CS246: Mining Massive Data Sets **Winter 2026**

Problem Set 3

Please read the homework submission policies at http://cs246.stanford.edu.

1 Dead ends in PageRank computations (25 points)

Let the matrix of the Web $M \in \mathbb{R}^{n \times n}$ be an n-by-n matrix where n is the number of Web pages. The entry m_{ij} in row i and column j is 0, unless there is a link from node (page) j to node i. In that case, the value of m_{ij} is $1/k$, where k is the number of links out of node j (also the degree of node j). Notice that if node j has 3 links out, then column j has 3 values of 1/3 and the rest 0's. If node j is a *dead end* (i.e., it has zero links out), then the column j is all 0's.

Let $\mathbf{r} = [r_1, r_2, \ldots, r_n]^T$ be (an estimate of) the PageRank vector; that is, r_i is the estimate of the PageRank of node i. Define $w(\mathbf{r})$ to be the sum of the components of $\mathbf{r}$; that is $w(\mathbf{r}) = \sum_{i=1}^{n} r_i$.

In one iteration of the PageRank algorithm, we compute the next estimate $\mathbf{r}'$ of the PageRank as: $\mathbf{r}' = M\mathbf{r}$. Specifically, for each i we compute $r'_i = \sum_{j=1}^{n} M_{ij} r_j$. Define $w(\mathbf{r}')$ to be the sum of components of r'; that is $w(\mathbf{r}') = \sum_{i=1}^{n} r'_i$.

You may use D (the set of dead nodes) in your equation.

(a) [6pts]

Suppose the Web has no dead ends, that is $D = \emptyset$. Prove that $w(\mathbf{r}') = w(\mathbf{r})$.

★ Solution ★

(b) [9pts]

Suppose there are still no dead ends, but we use a teleportation probability of $1 - \beta$ where we teleport to a uniformly random node ($0 < \beta < 1$). The expression for the next estimate of r_i becomes $r'_i = \beta(\sum_{j=1}^{n} M_{ij} r_j) + (1 - \beta)/n$. For a fixed β, under what circumstances will $w(\mathbf{r}') = w(\mathbf{r})$? Prove your conclusion.

★ Solution ★

(c) [10pts]

Now let us assume there are one or more dead ends. Call a node "dead" if it is a dead end and "live" if not. At each iteration, for each node j: if j is live, then with probability β we follow an outgoing link according to column j of M, and with probability $1-\beta$ we teleport uniformly at random; if j is dead, we teleport uniformly at random with probability 1. Assume $w(\mathbf{r}) = 1$.

Write the equation for r'_i in terms of β, M, $\mathbf{r}$, n, and D (where D is the set of dead nodes). Then, prove that $w(\mathbf{r}')$ is also 1.

★ **Solution** ★

What to submit

(i) Proof [1(a)]

(ii) Condition for $w(\mathbf{r}') = w(\mathbf{r})$ and Proof [1(b)]

(iii) Equation for r'_i and Proof [1(c)]

2 Implementing PageRank and HITS (30 points)

In this problem, you will learn how to implement the PageRank and HITS algorithms in Spark. The general computation should be done in Spark, and you may also include numpy operations whenever needed. You will be experimenting with a small randomly generated graph (assume graph has no dead-ends) provided at `graph-full.txt`.

There are 100 nodes ($n = 100$) in the small graph and 1000 nodes ($n = 1000$) in the full graph, and $m = 8192$ edges, 1000 of which form a directed cycle (through all the nodes) which ensures that the graph is connected. It is easy to see that the existence of such a cycle ensures that there are no dead ends in the graph. There may be multiple directed edges between a pair of nodes, and your solution should treat them as the same edge. The first column in `graph-full.txt` refers to the source node, and the second column refers to the destination node.

Implementation hint: You may choose to store the PageRank vector $\mathbf{r}$ either in memory or as an RDD. Only the matrix M of links is too large to store in memory, and you are allowed to store matrix M in an RDD. e.g. data = sc.textFile("graph−full.txt"). On an actual cluster, an RDD is partitioned across the nodes of the cluster. However, you cannot then M = data.collect() which fetches the entire RDD to a single machine at the driver node and stores it as an array locally.

(a) PageRank Implementation [15 points]

Assume the directed graph $G = (V, E)$ has n nodes (numbered $1, 2, \ldots, n$) and m edges, all nodes have positive out-degree (each node has at least one out-going edge). $M = [M_{ji}]_{n \times n}$ is an $n \times n$ matrix as defined in class such that for any $i, j \in [\![1, n]\!]$:

$$M_{ji} = \begin{cases} \frac{1}{\deg(i)} & \text{if } (i \to j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Here, $\deg(i)$ is the number of outgoing edges of node i in G. If there are multiple edges in the same direction between two nodes, treat them as a single edge. By the definition of PageRank, assuming $1 - \beta$ to be the teleport probability, and denoting the PageRank vector by the column vector r, we have the following equation:

$$\mathbf{r} = \frac{1 - \beta}{n}\mathbf{1} + \beta M \mathbf{r}, \tag{1}$$

Based on this equation, the iterative procedure to compute PageRank works as follows:

1. Initialize: $\mathbf{r}^{(0)} = \frac{1}{n}\mathbf{1}$
2. For i from 1 to k, iterate: $\mathbf{r}^{(i)} = \frac{1-\beta}{n}\mathbf{1} + \beta M \mathbf{r}^{(i-1)}$

Run the aforementioned iterative process in Spark for 40 iterations (assuming $\beta = 0.8$) and obtain the PageRank vector r. In particular, you don't have to implement the blocking algorithm from lecture. The matrix M can be large and should be processed as an RDD in your solution.

Compute the PageRank scores and report the node id for the following using `graph-full.txt`:

- List the top 5 node ids with the highest PageRank scores.
- List the bottom 5 node ids with the lowest PageRank scores.

For a sanity check, we have provided a smaller dataset (`graph-small.txt`). In that dataset, the top node has id 53 with value 0.036 (nodes are 1-indexed so this is the 53rd node). Note that the `graph-small.txt` dataset is only provided for sanity check purpose. Your write-up should include results obtained using `graph-full.txt` (for both part (a) and (b)).

★ Solution ★

(b) HITS Implementation [15 points]

Assume the directed graph $G = (V, E)$ has n nodes (numbered $1, 2, \ldots, n$) and m edges, all nodes have non-negative out-degree (this is different from part a), and $L = [L_{ij}]_{n \times n}$ is a an $n \times n$ matrix referred to as the *link matrix* such that for any $i, j \in [\![1, n]\!]$:

$$L_{ij} = \begin{cases} 1 & \text{if } (i \to j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Given the link matrix L and some scaling factors λ, μ, the hubbiness vector h and the authority vector a can be expressed using the equations:

$$h = \lambda La, a = \mu L^T h \tag{2}$$

where $\mathbf{1}$ is the $n \times 1$ vector with all entries equal to 1.

Based on this equation, the iterative method to compute h and a is as follows:

1. Initialize h with a column vector (of size $n \times 1$) of all 1's.
2. Compute $a = L^T h$ and scale so that the largest value in the vector a has value 1.
3. Compute $h = La$ and scale so that the largest value in the vector h has value 1.
4. Go to step 2.

Repeat the iterative process for 40 iterations, assume that $\lambda = 1, \mu = 1$ and then obtain the hubbiness and authority scores of all the nodes (pages). The link matrix L can be large and should be processed as an RDD. You may store a and h in memory or as an RDD. Compute the following using `graph-full.txt`:

- List the 5 node ids with the highest hubbiness score.
- List the 5 node ids with the lowest hubbiness score.
- List the 5 node ids with the highest authority score.
- List the 5 node ids with the lowest authority score.

For a sanity check, you should confirm that `graph-small.txt` has highest hubbiness node id 59 with value 1 and highest authority node id 66 with value 1.

★ Solution ★

What to submit

(i) List 5 node ids with the highest and least PageRank scores [2(a)] using `graph-full.txt`

(ii) List 5 node ids with the highest and least hubbiness and authority scores [2(b)] using `graph-full.txt`

(iii) Upload all the code via Gradescope [2(a) & 2(b)]

3 Clique-Based Communities (25 points)

Imagine an undirected graph G with nodes $2, 3, 4, \ldots, 1000000$. (Note that there is no node 1.) There is an edge between nodes i and j if and only if i and j have a common factor other than 1. Put another way, the only edges that are missing are those between nodes that are relatively prime; e.g., there is no edge between 15 and 56.

We want to find communities by starting with a clique (not a bi-clique) and growing it by adding nodes. However, when we grow a clique, we want to keep the density of edges at 1; i.e., the set of nodes remains a clique at all times. A *maximal clique* is a clique for which it is impossible to add a node and still retain the property of being a clique; i.e., a clique C is maximal if every node not in C is missing an edge to at least one member of C.

Let C_i be the set of nodes of G that are divisible by i, where i is a positive integer.

(a) [5 points]

Prove that C_i is a clique for any $i > 1$.

★ Solution ★

(b) [10 points]

Under what circumstances is C_i a maximal clique? Prove that your conditions are both necessary and sufficient. (Trivial conditions, like "C_i is a maximal clique if and only if C_i is a maximal clique," will receive no credit.) ★ Solution ★

(c) [10 points]

Prove that C_2 is the unique largest clique. That is, it has more elements than any other clique. (**Note**: Not all cliques are in the form of C_i.)

★ Solution ★

What to submit

(i) Proof that the specified nodes are a clique.

(ii) Necessary and sufficient conditions for C_i to be a maximal clique, with proof.

(iii) Proof that C_2 is the unique largest clique.

4 Dense Communities in Networks (20 points)

In this problem, we study the problem of finding dense communities in networks.

Definitions: Assume $G = (V, E)$ is an undirected graph (e.g., representing a social network).

- For any subset $S \subseteq V$, we let the *induced edge set* (denoted by $E[S]$) to be the set of edges both of whose endpoints belong to S.
- For any $v \in S$, we let $\deg_S(v) = |\{u \in S | (u, v) \in E\}|$.
- Then, we define the *density* of S to be:

$$\rho(S) = \frac{|E[S]|}{|S|}.$$

- Finally, the *maximum density* of the graph G is the density of the densest induced subgraph of G, defined as:

$$\rho^*(G) = \max_{S \subseteq V} \{\rho(S)\}.$$

Goal. Our goal is to find an induced subgraph of G whose density is not much smaller than $\rho^*(G)$. Such a set is very densely connected, and hence may indicate a community in the network represented by G. Also, since the graphs of interest are usually very large in practice, we would like the algorithm to be highly scalable. We consider the following algorithm:

Require: $G = (V, E)$ and $\epsilon > 0$
  $\tilde{S}, S \leftarrow V$
  while $S \neq \emptyset$ **do**
    $A(S) := \{i \in S \mid \deg_S(i) \leq 2(1 + \epsilon)\rho(S)\}$
    $S \leftarrow S \setminus A(S)$

if $\rho(S) > \rho(\tilde{S})$ **then**
$\tilde{S} \leftarrow S$
end if
end while
return $\tilde{S}$

The basic idea in the algorithm is that the nodes with low degrees do not contribute much to the density of a dense subgraph, hence they can be removed without significantly influencing the density.

We analyze the quality and performance of this algorithm. We start with analyzing its performance.

Intuition behind the definitions and the algorithm. For a subset of vertices $S \subseteq V$, we focus only on the edges whose endpoints both lie in S. The quantity $\deg_S(v)$ denotes the number of neighbors of vertex v that also belong to S, i.e., the degree of v in the subgraph induced by S.

The density $\rho(S) = \frac{|E[S]|}{|S|}$ measures how many edges the induced subgraph has per vertex. Equivalently, note that

$$2\rho(S) = \frac{2|E[S]|}{|S|}$$

is exactly the *average degree* of the subgraph induced by S. Thus, $\rho(S)$ is proportional to how well-connected the vertices in S are on average.

The maximum density $\rho^*(G)$ is the highest density achievable by any induced subgraph of G, and represents the densest community present in the graph.

At each iteration, the algorithm removes vertices whose degree within the current set S is not significantly larger than the average degree of the induced subgraph. Specifically, the set

$$A(S) := \{i \in S \mid \deg_S(i) \leq 2(1+\epsilon)\rho(S)\}$$

contains vertices whose degree is at most $(1+\epsilon)$ times the average degree $2\rho(S)$.

(a) [10 points]

We show through the following steps that the algorithm terminates in a logarithmic number of steps.

i. Prove that at any iteration of the algorithm, $|A(S)| > \frac{\epsilon}{1+\epsilon}|S|$, in which S refers to the set at the start of the iteration before any updates occur.

ii. Prove that the algorithm terminates in $O(\log_{1+\epsilon}(n))$ iterations, where n is the initial number of nodes.

★ Solution ★

(b) [10 points]

We show through the following steps that the density of the set returned by the algorithm is at most a factor $2(1+\epsilon)$ smaller than $\rho^*(G)$.

i. Assume S^* is the densest subgraph of G. Prove that for any $v \in S^*$, we have: $\deg_{S^*}(v) \geq \rho^*(G)$.

ii. Consider the **first** iteration of the while loop in which, (for the first time), there exists a node $v \in S^* \cap A(S)$. Prove that $2(1+\epsilon)\rho(S) \geq \rho^*(G)$.

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

★ Solution ★

What to submit

(a) i. Proof of $|A(S)| \geq \frac{\epsilon}{1+\epsilon}|S|$.
ii. Proof of number of iterations for algorithm to terminate.

(b) i. Proof of $\deg_{S^*}(v) \geq \rho^*(G)$.
ii. Proof of $2(1+\epsilon)\rho(S) \geq \rho^*(G)$.
iii. Conclude that $\rho(\tilde{S}) \geq \frac{1}{2(1+\epsilon)}\rho^*(G)$.

5 Learning Embeddings (10 points)

In this problem, we want to explore learning embeddings with Singular Value Decomposition. We have a corpus of 10 words and a set of 15 documents from a Conservation Zoo. We can represent the documents and the corpus of words in matrix form as

	zoo	kangaroo	monkey	alligator	tiger	camel	eagle	lemur	dragon	pizza
doc 1	51	92	14	71	60	20	82	86	74	74
doc 2	87	99	23	2	21	52	1	87	29	37
doc 3	1	63	59	20	32	75	57	21	88	48
doc 4	90	58	41	91	59	79	14	61	61	46
doc 5	61	50	54	63	2	50	6	20	72	38
doc 6	17	3	88	59	13	8	89	52	1	83
doc 7	91	59	70	43	7	46	34	77	80	35
doc 8	49	3	1	5	53	3	53	92	62	17
doc 9	89	43	33	73	61	99	13	94	47	14
doc 10	71	77	86	61	39	84	79	81	52	23
doc 11	25	88	59	40	28	14	44	64	88	70
doc 12	8	87	0	7	87	62	10	80	7	34
doc 13	34	32	4	40	27	6	72	71	11	33
doc 14	32	47	22	61	87	36	98	43	85	90
doc 15	34	64	98	46	77	2	0	4	89	13

You will first decompose the matrix above. You are allowed to use any SVD solver. We recommend using numpy.linalg.svd(). Feel free to use the below code to recreate the matrix above in memory:

```
import numpy as np

np.random.seed(42)
x = np.random.randint(0,100, size=(15,10))
```

Hint In this problem, an *embedding* refers to a low-dimensional vector (r-dimension) representation of a document or a word. The goal of learning embeddings is to represent high-dimensional data (such as word-count vectors) in a smaller vector space while preserving important patterns, such as which words tend to appear together in the same documents. In this embedded space, documents or words with similar usage patterns are expected to have similar vector representations.

(a) [5 Points]

Task: Using SVD and $r = 9$, compute the embedding for doc 2, 10, and 15. Round to the nearest 2nd decimal.

★ Solution ★

(b) [5 Points]

Task: Using SVD and $r = 9$, compute the embedding for "kangaroo", "tiger", and "pizza". Round to the nearest 2nd decimal.

★ Solution ★

(i) Values from [part (a)]

(ii) Values from [part (b)]

(iii) Submit your code for both parts a and b to Gradescope