_i w_{j,i})^2$$

s2array

Individual elements comprising s2.

See also:

s2

sd

Standard deviation of number of neighbors.

set_shapefile(*shapefile*, *idVariable=None*, *full=False*)

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

Parameters

- **shapefile** (*string*) – shapefile name used to construct weights
- **idVariable** (*string*) – name of attribute in shapefile to associate with ids in the weights
- **full** (*boolean*) – True - write out entire path for shapefile, False (default) only base of shapefile without extension

set_transform(*value='B'*)

Transformations of weights.

Notes

Transformations are applied only to the value of the weights at instantiation. Chaining of transformations cannot be done on a W instance.

Parameters **transform** (*string*) – not case sensitive)

:param .. table::: :widths: auto

transform string	value
B	Binary
R	Row-standardization (global sum=n)
D	Double-standardization (global sum=1)
V	Variance stabilizing
O	Restore original transformation (from instantiation)

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.weights[0]
[1.0, 1.0]
>>> w.transform
'O'
>>> w.transform='r'
>>> w.weights[0]
[0.5, 0.5]
>>> w.transform='b'
>>> w.weights[0]
[1.0, 1.0]
>>>
```

sparse
Sparse matrix object.

For any matrix manipulations required for w, w.sparse should be used. This is based on scipy.sparse.

to_WSP()
Generate a WSP object.

Returns **implicit** – Thin W class

Return type pysal.WSP

Examples

```
>>> import pysal as ps
>>> from pysal import W
>>> neighbors={'first':['second'],'second':['first','third'],'third':['second
↪']}
>>> weights={'first':[1],'second':[1,1],'third':[1]}
>>> w=W(neighbors,weights)
>>> wsp=w.towsp()
>>> isinstance(wsp, ps.weights.weights.WSP)
True
>>> wsp.n
```

```
3
>>> wsp.s0
4
```

See also:

WSP

towsp()

Generate a WSP object.

Returns **implicit** – Thin W class

Return type pysal.WSP

Examples

```
>>> import pysal as ps
>>> from pysal import W
>>> neighbors={'first':['second'],'second':['first','third'],'third':['second
↪']}
>>> weights={'first':[1],'second':[1,1],'third':[1]}
>>> w=W(neighbors,weights)
>>> wsp=w.towsp()
>>> isinstance(wsp, ps.weights.weights.WSP)
True
>>> wsp.n
3
>>> wsp.s0
4
```

See also:

WSP

transform

Getter for transform property.

Returns **transformation**

Return type string (or none)

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.weights[0]
[1.0, 1.0]
>>> w.transform
'O'
>>> w.transform='r'
>>> w.weights[0]
[0.5, 0.5]
>>> w.transform='b'
>>> w.weights[0]
[1.0, 1.0]
>>>
```

trcW2
Trace of WW.

See also:

diagW2

trcWtW
Trace of $W'W$.

See also:

diagWtW

trcWtW_WW
Trace of $W'W + WW$.

class pysal.weights.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.

n
int – description

s0
float – description

trcWtW_WW
float – description

Examples

From GAL information

```
>>> import scipy.sparse
>>> import pysal
>>> 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 = pysal.weights.WSP(sparse)
>>> w.s0
4.0
>>> w.trcWtW_WW
6.3949999999999996
>>> w.n
4
```

diagWtW_WW
Diagonal of $W'W + WW$.

classmethod from_W(*W*)
Constructs a WSP object from the W's sparse matrix

Parameters **W** (*pysal.weights.W*) – a pysal weights object with a sparse form and ids

Returns

Return type a WSP instance

s0

s0 is defined as –

$$s0 = \sum_i \sum_j w_{i,j}$$

to_W(*silent_island_warning=True*)

Construct a W object from the WSP's sparse matrix

Parameters **silence_island_warning** (*bool*) – a flag governing whether to state when islands are encountered.

trcWtW_WW

Trace of $W^{'}W + WW$.

weights.util — Utility functions on spatial weights

The `weights.util` module provides utility functions on spatial weights .. versionadded:: 1.0

`pysal.weights.util.`**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, ...) and 'string' ('id0', 'id1', 'id2', ...)

Returns **w** – instance of spatial weights class W

Return type *W*

Notes

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

Examples

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

`pysal.weights.util.`**`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

Return type spatial weights instance

Examples

```
>>> from pysal import block_weights
>>> 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.,  1.,  1.,  1.,  2.,  2.,  2.,
        2.,  2.,  2.,  2.,  2.,  2.,  2.,  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]}
```

`pysal.weights.util.`**`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** – combinations of size n taken from items

Return type generator

Examples

```
>>> x = range(4)
>>> for c in comb(x, 2):
...     print c
...
```

```
[0, 1]
[0, 2]
[0, 3]
[1, 2]
[1, 3]
[2, 3]
```

`pysal.weights.util.`**`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** – observation id is the key, value is a list of contiguity orders with a negative 1 in the ith position

Return type dictionary

Notes

Implements the algorithm in Anselin and Smirnov (1996) [Anselin1996b]

Examples

```
>>> from pysal import rook_from_shapefile as rfs
>>> w = rfs(pysal.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.weights.util.`**`higher_order`**(*w*, *k=2*)

Contiguity weights object of order k.

Parameters

- **w** (`W`) – spatial weights object
- **k** (*`int`*) – order of contiguity

Returns **implicit** – spatial weights object

Return type *W*

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

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

`pysal.weights.util.`**`shimbel`**(*w*)

Find the Shimbel matrix for first order contiguity matrix.

Parameters **w** (W) – spatial weights object

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

Return type list

Examples

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

`pysal.weights.util.`**`remap_ids`**(*w*, *old2new*, *id_order=[]*)

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** – Spatial weights object with new IDs

Return type *W*

Examples

```
>>> from pysal import lat2W, remap_ids
>>> 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.weights.util.`**`full2W`**(*m*, *ids=None*)

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** – PySAL weights object

Return type *W*

Examples

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

Create an array of zeros

```
>>> a = np.zeros((4, 4))
```

For loop to fill it with random numbers

```
>>> 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

```
>>> w = ps.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]], dtype=bool)
```

Create list of user ids

```
>>> ids = ['myID0', 'myID1', 'myID2', 'myID3']
>>> w = ps.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]], dtype=bool)
```

pysal.weights.util.**full**(*w*)

Generate a full numpy array.

Parameters **w** (W) – spatial weights object

Returns **(fullw, keys)** – first element being the full numpy array and second element keys being the ids associated with each row in the array.

Return type tuple

Examples

```
>>> from pysal import W, full
>>> 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.]])
>>> ids
['first', 'second', 'third']
```

pysal.weights.util.**WSP2W**(*wsp*, *silent_island_warning=False*)

Convert a pysal WSP object (thin weights matrix) to a pysal W object.

Parameters

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

Returns **w** – PySAL weights object

Return type *W*

Examples

```
>>> import pysal
```

Build a 10x10 scipy.sparse matrix for a rectangular 2x5 region of cells (rook contiguity), then construct a PySAL sparse weights object (wsp).

```
>>> sp = pysal.weights.lat2SW(2, 5)
>>> wsp = pysal.weights.WSP(sp)
>>> wsp.n
10
>>> print wsp.sparse[0].todense()
[[0 1 0 0 0 1 0 0 0 0]]
```

Convert this sparse weights object to a standard PySAL weights object.

```
>>> w = pysal.weights.WSP2W(wsp)
>>> w.n
10
>>> print w.full()[0][0]
[ 0.  1.  0.  0.  0.  1.  0.  0.  0.  0.]
```

`pysal.weights.util.`**`get_ids`**(*shapefile*, *idVariable*)

Gets the IDs from the DBF file that moves with a given shape file.

Parameters

- **shapefile** (*string*) – name of a shape file including suffix
- **idVariable** (*string*) – name of a column in the shapefile's DBF to use for ids

Returns **ids** – a list of IDs

Return type list

Examples

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

`pysal.weights.util.`**`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** – (n, 2) a data array of x and y coordinates

Return type array

Notes

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

Examples

Point shapefile

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

Polygon shapefile

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

pysal.weights.util.**min_threshold_distance**(*data*, *p=2*)

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
- **p** (*float*) – Minkowski p-norm distance metric parameter: 1<=p<=infinity 2: Euclidean distance 1: Manhattan distance

Returns **nnd** – maximum nearest neighbor distance between the n observations

Return type float

Examples

```
>>> from pysal.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.weights.util.**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** – instance of a scipy sparse matrix

Return type scipy.sparse.dia_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 strucuture of the contiguity type.

Examples

```
>>> from pysal import weights
>>> w9 = weights.lat2SW(3,3)
>>> w9[0,1]
1
>>> w9[3,6]
1
>>> w9r = weights.lat2SW(3,3, row_st=True)
>>> w9r[3,6]
0.33333333333333331
```

`pysal.weights.util.`**`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** – (w.n,1) local clustering coefficients

Return type array

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

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

`pysal.weights.util.`**`higher_order_sp`**(*w*, *k=2*, *shortest_path=True*, *diagonal=False*)

Contiguity weights for either a sparse W or pysal.weights.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** – WSP, type matches type of w argument

Return type *W*

Notes

Lower order contiguities are removed.

Examples

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

`pysal.weights.util.`**`hexLat2W`**(*nrows=5*, *ncols=5*)

Create a W object for a hexagonal lattice.

Parameters

- **nrows** (*int*) – number of rows
- **ncols** (*int*) – number of columns

Returns **w** – instance of spatial weights class W

Return type *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

```
>>> import pysal as ps
>>> w = ps.lat2W()
>>> w.neighbors[1]
[0, 6, 2]
>>> w.neighbors[21]
[16, 20, 22]
>>> wh = ps.hexLat2W()
>>> wh.neighbors[1]
[0, 6, 2, 5, 7]
>>> wh.neighbors[21]
[16, 20, 22]
>>>
```

pysal.weights.util.**regime_weights**(*regimes*)
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** (*array, list*) – ids of which regime an observation belongs to

Returns W

Return type spatial weights instance

Examples

```
>>> from pysal import regime_weights
>>> 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.,  1.,  1.,  1.,  2.,  2.,  2.,
        2.,  2.,  2.,  2.,  2.,  2.,  2.,  1.,  3.,  3.,  3.,  3.])
>>> w = regime_weights(regimes)
PendingDepricationWarning: regime_weights will be renamed to block_weights in␣
↪PySAL 2.0
>>> 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 = regime_weights(regimes)
PendingDepricationWarning: regime_weights will be renamed to block_weights in␣
↪PySAL 2.0
>>> w.neighbors
{0: [1], 1: [0], 2: [3], 3: [2], 4: [5, 8], 5: [4, 8], 6: [7], 7: [6], 8: [4, 5]}
```

Notes

regime_weights will be deprecated in PySAL 2.0 and renamed to block_weights.

`weights.user` — Convenience functions for spatial weights

The `weights.user` module provides convenience functions for spatial weights .. versionadded:: 1.0 Convenience functions for the construction of spatial weights based on contiguity and distance criteria.

`pysal.weights.user.`**`queen_from_shapefile`**(*shapefile*, *idVariable=None*, *sparse=False*)
Queen contiguity weights from a polygon shapefile.

Parameters

- **`shapefile`** (*`string`*) – name of polygon shapefile including suffix.
- **`idVariable`** (*`string`*) – name of a column in the shapefile's DBF to use for ids.
- **`sparse`** (*`boolean`*) – If True return WSP instance If False return W instance

Returns **w** – instance of spatial weights

Return type *W*

Examples

```
>>> wq=queen_from_shapefile(pysal.examples.get_path("columbus.shp"))
>>> "%.3f"%wq.pct_nonzero
'9.829'
>>> wq=queen_from_shapefile(pysal.examples.get_path("columbus.shp"),"POLYID")
>>> "%.3f"%wq.pct_nonzero
'9.829'
>>> wq=queen_from_shapefile(pysal.examples.get_path("columbus.shp"), sparse=True)
>>> pct_sp = wq.sparse.nnz *1. / wq.n**2
>>> "%.3f"%pct_sp
'0.098'
```

Notes

Queen contiguity defines as neighbors any pair of polygons that share at least one vertex in their polygon definitions.

See also:

`pysal.weights.W`

`pysal.weights.user.`**`rook_from_shapefile`**(*shapefile*, *idVariable=None*, *sparse=False*)
Rook contiguity weights from a polygon shapefile.

Parameters

- **`shapefile`** (*`string`*) – name of polygon shapefile including suffix.
- **`idVariable`** (*`string`*) – name of a column in the shapefile's DBF to use for ids
- **`sparse`** (*`boolean`*) – If True return WSP instance If False return W instance

Returns **w** – instance of spatial weights

Return type *W*

Examples

```
>>> wr=rook_from_shapefile(pysal.examples.get_path("columbus.shp"), "POLYID")
>>> "%.3f"%wr.pct_nonzero
'8.330'
>>> wr=rook_from_shapefile(pysal.examples.get_path("columbus.shp"), sparse=True)
>>> pct_sp = wr.sparse.nnz *1. / wr.n**2
>>> "%.3f"%pct_sp
'0.083'
```

Notes

Rook contiguity defines as neighbors any pair of polygons that share a common edge in their polygon definitions.

See also:

`pysal.weights.W`

`pysal.weights.user.`**`knnW_from_array`**(*array*, *k=2*, *p=2*, *ids=None*, *radius=None*)
Nearest neighbor weights from a numpy array.

Parameters

- **`data`** (*`array`*) – (n,m) attribute data, n observations on m attributes
- **`k`** (*`int`*) – number of nearest neighbors
- **`p`** (*`float`*) – Minkowski p-norm distance metric parameter: 1<=p<=infinity 2: Euclidean distance 1: Manhattan distance
- **`ids`** (*`list`*) – identifiers to attach to each observation
- **`radius`** (*`float`*) – If supplied arc_distances will be calculated based on the given radius. p will be ignored.

Returns **w** – instance; Weights object with binary weights.

Return type *W*

Examples

```
>>> import numpy as np
>>> x,y=np.indices((5,5))
>>> x.shape=(25,1)
>>> y.shape=(25,1)
>>> data=np.hstack([x,y])
>>> wnn2=knnW_from_array(data,k=2)
>>> wnn4=knnW_from_array(data,k=4)
>>> set([1, 5, 6, 2]) == set(wnn4.neighbors[0])
True
>>> set([0, 1, 10, 6]) == set(wnn4.neighbors[5])
True
>>> set([1, 5]) == set(wnn2.neighbors[0])
True
>>> set([0,6]) == set(wnn2.neighbors[5])
True
>>> "%.2f"%wnn2.pct_nonzero
'8.00'
```

```
>>> wnn4.pct_nonzero
16.0
>>> wnn4=knnW_from_array(data,k=4)
>>> set([ 1,5,6,2]) == set(wnn4.neighbors[0])
True
```

Notes

Ties between neighbors of equal distance are arbitrarily broken.

See also:

`pysal.weights.W`

`pysal.weights.user.`**`knnW_from_shapefile`**`(`*shapefile, k=2, p=2, idVariable=None, radius=None*`)`

Nearest neighbor weights from a shapefile.

Parameters

- **`shapefile`** (*`string`*) – shapefile name with shp suffix
- **`k`** (*`int`*) – number of nearest neighbors
- **`p`** (*`float`*) – Minkowski p-norm distance metric parameter: 1<=p<=infinity 2: Euclidean distance 1: Manhattan distance
- **`idVariable`** (*`string`*) – name of a column in the shapefile's DBF to use for ids
- **`radius`** (*`float`*) – If supplied arc_distances will be calculated based on the given radius. p will be ignored.

Returns **w** – instance; Weights object with binary weights

Return type *W*

Examples

Polygon shapefile

```
>>> wc=knnW_from_shapefile(pysal.examples.get_path("columbus.shp"))
>>> "%.4f"%wc.pct_nonzero
'4.0816'
>>> set([2,1]) == set(wc.neighbors[0])
True
>>> wc3=pysal.knnW_from_shapefile(pysal.examples.get_path("columbus.shp"),k=3)
>>> set(wc3.neighbors[0]) == set([2,1,3])
True
>>> set(wc3.neighbors[2]) == set([4,3,0])
True
```

1 offset rather than 0 offset

```
>>> wc3_1=knnW_from_shapefile(pysal.examples.get_path("columbus.shp"),k=3,
↪idVariable="POLYID")
>>> set([4,3,2]) == set(wc3_1.neighbors[1])
True
>>> wc3_1.weights[2]
[1.0, 1.0, 1.0]
```

```
>>> set([4,1,8]) == set(wc3_1.neighbors[2])
True
```

Point shapefile

```
>>> w=knnW_from_shapefile(pysal.examples.get_path("juvenile.shp"))
>>> w.pct_nonzero
1.1904761904761905
>>> w1=knnW_from_shapefile(pysal.examples.get_path("juvenile.shp"),k=1)
>>> "%.3f"%w1.pct_nonzero
'0.595'
>>>
```

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.

Ties between neighbors of equal distance are arbitrarily broken.

See also:

`pysal.weights.W`

`pysal.weights.user.`**`threshold_binaryW_from_array`**(*array*, *threshold*, *p=2*, *radius=None*)
Binary weights based on a distance threshold.

Parameters

- **array** (*array*) – (n,m) attribute data, n observations on m attributes
- **threshold** (*float*) – distance band
- **p** (*float*) – Minkowski p-norm distance metric parameter: 1<=p<=infinity 2: Euclidean distance 1: Manhattan distance
- **radius** (*float*) – If supplied arc_distances will be calculated based on the given radius. p will be ignored.

Returns **w** – instance Weights object with binary weights

Return type *W*

Examples

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

```
>>>
```

`pysal.weights.user.`**`threshold_binaryW_from_shapefile`**(*shapefile*, *threshold*, *p=2*, *idVariable=None*, *radius=None*)

Threshold distance based binary weights from a shapefile.

Parameters

- **`shapefile`** (*`string`*) – shapefile name with shp suffix
- **`threshold`** (*`float`*) – distance band
- **`p`** (*`float`*) – Minkowski p-norm distance metric parameter: 1<=p<=infinity 2: Euclidean distance 1: Manhattan distance
- **`idVariable`** (*`string`*) – name of a column in the shapefile's DBF to use for ids
- **`radius`** (*`float`*) – If supplied arc_distances will be calculated based on the given radius. p will be ignored.

Returns **w** – instance Weights object with binary weights

Return type *W*

Examples

```
>>> w = threshold_binaryW_from_shapefile(pysal.examples.get_path("columbus.shp"),
↪0.62,idVariable="POLYID")
>>> w.weights[1]
[1, 1]
```

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.

`pysal.weights.user.`**`threshold_continuousW_from_array`**(*array*, *threshold*, *p=2*, *alpha=-1*, *radius=None*)

Continuous weights based on a distance threshold.

Parameters

- **`array`** (*`array`*) – (n,m) attribute data, n observations on m attributes
- **`threshold`** (*`float`*) – distance band
- **`p`** (*`float`*) – Minkowski p-norm distance metric parameter: 1<=p<=infinity 2: Euclidean distance 1: Manhattan distance
- **`alpha`** (*`float`*) – distance decay parameter for weight (default -1.0) if alpha is positive the weights will not decline with distance.
- **`radius`** (*`If supplied arc_distances will be calculated`*) – based on the given radius. p will be ignored.

Returns **w** – instance; Weights object with continuous weights.

Return type *W*

Examples

inverse distance weights

```
>>> points=[(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
>>> wid=threshold_continuousW_from_array(points,11.2)
WARNING: there is one disconnected observation (no neighbors)
Island id:  [2]
>>> wid.weights[0]
[0.10000000000000001, 0.089442719099991588]
```

gravity weights

```
>>> wid2=threshold_continuousW_from_array(points,11.2,alpha=-2.0)
WARNING: there is one disconnected observation (no neighbors)
Island id:  [2]
>>> wid2.weights[0]
[0.01, 0.0079999999999999984]
```

`pysal.weights.user.`**`threshold_continuousW_from_shapefile`**(*shapefile*, *threshold*, *p=2*, *alpha=-1*, *idVariable=None*, *radius=None*)

Threshold distance based continuous weights from a shapefile.

Parameters

- **shapefile** (*string*) – shapefile name with shp suffix
- **threshold** (*float*) – distance band
- **p** (*float*) – Minkowski p-norm distance metric parameter: 1<=p<=infinity 2: Euclidean distance 1: Manhattan distance
- **alpha** (*float*) – distance decay parameter for weight (default -1.0) if alpha is positive the weights will not decline with distance.
- **idVariable** (*string*) – name of a column in the shapefile's DBF to use for ids
- **radius** (*float*) – If supplied arc_distances will be calculated based on the given radius. p will be ignored.

Returns **w** – instance; Weights object with continuous weights.

Return type *W*

Examples

```
>>> w = threshold_continuousW_from_shapefile(pysal.examples.get_path("columbus.shp
↪"),0.62,idVariable="POLYID")
>>> w.weights[1]
[1.6702346893743334, 1.7250729841938093]
```

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.

`pysal.weights.user.`**`kernelW`**(*points*, *k=2*, *function='triangular'*, *fixed=True*, *radius=None*, *diagonal=False*)

Kernel based weights.

Parameters

- **`points`** (*`array`*) – (n,k) n observations on k characteristics used to measure distances between the n objects
- **`k`** (*`int`*) – the number of nearest neighbors to use for determining bandwidth. Bandwidth taken as $h_i = max(dknn) \forall i$ where $dknn$ is a vector of k-nearest neighbor distances (the distance to the kth nearest neighbor for each observation).
- **`function`** (*`{'triangular','uniform','quadratic','epanechnikov', 'quartic','bisquare','gaussian'}`*) –

  $$z_{i,j} = d_{i,j}/h_i$$

  triangular

  $$K(z) = (1 - |z|) \ if |z| \le 1$$

  uniform

  $$K(z) = |z| \ if |z| \le 1$$

  quadratic

  $$K(z) = (3/4)(1 - z^2) \ if |z| \le 1$$

  epanechnikov

  $$K(z) = (1 - z^2) \ if |z| \le 1$$

  quartic

  $$K(z) = (15/16)(1 - z^2)^2 \ if |z| \le 1$$

  bisquare

  $$K(z) = (1 - z^2)^2 \ if |z| \le 1$$

  gaussian

  $$K(z) = (2\pi)^{(-1/2)} exp(-z^2/2)$$

- **`fixed`** (*`boolean`*) – If true then $h_i = h \forall i$. If false then bandwidth is adaptive across observations.
- **`radius`** (*`float`*) – If supplied arc_distances will be calculated based on the given radius. p will be ignored.
- **`diagonal`** (*`boolean`*) – If true, set diagonal weights = 1.0, if false (default) diagonal weights are set to value according to kernel function.

Returns **w** – instance of spatial weights

Return type *W*

Examples

```
>>> points=[(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
>>> kw=kernelW(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],
       [ 20.000002]])
```

use different k

```
>>> kw=kernelW(points,k=3)
>>> kw.neighbors[0]
[0, 1, 3, 4]
>>> kw.bandwidth
array([[ 22.36068201],
       [ 22.36068201],
       [ 22.36068201],
       [ 22.36068201],
       [ 22.36068201],
       [ 22.36068201]])
```

Diagonals to 1.0

```
>>> kq = kernelW(points,function='gaussian')
>>> kq.weights
{0: [0.3989422804014327, 0.35206533556593145, 0.3412334260702758], 1: [0.
↪35206533556593145, 0.3989422804014327, 0.2419707487162134, 0.3412334260702758,␣
↪0.31069657591175387], 2: [0.2419707487162134, 0.3989422804014327, 0.
↪31069657591175387], 3: [0.3412334260702758, 0.3412334260702758, 0.
↪3989422804014327, 0.3011374490937829, 0.26575287272131043], 4: [0.
↪31069657591175387, 0.31069657591175387, 0.3011374490937829, 0.3989422804014327,␣
↪0.35206533556593145], 5: [0.26575287272131043, 0.35206533556593145, 0.
↪3989422804014327]}
>>> kqd = kernelW(points, function='gaussian', diagonal=True)
>>> kqd.weights
{0: [1.0, 0.35206533556593145, 0.3412334260702758], 1: [0.35206533556593145, 1.0,␣
↪0.2419707487162134, 0.3412334260702758, 0.31069657591175387], 2: [0.
↪2419707487162134, 1.0, 0.31069657591175387], 3: [0.3412334260702758, 0.
↪3412334260702758, 1.0, 0.3011374490937829, 0.26575287272131043], 4: [0.
↪31069657591175387, 0.31069657591175387, 0.3011374490937829, 1.0, 0.
↪35206533556593145], 5: [0.26575287272131043, 0.35206533556593145, 1.0]}
```

`pysal.weights.user.`**`kernelW_from_shapefile`**(*shapefile*, *k=2*, *function='triangular'*, *idVariable=None*, *fixed=True*, *radius=None*, *diagonal=False*)

Kernel based weights.

Parameters

- **`shapefile`** (*`string`*) – shapefile name with shp suffix
- **`k`** (*`int`*) – the number of nearest neighbors to use for determining bandwidth. Bandwidth taken as $h_i = max(dknn) \forall i$ where $dknn$ is a vector of k-nearest neighbor distances (the

distance to the kth nearest neighbor for each observation).

- **function** (*{'triangular','uniform','quadratic','epanechnikov', 'quartic','bisquare','gaussian'}*) –

  $$z_{i,j} = d_{i,j}/h_i$$

  triangular

  $$K(z) = (1 - |z|) \ if |z| \le 1$$

  uniform

  $$K(z) = |z| \ if |z| \le 1$$

  quadratic

  $$K(z) = (3/4)(1 - z^2) \ if |z| \le 1$$

  epanechnikov

  $$K(z) = (1 - z^2) \ if |z| \le 1$$

  quartic

  $$K(z) = (15/16)(1 - z^2)^2 \ if |z| \le 1$$

  bisquare

  $$K(z) = (1 - z^2)^2 \ if |z| \le 1$$

  gaussian

  $$K(z) = (2\pi)^{(-1/2)} exp(-z^2/2)$$

- **idVariable** (*string*) – name of a column in the shapefile's DBF to use for ids
- **fixed** (*binary*) – If true then $h_i = h \forall i$. If false then bandwidth is adaptive across observations.
- **radius** (*float*) – If supplied arc_distances will be calculated based on the given radius. p will be ignored.
- **diagonal** (*boolean*) – If true, set diagonal weights = 1.0, if false (default) diagonal weights are set to value according to kernel function.

Returns **w** – instance of spatial weights

Return type *W*

Examples

```
>>> kw = pysal.kernelW_from_shapefile(pysal.examples.get_path("columbus.shp"),
↪idVariable='POLYID', function = 'gaussian')
```

```
>>> kwd = pysal.kernelW_from_shapefile(pysal.examples.get_path("columbus.shp"),
↪idVariable='POLYID', function = 'gaussian', diagonal = True)
>>> set(kw.neighbors[1]) == set([4, 2, 3, 1])
True
>>> set(kwd.neighbors[1]) == set([4, 2, 3, 1])
True
>>>
>>> set(kw.weights[1]) == set( [0.2436835517263174, 0.29090631630909874, 0.
↪29671172124745776, 0.3989422804014327])
True
>>> set(kwd.weights[1]) == set( [0.2436835517263174, 0.29090631630909874, 0.
↪29671172124745776, 1.0])
True
```

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.

`pysal.weights.user.`**`adaptive_kernelW`**(*points*, *bandwidths=None*, *k=2*, *function='triangular'*, *radius=None*, *diagonal=False*)

Kernel weights with adaptive bandwidths.

Parameters

- **points** (*array*) – (n,k) n observations on k characteristics used to measure distances between the n objects
- **bandwidths** (*float*) – or array-like (optional) the bandwidth h_i for the kernel. if no bandwidth is specified k is used to determine the adaptive bandwidth
- **k** (*int*) – the number of nearest neighbors to use for determining bandwidth. For fixed bandwidth, $h_i = max(dknn) \forall i$ where $dknn$ is a vector of k-nearest neighbor distances (the distance to the kth nearest neighbor for each observation). For adaptive bandwidths, $h_i = dknn_i$
- **function** (*{'triangular','uniform','quadratic','quartic', 'gaussian'}*) – kernel function defined as follows with

$$z_{i,j} = d_{i,j}/h_i$$

triangular

$$K(z) = (1 - |z|) \ if |z| \le 1$$

uniform

$$K(z) = |z| \ if |z| \le 1$$

quadratic

$$K(z) = (3/4)(1 - z^2) \ if |z| \le 1$$

quartic

$$K(z) = (15/16)(1 - z^2)^2 \ if |z| \le 1$$

gaussian

$$K(z) = (2\pi)^{(-1/2)} exp(-z^2/2)$$

- **radius** (*float*) – If supplied arc_distances will be calculated based on the given radius. p will be ignored.
- **diagonal** (*boolean*) – If true, set diagonal weights = 1.0, if false (default) diagonal weights are set to value according to kernel function.

