PageRank

In this notebook, you'll build on your knowledge of eigenvectors and eigenvalues by exploring the PageRank algorithm. The notebook is in two parts, the first is a worksheet to get you up to speed with how the algorithm works - here we will look at a micro-internet with fewer than 10 websites and see what it does and what can go wrong. The second is an assessment which will test your application of eigentheory to this problem by writing code and calculating the page rank of a large network representing a sub-section of the internet.

Part 1 - Worksheet

Introduction ¶

PageRank (developed by Larry Page and Sergey Brin) revolutionized web search by generating a ranked list of web pages based on the underlying connectivity of the web. The PageRank algorithm is based on an ideal random web surfer who, when reaching a page, goes to the next page by clicking on a link. The surfer has equal probability of clicking any link on the page and, when reaching a page with no links, has equal probability of moving to any other page by typing in its URL. In addition, the surfer may occasionally choose to type in a random URL instead of following the links on a page. The PageRank is the ranked order of the pages from the most to the least probable page the surfer will be viewing.

In [32]:
```
# Before we begin, let's load the libraries.
%pylab notebook
import numpy as np
import numpy.linalg as la
from readonly.PageRankFunctions import *
np.set_printoptions(suppress=True)
```

```
Populating the interactive namespace from numpy and matplotlib
```

PageRank as a linear algebra problem

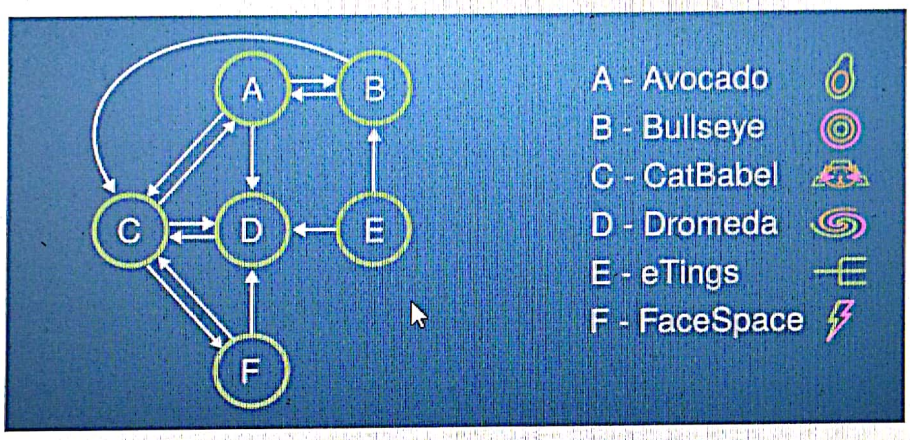
Let's imagine a micro-internet, with just 6 websites (**A**vocado, **B**ullseye, **C**alBabel, **D**romeda, **e**Tings, and **F**aceSpace). Each website links to some of the others, and this forms a network as shown,

Markdown

Submit Assignment

PageRank as a linear algebra problem

Let's imagine a micro-internet, with just 6 websites (**A**vocado, **B**ullseye, **C**atBabel, **D**romeda, **e**Tings, and **F**aceSpace). Each website links to some of the others, and this forms a network as shown,



The design principle of PageRank is that important websites will be linked to by important websites. This somewhat recursive principle will form the basis of our thinking.

Imagine we have 100 *Procrastinating Pat*s on our micro-internet, each viewing a single website at a time. Each minute the Pats follow a link on their website to another site on the micro-internet. After a while, the websites that are most linked to will have more Pats visiting them, and in the long run, each minute for every Pat that leaves a website, another will enter keeping the total numbers of Pats on each website constant. The PageRank is simply the ranking of websites by how many Pats they have on them at the end of this process.

