Data Science
A Tutorial on Network Centrality and Mapping in Python and R

1 Pre-Requisites

This tutorial will rely on knowledge of

1. Markov chains
2. Eigenvectors (intuitive understanding and power iteration method)
3. Familiarity with Google's pageRank algorithm will be useful (see e.g. [1])
4. Having read the following papers before will also help your understanding [2–4]

2 The General Idea

We will examine flows of air traffic between the major global cities (more precisely *Metropolitan Statistical Areas*[1]). The global air traffic network is a very interesting network to study; a nice survey can be found here [2]. In this paper [3] it was found that the flow of Twitter communications between cities correlates well with air traffic flows between those same cities. Intuitively this makes sense; if 2 cities have the same principal language then its inhabitants are more likely to follow each other on Twitter and there are more likely to be air connections between the cities to facilitate trade and business.

In this paper that came out of SCAILab [4] we used this idea to show how people spreading news of the Tag challenge made a strong effort to route the message to the tag cities compared to simply broadcasting to all of their friends at random and hoping that the message would reach the target cities. We will reproduce this work using Python to perform random walks on the network and finally produce a nice map using R.

3 The Adjacency Matrix

We will start off by reading in the adjacency matrix between the cities as stored in `ADJACENCY.dat` (this is a NumPy array stored as a binary object). This matrix is row-normalised so each row sums to 1. In other words if city A has 100 flights leaving from it and 50 of these go to city B, 25 go to city C and 25 go to city D, the weights will be (A,B)=0.5, (A,C)=0.25 and (A,D)=0.25; the weight between i and j is the *proportion of the total flights leaving i which depart for j*. Note that each node in this network is a *city* and not an *airport*, for example London has several ariports; Heathrow, Stanstead and Luton. These airports are all grouped together and any flight leaving from them are grouped under the London node.[2].

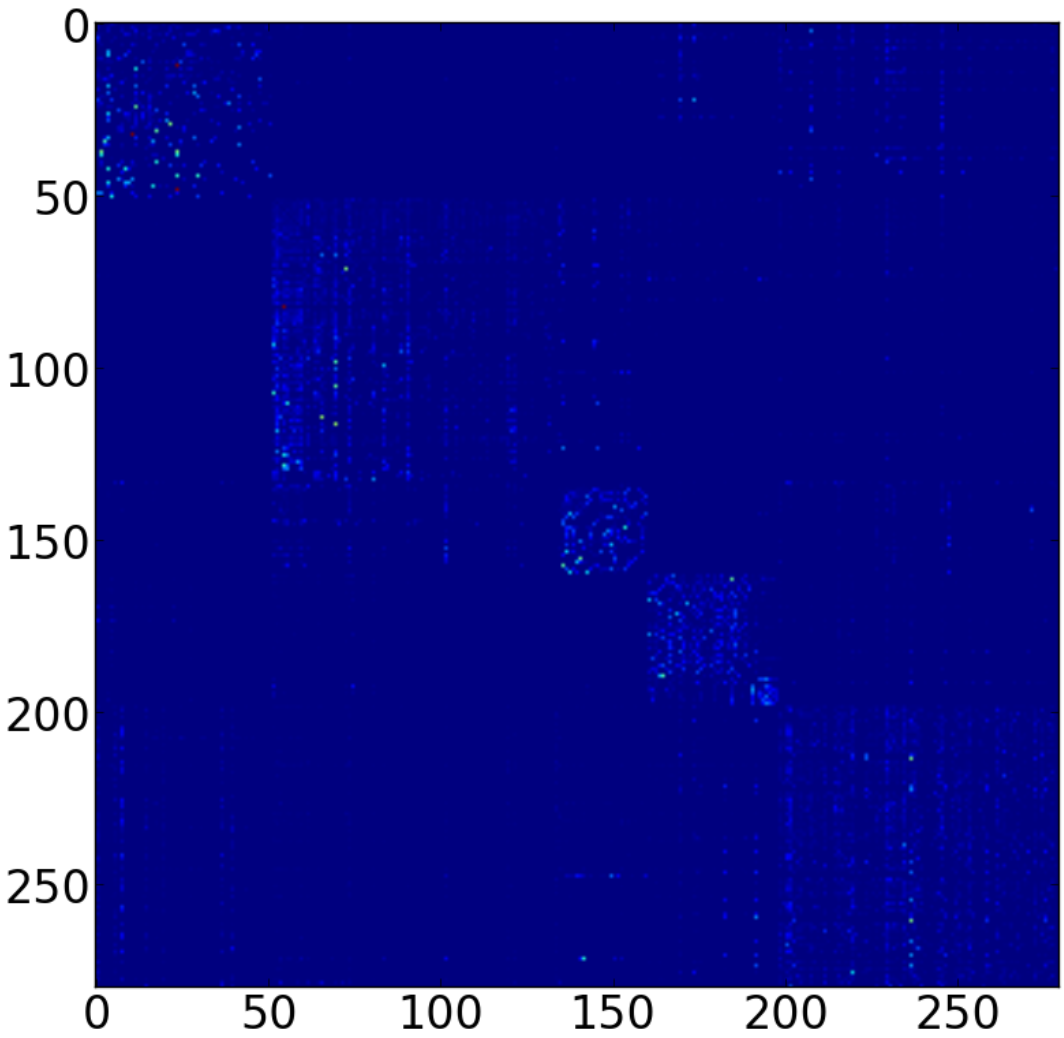
First of all let us read in the file and plot the connections as a heat map.

