CS224W: Analysis of Networks **Fall 2018**

Homework 2

Due 11:59pm PDT October 25, 2018

This problem set should be completed individually.

General Instructions

These questions require thought, but do not require long answers. Please be as concise as possible. You are allowed to take a maximum of 1 late period (see the information sheet at the end of this document for the definition of a late period).

Submission instructions: You should submit your answers via Gradescope and your code via the SNAP submission site. Register for Gradescope at `http://gradescope.com` using your Stanford e-mail (if not SCPD) and include your student ID number with sign-up. Use the entry code **9ZZ2XY** to sign up for CS224W.

Submitting answers: Prepare answers to your homework in a single PDF file and submit it via Gradescope. Make sure that the answer to each sub-question is on a *separate, single page*. The number of the question should be at the top of each page. Please use the submission template files included in the bundle to prepare your submission. Failure to use the submission template file will result in a reduction of 2 points from your homework score.

Information sheet: Fill out the information sheet located at the end of this problem set or at the end of the submission template file, and sign it in order to acknowledge the Honor Code (if typesetting the homework, you may type your name instead of signing). This should be the last page of your submission. Failure to fill out the information sheet will result in a reduction of 2 points from your homework score.

Submitting code: Upload your code at `http://snap.stanford.edu/submit`. Put all the code for a single question into a single file and upload it. Failure to submit your code will result in reduction of all points for that part from your homework score.

Homework survey: After submitting your homework, please fill out the Homework 2 Feedback Form. Respondents will be awarded extra credit.

Questions

1 Structural Roles: Rolx and ReFex [20 points – Shuyang Shi]

In this problem, we will explore the structural role extraction algorithm Rolx and its recursive feature extraction method (ReFex). The data we are using is a scientist co-authorship network, which can be downloaded at `http://www-personal.umich.edu/~mejn/netdata/netscience.zip`. [1] Although it is a weighted graph, for simplicity we treat it as **undirected and unweighted** in this problem.

[1]We provide a binary file named *hw2-q1.graph* for you to directly load into snap by `G = snap.TUNGraph.Load(snap.TFIn("hw2-q1.graph"))` . You are welcome to either use this file or load from raw data yourself.

Feature extraction is composed by two steps: it firstly extracts some basic local features from every node, and then recursively aggregates them along graph edges, so that global features are also obtained. Collectively, it constructs a matrix $V \in \mathbb{R}^{n \times f}$ where for each of the n nodes we have f features to cover local and global information. Rolx extracts node roles from that matrix.

1.1 Basic Features [4 points]

Load the graph provided in the bundle, and compute three basic features for the nodes. For each node v, we choose 3 basic local features (in this order):

1. the degree of v, i.e., $\deg(v)$;
2. the number of edges in the egonet of v, where egonet of v is defined as a subgraph whose nodes are v, and its neighbors and edges are induced from the whole graph;
3. the number of edges that connects the egonet of v and the rest of the graph, i.e., the number of edges that enters or leaves the egonet of v.

We use $\tilde{V_u}$ to represent the vector of the basic features of node u. For any pair of nodes u and v, we can use cosine similarity to measure how similar two nodes are according to their feature vectors x and y:

$$\text{Sim}(x, y) = \frac{x \cdot y}{||x||_2 \cdot ||y||_2} = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2} \cdot \sqrt{\sum_i y_i^2}};$$

Also, when $||x||_2 = 0$ or $||y||_2 = 0$, $\text{Sim}(x, y) = 0$.

For node with ID 9, what is the basic feature vector for it? Report the top 5 nodes that are most similar to node 9.

For sanity check, no element in $\tilde{V_9}$ is larger than 10.

1.2 Recursive Features [5 points]

In this next step, we recursively generate some more features. We use *mean* and *sum* as aggregation functions.

Initially, we have a feature vector $\tilde{V_u} \in \mathbb{R}^3$ for every node u. In the first iteration, we concatenate the mean of all u's neighbors' feature vectors to $\tilde{V_u}$, and do the same for *sum*, i.e.,

$$\tilde{V}_u^{(1)} = [\tilde{V_u}; \frac{1}{|N(u)|} \sum_{v \in N(u)} \tilde{V_v}; \sum_{v \in N(u)} \tilde{V_v}] \in \mathbb{R}^9,$$

where $N(u)$ is the set of u's neighbors in the graph. If $N(u) = \emptyset$, set the mean and sum to 0.

After K iterations, we obtain the overall feature matrix $V = \tilde{V}^{(K)} \in \mathbb{R}^{3^{K+1}}$.

For this exercise, run $K = 2$ iterations, and report the top 5 nodes that are most similar to node 9. [2 points] If there are ties, e.g. 4th, 5th, and 6th have the same similarity, report any of them to fill up the top-5 ranking. (For sanity check, the similarity between the reported nodes and node 9 are greater than 0.9)

Compare them with previous results: are there common nodes? Are there different nodes? Why would this change? (in one sentence) [3 points]

1.3 Role Discovery [11 points]

In this part, we explore more about the graph according to the feature vector and node similarity.

(a) Produce a 20-bin histogram to show the distribution of cosine similarity between node 9 and any other node in the graph. (where the X-axis is cosine similarity with node 9, and the Y-axis is the number of nodes) [2 points]

According to the histogram, can you spot some *groups* / *roles*? How many can you spot? (*Clue: look for the spikes!*) [2 points]

(b) For these groups / roles in the cosine similarity histogram, take one node u from each group to examine the feature vector, and draw the subgraph of the node based on its feature vector. (You can draw it by hand, or use libraries like networkx or graphviz.)

You should use the local features for u, and pay attention to the features aggregated from its 1-hop neighbors, but feel free to ignore further features if they are difficult to incorporate. Also, you should not draw nodes that are more than 3-hops away from u on it. [6 points]

Briefly argue how different structural roles are captured in 1-2 sentences. [1 point]

What to submit

Page 1:
- (1.1) The basic feature vector for node with ID 9, and the nodes most similar to node 9 in terms of cosine similarity.

Page 2:
- (1.2) Top 5 nodes that are closest to node 9 regarding cosine similarity
- Whether there are common and different nodes in the closest nodes reported for basic feature vectors and full feature vectors. And a one-sentence reason.

Page 3:
- (1.3) (a) A histograms for similarity distribution. Whether you can spot some groups, and if yes, how many.
- (b) Several hypothetical subgraphs, one for each groups in the cosine similarity histogram. Brief argument how different structural roles are captured.

2 Community detection using the Louvain algorithm [20 points – Jayadev Bhaskaran]

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

Communities are a fundamental aspect of several networks. However, it is often not an easy task to come up with an "optimal" grouping of nodes into communities. Through this problem, we will explore some properties of the Louvain algorithm for community detection (so named because the authors were all affiliated with the University of Louvain in Belgium at some point). The original paper from Blondel et al. is available here: `https://arxiv.org/pdf/0803.0476.pdf`. If you are stuck on this question at any point please refer to the paper; there is a good chance that you will find what you seek there.

We will first explore the idea of modularity. The modularity of a weighted graph is a measure that compares the density of edges within a community to the density of edges between communities. Formally, we define the modularity Q for a given graph as follows:

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

Here $2m = \Sigma A_{ij}$ is the sum of all entries in the adjacency matrix, A_{ij} represents the $(i,j)^{th}$ entry of the adjacency matrix, d_i represents the degree of node i, $\delta(c_i, c_j)$ is 1 when i and j are in the same community ($c_i = c_j$) and 0 otherwise. Note that we treat communities as disjoint. In other words, a given node from a graph can only belong to one community in that graph.

The modularity of a graph lies in the range $[-1, 1]$. Maximizing the modularity of a given graph is a computationally hard problem, so we try different heuristics for this purpose. One such heuristic is the **Louvain algorithm**. This algorithm outperforms many similar algorithms in terms of both speed as well as maximum modularity obtained. We will run a few steps of the algorithm on a couple of example networks to gain some insights about its behavior and properties.

Each pass of the algorithm has two phases, and proceeds as follows:

- **Phase 1 (Modularity Optimization):** Start with each node in its own community.
- For each node i, go over all the neighbors j of i. Calculate the change in modularity when i is moved from its present community to j's community. Find the neighbor j_m for which this process causes the greatest increase in modularity, and assign i to j_m's community (break ties arbitrarily). If there is no positive increase in modularity during this process, keep i in its original community.
- Repeat the above process for each node (going over nodes multiple times if required) until there is no further maximization possible (that is, each node remains in its community). This is the end of Phase 1.
- **Phase 2 (Community Aggregation):** Once Phase 1 is done, we contract the original graph G to a new graph H. Each community found in G after Phase 1 becomes a node in H. The weights of edges in between 2 nodes in H are given by the sums of the weights between the respective 2 communities in G. The weights of edges within a community in G sum up to form a self-edge of the same weight in H (be careful while calculating self-edge weights, note that you will have to go once over each original node within the community in G). This is the end of Phase 2. Phase 1 and Phase 2 together make up a single pass of the algorithm.
- Repeat Phase 1 and Phase 2 again on H and keep proceeding this way until no further improvement is seen (you will reach a point where each node in the graph stays in its original community to maximize modularity). The final modularity value is a heuristic for the maximum modularity of the graph.

