Homework 2

Jonathan McFadden ***TCSS-554 : Autumn 2017***

GitHub repo located at: *https://github.com/mcfaddja/IRhw2 (https://github.com/mcfaddja/IRhw2)*

Part 1

Import Data and Initialize

First, we import the required libraries. In this case, its just *numpy*:

In [1]:
```
import numpy as np
```

For the first part of the homework, we will define the input file name and value of β as constants

In [2]:
```
filename = "input.txt"
beta = 0.85
```

We can now import the file to the variable *df*

In [3]:
```
df = np.genfromtxt(filename, dtype=float)
```

Next, we must get the maximim matrix index in the imported file, so we know how big to make our transitition matrix.

In [4]:
```
nDim0 = np.amax(df, axis=0)
nDim = int(np.amax(nDim0[0:2]))
```

The max index value of the imported data is

In [5]:
```
print(nDim)
```

```
6
```

With this value for the dimension, we can now initialize an initial transition matrix $\bar{\bar{\mathbf{A}}}$ as a 2D array of zeros.

In [6]:
```
Amat = np.zeros((nDim, nDim))
```

We now load the values from the imported file into our initial transition matrix

In [7]:
```
for row in df:
        i = int(row[0] - 1)
        j = int(row[1] - 1)
        k = row[2]

        Amat[i,j]=k
```

Checking our import gives

In [8]:
```
print(Amat)
```

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

Normalize columns

We now need to normalize the imported matrix by column. Thus, we need the sum of each column in $\bar{\bar{\mathbf{A}}}$.

In [9]:
```
colSums = Amat.sum(axis=0)
```

Since the *.sum()* routine returns a row vector, we must transpose the initial transition matrix in order for the division operation to properly normalize the columns.

In [10]:
```
Amat = np.transpose(Amat)
```

We also need a variable to store our final (*i.e. column normalized*) transition matrix, $\bar{\bar{\mathbf{M}}}$

In [11]:
```
Mmat = np.zeros((nDim,nDim))
```

We now loop through our initial transition matrix $\bar{\bar{\mathbf{A}}}$ and the column sums stored in *colSums*[ ], normalizing the *now rows* of $\bar{\bar{\mathbf{A}}}$ and storing the result in our variable for $\bar{\bar{\mathbf{M}}}$

In [12]:
```
for i, (row, colSums) in enumerate(zip(Amat, colSums)):
        if colSums != 0:
                Mmat[i,:] = row / colSums
        else:
                Mmat[i,:] = row
```

Note that we had to take into account the case of a column summing to 0 via the *if* statement in the above loop.

We now reverse the earlier transpose to get the final transition matrix out.

In [13]:
```
Mmat = np.transpose(Mmat)
```

Thus, our final transition matrix, $\bar{\bar{\mathbf{M}}}$, is

In [14]:
```
print(Mmat)
```

```
[[ 0.          0.25        0.          0.          0.          0.
  ]
 [ 0.          0.          0.33333333  0.33333333  0.33333333  0.
  ]
 [ 0.          0.25        0.          0.33333333  0.33333333  0.
  ]
 [ 0.          0.25        0.33333333  0.          0.33333333  0.
  ]
 [ 0.          0.25        0.33333333  0.33333333  0.          1.
  ]
 [ 0.          0.          0.          0.          0.          0.
  ]]
```

Initial Rank Vector

Continuing, we need an initial rank vector, $\vec{\mathbf{r}}_0$. Since the initial rank vector is defined

$$\vec{\mathbf{r}}_0 \cong \frac{1}{n}\vec{\mathbf{e}}$$

where $n \in \mathbb{Z}^+$ is the dimension of the system. Thus, we initialize and store our initial rank vector by calling

In [15]:
```
r0 = np.ones((nDim,1))
r0 = r0 * 1/nDim
```

to give its value as

In [16]:
```
print(r0)
```

```
[[ 0.16666667]
 [ 0.16666667]
 [ 0.16666667]
 [ 0.16666667]
 [ 0.16666667]
 [ 0.16666667]]
```

Itterate to Steady State

Now, we will itterate our rank vector through our stochastic equation

$$\vec{\mathbf{r}}_n = \beta \bar{\bar{\mathbf{M}}} \cdot \vec{\mathbf{r}}_{n-1} + \frac{(1-\beta)}{n}\vec{\mathbf{e}}$$

. Since we are interested in how long it takes to reach steady state, we begin by initializing a counter

In [17]:
```
nCnt = 0
```

As we are itterating until we reach steady state, we must initialize a test condition for our *while* loop

In [18]:
```
testCond = False
```

With everything now ready, we can run the *while* loop to itterate until we reach steady state

In [19]:
```
while testCond == False:
        r1 = beta*np.dot(Mmat,r0) + (1-beta)*np.ones((nDim,1))*(1/nDim)
        #testCond = True
        testCond = np.allclose(r0, r1)
        if testCond == False:
                r0 = r1
        nCnt = nCnt + 1
```

This gives the number of itterations required to reach steady state as

In [20]:
```
print(nCnt)
```

```
41
```

and the final value of the rank vector as

In [21]:
```
print(r1)
```

```
[[ 0.05704271]
 [ 0.15078799]
 [ 0.14246522]
 [ 0.14246522]
 [ 0.15902366]
 [ 0.025     ]]
```

Part 2 (*EC*)

For the extra-credit part of the assignment, the above code has been adapted to work in the general case. To do this, three changes were made to the above code. These changes are

- The code was defined as a class with an initializer that accepts a string for the filename and a double for the value of β. When initialized, the class computes all the requested values and prints them to the terminal.
- The constants which previously held the values for the file name and β are changed to point at the values passed by the initializer.
- A safety counter was added to the *while* loop to prevent it from looping infinitely.