[1] `http://en.wikipedia.org/wiki/Metropolitan_statistical_area`

[2] This adjacency matrix was calculated with flight information taken from the OpenFlights website `http://openflights.org/`

```
import pickle
# This is used for quick reading/writing of Python objects as binary
import numpy as np
import matplotlib.pyplot as plt

adjacency=pickle.load(open('ADJACENCY.dat','r'))

plt.imshow(adjacency)
plt.show()
```

This gives us the basic heatmap of connections below.



The reason this plot is not clear is that there are some entries which are very high while most are very small. In such a case it is better to use a log scale.

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm

adjacency=pickle.load(open('ADJACENCY.dat','r'))

plt.imshow(adjacency,norm=LogNorm(vmin=np.min(adjacency),vmax=np.max(adjacency)))
# We have defined a log scaled colour map that spans the range of adjacency matrix
plt.show()
```

You will find an error at this point, the reason is that you tried to set the lower bound of the log scale to log(0) which is of course $-\infty$. So it is best to set the minimum of the scale not to 0, but to an arbitrarily small number.

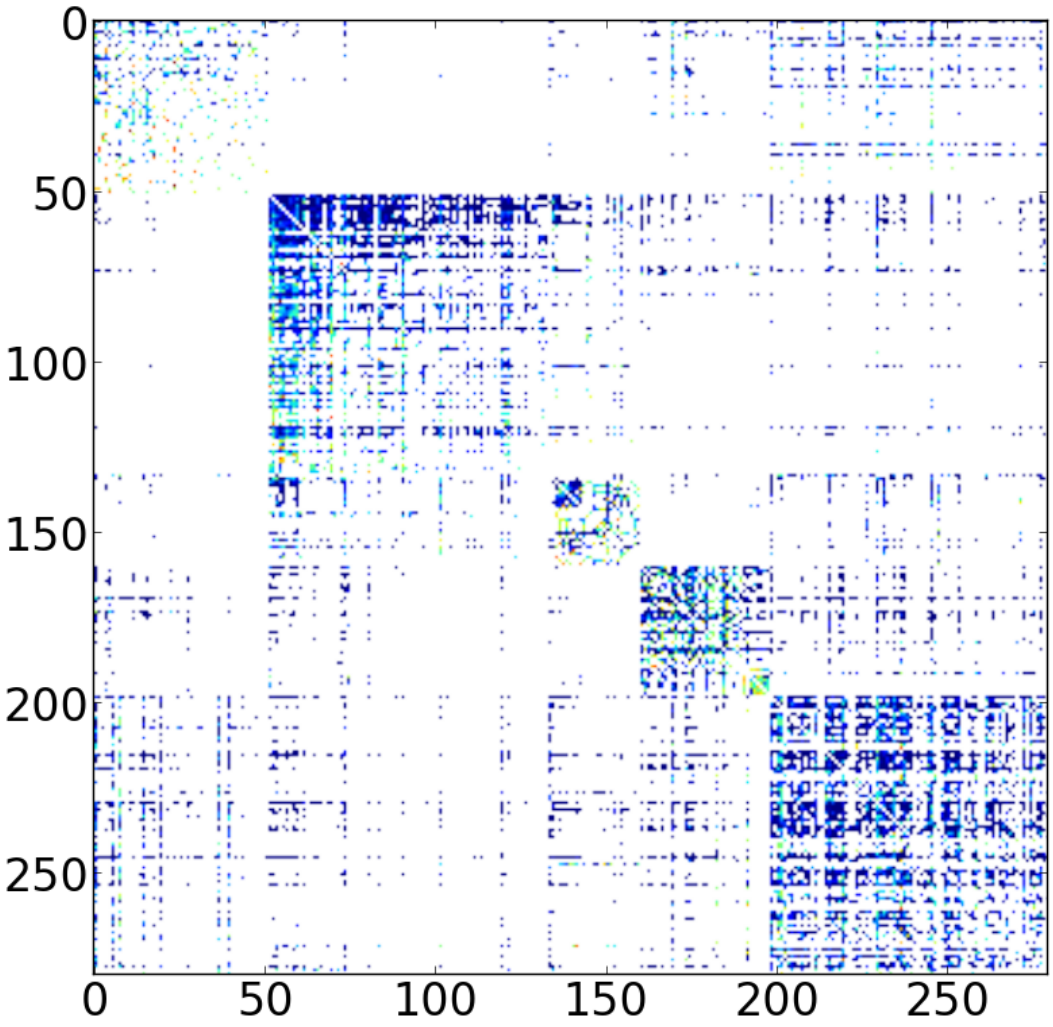
```
import pickle
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm

adjacency=pickle.load(open('ADJACENCY.dat','r'))

plt.imshow(adjacency,norm=LogNorm(vmin=0.0001,vmax=np.max(adjacency)))
# We have defined a log scaled colour map that spans the range of adjacency matrix
plt.show()
```

The problem now is that Matplotlib doesn't know what to do with the zero values in the matrix (recall that log(0)=-∞), so they are shown in white. To solve this we add the same very small constant value to each entry in the matrix so that the values are all non-zero and we add a colour bar.

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm

adjacency=pickle.load(open('ADJACENCY.dat','r'))

adjacency=adjacency+0.0001

plt.imshow(adjacency,norm=LogNorm(vmin=0.0001,vmax=np.max(adjacency)))
plt.colorbar()
plt.show()
```

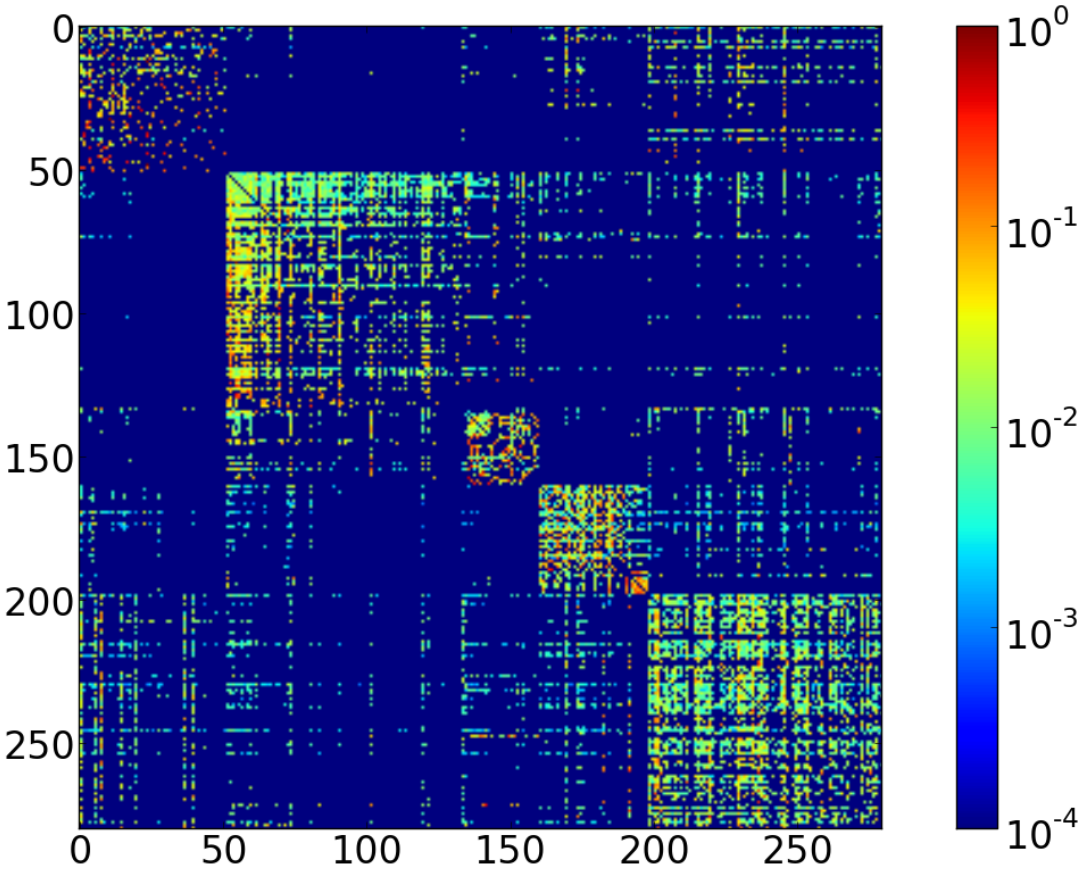
