Practica 1 – Pagerank

Miguel Garcia Lafuente

Índice de contenido

A. Calcular la matriz G de pagerank del grafo....3
B. Codificar la rutina del método de la potencia. Reescalar una matriz....7
    Reescalar una matriz....8
    Ejecutar el metodo de la potencia....8
    Teorema de Perron-Frobenius....9
C. Calcular el pagerank del grafo....12

Todo el codigo de esta practica se puede encotrar en el siguiente link para que sea mas comodo el acceso a todo: https://gist.github.com/rockneurotiko/35b1061a1fd182e162cc

A. Calcular la matriz G de pagerank del grafo

Para este apartado, como para el resto de la practica, he intentado hacerlo muy generico, haciendo funciones que hagan solamente una funcion especifica, para mejorar la reutilizacion posterior.

Empezamos con una funcion auxiliar, la cual le das un vector i, un vector j y el tamaño n, y crea la matrix sparse con 1 en las coincidencias, la transforma a matriz total, calcula Nj (suma por columnas) y a partir de Nj, calcula dj, usando comprehension de listas, basicamente donde algo distinto de cero, pone un 0, y si hay un cero, pone un 1. Esta funcion devuelve todos los valores calculados.

```
1 function create_inital_values(i,j,n)
2     Cs = sparse(j,i,1.0,n,n)
3     C = full(Cs)
4     Nj = sum(C,1)
5     dj = [x == 0 ? 1:0 for x=Nj]
6     return Cs,C,Nj,dj
7 end
```

A partir de los resultados, calculamos S, donde, en una primera iteracion, he hecho este codigo, probablemente se pueda mejorar.

Inicializa el nuevo vector, esto se hace para asignar tamaño, reservar el espacio y asignar el tipo.

Recorre el vector Nj (suma por columnas, se podria pedir como parametro), guardando el valor y el indice de la columna (enumerate de vuelve la tupla (indice, valor) de un iterador)

Si la suma es distinta de cero, entonces substituimos la columna por esa columna entre la suma. Si la suma es cero, en todas las posiciones de la columna ponemos 1/n.

```
 1 function create_S(C,n)
 2     S = zeros(Float64,n,n) # Inicializar para asignar y reservar
 3     for (col,s) in enumerate(sum(C,1))
 4         if s != 0
 5             S[:,col] = C[:,col]/s
 6         else
 7             S[:,col] = 1/n
 8         end
 9     end
10     return S
11 end
```

Para crear G haremos uso de una funcion tambie.

Esta funcion genera la matriz G a partir de un valor alfa, una matriz (C) y su tamaño.

Inicializa la matriz, y recorre los indices de columnas. Para cada columna, le aplica la funcion por columnas y substituye.

```
function create_G(alfa::Float64,C,n)
    G = zeros(Float64,n,n)
    for col in 1:n
        G[:,col] = (alfa * C[:,col]) + ((1-alfa) * 1/n) # Applying the formula
    end
    return G
end
```

Vamos a comprobar las funciones creadas.

Inicializamos los valores:

```
i=[1, 1, 1, 2, 2, 3, 3, 4, 4, 6, 6, 7, 7]
j=[2, 4, 5, 3, 7, 4, 6, 2, 7, 7, 5, 4, 2]
n = 7
```

Aplicamos todas las funciones para conseguir los valores.

```
Cs,C,Nj,dj = create_inital_values(i,j,n)
S = create_S(C,n)
alpha = 0.85
G = create_G(alpha,S,n)
```

Los valores conseguidos:

```
julia> Cs
7x7 sparse matrix with 13 Float64 entries:
        [2, 1]  =  1.0
        [4, 1]  =  1.0
        [5, 1]  =  1.0
        [3, 2]  =  1.0
        [7, 2]  =  1.0
        [4, 3]  =  1.0
        [6, 3]  =  1.0
        [2, 4]  =  1.0
        [7, 4]  =  1.0
        [5, 6]  =  1.0
        [7, 6]  =  1.0
        [2, 7]  =  1.0
        [4, 7]  =  1.0

julia> C
```

```
7x7 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0
 1.0  0.0  0.0  1.0  0.0  0.0  1.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0
 1.0  0.0  1.0  0.0  0.0  0.0  1.0
 1.0  0.0  0.0  0.0  0.0  1.0  0.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0
 0.0  1.0  0.0  1.0  0.0  1.0  0.0

julia> Nj
1x7 Array{Float64,2}:
 3.0  2.0  2.0  2.0  0.0  2.0  2.0

julia> dj
1x7 Array{Int64,2}:
 0  0  0  0  1  0  0

julia> S
7x7 Array{Float64,2}:
 0.0       0.0  0.0  0.0  0.142857  0.0  0.0
 0.333333  0.0  0.0  0.5  0.142857  0.0  0.5
 0.0       0.5  0.0  0.0  0.142857  0.0  0.0
 0.333333  0.0  0.5  0.0  0.142857  0.0  0.5
 0.333333  0.0  0.0  0.0  0.142857  0.5  0.0
 0.0       0.0  0.5  0.0  0.142857  0.0  0.0
 0.0       0.5  0.0  0.5  0.142857  0.5  0.0

julia> G
7x7 Array{Float64,2}:
 0.0214286  0.0214286  0.0214286  0.0214286  0.142857  0.0214286  0.0214286
 0.304762   0.0214286  0.0214286  0.446429   0.142857  0.0214286  0.446429
 0.0214286  0.446429   0.0214286  0.0214286  0.142857  0.0214286  0.0214286
 0.304762   0.0214286  0.446429   0.0214286  0.142857  0.0214286  0.446429
 0.304762   0.0214286  0.0214286  0.0214286  0.142857  0.446429   0.0214286
 0.0214286  0.0214286  0.446429   0.0214286  0.142857  0.0214286  0.0214286
 0.0214286  0.446429   0.0214286  0.446429   0.142857  0.446429   0.0214286
```

Respondiendo a las preguntas planteadas:

1. Viendo Nj se puede observar que no todos los nodos tienen salidas, el nodo 5 no tiene ninguna salida.

2. Viendo S, se ve que ningun valor es negativo, y calculando su suma por columnas obtenemos que, en efecto, es estocastica por columnas:

```
julia> sum(S,1)
1x7 Array{Float64,2}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

3. Viendo G, se observa de nuevo que ningun valor es negativo, y calculando la suma por columnas se ve de nuevo que es estocastica por columnas:

```
julia> sum(G,1)
1x7 Array{Float64,2}:
```

```
 1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

Ademas de estas preguntas, he decidido observar la velocidad y memoria consumida, y estos son los resultados:

```
julia> @time create_inital_values(i,j,n)
elapsed time: 1.3941e-5 seconds (2592 bytes allocated)

julia> @time create_S(C,n)
elapsed time: 1.3629e-5 seconds (1992 bytes allocated)

julia> @time create_G(alpha,S,n)
elapsed time: 8.863e-6 seconds (3048 bytes allocated)
```

Se puede observar que en velocidad no estan mal, pero los bytes usados se podrian reducir, probablemente sin tener que usar la matriz total se reducirian mucho.

B. Codificar la rutina del método de la potencia. Reescalar una matriz

Primero, Julia no permite llamar a funciones sin todos los parametros (a no ser que pongas parametros por defecto), pero si permite sobreescribir funciones con especificaciones distintas. Seria mas util poner valores por defecto, ya que asi puedes pasar uno, o dos, y podria seguir funcionando.

La funcion que solo coge una matriz, basicamente calcula su tamaño y llama a potencia de nuevo con parametros por defecto.

```
1 function potencia(M::Array{Float64,2})
2     n = size(M)[1]
3     potencia(M,1e-12,500,ones(n,1))
4 end
```

La funcion que hace el calculo es la siguiente.

Antes de nada, decir que Julia, al tener un tipado gradual, puedes asignar el tipo de los datos o no, en este caso he decidido asignarselo para hacer que la ejecucion sea mas rapida, ya que el compilador JIT no tiene que hacer inferencia de tipos.

He decidido hacer el bucle con un while y aumentando el contador, aunque se podria hacer perfectamente con un for y comprobar dentro, pero esta forma me parece mas limpia, tendre que hacer pruebas para saber cual es mas eficiente.

El codigo inicializa valores (el contador i, el valor n y el vector “last” que es el denominada “x”).

Dentro del bucle, asigna a “last” el valor de “r”, (“r” es el vector denominado “x1”), divide “last” entre su norma asignandolo a “last” y por ultimo se asigna a “r” el valor de multiplicar la matriz “M” por “last”

```
1  function potencia(M::Array{Float64,2},
 2     tolerancia::Float64,
 3     n_iter::Int64,
 4     r::Array{Float64})
 5
 6     i = 0
 7     n = size(M)[1]
 8     last = zeros(n,1) ## Just to initialize
 9     while i < n_iter && maximum(abs((r/norm(r)) - last)) > tolerancia
10         i += 1
11         last = r
12
13         last = last ./ norm(last)
14         r = M * last
15     end
16     return norm(r), last, i        # This to return the eigvalues
17 end
```

Reescalar una matriz

Aqui estan los resultados de crear la matriz, normalizar y comprobar que es estocastica por columnas (no se muestra la matriz al ser muy grande):

Creacion:

```
julia> A = rand(10,10);
```

Normalizacion:

```
julia> B = A./sum(A,1);
```

Comprobacion:

```
julia> sum(B,1)
1x10 Array{Float64,2}:
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

Como se pede ver, es estocastica por columnas.

Ejecutar el metodo de la potencia.

Ejecutamos el metodo de la potencia:

```
julia> lambda, x, it = potencia(B)
```

Veamos los valores:

```
julia> lambda
1.000000000000057

julia> x
10x1 Array{Float64,2}:
 0.278691
 0.285671
 0.367552
 0.355962
 0.293077
 0.315602
 0.395101
 0.312036
 0.249719
 0.278551
```

```
julia> it
17
```

Calculemos la precision obtenida:

```
julia> precision = maximum((B * x) - (lambda * x)) :: Float64
2.810529586838584e-13
```

Una precision que, en efecto, es mayor que la marcada (por defecto 1e-12).

Teorema de Perron-Frobenius

Condiciones de entrada:

Matriz S irreducible, estocastica y no negativa.

Al no tener ningun elemento cero, no es reducible, ya que no se puede llegar a una forma triangular superior haciendo permutaciones de columas o filas.

Se puede comprobar que es estocastica asi:

```
julia> sum(B,1)
1x10 Array{Float64,2}:
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

Podemos comprobar que no es negativa con la siguiente expresion:

```
julia> any(map(x -> x<0, B))
false
```

La funcion map aplica una funcion a cada elemento de un iterable (la funcion en este caso es una funcion lambda, que devuelve true si el elemento es menor que 0 y false si es mayor o igual que cero), y la funcion any devuelve true si hay al menos un elemento que sea true, y en este caso, al devolver false, significa que todos los elementos son mayor o igual que cero.

Condiciones de salida:

Su mayor autovalor es 1 y su correspondiente autovector tiene todos sus elementos no negativos.

Por desgracia, Julia no tiene un modo de comparar numeros complejos, por lo que voy a usar estas funciones:

Una funcion auxiliar que, para todo un vector, si un numero complejo tiene su parte imaginaria como cero, lo transforma a Real, y todos aquellos que no su valor imaginario es distinto de cero, los descarta.

```
function take_out_complex(V)
    map(real, filter(x -> imag(x) == 0, V))
end
```

Otra funcion auxiliar, que dado un numero real y un vector de numeros complejos, va a encontrar su posicion en el vector.

```
function find_n_complex(n,V)
    compl = convert(Complex,n)
    for (i,item) in enumerate(V)
        if imag(item) == 0 && (abs(real(item) - real(compl)) < 10e-15)
            return i
        end
    end
    return -1
end
```

La funcion principal, que va a aplicar la funcion eig, y luega aplicar una serie de transformaciones para encontrar el autovalor maximo y su autovector asociado, y los devuelve.

```
function from_eig(B)
    D,V = eig(B)
    lambda = maximum(take_out_complex(D))
    i = find_n_complex(lambda,D)
    x_compl = V[:,i]
    x = map(abs,take_out_complex(x_compl))
    return x, lambda
end
```

Asi que veamos lo que da aplicada a nuestra matriz B:

```
julia> x2, lambda2 = from_eig(B);

julia> x2
10-element Array{Float64,1}:
 0.223472
 0.255523
 0.228886
 0.391131
 0.346778
 0.342847
 0.271734
 0.2851
 0.357474
 0.398357

julia> lambda2
1.0
```

Como podemos comprobar, el mayor autovalor es 1.0 y su autovector asociado tiene todos los elementos positivos.

Si aplicamos nuestro metodo de la potencia con error 1e-15, podemos ver que la precision se acerca mucho a la que conseguimos con eig:

```
julia> lambda, x, it = potencia(B,1e-15,500,ones(10,1));

julia> x2, lambda2 = from_eig(B);

julia> maximum((B * x) - (lambda * x))
2.220446049250313e-16

julia> maximum((B * x2) - (lambda2 * x2))
3.885780586188048e-16
```

Y, si medimos el tiempo y la memoria “allocated” , podemos ver que en tiempo el metodo eig (que viene por defecto, no estoy usando “from_eig” explicado antes) y nuestro metodo de la potencia estan muy cerca, y en memoria el nuestro consume menos.

```
julia> @time eig(B);
elapsed time: 8.5984e-5 seconds (20832 bytes allocated)

julia> @time potencia(B,1e-15,500,ones(10,1));
elapsed time: 3.9387e-5 seconds (17056 bytes allocated)
```

C. Calcular el pagerank del grafo.

Para conseguir el pagerank simplemente hay que conseguir la norma 1 al finalizar.

Para ello vamos a usar esta funcion auxiliar.

```
function pagerank(V::Array{Float64})
    return V./norm(V,1)  # Aplicando la norma 1
    # return V./sum(V,1) # Sumando la columna
end
```

Y la funcion “potencia” modificada, es tan sencillo como cambiar lo que se devuelve:

```
function potencia(M::Array{Float64,2},
    tolerancia::Float64,
    n_iter::Int64,
    r::Array{Float64,2})

    i = 0
    n = size(M)[1]
    last = zeros(n) ## Just to initialize
    while i < n_iter && maximum(abs((r/norm(r)) - last)) > tolerancia
        i += 1
        last = r

        last = last ./ norm(last)
        r = M * last
    end
    return norm(r),pagerank(last),i # This to return the pagerank
end
```

Voy a aplicar esta funcion para realizar todos los pasos y que me devuelva la salida de la potencia y su precision.

```
function test()
    i=[1, 1, 1, 2, 2, 3, 3, 4, 4, 6, 6, 7, 7]
    j=[2, 4, 5, 3, 7, 4, 6, 2, 7, 7, 5, 4, 2]
    n = 7
    Cs,C,Nj,dj = create_inital_values(i,j,n)
    S = create_S(C,n)
    alpha = 0.85
    G = create_G(alpha,S,n)
    l,x,i = potencia(G)
    prec = calc_precision(G,x,l)
    return l,x,i,prec
end
```

Que usa esta otra que todavia no he puesto, que basicamente ayuda para calcular la precision:

```
1 function calc_precision(A,x,lambda)
2     maximum((A * x) - (lambda * x))
3 end
```

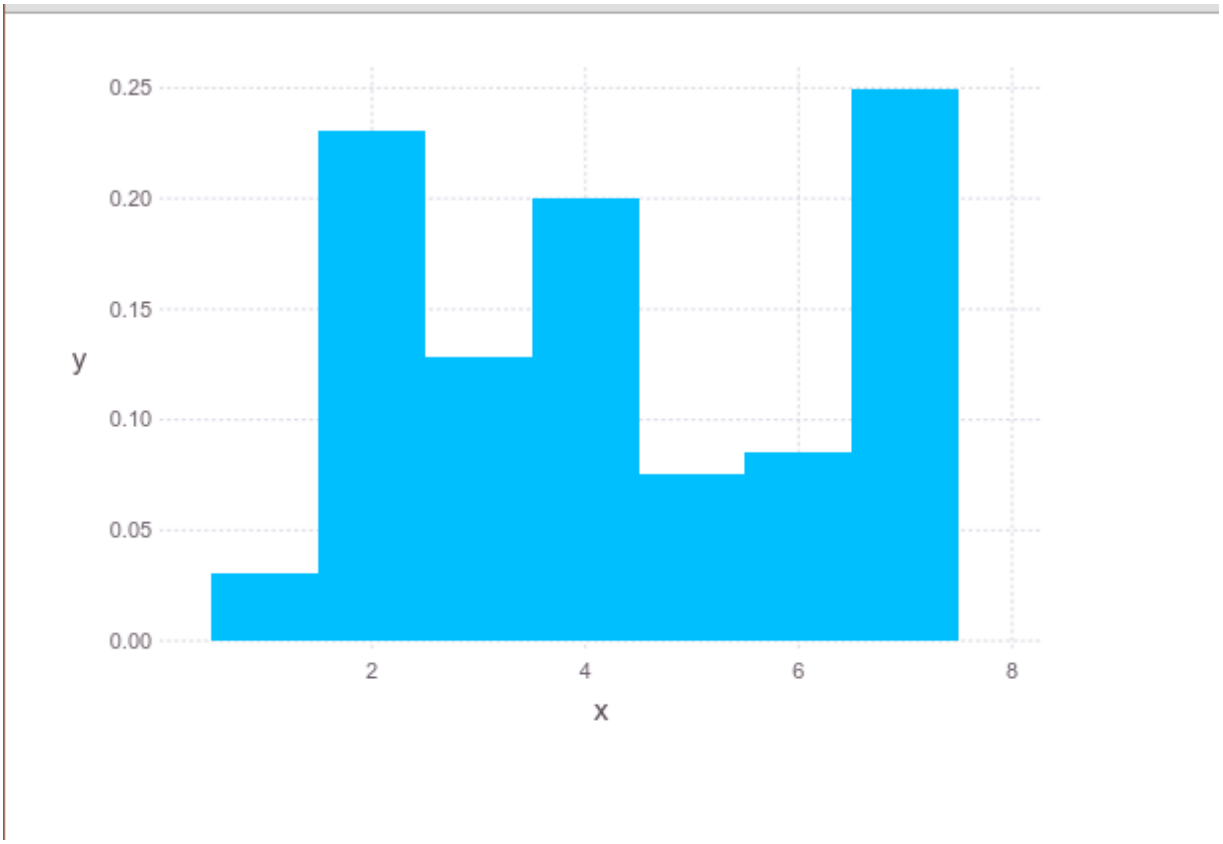
Asi que, simplemente llamamos a la funcion, para tener los valores guardados y pedimos la precision:

```
julia> lambda, x, it, prec = test();
julia> prec
1.711686348215835e-13
```

Para visualizar los resultados, julia todavia no tiene nada en su librería estandar, pero voy a usar una llamada “Gladfy”, y voy a usar la siguiente funcion para graficar como grafica “bar” de Matlab (mas o menos):

```
 1 function bar(V)
 2     n = length(V)
 3     try
 4         plot(x=range(1,n), y=V,Geom.bar)
 5     catch y
 6         if isa(y,UndefVarError)
 7             println("You don't have any plot library, maybe use \"using Gadfly\"")
 8         end
 9     end
10 end
```

El resultado es:



Por lo que el orden deberia ser:

$P_7 > P_2 > P_4 > P_3 > P_6 > P_5 > P_1$

Para hacerlo de forma matematica, voy a usar esta funcion, que basicamente ordena de mayor a menor un vector y devuelve una lista de tuplas (indice, valor).

```
1 function sort_with_index(V)
2     sort([(i,x) for (i,x) in enumerate(V)], by=x -> x[2],rev=true)
3 end
```

Y ahora, voy a usar esta otra funcion que dadas dos matrices, les aplica la funcion anterior y te devuelve los resultados:

```
function get_orders_S_G(S,G)
    _,s_p,_ = potencia(S)
    _,g_p,_ = potencia(G)
    return sort_with_index(s_p), sort_with_index(g_p)
end
```

Y ahora simplemente lo aplicamos y vemos resultados:

```
julia> so, go = get_orders_S_G(S,G);

julia> so
7-element Array{(Int64,Float64),1}:
 (7,0.2761087267526768)
 (2,0.25321888412017163)
 (4,0.2131616595134176)
 (3,0.13304721030042918)
 (6,0.07296137339055793)
 (5,0.04506437768240343)
 (1,0.0064377682403433546)

julia> go
7-element Array{(Int64,Float64),1}:
 (7,0.2497485763261237)
 (2,0.2304165058998948)
 (4,0.2000265689989668)
 (3,0.1285208351139991)
 (6,0.08521517502999337)
 (5,0.07547851852447843)
 (1,0.03059382010654383)
```

Como podemos observar, a pesar de que los valores son distintos, el orden es el mismo.