CSEP 590A: Machine Learning for Big Data **Spring 2022**

Problem Set 3

Please read the homework submission policies.

Assignment Submission All students should submit their assignments electronically via GradeScope. No handwritten work will be accepted. Math formulas **must** be typeset using LaTeX or other word processing software that supports mathematical symbols (E.g. Google Docs, Microsoft Word). Simply sign up on Gradescope and use the course code X3WYKY. Please use your UW NetID if possible.

For the non-coding component of the homework, you should upload a PDF rather than submitting as images. We will use Gradescope for the submission of code as well. Please make sure to tag each part correctly on Gradescope so it is easier for us to grade. There will be a small point deduction for each mistagged page and for each question that includes code. Put all the code for a single question into a single file and upload it. Only files in text format (e.g. .txt, .py, .java) will be accepted. **There will be no credit for coding questions without submitted code on Gradescope, or for submitting it after the deadline**, so please remember to submit your code.

Coding You may use any programming languages and standard libraries, such as NumPy and PySpark, but you may not use specialized packages and, in particular, machine learning libraries (e.g. sklearn, TensorFlow), unless stated otherwise. Ask on the discussion board whether specific libraries are allowed if you are unsure.

Late Day Policy All students will be given two no-questions-asked late periods, but only one late period can be used per homework and cannot be used for project deliverables. A late-period lasts 48 hours from the original deadline (so if an assignment is due on Thursday at 11:59 pm, the late period goes to the Saturday at 11:59pm Pacific Time).

Academic Integrity We take academic integrity extremely seriously. We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions and the code independently. In addition, each student should write down the set of people whom they interacted with.

Discussion Group (People with whom you discussed ideas used in your answers):

On-line or hardcopy documents used as part of your answers:

I acknowledge and accept the Academic Integrity clause.

*(Signed)*_____

0 HW3 Survey

Please complete HW3 Survey on Gradescope after finishing the homework.

1 Dead Ends in PageRank Computations (28 points)

We learned about PageRank in lecture which is an algorithm for ranking webpages. In this problem, we are going to focus on dead ends and examine how they affect the PageRank computations.

Suppose we denote the matrix of the Internet as the n-by-n matrix M, where n is the number of webpages. Suppose there are k links out of the node (webpage) j, and

$$M_{ij} = \begin{cases} 1/k & \text{if there is a link from } j \text{ to } i \\ 0 & \text{otherwise} \end{cases}$$

For a webpage j that is a *dead end* (i.e., one having zero links out), the column j is all zeroes.

Let $\mathbf{r} = [r_1, r_2, \ldots, r_n]^\top$ be *an estimate* of the PageRank vector. In one iteration of the PageRank algorithm, we compute the next estimate $\mathbf{r}'$ of the PageRank as: $\mathbf{r}' = M\mathbf{r}$.

Given any PageRank estimate vector $\mathbf{r}$, define $w(\mathbf{r}) = \sum_{i=1}^{n} r_i$.

(a) **[7pts]** Suppose the Web has no dead ends. Prove that $w(\mathbf{r}') = w(\mathbf{r})$.

(b) **[10pts]** Suppose there are still no dead ends, but we use a teleportation probability (i.e., the probability of jumping to some random page) of $1 - \beta$, where $0 < \beta < 1$. The expression for the next estimate of r_i becomes $r'_i = \beta \sum_{j=1}^{n} M_{ij} r_j + (1 - \beta)/n$. Under what conditions will $w(\mathbf{r}') = w(\mathbf{r})$? Prove your conclusion.

Hint: The conditions required for the equality are related to $w(\mathbf{r})$.

(c) **[11pts]** Now, let us assume a teleportation probability of $1 - \beta$ in addition to the fact that there are one or more dead ends. Call a node "dead" if it is a dead end and "live" if not. Assume $w(\mathbf{r}) = 1$. At each iteration, when not teleporting, each live node j distributes βr_j PageRank uniformly across each of the nodes it links to, and each dead node j distributes r_j/n PageRank to all the nodes.

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 $w(\mathbf{r}') = 1$.

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. You will be experimenting with a small randomly generated graph (assume that the graph has no dead-ends) provided in `graph-full.txt` in the folder `pagerank_hits/data`.

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 r either in memory or as an RDD. Only the matrix of links is too large to store in memory.

(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:

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

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

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

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

Run the aforementioned iterative process in Spark for 40 iterations (assuming $\beta = 0.8$) and (the estimate) of 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 following:

- 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.

(b) HITS Implementation [15 points]

Assume that 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 scaling factors λ and 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.

Assume that $\lambda = 1, \mu = 1$. Then 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, and 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:

- 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.

What to submit

(i) List 5 node ids with the highest and least PageRank scores [2(a)]

(ii) List 5 node ids with the highest and least hubbiness and authority scores [2(b)]

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

3 The Louvain Algorithm for Community Detection (42 points)

Note: For this question, assume all graphs are undirected and weighted.

The Louvain Algorithm is commonly used for community detection in graphs which can be especially a good fit for graphs with a hierarchical structure. In this problem, we are going to understand how the Louvain algorithm works by following the steps of the algorithm. Furthermore, we are going to focus on how the algorithm works with the hierarchically structured graphs and how we can evaluate the quality of the communities we discover.

Communities, or clusters of densely linked nodes in a graph, are important elements of a graph's structure. Thus, discovering communities from an observed graph can help us summarize the overall structure of a graph and learn more about the underlying process. However, finding the "best" set of communities from data is often a difficult problem. One way to measure how well a network is partitioned into communities is to calculate the number of within-community edges relative to the number of between-community edges. This can be formalized using *modularity*, defined as:

$$Q = \frac{1}{2m} \sum_{1 \leq i,j \leq n} \left(\left[A_{ij} - \frac{d_i d_j}{2m} \right] \delta\left(c_i, c_j\right) \right)$$

Where A is the adjacency matrix of a graph G with n vertices and m edges, A_{ij} is the (i, j)-th entry of A, $2m = \sum_{i,j} A_{ij}$ is the sum of all entries in A, d_i is the degree of node i, δ is the Kronecker delta, i.e. $\delta(k, l) = 1$ when $k = l$, otherwise - $\delta(k, l) = 0$, and c_i and c_j are the communities of i and j respectively. Here, we assume that communities are disjoint, i.e. each node can only belong to one community. The modularity of a graph lies in the range $[-1, 1]$.

Maximizing the modularity of a given graph is a computationally hard problem. The Louvain algorithm is a popular and efficient heuristic used to solve this problem. It is a greedy algorithm, meaning that at every step, it will take the path that provides the largest possible increase to the objective (modularity) at that step. Each pass of the algorithm has two phases:

- **Phase 1 (Modularity Optimization)** aims to group nodes in the graph G into communities in a way that maximizes the modularity of the graph. After the first pass, the input graph to this step is the graph produced by phase 2 in the previous pass (see below).
- **Phase 2 (Community Aggregation)** combines each community into a single node, producing a new graph H where each node represents a community of nodes in the graph G. This new graph H is fed into phase 1 in the next pass of the algorithm.

We repeat these two phases until we no longer increase modularity through one pass. The algorithm proceeds as follows:

Phase 1 (Modularity Optimization)

Input: a graph $G = (V, E)$ with a vertex set V and an edge set E. ▷ The input changes in each pass of the algorithm; after pass 1 the input is the graph output by phase 2.

Output: a partition of G into communities.

Initialize each node i as its own community
for each node $i \in V$ **do**
    $\mathcal{N}_i \leftarrow$ the set of neighbors of i in G
    for each node j in $\mathcal{N}_i$ **do**
        $\Delta Q_j \leftarrow$ the change in modularity when i is assigned to the community of j
    end for
    $j^* \leftarrow$ the neighbor that gives the most positive change in modularity ΔQ_{j^*}
    if $\Delta Q_{j^*} > 0$ **then**
        Assign node i to the community of j^*
    else
        Keep node i in its current community
    end if
end for

Phase 2 (Community Aggregation)

Input: a graph $G = (V_G, E_G)$ with a vertex set V_G and an edge set E_G; and the communities from Phase 1.

Output: a graph $H = (V_H, E_H)$ where each node now represents a community in the Phase 1 graph G.

$V_H \leftarrow \emptyset$
$E_H \leftarrow \emptyset$
for each community C **do**
    $V_H \leftarrow V_H \cup \{C\}$
end for
for each community C **do**
    for each community C' **do**
        $e \leftarrow \{C, C'\}$
        if $e \notin E_H$ **then**
            $E_H \leftarrow E_H \cup \{e\}$ (including a self-edge if $C = C'$)
            $w_e \leftarrow \sum_{\substack{v \in C \\ u \in C'}} w_{v,u}$. ▷ w_e is the weight assigned to the edge e in the graph H. Initially, $w_{v,u} = 1$ for all v and u.
        end if
    end for
end for

Again, we repeat phases 1 and 2 until no change in modularity occurs. We call each iteration of phases 1 and 2 one pass of the algorithm. Figure 1 below illustrates the Louvain algorithm.

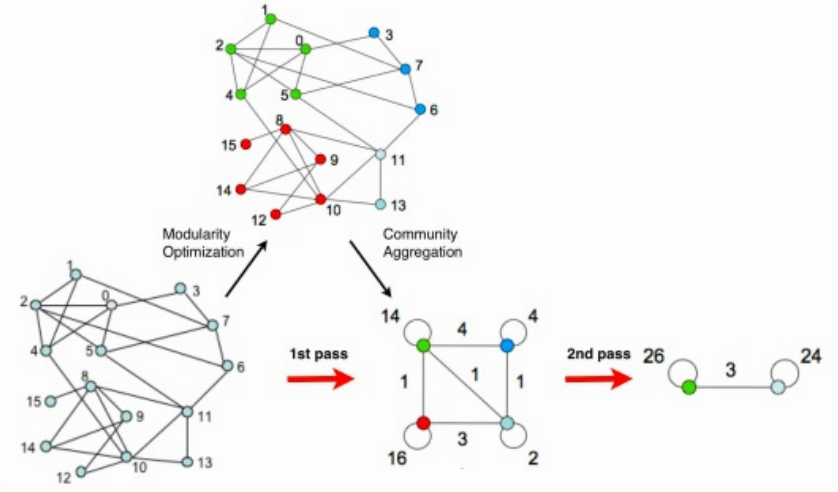

Figure 1: Example from Blondel et al. showing the two phases of the Louvain algorithm

(a) **[12 points]** Consider a node i that is in a community all by itself. Let C represent an existing community in the graph. Node i feels lonely and decides to move into the community C. This situation can be modeled by a graph (Figure 2) with C represented by a single node. We observe the following sums of weights:

- $\Sigma_{in} = \sum_{j,k \in C} w_{j,k}$ - the sum of the weights of edges within C.
- Σ_{tot} - the sum of the weights of the edges incident to a vertex in C.
- k_i - the sum of weights of edges incident to i (i.e., its weighted degree).
- $k_{i,in}/2$ - the sum of the weights of the edges between i and a vertex in C, i.e. the community of i and C are connected by an edge of weight $k_{i,in}/2$.
- As always, $2m = \sum_{i,j} A_{ij}$ is the sum of all entries in the adjacency matrix.

To begin with, C and i are in separate communities (colored green and red respectively). The third node represents the remainder of the graph.

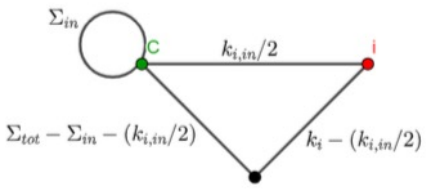

Figure 2: Before merging, i is an isolated node and C is a community. The rest of the graph is represented by a single node.

Prove that the modularity gain seen when i merges with C (i.e., the change in modularity after they merge into one community) is given by:

$$\Delta Q = \left[\frac{\Sigma_{in} + k_{i,in}}{2m} - \left(\frac{\Sigma_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right].$$

Note that this expression gives us a computationally efficient way to compute the modularity changes in phase 1.

Hint: Apply the community aggregation step of the Louvain algorithm to simplify the calculations.

(b) **[15 points]** Consider the graph G in Figure 3, with 4 cliques of 4 nodes each, arranged in a ring. Assume all the edges have the same weight value 1. There exists exactly one edge between any two adjacent cliques. We will manually (by hand) inspect the results of the Louvain algorithm on this network.

1. **[5 points]** The first phase of modularity optimization detects each clique as a single community, so there are 4 communities in total. Thus, the graph H output by the first pass of the Louvain algorithm is a graph with four nodes, each corresponding

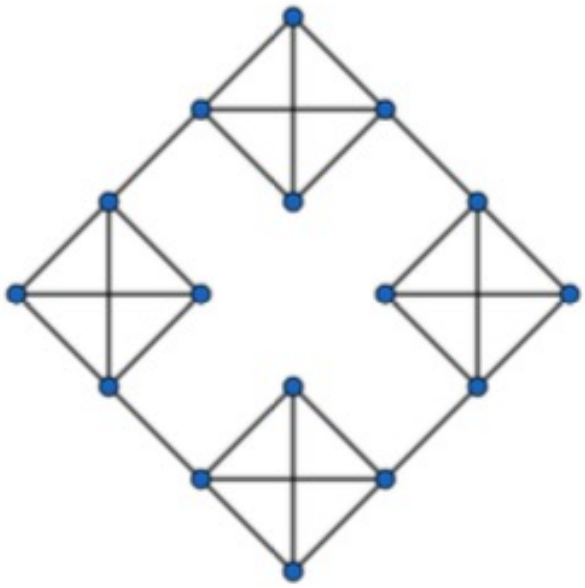

Figure 3: G is a subgraph. The whole graph has 16 nodes (4 cliques with 4 nodes per clique)

to one of the four cliques in G. What are the weights of each edge in the graph H? Explain.
Hint: Note that the symmetry of the ring structure simplifies the calculation.

2. [**4 points**] Derive the modularity of the graph H after the first pass of the Louvain algorithm.
3. [**6 points**] Show mathematically that the modularity of H would not increase in the second pass of the algorithm, hence the algorithm terminates.
Hint: Due to the symmetry in H, you only need to calculate a single value of ΔQ. You may either calculate the modularity directly or extend the result of part (a).

(c) [**15 points**] Modularity optimization often fails to identify communities smaller than a certain scale, which is known as the **resolution limit problem**. We illustrate this problem using a dataset with ground-truth communities; that is, we have labels for the "true" communities of the graph. We provide the following undirected YouTube social network. In the YouTube social network, users can form friendships with each other and users can create groups which other users can join. We consider such user-defined groups as ground-truth communities.

We are interested in quantifying how "good" the communities chosen by modularity by evaluating them via a goodness metric, which we present below. Here we compare 2 scoring functions: (1) *modularity* (higher is better) (2) *cut ratio* (lower is better): $f(S) = \frac{c_S}{n_S(n-n_S)}$, where c_s is the number of edges crossing the boundary of community S, n_S is the number of nodes in S and n is the number of nodes in the entire graph. Our goodness metric is *density* $g(S) = \frac{2m_S}{n_S(n_S-1)}$, where m_S is the number of edges within S.

Note that density favors small, highly connected communities; hence, we expect that scoring functions that do poorly with smaller communities will not perform well with respect to this metric, and that scoring functions that can accurately identify smaller communities will have strong performance with respect to this goodness metric. Thus, this creates a good test case for the resolution limit of *modularity*. From the definition, we could see cut ratio has already taken community size into account.

We run the following experiment: the file `youtube_community_top1000.txt` in the folder `louvain/data` contains the top 1000 ground-truth communities. For each community scoring function f, we rank the ground-truth communities by decreasing score of f. So, lower values of rank correspond to the "better" communities by each scoring function, whereas higher values of rank correspond to the "worse" communities under each scoring function. We measure the cumulative running average value of the goodness metric g of the top-k ground-truth communities under the ordering induced by f. Intuitively, a perfect community scoring function would rank the communities in decreasing order of the goodness metric, and thus the cumulative running average of the goodness metric would decrease monotonically with k.

You have been provided a skeleton in the file `PartitionQuality.py` in the folder `louvain/code`. Your task is to complete the functions `community_modularity`, `cut_ratio`, and `density`. Then, run the file, which executes the functions you have written and applies them to the YouTube dataset. Submit the plot `density.jpg` produced by the code as part of your write-up. Interpret the plot you produce. Which metric is performing better for this dataset? Using 2-3 sentences, explain why this might be the case.
Hint: For the first ground-truth community in `youtube_community_top1000.txt`, the modularity is approximately 2.15×10^{-5}, the cut ratio is approximately 1.39×10^{-4}, and the density is approximately 3.62×10^{-2}.

What to submit

(i) Proof of 3(a).

(ii) Answers to the 3 subparts of 3(b).

(iii) The plot `density.jpg` produced by your code and an interpretation thereof. [3(c)]

(iv) Upload your completed implementation of `PartitionQuality.py` to Gradescope.