Basic Analysis on Venue Networks
Today’s Outline

1. Getting started with Python and NetworkX
2. Basic network analysis.
3. Writing your own code.
In this course

We will learn how to analyze networks and geographically annotated data sourced from social media.

We will learn how to use machine learning algorithms in a practical context.

We will learn how to make cool visualizations using advanced programming and designer tools.
Week 1

place network analysis in the city

data mining and knowledge representation in the city

principles of human mobility

introduction to visualization of geographic information
Week 2

introduction to machine learning

practical machine learning (1): clustering on the spatial domain

practical machine learning (2): focus mobility prediction.

advanced visualization using processing
Let's load the Cambridge Network of Places (cam_net): it's a directed graph with integers as nodes.

```
import networkx as nx

# Load the Cambridge Network of Places
cam_net = nx.read_edgelist('cambridge_net.txt', create_using=nx.DiGraph(), nodetype=int)

# Calculate graph properties
N, K = cam_net.order(), cam_net.size()
avg_deg = float(K)/N

# Print the properties
print "Nodes: ", N
print "Edges: ", K
print "Average degree: ", avg_deg
```
Calculate in (and out) and degrees of directed graph...

```python
in_degrees = cam_net.in_degree() # dictionary node:degree
in_values = sorted(set(in_degrees.values()))
in_hist = [in_degrees.values().count(x) for x in in_values]
```

Then use matplotlib (pylab) to plot the degree distribution

```python
plt.figure()   # you need to first do ‘import pylab as plt’
plt.grid(True)
plt.plot(in_values,in_hist,'ro-') # in-degree
plt.plot(out_values,out_hist,'bv-') # out-degree
plt.legend(['In-degree','Out-degree'])
plt.xlabel('Degree')
plt.ylabel('Number of nodes')
plt.title('network of places in cambridge')
plt.xlim([0,2*10**2])
plt.savefig('./output/cam_net_degree_distribution.pdf')
plt.close()
```
Degree Distribution

network of places in cambridge

- Red circles: In-degree
- Blue triangles: Out-degree

OOPS! We can’t see much this way, can’t we?
Change scale of x,y axis by replacing
plt.plot(in_values,in_hist,'ro-') # in-degree
with
plt.loglog(in_values,in_hist,'ro-') # in-degree
Clustering Coefficient

We can get the clustering coefficient of individual nodes or of all the nodes (but the first we convert the graph to an undirected one).

cam_net_ud = cam_net.to_undirected()

# Clustering coefficient of node 0
print nx.clustering(cam_net_ud, 0)

# Clustering coefficient of all nodes (in a dictionary)
clust_coefficients = nx.clustering(cam_net_ud)

# Average clustering coefficient
ccs = nx.clustering(cam_net_ud)
avg_clust = sum(ccs.values()) / len(ccs)
print avg_clust
Node Centralities

Now, we will extract the main connected component; then we will compute node centrality measures.

cam_net_components = nx.connected_component_subgraphs(cam_net_ud)
cam_net_mc = cam_net_components[0]

# Betweenness centrality
bet_cen = nx.betweenness_centrality(cam_net_mc)

# Closeness centrality
clo_cen = nx.closeness_centrality(cam_net_mc)

# Eigenvector centrality
eig_cen = nx.eigenvector_centrality(cam_net_mc)
Most Central Nodes

First we introduce a utility method: given a dictionary and a threshold parameter, the top-K elements of the dictionary are returned according to element values.

```python
### get top keys from a python dictionary
def getTopDictionaryKeys(dictionary,number):
    topList = []
    a = dict(dictionary)
    for i in range(0,number):
        m = max(a, key=a.get)
        topList.append([m,a[m]])
        del a[m]
    return topList
```

We can then apply the method on the various centrality methods available. Below we extract the top-10 most central nodes for each case.

```python
top_bet_cen = getTopDictionaryKeys(bet_cen,10)
top_clo_cen = getTopDictionaryKeys(clo_cen,10)
top_eig_cent = getTopDictionaryKeys(eig_cen,10)
```
Interpretability Matters