We represent the number of Pats on each website with the vector,

$$r = \begin{bmatrix} r_A \\ r_B \\ r_C \\ r_D \\ r_E \\ r_F \end{bmatrix}$$

And say that the number of Pats on each website in minute $i + 1$ is related to those at minute i by the matrix transformation

And say that the number of Pats on each website in minute $i + 1$ is related to those at minute i by the matrix transformation

$$r^{(i+1)} = L\, r^{(i)}$$

with the matrix L taking the form,

$$L = \begin{bmatrix} L_{A\to A} & L_{B\to A} & L_{C\to A} & L_{D\to A} & L_{E\to A} & L_{F\to A} \\ L_{A\to B} & L_{B\to B} & L_{C\to B} & L_{D\to B} & L_{E\to B} & L_{F\to B} \\ L_{A\to C} & L_{B\to C} & L_{C\to C} & L_{D\to C} & L_{E\to C} & L_{F\to C} \\ L_{A\to D} & L_{B\to D} & L_{C\to D} & L_{D\to D} & L_{E\to D} & L_{F\to D} \\ L_{A\to E} & L_{B\to E} & L_{C\to E} & L_{D\to E} & L_{E\to E} & L_{F\to E} \\ L_{A\to F} & L_{B\to F} & L_{C\to F} & L_{D\to F} & L_{E\to F} & L_{F\to F} \end{bmatrix}$$

where the columns represent the probability of leaving a website for any other website, and sum to one. The rows determine how likely you are to enter a website from any other, though these need not add to one. The long time behaviour of this system is when $r^{(i+1)} = r^{(i)}$, so we'll drop the superscripts here, and that allows us to write,

$$L\, r = r$$

which is an eigenvalue equation for the matrix L, with eigenvalue 1 (this is guaranteed by the probabalistic structure of the matrix L).

Complete the matrix L below, we've left out the column for which websites the *FaceSpace* website (F) links to. Remember, this is the probability to click on another website from this one, so each column should add to one (by scaling by the number of links).

In [33]:
```
# Replace the ??? here with the probability of clicking a link to each website when leaving Website F (FaceSpace).
L = np.array([[0,   1/2, 1/3, 0, 0,   0 ],
              [1/3, 0,   0,   0, 1/2, 0 ],
              [1/3, 1/2, 0,   1, 0,   1/2 ],
              [1/3, 0,   1/3, 0, 1/2, 1/2 ],
              [0,   0,   0,   0, 0,   0 ],
              [0,   0,   1/3, 0, 0,   0 ]])
```

In principle, we could use a linear algebra library, as below, to calculate the eigenvalues and vectors. And this would work for a small system. But this gets unmanagable for large systems. And since we only care about the principal eigenvector (the one with the largest eigenvalue, which will be 1 in this case), we can use the *power iteration method* which will scale better, and is faster for large systems.

Use the code below to peek at the PageRank for this micro-internet.

In [34]:
```
eVals, eVecs = la.eig(L) # Gets the eigenvalues and vectors
order = np.absolute(eVals).argsort()[::-1] # Orders them by their eigenvalues
eVals = eVals[order]
eVecs = eVecs[:,order]

r = eVecs[:, 0] # Sets r to be the principal eigenvector
100 * np.real(r / np.sum(r)) # Make this eigenvector sum to one, then multiply by 100 Procrastinating Pats
```

Out[34]:
```
array([ 16.        ,   5.33333333,  40.        ,  25.33333333,
         0.        ,  13.33333333])
```

We can see from this list, the number of Procrastinating Pats that we expect to find on each website after long times. Putting them in order of *popularity* (based on this metric), the PageRank of this micro-internet is:

CatBabel, **D**romeda, **A**vocado, **F**aceSpace, **B**ullseye, **e**Tings

Referring back to the micro-internet diagram; is this what you would have expected? Convince yourself that based on which pages seem important given which others link to them, that this is a sensible ranking.

Let's now try to get the same result using the Power-Iteration method that was covered in the video. This method will be much better at dealing with large systems.

First let's set up our initial vector, $\mathbf{r}^{(0)}$, so that we have our 100 Procrastinating Pats equally distributed on each of our 6 websites.

In [35]:
```
r = 100 * np.ones(6) / 6 # Sets up this vector (6 entries of 1/6 × 100 each)
r # Shows it's value
```

Out[35]:
```
array([ 16.66666667,  16.66666667,  16.66666667,  16.66666667,
        16.66666667,  16.66666667])
```

Next, let's update the vector to the next minute, with the matrix L. Run the following cell multiple times, until the answer stabilises.

In [38]:
```
r = L @ r # Apply matrix L to r
r # Show it's value
# Re-run this cell multiple times to converge to the correct answer.
```

Out[38]:
```
array([ 16.35802469,   6.63580247,  35.80246914,  27.16049383,
         0.        ,  14.04320988])
```

We can automate applying this matrix multiple times as follows,

In [7]:
```
r = 100 * np.ones(6) / 6 # Sets up this vector (6 entries of 1/6 × 100 each)
for i in np.arange(100) : # Repeat 100 times
    r = L @ r
r
```

Out[7]:
```
array([ 16.        ,   5.33333333,  40.        ,  25.33333333,
         0.        ,  13.33333333])
```

Or even better, we can keep running until we get to the required tolerance.

In [8]:
```
r = 100 * np.ones(6) / 6 # Sets up this vector (6 entries of 1/6 × 100 each)
lastR = r
r = L @ r
i = 0
while la.norm(lastR - r) > 0.01 :
    lastR = r
    r = L @ r
```

In [8]:
```
r = 100 * np.ones(6) / 6 # Sets up this vector (6 entries of 1/6 × 100 each)
lastR = r
r = L @ r
i = 0
while la.norm(lastR - r) > 0.01 :
    lastR = r
    r = L @ r
    i += 1
print(str(i) + " iterations to convergence.")
r
```

```
18 iterations to convergence.
```

Out[8]:
```
array([ 16.00149917,   5.33252025,  39.99916911,  25.3324738 ,
         0.        ,  13.33433767])
```

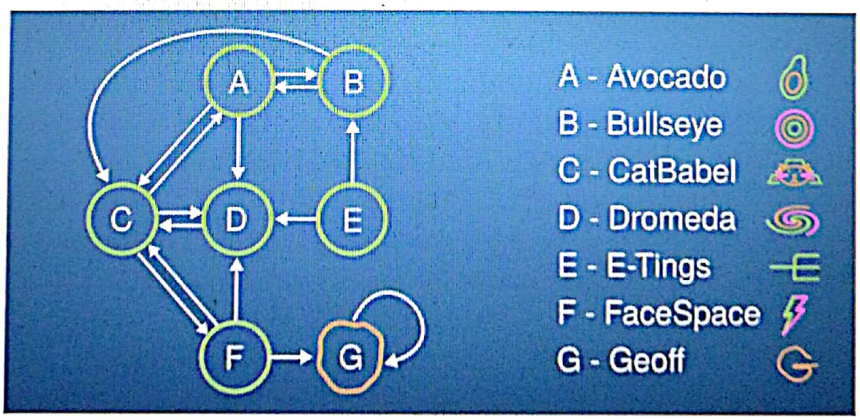
See how the PageRank order is established fairly quickly, and the vector converges on the value we calculated earlier after a few tens of repeats.

Congratulations! You've just calculated your first PageRank!

Damping Parameter

The system we just studied converged fairly quickly to the correct answer. Let's consider an extension to our micro-internet where things start to go wrong.

Say a new website is added to the micro-internet: *Geoff*'s Website. This website is linked to by *FaceSpace* and only links to itself.



Intuitively, only *FaceSpace*, which is in the bottom half of the page rank, links to this website amongst the two others it links to, so we might expect *Geoff*'s site to have a correspondingly low PageRank score.

Build the new L matrix for the expanded micro-internet, and use Power-Iteration on the Procrastinating Pat vector. See what happens...

In [9]:
```
# We'll call this one L2, to distinguish it from the previous L.
L2 = np.array([[0,   1/2, 1/3, 0, 0,   0, 0 ],
               [1/3, 0,   0,   0, 1/2, 0, 0 ],
               [1/3, 1/2, 0,   1, 0,   1/3, 0 ],
               [1/3, 0,   1/3, 0, 1/2, 1/3, 0 ],
               [0,   0,   0,   0, 0,   0, 0 ],
               [0,   0,   1/3, 0, 0,   0, 0 ],
               [0,   0,   0,   0, 0,   1/3, 1 ]])
```

In [10]:
```
r = 100 * np.ones(7) / 7 # Sets up this vector (6 entries of 1/6 × 100 each)
lastR = r
r = L2 @ r
i = 0
while la.norm(lastR - r) > 0.01 :
    lastR = r
    r = L2 @ r
    i += 1
print(str(i) + " iterations to convergence.")
r
```

```
131 iterations to convergence.
```

Out[10]:
```
array([  0.03046998,   0.01064323,   0.07126612,   0.04423198,
         0.        ,   0.02489342,  99.81849527])
```

That's no good! *Geoff* seems to be taking all the traffic on the micro-internet, and somehow coming at the top of the PageRank. This behaviour can be understood, because once a Pat get's to *Geoff's* Website, they can't leave, as all links head back to Geoff.

To combat this, we can add a small probability that the Procrastinating Pats don't follow any link on a webpage, but instead visit a website on the micro-internet at random. We'll say the probability of them following a link is d and the probability of choosing a random website is therefore $1 - d$. We can use a new matrix to work out where the Pat's visit each minute.

$$M = dL + \frac{1-d}{n} J$$

where J is an $n \times n$ matrix where every element is one.

If d is one, we have the case we had previously, whereas if d is zero, we will always visit a random webpage and therefore all webpages will be equally likely and equally ranked. For this extension to work best, $1 - d$ should be somewhat small - though we won't go into a discussion about exactly how small.

Let's retry this PageRank with this extension.

In [11]:
```
d = 0.5 # Feel free to play with this parameter after running the code once.
M = d * L2 + (1-d)/7 * np.ones([7, 7]) # np.ones() is the J matrix, with ones for each entry.
```

In [12]:
```
r = 100 * np.ones(7) / 7 # Sets up this vector (6 entries of 1/6 × 100 each)
lastR = r
r = M @ r
i = 0
while la.norm(lastR - r) > 0.01 :
    lastR = r
    r = M @ r
    i += 1
print(str(i) + " iterations to convergence.")
r
```

```
8 iterations to convergence.
```

Out[12]:
```
array([ 13.68217054,  11.20902965,  22.41964343,  16.7593433 ,
         7.14285714,  10.87976354,  17.90719239])
```

This is certainly better, the PageRank gives sensible numbers for the Procrastinating Pats that end up on each webpage. This method still predicts Geoff has a high ranking webpage however. This could be seen as a consequence of using a small network. We could also get around the problem by not counting self-links when producing the L matrix (an if a website has no outgoing links, make it link to all websites equally). We won't look further down this route, as this is in the realm of improvements to PageRank, rather than eigenproblems.

You are now in a good position, having gained an understanding of PageRank, to produce your own code to calculate the PageRank of a website with thousands of entries.

Good Luck!

Part 2 - Assessment

In this assessment, you will be asked to produce a function that can calculate the PageRank for an arbitrarily large probability matrix. This, the final assignment of the course, will give less guidance than previous assessments. You will be expected to utilise code from earlier in the worksheet and re-purpose it to your needs.

How to submit

Edit the code in the cell below to complete the assignment. Once you are finished and happy with it, press the *Submit Assignment* button at the top of this notebook.

Please don't change any of the function names, as these will be checked by the grading script.

If you have further questions about submissions or programming assignments, here is a list of Q&A. You can also raise an issue on the discussion forum. Good luck!

If you have further questions about submissions or programming assignments, here is a list of Q&A. You can also raise an issue on the discussion forum. Good luck!

In [39]:
```
# PACKAGE
# Here are the imports again, just in case you need them.
# There is no need to edit or submit this cell.
import numpy as np
import numpy.linalg as la
from readonly.PageRankFunctions import *
np.set_printoptions(suppress=True)
```

In [40]:
```
# GRADED FUNCTION
# Complete this function to provide the PageRank for an arbitrarily sized internet.
# I.e. the principal eigenvector of the damped system, using the power iteration method.
# (Normalisation doesn't matter here)
# The functions inputs are the linkMatrix, and d the damping parameter - as defined in this worksheet.
# (The damping parameter, d, will be set by the function - no need to set this yourself.)
def pageRank(linkMatrix, d) :
    n = linkMatrix.shape[0]
    M = d * linkMatrix + (1-d)/n * np.ones([n, n])
    r = 100 * np.ones(n) / n
    lastR = r
    r = M @ r
    i = 0
    while la.norm(lastR - r) > 0.01 :
        lastR = r
        r = M @ r
        i += 1
    print(str(i) + " iterations to convergence.")

    return r
```

Test your code before submission

To test the code you've written above, run the cell (select the cell above, then press the play button [►|] or press shift-enter). You can then use the code below to test out your function. You don't need to submit this cell; you can edit and run it as much as you like.

In [41]:
```
# Use the following function to generate internets of different sizes.
generate_internet(5)
```

Out[41]:
```
array([[ 0.        ,  0.2       ,  0.        ,  0.        ,  0.33333333],
       [ 0.        ,  0.2       ,  0.5       ,  0.        ,  0.        ],
       [ 1.        ,  0.2       ,  0.5       ,  0.        ,  0.        ],
       [ 0.        ,  0.2       ,  0.        ,  1.        ,  0.33333333],
       [ 0.        ,  0.2       ,  0.        ,  0.        ,  0.33333333]])
```

In [42]:
```
# Test your PageRank method against the built in "eig" method.
# You should see yours is a lot faster for large internets
L = generate_internet(10)
```

In [43]:
```
pageRank(L, 1)
```

```
18 iterations to convergence.
```

Out[43]:
```
array([ 12.34678345,   3.70332638,  14.81622862,   0.00097287,
         9.87758282,   0.0005337 ,  18.51551368,  17.28437325,
        19.75135884,   3.70332638])
```

In [44]:
```
# Do note, this is calculating the eigenvalues of the link matrix, L,
# without any damping. It may give different results that your pageRank function.
# If you wish, you could modify this cell to include damping
# (There is no credit for this though)
eVals, eVecs = la.eig(L) # Gets the eigenvalues and vectors
order = np.absolute(eVals).argsort()[::-1] # Orders them by their eigenvalues
eVals = eVals[order]
eVecs = eVecs[:,order]

r = eVecs[:, 0]
100 * np.real(r / np.sum(r))
```

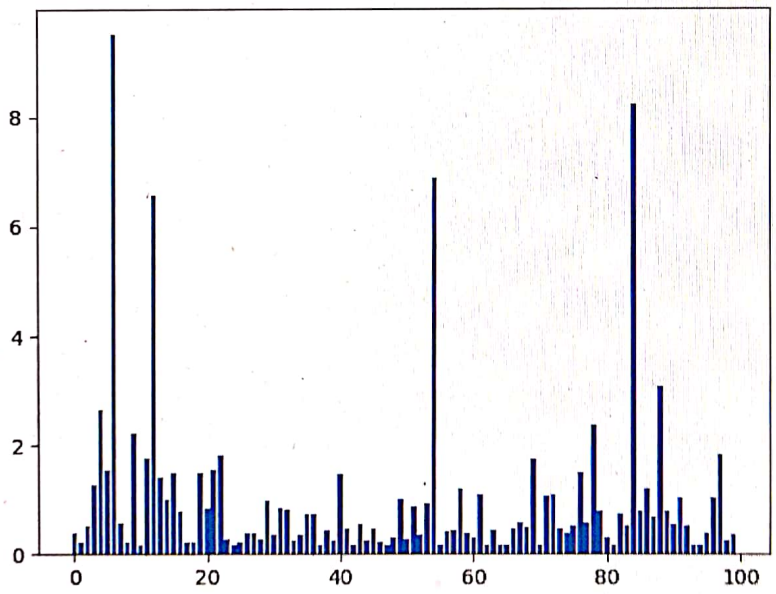
Out[44]:
```
array([ 12.34567901,   3.70370371,  14.81481481,   0.00000002,
         9.87654321,   0.00000001,  18.51851851,  17.28395061,
        19.75308641,   3.70370371])
```

In [45]:
```
# You may wish to view the PageRank graphically.
# This code will draw a bar chart, for each (numbered) website on the generated internet,
# The height of each bar will be the score in the PageRank.
# Run this code to see the PageRank for each internet you generate.
# Hopefully you should see what you might expect
# - there are a few clusters of important websites, but most on the internet are rubbish!
%pylab notebook
r = pageRank(generate_internet(100), 0.9)
plt.bar(arange(r.shape[0]), r);
```

```
Populating the interactive namespace from numpy and matplotlib
34 iterations to convergence.
```

In [45]:
```
# You may wish to view the PageRank graphically.
# This code will draw a bar chart, for each (numbered) website on the generated internet,
# The height of each bar will be the score in the PageRank.
# Run this code to see the PageRank for each internet you generate.
# Hopefully you should see what you might expect
# - there are a few clusters of important websites, but most on the internet are rubbish!
%pylab notebook
r = pageRank(generate_internet(100), 0.9)
plt.bar(arange(r.shape[0]), r);
```

```
Populating the interactive namespace from numpy and matplotlib
34 iterations to convergence.
```



In [ ]: