CS246: Mining Massive Data Sets Spring 2023

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 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 an arc from node (page) j to node i. In that case, the value of m_{ij} is $1/k$, where k is the number of arcs (links) out of node j. Notice that if node j has $k > 0$ arcs out, then column j has k values of $1/k$ and the rest 0's. If node j is a *dead end* (i.e., it has zero arcs out), then 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. Prove that $w(\mathbf{r}') = w(\mathbf{r})$.

(b) [9pts]

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

(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, we teleport from live nodes with probability $1 - \beta$ and teleport from dead nodes with probability 1. In both cases, we choose a random node uniformly to teleport to. 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.

What to submit

(a) Proof [1(a)]

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

(c) 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, and $M = [M_{ji}]_{n\times n}$ is a 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. 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)). Some key spark functions that *may* be relevant to your implementation are the following: `map()`, `distinct()`, `groupByKey()`, `cache()`, `count()`, `flatMap()`, `reduceByKey()`

(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, 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. 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.

Some key spark functions that *may* be relevant to your implementation are the following: `map(), mapValues(), distinct(), groupByKey(), cache()`

What to submit

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

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

(c) 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 $i > 1$.

(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.)

(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)

What to submit

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

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

(c) 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.

(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, $|A(S)| \geq \frac{\epsilon}{1+\epsilon}|S|$.

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

(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 there exists a node $v \in S^* \cap A(S)$. Prove that $2(1+\epsilon)\rho(S) \geq \rho^*(G)$.

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

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 Graph Neural Network (15 points)

Graph Neural Networks (GNNs) are a class of neural network architectures used for deep learning on graph-structured data. Broadly, GNNs aim to generate high-quality embeddings of nodes by iteratively aggregating feature information from local graph neighborhoods using neural networks; embeddings can then be used for recommendations, classification, link prediction, or other downstream tasks.

Let $G = (V, E)$ denote a graph with node feature vectors X_u for $u \in V$. To generate the embedding for a node u, we use the neighborhood of the node as the computation graph. At every layer l, for each pair of nodes $u \in V$ and its neighbor $v \in V$, we compute a message function via neural networks, and apply a convolutional operation that aggregates the messages from the node's local graph neighborhood, and updates the node's representation at the next layer. By repeating this process through K GNN layers, we capture feature and structural information from a node's local K-hop neighborhood. For each of the message computation, aggregation, and update functions, the learnable parameters are shared across all nodes in the same layer.

We initialize the feature vector for node X_u based on its individual node attributes. If we already have outside information about the nodes, we can embed that as a feature vector.

Otherwise, we can use a constant feature (vector of 1) or the degree of the node as the feature vector.

These are the key steps in each layer of a GNN:

- Message computation: We use a neural network to learn a message function between nodes. For each pair of nodes u and its neighbor v, the neural network message function can be expressed as $M\left(h_u^k, h_v^k, e_{u,v}\right)$.

Here h_u^k refers to the hidden representation of node u at layer k, and $e_{u,v}$ denotes available information about the edge (u, v), like the edge weight or other features.

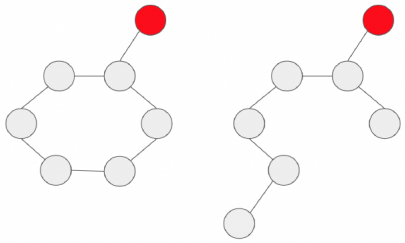
- Aggregation: At each layer, we apply a function to aggregate information from all of the neighbors of each node. The aggregation function is usually permutation invariant, to reflect the fact that nodes' neighbors have no canonical ordering.

- Update: We update the representation of a node based on the aggregated representation of the neighborhood.

- Pooling: The representation of an entire graph can be obtained by adding a pooling layer at the end. The simplest pooling methods are just taking the mean, max, or sum of all of the individual node representations. This is usually done for the purposes of graph classification.

(a) Effect of Depth on Expressiveness [3pts]

Consider the following 2 graphs



, where all nodes have 1-dimensional initial feature vector $x = [1]$. We use a simplified version of GNN, with no nonlinearity, no learned linear transformation, and sum aggregation. Specifically, at every layer, the embedding of node v is updated as the sum over the embeddings of its neighbors ($\mathcal{N}_v$) and its current embedding h_v^k to get h_v^{k+1}. We run the GNN to compute node embeddings for the 2 red nodes respectively. Note that the 2 red nodes have different 4-hop neighborhood structure (note this is not the minimum number of hops for which the neighborhood structure of the 2 nodes differs). How many layers of message passing are needed so that these 2 nodes can be distinguished (i.e., have different GNN embeddings)?

(b) Neighborhood Aggregation with Janossy Pooling [5pts]

Neighborhood aggregation is an important operation in graph neural networks. At each layer k, the embeddings of neighboring nodes are aggregated as:

$$h_{\mathcal{N}(v)}^k = \text{aggregate}\left(\left\{h_u^{k-1}, \forall u \in \mathcal{N}(v)\right\}\right)$$

In this question, we consider an alternative aggregation scheme, the **Janossy pooling**. The definition of Janossy pooling requires a *permutation function*, $\pi \in \Pi$, that maps the set $\{h_u, \forall u \in \mathcal{N}(v)\}$ to a specific sequence $\left(h_{u_1}, h_{u_2}, \cdots, h_{u_{|\mathcal{N}(v)|}}\right)_\pi$. In other words, π takes the unordered set of neighbor embeddings, and places these embeddings in a sequence based on a particular ordering. Another necessary component is a function, ρ_ϕ, that *operates on sequences*. In practice, it is usually chosen to be an LSTM.

The Janossy pooling approach then performs neighborhood aggregation by:

$$\text{aggregate}_{\text{janossy}}\left(\left\{h_u^{k-1}, \forall u \in \mathcal{N}(v)\right\}\right) = \text{MLP}_\theta\left(\frac{1}{|\Pi|}\sum_{\pi\in\Pi}\rho_\phi\left(h_{u_1}^{k-1}, \cdots, h_{u_{\mathcal{N}(v)}}^{k-1}\right)_\pi\right)$$

where the summation is done over all possible permutations Π of the inputs. For our purposes in this problem, we drop the MLP. Describe a function, ρ_ϕ, that reduces Janossy pooling to mean aggregation. What does that imply about the expressivity of Janossy pooling compared to mean pooling (i.e., which one is more expressive)(1-2 sentences)?

$$\rho_\phi\left(h_{u_1}^{k-1}, \cdots, h_{u_{|\mathcal{N}(v)|}}^{k-1}\right)_\pi =$$

(c) Learning BFS with GNN [7pts]

Next, we investigate the expressive power of GNN for learning simple graph algorithms. Consider breadth-first search (BFS), where at every step, nodes that are connected to already visited nodes become visited. Suppose that we use GNN to learn to execute the BFS algorithm. Suppose that the embeddings are 1-dimensional. Initially, all nodes have input feature 0, except a source node which has input feature 1. At every step, nodes reached by BFS have embedding 1, and nodes not reached by BFS have embedding 0. Describe a message function

$$M(h_v^k) =$$

an aggregation function

$$h_{N(v)}^{k+1} =$$

and an update rule

$$h_v^{k+1} =$$

for the GNN such that it learns the task perfectly.

What to submit

(a) Number of layers needed to distinguish the 2 nodes and a brief explanation.

(b) A mathematical expression for ρ_ϕ along with optional, brief explanations; 1-2 sentences describing the expressiveness of Janossy pooling.

(c) Mathematical expressions for message function $M(h_v^k)$, aggregation function $h_{N(v)}^{k+1}$ and update function h_v^{k+1}.