Returns **w** – instance of spatial weights

Return type *W*

Examples

User specified bandwidths

```
>>> points=[(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
>>> bw=[25.0,15.0,25.0,16.0,14.5,25.0]
>>> kwa=adaptive_kernelW(points,bandwidths=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=adaptive_kernelW(points)
>>> kwea.weights[0]
[1.0, 0.10557289844279438, 9.99999900663795e-08]
>>> kwea.neighbors[0]
[0, 1, 3]
>>> kwea.bandwidth
array([[ 11.18034101],
       [ 11.18034101],
       [ 20.000002  ],
       [ 11.18034101],
       [ 14.14213704],
       [ 18.02775818]])
```

Endogenous adaptive bandwidths with Gaussian kernel

```
>>> kweag=adaptive_kernelW(points,function='gaussian')
>>> kweag.weights[0]
[0.3989422804014327, 0.2674190291577696, 0.2419707487162134]
>>> kweag.bandwidth
array([[ 11.18034101],
       [ 11.18034101],
       [ 20.000002  ],
       [ 11.18034101],
       [ 14.14213704],
       [ 18.02775818]])
```

with diagonal

```
>>> kweag = pysal.adaptive_kernelW(points, function='gaussian')
>>> kweagd = pysal.adaptive_kernelW(points, function='gaussian', diagonal=True)
>>> kweag.neighbors[0]
[0, 1, 3]
>>> kweagd.neighbors[0]
[0, 1, 3]
>>> kweag.weights[0]
[0.3989422804014327, 0.2674190291577696, 0.2419707487162134]
>>> kweagd.weights[0]
[1.0, 0.2674190291577696, 0.2419707487162134]
```

`pysal.weights.user.`**`adaptive_kernelW_from_shapefile`**(*shapefile, bandwidths=None, k=2, function='triangular', idVariable=None, radius=None, diagonal=False*)

Kernel weights with adaptive bandwidths.

Parameters

- **`shapefile`** (*string*) – shapefile name with shp suffix
- **`bandwidths`** (*float*) – or array-like (optional) the bandwidth h_i for the kernel. if no bandwidth is specified k is used to determine the adaptive bandwidth
- **`k`** (*int*) – the number of nearest neighbors to use for determining bandwidth. For fixed bandwidth, $h_i = max(dknn) \forall i$ where $dknn$ is a vector of k-nearest neighbor distances (the distance to the kth nearest neighbor for each observation). For adaptive bandwidths, $h_i = dknn_i$
- **`function`** (*{'triangular','uniform','quadratic','quartic', 'gaussian'}*) – kernel function defined as follows with

  $$z_{i,j} = d_{i,j}/h_i$$

  triangular

  $$K(z) = (1 - |z|) \ if |z| \le 1$$

  uniform

  $$K(z) = |z| \ if |z| \le 1$$

  quadratic

  $$K(z) = (3/4)(1 - z^2) \ if |z| \le 1$$

  quartic

  $$K(z) = (15/16)(1 - z^2)^2 \ if |z| \le 1$$

  gaussian

  $$K(z) = (2\pi)^{(-1/2)} exp(-z^2/2)$$

- **`idVariable`** (*string*) – name of a column in the shapefile's DBF to use for ids
- **`radius`** (*float*) – If supplied arc_distances will be calculated based on the given radius. p will be ignored.

- **diagonal** (*boolean*) – If true, set diagonal weights = 1.0, if false (default) diagonal weights are set to value according to kernel function.

Returns **w** – instance of spatial weights

Return type *W*

Examples

```
>>> kwa = pysal.adaptive_kernelW_from_shapefile(pysal.examples.get_path("columbus.
↪shp"), function='gaussian')
>>> kwad = pysal.adaptive_kernelW_from_shapefile(pysal.examples.get_path(
↪"columbus.shp"), function='gaussian', diagonal=True)
>>> kwa.neighbors[0]
[0, 2, 1]
>>> kwad.neighbors[0]
[0, 2, 1]
>>> kwa.weights[0]
[0.3989422804014327, 0.24966013701844503, 0.2419707487162134]
>>> kwad.weights[0]
[1.0, 0.24966013701844503, 0.2419707487162134]
```

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.

pysal.weights.user.**min_threshold_dist_from_shapefile**(*shapefile*, *radius=None*, *p=2*)
Kernel weights with adaptive bandwidths.

Parameters

- **shapefile** (*string*) – shapefile name with shp suffix.
- **radius** (*float*) – If supplied arc_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** – Maximum nearest neighbor distance between the n observations.

Return type float

Examples

```
>>> md = min_threshold_dist_from_shapefile(pysal.examples.get_path("columbus.shp
↪"))
>>> md
0.61886415807685413
>>> min_threshold_dist_from_shapefile(pysal.examples.get_path("stl_hom.shp"),␣
↪pysal.cg.sphere.RADIUS_EARTH_MILES)
31.846942936393717
```

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.

`pysal.weights.user.`**`build_lattice_shapefile`**(*nrows*, *ncols*, *outFileName*)
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

Return type *None*

`weights.Contiguity` — Contiguity based spatial weights

The `weights.Contiguity.` module provides for the construction and manipulation of spatial weights matrices based on contiguity criteria.

New in version 1.0.

`pysal.weights.Contiguity.`**`buildContiguity`**(*polygons*, *criterion='rook'*, *ids=None*)
This is a deprecated function.

It builds a contiguity W from the polygons provided. As such, it is now identical to calling the class constructors for Rook or Queen.

class `pysal.weights.Contiguity.`**`Queen`**(*polygons*, *method='binning'*, ***kw*)

`asymmetries`
List of id pairs with asymmetric weights.

`asymmetry`(*intrinsic=True*)
Asymmetry check.

Parameters **`intrinsic`** (*`boolean`*) – default=True

intrinsic symmetry: $w_{i,j} == w_{j,i}$

if intrisic is False: symmetry is defined as $i \in N_j \ AND \ j \in N_i$ where N_j is the set of neighbors for j.

Returns **asymmetries** – empty if no asymmetries are found if asymmetries, then a list of (i,j) tuples is returned

Return type list

Examples

```
>>> from pysal import lat2W
>>> w=lat2W(3,3)
>>> w.asymmetry()
[]
```

```
>>> w.transform='r'
>>> w.asymmetry()
[(0, 1), (0, 3), (1, 0), (1, 2), (1, 4), (2, 1), (2, 5), (3, 0), (3, 4), (3,␣
↪6), (4, 1), (4, 3), (4, 5), (4, 7), (5, 2), (5, 4), (5, 8), (6, 3), (6, 7),␣
↪(7, 4), (7, 6), (7, 8), (8, 5), (8, 7)]
>>> result = w.asymmetry(intrinsic=False)
>>> result
[]
>>> neighbors={0:[1,2,3], 1:[1,2,3], 2:[0,1], 3:[0,1]}
>>> weights={0:[1,1,1], 1:[1,1,1], 2:[1,1], 3:[1,1]}
>>> w=W(neighbors,weights)
>>> w.asymmetry()
[(0, 1), (1, 0)]
```

cardinalities
Number of neighbors for each observation.

diagW2
Diagonal of WW.

See also:

trcW2

diagWtW
Diagonal of $W^{'}W$.

See also:

trcWtW

diagWtW_WW
Diagonal of $W^{'}W + WW$.

classmethod from_dataframe (*df*, *geom_col='geometry'*, ***kwargs*)
Construct a weights object from a pandas dataframe with a geometry column. This will cast the polygons to PySAL polygons, then build the W using ids from the dataframe.

Parameters

- **df** (`DataFrame`) – a :class: *pandas.DataFrame* containing geometries to use for spatial weights
- **geom_col** (*string*) – the name of the column in *df* that contains the geometries. Defaults to *geometry*
- **idVariable** (*string*) – the name of the column to use as IDs. If nothing is provided, the dataframe index is used
- **ids** (*list*) – a list of ids to use to index the spatial weights object. Order is not respected from this list.
- **id_order** (*list*) – an ordered list of ids to use to index the spatial weights object. If used, the resulting weights object will iterate over results in the order of the names provided in this argument.

See also:

`pysal.weights.W`, `pysal.weights.Queen`

classmethod from_iterable (*iterable*, *sparse=False*, ***kwargs*)
Construct a weights object from a collection of arbitrary polygons. This will cast the polygons to PySAL polygons, then build the W.

Parameters

- **iterable** (*iterable*) – a collection of of shapes to be cast to PySAL shapes. Must support iteration. Contents should at least implement a *__geo_interface__* attribute or be able to be coerced to geometries using pysal.cg.asShape
- ****kw** (*keyword arguments*) – optional arguments for `pysal.weights.W`

See also:

`pysal.weights.W`, `pysal.weights.Queen`

classmethod from_shapefile (*filepath, idVariable=None, full=False, **kwargs*)

Queen contiguity weights from a polygon shapefile.

Parameters

- **shapefile** (*string*) – name of polygon shapefile including suffix.
- **idVariable** (*string*) – name of a column in the shapefile's DBF to use for ids.
- **sparse** (*boolean*) – If True return WSP instance If False return W instance

Returns **w** – instance of spatial weights

Return type *W*

Examples

```
>>> wq=Queen.from_shapefile(pysal.examples.get_path("columbus.shp"))
>>> "%.3f"%wq.pct_nonzero
'9.829'
>>> wq=Queen.from_shapefile(pysal.examples.get_path("columbus.shp"),"POLYID")
>>> "%.3f"%wq.pct_nonzero
'9.829'
>>> wq=Queen.from_shapefile(pysal.examples.get_path("columbus.shp"),␣
↪sparse=True)
>>> pct_sp = wq.sparse.nnz *1. / wq.n**2
>>> "%.3f"%pct_sp
'0.098'
```

Notes

Queen contiguity defines as neighbors any pair of polygons that share at least one vertex in their polygon definitions.

See also:

`pysal.weights.W`, `pysal.weights.Queen`

full()

Generate a full numpy array.

Returns **implicit** – first element being the full numpy array and second element keys being the ids associated with each row in the array.

Return type tuple

Examples

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

See also:

full

get_transform()
Getter for transform property.

Returns transformation

Return type string (or none)

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.weights[0]
[1.0, 1.0]
>>> w.transform
'O'
>>> w.transform='r'
>>> w.weights[0]
[0.5, 0.5]
>>> w.transform='b'
>>> w.weights[0]
[1.0, 1.0]
>>>
```

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.

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.id_order_set
True
```

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.

