Il metodo Arnoldi-PET per il PageRank

Silvio Martinico

Abstract

In questa relazione analizzeremo il metodo Arnoldi-PET per il calcolo del vettore di PageRank. L'algoritmo consiste nell'alternanza di due metodi: il metodo di Arnoldi e il metodo delle potenze. Tuttavia non utilizzeremo i due metodi nelle loro versioni originali, infatti al posto del metodo di Arnoldi utilizzeremo una sua variante che ci permette di risparmiare memoria (fondamentale quando si lavora con matrici molto grandi come la matrice di adiacenza relativa al grafo di una rete), mentre al posto del metodo delle potenze utilizzeremo il P.E.T. (Power method with the Extrapolation process based on Trace) che ci permette di accelerare il metodo delle potenze classico.

1 Il problema del PageRank

Introduciamo adesso brevemente il problema del PageRank, in modo da avere più chiaro quello che andremo a fare.
Immaginiamo di avere una rete (che può essere l'intero web o un suo sottoinsieme proprio) e di rappresentare le pagine della rete tramite un grafo orientato con n nodi (dove n è il numero di pagine della rete); l'arco (i, j) nel grafo indica la presenza, nella pagina j, di un link alla pagina i. Al grafo possiamo associare una matrice di adiacenza H e, ad ogni pagina, possiamo associare un valore w_i che rappresenza la sua importanza. Per risolvere alcuni problemi legati alla modellazione del problema al posto di H useremo $\hat{H} = H + eu^T$, dove e è il vettore con tutte le componenti uguali ad 1 ed u è il vettore che ha 1 in corrispondenza dei *dangling nodes* (le colonne nulle) e 0 altrove. Il problema che vogliamo risolvere è quindi:

$$w_i = \sum_{j=1}^{n} \frac{w_j \hat{h_{ij}}}{\hat{d}_j} \qquad \forall\ i \in \{1, ..., n\}$$

dove $\hat{d_j} = \sum_{k=1}^{n} \hat{h}_{kj}$. Questo è proprio il problema di trovare un autovettore relativo all'autovalore $\lambda = 1$.
Per poter applicare il **teorema di Perron-Frobenius** che ci garantisce l'esistenza di una soluzione a componenti positive e l'unicità di essa, dobbiamo modificare ulteriormente il modello, fino ad ottenere la formulazione finale del problema, che è anche quella con la quale andremo a lavorare:

$$Ax = x, \qquad A = \alpha P + (1 - \alpha) v e^T$$

dove $\alpha \in (0,1)$ è detto *damping factor*, $e = (1, ..., 1) \in \mathbb{R}^n$, $v \in \mathbb{R}^n$ è un vettore a componenti positive tale che $e^T v = 1$ ed è detto vettore di personalizzazione e $P = \hat{H}\hat{D}^{-1}$ con $\hat{D}^{-1} = diag(\hat{d})$.

2 Illustrazione degli algoritmi

Gli algoritmi utilizzati in questa sperimentazione sono il **Thick restarted Arnoldi algorithm** e la variante **PET** del metodo delle potenze. Vediamoli nel dettaglio.

2.1 Il PET method

A è una matrice primitiva e stocastica per colonne, quindi per i suoi autovalori λ_i vale $1 = \lambda_1 > |\lambda_2| \geq ... \geq |\lambda_n| \geq 0$ e il suo polinomio caratteristico sarà della forma $(\lambda - 1)q(\lambda)$, dove $q(\lambda)$ è un polinomio di grado $n-1$ senza il fattore $\lambda - 1$. Usando il teorema di Hamilton-Cayley abbiamo $(A - I)q(A) = 0$ e quindi $(A - I)q(A)u = 0$ per ogni vettore $u \in \mathbb{R}^n$. Dato che esiste almeno un vettore $u^{(0)}$ tale che $q(A)u^{(0)} \neq 0$ abbiamo che allora $q(A)u^{(0)}$ è un autovettore relativo all'autovalore 1 e ci dà quindi una rappresentazione del vettore di PageRank. Calcolare q(A) è però dispendioso, quindi ci limiteremo ad usare una sua approssimazione data dai soli due termini di grado maggiore:

$$(A^{n-1} - (\mu - 1)A^{n-2})u^{(0)} = A^{n-m_1-1}[u^{(m1)} - (\mu - 1)u^{(m_1-1)}]$$

dove $u^{(i)}$ è l'i-esima iterazione del metodo delle potenze.

Vediamo quindi l'implementazione dell'algoritmo, in cui useremo la tecnica di estrapolazione ogni m_1 iterazioni del metodo delle potenze:

```
3.  for i=1:m1
        x_i = A*x_(i-1);
        r = norm(x_i - x_(i-1));
        x_i = x_i/norm(x_i, 1);
        k = k + 1;
        if r <= tol
            break;
        end
    end
    %ESTRAPOLAZIONE:
    x_0 = x_m1 - (mu-1)*x_(m1-1);
    x_0 = x_0/norm(x_0, 1);
    r = norm(x_0 - x_m1);
    if r <= tol
        break
    else
        goto step 3;
    end
```

2.2 Thick restarted Arnoldi algorithm

Data una matrice $A \in \mathbb{R}^{n\times n}$, un vettore $v_1 \in \mathbb{R}^n$ ed un nunero di passi m, il processo di Arnoldi genera, tramite il processo di Gram-Schmidt modificato, una base ortonormale $\{v_1, ..., v_m\}$ del sottospazio di Krylov $\mathcal{K}_m(A, v_1) := Span(v_1, Av_1, ..., A^{m-1}v_1)$.

```
for j = 1:m
    q = A*v_j;
    for i = 1:j
        H(i,j) = (v_i)'*q;
        q = q - H(i,j)*v_i;
    end
    H(j+1,i) = norm(q);
    if H(j+1,j) == 0
        break;
    end
    v_(j+1) = q/H(j+1,j);
end
```

Dall'Algoritmo di sopra abbiamo: $AV_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^T = V_{m+1}\bar{H}_m$, dove $V_m = [v_1, ..., v_m] \in \mathbb{R}^{n\times m}$ è una matrice le cui colonne sono ortogonali, $H_m \in \mathbb{R}^{m\times m}$ e $\bar{H}_m \in \mathbb{R}^{(m+1)\times m}$ sono matrici di Hessenberg superiori e vale $\bar{H}_m = \begin{pmatrix} H_m \\ h_{m+1,m} e_m^T \end{pmatrix}$.
Al crescere di m aumenta l'ingombro di memoria del processo di Arnoldi, è qui che interviene la tecnica di restart, in modo da mantenere m molto più piccolo di n. Ecco una possibile implementazione dell'algoritmo:

```
function x1 = thick (v1, M, dang, d1, a, v, m, p, tol)

    % v1: vettore iniziale unitario
    % M: matrice di Adiacenza del grafo
    % dang: vettore con 1 in corrisp. dei dangling nodes
    % d1: vettore dei d_i
    % a: damping factor
    % v: vettore di personalizzazione
    % m: dim. ssp. di Krylov
    % p: numero di autovett. approssimati desiderati
    % tol: tolleranza

    [V, H2] = Arnoldi(M, dang, d1, a, v, v1, m);
    [B,~] = eigs(H2(1:m,:), p);

    check = 1;
    for cont = 1:2 %Nel nostro caso decidiamo di eseguire due
                   %volte il thick ogni volta che viene richiamato
```

```
        if (check == 0)

            %RESTART
            [V, H2] = Arnoldi2(M, dang, d1, a, v, m, Vn, Hn, p);
            [B, ~] = eigs(H2(1:m,:), p);

            em = zeros(1, m);
            em(1,m) = 1;
            if (H2(m+1,m)*abs(em*B(:,1)) <= tol || cont == 2)
                x1 = V(:,1:m)*B(:,1);
                return;
            end
        end

        check = 0;

        B = MGS(B);

        for i = 1:p
            if (norm(imag(B(:,i))) ~= 0)
                if (i == p)
                    y = B(:,p);
                    B(:,p) = real(y);
                    B(:,p+1) = imag(y);
                end
                if (i < p)
                    y = B(:,i);
                    B(:,i) = real(y);
                    B(:,i+2:p+1) = B(:,i+1:p);
                    B(:,i+1) = imag(y);
                end
            end
        end
        B = B(:,1:p);

        Wpt = B;
        Wpt(m+1,:) = 0;
        g = zeros(m+1,1);
        g(m+1,1) = 1;
        Wpp= Wpt;
        Wpp(:,p+1) = g;
        Vn = V*Wpp;
        Hn = Wpp'*H2*B;
    end
end
```

La funzione *Arnoldi* è semplicemente l'algoritmo di Arnoldi, *Arnoldi2* è sempre l'algoritmo di Arnoldi ma parte dai V_{p+1} e $\bar{H}_p$ ottenuti poco prima. Infine "MGS" non è altro che l'algoritmo di Gram-Schmidt modificato.

2.3 Arnoldi-PET

L'algoritmo *Arnoldi-PET* non fa altro che combinare i due precedenti algoritmi, alternandoli periodicamente fino al raggiungimento dell'approssimazione desiderata.

```
function x = ArnoldiPET(H,dang,d1,v,x0,m,m1,p,beta,maxit,tol,l,a)
    n = length(H);
    e = ones(n,1);
    k = 1; r = 1; r0 = r; r1 = r;
    mu = 1 + a*(l/n-1);
    f = 0;

    while(f < 1)
        x1 = thick (x0, H, dang, d1, a, v, m, p, tol);
        x1 = x1/norm(x1,1);
        y1 = d1.*x1;
        uy = sum(dang.*y1);
        ax = a*H*y1 + a*uy*e + (1-a)*v;
        if (norm(ax-x1,1) < tol)
            x = x1;
            return;
        end
        restart = 0;
        x = x1;
        while (restart < maxit && r > tol)
            ratio = 0;
            while (ratio < beta && r > tol)
                xk=x;
                y = d1.*x;  %y=(D_cappuccio)^(-1)*x
                uy = sum(dang.*y);  %uy=sum_{i \in dang} {y_i}
                x = a*H*y+a*uy*e+(1-a)*sum(x)*v;  %x_(k+1)=Ax_k
                r = norm(x-xk);
                ratio = r/r0;
                r0 = r;
                k = k + 1;
                if (mod(k,m1) == 0)
                    x0 = x - (mu - 1)*xk;
                    x0 = x0/norm(x0,1);
                    r = norm(x0 - x);
                    if (r <= tol)
                        break;
```

```
                    end
                    x = x0;
                end
            end
            if (r/r1 > beta)
                restart = restart + 1;
            end
            r0 = r;
            r1 = r;
        end
        if (r <= tol)
            return;
        end
        x0 = x;
    end
end
```

Nel nostro caso abbiamo deciso, ad ogni ciclo, di effettuare 2 iterazioni del *Thick Restarted* seguite dall'applicazione del metodo *PET*.

Riportiamo adesso le parti di codice mancanti, cioè le funzioni *Arnoldi*, *Arnoldi2* e *MGS*, oltre al codice di avvio che carica la matrice di adiacenza e tutti gli altri dati che ci servono e richiama la funzione *ArnoldiPET* passandogli i rispettivi parametri.

Tutto parte quindi da questo codice:

```
W = load('nome_matrice.txt');
n = "dimensione matrice";
H = sparse(W(:, 1), W(:, 2), ones(size(W, 1), 1), n, n);
H = H';                  %MATRICE DI ADIACENZA
alpha = 0.99;
beta = alpha - 0.1;
e = ones(n, 1);
d = ((e')*H)';
dang = zeros(n,1);

for i = 1:n
    if (d(i) == 0)
        dang(i) = 1;  %dang HA 1 IN CORRISPONDENZA DEI
    end               %DANGLING NODES E 0 ALTROVE
end

n_dang = sum(dang);         %è IL NUMERO DI DANGLING NODES

dh = d + dang*n;
d1 = 1./dh;
```

```
v = e/n;                    %VETTORE DI PERSONALIZZAZIONE

tol = 10^(-8);
x0 = e/n;                   %VETTORE DI PARTENZA

x=ArnoldiPET(H,dang,d1,v,x0,m,m1,p,beta,maxit,tol,n_dang,alpha);
```

Queste invece sono le funzioni *Arnoldi* ed *Arnoldi2*:

```
function [Q, H] = Arnoldi (M, dang, d1, a, v, q1, m)
  n = length(M);
  q1 = q1/norm(q1);
  Q = zeros(n,m+1);
  Q(:,1) = q1;
  H = zeros(m+1,m);
  e = ones(n,1);

  for k = 1:m
      x = Q(:,k);
      y = d1.*x;
      uy = sum(dang.*y);
      z = a*M*y + a*uy*e + (1-a)*sum(x)*v;
      for i = 1:k
          H(i,k) = Q(:,i)'*z;
          z = z - H(i,k)*Q(:,i);
      end
      H(k+1,k) = norm(z);
      if (H(k+1,k) == 0)
          return;
      end
      Q(:,k+1) = z/H(k+1,k);
  end
end


function [Q, H] = Arnoldi2 (M, dang, d1, a, v, m, Vn, Hn, p)
  n = length(M);
  Q = zeros(n,m+1);
  Q(:,1:p+1) = Vn;
  Q(:,p+1) = Q(:,p+1)/norm(Q(:,p+1));
  H = zeros(m+1,m);
  H(1:p+1,1:p) = Hn;  %RIPARTO DA v_p CON I
  e = ones(n,1);      %V_p+1 e (H_bar)_n NUOVI

  for k = p+1:m
      x = Q(:,k);
```

```
        y = d1.*x;
        uy = sum(dang.*y);
        z = a*M*y + a*uy*e + (1-a)*sum(x)*v;
        for i = 1:k
            H(i,k) = Q(:,i)'*z;
            z = z - H(i,k)*Q(:,i);
        end
        H(k+1,k) = norm(z);
        if (H(k+1,k) == 0)
            return;
        end
        Q(:,k+1) = z/H(k+1,k);
    end
end
```

Ed infine la funzione *MGS*:

```
function Q = MGS(A)
  [~,n] = size(A);
  R = zeros(n);
  Q = A;
  for k = 1:n
  for i = 1:k-1
      R(i,k) = Q(:,i)'*Q(:,k);
      Q(:,k) = Q(:,k)-R(i,k)*Q(:,i);
  end
  R(k,k) = norm(Q(:,k));
  Q(:,k) = Q(:,k)/R(k,k);
  end
end
```

3 Esperimenti numerici

Per i nostri esperimenti abbiamo usato due diverse reti (e quindi due diverse matrici), entrambe reperibili sul sito Snap Stanford.

Matrici				
Nome	**n**	**nnz**	**numd**	**den**
$Web - Stanford$	$281,903$	$2,312,497$	172	0.291×10^{-2}
$Stanford–Berkeley$	$683,446$	$7,583,376$	$68,062$	0.162×10^{-2}

Nella tabella n indica il numero di nodi (e quindi il numero di righe e colonne della matrice H), nnz il numero di elementi diversi da 0, $numd$ il numero di *dangling nodes* e den la densità della matrice. Per i test sulla matrice *web-Stanford* abbiamo posto $m = 5$, $p = 3$, $maxit = 12$ ed $m_1 = 40$, mentre per la matrice *Berkeley-Stanford* abbiamo posto $m = 8$, $p = 5$, $maxit = 6$ ed $m_1 = 40$.

Andremo a confrontare il metodo *Arnoldi-PET* con il metodo *PET* ed il metodo delle potenze classico (*Power method*), in particolare confronteremo il numero di iterazioni IT, il numero di prodotti matrice-vettore (Mv) ed il tempo impiegato per l'esecuzione (CPU).

Riportiamo quindi di seguito il codice del metodo delle potenze:

```
function x = potenze (H, dang, d1, v, x0, tol, a)

    n = length(H);
    e = ones(n,1);
    r = 1;
    it = 0;

    x = x0;

    while (r > tol)
        it = it + 1;
        xk = x;
        y = d1.*x;  % y = (D_cappuccio)^(-1) * x
        uy = sum(dang.*y);  % uy = sum_{i \in dang} {y_i}
        x = a*H*y + a*uy*e + (1-a)*sum(x)*v;   % x_(k+1) = Ax_k
        r = norm(x-xk);
    end
end
```

il quale verrà richiamato dopo aver caricato in memoria tutti i parametri necessari, proprio come si è visto prima con il metodo *Arnoldi-PET*.

Per tutti i test verrà utilizzato lo stesso dato iniziale $x_0 = v = e/n$, dove $e = [1, ..., 1]^T \in \mathbb{R}^n$. La tolleranza è $tol = 10^{-8}$, il damping factor α varia nell'insieme $\{0.99,\ 0.993,\ 0.995,\ 0.997\}$ ed il parametro β usato nell'*Arnoldi-PET* per l'alternanza tra il *Thick Restarted* ed il *PET* è $\beta = \alpha - 0.1$.

web-Stanford			
α	**Power**	**PET**	**Arnoldi-PET**
$\alpha = 0.99$			
IT	1141	679	235
Mv	1141	679	333
CPU (secondi)	63, 180492	33.944073	16.659422
$\alpha = 0.993$			
IT	1632	919	293
Mv	1632	919	419
CPU	87, 660203	50.105789	20.194546
$\alpha = 0.995$			
IT	2287	1199	336
Mv	2287	1199	469
CPU	119, 043108	64.616085	23.607478
$\alpha = 0.997$			
IT	3815	1759	352
Mv	3815	1759	513
CPU	200, 317533	91.391905	28.054759