The reason for the well defined square blocks of colour is that the cities are listed by continent. So all of the European cities are numbered together and all of the Asian cities together. These cities are all mutually well connected but less so for cities outside of the same continent. The structure of communities in the airport network is analysed in more detail in [2].

Another feature that you might notice is that the diagonal is uniformly zero; there aren't any flights which take off and leave from the same airport so this makes sense. However, if we want to use this network to describe how people communicate we need to account for their friends that *are in the same city*. This is addressed in [3], the authors found that on average 39% of communications originating in a



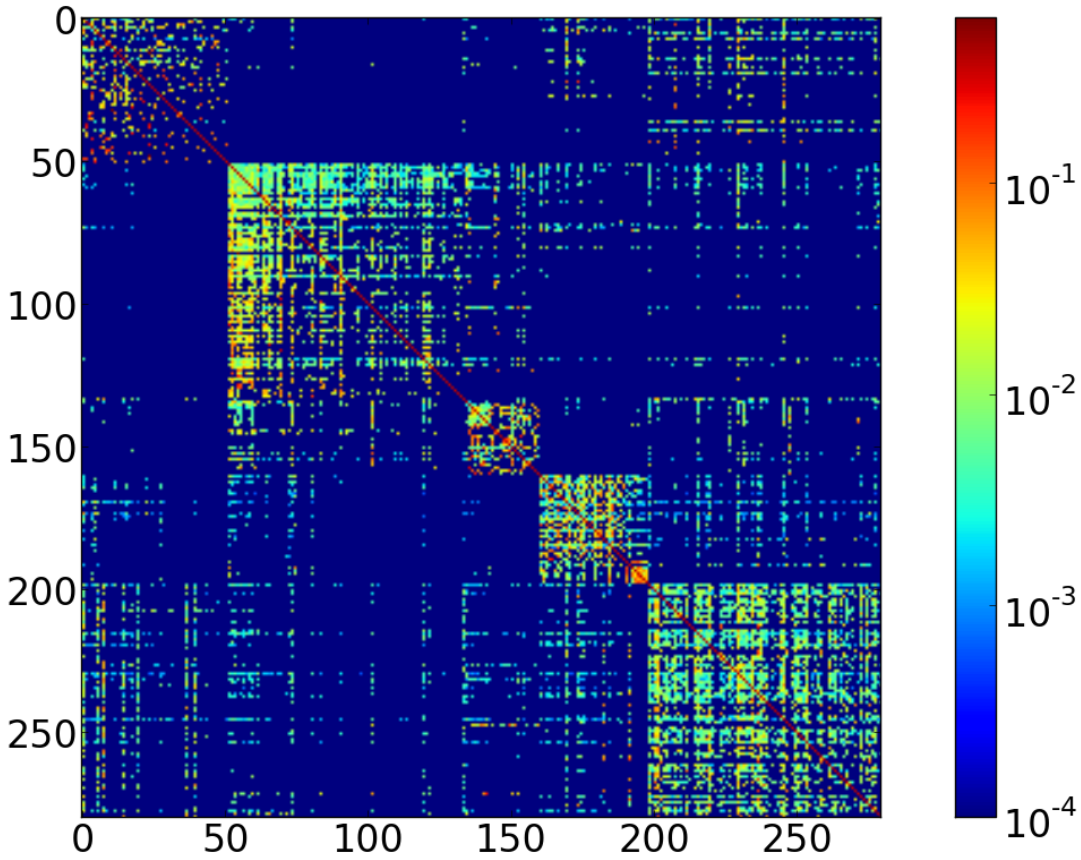
city were to friends in that same city. So we need to adjust our adjacency matrix so that the diagonal is uniformly set to 0.39, but we need each row to still sum to 1.

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm

adjacency=pickle.load(open('ADJACENCY.dat','r'))

adjacency=adjacency+0.001

selfLoopWeight=0.39
for i in range(np.shape(adjacency)[0]):
# Loop over each row
        adjacency[i,:]*=(1.0-selfLoopWeight)
```




```
        # Scale the row to 1-selfLoopWeight
        adjacency[i,i]=selfLoopWeight
        # Set diagonal term to selfLoopWeight

plt.imshow(adjacency,norm=LogNorm(vmin=np.min(adjacency),vmax=np.max(adjacency)))
plt.colorbar()
plt.show()
```

This produces the final adjacency matrix that we will use in the next section.


0
50
100
150
200
250
0
50
100
150
200
250
10^{-1}
10^{-2}
10^{-3}
10^{-4}

4 Calculating Unbiased Centrality

One definition of the centrality of a network can be made in terms of a random walk on a network. If an imaginary agent is at a node, she will move to a new node chosen at random according to the weight of the edge between them; if there is no edge between them she cannot move there. Once at the new node, the walker does the same thing again. If we follow the walker for a long time there will be some nodes where she spends a lot of her time; these are the more important nodes in the network which are linked to many other nodes. By keeping a log of how many times the walker returns to each node we can get a number ranking the importance of each node (this is the same idea Google uses to calculate the importance of webpages).

This is exactly what we are going to do with our communication network. We will perform a random walk on the network, record how much time is spent at each node and use this to find out how central certain cities are.

First of all we want to save a copy of the adjacency matrix that we created by adjusting the diagonal, we do this by adding a single line to dump a copy as a binary file.

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm

adjacency=pickle.load(open('ADJACENCY.dat','r'))

adjacency=adjacency+0.001

selfLoopWeight=0.39
for i in range(np.shape(adjacency)[0]):
        adjacency[i,:]/=(1.0-selfLoopWeight)
        adjacency[i,i]=selfLoopWeight

pickle.dump(adjacency,open('ADJACENCY_ADJUSTED.dat','w'))
# Open a file for writing and dump the adjusted adjacency matrix

plt.imshow(adjacency,norm=LogNorm(vmin=np.min(adjacency),vmax=np.max(adjacency)))
plt.colorbar()
plt.show()
```

Now we create a script to read this array back in, then we will put our walker on a randomly chosen node and let her take 1,000,000 steps on the network.

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
import random
import bisect

adjacency=pickle.load(open('ADJACENCY_ADJUSTED.dat'))

nCities=np.shape(adjacency)[0]
# Get number of cities
```

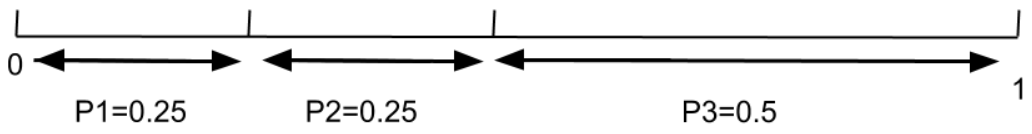
```
timeAtCity=np.zeros(shape=nCities)
# This is an array to store number of steps spent at each city

newPosition=-999
oldPosition=random.randint(0,nCities-1)
timeAtCity[oldPosition]+=1
# Choose a random city to start from

for i in xrange(1000000):

        randSample=random.random()
        # This gives us a number between 0-1

        newPosition=bisect.bisect_left(np.cumsum(adjacency[oldPosition,:]),randSample)
        # This selects a node that is connected to the current node,
        # with a probability according to the edge weight

        timeAtCity[newPosition]+=1
        oldPosition=newPosition
        # Keep a log and update

timeAtCity/=np.sum(timeAtCity)
# Normalise

plt.plot(timeAtCity)
plt.show()
```

There is one line in this script which might be somewhat confusing which we look at in a bit more detail since it is a useful trick when doing stochastic (random) simulations;

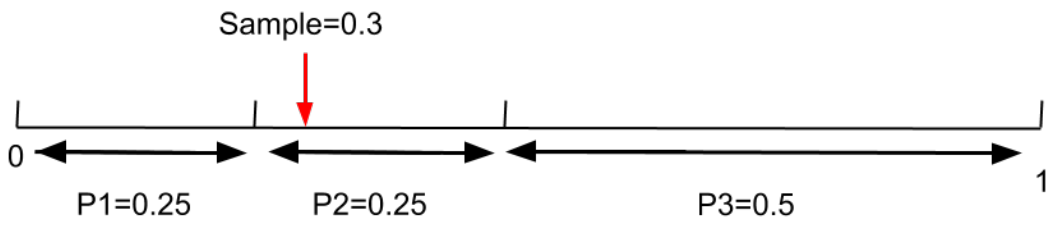
```
newPosition=bisect.bisect_left(np.cumsum(adjacency[oldPosition,:]),randSample)
```

We are interested in the weights of the edges of the city at index `oldPosition`, we can access this row with `adjacency[oldPosition,:]`. This is a vector of numbers which sums to 1. What we want is to choose an index randomly, but not *uniformly*, rather to choose an index according to the size of the edge weight.

Imagine a simple example with 3 values in a vector; P1=0.25, P2=0.25 and P3=0.5. The weights of the 3 are not equal so we want to make sure we choose P3 twice as often as we choose P1 or P2. To do this we create a number line running from 0 to 1 and divide the line up according to the probabilities as below.

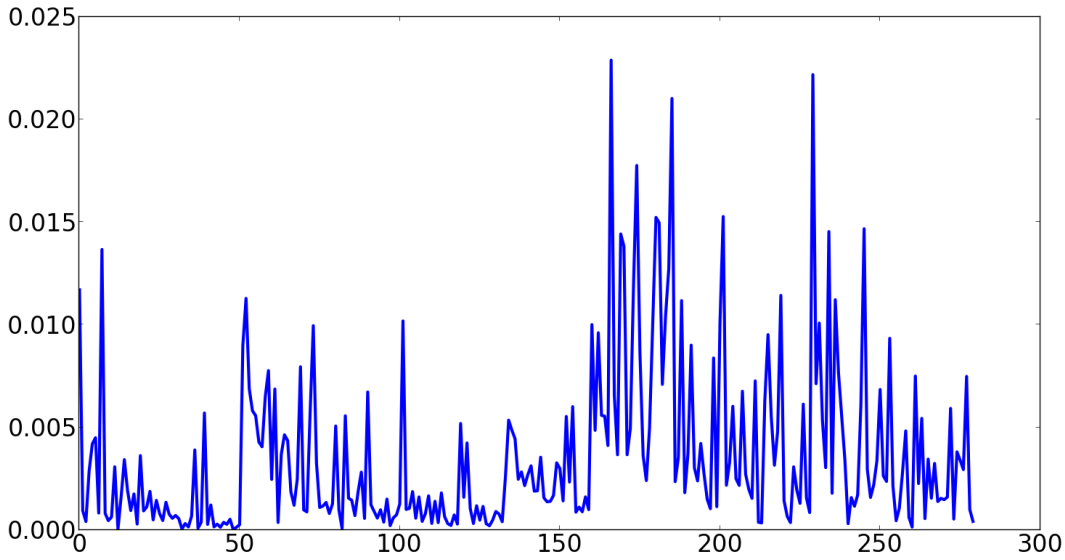


Next we generate a random number and see where the random sample lands on the number line, is it in the domain of P1, P2 or P3? If our random sample is 0.3 then we land in the domain of P2 and we return the second index. If the sample was 0.6, we would return the third index and so on. You can see by extension that this will always return the indices in the desired proportions.



`bisect_left` simply returns the index of the entry that the sample corresponds to. We use the cumulative sum because we want the entries to lie next to each other and to stretch from 0-1.

Once we run this code we produce the output below.



We see that the centrality of each of these cities varies considerably. But at this point it is hard to see what is happening since the cities are not labelled. The file `MSA_COORDS.txt` stores the names of the cities (as well as the coordinates) in order, so we will read these in and label the city with the highest centrality.

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
import random
import bisect
import csv

msaNames=[]
namesFile=csv.reader(open('MSA_COORDS.txt','r'),delimiter='')
```

```
for line in namesFile:
        msaNames.append(line[0])

adjacency=pickle.load(open('ADJACENCY\_ADJUSTED.dat'))

nCities=np.shape(adjacency)[0]

timeAtCity=np.zeros(shape=nCities)

newPosition=-999
oldPosition=random.randint(0,nCities-1)
timeAtCity[oldPosition]+=1

for i in xrange(1000000):

        randSample=random.random()

        newPosition=bisect.bisect_left(np.cumsum(adjacency[oldPosition,:]),randSample)

        timeAtCity[newPosition]+=1

        oldPosition=newPosition

timeAtCity/=np.sum(timeAtCity)

plt.plot(timeAtCity)

maxIndex=np.argmax(timeAtCity)
# Get the index of the city with the highest centrality
plt.axvline(maxIndex,color='k',linestyle='--')
# Draw a line here
plt.xticks([maxIndex],[msaNames[maxIndex]])
# Add a single tick mark for this city on x-axis

plt.show()
```

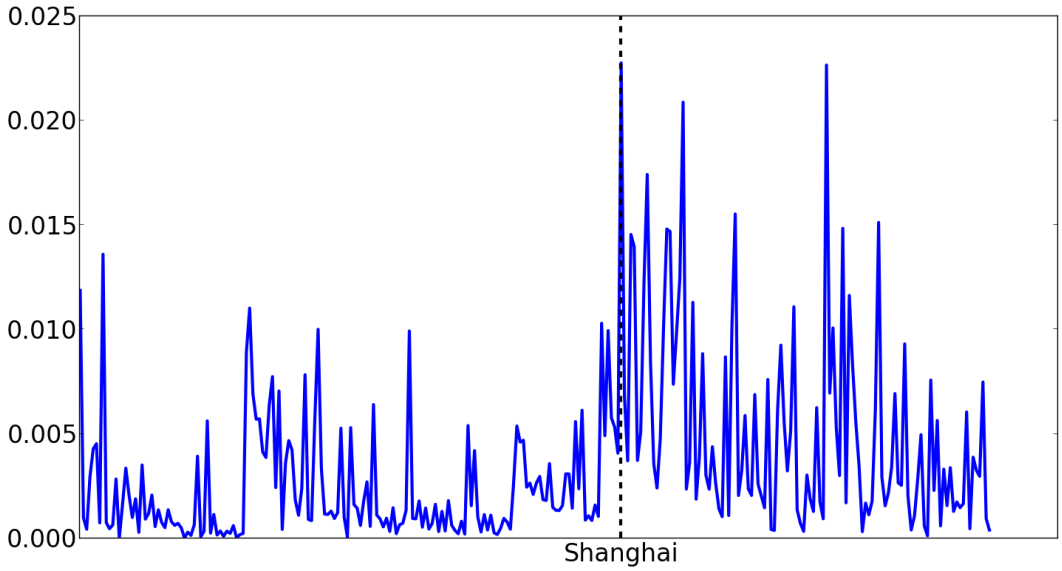
The final figure shows us that Shanghai in fact has the largest centrality. (This code can be found in `random_walk.py`)

In the context of the Tag Challenge, we would like to concentrate on the 5 cities where targets were found; London, New York, Washington DC, Stockholm and Bratislava. **Using `grep` and `MSA_COORDS.txt` find which index corresponds to each of these cities and mark them with a line as we did for Shanghai (Hint: remember that in Python counting starts from 0 and in `grep`, from 1). the result will look like the figure below.**

5 Calculating Targeted Centrality

References

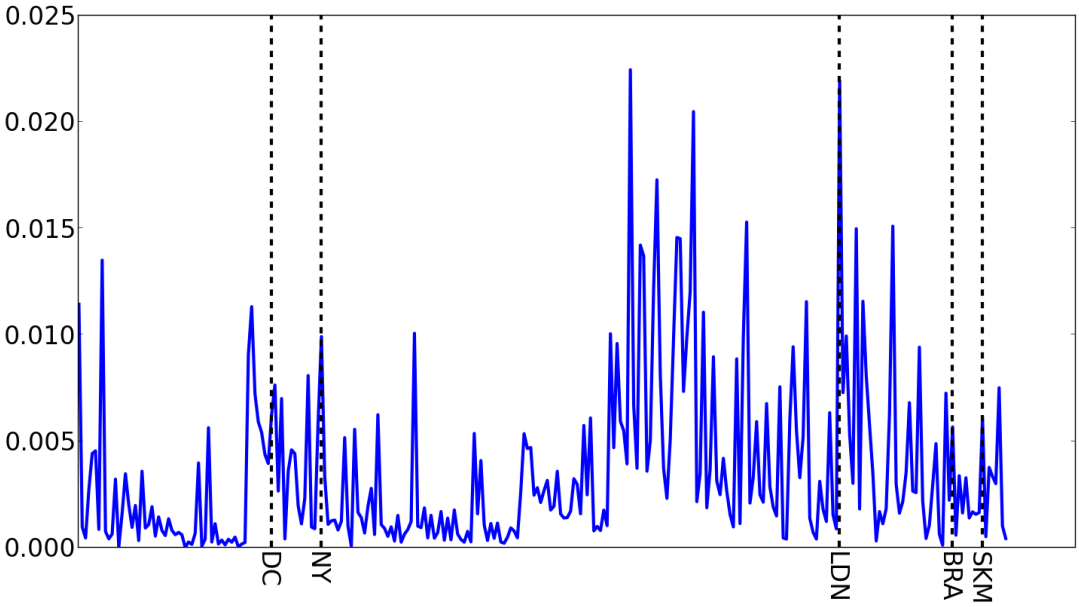
[1] M. Newman, *Networks: An Introduction*. New York, NY, USA: Oxford University Press, Inc., 2010.



[2] R. Guimera, S. Mossa, A. Turtschi, and L. A. N. Amaral, "The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles," *Proceedings of the National Academy of Sciences*, vol. 102, no. 22, pp. 7794–7799, 2005.

[3] Y. Takhteyev, A. Gruzd, and B. Wellman, "Geography of twitter networks," *Social Networks*, vol. 34, no. 1, pp. 73 – 81, 2012. Capturing Context: Integrating Spatial and Social Network Analyses.

[4] A. Rutherford, M. Cebrian, I. Rahwan, S. Dsouza, J. McInerney, V. Naroditskiy, M. Venanzi, N. R. Jennings, J. R. deLara, E. Wahlstedt, and S. U. Miller, "Targeted Social Mobilisation in a Global Manhunt," *ArXiv e-prints*, Apr. 2013.


0.025
0.020
0.015
0.010
0.005
0.000
DC
NY
LDN
BRA
SKM