An example taken from Blondel et al. (Figure 1) illustrates the two phases of the algorithm. Note how weights for self-edges are assigned in the Community Aggregation phase - we will need to use

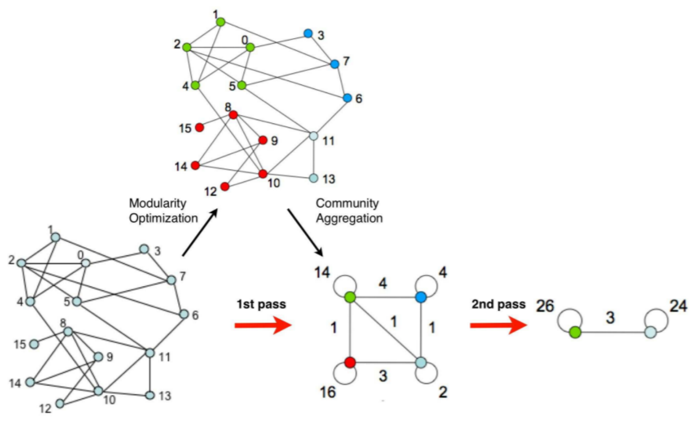

Figure 1: Example from Blondel et al. showing the two phases of the Louvain algorithm. Note how weights for self-edges are assigned in the Community Aggregation phase.

this later. The weight of the self-edge formed by merging all nodes in a community K would be given by $\Sigma_{i \in K} \Sigma_{j \in K} A_{ij}$ - you can verify this for yourself by checking in Figure 1 that node 11 and 13 have an edge of weight 1 between them, but the corresponding self-edge has a weight of 2.

2.1 Modularity gain when an isolated node moves into a community [3 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, we will inspect the change in modularity when this happens.

This situation can be modeled by a graph (Figure 2) with C being represented by a single node. C has a self-edge of weight Σ_{in}. There is an edge between i and C of weight $k_{i,in}/2$ (to stay consistent with the notation of the paper). The total degree of C is Σ_{tot} and the degree of i is k_i. As always, $2m = \Sigma A_{ij}$ 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). 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:

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

Hint: Using the community aggregation step of the Louvain method may make computation easier.

In practice, this result is used while running the Louvain algorithm (along with a similar related result) to make incremental modularity computations much faster.

2.2 Louvain algorithm on a 16 node network [8 points]

Consider the graph G (Figure 3), with 4 cliques of 4 nodes each arranged in a ring. There exists exactly one edge between any two cliques. We will manually inspect the results of the Louvain algorithm on this network. The first phase of modularity optimization detects each clique as a single community (giving 4 communities in all). After the community aggregation phase, the new network H will have 4 nodes.

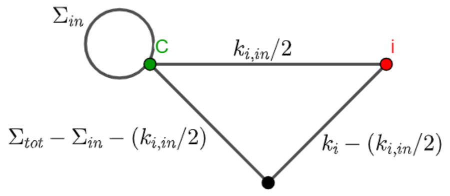

Figure 2: Before merging, i is an isolated node and C represents an existing community. The rest of the graph can be treated as a single node for this problem.

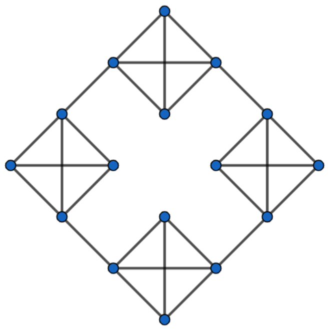

Figure 3: G is a graph with 16 nodes (4 cliques with 4 nodes per clique).

- What is the weight of any edge between two distinct nodes in H? [1 point]
- What is the weight of any self-edge in H? [1 point]
- What is the modularity of H (with each node in its own community)? The easiest way to calculate modularity would be to directly apply the definition from 1 to H (also holds for upcoming questions of this type). [2 points]

Spoiler alert: In this network, this is the maximum modularity and the algorithm will terminate here. However, assume that we wanted to contract the graph further into a two node network (call it J) by grouping two adjacent nodes in H into one community and the other two adjacent nodes into another community, and then aggregating (following the same rules of the community aggregation phase).

