DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabájo Práctico 3

PageRank

12 de diciembre de 2013 — Métodos Numéricos

Integrante	LU	Correo electrónico
Escalante, José	822/06	joe.escalante@gmail.com
Osinski, Andrés	405/07	andres.osinski@gmail.com
Raskovsky, Iván Alejandro	57/07	iraskovsky@dc.uba.ar

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Abstract

En este trabajo nos concentraremos en analizar la teoría detrás del algoritmo PageRank, famoso por su utilización por Google en su motor de búsquedas, y un conjunto de optimizaciones propuestas por *Kamvar* y *HaveliWala*.

Dado a conocer a través de un paper en 1998, el algoritmo Page Rank, se convirtió en una de las claves del éxito del motor de búsqueda Google. Su implementación se basa en la creación de un ranking en el cual se pondera con un cierto criterio cada una de las páginas según un modelo de *navegante aleatório*.

Al final presentaremos resultados de experimentaciones que nos parecieron pertinentes, mostrando que efectivamente la variación del método iterativo sí optimiza considerablemente la convergencia de PageRank.

Palabras clave:

- PageRank
- Metodo de Potencia
- Extrapolación Cuadrática
- QR

2. Introducción Teórica

2.1. PageRank

PageRank es un algoritmo que modela un proceso aleatorio de navegación a través de distintas páginas Web, cuyo objetivo es generar un ranking de importancia de páginas, que en la práctica se utiliza para refinar los resultados de búsquedas de texto en las mismas para refinar los resultados por relevancia.

El problema se define como encontrar el autovector asociado al autovalor 1 de la matriz de transiciones correspondientes a un grafo de páginas Web interconectadas. Dado que esta matriz se puede ajustar para que sea estocástica, el autovector corresponde a la proporción de tiempo que pasa un navegante aleatorio en una página, que en la realidad resulta un buen indicador de importancia.

La manera en cómo se modela el problema del ranking es usando una matriz de adyacencias. Sea $W \in \mathbb{R}^{n \times n}$ donde n es la cantidad de sitios indexados, luego el elemento w_{ij} es igual 1 si existe un link de la página i a la página j y 0 en caso contrario. A su vez los links autoreferenciados se ignoran por lo que en diagonal tenemos todos valores nulos.

Ahora con W podemos extraer la cantidad de links salientes de cada página, simplemente sumando los elementos de la fila correspondiente. Llamamos n_j al grado de la página j donde $n_j = \sum_{i=1}^{n} w_{ij}$.

2.2. Matriz de transiciones

Con la matriz de adyacencias originales, ahora queremos tomar sus datos y generar con ellos una matriz estocástica que defina la probabilidad de clickear en un link al azar en una página y transicionar hacia otra.

Para ello, consideramos P la matriz que se obtiene a partir de tomar la matriz de adyacecias y normalizando sus filas para que tengan norma 2 igual a 1. Es decir, si una página tiene 2 links salientes, la probabilidad de clickear en uno de ellos es $frac12$, con tres links es $frac13$, y así sucesivamente.

El problema que hay con esto es que no todas las páginas necesariamente tienen links salientes, por lo que se asume que si un navegante llega a una página sin *outlinks*, elige una página al azar sobre el universo de todas las páginas.

Para modelar esto, sea $\vec{d}$ un vector con longitud n, la posición d_i toma los valores

$$d_i = \begin{cases} 0 & \text{si el sitio tiene links salientes} \\ 1 & \text{si el sitio no tiene links salientes} \end{cases}$$

Luego, sea $\vec{v}$ un vector con logitud v con $v_i = [\frac{1}{n}]$, definimos la matriz D como

$$D = d^t * v$$

Luego definimos $P' = P + D$, que es estocástica.

2.3. Teleporteo

Además del manejo de navegación aleatoria, *PageRank* considera una probabilidad de que en cualquier momento el navegador eliga saltar a una nueva página al azar, sin considerar los links de la página actual. Esto se llama *teleporteo*, y se modela como un factor c que multiplica a P' y

una matriz $E \in R^{nxn}$, con una probabilidad uniforme de $\frac{1}{n}$.

Luego la expresión que resuelve el problema de *PageRank* de la matriz de transiciones con teleporteo, dado el vector de probabilidades $\vec{x}$, es

$$Ax = (cP' + (1-c)E)^t \vec{x}$$

2.4. Resolución de PageRank

La resolución de *PageRank*, en su forma más básica, consiste en obtener el autovector asociado al autovalor 1. La forma más simple de hacerlo es por medio del método de potencias que funciona computando $A\vec{x}$ iterativamente hasta cumplir con un criterio de detención.

Este método presenta un problema práctico, en que la matriz de transiciones suele ser muy esparsa, pero si se considera la información agregada por las matrices E y D, la misma resulta densa. A continuación veremos que esto se puede resolver.

2.5. Cálculo alternativo de $x^{(k+1)} = P_2 x^{(k)}$

Veamos primero cómo utilizando el algoritmo de Kamvar podemos optimizar el espacio requerido en memoria para el almacenamiento de la matriz P_2 y el tiempo de ejecución requerido para ha cer la multiplicación entre matrices y vectores.

Queremos ver que el algoritmo propuesto por [**?**, Algoritmo 1] es equivalente a la operación $\vec{y} = A\vec{x}$, para $A = (cP + (1-c)E)^t$, donde P es la matriz de transiciones de links, no ajustada por las páginas sin outlinks, y E es la matriz uniforme de teletransportación con valor $\frac{1}{n}$ en cada celda.

Para ello, expandimos las ecuaciones de ambos y veremos que las mismas producen el mismo cálculo.

Primero, la ecuación $A\vec{x}$ se desarrolla como

$$(c(P+D) + (1-c)E)^t \vec{x}$$

Y la matrix de [**?**, Algoritmo 1] como

$$cP^t\vec{x} + (\|\vec{x}\|_1 - \|\vec{y}\|_1)\vec{v}$$

donde $\vec{y}$ es el vector resultante de $cP^t\vec{x}$ y $\vec{v}$ es el vector de probabilidad uniforme de valor $\frac{1}{n}$ en cada elemento. Luego planteamos la equivalencia

$$\begin{aligned} c(P+D)^t + (1-c)E^t\vec{x} &= cP^t\vec{x} + (\|\vec{x}\|_1 - \|\vec{y}\|_1)\vec{v} \\ cP^t\vec{x} + cD^t\vec{x} + (1-c)E^t\vec{x} &= cP^t\vec{x} + (\|\vec{x}\|_1 - \|\vec{y}\|_1)\vec{v} \\ cD^t\vec{x} + (1-c)E^t\vec{x} &= (\|\vec{x}\|_1 - \|\vec{y}\|_1)\vec{v} \end{aligned}$$

Veamos por qué el término del lado izquierdo es equivalente al derecho.

La norma 1 del vector es la suma de los valores absolutos de sus elementos. Observemos que, para las columnas de P, las mismas o bien tienen norma 1 que vale 1, o cero. Es decir, las columnas no-cero de P contribuyen a la norma 1 de $\vec{y}$.

Observemos también que en la multiplicación $(1-c)E\vec{x}$, la norma 1 de este producto es $(1-c)$, pues la norma 1 de cada columna de E es 1 y $\|\vec{x}\| = 1$.

Por ultimo, también vemos que para aquellas columnas de ceros en P^t, las columnas no-cero de D preservan la norma de las mismas.

Dados los hechos anteriores, podemos obervar que $\|\vec{x}\|_1 - \|\vec{y}\|_1$ se puede interpretar como la norma que se "pierde" cuando $\vec{x}$ es multiplicada por cP^t. Esta "pérdida de norma" se debe justamente a que falta sumar los products de $\vec{x}$ por E y D de la ecuación, pues A preserva la norma 1.

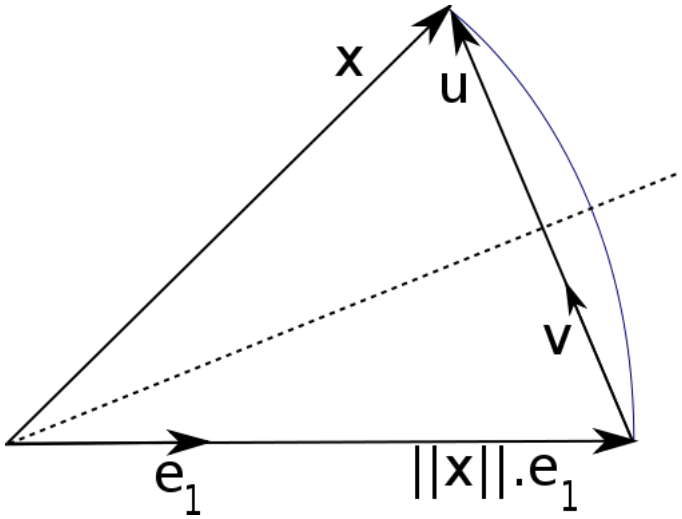
Ahora, como en E^t y D^t, las filas que componen cada matriz son iguales entre sí (en E^t todos elementos valen $\frac{1}{n}$, y en D^t como fue descrito en la sección anterior), cuando se realiza $E\vec{x}$ o $D^t\vec{x}$, el resultado produce un vector de valores idénticos en cada posición. Estos mismos valores son la diferencia que captura $(\|\vec{x}\|_1 - \|\vec{y}\|_1)\vec{v}$. Luego agregar esto a $\vec{y}$ resulta equivalente a haber realizado las sumas de vectores resultantes correspondientes, con la diferencia de que no hizo falta materializar dichas matrices.

Con ello concluimos que los dos términos del algoritmo de Kamvar son equivalentes a la matriz A de transiciones. ■

2.6. QR y Reflecciones de Householder

Existen variadas formas de descomponer una matriz. En este trabajo en particular usaremos la descomposición QR, la cual consiste en descomponer una matriz A en una matriz ortogonal Q y una triangular superior R de manera que $A = QR$. Al tener descompuesta A en esa forma y teniendo un sistema $Ax = b$, la obtención del vector solución se realiza resolviendo el sistema $Rx = Q^t b$.

A su vez existen distintos procedimientos para poder hallar la descomposición QR de una matriz. Uno de ellos es las *reflecciones de Householder*, procedimiento por el cual en cada iteración se obtienen ceros por debajo de la diagonal, llevando la matriz A a ser la matriz R diagonal superior.



El objetivo es hallar una transformación lineal que cambie el vector x en un vector de la misma longitud que sea colineal con e_1. Householder refleja a través de la línea punteada (elegida para dividir el ángulo entre x y e_1). El máximo ángulo con esta transformación es a lo sumo de 45º.

Veamos un paso del procedimiento para poder ilustrar mejor: Sea $A \in \mathbb{R}^{n \times n}$ y sea x_1 el vector correspondiente a la primer columna de A y e_1 el vector canónico, definimos:

$$\alpha_1 = \|x_1\|$$

$$u = x_1 - \alpha_1 e_1$$

$$v = \frac{u}{\|u\|}$$

$$Q_1 = I - 2vv^T$$

Luego Q_1 es la matriz que al multiplicarla a izquierda por A coloca ceros por debajo de la diagonal.

$$Q_1 A = \begin{bmatrix} \alpha_1 & \star & \dots & \star \\ 0 & & & \\ \vdots & & A' & \\ 0 & & & \end{bmatrix}$$

Repitiendo estos pasos: $Q_{n-1} \dots Q_2 Q_1 A = R$. Luego el producto de las Q's forma una matriz ortogonal, lo cual hace fácil y rápido hallar el vector solución del sistema.

$$Q^T = Q_{n-1} \dots Q_2 Q_1$$

$$Ax = b \iff Q^T Ax = Q^T b \iff Rx = Q^T b$$

3. Desarrollo

3.1. Detalles de Implementación

3.1.1. Enfoque Inicial - Etapa Python

En principio para evitarnos detalles de manejo de memoria y contar con una mayor de expresividad de lenguaje, implementamos el trabajo en Python.

Usando librerías como Numpy y Scipy, pudimos hacer uso de matrices esparsas y operarlas cómodamente de manera eficiente; dado que las todas las distintas clases de matrices esparses disponibles exponen la misma interfaz, pudimos probar entre varias hasta encontrar una que funcione para nuestro caso de uso.

Tan sólo unas horas de trabajo y terminamos una implementación que devolvía resultados que parecían correctos. Inclusive con datasets enormes, como los que se pueden encontrar en la página de Stanford, el programa en Python tardaba pocos segundos por iteración y en cuestión de minutos armaba el ranking.

Las unicas diferencia sustanciales de tiempo aparecian en la lectura del archivo de entrada y armado de matriz inicial, donde el overhead en Python dominaba y no habia ningun conjunto de subrutines optimizadas para esa tarea.

3.1.2. Implementación - Etapa C++

Una vez habiendo comprobado que la idea de cómo implementar este trabajo funcionaba, es que usamos el código de Python a manera de pseudocódigo para el de C++.

En C++ para armar las matrices esparsas usamos STL y la estructura de datos *std::map*, que expone una interfaz de contenedor genérico que preserva órden, internamente implementado como un Red-Black tree.

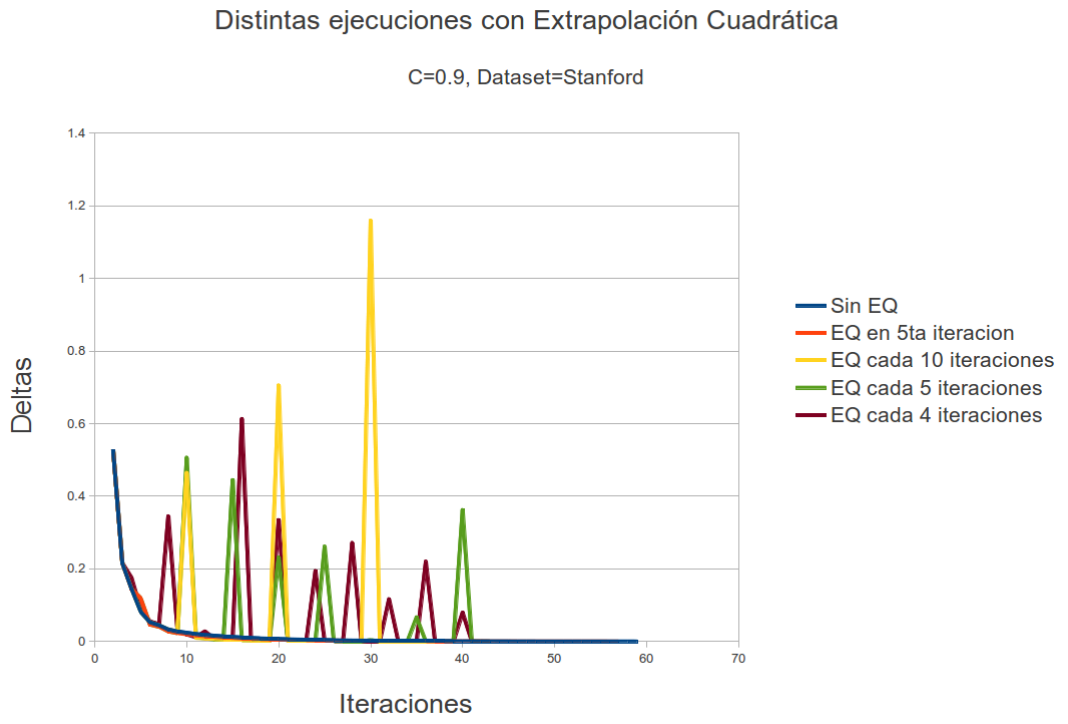
A diferencia de la implementación en Python donde hacemos uso indiscriminado de la riqueza de las librerías, nos vimos forzados a acomodar las operaciones de manera tal que tengamos que implementar solamente las operaciones exclusivamente necesarias.

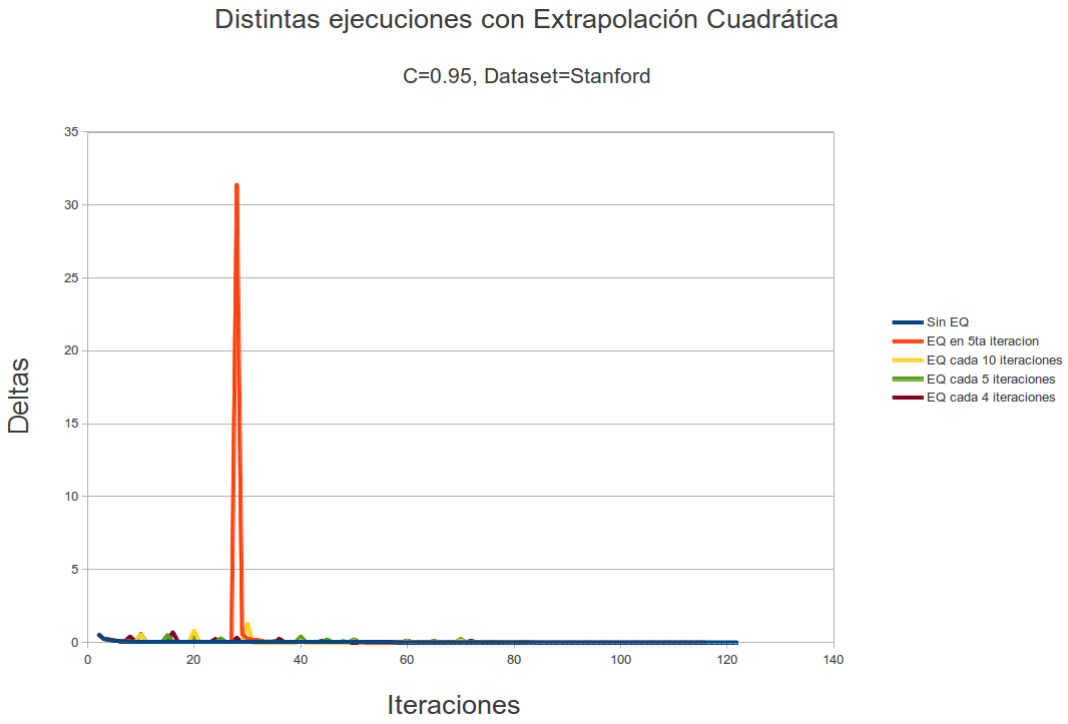
La reclamación de recursos de memoria es uno de los mayores problemas de usar lenguajes con manejo de memoria manual. Esto en la práctica no resulto ser un problema, dado que tomamos ventaja de las faciliades de manejo de memoria provistas por la nueva versión de C++: los *unique_ptr* y los *shared_ptr*, que permiten reclamar objetos que se van fuera de *scope*, y en el caso de los *unique_ptr*, también garantizan que un objeto tenga una sola variable como "dueño" a la vez, permitiendo razonar más facilmente sobre la misma.

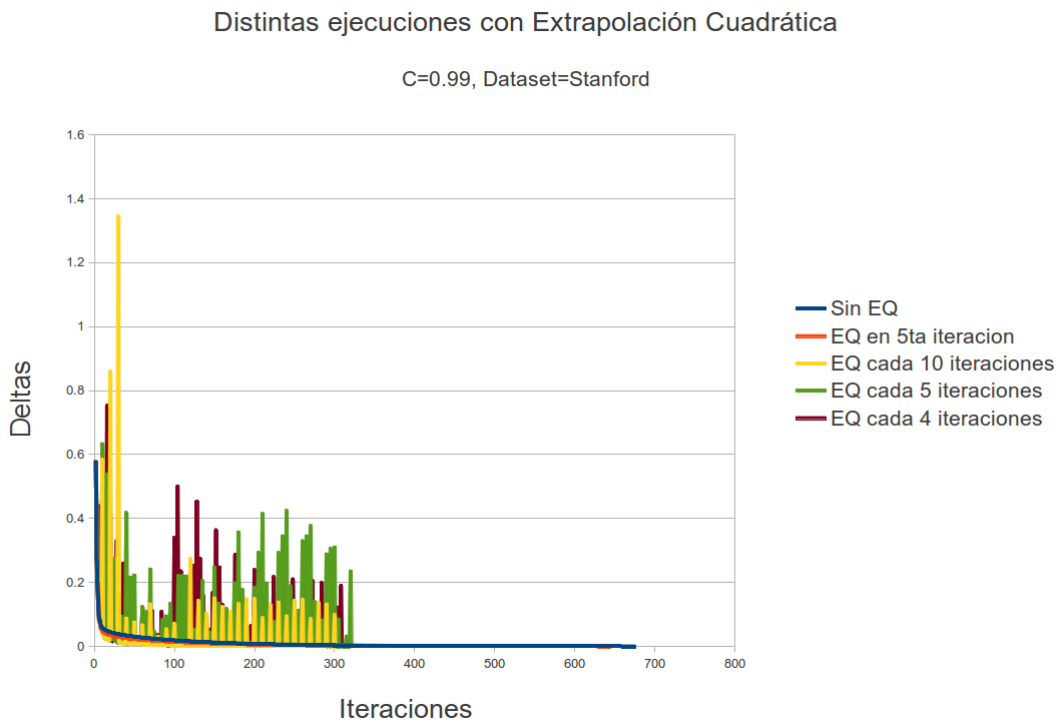
Un problema que sí hubo fue que nuestro primer intento de implementación intentaba simplificar las operaciones matriciales haciendo uso de la sobrecarga de operadores en C++. El problema con esto es que los operadores retornan objetos por copia, por lo que cada operación numérica hecha con estos implicaría una copia masiva de datos (en teoria C++ soporta una clase de optimización llamada *elisión de copia* para evitar retornar objetos por copia, pero su utilización es muy acotada y es difícil contar con la misma siempre).

Para resolver esto, implementamos las operaciones con variantes que pasan objetos por referencia, y en muchos casos, para ahorrar memoria, hicimos operaciones que modifican la matriz original. En la práctica esto fue necesario, dado que varias matrices esparsas llegaban a consumir múltiples gigabytes de memoriya.

4. Resultados






Distintas ejecuciones con Extrapolación Cuadrática
C=0.99, Dataset=Stanford
Deltas
Iteraciones
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
100
200
300
400
500
600
700
800
Sin EQ
EQ en 5ta iteracion
EQ cada 10 iteraciones
EQ cada 5 iteraciones
EQ cada 4 iteraciones

5. Discusión y Conclusiones

A. Referencias

Wikipedia Burden