CS 246: Problem Set 3

Tony Hyun Kim

February 23, 2012

1 Similarity Ranking

1.1 Similarity by "hand"

I did the first iteration by hand (really), and realized that I was not sufficiently awake to carry out the computation for three more iterations without making a mistake. So, I wrote a short Matlab script (attached) to compute the similarity. The rows and columns of S_A are ordered {cameras, phones, printers}; and the rows and columns of S_B are ordered {HP, Nokia, Kodak, Apple, Canon}. Sim(camera,phone) and sim(camera,printer) correspond respectively to the two boldface entries of S_A indicated below. Note that we expect sim(camera,printer)= 0 since the printer-HP graph is disjoint from the remaining bipartite graph.

Iteration 1:

$$S_A^{(1)} = \begin{bmatrix} 1 & \mathbf{0.1333} & \mathbf{0} \\ 0.1333 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad S_B^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0.4000 & 0.4000 & 0.4000 \\ 0 & 0.4000 & 1 & 0 & 0.8000 \\ 0 & 0.4000 & 0 & 1 & 0 \\ 0 & 0.4000 & 0.8000 & 0 & 1 \end{bmatrix}$$

Iteration 2:

$$S_A^{(2)} = \begin{bmatrix} 1 & \mathbf{0.2933} & \mathbf{0} \\ 0.2933 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad S_B^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0.4533 & 0.4533 & 0.4533 \\ 0 & 0.4533 & 1 & 0.1067 & 0.8000 \\ 0 & 0.4533 & 0.1067 & 1 & 0.1067 \\ 0 & 0.4533 & 0.8000 & 0.1067 & 1 \end{bmatrix}$$

Iteration 3:

$$S_A^{(3)} = \begin{bmatrix} 1 & \mathbf{0.3431} & \mathbf{0} \\ 0.3431 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad S_B^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0.5173 & 0.5173 & 0.5173 \\ 0 & 0.5173 & 1 & 0.2347 & 0.8000 \\ 0 & 0.5173 & 0.2347 & 1 & 0.2347 \\ 0 & 0.5173 & 0.8000 & 0.2347 & 1 \end{bmatrix}$$

```
%-------------------------------------------------------------
% Tony Hyun Kim
% CS 246, PS 3, Problem 1
% Computing similarity by "hand"
%-------------------------------------------------------------
clear all;

% Adjacency matrix
R = [0 1 1 0 1;
     0 1 0 1 0;
     1 0 0 0 0];

% Initialize the similarity matrices
nA = 3; sA = eye(nA);
nB = 5; sB = eye(nB);

% Decay constants
C1 = 0.8;
C2 = 0.8;

Niter = 3;
for iter = 1:Niter

    % Compute the next sA matrix
    sAnew = 0.5*eye(nA);
    for i = 1:(nA-1)
        for j = (i+1):nA
            Oi = find(R(i,:));
            Oj = find(R(j,:));
            for oi = Oi
                for oj = Oj
                    sAnew(i,j) = sAnew(i,j) + sB(oi,oj);
                end
            end
            sAnew(i,j) = C1/(length(Oi)*length(Oj))*sAnew(i,j);
        end
    end
    sAnew = sAnew + sAnew'; % Symmetric matrix

    % Compute the next sB matrix
    sBnew = 0.5*eye(nB);
    for i = 1:(nB-1)
        for j = (i+1):nB
            Ii = find(R(:,i));
            Ij = find(R(:,j));
            for ii = Ii'
                for ij = Ij'
                    sBnew(i,j) = sBnew(i,j) + sA(ii,ij);
                end
            end
            sBnew(i,j) = C2/(length(Ii)*length(Ij))*sBnew(i,j);
        end
    end
    sBnew = sBnew + sBnew';

    % Batch update
    fprintf('Iter %d\n',iter);
    sA = sAnew;
    sB = sBnew;
    disp(sA);
    disp(sB);
end
```

1.2 Using weights

If we associate a weight $W_{(X,y)}$ with an edge (X, y) in the bipartite graph, a natural way to re-write the update equations is

$$s(X,Y) = \frac{C_1}{\sum_{i=1}^{|O(X)|}\sum_{j=1}^{|O(Y)|} W_{(X,O_i(X))} \cdot W_{(Y,O_j(Y))}} \sum_{i=1}^{|O(X)|}\sum_{j=1}^{|O(Y)|} W_{(X,O_i(X))}W_{(Y,O_j(Y))} \cdot s(O_i(X), O_j(Y))$$

and, similarly,

$$s(x,y) = \frac{C_2}{\sum_{i=1}^{|I(x)|}\sum_{j=1}^{|I(y)|} W_{(I_i(x),x)} \cdot W_{(I_j(y),y)}} \sum_{i=1}^{|I(x)|}\sum_{j=1}^{|I(y)|} W_{(I_i(x),x)}W_{(I_j(y),y)} \cdot s(I_i(x), I_j(y))$$

1.3 Similarities in $K_{2,1}$ and $K_{2,2}$

Similarity matrix of set A in $K_{2,1}$ after 3 iterations is

$$S_A^{(3)} = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix},$$

and in $K_{2,2}$ it is:

$$S_A^{(3)} = \begin{bmatrix} 1 & 0.6240 \\ 0.6240 & 1 \end{bmatrix}.$$

So, the similarity of the elements in A is higher in the smaller bipartite graph $K_{2,1}$ than $K_{2,2}$, contrary to the "intuitive" expectation.

1.4 Fix the problem

2 Dense Communities in Networks

2.1 Running time

2.1.1 Prove that at any iteration of the algorithm, $|A(S)| \geq \frac{\epsilon}{1+\epsilon}|S|$

We present a proof by contradiction. Suppose, at any iteration, we have $|A(S)| < \frac{\epsilon}{1+\epsilon}|S|$. Consider the complement set $S - A(S)$ consisting of elements $x \in S$ that satisfy $\deg_S(x) > 2(1+\epsilon)\rho(S)$. It follows that $S - A(S)$ contains at least $\frac{1}{1+\epsilon}|S|$ elements.

We then count the total degree of all elements in S:

$$\sum_{x\in S} \deg_S(x) \geq \sum_{x\in(S-A(S))} \deg_S(x) > \frac{1}{1+\epsilon}|S| \cdot 2(1+\epsilon)\rho(S) = 2|E(S)|,$$

where we have used the definition $\rho(S) = |E(S)|/|S|$. We find that the total degree of all nodes in S strictly exceeds $2|E(S)|$, which is impossible since each edge in $E(S)$ contributes exactly twice (at each endpoint) to the total degree $\sum_{x\in S} \deg_S(x)$. Hence, the initial assumption is false and we must have $|A(S)| \geq \frac{\epsilon}{1+\epsilon}|S|$.

2.1.2 Prove that the algorithm terminates in at most $\log_{1+\epsilon}(n)$ iterations

This is a simple extension of the previous result that $|A(S)| \geq \frac{\epsilon}{1+\epsilon}|S|$. Denote the size of the original graph by $|S_0|$. Since we remove $A(S)$ from S at each iteration, the size of the set after the n-th step is bounded by

$$|S_n| < \frac{1}{1+\epsilon}|S_{n-1}| < \left(\frac{1}{1+\epsilon}\right)^n |S_0|,$$

which, for $n \geq \log_{1+\epsilon}(|S_0|)$, yields $|S_n| < 1$. In other words, the set S_n is empty and the algorithm must terminate since there's no more work to do!

2.2 Quality of result

Our aim to show that the density of the set returned by the algorithm is at most a factor $2(1+\epsilon)$ smaller than $\rho^*(G)$. (So, larger ϵ yields poorer results but converges faster. Typical story.)

2.2.1 Step 1

Let S^* be the densest subgraph of G. For any $v \in S^*$, we wish to show that $\deg_{S^*}(v) \geq \rho^*(G) = \rho(S^*) = |E(S^*)|/|S^*|$. Again, we proceed by contradiction.

Suppose there exists a $v \in S^*$ such that $\deg_{S^*}(v) < \rho(S^*)$. Now consider the set S' obtained by removing v from S, *i.e.* $S' = S - \{v\}$. We compute the density of the new set S'

$$\rho(S') = \frac{|E(S')|}{|S'|} = \frac{|E(S^*)| - \deg_{S^*}(v)}{|S^*| - 1} > \frac{|E(S^*)| - \rho(S^*)}{|S^*| - 1},$$

where we have made use of the fact that, by removing v from S, we have deleted $\deg_{S^*}(v) < \rho(S^*)$ edges from $E(S^*)$. We now substitute $\rho(S^*) = |E(S^*)|/|S^*|$ to yield

$$\rho(S') > \frac{|E(S^*)| - \frac{|E(S^*)|}{|S^*|}}{|S^*| - 1} = \frac{\frac{|E(S^*)|}{|S^*|} \cdot (|S^*| - 1)}{|S^*| - 1} = \rho(S^*)$$

showing that S' is a denser subgraph of G than S^*, thereby contradicting the premise that S^* is the densest subgraph. Hence, we must have $\deg_{S^*}(v) \geq \rho(S^*)$.

2.2.2 Step 2

Note that all nodes of the original graph (including those belonging to S^*) will occur in $A(S)$ at some iteration. We consider the first iteration in which there exists a node $v \in S^* \cap A(S)$.

Since $v \in S^*$, we have $\deg_{S^*}(v) \geq \rho(S^*)$ by the result of Section 2.2.1. And since $v \in A(S)$, we also have that $\deg_S(v) \leq 2(1+\epsilon)\rho(S)$.

Now, we also have that $S^* \subseteq S$ since we are considering the first iteration for which $S^* \cap A(S) \neq \emptyset$. It follows that $\deg_S(v) \geq \deg_{S^*}(v)$. Combining all of these inequalities, we find

$$2(1+\epsilon)\rho(S) \geq \deg_S(v) \geq \deg_{S^*}(v) \geq \rho(S^*).$$

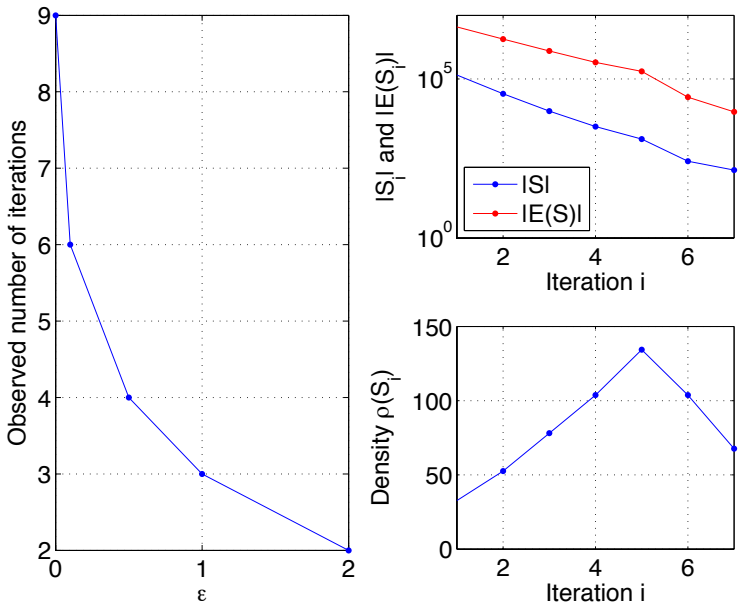

Figure 1: (Left column) Number of iterations required to find a single dense subgraph as a function of ϵ. The observed performance is significantly better than the theoretical bound. (Right) Variation in $\rho(S_i)$, $|E(S_i)|$, and $|S_i|$ over the algorithm execution, where i is the iteration of the while loop.

2.2.3 Step 3

Finally, we wish to show that the density of the returned set $\tilde{S}$ satisfies

$$\rho(\tilde{S}) \geq \frac{1}{2(1+\epsilon)}\rho(S^*).$$

This is a trivial consequence of Section 2.2.2. We have already shown that there exists an iteration for which $\rho(S) \geq \frac{1}{2(1+\epsilon)}\rho(S^*)$. Since the algorithm keeps track of (and returns) the set $\tilde{S}$ with the highest observed density, it follows immediately that $\rho(\tilde{S}) \geq \frac{1}{2(1+\epsilon)}\rho(S^*)$

2.3 Implementation of dense subgraph search

Fig. 1(a) shows the number of iterations to find a single dense subgraph, as a function of $\epsilon = \{0, 0.1, 0.5, 1, 2\}$. The algorithm seems to use significantly less steps than the theoretical bounds of Section 2.1. Fig. 1(b) shows the evolution of $\rho(S_i)$, $|E(S_i)|$, and $|S_i|$ over the algorithm execution, where i denotes the iteration of the main (while) loop.

I then repeated the algorithm to reveal 20 communities in the graph. Fig. 2 shows the density $\rho(\tilde{S}_j)$ of the j-th community. I present the remaining results, namely $|E(\tilde{S}_j)|$ and $|\tilde{S}_j|$, in Table 1, since they tend to vary somewhat wildly over j. This owes to the fact that the algorithm orders the results by density, rather than absolute edge or vertex count.

```c
//-------------------------------------------------------------
// Tony Hyun Kim
// CS 246, PS 3, Problem 2(c)
// Dense subgraph search
//-------------------------------------------------------------
#include <stdio.h>
#include <string.h>

int sum(int* v, int n)
{
  int acc = 0;
  for(int i=0; i<n; i++)
    acc += v[i];
  return acc;
}

float computeDensity(int *s, int *deg, int n) {
  return (0.5f*sum(deg,n))/sum(s,n);
}

int streamFile(int *s, int *deg, FILE* ifp)
{
  // Rewind file pointer to beginning
  rewind(ifp);
  int count = 0;
  int n1, n2;
  while(1)
  {
    count++;
    // Read a single edge
    fscanf(ifp,"%d %d",&n1,&n2);
    if(feof(ifp))
      return count;
    // Add edge to degree vector if it contained entirely within S
    if(s[n1]&&s[n2]) {
      deg[n1]++;
      deg[n2]++;
    }
  }
}

int main(int argc, char** argv)
{
  // Parameterize the problem
  char* source;
  int n;

//  source = "livejournal-undirected-small.txt"; n = 50
  source = "livejournal-undirected.txt"; n = 500000;
  float eps = 0.05; // Algorithm parameter

  printf("Input source: %s\n",source);
  printf("Expected node count: %d\n",n);
  printf("Algorithm epsilon: %.3f\n",eps);

  // No need to touch anything below this line
  //-------------------------------------------------------------
  FILE* ifp = fopen(source,"r");

  // Break if input file could not be opened
  if(ifp == NULL)
  {
    printf("Could not open file %s!\n",source);
    return 1;
  }

  // Declare and initialize algorithm state
  int s[n];
  int st[n];
```



```c
  int deg[n];
  int used[n]; // This is a flag that indicates that the variable has
               // been used in a previous community and should be skipped
  for(int i=0; i<n; i++)
    used[i] = 0;

  float rhoS, rhoSt; // Densities of S and Stilde

  // We now find K communities by repeated applications of the algorithm
  for(int K=0; K<20; K++)
  {
    //printf("K=%d community\n",K);
    // Initialize S and St. Do not include variable if it has occurred
    //  previously in a community
    for(int i=0; i<n; i++) {
      s[i]   = used[i] ? 0 : 1;
      st[i]  = s[i];
      deg[i] = 0;
    }

    streamFile(s,deg,ifp);
    rhoS  = computeDensity(s,deg,n);
    rhoSt = rhoS;

    int iter = 0;
    while(sum(s,n)>0)
    {
      //printf("  Iter %2d, |S| %6d, |E(S)| %8d rho(S) %3.2f rho(St) %3.2f\n",
      //          iter++,sum(s,n),sum(deg,n)/2,rhoS,rhoSt);

      // Remove A(S) from S (see definition in assignment)
      for(int i=0; i<n; i++)
        if(deg[i] <= 2*(1+eps)*rhoS)
          s[i] = 0;

      // Stream through file, using the new definition of S
      memset(deg,0,sizeof(int)*n);
      streamFile(s,deg,ifp);
      rhoS = computeDensity(s,deg,n);

      if(rhoS > rhoSt)
      {
        rhoSt = rhoS;
        memcpy(st,s,sizeof(int)*n);
      }
    }

    // St holds our community
    for(int i=0; i<n; i++)
      if(st[i])
        used[i] = 1;
    int stLen = sum(st,n);
    printf("K %2d rho(St) %3.2f |St| %4d |E(St)| %6d\n",K,rhoSt,stLen,(int)(2*rhoSt*st
Len));
  }

  fclose(ifp);

  // Optional: We now write the output Stilde
  FILE* ofp = fopen("Stilde.txt","w");
  for(int i=0; i<n; i++)
    if(st[i])
      fprintf(ofp,"%d\n",i);
  fclose(ofp);

  return 0;
}
```

3 PageRank Computation

3.1 L_1 convergence

We telescope the expression

$$\begin{aligned}
\pi - \pi^{(k)} &= \left(\frac{\epsilon}{n}\mathbf{1}^T + (1-\epsilon)\pi P\right) - \left(\frac{\epsilon}{n}\mathbf{1}^T + (1-\epsilon)\pi^{(k-1)}P\right), \\
&= (1-\epsilon)(\pi - \pi^{(k-1)})P, \\
&= (1-\epsilon)^2(\pi - \pi^{(k-2)})P^2, \\
&= \cdots \\
&= (1-\epsilon)^k(\pi - \pi^{(0)})P^k.
\end{aligned}$$

Recall now that the matrix P is row-stochastic, which implies that πP^k and $\pi^{(0)}P^k$ are discrete probability distribution vectors. The largest L_1-distance between two discrete probability vectors is 2, which occurs when the two vectors encode unit probabilities for two distinct outcomes. Thus,

$$\left|\left|\pi - \pi^{(k)}\right|\right|_1 \leq 2 \cdot (1-\epsilon)^k. \tag{1}$$

3.2 Running time of Power Iteration

First, we can rearrange Eq. 1 to find that, in order to bound the L_1 norm $\left|\left|\pi - \pi^{(k)}\right|\right|_1$ at some fixed $c < 1$, we have to perform

$$k > \frac{|\log(c/2)|}{\log(1/(1-\epsilon))}$$

iterations of Power Iteration. (Note that for $c < 1$, the quantity $\log(c/2)$ is negative. The above equation has been configured so that both the numerator and denominator are positive.)

Next, we consider the update equation in Power Iteration:

$$\pi^{(i)} = \frac{\epsilon}{n}\mathbf{1}^T + (1-\epsilon)\pi^{(i-1)}P.$$

This computation can be performed as follows:

1. Begin with an n-entry array $\pi^{(i)}$, where all entries are initialized to the value ϵ/n. This initialization is assumed to be free.
2. We perform the vector-matrix multiplication $(1-\epsilon)\pi^{(i-1)}P$. In the typical PageRank context, the (extremely large) $n \times n$ matrix P is sparse and has exactly m nonzero entries. Hence, this vector-matrix multiplication involves exactly m multiplications (and subsequent additions to the array $\pi^{(i)}$).

Hence, the overall time complexity is $O(km) = O(\frac{m}{\log(1/(1-\epsilon))})$.

Index j	$\rho(\tilde{S}_j)$	$\lvert\tilde{S}_j\rvert$	$\lvert E(\tilde{S}_j)\rvert$
0	134.35	1286	345560
1	87.53	238	41666
2	52.75	3989	420860
3	32.59	70	4561
4	26.12	45975	2401632
5	17.62	110	3875
6	14.28	60	1714
7	10.52	196	4122
8	9.62	21	403
9	10.39	23	478
10	9.17	812	14898
11	8.28	383	6346
12	8.85	39	690
13	8.12	164398	2670910
14	5.46	24	262
15	4.50	20	180
16	4.30	10	86
17	3.86	63	486
18	3.37	106	714
19	3.03	293	1778

Table 1: Properties of the top 20 dense communities (as identified by repeated application of our dense subgraph search algorithm) in 'livejournal-undirected.txt'.

3.3 Algorithm MC1

3.3.1 Expectation

Let us begin by finding a formal expression for the stationary PageRank distribution π. From the results of Section 3.1, we know that Power Iteration initialized at $\pi^{(0)}$ converges to the stationary distribution. We write the evolution of $\pi^{(i)}$ explicitly:

$$\begin{aligned}
\pi^{(0)} &= \frac{1}{n}\mathbf{1}^T \\
\pi^{(1)} &= \epsilon\frac{1}{n}\mathbf{1}^T + (1-\epsilon)\cdot\pi^{(0)}\cdot P \\
&= \epsilon\frac{1}{n}\mathbf{1}^T + (1-\epsilon)\cdot\frac{1}{n}\mathbf{1}^T\cdot P \\
\pi^{(2)} &= \epsilon\frac{1}{n}\mathbf{1}^T + (1-\epsilon)\cdot\pi^{(1)}\cdot P \\
&= \epsilon\frac{1}{n}\mathbf{1}^T + \epsilon(1-\epsilon)\cdot\frac{1}{n}\mathbf{1}^T\cdot P + (1-\epsilon)^2\cdot\frac{1}{n}\mathbf{1}^T\cdot P^2
\end{aligned}$$

and so on. By repeating the iteration rule repeatedly, we find the stationary distribution to be

$$\pi = \frac{1}{n}\mathbf{1}^T\cdot\left[\epsilon + \epsilon(1-\epsilon)P + \epsilon(1-\epsilon)^2P^2 + \epsilon(1-\epsilon)^3P^3 + \cdots\right]. \tag{2}$$

Now we consider the MC1 algorithm. For a single random walker, we define the binary random variable X_j

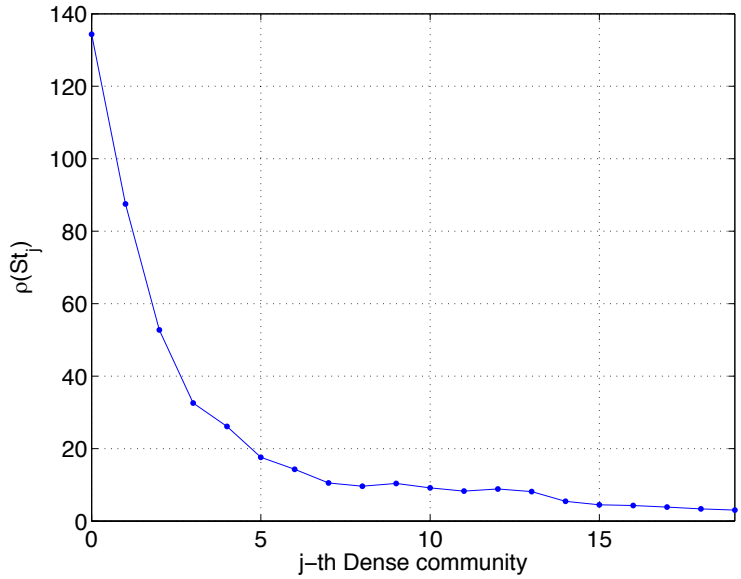

Figure 2: The density of the j-th community returned by repeated execution of the dense graph search algorithm. The density apparently trails off after the first 5 or so communities.

where $X_j = 1$ indicates that the walk terminated on node j. Note that the estimator $\hat{\pi}_{j,N}$ may be written in terms of X_j as:

$$\hat{\pi}_{j,N} = \frac{1}{N} \sum_{k=1}^{N} X_j^{(k)}, \tag{3}$$

where k indexes over the different instatiations of the random walk. The variables $X_j^{(k)}$ are i.i.d.

We may compute the probability $E(X_j)$ by considering all possible paths that terminate on node j of a given path length:

- Path length 0: The walk is initialized at the node j, and terminates immediately. This occurs with probability $\epsilon \frac{1}{n}$.
- Path length 1: We take one step and then terminate, which gives the factor $(1-\epsilon)\epsilon$. We also need to sum over the prior probability of all nodes that can walk (in a single step) to node j. Hence, the overall probability is $\epsilon(1-\epsilon)(\frac{1}{n}\mathbf{1}^T P)_j$, where we have made use of the fact that P implements a single step of our "random surfer" in the graph.

Similarly, we find that paths of length t have probability $\epsilon(1-\epsilon)^t(\frac{1}{n}\mathbf{1}^T P^t)_j$ of terminating on node j when initialized uniformly over the entire graph. Summing these disjoint probabilities, we find

$$E[X_j] = \epsilon(\frac{1}{n}\mathbf{1}^T)_j + \epsilon(1-\epsilon)(\frac{1}{n}\mathbf{1}^T P)_j + \epsilon(1-\epsilon)^2(\frac{1}{n}\mathbf{1}^T P^2)_j + \cdots,$$

which is precisely the j-th component of the stationary distribution π in Eq. 2. Since the variables $X_j^{(k)}$ are i.i.d., we obtain from Eq. 3 that $E[\hat{\pi}_{j,N}] = E[X_j^{(1)}] = \pi_j$.

3.3.2 Variance

The random variable X_j is a Bernoulli R.V. with probability $E(X_j) = \pi_j$ as shown in the previous section. Correspondingly, the variance of X_j is $\pi_j(1-\pi_j)$.

Using the i.i.d. property of $X_j^{(k)}$ in Eq. 3, it follows that $\mathrm{Var}(\hat{\pi}_{j,N}) = N^{-1}\pi_j(1-\pi_j)$.

3.4 Algorithm MC2

Similarly to before, we use the binary variable $X_{i,j} = 1$ to denote that the walk – now initialized at node i – ultimately terminated at node j.

The MC2 estimator $\hat{\hat{\pi}}$ may then be written

$$\hat{\hat{\pi}}_{j,N} = \frac{1}{N}\sum_{k=1}^{R}\sum_{i=1}^{n} X_{i,j}^{(k)} = \frac{1}{R}\sum_{k=1}^{R}\frac{1}{n}\sum_{i=1}^{n} X_{i,j}^{(k)}. \tag{4}$$

Now consider the probability $E(X_{i,j})$. This quantity can be computed by the same strategy as in Section 3.3.1, but where the prior distribution on the graph is not $\pi^{(0)} = \frac{1}{n}\mathbf{1}^T$ but the deterministic distribution $\rho_i^T = [0\ \cdots\ 0\ 1\ 0\ \cdots\ 0]$ which is nonzero only at the i-th node. This substitution yields:

$$E[X_{i,j}] = \epsilon(\rho_i^T)_j + \epsilon(1-\epsilon)(\rho_i^T P)_j + \epsilon(1-\epsilon)^2(\rho_i^T P^2)_j + \cdots,$$

which we then utilize to take the expectation of Eq. 4

$$\begin{aligned}
E[\hat{\hat{\pi}}_{j,N}] &= \frac{1}{R}\sum_{k=1}^{R}\frac{1}{n}\sum_{i=1}^{n} E[X_{i,j}^{(k)}] \\
&= \frac{1}{R}\sum_{k=1}^{R}\frac{1}{n}\sum_{i=1}^{n} \epsilon(\rho_i^T)_j + \epsilon(1-\epsilon)(\rho_i^T P)_j + \epsilon(1-\epsilon)^2(\rho_i^T P^2)_j + \cdots \\
&= \frac{1}{R}\sum_{k=1}^{R}\frac{1}{n}\sum_{i=1}^{n} \left\{\rho_i^T \cdot \left[\epsilon + \epsilon(1-\epsilon)P + \epsilon(1-\epsilon)^2P^2 + \cdots\right]\right\}_j.
\end{aligned}$$

Now, we recognize that the infinite series in the "innermost-loop" does not depend on the index i, and that the summation $\frac{1}{n}\sum_{i=1}^{n}\rho_i$ is equivalent to $\frac{1}{n}\mathbf{1}^T$. This yields

$$E[\hat{\hat{\pi}}_{j,N}] = \frac{1}{R}\sum_{k=1}^{R}\left\{\frac{1}{n}\mathbf{1}^T \cdot \left[\epsilon + \epsilon(1-\epsilon)P + \epsilon(1-\epsilon)^2P^2 + \cdots\right]\right\}_j = \frac{1}{R}\sum_{k=1}^{R}\pi_j = \pi_j.$$

So, MC2 algorithm also provides an unbiased estimate of the stationary PageRank distribution.

3.5 Algorithm MC3

The analysis is similar to that of MC2. Now let $\tilde{X}_{i,j}$ denote the number of times that a walk initialized at node i visits nodes j some time during the walk. We then have

$$E[\tilde{\pi}_j] = \frac{\epsilon}{nR}\cdot\sum_{k=1}^{R}\sum_{i=1}^{n} E[\tilde{X}_{i,j}]$$

The quantity $E[\tilde{X}_{i,j}]$ is given by

$$E[\tilde{X}_{i,j}] = (\rho_i^T)_j + (1-\epsilon)(\rho_i^T P)_j + (1-\epsilon)^2(\rho_i^T P^2)_j + \cdots ,$$

where ρ_i was defined in our discussion of MC2. Note that the only difference between $E[X_{i,j}]$ of MC2 and $E[\tilde{X}_{i,j}]$ of MC3 is the omission of factor ϵ in the latter, which follows since we are no longer requiring the walk to terminate at node j.

We then have

$$\begin{aligned} E[\tilde{\pi}_j] &= \frac{\epsilon}{nR} \cdot \sum_{k=1}^{R} \sum_{i=1}^{n} E[\tilde{X}_{i,j}] \\ &= \frac{1}{R} \sum_{k=1}^{R} \frac{1}{n} \sum_{i=1}^{n} \epsilon(\rho_i^T)_j + \epsilon(1-\epsilon)(\rho_i^T P)_j + \epsilon(1-\epsilon)^2(\rho_i^T P^2)_j + \cdots \end{aligned}$$

which was shown in our analysis of MC2 to be equivalent to π_j.

3.6 Running time of MC3

Given a termination probability ϵ, the length T of a single random walk is distributed geometrically as

$$p(T = t) = \epsilon(1-\epsilon)^t \qquad \text{for } t = 0, 1, 2, \cdots$$

whose expectation is $E(T) = 1/\epsilon$.

Since we are instantiating $N = nR$ random walks, and each step of the walk takes a unit amount of time, the expected running time of MC3 is $O(nR/\epsilon)$.

3.7 Numerical results

See Table 2. Indeed, for a fixed number of random walkers, MC3 provides the best performance against the true PageRank vector (estimated from Power Iteration). The comparison between MC2 and MC1, on the other hand, appears more subtle. I performed this problem in Matlab. The source code is attached.

```
%-------------------------------------------------------------
% Tony Hyun Kim
% CS 246, PS #3, Problem 3(g)
%-------------------------------------------------------------
clear all;

% Load the graph from the text file
source = 'graph.txt';
edges = load(source);
m = length(edges(:,1));

n = 100; % Number of nodes
global P;
P = zeros(n,n);

% Fill in the edges
for i = 1:m
    edge = edges(i,:);
    P(edge(1),edge(2)) = 1;
end

% Now renormalize the rows of P by degree
for i = 1:n
    p = P(i,:);
    P(i,:) = p/sum(p);
end

% Run Power Iteration
%-------------------------------------------------------------
global eps;
eps = 0.2;
Niter = 40;
pis = zeros(1+Niter,n);
pi0 = 1/n*ones(1,n);
pis(1,:) = pi0;

ts = zeros(1,Niter);
for j = 1:Niter
    tic;
    pis(1+j,:) = eps*pi0+(1-eps)*pis(j,:)*P;
    ts(j) = toc;
end
fprintf('Power iteration: %d iterations, %.3f ms per iter, %.3f ms total\n',...
    Niter,mean(ts)*1e3,sum(ts)*1e3);

pi = pis(end,:); % The (approximate) PageRank distribution

% Sort the nodes
[~, rank] = sort(pi,'descend');

% MC3 algorithm
%-------------------------------------------------------------
R = 5;

pi_mc3 = zeros(1,n);

count = 0;
time = 0;
for i = 1:n % Fix initial node
    for j = 1:R % Iterations
        tic;

        % Random surfer (MC3)
        xi = i;
        pi_mc3(xi) = pi_mc3(xi)+1;
        while(rand>eps) % Keep on surfing
           p = cumsum(P(xi,:));
           xi = find(rand<p,1);
```



```
           pi_mc3(xi) = pi_mc3(xi)+1;
        end

        time  = time + toc;
        count = count + 1;
    end
end
fprintf('MC3 with R=%d: %.3f ms per walk, %.3f ms total\n',...
    R,time/count*1e3,time*1e3);

% Normalize result
pi_mc3 = eps/(n*R)*pi_mc3;

% Compute errors
for c = [10 30 50 100]
    e = abs(pi(rank(1:c))-pi_mc3(rank(1:c)));
    fprintf('Top-%d error: %.3f\n',c,mean(e));
end

% MC2 algorithm
%-------------------------------------------------------------
R = 5;

pi_mc2 = zeros(1,n);

count = 0;
time = 0;
for i = 1:n % Fix initial node
    for j = 1:R % Iterations
        tic;

        % Random surfer (MC2)
        xi = i;
        while(rand>eps) % Keep on surfing
           p = cumsum(P(xi,:));
           xi = find(rand<p,1);
        end
        pi_mc2(xi) = pi_mc2(xi) + 1;

        time  = time + toc;
        count = count + 1;
    end
end
fprintf('MC2 with R=%d: %.3f ms per walk, %.3f ms total\n',...
    R,time/count*1e3,time*1e3);

% Normalize result
pi_mc2 = 1/(n*R)*pi_mc2;

% Compute errors
for c = [10 30 50 100]
    e = abs(pi(rank(1:c))-pi_mc2(rank(1:c)));
    fprintf('Top-%d error: %.3f\n',c,mean(e));
end

% MC1 algorithm
%-------------------------------------------------------------
R = 5;
N = n*R;

pi_mc1 = zeros(1,n);

count = 0;
time = 0;
for i = 1:N
    tic;

    % Random surfer (MC1)
```



```
    xi = randi(n);
    while(rand>eps) % Keep on surfing
       p = cumsum(P(xi,:));
       xi = find(rand<p,1);
    end
    pi_mc1(xi) = pi_mc1(xi) + 1;

    time  = time + toc;
    count = count + 1;
end
fprintf('MC1 with R=%d: %.3f ms per walk, %.3f ms total\n',...
    R,time/count*1e3,time*1e3);

% Normalize result
pi_mc1 = 1/(n*R)*pi_mc1;

% Compute errors
for c = [10 30 50 100]
    e = abs(pi(rank(1:c))-pi_mc1(rank(1:c)));
    fprintf('Top-%d error: %.3f\n',c,mean(e));
end
```

4 Latent Features for Recommendations

4.1 Update equations of stochastic descent

The error (with regularization) is defined as

$$E = \sum_{(u,i)\in \text{Ratings}} (R_{iu} - q_i \cdot p_u^T)^2 + \lambda \left[\sum_u ||p_u||^2 + \sum_i ||q_i||^2 \right] \tag{5}$$

where q_i is the i-th row of Q, and p_u is the u-th row of P. Note: at the time of writing, there are some inconsistencies in the definitions as provided by the problem set (*e.g.* is p_u a row- or column-vector, ϵ_{ui} vs. ϵ_{iu}, etc.). In these ambiguous cases, I will just "do the right thing."

- The error for any pair (i, u) is: $\epsilon_{iu} = R_{iu} - q_i \cdot p_u$.
- The stochastic update equation for q_i is: $q_i \leftarrow q_i + \eta \cdot (\epsilon_{iu} p_u - \lambda q_i)$. (This is basically a derivative of the error E *with respect to a single example* with respect to q_i.)
- Likewise, the stochastic update equation for p_u is: $p_u \leftarrow p_u + \eta \cdot (\epsilon_{iu} q_i - \lambda p_u)$.

4.2 Stochastic latent features implementation

In our dataset, there are $n = 943$ users and $m = 1682$ movies. Fig. 3 shows a number of different learning rates η that I have tried. I find $\eta = 0.01$ to be a fairly good candidate. (I've noticed that larger values for η can often lead to divergence!)

My Matlab implementation is attached on the following page.

4.3 Training and test errors as function of latent factor dimension

Fig. 4 shows the training and test errors as a function of latent factor dimension k, at different values of the regularization parameter λ.

The three valid statements are:

- Regularization **decreases** the test error for $k \geq 5$.
- Regularization **increases** the training error for all k.
- Regularization **decreases** overfitting.

```
%-------------------------------------------------------------
% Tony Hyun Kim
% CS 246, PS #3, Problem 4(b)
% Latent factor implementation
%-------------------------------------------------------------
clear all;

% Data parameters
%-------------------------------------------------------------
m = 1682; % Number of movies
n = 943;  % Number of users

% Algorithm parameters
%-------------------------------------------------------------
k = 20;     % Latent factor dimension
lam = 0.2;  % Regularization
eta = 0.001;  % Learning rate

% User/item representation in latent space
%-------------------------------------------------------------
Q = rand(m,k); % Movie
P = rand(n,k); % User

Niter = 40;
E = zeros(1,Niter);
source = 'ratings.train.txt';
counter = 0;
verbose = 0;
for iter = 1:Niter
    % We are "pretending" that the data is sufficiently large so
    %   that we must stream the entries of R and perform stochastic
    %   gradient descent.
    tic;
    fid = fopen(source);
    tline = fgetl(fid);

    while ischar(tline)
        counter = counter + 1;

        % Parse a single line of data from file in the format
        %   [UserID MovieID Rating]
        R = sscanf(tline,'%d %d %d');

        % Stochastic update equations
        p_old = P(R(1),:);
        q_old = Q(R(2),:);
        error = R(3)-q_old*p_old';

        if(verbose)
            fprintf('Iter %3d, counter %09d: user %04d movie %04d rating %d error %02.3f\n',...
                iter,counter,R(1),R(2),R(3),error);
        end

        if(isnan(error))
            fprintf('Detected divergence after %d stochastic updates!\n',counter);
            return;
        end

        p_new = p_old + eta*(error*q_old - lam*p_old);
        q_new = q_old + eta*(error*p_old - lam*q_old);

        P(R(1),:) = p_new;
        Q(R(2),:) = q_new;

        tline = fgetl(fid); % Read next line
    end
    fclose(fid);
```



```
    t1 = toc;

    % Compute the error (including regularization)
    tic;
    fid = fopen(source);
    tline = fgetl(fid);
    while ischar(tline)
        R = sscanf(tline,'%d %d %d');
        p = P(R(1),:);
        q = Q(R(2),:);
        E(iter) = E(iter) + (R(3)-q*p')^2;
        tline = fgetl(fid);
    end
    fclose(fid);

    % Regularization terms
    for u = 1:n
        p = P(u,:);
        E(iter) = E(iter) + lam*(p*p');
    end
    for i = 1:m
        q = Q(i,:);
        E(iter) = E(iter) + lam*(q*q');
    end
    t2 = toc;

    fprintf('Iteration %d: Elapsed time %.1f s + %.1f s = %.1f s\n',...,
        iter,t1,t2,t1+t2);
end

% Automatically save
 savename = sprintf('eta%01d_%04d.mat',floor(eta),floor(rem(eta,1)*10000));
 save(savename);
```

Method	Total elapsed time (ms)	Average absolute error			
Power Iteration (40)	3.1	Top 10	Top 30	Top 50	All
MC3 ($R = 1$)	17.4	0.005	0.004	0.003	0.003
MC3 ($R = 3$)	49.0	0.004	0.003	0.002	0.002
MC3 ($R = 5$)	80.0	0.002	0.001	0.001	0.001
MC2 ($R = 1$)	14.9	0.014	0.011	0.010	0.008
MC2 ($R = 3$)	41.2	0.014	0.008	0.006	0.005
MC2 ($R = 5$)	64.6	0.004	0.004	0.003	0.003
MC1 ($N = 100$)	14.5	0.017	0.011	0.010	0.008
MC1 ($N = 300$)	43.0	0.009	0.007	0.005	0.004
MC1 ($N = 500$)	66.8	0.005	0.004	0.003	0.003

Table 2: Comparison of the Power Iteration method for computing PageRank against Monte-Carlo methods.

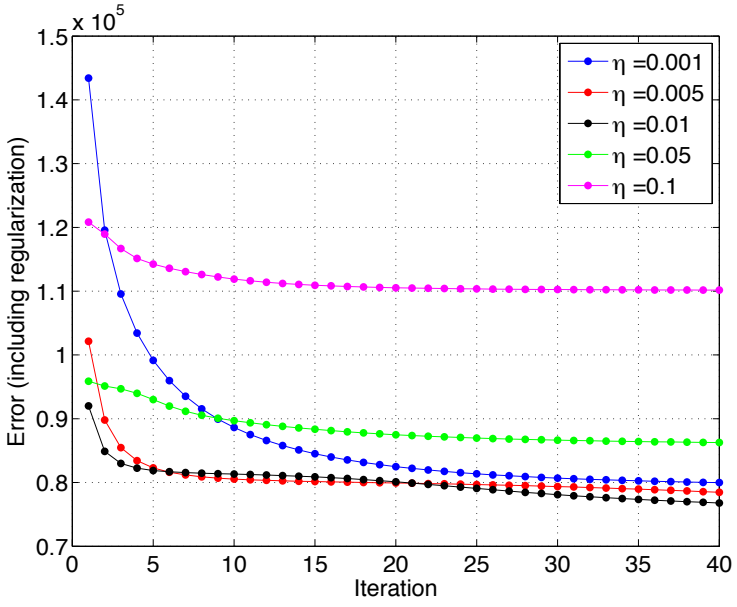

Figure 3: Decrease in the error function (with regularization) as a function of the learning rate η. For these runs, $k = 20$ and $\lambda = 0.2$. It appears that $\eta = 0.01$ is suitable for this particular dataset.

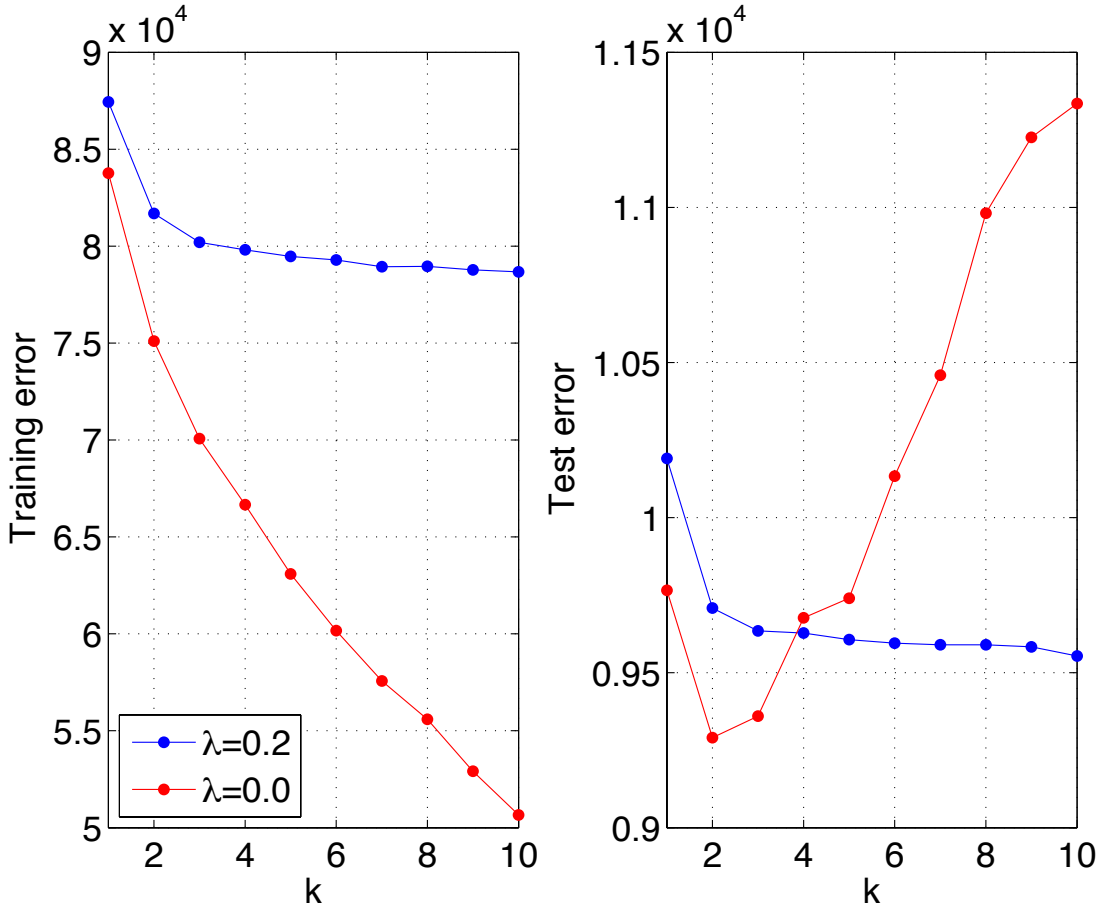

Figure 4: Variation in the training and test errors as a function of latent factor dimension k, for different values of the regularization parameter λ. Regularization counteracts overfitting of the training set, and yields better generalization performance (*i.e.* on the test set).