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

Returns offsets of the id's neighbors in id_order

Return type list

Examples

```
>>> from pysal import W
>>> neighbors={'c': ['b'], 'b': ['c', 'a'], 'a': ['b']}
>>> weights ={'c': [1.0], 'b': [1.0, 1.0], 'a': [1.0]}
>>> w=W(neighbors,weights)
>>> w.id_order = ['a','b','c']
>>> w.neighbor_offsets['b']
[2, 0]
>>> w.id_order = ['b','a','c']
>>> w.neighbor_offsets['b']
[2, 1]
```

nonzero
Number of nonzero weights.

pct_nonzero
Percentage of nonzero weights.

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

...

Parameters **new_ids** (*list*) – /ndarray Aligned list of new ids to be inserted. Note that first element of new_ids will replace first element of w.id_order, second element of new_ids replaces second element of w.id_order and so on.

Example

```
>>> import pysal as ps
>>> w = ps.lat2W(3, 3)
>>> w.id_order
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> w.neighbors[0]
[3, 1]
>>> new_ids = ['id%i'%id for id in w.id_order]
>>> _ = w.remap_ids(new_ids)
>>> w.id_order
['id0', 'id1', 'id2', 'id3', 'id4', 'id5', 'id6', 'id7', 'id8']
>>> w.neighbors['id0']
['id3', 'id1']
```

s0

s0 is defined as

$$s0 = \sum_i \sum_j w_{i,j}$$

s1

s1 is defined as

$$s1 = 1/2 \sum_i \sum_j (w_{i,j} + w_{j,i})^2$$

s2

s2 is defined as

$$s2 = \sum_j (\sum_i w_{i,j} + \sum_i w_{j,i})^2$$

s2array

Individual elements comprising s2.

See also:

s2

sd

Standard deviation of number of neighbors.

set_shapefile (*shapefile*, *idVariable=None*, *full=False*)

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

Parameters

- **shapefile** (*string*) – shapefile name used to construct weights
- **idVariable** (*string*) – name of attribute in shapefile to associate with ids in the weights
- **full** (*boolean*) – True - write out entire path for shapefile, False (default) only base of shapefile without extension

set_transform (*value='B'*)

Transformations of weights.

Notes

Transformations are applied only to the value of the weights at instantiation. Chaining of transformations cannot be done on a W instance.

Parameters **transform** (*string*) – not case sensitive)

:param .. table::: :widths: auto

transform string	value
B	Binary
R	Row-standardization (global sum=n)
D	Double-standardization (global sum=1)
V	Variance stabilizing
O	Restore original transformation (from instantiation)

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.weights[0]
[1.0, 1.0]
>>> w.transform
'O'
>>> w.transform='r'
>>> w.weights[0]
[0.5, 0.5]
>>> w.transform='b'
>>> w.weights[0]
[1.0, 1.0]
>>>
```

sparse
Sparse matrix object.

For any matrix manipulations required for w, w.sparse should be used. This is based on scipy.sparse.

to_WSP()
Generate a WSP object.

Returns **implicit** – Thin W class

Return type pysal.WSP

Examples

```
>>> import pysal as ps
>>> from pysal import W
>>> neighbors={'first':['second'],'second':['first','third'],'third':['second
↪']}
>>> weights={'first':[1],'second':[1,1],'third':[1]}
>>> w=W(neighbors,weights)
>>> wsp=w.towsp()
>>> isinstance(wsp, ps.weights.weights.WSP)
True
>>> wsp.n
```

```
3
>>> wsp.s0
4
```

See also:

WSP

towsp()
Generate a WSP object.

Returns **implicit** – Thin W class

Return type pysal.WSP

Examples

```
>>> import pysal as ps
>>> from pysal import W
>>> neighbors={'first':['second'],'second':['first','third'],'third':['second
↪']}
>>> weights={'first':[1],'second':[1,1],'third':[1]}
>>> w=W(neighbors,weights)
>>> wsp=w.towsp()
>>> isinstance(wsp, ps.weights.weights.WSP)
True
>>> wsp.n
3
>>> wsp.s0
4
```

See also:

WSP

transform
Getter for transform property.

Returns **transformation**

Return type string (or none)

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.weights[0]
[1.0, 1.0]
>>> w.transform
'O'
>>> w.transform='r'
>>> w.weights[0]
[0.5, 0.5]
>>> w.transform='b'
>>> w.weights[0]
[1.0, 1.0]
>>>
```

trcW2
Trace of WW.

See also:

diagW2

trcWtW
Trace of $W'W$.

See also:

diagWtW

trcWtW_WW
Trace of $W'W + WW$.

class pysal.weights.Contiguity.**Rook**(*polygons*, *method='binning'*, ***kw*)

asymmetries
List of id pairs with asymmetric weights.

asymmetry(*intrinsic=True*)
Asymmetry check.

Parameters intrinsic (*boolean*) – default=True

intrinsic symmetry: $w_{i,j} == w_{j,i}$

if intrisic is False: symmetry is defined as $i \in N_j \ AND \ j \in N_i$ where N_j is the set of neighbors for j.

Returns asymmetries – empty if no asymmetries are found if asymmetries, then a list of (i,j) tuples is returned

Return type list

Examples

```
>>> from pysal import lat2W
>>> w=lat2W(3,3)
>>> w.asymmetry()
[]
>>> w.transform='r'
>>> w.asymmetry()
[(0, 1), (0, 3), (1, 0), (1, 2), (1, 4), (2, 1), (2, 5), (3, 0), (3, 4), (3,␣
↪6), (4, 1), (4, 3), (4, 5), (4, 7), (5, 2), (5, 4), (5, 8), (6, 3), (6, 7),␣
↪(7, 4), (7, 6), (7, 8), (8, 5), (8, 7)]
>>> result = w.asymmetry(intrinsic=False)
>>> result
[]
>>> neighbors={0:[1,2,3], 1:[1,2,3], 2:[0,1], 3:[0,1]}
>>> weights={0:[1,1,1], 1:[1,1,1], 2:[1,1], 3:[1,1]}
>>> w=W(neighbors,weights)
>>> w.asymmetry()
[(0, 1), (1, 0)]
```

cardinalities
Number of neighbors for each observation.

diagW2

Diagonal of WW.

See also:

trcW2

diagWtW

Diagonal of $W^{'}W$.

See also:

trcWtW

diagWtW_WW

Diagonal of $W^{'}W + WW$.

classmethod from_dataframe(*df*, *geom_col='geometry'*, *idVariable=None*, *ids=None*, *id_order=None*, ***kwargs*)

Construct a weights object from a pandas dataframe with a geometry column. This will cast the polygons to PySAL polygons, then build the W using ids from the dataframe.

Parameters

- **df** (*DataFrame*) – a :class: *pandas.DataFrame* containing geometries to use for spatial weights
- **geom_col** (*string*) – the name of the column in *df* that contains the geometries. Defaults to *geometry*
- **idVariable** (*string*) – the name of the column to use as IDs. If nothing is provided, the dataframe index is used
- **ids** (*list*) – a list of ids to use to index the spatial weights object. Order is not respected from this list.
- **id_order** (*list*) – an ordered list of ids to use to index the spatial weights object. If used, the resulting weights object will iterate over results in the order of the names provided in this argument.

See also:

`pysal.weights.W`, `pysal.weights.Rook`

classmethod from_iterable(*iterable*, ***kwargs*)

Construct a weights object from a collection of arbitrary polygons. This will cast the polygons to PySAL polygons, then build the W.

Parameters

- **iterable** (*iterable*) – a collection of of shapes to be cast to PySAL shapes. Must support iteration. Contents should at least implement a *__geo_interface__* attribute or be able to be coerced to geometries using pysal.cg.asShape
- ****kw** (*keyword arguments*) – optional arguments for `pysal.weights.W`

See also:

`pysal.weights.W`, `pysal.weights.Rook`

classmethod from_shapefile(*filepath*, *idVariable=None*, *full=False*, ***kwargs*)

Rook contiguity weights from a polygon shapefile.

Parameters

- **shapefile** (*string*) – name of polygon shapefile including suffix.

- **sparse** (*boolean*) – If True return WSP instance If False return W instance

Returns **w** – instance of spatial weights

Return type *W*

Examples

```
>>> wr=rook_from_shapefile(pysal.examples.get_path("columbus.shp"), "POLYID")
>>> "%.3f"%wr.pct_nonzero
'8.330'
>>> wr=rook_from_shapefile(pysal.examples.get_path("columbus.shp"),␣
↪sparse=True)
>>> pct_sp = wr.sparse.nnz *1. / wr.n**2
>>> "%.3f"%pct_sp
'0.083'
```

Notes

Rook contiguity defines as neighbors any pair of polygons that share a common edge in their polygon definitions.

See also:

`pysal.weights.W`, `pysal.weights.Rook`

full()

Generate a full numpy array.

Returns **implicit** – first element being the full numpy array and second element keys being the ids associated with each row in the array.

Return type tuple

Examples

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

See also:

full

get_transform()

Getter for transform property.

Returns **transformation**

Return type string (or none)

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.weights[0]
[1.0, 1.0]
>>> w.transform
'O'
>>> w.transform='r'
>>> w.weights[0]
[0.5, 0.5]
>>> w.transform='b'
>>> w.weights[0]
[1.0, 1.0]
>>>
```

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.

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.id_order_set
True
```

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.

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

Returns offsets of the id's neighbors in id_order

Return type list

Examples

```
>>> from pysal import W
>>> neighbors={'c': ['b'], 'b': ['c', 'a'], 'a': ['b']}
>>> weights ={'c': [1.0], 'b': [1.0, 1.0], 'a': [1.0]}
>>> w=W(neighbors,weights)
>>> w.id_order = ['a','b','c']
>>> w.neighbor_offsets['b']
[2, 0]
>>> w.id_order = ['b','a','c']
>>> w.neighbor_offsets['b']
[2, 1]
```

nonzero
Number of nonzero weights.

pct_nonzero
Percentage of nonzero weights.

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

...

Parameters new_ids (*list*) – /ndarray Aligned list of new ids to be inserted. Note that first element of new_ids will replace first element of w.id_order, second element of new_ids replaces second element of w.id_order and so on.

