1 An introduction to network tools in Python

Python has been a very popular choice for scientific computing. I could say many things about it but I feel the following XKCD comic pretty much sums it up.

\begin{Verbatim}
In [1]: from IPython.display import Image
   \indent Image('http://imgs.xkcd.com/comics/python.png')
\end{Verbatim}

*Email: vsvijayaraghavan@ucdavis.edu*
1.1 Useful modules (or libraries)

Numpy and Scipy are useful for standard scientific computing. Networkx and iGraph are more specialized for networks. Matplotlib is a great plotting library. For this introduction we will not be requiring functions from Scipy or iGraph. The links to all these modules are can be found in the list of resources.

In [2]: import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from __future__ import division
1.2 Basic graph operations in Networkx

In [3]: g = nx.erdos_renyi_graph(10, 0.5)  #create an ER random graph with 10 nodes and probability of connection = 0.5
In [4]: g.add_node(12)  #add a node
In [5]: g.add_edge(12, "A")  #add a new node "A" and connect it to node 12
In [6]: nx.connected_components(g)  #list the connected components of the graph
Out[6]: [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], ['A', 12]]
In [7]: g.degree()  #return a dictionary of nodes and degree
Out[7]: {0: 5, 1: 5, 2: 4, 3: 7, 4: 3, 5: 4, 6: 5, 7: 5, 8: 7, 9: 3, 12: 1, 'A': 1}
In [8]: nx.clustering(g)  #list the clustering coefficient for each node
Out[8]: {0: 0.9, 1: 0.6, 2: 0.5, 3: 0.42857142857142855, 4: 0.3333333333333333, 5: 0.6666666666666666, 6: 0.9, 7: 0.9, 8: 0.5238095238095238, 9: 0.6666666666666666, 12: 0.0, 'A': 0.0}
In [9]: nx.betweenness_centrality(g)  #compute betweenness-centrality for each node
Out[9]: {0: 0.0036363636363636364, 1: 0.03636363636363636, 2: 0.033333333333333326, 3: 0.13333333333333333, 4: 0.02181818181818182, 5: 0.01212121212121212, 6: 0.0036363636363636364, 7: 0.0036363636363636364, 8: 0.1278787878787879, 9: 0.00606060606060606, 12: 0.0, 'A': 0.0}

1.3 Random walk on a graph

We will compute the steady state distribution for a random walk on a given network. We will also learn how to visualize a simple network as part of this exercise.

In [10]: g = nx.DiGraph()  #define g to be a directed graph
In [11]: g.add_edges_from([(1, 3), (2, 1), (2, 4), (3, 2), (3, 4), (4, 3), (5, 1), (5, 3)])  #let's create a simple directed graph
In [12]: nx.draw(g)  #visualize the graph. The thicker stubs are used in place of arrows.
In [13]: M = nx.adjacency_matrix(g)  # obtain the adj. matrix for the graph

In [14]: print(M)

[[ 0.  0.  1.  0.  0. ]
 [ 1.  0.  0.  1.  0. ]
 [ 0.  1.  0.  1.  0. ]
 [ 0.  0.  1.  0.  0. ]
 [ 1.  0.  1.  0.  0. ]]

In [15]: for i in range(5):
   if (np.sum(M[i]) > 0):
       M[i] = M[i]/np.sum(M[i])

In [16]: print M # normalized matrix

[[ 0.  0.  1.  0.  0. ]
 [ 0.5 0.  0.  0.5 0. ]
 [ 0.  0.5 0.  0.5 0. ]
 [ 0.  0.  1.  0.  0. ]
 [ 0.5 0.  0.5 0.  0. ]]

1.3.1 Numerical approach

If we denote the probability of finding a random walker on each node as a column vector, the transition matrix T is given by:

In [17]: T = np.matrix(M).transpose()
In order to find the stationary distribution start with a state vector with equal probability on being on any of the nodes and repeatedly apply the transition matrix. In the limit of number of times going to infinity the resulting state vector gives us the stationary distribution. For small graphs, the number steps required is quite small.

```
In [18]: state_vec = np.matrix((1/5)*np.ones(5)).transpose()
    #start with equal probability of finding a random walker on any node

In [19]: for i in range(15): # here 15 is the number of times we apply T on the state vector.
    #choosing a small number to illustrate a point.
    state_vec = T*state_vec
    print state_vec

[[ 0.09960938]
 [ 0.2      ]
 [ 0.40078125]
 [ 0.29960938]
 [ 0.      ]]

In [20]: #check if state_vec has converged
    print T*state_vec

[[ 0.1      ]
 [ 0.20039062]
 [ 0.39921875]
 [ 0.30039063]
 [ 0.      ]]

In [21]: # since the vector obtained above is not the same as the state_vec,
    # apply T on state_vec a few more times
    for i in range(35):
        state_vec = T*state_vec
        print state_vec

[[ 0.1]
 [ 0.2]
 [ 0.4]
 [ 0.3]
 [ 0. ]]

In [22]: # check if state_vec has converged.
    print T*state_vec

[[ 0.1]
 [ 0.2]
 [ 0.4]
 [ 0.3]
 [ 0. ]]
```

Since the vector has now converged this is the stationary distribution. Note that the vector obtained after 15 iterations was still a good approximation due to the small number of nodes present in this network.

### 1.3.2 Analytical approach

We know that the steady state distribution is related to the spectrum of the normalized adjacency matrix. In this part we illustrate this approach.
In [23]: import numpy.linalg as la  
   #import linear algebra module

In [24]: T = np.matrix(M.transpose())
   print T

[[ 0.  0.5  0.  0.  0.5]
 [ 0.  0.5  0.  0.  0. ]
 [ 1.  0.  0.  1.  0.5]
 [ 0.  0.5  0.5 0.  0. ]
 [ 0.  0.  0.  0.  0. ]]

In [25]: eigvals, eigvec = la.eig(T)

In [26]: print eigvals
   #list the eigenvalues. The eigenvector corresponding to the eigenvalue of 1 is the one that gives us the steady state

[ 1.00000000e+00+0.j -5.00000000e-01+0.5j -5.00000000e-01-0.5j
  2.28576394e-17+0.j  0.00000000e+00+0.j ]

In [27]: print eigvec[:,0]
   #eigenvector corresponding to eigenvalue of 1

[[-0.18257419+0.j]
 [-0.36514837+0.j]
 [-0.73029674+0.j]
 [-0.54772256+0.j]
 [ 0.00000000+0.j]]

In [28]: steady_state = np.real(eigvec[:,0]/np.sum(eigvec[:,0]))
   #normalize to get steady state. Note that the imaginary part is zero, which it should be if everything was done correctly.
   print steady_state

[[ 0.1]
 [ 0.2]
 [ 0.4]
 [ 0.3]
 [-0. ]]

1.4 Preferential attachment and degree distribution

In this section we explore the degree distribution of networks generated using preferential attachment.

In [29]: for N in [100000, 10000, 1000, 100]:
   g = nx.barabasi_albert_graph(N, 2)  #generate a Barabasi-Albert graph with N nodes with each incomming node having 2 stubs
   k = np.sort(g.degree().values())  #get the degree of all nodes in the graph and sort them
   y = 1 - np.arange(1., N+1)/N  #probability in terms of a rank order
   plt.loglog(k,y,'.',label='N')  #plot the CCDF. This makes the tail easy to visualize
   plt.xlabel('k')
   plt.ylabel('CCDF')
   plt.grid()  #switch on grid
   plt.legend(loc=0)  #show the legend and use the best location for it

Out[29]: <matplotlib.legend.Legend at 0x2ec5190>
The figure above shows how the degree distribution appears to be more heavy tailed as we increase the system size. Recall that the largest degree that can be present in any network is limited by the total number of nodes present. It is important to test and account for finite size effects when we study a system based on simulations.

1.5 Small World Networks

In [30]: g = nx.watts_strogatz_graph(20, 4, 0.4)  # create a Watts-Strogatz network
   # with N = 20 nodes, m = 4 initial neighbors and a rewiring probability of p = 0.4

In [32]: nx.draw_circular(g)  # visualize with a circular layout
Let us now try to explore what happens to the clustering coefficient and the diameter of the network as a function of the rewiring probability. The following code might take several minutes to run.

In [33]: N = 1000 # Number of nodes in the network
clustering_array = [] # array to store the clustering coeff.
diameter_array = [] # array to store the diameter
g = nx.watts_strogatz_graph(N,4,0.) # initial network with p = 0
c0 = np.mean(nx.clustering(g).values()) # used for scaling
l0 = nx.diameter(g) # used for scaling
prange = np.logspace(-3,0,50) # create 50 equally spaced points between 0.001 and 1
for p in prange:
    g = nx.watts_strogatz_graph(N,4,p)
c = np.mean(nx.clustering(g).values())
    clustering_array.append(c)
    l = nx.diameter(g)
    diameter_array.append(l)

In [36]: plt.semilogx(prange,np.array(clustering_array)/c0,'-',label='C(p)/C(0)')
plt.semilogx(prange,np.array(diameter_array)/l0,'-',label='l(p)/l(0)')
plt.xlabel('p')
plt.legend(loc=0)

Out[36]: <matplotlib.legend.Legend at 0xd8d9650>
Note that the above plot is noisy since we are collecting the metrics from just a single realization. You could write an inner loop to create many networks with the same rewiring probability and average the metrics over those realizations to get a much smoother cure.

1.6 Links to the software packages mentioned

Here are the links to all the modules mentioned in this notebook. The links below will take you to homepage of the modules. It should be easy to find tutorials and installation instructions there.

- Python: https://www.python.org/
- IPython: https://www.python.org/
- Numpy: http://www.numpy.org/
- Scipy: http://www.scipy.org/
- Networkx: http://networkx.github.io/
- Matplotlib: http://matplotlib.org/
- iGraph: http://igraph.org/

In []: