Implementation of the PageRank Algortihm

notes by
Cesar O. Aguilar[1]

1. Power Method Review

Let $\mathbf{G}$ denote the Google matrix. The practical implementation of the PageRank algorithm is best handled with the Power Method which, starting with an initial vector $\mathbf{x}_0$, requires us to perform the iteration

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{G}\mathbf{x}_0 \\ \mathbf{x}_2 &= \mathbf{G}\mathbf{x}_1 \\ \mathbf{x}_3 &= \mathbf{G}\mathbf{x}_2 \\ \vdots &= \vdots \\ \mathbf{x}_N &= \mathbf{G}\mathbf{x}_{N-1} \end{aligned}$$

where N is sufficiently large (usually $50 \leq N \leq 100$) so that $\mathbf{x}_N$ is a good approximation to the PageRank eigenvector $\mathbf{x}^*$:

$$\mathbf{x}_N \approx \mathbf{x}^*$$

We therefore need to perform many matrix-vector multiplications which can be a time consuming operation when $\mathbf{G}$ is a large matrix because *every* entry of $\mathbf{G}$ is non-zero. A direct approach to compute $\mathbf{G}\mathbf{x}$ would require n^2 multiplications if $\mathbf{G}$ is a $n \times n$ matrix. Storing $\mathbf{G}$ is also problematic because we would need $8n^2$ bytes of memory to store $\mathbf{G}$; if say $n = 3,000,000$ then $8n^2$ bytes $= 72000$ GB $= 72$ TB . Instead, we will see how $\mathbf{G}$ can be decomposed into matrices that are easy to store and/or easy to compute with.

2. Notation

We will use notation that is consistent with Matlab. For example, for any matrix $\mathbf{A}$, the ith column of $\mathbf{A}$ is denoted by $\mathbf{A}(:, i)$, and the jth row of $\mathbf{A}$ will be denoted by $\mathbf{A}(j, :)$. By $\mathbf{J}$ we denote the all 1's matrix of appropriate size. For example, the $\mathbf{J}$ matrix of size 4×4 is

$$\mathbf{J} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

[1] Assistant Professor, Department of Mathematics, SUNY Geneseo

By $\mathbf{e}$ we denote the all 1's vector of appropriate size. For example, in $\mathbb{R}^4$ the $\mathbf{e}$ vector is $\mathbf{e} = (1, 1, 1, 1)$. It is not hard to see that

$$\mathbf{J} = \mathbf{e} \cdot \mathbf{e}^T$$

where as usual $\mathbf{e}^T$ denotes the transpose of the vector $\mathbf{e}$. For example, in $\mathbb{R}^4$:

$$\mathbf{e} \cdot \mathbf{e}^T = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} = \mathbf{J}$$

We will also use some Matlab notation to denote operations on vectors. For example, if $\mathbf{x} = (x_1, x_2, \dots, x_n)$ then $\texttt{sum}(\mathbf{x}) = \sum_{i=1}^n x_i$ is the sum of the entries of $\mathbf{x}$. Notice that

$$\texttt{sum}(\mathbf{x}) = \mathbf{e}^T \mathbf{x} = \sum_{i=1}^{n} x_i$$

In Matlab, if $\mathbf{A}$ is a matrix then

$$\texttt{sum}(\mathbf{A}, 1) = \text{sum of the columns of } \mathbf{A}$$

$$\texttt{sum}(\mathbf{A}, 2) = \text{sum of the rows of } \mathbf{A}$$

3. Creating the Hyperlink Matrix

To create $\mathbf{G}$ we need to create the hyperlink matrix $\mathbf{H}$ which in turn requires the creation of the **adjacency matrix** $\mathbf{A}$ of the network. Consider the directed network shown in Figure 1.

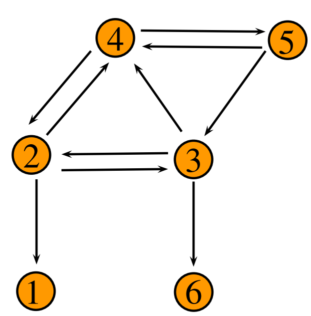

Figure 1: A tiny network

The adjacency matrix for this directed graph is

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

In the adjacency matrix, each column corresponds to the out-going links for the corresponding vertex. For example, vertex $i = 2$ links to vertices $\{1, 3, 4\}$ and thus column $i = 2$ of $\mathbf{A}$ has 1's in positions $\{1, 3, 4\}$ and 0's elsewhere. The rows of $\mathbf{A}$ correspond to the in-links for the corresponding vertex. For example, the non-zero entries of row $j = 3$ are $\{2, 5\}$ because vertex $j = 3$ has in-links from vertices $\{2, 5\}$.