Example

```
>>> import pysal as ps
>>> w = ps.lat2W(3, 3)
>>> w.id_order
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> w.neighbors[0]
[3, 1]
>>> new_ids = ['id%i'%id for id in w.id_order]
>>> _ = w.remap_ids(new_ids)
>>> w.id_order
['id0', 'id1', 'id2', 'id3', 'id4', 'id5', 'id6', 'id7', 'id8']
>>> w.neighbors['id0']
['id3', 'id1']
```

s0
s0 is defined as

$$s0 = \sum_i \sum_j w_{i,j}$$

s1
s1 is defined as

$$s1 = 1/2 \sum_i \sum_j (w_{i,j} + w_{j,i})^2$$

s2

s2 is defined as

$$s2 = \sum_j (\sum_i w_{i,j} + \sum_i w_{j,i})^2$$

s2array

Individual elements comprising s2.

See also:

s2

sd

Standard deviation of number of neighbors.

set_shapefile (*shapefile*, *idVariable=None*, *full=False*)

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

Parameters

- **shapefile** (*string*) – shapefile name used to construct weights
- **idVariable** (*string*) – name of attribute in shapefile to associate with ids in the weights
- **full** (*boolean*) – True - write out entire path for shapefile, False (default) only base of shapefile without extension

set_transform (*value='B'*)

Transformations of weights.

Notes

Transformations are applied only to the value of the weights at instantiation. Chaining of transformations cannot be done on a W instance.

Parameters transform (*string*) – not case sensitive)

:param .. table::: :widths: auto

transform string	value
B	Binary
R	Row-standardization (global sum=n)
D	Double-standardization (global sum=1)
V	Variance stabilizing
O	Restore original transformation (from instantiation)

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.weights[0]
[1.0, 1.0]
>>> w.transform
'O'
>>> w.transform='r'
>>> w.weights[0]
```

```
[0.5, 0.5]
>>> w.transform='b'
>>> w.weights[0]
[1.0, 1.0]
>>>
```

sparse
Sparse matrix object.

For any matrix manipulations required for w, w.sparse should be used. This is based on scipy.sparse.

to_WSP()
Generate a WSP object.

Returns implicit – Thin W class

Return type pysal.WSP

Examples

```
>>> import pysal as ps
>>> from pysal import W
>>> neighbors={'first':['second'],'second':['first','third'],'third':['second
↪']}
>>> weights={'first':[1],'second':[1,1],'third':[1]}
>>> w=W(neighbors,weights)
>>> wsp=w.towsp()
>>> isinstance(wsp, ps.weights.weights.WSP)
True
>>> wsp.n
3
>>> wsp.s0
4
```

See also:

WSP

towsp()
Generate a WSP object.

Returns implicit – Thin W class

Return type pysal.WSP

Examples

```
>>> import pysal as ps
>>> from pysal import W
>>> neighbors={'first':['second'],'second':['first','third'],'third':['second
↪']}
>>> weights={'first':[1],'second':[1,1],'third':[1]}
>>> w=W(neighbors,weights)
>>> wsp=w.towsp()
>>> isinstance(wsp, ps.weights.weights.WSP)
True
>>> wsp.n
```

```
3
>>> wsp.s0
4
```

See also:

`WSP`

transform

Getter for transform property.

Returns **transformation**

Return type string (or none)

Examples

```
>>> from pysal import lat2W
>>> w=lat2W()
>>> w.weights[0]
[1.0, 1.0]
>>> w.transform
'O'
>>> w.transform='r'
>>> w.weights[0]
[0.5, 0.5]
>>> w.transform='b'
>>> w.weights[0]
[1.0, 1.0]
>>>
```

trcW2

Trace of WW.

See also:

diagW2

trcWtW

Trace of $W'W$.

See also:

diagWtW

trcWtW_WW

Trace of $W'W + WW$.

weights.Distance — Distance based spatial weights

The `weights.Distance` module provides for spatial weights defined on distance relationships.

New in version 1.0.

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

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

Return type *W*

Examples

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

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

```
>>> wnn2 = 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
```

Notes

Ties between neighbors of equal distance are arbitrarily broken.

See also:

`pysal.weights.W`

classmethod from_array (*array*, ***kwargs*)

Creates nearest neighbor weights matrix based on k nearest neighbors.

Parameters

- **array** (*np.ndarray*) – (n, k) array representing n observations on k characteristics used to measure distances between the n objects
- ****kwargs** (*keyword arguments, see Rook*) –

Returns w – instance Weights object with binary weights

Return type *W*

Examples

```
>>> points = [(10, 10), (20, 10), (40, 10), (15, 20), (30, 20), (30, 30)]
>>> wnn2 = pysal.KNN.from_array(points, 2)
>>> [1,3] == wnn2.neighbors[0]
True
```

ids

```
>>> wnn2 = KNN.from_array(points,2)
>>> wnn2[0]
{1: 1.0, 3: 1.0}
>>> wnn2[1]
{0: 1.0, 3: 1.0}
```

now with 1 rather than 0 offset

```
>>> wnn2 = KNN.from_array(points, 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
```

Notes

Ties between neighbors of equal distance are arbitrarily broken.

See also:

class *pysal.weights.KNN*

`pysal.weights.W`

classmethod from_dataframe(*df*, *geom_col='geometry'*, *ids=None*, ***kwargs*)
Make KNN weights from a dataframe.

Parameters

- **df** (*pandas.dataframe*) – a dataframe with a geometry column that can be used to construct a W object
- **geom_col** (*string*) – column name of the geometry stored in df
- **ids** (*string or iterable*) – if string, the column name of the indices from the dataframe if iterable, a list of ids to use for the W if None, df.index is used.

See also:

class *pysal.weights.KNN*

`pysal.weights.W`

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

Parameters

- **`data`** (*`string`*) – shapefile containing attribute data.
- **`k`** (*`int`*) – number of nearest neighbors
- **`p`** (*`float`*) – Minkowski p-norm distance metric parameter: 1<=p<=infinity 2: Euclidean distance 1: Manhattan distance
- **`ids`** (*`list`*) – identifiers to attach to each observation
- **`radius`** (*`float`*) – If supplied arc_distances will be calculated based on the given radius. p will be ignored.

Returns **w** – instance; Weights object with binary weights.

Return type *KNN*

Examples

Polygon shapefile

```
>>> wc=knnW_from_shapefile(pysal.examples.get_path("columbus.shp"))
>>> "%.4f"%wc.pct_nonzero
'4.0816'
>>> set([2,1]) == set(wc.neighbors[0])
True
>>> wc3=pysal.knnW_from_shapefile(pysal.examples.get_path("columbus.shp"),k=3)
>>> set(wc3.neighbors[0]) == set([2,1,3])
True
>>> set(wc3.neighbors[2]) == set([4,3,0])
True
```

1 offset rather than 0 offset

```
>>> wc3_1=knnW_from_shapefile(pysal.examples.get_path("columbus.shp"),k=3,
↪idVariable="POLYID")
>>> set([4,3,2]) == set(wc3_1.neighbors[1])
True
>>> wc3_1.weights[2]
[1.0, 1.0, 1.0]
>>> set([4,1,8]) == set(wc3_1.neighbors[2])
True
```

Point shapefile

```
>>> w=knnW_from_shapefile(pysal.examples.get_path("juvenile.shp"))
>>> w.pct_nonzero
1.1904761904761905
>>> w1=knnW_from_shapefile(pysal.examples.get_path("juvenile.shp"),k=1)
>>> "%.3f"%w1.pct_nonzero
```

Notes

Ties between neighbors of equal distance are arbitrarily broken.

See also:

`pysal.weights.KNN`, `pysal.weights.W`

reweight(*k=None*, *p=None*, *new_data=None*, *new_ids=None*, *inplace=True*)

Redo K-Nearest Neighbor weights construction using given parameters

Parameters

- **new_data** (*np.ndarray*) – an array containing additional data to use in the KNN weight
- **new_ids** (*list*) – a list aligned with new_data that provides the ids for each new observation
- **inplace** (*bool*) – a flag denoting whether to modify the KNN object in place or to return a new KNN object
- **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

Returns

- *A copy of the object using the new parameterization, or None if the*
- *object is reweighted in place.*

class `pysal.weights.Distance.`**Kernel**(*data*, *bandwidth=None*, *fixed=True*, *k=2*, *function='triangular'*, *eps=1.0000001*, *ids=None*, *diagonal=False*)

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, $h_i = max(dknn) \forall i$ 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

  $$z_{i,j} = d_{i,j}/h_i$$

  triangular

  $$K(z) = (1 - |z|) \; if|z| \le 1$$

  uniform

  $$K(z) = 1/2 \; if|z| \le 1$$

quadratic

$$K(z) = (3/4)(1 - z^2)\ if|z| \leq 1$$

quartic

$$K(z) = (15/16)(1 - z^2)^2\ if|z| \leq 1$$

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

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

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],
       [ 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.],
       [ 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)
>>> kwea.weights[0]
[1.0, 0.10557289844279438, 9.99999900663795e-08]
>>> kwea.neighbors[0]
[0, 1, 3]
>>> kwea.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, 0.2419707487162134]
>>> kweag.bandwidth
array([[ 11.18034101],
       [ 11.18034101],
       [ 20.000002  ],
       [ 11.18034101],
       [ 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, 0.2419707487162134, 0.3412334260702758,␣
↪0.31069657591175387], 2: [0.2419707487162134, 0.3989422804014327, 0.
↪31069657591175387], 3: [0.3412334260702758, 0.3412334260702758, 0.
↪3989422804014327, 0.3011374490937829, 0.26575287272131043], 4: [0.
↪31069657591175387, 0.31069657591175387, 0.3011374490937829, 0.3989422804014327,␣
↪0.35206533556593145], 5: [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,␣
↪0.2419707487162134, 0.3412334260702758, 0.31069657591175387], 2: [0.
↪2419707487162134, 1.0, 0.31069657591175387], 3: [0.3412334260702758, 0.
↪3412334260702758, 1.0, 0.3011374490937829, 0.26575287272131043], 4: [0.
↪31069657591175387, 0.31069657591175387, 0.3011374490937829, 1.0, 0.
↪35206533556593145], 5: [0.26575287272131043, 0.35206533556593145, 1.0]}
```

classmethod **`from_array`**(*array*, ***kwargs*)

Construct a Kernel weights from an array. Supports all the same options as `pysal.weights.Kernel`

See also:

`pysal.weights.Kernel`, `pysal.weights.W`

classmethod **`from_dataframe`**(*df*, *geom_col='geometry'*, *ids=None*, ***kwargs*)

Make Kernel weights from a dataframe.

