Power Iteration and PageRank

QEA

Spring 2016

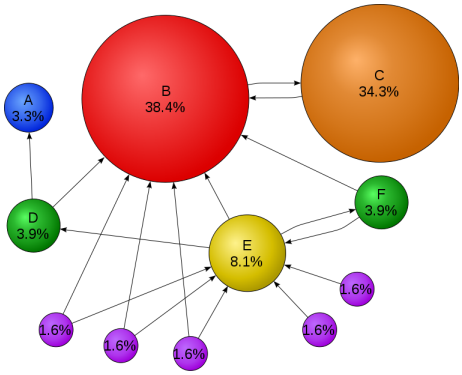

Figure 1: The page linking network analysis used by Google to rank pages in a search uses a power iteration method. Credit: from Wikipedia

The Power Iteration Method

Introduction

We have been working with matrices as an expression of a linear transformation. A vector $\vec{v}_1$ is transformed to a vector $\mathbf{v_2}$ by the matrix $\mathbf{A}$.

$$\mathbf{A v_1} = \mathbf{v_2} \tag{1}$$

If the vector $\mathbf{v_2}$ is simply a multiple of $\mathbf{v_1}$ by some scalar factor λ ($\mathbf{v_2} = \lambda \mathbf{v_1}$) then $\mathbf{v_1}$ is considered to be an eigenvector of the matrix $\mathbf{A}$ with an eigenvalue of λ. A matrix of rank n will have n eigenvalues and associated eigenvectors. These eigenvectors capture very important states of the system which is characterized by our matrix: stable 'stationary' states which are not transformed into any other states by the action of our matrix.

The method you have learned for finding the eigenvalues of a matrix is to solve the characteristic equation. However, for high-rank matrices, this problem involves finding the roots of a high order polynomial, which is in itself often an intractably difficult problem. In the Google PageRank paper, you were briefly introduced to the concept of the power iteration method: here we will walk through this in more detail.

Method

The power iteration method is a way of finding the principal eigenvector (the eigenvector with the largest eigenvalue) of a matrix. While this method will only give you one eigenvector (and you can calculate its eigenvalue easily from that) this is often sufficient to give you a lot of important information about your problem.

The power iteration method begins with a **random guess for the eigenvector**. So long as the matrix has a principal eigenvector (there is only a single eigenvalue which has the largest absolute magnitude) and this guess has some projection along the principal eigenvector, the power iteration method will converge to the principal eigenvector.

Let's consider that our guess for the eigenvector is vector $\mathbf{c}$, while the actual (unknown) eigenvectors of our matrix $\mathbf{A}$ are the vectors $\mathbf{v_i}$ with associated eigenvalues λ_i. Because the eigenvectors span the vector space of our problem, as you learned in the weekend exercises,

we can decompose our guess vector $\mathbf{c}$ into a linear combination of the actual eigenvectors:

$$\mathbf{c} = n_1\mathbf{v_1} + n_2\mathbf{v_2} + n_3\mathbf{v_3} + ... \tag{2}$$

for some set of scalar values n_i.

Now let's consider operating on vector $\mathbf{c}$ with our matrix $\mathbf{A}$:

$$\mathbf{Ac} = n_1\lambda_1\mathbf{v_1} + n_2\lambda_2\mathbf{v_2} + n_3\lambda_3\mathbf{v_3} + ... \tag{3}$$

Since we can distribute $\mathbf{A}$ across the linear decomposition of $\mathbf{c}$, we can express the resulting vector as a combination of the eigenvectors as shown. Now, let's operate on the vector $\mathbf{c}$ with matrix $\mathbf{A}$ repeatedly k times!

$$\mathbf{A}^k\mathbf{c} = n_1\lambda_1^k\mathbf{v_1} + n_2\lambda_2^k\mathbf{v_2} + n_3\lambda_3^k\mathbf{v_3} + ... \tag{4}$$

No sweat: we just end up with k powers of each eigenvalue multiplying each eigenvector. Now, let's assume that vector $\mathbf{v_1}$ is our principal eigenvector, and hence $\lambda_1 > \lambda_i$ for all other i, and let's factor out λ_1^k:

$$\mathbf{A}^k\mathbf{c} = \lambda_1^k(n_1\mathbf{v_1} + n_2(\lambda_2/\lambda_1)^k\mathbf{v_2} + n_3(\lambda_3/\lambda_1)^k\mathbf{v_3} + ...) \tag{5}$$

Now since every other eigenvalue λ_i is smaller than the principal eigenvalue λ_1, all the ratios λ_i/λ_1 are less than 1, and when taken to a large power k, these ratios will rapidly approach zero, allowing us to write, for large k:

$$\mathbf{A}^k\mathbf{c} \approx \lambda_1^k n_1\mathbf{v_1} \tag{6}$$

In other words, if we operate on our guess vector $\mathbf{c}$ repeatedly with our matrix $\mathbf{A}$, the answer will approach a scalar multiple of our principle eigenvector $\mathbf{v_1}$. The normalized eigenvector can be found then by just normalizing this result. Note that in the in-class exercise from last Thursday, this is exactly what you should have observed with the shear matrix iteration example!

Exercises

Consider the matrix

$$\mathbf{A} = \begin{pmatrix} -2 & 8 \\ 4 & 12 \end{pmatrix}$$

1. By hand, find the eigenvalues of this matrix using the characteristic equation and subsequently the eigenvectors.
2. Choose a random (two-component) vector $\mathbf{c}$. Use the method outlined in the overnight for linear decomposition to write your vector $\mathbf{c}$ as a linear combination of the eigenvectors. Does your random vector have a non-zero projection along the principle eigenvector? (If not, choose another one!)

3. Now we will try our hand at the power iteration method for your vector $\mathbf{c}$

   (a) Find $\mathbf{Ac}$ and normalize it. (Do this one by hand!)

   (b) Find $\mathbf{A}^2\mathbf{c}$ and normalize it. (Ok, you can use Matlab for the rest!)

   (c) Find $\mathbf{A}^3\mathbf{c}$ and normalize it.

   (d) Find $\mathbf{A}^4\mathbf{c}$ and normalize it.

   (e) Find $\mathbf{A}^5\mathbf{c}$ and normalize it.

   (f) Etc.

   How many iterations does it take before the variation from one iteration to the next is less than 0.01 per component (this is a convergence test!)? How does your result for the primary eigenvector from this calculation compare with your answer calculated by hand from above?

4. Create a code in Matlab which automates the power iteration method! Begin with having Matlab select a random vector, and make sure to have you algorithm text for convergence!

What about the rest of the eigenvectors?

The power iteration method clearly is not the end of the story of numerical methods in linear algebra. Extensions of this method have been developed to find the rest of the eigenvectors and eigenvalues in an efficient manner, but they are substantially more complex. You can look up information on the Lanczos and Arnoldi's Algorithms if you would like to learn more.

PageRank

Over the weekend, you should have read through the paper on the Google PageRank algorithm. The paper provides much more mathematical justification for what is happening. Here were will just walk through the key points and let you work through some examples!

PageRank is based on the idea that we can model a person surfing the web as taking a random walk through a series of interconnected pages. A random walk is a standard probability concept in which, for each timestep, a walker (or in this case a surfer) takes a step in any of the directions open to them with equal probability.

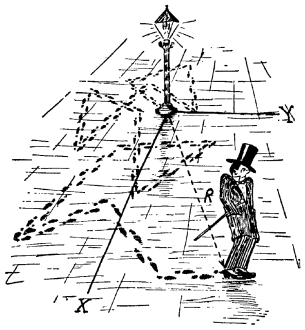

Figure 2: The classic drunkard's walk problem. Credit: from www.sos.siena.edu

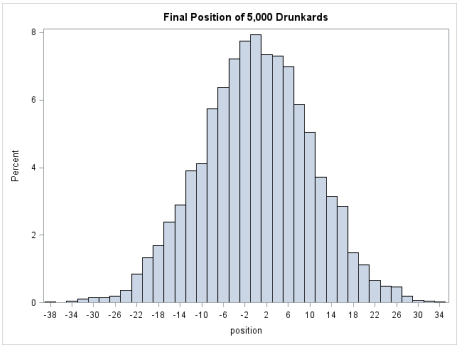

Figure 3: Probability distribution for the drunkard's walk. Credit: from www.statblogs.com

The Drunkard's Walk

The classic version of this is what is referred to as the 'drunkard's walk' problem. In this problem, at time $t = 0$, a very inebriated person is learning against a lamppost, and starts to try to walk home. However, being as this person is very drunk, at any time they are just as likely to take a step in any direction. If we follow the path of a single drunkard's walk, this path will tell us very little. However, if we iterate this experiment, and look at the average over the paths of a very large number of drunkards, we start to build up a 'probability distribution', which can tell us a lot about the expected path of our tipsy friend. For example, on average, where is the drunkard? On average, right where he started: back at the lamppost! However, as time increases, there is an increasingly broad range of space where the drunkard may be found.

A probability distribution is just what it sounds like: a function which tells us what the probability is for our system to end up in any of the accessible states. This function can take many forms, but in order for it to be a probability, it needs to meet one very important condition. In our drunkard's walk problem, the one thing we can say with surety is the the drunkard is definitely SOMEWHERE, with probability 1. So the integral (or sum) of our probability distribution function over all of our possible states must be one. We have already encountered probability distributions of a sort! Remember back when we were histogramming data? If we had normalized those distributions by dividing out the sum over the entire distribution, the resulting histogram would have shown the probability of any given data point lying in each bin of the histogram!

Let's consider an example of a drunkard's walk problem: a drunkard's walk on a one-dimensional number line with ten positions. We can indicate the position probability distribution of our drunkards with a vector with ten elements and a norm of 1, indicating the population probability at each point along the number line. For example, with any number of drunkards starting with equal numbers as positions 5 and 6, the state vector would be: $(0, 0, 0, 0, 0.5, 0.5, 0, 0, 0, 0)$. The new probability distribution after each timestep can be indicated by acting on this state vector (ok Dhash: on the transpose of this state vector) with a transition probability matrix which captures the probability of moving from one element on the number line to any other element on the number line.

5. Create the transition matrix for our ten-position linear drunk walk. The rules are: the drunks never stays still, they always move to an adjacent position on the numberline at each timestep. There is an equal probability they will move to any of the available adjacent

sites.

6. A drunkard at any position in the system, must always be found somewhere else in the system after each application of our transition matrix. What does this mean for the sums over the columns of our transition matrix? Does your transition matrix meet this criteria?

7. Act on our starting vector from above $(0, 0, 0, 0, 0.5, 0.5, 0, 0, 0, 0)$ with this matrix. Do the drunks go where you expect?

8. Now, iterate the action of this matrix on our starting vector. Plot the position state vector after each iteration. What do you see? What happens to the variation of the state vector from iteration to iteration after many iterations? Can you relate this to what we just did with the power iteration method?

The matrix we have just created is an example of what is known as a Markov Chain: a concept which shows up repeatedly in many contexts. The probability distribution (eigenvector) we found after multiple iterations is a stationary state of the system. Just like in the power iteration method, this should converge for *most* starting vectors.

(Note, due to some issues with the small size and discretized nature of this example, it has some peculiarities if we play around with different starting vectors. If you are so inclined, explore this!).

A Random Walk through the Web

From our drunkard's walk, it is a small step to the PageRank algorithm! There are a few differences:

- Just like above, we create the transition probability matrix by allowing equal probability for the web surfer to take any of the outgoing links from a given site. If a site has no links, they stay at that site with probability 1. However, since the web does not live on a one-dimensional space, the geometry of the transition matrix will look very different!

- The PageRank algorithm incorporates a 15% chance that the surfer will randomly jump to any other page in the web. If our transition probability matrix is $\mathbf{A}$ then the composite matrix including the random jumps is $0.85 * \mathbf{A} + 0.15 * \mathbf{B}/N$ where $\mathbf{B}$ is a matrix of ones with the same size as our transition probability matrix, and N is the total number of sites in our web. Note, this combination should keep all the total probabilities in our system equal to one.

- Google uses the steady-state probability distribution that results from this matrix to assign a rank to each page in the web, and orders search results based on this!

  Ok, try this out on your own little 'web'!

9. On paper, create your own little version of the internet: use about 6-10 'websites' and indicate with arrows which sites connect with which other sites. For starters, I recommend creating a highly interconnected 'web': the paper you read discusses some of the difficulties which arise from disconnected subsystems.

10. Capture your 'web' in a transition matrix! Make sure to check that your transition probabilities out of each site (the columns in your transition matrix) sum to one.

11. Create the PageRank matrix for your web by including the 15% chance of a random jump. Make sure your probabilities still sum to one!

12. Choose a starting configuration for your random surfer population distribution and create the appropriate state vector.

13. Iterate application of your PageRank matrix to your starting state vector. Does it converge? To what distribution? Which 'sites' in your web would be ranked highest in a Google search?