To create the hyperlink matrix $\mathbf{H}$, each non-zero column of $\mathbf{A}$ is normalized so that its sum is equal to 1. For example, if column $\mathbf{A}(:, i)$ is non-zero, we compute $d_i = \texttt{sum}(\mathbf{A}(:, i))$ and then the ith column of $\mathbf{H}$ is

$$\mathbf{H}(:, i) = \frac{1}{d_i}\mathbf{A}(:, i).$$

For the tiny network above, the hyperlink matrix is

$$\mathbf{H} = \begin{bmatrix} 0 & \frac{1}{3} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 & 0 & 0 \end{bmatrix}$$

Here is Matlab code to compute $\mathbf{H}$ once $\mathbf{A}$ is known:

```
H = zeros(n, n);       % allocate memory for H
d = sum(A, 1);         % sums the columns of A, d is a row vector
for i = 1:n
    if d(i) > 0
```

```
        H(:, i) = 1/d(i) * A(:, i);
    end
end
```

Because the columns of $\mathbf{A}$ are the out-going links of the corresponding vertex, the vector $\mathbf{d}$ stores the out-going degrees of each vertex. If i is a dangling node (has no out-going links) then $\mathbf{d}(i) = 0$.

4. Creating the Google Matrix

Recall that the Google matrix was defined to be

$$\mathbf{G} = \alpha\overline{\mathbf{H}} + (1-\alpha)\tfrac{1}{n}\mathbf{J} \tag{1}$$

where $\alpha = 0.85$. The matrix $\overline{\mathbf{H}}$ is obtained by replacing each zero column of $\mathbf{H}$ with the vector $\frac{1}{n}\mathbf{e}$. This modification of $\mathbf{H}$ fixes the **dangling node** problem (recall that a dangling node corresponds to a zero column of $\mathbf{H}$). It is straightforward to compute $\overline{\mathbf{H}}$ using a `for` loop but we do not actually want to compute or store $\overline{\mathbf{H}}$. Instead we want to decompose $\overline{\mathbf{H}}$ in the form

$$\overline{\mathbf{H}} = \mathbf{H} + \mathbf{X}$$

where $\mathbf{X}$ has two types of columns: if vertex i is a dangling node then $\mathbf{X}(:,i) = \frac{1}{n}\mathbf{e}$ and, if i is not a dangling node then $\mathbf{X}(:,i) = \mathbf{0}$. To see how $\mathbf{X}$ can be computed, first define the *dangling node vector* $\mathbf{a}$ as:

$$\mathbf{a}(i) = \begin{cases} 1, & \text{if } i \text{ is a dangling node} \\ 0, & \text{otherwise.} \end{cases}$$

Hence, $\mathbf{a}$ identifies which nodes are dangling nodes. In Matlab, we can compute $\mathbf{a}$ as:

```
a = (d == 0);
```

For example, if $\mathbf{d} = (3,8,0,4,11,0,9,0) \in \mathbb{R}^8$ then Matlab would return the vector

```
a = [0, 0, 1, 0, 0, 1, 0, 1];
```

because the nodes $\{3,6,8\}$ are the dangling nodes. Then

$$\mathbf{X} = \tfrac{1}{n}\mathbf{e} \cdot \mathbf{a}^T$$

and therefore

$$\overline{\mathbf{H}} = \mathbf{H} + \tfrac{1}{n}\mathbf{e} \cdot \mathbf{a}^T \tag{2}$$

For example, for the hyperlink matrix $\mathbf{H}$ of the tiny network above,

$$\overline{\mathbf{H}} = \begin{bmatrix} \frac{1}{6} & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{6} \\ \frac{1}{6} & 0 & \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{6} \\ \frac{1}{6} & \frac{1}{3} & 0 & 0 & \frac{1}{2} & \frac{1}{6} \\ \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{2} & \frac{1}{6} \\ \frac{1}{6} & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{6} \\ \frac{1}{6} & 0 & \frac{1}{3} & 0 & 0 & \frac{1}{6} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & \frac{1}{3} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{H}} + \underbrace{\begin{bmatrix} \frac{1}{6} & 0 & 0 & 0 & 0 & \frac{1}{6} \\ \frac{1}{6} & 0 & 0 & 0 & 0 & \frac{1}{6} \\ \frac{1}{6} & 0 & 0 & 0 & 0 & \frac{1}{6} \\ \frac{1}{6} & 0 & 0 & 0 & 0 & \frac{1}{6} \\ \frac{1}{6} & 0 & 0 & 0 & 0 & \frac{1}{6} \\ \frac{1}{6} & 0 & 0 & 0 & 0 & \frac{1}{6} \end{bmatrix}}_{\frac{1}{n}\mathbf{e}\cdot\mathbf{a}^T}$$

From the definition (1) of the Google matrix, equation (2), and the fact that $\mathbf{J} = \mathbf{e} \cdot \mathbf{e}^T$ we have that

$$\begin{aligned} \mathbf{G} &= \alpha\overline{\mathbf{H}} + (1-\alpha)\tfrac{1}{n}\mathbf{J} \\ &= \alpha(\mathbf{H} + \tfrac{1}{n}\mathbf{e}\cdot\mathbf{a}^T) + (1-\alpha)\tfrac{1}{n}\mathbf{e}\cdot\mathbf{e}^T \\ &= \alpha\mathbf{H} + \tfrac{\alpha}{n}\mathbf{e}\cdot\mathbf{a}^T + (1-\alpha)\tfrac{1}{n}\mathbf{e}\cdot\mathbf{e}^T \\ &= \alpha\mathbf{H} + \mathbf{e}\cdot\left(\tfrac{\alpha}{n}\mathbf{a}^T + (1-\alpha)\tfrac{1}{n}\mathbf{e}^T\right) \end{aligned}$$

Then,

$$\mathbf{Gx} = \alpha\mathbf{Hx} + \left(\tfrac{\alpha}{n}\mathbf{a}^T\mathbf{x} + (1-\alpha)\tfrac{1}{n}\mathbf{e}^T\mathbf{x}\right)\mathbf{e}$$

Therefore, to compute $\mathbf{Gx}$ all we need to compute is the vector $\mathbf{Hx}$, and the two scalar quantities $\mathbf{a}^T\mathbf{x}$ and $\mathbf{e}^T\mathbf{x}$, and then add the terms using the above formula for $\mathbf{Gx}$.

5. Iteration Loop in Power Method

If you have taken a course in numerical analysis, you will have noticed that the Power Method is simply just fixed-point iteration. To decide on how many iteration steps to perform, successive approximations $\mathbf{x}_{k+1}$ and $\mathbf{x}_k$ are compared by computing the norm of their differences:

$$\texttt{norm}(\mathbf{x}_{k+1} - \mathbf{x}_k) = \|\mathbf{x}_{k+1} - \mathbf{x}_k\|$$

If $\|\mathbf{x}_{k+1} - \mathbf{x}_k\| < \varepsilon$ where $\varepsilon > 0$ is some chosen small quantity (for example $\varepsilon = 1 \times 10^{-8}$), then $\mathbf{x}_{k+1} \approx \mathbf{x}_k$ and thus $\mathbf{x}_{k+1}$ will be a good approximation to the PageRank vector $\mathbf{x}^*$.

Thus, when the condition

$$\|\mathbf{x}_{k+1} - \mathbf{x}_k\| < \varepsilon$$

is satisfied, we should exit the Power Method iteration and return the vector $\mathbf{x}_{k+1}$. Below we present pseudocode implementing the Power Method.

Algorithm: Power method to compute PageRank vector

INPUT: $\mathbf{H}$, $\mathbf{a}$, $\varepsilon > 0$, N_{max}

OUTPUT: Approximate PageRank vector

$\mathbf{x}_{\text{old}} = \frac{1}{n}\mathbf{e}$
for $k = 1, 2, \ldots, N_{\text{max}}$
  $\beta = \frac{\alpha}{n}\mathbf{a}^T\mathbf{x}_{\text{old}} + \frac{(1-\alpha)}{n}\mathbf{e}^T\mathbf{x}_{\text{old}}$
  $\mathbf{x}_{\text{new}} = \alpha\mathbf{H} \cdot \mathbf{x}_{\text{old}} + \beta\mathbf{e}$
  if $\|\mathbf{x}_{\text{new}} - \mathbf{x}_{\text{old}}\| < \varepsilon$
    break
  else
    $\mathbf{x}_{\text{old}} = \mathbf{x}_{\text{new}}$
  end
end

6. The Data

6.1. 2016 College Football Bowl Subdivision

The 2016 College Football Bowl Subdivision regular and post-reason results are stored in the text file `2016-fbs-games.txt`. In this text file, each line represents a game, and the format of each line is

```
teamA, scoreA, location, teamB, scoreB
```

where `location` is either `vs` or `at`. For example, the line

```
Alabama,34,vs,Kentucky,6
```

means that Alabama played at home versus Kentucky and Alabama scored 34 points and Kentucky scored 6, whereas the line

```
Arizona,24,at,UCLA,45
```

means that Arizona played at UCLA and Arizona scored 24 and UCLA scored 45. The Matlab code to read the entire contents of this file is

```
fid = fopen(‘2016-fbs-games.txt‘,‘r‘);
C = textscan(fid, ‘%s %d %s %s %d‘,‘Delimiter‘, ‘ , ‘);
fclose(fid);
```

The variable `C` is a cell array in Matlab containing 5 sub-cells:

```
C = { C{1}, C{2}, C{3}, C{4}, C{5} }
```

A cell in Matlab is a list whose elements are not necessarily of the same type. The cells of `C` are:

Cell	Description of contents
`C{1}`	a cell containing all the `teamA` names
`C{2}`	a vector containing all the `scoreA` points
`C{3}`	a cell containing the locations (either `vs` or `at`)
`C{4}`	a cell containing all the `teamB` names
`C{5}`	a vector containing all the `scoreB` points

Table 1: Description of the cells in `C`

Thus, the data of line 200 of the text file can be accessed using

```
C{1}{200}, C{2}(200), C{3}{200}, C{4}{200}, C{5}(200)
```

Notice that we used parenthesis to access the data of `C{2}` and `C{5}` because they are vectors, while for the cells `C{1}`, `C{3}`, `C{4}`, we used braces { } to access the data.

A complete list of all the team names are stored in the comma separated file `2016-fbs-teams.txt` and can be loaded with the command

```
teams = textscan(fid,'%s','Delimiter',',');
```

This creates the cell `teams` and the name of the ith team is accessed by typing `teams{1}{i}`

6.2. The 2007 Wikipedia Site

The adjacency matrix **A** of the 2007 Wikipedia website is stored in the file

```
wikipedia-adjacency-matrix-2007.mat
```

To load this file in Matlab, you type

```
load('wikipedia-adjacency-matrix-2007.mat');
```

In Matlab, if you type

```
>> whos
```

you will see a list of variables, one of which should be **A**. You will see that **A** is of size $n = 3357835$, needs 673 MB of storage, and that it has the attribute `sparse`. The matrix **A** is stored as a `sparse` matrix because it contains mostly zeros. In the `sparse` format, only the locations and the corresponding values of the non-zero entries of **A** are stored. For example, suppose that

$$\mathbf{A} = \begin{bmatrix} 0 & a_{1,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{2,4} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{4,2} & a_{4,3} & 0 & a_{4,5} & 0 \\ 0 & 0 & 0 & a_{5,4} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Then in sparse format, **A** is stored as the list

$$\begin{array}{cc} (1,2) & a_{1,2} \\ (4,2) & a_{4,2} \\ (4,3) & a_{4,3} \\ (2,4) & a_{2,4} \\ (5,4) & a_{5,4} \\ (4,5) & a_{4,5} \end{array}$$

To create the matrix **A** above in Matlab as a sparse matrix we type

```
A = sparse([1,4,4,2,5,4],[2,2,3,4,4,5],[a12, a42, a43, a24, a54, a45],6,6);
```

Notice that the first argument to `sparse` correspond to the rows, the second argument corresponds to the columns, the third is the list of the actual numerical values at the row-column locations specified by the first and second arguments, and the last two arguments specify the size of the matrix, in this case 6×6. The general call to define an $n \times m$ `sparse` matrix in Matlab is

```
sparse(row_indices, column_indices, values, n, m);
```

7. Useful Matlab Functions

Below is a list of Matlab functions that you may find useful.

- `[bool, i] = ismember(x, C)` : if `x` is a member of `C` then `bool = True` and `i` is the location (or index) of `x` in `C`. For example, suppose that `C` is the cell

  `C = {'Duke', 'Notre Dame', 'Alabama', 'USC'}`.

  Then

  `[bool, i] = ismember('Alabama', C)`

  returns `bool = True` and `i = 3`.

- `[`$\mathbf{x}_{\texttt{sorted}}$`, J] = sort(`$\mathbf{x}$, 'descend'`)`: sorts the entries of the vector $\mathbf{x}$ in descending order and saves the sorted vector in $\mathbf{x}_{\text{sorted}}$ and saves the sorted indices in the variable `J`. As example, suppose that

  $$\mathbf{x} = (1.3, 0.3, 5.6, 8.9, 2.3)$$

  Then $\mathbf{x}_{\text{sorted}} = (8.9, 5.6, 2.3, 1.3, 0.3)$ and `J` = (4, 3, 5, 1, 2). If all you want is the index vector `J` then you can type `[~, J]`

- `find(`$\mathbf{d}$` == 0)`: returns the indices where the vector $\mathbf{d}$ is zero

- `find(`$\mathbf{d}$` ~= 0)`: returns the indices where the vector $\mathbf{d}$ is non-zero

- `ones(n,1)`: creates the all 1's vector of size n, that is, $\mathbf{e}$ =`ones(n, 1)`

- `numel(`$\mathbf{y}$`)`: returns the number of entries in $\mathbf{y}$ regardless of whether $\mathbf{y}$ is a row or column vector, also works if $\mathbf{y}$ is a cell

- `size(`$\mathbf{A}, 1$`)`: returns the row dimension of $\mathbf{A}$

- `size(`$\mathbf{A}, 2$`)`: returns the column dimension of $\mathbf{A}$

References

[1] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107 – 117, 1998.