The code for this class is given below

In [22]:
```
class pageRanker:
    import numpy as np

    def __init__(self, filename, beta):
        # import file
        df = np.genfromtxt(filename, dtype=float)

        # get required size of matrix
        nDim0 = np.amax(df, axis=0)
        nDim = int(np.amax(nDim0[0:2]))

        # create initial matrix
        Amat = np.zeros((nDim, nDim))

        # load data into initial matrix
        for row in df:
            i = int(row[0] - 1)
            j = int(row[1] - 1)
            k = row[2]

            Amat[i,j]=k

        # normalize by column
        colSums = Amat.sum(axis=0)
        # transpose to match dimensionality
        Amat = np.transpose(Amat)
        #Mmat = np.zeros((nDim,nDim))
        for i, (row, colSums) in enumerate(zip(Amat, colSums)):
            if colSums != 0:
                Amat[i,:] = row / colSums
            else:
                Amat[i,:] = row

        # reverse the transpose
        Amat = np.transpose(Amat)

        # create initial r vector
        r0 = np.ones((nDim,1))
        r0 = r0 * 1/nDim
```

```
        r00 = r0

        # iterate rank vector
        nCnt = 0 # number of itterations
        testCond = False # initialize test condition
        nMax = 1000000 # safety counter to prevent infinite looping
        while testCond == False:
            r1 = beta*np.dot(Amat,r0) + (1-beta)*np.ones((nDim,1))*(1/nD
im)

            testCond = np.allclose(r0, r1)
            if testCond == False:
                r0 = r1

            nCnt = nCnt + 1
            if nCnt == nMax:
                break

        print("Normalized adjacency matrix")
        print(Amat)
        print("")

        print("The original page rank vector")
        print(r00)
        print("")

        print("number of itterations")
        print(nCnt)
        print("")

        print("Final page rank vector")
        print(r1)
```

Running this class on the input from the first part of the homework yields

In [23]:
```
pageRanker("input.txt", 0.85)
```

```
Normalized adjacency matrix
[[ 0.          0.25        0.          0.          0.          0.
  ]
 [ 0.          0.          0.33333333  0.33333333  0.33333333  0.
  ]
 [ 0.          0.25        0.          0.33333333  0.33333333  0.
  ]
 [ 0.          0.25        0.33333333  0.          0.33333333  0.
  ]
 [ 0.          0.25        0.33333333  0.33333333  0.          1.
  ]
 [ 0.          0.          0.          0.          0.          0.
  ]]

The original page rank vector
[[ 0.16666667]
 [ 0.16666667]
 [ 0.16666667]
 [ 0.16666667]
 [ 0.16666667]
 [ 0.16666667]]

number of itterations
41

Final page rank vector
[[ 0.05704271]
 [ 0.15078799]
 [ 0.14246522]
 [ 0.14246522]
 [ 0.15902366]
 [ 0.025     ]]
```

Out[23]:
```
<__main__.pageRanker at 0x10fbfd7f0>
```

which is the same as before.

Using this method on the provided large adjacency matrix file, "AdjacencyMatrix.txt", gives

In [ ]:
```
#pageRanker("AdjacencyMatrix.txt", 0.85)
```

The above implementation of the '*pageRanker*( )' class consumes too much memory for large matrices. Therefore, it has been recoded to implement sparse matrices. This requires importing the ***scipy*** library

In [29]:
```
import scipy as sp
```

The version with sparse matrices implemented is called '*pageRankerSPAR*( )' and its code is as follow:

In [38]:
```
class pageRankerSPAR:
```

```
    import numpy as np
    #from scipy import sparse
    import scipy as sp

    def __init__(self, filename, beta):
        # import file
        df = np.genfromtxt(filename, dtype=float)

        # get required size of matrix
        nDim0 = np.amax(df, axis=0)
        nDim = int(np.amax(nDim0[0:2]))

        # create initial matrix
        Amat = sp.sparse.lil_matrix(np.zeros((nDim, nDim)))

        # load data into initial matrix
        for row in df:
            i = int(row[0] - 1)
            j = int(row[1] - 1)
            k = row[2]

            Amat[i,j]=k

        # normalize by column
        colSums = Amat.sum(axis=0)
        # transpose to match dimensionality
        Amat = np.transpose(Amat)
        #Mmat = np.zeros((nDim,nDim))
        for i, (row, colSum) in enumerate(zip(Amat, colSums)):
            if np.any(colSum):
                Amat[i,:] = row / colSums
            else:
                Amat[i,:] = row

        # reverse the transpose
        Amat = np.transpose(Amat)

        # create initial r vector
        r0 = np.ones((nDim,1))
        r0 = r0 * 1/nDim
        r00 = r0

        # iterate rank vector
        nCnt = 0 # number of itterations
        testCond = False # initialize test condition
        nMax = 1000000 # safety counter to prevent infinite looping
        while testCond == False:
            r1 = beta*sp.sparse.lil_matrix.dot(Amat,r0) + (1-beta)*np.on
es((nDim,1))*(1/nDim)

            testCond = np.allclose(r0, r1)
            if testCond == False:
                r0 = r1

            nCnt = nCnt + 1
            if nCnt == nMax:
                break
```