Considering node ids in a graph is handy, yet each node remains an anonymous object! Nonetheless in a social network usually nodes correspond to real entities. In our case for each place in the network we have its title and geographic coordinates.

```python
### READ META DATA ###
node_data = {}
for l in open('./output/cambridge_net_titles.txt'):
    lineSplits = l.split(';;')
    node_id = int(lineSplits[0])
    place_title = lineSplits[1]
    latit = float(lineSplits[2])
    longit = float(lineSplits[3])
    node_data[node_id] = (place_title, latit, longit)

Iterate through lists of top centrality nodes and use meta data to print the titles of the respective places.

print 'Top-10 places for betweenness centrality.'
for [node_id, value] in top_bet_cen:
    title = node_data[node_id][0]
    print title

print 'Top-10 places for closeness centrality.'
for [node_id, value] in top_clo_cen:
    title = node_data[node_id][0]
    print title

print 'Top-10 places for eigenvector centrality.'
for [node_id, value] in top_eig_cent:
    title = node_data[node_id][0]
    print title
```
Most Central Nodes

Betweenness Centrality

**Top - 10**
- Cambridge Railway Station (CBG)
- Grand Arcade
- Cineworld Cambridge
- Greens
- King's College
- Cambridge Market
- Grafton Centre
- Apple Store
- Anglia Ruskin University
- Addenbrooke's Hospital

Closeness Centrality

**Top - 10**
- Cambridge Railway Station (CBG)
- Grand Arcade
- Cineworld Cambridge
- Apple Store
- Grafton Centre
- Cambridge Market
- Greens
- King's College
- Addenbrooke's Hospital
- Parker's Piece

Eigenvector Centrality

**Top - 10**
- Cambridge Railway Station (CBG)
- Cineworld Cambridge
- Grand Arcade
- King's College
- Apple Store
- Cambridge Market
- Greens
- Addenbrooke's Hospital
- Parker's Piece
- Revolution Bar (Vodka Revolutions)

The ranking for the different centrality metrics does not change much, although this may well depend on the type of network under consideration. The results meet ones intuition (if they know Cambridge), yet biases may exist (see Greens Gym for instance).
# draw the graph using information about nodes geographic positions
pos_dict = {}
for node_id, node_info in node_data.items():
    pos_dict[node_id] = (node_info[2], node_info[1])

nx.draw(cam_net, pos=pos_dict, with_labels=False, node_size=20)
plt.savefig('cam_net_graph.pdf')
plt.close()
Power-laws ...

\[ p(x) \propto x^{-\alpha} \]
They are everywhere (1) ...
They are everywhere (2) ...
They are everywhere (3) ...
‘Fitting’ the Degree Distribution

Change scale of x,y axis by replacing
plt.plot(in_values,in_hist,'ro-') # in-degree
with
plt.loglog(in_values,in_hist,'ro-') # in-degree

Check how you could fit this curve with SciPy:
www.scipy.org/Cookbook/FittingData
Fitting empirical data

Import or relevant libraries first to a new file named fit_example.py

```python
import pylab
from scipy import *
import scipy.optimize as optimize
import numpy as np
```
Fitting empirical data

1. Output the data to file.  
2. In the new file fit_example.py, read data.

```python
#output data to file
x = in_values
y = in_hist
f_out = open('in_values_xy.txt','w')
print >> f_out, str((x,y))
f_out.close()

#read data from file
f_in = open('in_values_xy.txt','r')
(x,y) = eval(f_in.readline())
f_in.close()
xdata = x
ydata = y
```
Fitting empirical data

We will use the least squares method to fit the data.

```python
# define our (line) fitting function and then initialize parameters
# using the pinit variable
fitfunc = lambda p, x: p[0] * (x+p[1])**(-p[2])
pinit = [1.0,1.0,1.0]

errfunc = lambda p, x, y: np.log(y) - np.log(fitfunc(p, x))
out = optimize.leastsq(errfunc, pinit, args=(xdata, ydata), xtol=1e-12)

pfinal = out[0]
fitted = np.array([fitfunc(pfinal,i) for i in xdata])
fit_label = r'$%.2f (x+%.2f)^{%.2f}$'%(pfinal[0],pfinal[1],pfinal[2])
```
Fitting empirical data

Finally we would like to plot the data and see how our results look like.

```python
#normalization of frequencies to get probability distribution
normed_ydata = [y/(1.0*sum(ydata)) for y in ydata]
print 'The sum of frequencies is: %.2f.' %sum(normed_ydata)

#plotting empirical data
pylab.loglog(xdata, normed_ydata, 'ks-')
#plotting fitted function after normalization is taking place
normed_fitted = [y/(1.0*sum(fitted)) for y in fitted]
pylab.loglog(x,normed_fitted,'r--',linewidth=2, label=fit_label)
pylab.xlabel('Degree')
pylab.ylabel('Frequency')
pylab.legend()
pylab.savefig('power_law_fit_scipy.pdf')
```
Code for fitting data to power-law distributions

Download the following python software to fit power-laws:

Powerlaw: a Python package for analysis of heavy-tailed distributions

code  paper/manual

Create a new file powerfit.py and initialise the following:

```
import powerlaw
#read data from file as before
```
Fitting empirical data

Write and read data from file as before BUT now we will operate on a python list ...

```python
import powerlaw

#read data from file
f_in = open('indegreeList.txt','r')
data = eval(f_in.readline())
f_in.close()

#fit the data to a power-law function
fit = powerlaw.Fit(data)

#print out exponent
alpha = fit.power_law.alpha
print 'Power-law exponent of the data is %f' %alpha
```
Writing your own code
Breadth First Search

With Python and NetworkX it's easy to write any graph-based algorithm

```python
from collections import deque

def breadth_first_search(g, source):
    queue = deque([(None, source)])
    enqueued = set([source])
    while queue:
        parent, n = queue.popleft()
        yield parent, n
        new = set(g[n]) - enqueued
        enqueued |= new  # s.update(new)
        queue.extend([(n, child) for child in new])
```

http://stackoverflow.com/questions/102535/what-can-you-use-python-generator-functions-for
Network Triads

Extract all unique triangles in a graph with integer node IDs

```python
def get_triangles(g):
    nodes = g.nodes()
    for n1 in nodes:
        neighbors1 = set(g[n1])
        for n2 in filter(lambda x: x>n1 and g.has_edge(n1,x), nodes):
            neighbors2 = set(g[n2])
            common = neighbors1 & neighbors2
            for n3 in filter(lambda x: x>n2, common):
                yield n1,n2,n3
```

average neighbours’ degree

Compute the average degree of each node’s neighbours (long version).

```python
def avg_neigh_degree(g):
    data = {}
    for n in g.nodes():
        if g.degree(n):
            data[n] = float(sum(g.degree(i) for i in g[n]))/g.degree(n)
    return data
```

and the more compact version in a single line:

```python
def avg_neigh_degree(g):
    return dict((n,float(sum(g.degree(i) for i in g[n]))/g.degree(n))
                 for n in g.nodes() if g.degree(n))
```
What you have learnt today about NetworkX

1. How to create graphs from scratch, with generators and by loading local data

2. How to compute basic network measures, how they are stored in NetworkX and how to manipulate them with list comprehension.

3. Getting data out of NetworkX as raw network data or analytics.

4. How to use matplotlib to visualize and plot results (useful for final report!)

5. How to use and include NetworkX features to design your own algorithms/analysis.
And some useful links...

1. Code&data used in this lecture: by request.

2. NodeXL: a graphical front-end that integrates network analysis into Microsoft Office and Excel. (http://nodexl.codeplex.com/)


4. Gephi: an interactive visualization and exploration platform (http://gephi.org/)

5. Power-law Distributions in Empirical Data: tools for fitting heavy-tailed distributions to data (http://www.santafe.edu/~aaronc/powerlaws/)

6. GraphViz: graph visualization software (http://www.graphviz.org/)

7. Matplotlib: full documentation for the plotting library (http://matplotlib.sourceforge.net/)
Be smart and save time by ...

1. Testing your code in small networks. Don’t wait for a whole day to see if it works!

2. Googling around and look for already existing code (for example ‘Louvain method for community detection’).

3. If Python too slow, two rules of thumb: a) optimise (again and again) your code. b) migrate to C++.

4. If you are totally lost and need help feel free to email me and I will try to do my best :).