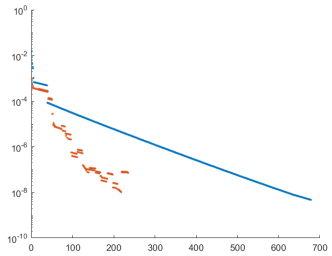

Figure 1: Errore per $\alpha = 0.99$

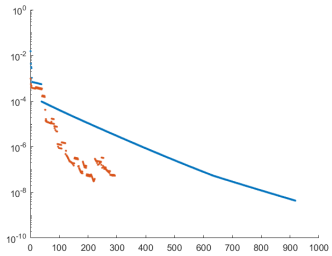

Figure 2: Errore per $\alpha = 0.993$

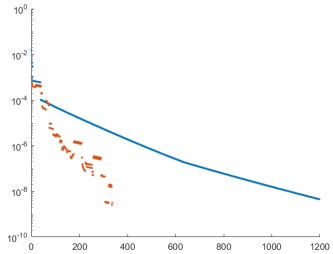

Figure 3: Errore per $\alpha = 0.995$

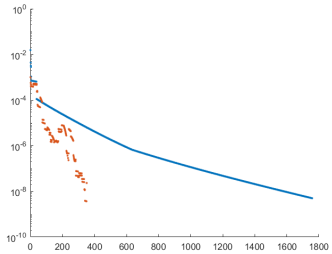

Figure 4: Errore per $\alpha = 0.997$

Berkeley-Stanford			
α	**Power**	**PET**	**Arnoldi-PET**
$\alpha = 0.99$			
IT	1216	679	225
Mv	1216	679	511
CPU (secondi)	172, 626647	118.246344	77.357104
$\alpha = 0.993$			
IT	1746	879	305
Mv	1746	879	701
CPU	247, 097334	151.458633	109.179700
$\alpha = 0.995$			
IT	2449	1159	449
Mv	2449	1159	1054
CPU	321, 427541	194.437221	163.417412
$\alpha = 0.997$			
IT	4086	1679	661
Mv	4086	1679	1552
CPU	563, 241061	278.154462	240.496568

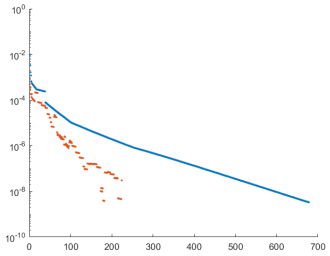

Figure 5: Errore per $\alpha = 0.99$

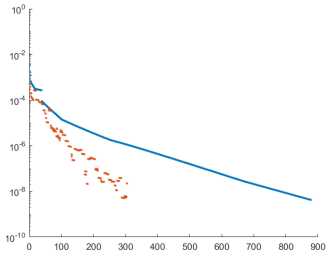

Figure 6: Errore per $\alpha = 0.993$

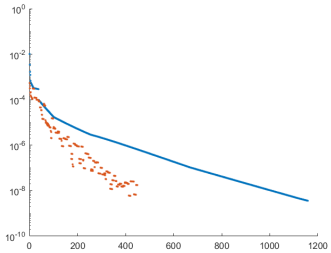

Figure 7: Errore per $\alpha = 0.995$

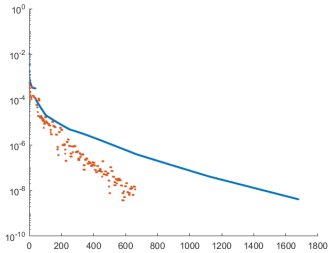

Figure 8: Errore per $\alpha = 0.997$

La prima tabella ed i primi quattro grafici (da Figura 1 a Figura 4) sono relativi alla matrice *web-Stanford*, mentre la seconda tabella e i rispettivi quattro grafici sotto ad essa sono relativi alla matrice *web-BerkStan*. Sulle ascisse abbiamo le iterazioni, mentre sulle ordinate abbiamo l'errore (in scala logaritmica); il blu rappresenta l'algoritmo *PET*, mentre il rosso è associato all'*Arnoldi-PET*.
L'errore all'iterazione i-esima corrisponde al valore $||Ax_i - x_i||_\infty$, dove x_i è l'approssimazione ottenuta al passo i (normalizzata in norma 1).

4 Considerazioni finali

Possiamo quindi concludere che, per tutte le matrici testate (le quali sono in totale 8, date dalle due diverse reti al variare del damping factor α tra quattro possibili valori), il nuovo algoritmo testato ha delle performance migliori rispetto al medoto PET ed al metodo delle potenze con il quale è stato confrontato, sia in termini di tempo che in termini di iterazioni e prodotti matrice-vettore. Notiamo anche che il metodo PET accelera e migliora notevolmente il metodo delle potenze classico.
Il nuovo algoritmo promette quindi bene e sembra riuscire a trovare un compromesso tra la velocità (accelerando il metodo delle potenze) e la memoria occupata (usando la tecnica di Restart nel metodo di Arnoldi) in modo da sfruttare i punti di forza di entrambi gli algoritmi utilizzati.