# 14 The PageRank Algorithm

**Lab Objective:** *Model a network as a graph and implement the PageRank algorithm based on this model. Use PageRank to predict the rankings of sports teams.*

As of 2013, the PageRank algorithm is one of over 200 algorithms that Google uses to determine the *rank*, or relative importance, of a webpage. Named for Larry Page, cofounder of Google, this algorithm ranks pages based on how many other pages link to them.

## The Internet as a Graph

The PageRank algorithm models the internet with a directed graph. Each webpage is a node, and there is an edge from node $i$ to node $j$ if page $i$ links to page $j$. Let $\text{In}(i)$ be the websites linking to page $i$ and let $\text{Out}(i)$ be the websites that page $i$ links to. That is, $\text{In}(i)$ is the set of nodes with an arrow to node $i$, and $\text{Out}(i)$ is the set of nodes with an arrow from node $i$. An example is illustrated in Figure 14.1.

The PageRank algorithm ranks pages by how many others link to them. A link from a more important page counts more than one from a less important page. For example, in Figure 14.1 we would expect node 0 to have a very high rank because every other node links to it. Consequently, we would expect node 7 to have a fairly high rank because node 0 links to it, even though node 0 is the only node to do so.

## The PageRank Algorithm

The PageRank algorithm assumes that a surfer chooses a starting webpage randomly. Then, if the surfer is at page $i$, they randomly select a page from $\text{Out}(i)$ to visit next. This means that the surfer's chance of being on page $i$ at time $t$ is determined by where they were at time $t-1$.

Suppose the internet has $N$ webpages, and let $p_i(t)$ be the likelihood that the surfer is on page $i$ at time $t$. Then the probabilities $p_i(t)$ are given by

$$p_i(0) = \frac{1}{N} \qquad p_i(t+1) = \sum_{j \in \text{In}(i)} \frac{p_j(t)}{|\text{Out}(j)|}. \tag{14.1}$$

For example, in Figure 14.1 we have $N = 8$, and

$$p_6(t+1) = \frac{p_3(t)}{3} + \frac{p_4(t)}{3} + \frac{p_5(t)}{2}.$$

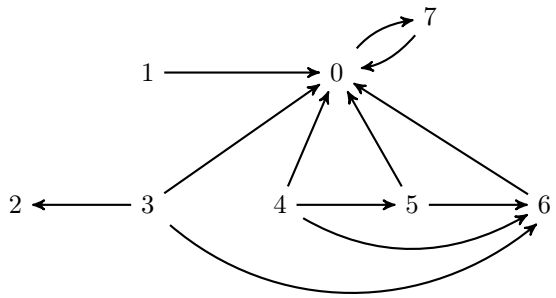


Figure 14.1: This directed graph describes the links between 8 webpages. In this example, $\text{In}(0) = \{1, 3, 4, 5, 6, 7\}$ and $\text{Out}(0) = \{7\}$.

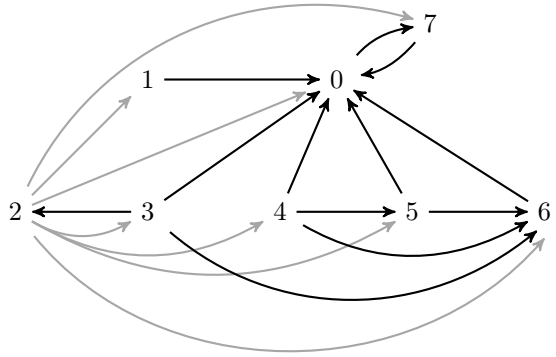


Figure 14.2: Here Figure 14.1 has been modified to guarantee that page 2 is no longer a sink. A new link has been added from page 2 to every other page (the added links are grey).

## Refining the Model: Pages with No Outbound Links

A node with no outbound links, such as node 2 in Figure 14.1, is called a *sink*. According to our model, if the surfer ever visits a sink, they will stay there forever.

This is not very realistic; in this situation, a person would likely select another webpage at random and begin surfing again. Hence, in our model we replace sinks with nodes linking to every other page. This means we modify Figure 14.1 (where node 2 is a sink) to look like Figure 14.2.

## Refining the Model: Adding Boredom

The equations in (14.1) assume that the current page must link to the next page. However, the model is more realistic if we assume that the surfer sometimes gets bored and randomly picks a new starting page. We will denote the probability that a surfer stays interested at step $t$ by a constant $d$, called the *damping factor*. Then the probability that the surfer gets bored at time $t$ is $1 - d$. The formulas in (14.1) then become

$$p_i(0) = \frac{1}{N} \qquad p_i(t+1) = d \sum_{j \in \text{In}(i)} \frac{p_j(t)}{|\text{Out}(j)|} + \frac{1-d}{N}. \tag{14.2}$$

### Matrix Form of the PageRank Algorithm

We can rewrite (14.2) as the matrix equation

$$\mathbf{p}(0) = \frac{1}{N}\mathbf{1} \qquad \mathbf{p}(t+1) = dK\mathbf{p}(t) + \frac{1-d}{N}\mathbf{1} \tag{14.3}$$

where $\mathbf{p}(t) = (p_1(t), p_2(t), \ldots, p_N(t))^T$, $\mathbf{1}$ is a vector of $N$ ones, and $K$ is defined by

$$K_{ij} = \begin{cases} \frac{1}{|\mathrm{Out}(j)|} & \text{if j links to i} \\ 0 & \text{otherwise.} \end{cases}$$

### Defining Page Rank

As given by the PageRank algorithm, the *rank* of page $i$ is

$$p_i = \lim_{t\to\infty} p_i(t).$$

For those familiar with Markov Chains, Equation 14.3 defines a Markov chain. Page ranks are simply the steady state of this Markov chain.

## Implementation in Python

The adjacency matrix $A$ of a directed graph has $A_{ij} = 1$ if there is an edge from node $i$ to node $j$, and $A_{ij} = 0$ otherwise. The adjacency matrix of the graph in Figure 14.1 is defined below. We use a code environment to describe $A$ so you can easily use this example to debug the problems in this lab.

```
A = np.array([[ 0,  0,  0,  0,  0,  0,  0,  1],
              [ 1,  0,  0,  0,  0,  0,  0,  0],
              [ 0,  0,  0,  0,  0,  0,  0,  0],
              [ 1,  0,  1,  0,  0,  0,  1,  0],
              [ 1,  0,  0,  0,  0,  1,  1,  0],
              [ 1,  0,  0,  0,  0,  0,  1,  0],
              [ 1,  0,  0,  0,  0,  0,  0,  0],
              [ 1,  0,  0,  0,  0,  0,  0,  0]])
```

**Problem 1.** Write a function that creates an adjacency matrix from a file. The function should accept a filename, and an integer $N$ that represents the number of nodes in the graph described by the datafile. Return the adjacency matrix as a `SciPy` sparse `dok_matrix`.
Hints:

1. The file `matrix.txt` included with this lab describes the matrix in Figure 14.1 and has the adjacency matrix `A` given above. You may use it to test your function.
2. You can open a file in Python using the `with` syntax. Then, you can iterate through the lines using a `for` loop. Here is an example.

```
# Open `matrix.txt' for read-only
with open('./matrix.txt', 'r') as myfile:
```

```
    for line in myfile:
        print line
```

3. Here is an example of how to process a line of the form in `datafile`.

```
>>> line = '0\t4\n'
# strip() removes trailing whitespace from a line.
# split() returns a list of the space-separated pieces of the line.
>>> line.strip().split()
['0', '4']
```

4. Rather than testing for lines of `matrix.txt` that contain comments, put all your string operations in a `try` block with an `except` block following.

**Note**

It makes sense to initialize $A$ as a sparse matrix, since $A$ is mostly zeros. To make the coding easier, throughout the rest of the lab the algorithms will be coded using non-sparse matrices. This means when using an adjacency matrix created in Problem 1, cast it `toarray()` when inputing it to test the other functions. In other words, you may say `test_calculateK(A.toarray(), N)`. Don't forget, however, that in a real-world sparse adjacency matrices are generally much more time-efficient than dense matrices.

The next step is to compute $K$ from (14.3). A good strategy for computing $K$ comes from writing

$$K = (D^{-1}A)^T$$

where $A$ is the adjacency matrix of the directed graph representing the internet and $D$ is a diagonal matrix with $D_{jj} = |\text{Out}(j)|$. Modify $A$ so that rows corresponding to sinks have all ones instead of all zeros. For Figure 14.2, the modified adjacency matrix is defined below.

```
Am = np.array([[ 0,  0,  0,  0,  0,  0,  0,  1],
               [ 1,  0,  0,  0,  0,  0,  0,  0],
               [ 1,  1,  1,  1,  1,  1,  1,  1],
               [ 1,  0,  1,  0,  0,  0,  1,  0],
               [ 1,  0,  0,  0,  0,  1,  1,  0],
               [ 1,  0,  0,  0,  0,  0,  1,  0],
               [ 1,  0,  0,  0,  0,  0,  0,  0],
               [ 1,  0,  0,  0,  0,  0,  0,  0]])
```

The matrix $D$ is easily obtained by summing the rows of $A$. Although $K = (D^{-1}A)^T$, it is better practice to only store the diagonal entries of $D$ as a vector, and then use array broadcasting to divide $A$ by $D$.

Notice that we need to transpose $D^{-1}A$ to get $K$. This is because $D^{-1}A$ is *row stochastic* (meaning that the rows sum to 1), but we need to multiply *column stochastic* matrices (where the columns sum to 1) To make $K$ be column stochastic, we have to take a transpose.

For Figure 14.2, the matrix $K$ is as follows.

```
K = np.array([[ 0   ,  1   ,  1./8,  1./3,  1./3,  1./2,  1   ,  1   ],
              [ 0   ,  0   ,  1./8,  0   ,  0   ,  0   ,  0   ,  0   ],
              [ 0   ,  0   ,  1./8,  1./3,  0   ,  0   ,  0   ,  0   ],
              [ 0   ,  0   ,  1./8,  0   ,  0   ,  0   ,  0   ,  0   ],
              [ 0   ,  0   ,  1./8,  0   ,  0   ,  0   ,  0   ,  0   ],
              [ 0   ,  0   ,  1./8,  0   ,  1./3,  0   ,  0   ,  0   ],
              [ 0   ,  0   ,  1./8,  1./3,  1./3,  1./2,  0   ,  0   ],
              [ 1   ,  0   ,  1./8,  0   ,  0   ,  0   ,  0   ,  0   ]])
```

**Problem 2.** Write a function that computes and returns the K matrix given an adjacency matrix.

1. Compute the diagonal matrix $D$.
2. Compute the modified adjacency matrix where the rows corresponding to sinks all have ones instead of zeros.
3. Compute $K$ using array broadcasting.

## Solving for the Page Ranks

There are several ways to solve for $\lim_{t\to\infty}\mathbf{p}(t)$.

### Algebraic Method

Again, for those familiar with Markov chains, one possibility is to assume the modified Markov chain has a steady state $\mathbf{p}$ and solve for it algebraically:

$$(I - dK)\mathbf{p} = \frac{1-d}{N}\mathbf{1}. \tag{14.4}$$

We can use SciPy's solver to find the page ranks of the network in Figure 14.2.

```
>>> from scipy import linalg as la
>>> I = np.eye(8)
>>> d = .85
>>> la.solve(I-d*K, ((1-d)/8)*np.ones(8))
array([ 0.43869288,  0.02171029,  0.02786154,  0.02171029,  0.02171029,
        0.02786154,  0.04585394,  0.39459924])
```

As expected, node 0 has the highest rank, approximately equal to .44. Node 7 has a higher rank than node 6, even though $\text{In}(7) = 1$ and $\text{In}(6) = 3$. This is because node 7's single in-edge comes from a node that has a very high rank (node 0).

### Iterative Method

Solving the system in (14.4) is feasible for our small working example, but this is not an efficient strategy for very large systems.

One option for large systems is an iterative method. Starting with a guess for $\mathbf{p}(0)$, we iterate on Equation (14.3) until $\|\mathbf{p}(t) - \mathbf{p}(t-1)\|$ is sufficiently small. At this point we assume we have reached the steady state.

**Problem 3.** Implement a function that uses the iterative method to find the steady state of the PageRank algorithm. You function should accept an adjacency matrix `A`, an integer `N` that defaults to `None`, the damping factor `d` that defaults to `0.85`, and a tolerance `tol` that defaults to `1E-5`. Return the approximation to the steady state as a float. When the argument `N` is not `None`, work with only the upper $N \times N$ portion of the array `adj`. Test your function against the example datafile that accompanies this lab.
Hints:

1. Try making your initial guess for $\mathbf{p}(0)$ a random vector.
2. NumPy can do unexpected things with the dimensions when performing matrix-vector multiplication. When debugging, check at each iteration that all arrays have the dimensions you expect.

### Eigenvalue Method

Another way to solve this problem is to make it into an eigenvalue problem. Let $E$ be an $N \times N$ matrix of ones; then $E\mathbf{p}(t) = \mathbf{1}$. Hence, the matrix equation (14.3) for $\mathbf{p}(t+1)$ becomes

$$\mathbf{p}(t+1) = \Big(dK + \frac{1-d}{N}E\Big)\mathbf{p}(t).$$

If we write $B = dK + \frac{1-d}{N}E$, this simplifies to $\mathbf{p}(t+1) = B\mathbf{p}(t)$. Thus, the steady state $\mathbf{p}(t)$ is an eigenvector of $B$ corresponding to the eigenvalue 1.

The columns of $B$ sum to 1, and the entries of $B$ are strictly positive (because the entries of $E$ are all positive). With these hypotheses, the Perron-Frobenius theorem says that 1 is the unique eigenvalue of B of largest magnitude, and the corresponding eigenvector is unique. In this case, the "iterative method" described above is just the power method for finding the eigenvector corresponding to a dominant eigenvalue, introduced in the lab on eigensolvers.

We can also compute $\mathbf{p}$ using eigenvalue solvers in SciPy.

**Problem 4.** Implement a function that uses the eigenvalue method to find the steady state of the PageRank algorithm. Your function should accept an adjacency matrix `A`, an integer `N` that defaults to `None`, and the damping factor `d` that defaults to `0.85`. Return the approximation to the steady state as a float.

## Application: Ranking Sports Teams

This ranking algorithm can be applied not only to webpages, but to any problem with a directed graph structure. One such application is ranking sports teams.

Suppose we have data about a collection of sports teams, including which teams played each other and who won each match. We can model this as a directed graph. Each node in the graph represents a team. An edge between two nodes points from the losing team to the winning team. If two teams never played each other, there is no edge between them. Wins and losses do not cancel out; if BYU and Boise played twice, and each team won once, then there is an edge from BYU to Boise and another edge from Boise to BYU.

To simplify our model, edges are not weighted. So if Duke ever beat Harvard, no matter whether they beat them once or 5 times, there is only one edge pointing from Harvard to Duke.

The key here is that edges tend to lead from worse teams to better teams. So by starting with some team and randomly following edges, we should end up visiting better teams more often. This is reminiscent of the PageRank algorithm! Given an appropriate dataset, we can use PageRank to estimate team rankings.

Note that in this scenario, the parameter $d$ no longer represents boredom. It allows us to jump randomly from one team to another, so it could represent a surprise upset, or the random outcome of a game between two teams who have never played each other.

**Problem 5.** By applying the PageRank algorithm to win-loss data from the 2013 NCAA basketball season, produce a comparative ranking of the teams.

1. The file `ncaa2013.csv` contains data on over 5000 basketball games. The first line is a header. After the header, each line represents a game and has the winning team followed by the losing team (there are no ties in basketball).

   Load this file and use it to create the adjacency matrix $A$, where $A_{ij} = 1$ if team $j$ beat team $i$. Make sure to ignore the header line. You will need some way of mapping from team names to the integers and vice versa.

2. Use the iterative method from Problem 3 with $d = 0.7$ to find the steady state. The steady-state solution is your vector of ranks.

3. Return the ranks sorted from largest to smallest, and the corresponding list of teams sorted from "best" to "worst".

Hints:

1. The code below may be helpful for processing the .csv file:

```
>>> with open('./ncaa2013.csv', 'r') as ncaafile:
>>>     ncaafile.readline() #reads and ignores the header line
>>>     for line in ncaafile:
>>>         teams = line.strip().split(',') #split on commas
>>>         print teams
>>> ['Middle Tenn St', 'Alabama St']
>>> ...
>>> ['Mississippi', 'Florida']
```

2. Before creating the adjacency matrix, you can get all the unique teams by running through all the matches once and adding every team to a set. Next, count the number of unique teams and initialize $A$ to be the right size. Try using dictionaries, lists, or both to map numbers to teams and teams to numbers and fill in $A$. There is more than one right way to do this.
3. The function `np.argsort()` will be useful for sorting the ranks and teams.
4. There should be 347 teams. PageRank should predict that the top five ranked teams are Duke, Butler, Louisville, Illinois, and Indiana (in that order). Use this to check your results.

## NetworkX: Python package for networks

The purpose of this section is not to give an introduction to NetworkX, but rather, to introduce you to just enough to be able to analyze the basic properties of a network and apply NetworkX's implementation of PageRank. NetworkX takes advantage of the sparse nature of these networks. Therefore, they are more efficient than the ones we have coded in this lab.

We will first run through the simple steps to initialize the graph defined in Figure 14.1. We will initialize this graph using the edges defined in `matrix.txt`. If we create an $n \times 2$ matrix of the edges of this graph, we would get,

```
>>> edges = array([[ 0,  7],
...                [ 1,  0],
...                [ 3,  0],
...                [ 3,  6],
...                [ 4,  0],
...                [ 4,  5],
...                [ 4,  6],
...                [ 5,  0],
...                [ 5,  6],
...                [ 6,  0],
...                [ 7,  0]])
```

.

We can now initialize a NetworkX graph using this array of edges.

```
>>> import networkx as nx
>>> G = nx.from_edgelist(edges, create_using=nx.DiGraph())
```

Now that we have initialized the `DiGraph` object, we can use all the analysis tools that come with NetworkX to gain further insight into the structure of the graph. Verify the following characteristics match the graph in Figure 14.1.

```
>>> G.in_degree()
{0: 6, 1: 0, 3: 0, 4: 0, 5: 1, 6: 3, 7: 1}

>>> G.out_degree()
{0: 1, 1: 1, 3: 2, 4: 3, 5: 2, 6: 1, 7: 1}
```

```
>>> G.in_edges(0)
[(1, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0)]

>>> G.out_edges(0)
[(0, 7)]
```

NetworkX also comes with a `pagerank()` function that can be used simply by passing the function your `DiGraph` object. Compare the results to the values calculated using our methods.

```
>>> nx.pagerank(G, alpha=0.85) # alpha is the dampening factor.
{0: 0.45323691210120065,
 1: 0.021428571428571432,
 3: 0.021428571428571432,
 4: 0.021428571428571432,
 5: 0.027500000000000004,
 6: 0.04829464285714287,
 7: 0.406682730755942}
```

## Application: Twitter Datasets

The SNAP graph library, located at `http://snap.stanford.edu/data/index.html`, provides a variety of medium sized data sets for public use. These datasets have to do with networks, including road systems, social networks, and online communities. There are some interesting resources here for those wanting to experiment further with the PageRank algorithm on different datasets.

**Problem 6 (Optional).** The `twitter_combined.txt` file contains a list of edges that represent a Twitter network. To protect the privacy of users, the data has been anonymized. Each number is an ID for a user. The users in the first column represent Twitter users that follow the users in the second column.

Using these edges,

1. Create a `DiGraph` object using all the edges described in `twitter_combined.txt`.
2. Calcualte the page ranks for this graph. The page ranks create a ranking of which users are the most “influential". Even though the results will just be numbers, remember that they represent actual Twitter users.
3. Analyze the in-degree and out-degree of the top 10 ranked users. What do you notice about the second-highest ranked user? Why would this user be ranked so high? HINT: Use `G.in_edges()` and `G.out_edges()` to gain further insight into this result.
4. Return the top 10 most influential users and their scores