Parameters

- **`df`** (*`pandas.dataframe`*) – a dataframe with a geometry column that can be used to construct a W object
- **`geom_col`** (*`string`*) – column name of the geometry stored in df
- **`ids`** (*`string or iterable`*) – if string, the column name of the indices from the dataframe if iterable, a list of ids to use for the W if None, df.index is used.

See also:

`pysal.weights.Kernel`, `pysal.weights.W`

classmethod **`from_shapefile`**(*filepath*, *idVariable=None*, ***kwargs*)

Kernel based weights from shapefile

Parameters

- **`shapefile`** (*`string`*) – shapefile name with shp suffix
- **`idVariable`** (*`string`*) – name of column in shapefile's DBF to use for ids

Returns

Return type Kernel Weights Object

See also:

`pysal.weights.Kernel`, `pysal.weights.W`

class `pysal.weights.Distance.`**`DistanceBand`**(*data*, *threshold*, *p=2*, *alpha=-1.0*, *binary=True*, *ids=None*, *build_sp=True*, *silent=False*)

Spatial weights based on distance band.

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
- **`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 dending on the sparsity of the of distance_matrix and threshold that is applied

- **silent** (*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.

weights
dict – of neighbor weights keyed by observation id

neighbors
dict – of neighbors keyed by observation id

Examples

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

inverse distance weights

```
>>> w=DistanceBand(points,threshold=11.2,binary=False)
WARNING: there is one disconnected observation (no neighbors)
Island id:  [2]
>>> w.weights[0]
[0.10000000000000001, 0.089442719099991588]
>>> w.neighbors[0]
[1, 3]
>>>
```

gravity weights

```
>>> w=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, 0.0079999999999999984]
```

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.

classmethod from_array (*array*, *threshold*, ***kwargs*)
Construct a DistanceBand weights from an array. Supports all the same options as `pysal.weights.DistanceBand`

See also:

`pysal.weights.DistanceBand`, `pysal.weights.W`

classmethod **`from_dataframe`** (*df*, *threshold*, *geom_col='geometry'*, *ids=None*, ***kwargs*)
Make DistanceBand weights from a dataframe.

Parameters

- **`df`** (*`pandas.dataframe`*) – a dataframe with a geometry column that can be used to construct a W object
- **`geom_col`** (*`string`*) – column name of the geometry stored in df
- **`ids`** (*`string or iterable`*) – if string, the column name of the indices from the dataframe if iterable, a list of ids to use for the W if None, df.index is used.

See also:

`pysal.weights.DistanceBand`, `pysal.weights.W`

classmethod **`from_shapefile`** (*filepath*, *threshold*, *idVariable=None*, ***kwargs*)
Distance-band based weights from shapefile

Parameters

- **`shapefile`** (*`string`*) – shapefile name with shp suffix
- **`idVariable`** (*`string`*) – name of column in shapefile's DBF to use for ids

Returns

Return type Kernel Weights Object

See also:

class *pysal.weights.DistanceBand*

class *pysal.weights.W*

`weights.Wsets` — Set operations on spatial weights

The `weights.user` module provides for set operations on weights objects .. versionadded:: 1.0 Set-like manipulation of weights matrices.

`pysal.weights.Wsets.`**`w_union`**(*w1*, *w2*, *silent_island_warning=False*)
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
- **`silent_island_warning`** (*`boolean`*) – Switch to turn off (default on) print statements for every observation with islands

Returns **w** – object

Return type *W*

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.

```
>>> import pysal
>>> w1 = pysal.lat2W(4,4)
>>> w2 = pysal.lat2W(6,4)
>>> w = pysal.weights.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.weights.Wsets.`**`w_intersection`**(*w1*, *w2*, *w_shape='w1'*, *silent_island_warning=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.
- **silent_island_warning** (*boolean*) – Switch to turn off (default on) print statements for every observation with islands

Returns w – object

Return type *W*

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.

```
>>> import pysal
>>> w1 = pysal.lat2W(4,4)
>>> w2 = pysal.lat2W(6,4)
>>> w = pysal.weights.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]
>>>
```

pysal.weights.Wsets.**w_difference**(*w1*, *w2*, *w_shape='w1'*, *constrained=True*, *silent_island_warning=False*)

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

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 the IDs occurring in w1 and not in w2.
- **constrained** (*boolean*) – If False then the full set of neighbor pairs in w1 that are not in 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'.
- **silent_island_warning** (*boolean*) – Switch to turn off (default on) print statements for every observation with islands

Returns **w** – object

Return type *W*

Notes

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

Examples

Construct rook (w2) and queen (w1) weights matrices for two 4x4 regions (16 areas). A queen matrix has all the joins a rook matrix does plus joins between areas that share a corner. The new matrix formed by the difference of rook from queen contains only join at corners (typically called a bishop matrix). Note that the difference of queen from rook would result in a weights matrix with no joins.

```
>>> import pysal
>>> w1 = pysal.lat2W(4,4,rook=False)
>>> w2 = pysal.lat2W(4,4,rook=True)
>>> w = pysal.weights.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.weights.Wsets.`**`w_symmetric_difference`**(*w1*, *w2*, *w_shape='all'*, *constrained=True*, *silent_island_warning=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'.
- **`silent_island_warning`** (*`boolean`*) – Switch to turn off (default on) print statements for every observation with islands

Returns **w** – object

Return type *W*

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.

```
>>> import pysal
>>> w1 = pysal.lat2W(4,4,rook=False)
>>> w2 = pysal.lat2W(6,4,rook=True)
>>> w = pysal.weights.w_symmetric_difference(w1, w2, constrained=False)
>>> w1[0] == w[0]
False
>>> w1.neighbors[15]
[10, 11, 14]
>>> w2.neighbors[15]
[11, 14, 19]
>>> w.neighbors[15]
[10, 19]
>>>
```

`pysal.weights.Wsets.`**`w_subset`**(*w1*, *ids*, *silent_island_warning=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 include in the returned weights object.

- **silent_island_warning** (*boolean*) – Switch to turn off (default on) print statements for every observation with islands

Returns **w** – object

Return type *W*

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.

```
>>> import pysal
>>> w1 = pysal.lat2W(6,4)
>>> ids = range(16)
>>> w = pysal.weights.w_subset(w1, ids)
>>> w1[0] == w[0]
True
>>> w1.neighbors[15]
[11, 14, 19]
>>> w.neighbors[15]
[11, 14]
>>>
```

pysal.weights.Wsets.**w_clip**(*w1*, *w2*, *outSP=True*, *silent_island_warning=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) – pysal.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) – pysal.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 pysal.W object of the clipped matrix
- **silent_island_warning** (*boolean*) – Switch to turn off (default on) print statements for every observation with islands

Returns **wc** – pysal.W, scipy.sparse.csr.csr_matrix Clipped W object (sparse if outSP=Ture). It inherits `id_order` from w1.

Return type *W*

Examples

```
>>> import pysal as ps
```

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 neccesarily represent a row-standardization anymore):

```
>>> w1 = ps.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:

```
>>> w2 = ps.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

```
>>> 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

```
>>> wcs = ps.weights.Wsets.w_clip(w1, w2, outSP=True)
```

This will create a sparse object (recommended when n is large).

```
>>> wcs.sparse.toarray()
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.        ,  0.2       ,  0.        ,  0.2       ,
         0.        ],
       [ 0.        ,  0.        ,  0.33333333,  0.33333333,  0.        ,
         0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ]])
```

If we wanted an original W object, we can control that with the argument `outSP`:

```
>>> wc = ps.weights.Wsets.w_clip(w1, w2, outSP=False)
WARNING: there are 2 disconnected observations
Island ids:  [1, 5]
>>> 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.        ,  0.2       ,  0.        ,  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:

```
>>> 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]], dtype=bool)
```

`weights.spatial_lag` — Spatial lag operators

The `weights.spatial_lag` Spatial lag operators for PySAL

New in version 1.0. Spatial lag operations.

`pysal.weights.spatial_lag.`**`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 spatial weightsobject
- **y** (*`array`*) – numpy array with dimensionality conforming to w (see examples)

Returns **wy** – array of numeric values for the spatial lag

Return type array

Examples

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

```
>>> import pysal
>>> import numpy as np
>>> w = pysal.lat2W(3, 3)
>>> y = np.arange(9)
>>> yl = pysal.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

```
>>> w.transform = 'r'
>>> yl = pysal.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

```
>>> y.shape = (9, 1)
>>> yl = pysal.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

```
>>> yr = np.arange(8, -1, -1)
>>> yr.shape = (9, 1)
>>> x = np.hstack((y, yr))
>>> yl = pysal.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.weights.spatial_lag.`**`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 weightsobject
- **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

Return type 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.

```
>>> import pysal
>>> import numpy as np
>>> np.random.seed(12345)
>>> w = pysal.lat2W(3, 3)
>>> y = ['a','b','a','b','c','b','c','b','c']
>>> y_l = pysal.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

```
>>> yvect = np.array(y).reshape(9,1)
>>> yvect_l = pysal.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

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

pysal.network — Network Constrained Analysis

pysal.network — Network Constrained Analysis

The `network` Network Analysis for PySAL

New in version 1.9.

class `pysal.network.network.`**`Network`**(*in_shp=None*, *node_sig=11*, *unique_segs=True*)

Spatially constrained network representation and analytical functionality.

Parameters

- **in_shp** (*str*) – The input shapefile. This must be in .shp format.
- **node_sig** (*int*) – Round the x and y coordinates of all nodes to node_sig significant digits (combined significant digits on the left and right of the decimal place) – Default is 11 – Set to None for no rounding
- **unique_segs** (*bool*) – If True (default), keep only unique segments (i.e., prune out any duplicated segments). If False keep all segments.

in_shp

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. Values are tuples with two elements:

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.

Examples

Instantiate an instance of a network.

```
>>> ntw = ps.Network(ps.examples.get_path('streets.shp'))
```

Snap point observations to the network with attribute information.

```
>>> ntw.snapobservations(ps.examples.get_path('crimes.shp'), 'crimes',␣
↪attribute=True)
```

And without attribute information.

```
>>> ntw.snapobservations(ps.examples.get_path('schools.shp'), 'schools',␣
↪attribute=False)
```

NetworkF (*pointpattern*, *nsteps=10*, *permutations=99*, *threshold=0.2*, *distribution='uniform'*, *lowerbound=None*, *upperbound=None*)
Computes a network constrained F-Function

Parameters

- **pointpattern** (*object*) – A PySAL 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 **NetworkF** – A network F class instance.

Return type object

NetworkG (*pointpattern*, *nsteps=10*, *permutations=99*, *threshold=0.5*, *distribution='uniform'*, *lowerbound=None*, *upperbound=None*)
Computes a network constrained G-Function

Parameters

- **pointpattern** (*object*) – A PySAL 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 **NetworkG** – A network G class instance.

Return type object

NetworkK (*pointpattern*, *nsteps=10*, *permutations=99*, *threshold=0.5*, *distribution='uniform'*, *lowerbound=None*, *upperbound=None*)
Computes a network constrained K-Function

Parameters

- **pointpattern** (*object*) – A PySAL 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 K-function is computed. (Default 0)
- **upperbound** (*float*) – The upper bound at which the K-function is computed. Defaults to the maximum observed nearest neighbor distance.

Returns **NetworkK** – A network K class instance.

Return type object

allneighbordistances(*sourcepattern*, *destpattern=None*, *fill_diagonal=None*, *n_processes=None*)

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*) – The key of a point pattern snapped to the network.
- **destpattern** (*str*) – (Optional) The key of a point pattern snapped to the network.
- **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.

Returns **nearest** – An array of shape (n,n) storing distances between all points.

Return type array (*n*,*n*)

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-coordiante 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

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.

Return type object

Examples

```
>>> ntw = ps.Network(ps.examples.get_path('streets.shp'))
>>> w = ntw.contiguityweights(graph=False)
>>> ntw.snapobservations(ps.examples.get_path('crimes.shp'), 'crimes',␣
↪attribute=True)
>>> counts = ntw.count_per_edge(ntw.pointpatterns['crimes'].obs_to_edge,␣
↪graph=False)
```

Using the W object, access to ESDA functionality is provided. First, a vector of attributes is created for all edges with observations.

```
>>> 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]
```

Next, a standard call ot Moran is made and the result placed into *res*

```
>>> res = ps.esda.moran.Moran(y, w, permutations=99)
```

count_per_edge(*obs_on_network*, *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 – {(edge):count}

Return type dict

Example

Note that this passes the obs_to_edge attribute of a point pattern snapped to the network.

```
>>> ntw = ps.Network(ps.examples.get_path('streets.shp'))
>>> ntw.snapobservations(ps.examples.get_path('crimes.shp'), 'crimes',␣
↪attribute=True)
>>> counts = ntw.count_per_edge(ntw.pointpatterns['crimes'].obs_to_edge,
↪graph=False)
>>> s = sum([v for v in counts.itervalues()])
>>> s
287
```

distancebandweights(*threshold*, *n_proccess=None*)
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.

enum_links_node(*v0*)
Returns the edges (links) around node

Parameters **v0** (*int*) – Node id

Returns **links** – List of tuple edges adjacent to the node.

Return type list

extractgraph()
Using the existing network representation, create a graph based representation by removing all nodes with a neighbor incidence of two. That is, we assume these nodes are bridges between nodes with higher incidence.

nearestneighbordistances(*sourcepattern*, *destpattern=None*, *n_processes=None*)
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.

Returns **nearest** – With column[:,0] containing the id of the nearest neighbor and column [:,1] containing the distance.

Return type ndarray (n,2)

node_distance_matrix(*n_processes*)

Called from: allneighbordistances() nearestneighbordistances() distancebandweights()

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 the file is saved whereever this method is called from.

Example

```
>>> ntw = ps.Network(ps.examples.get_path('streets.shp'))
>>> ntw.savenetwork('mynetwork.pkl')
```

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 **sn** – PySAL Network Object.

Return type object

Example

```
>>> ntw = ps.Network(ps.examples.get_path('streets.shp'))
>>> n200 = ntw.segment_edges(200.0)
>>> len(n200.edges)
688
```

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.

Returns **random_pts** – Keys are the edge tuple. Value are a list of new point coordinates.

Return type dict

Example

```
>>> ntw = ps.Network(ps.examples.get_path('streets.shp'))
>>> ntw.snapobservations(ps.examples.get_path('crimes.shp'), 'crimes',␣
↪attribute=True)
>>> npts = ntw.pointpatterns['crimes'].npoints
>>> sim = ntw.simulate_observations(npts)
>>> isinstance(sim, ps.network.network.SimulatedPointPattern)
True
```

snapobservations(*shapefile*, *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

- **shapefile** (*str*) – The path to the shapefile.
- **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 extraaction.

class pysal.network.network.**PointPattern**(*shapefile*, *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

- **shapefile** (*str*) – The input shapefile.
- **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.

points
dict – Keys are the point ids. Values are the coordinates.

npoints
int – The number of points.

class `pysal.network.network.`**NetworkG**(*ntw*, *pointpattern*, *nsteps=10*, *permutations=99*, *threshold=0.5*, *distribution='poisson'*, *lowerbound=None*, *upperbound=None*)
Compute a network constrained G statistic.

class `pysal.network.network.`**NetworkK**(*ntw*, *pointpattern*, *nsteps=10*, *permutations=99*, *threshold=0.5*, *distribution='poisson'*, *lowerbound=None*, *upperbound=None*)
Compute a network constrained K statistic.

class `pysal.network.network.`**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

`pysal.contrib` – Contributed Modules

Intro

The PySAL Contrib library contains user contributions that enhance PySAL, but are not fit for inclusion in the general library. The primary reason a contribution would not be allowed in the general library is external dependencies. PySAL has a strict no dependency policy (aside from Numpy/Scipy). This helps ensure the library is easy to install and maintain.

However, this policy often limits our ability to make use of existing code or exploit performance enhancements from C-extensions. This contrib module is designed to alleviate this problem. There are no restrictions on external dependencies in contrib.

Ground Rules

1. Contribs must not be used within the general library.
2. *Explicit imports*: each contrib must be imported manually.
3. *Documentation*: each contrib must be documented, dependencies especially.

Contribs

Currently the following contribs are available:

1. World To View Transform – A class for modeling viewing windows, used by Weights Viewer.
   - New in version 1.3.
   - Path: pysal.contrib.weights_viewer.transforms
   - Requires: None
2. Weights Viewer – A Graphical tool for examining spatial weights.
   - New in version 1.3.

   - Path: pysal.contrib.weights_viewer.weights_viewer
   - Requires: wxPython

3. Shapely Extension – Exposes shapely methods as standalone functions
   - New in version 1.3.
   - Path: pysal.contrib.shapely_ext
   - Requires: shapely
4. Shared Perimeter Weights – calculate shared perimeters weights.
   - New in version 1.3.
   - Path: pysal.contrib.shared_perimeter_weights
   - Requires: shapely
5. Visualization – Lightweight visualization layer (Project page).
   - New in version 1.5.
   - Path: pysal.contrib.viz
   - Requires: matplotlib
6. Clusterpy – Spatially constrained clustering.
   - New in version 1.8.
   - Path: pysal.contrib.clusterpy
   - Requires: clusterpy
7. Pandas utilities – Tools to work with spatial file formats using *pandas*.
   - New in version 1.8.
   - Path: pysal.contrib.pdutilities
   - Requires: **‘pandas‘_**
8. Spatial Interaction – Tools for spatial interaction (SpInt) modeling.

- New in version 1.10.
- Path: pysal.contrib.spint
- Requires: **‘pandas‘_**

9. Githooks – Optional hooks for *git* to make development on *PySAL* easier
   - New in version 1.10.
   - Path: pysal.contrib.githooks (Note: not importable)
   - Requires: *git*
10. Handler – A model ingester to standardize model extension

- New in version 1.10.
- Path: pysal.contrib.handler
- Requires: None
- Optional: patsy

Bibliography

[Anselin2000] Anselin, Luc (2000) Computing environments for spatial data analysis. *Journal of Geographical Systems* 2: 201-220

[ReyJanikas2006] Rey, S.J. and M.V. Janikas (2006) STARS: Space-Time Analysis of Regional Systems, *Geographical Analysis* 38: 67-86.

[ReyYe2010] Rey, S.J. and X. Ye (2010) Comparative spatial dyanmics of regional systems. In Paez, A. et al. (eds) *Progress in Spatial Analysis: Methods and Applications*. Springer: Berlin, 441-463.

[Python271] http://www.python.org/download/releases/2.7.1/

[PythonNewIn3] http://docs.python.org/release/3.0.1/whatsnew/3.0.html

[Python2to3] http://docs.python.org/release/3.0.1/library/2to3.html#to3-reference

[NumpyANN150] http://mail.scipy.org/pipermail/numpy-discussion/2010-August/052522.html

[SciPyRoadmap] http://projects.scipy.org/scipy/roadmap#python-3

[SciPyANN090rc2] http://mail.scipy.org/pipermail/scipy-dev/2011-January/015927.html

[Rtree] http://pypi.python.org/pypi/Rtree/

[pyrtree] http://code.google.com/p/pyrtree/

Python Module Index

Index