- What is the weight of any edge between two distinct nodes in J? [1 point]
- What is the weight of any self-edge in J? [1 point]
- What is the modularity of J (with each node in its own community)? [2 points]

As expected, the modularity of J is less than the modularity of H.

2.3 Louvain algorithm on a 128 node network [8 points]

Now consider a larger version of the same network, with 32 cliques of 4 nodes each (arranged in a ring as earlier); call this network G_{big}. Again, there exists exactly one edge between each clique. The first phase of modularity optimization, as expected, detects each clique as a single community. After aggregation, this forms a new network H_{big} with 32 nodes.

- What is the weight of any edge between two distinct nodes in H_{big}? [1 point]
- What is the weight of any self-edge in H_{big}? [1 point]
- What is the modularity of H_{big} (with each node in its own community)? [2 points]

After what we saw in the earlier example, we would expect the algorithm to terminate here. However (spoiler alert again), that doesn't happen and the algorithm proceeds. The next phase of modularity optimization groups H_{big} into 16 communities with two adjacent nodes from H_{big} in each community. Call the resultant graph (after community aggregation) J_{big}.

- What is the weight of any edge between two distinct nodes in J_{big}? [1 point]
- What is the weight of any self-edge in J_{big}? [1 point]
- What is the modularity of J_{big} (with each node in its own community)? [2 points]

This particular grouping of communities corresponds to the maximum modularity in this network (and not the one with one clique in each community). The community grouping that maximizes the modularity here corresponds to one that would not be considered intuitive based on the structure of the graph.

2.4 What just happened? [1 point]

Explain (in a few lines) why you think the algorithm behaved the way it did for the larger network (you don't need to prove anything rigorously, a rough argument will do). In other words, what might have caused modularity to be maximized with an "unintuitive" community grouping for the larger network?

What to submit:

Page 4:	• Proof for change in modularity when node i moves to community C.
Page 5:	• Weight of edge between two distinct nodes in H.
	• Weight of self-edge in H.
	• Modularity of H with each node in its own community.
	• Weight of edge between two distinct nodes in J.
	• Weight of self-edge in J.
	• Modularity of J with each node in its own community.
Page 6:	• The same as above, except for H_{big} and J_{big}.
Page 7:	• Explanation of algorithm behavior in the larger network (a few lines).

3 The Configuration Model and Motif Detection [35 points – Javier]

Note: For this question, assume all graphs are directed and unweighted.

For this question, you will explore the configuration model we covered in class. You will then implement the Exact Subgraph Enumeration algorithm presented and use it to count how often size 3 subgraphs appear in the graph. Lastly, you will compare your measurements on real graphs with those obtained using the configuration model as a null model to plot network significance profiles.

3.1 The Configuration Model [10 points]

A common method for analyzing the properties of real world networks is to analyze their behavior in comparison with a generated theoretical model referred to as a "null model." We have previously discussed some null models in class (Lecture 2), such as the Erdős-Rényi model and the configuration model. While the Erdős-Rényi model has many nice theoretical properties, the configuration model is useful because it generates random networks with a specified degree sequence. In other words, given a real network, the configuration model allows you to sample from the space of networks that have the exact same sequence of degrees (i.e., the sampled random networks have the same degree distribution as the network you are studying). For more background on the configuration model, check out these lecture notes by Aaron Clauset: http://tuvalu.santafe.edu/~aaronc/courses/5352/fall2013/csci5352_2013_L11.pdf

One way of using the configuration model to generate such a network is through the spoke matching algorithm covered in Lecture 5. The intuition behind this algorithm is that we first break the network apart into a bunch of "stubs", which are basically nodes with dangling edges (spokes); then we generate a random network by randomly pairing up these spokes and connecting them. Using this approach, however, has a couple of subtleties. For instance, this algorithm can sometimes create improper (i.e. non-simple) networks with self-loops or multiple edges between two nodes; if this happens, then you must reject this sampled network and try again. If you wish, you may try to implement the configuration model using this method, but instead, **we will use another approach**.

A second (and more popular) approach to sampling from the configuration model is to do "edge rewiring". The idea with this algorithm is to start with an empirical network and then randomly rewire edges until it is essentially random.

Implement "edge rewiring" for the US power grid graph. To do this, load the `USpowergrid_n4941.txt` dataset as a `PNGraph` and iteratively repeat the following process:

1. Randomly select two distinct edges $e_1 = (a, b)$ and $e_2 = (c, d)$ from the graph. Now, try to re-wire these edges.
2. Randomly select one of endpoint of edge e_1 and call it u. Let v be the other endpoint in e_1. At this point, either $u = a$, $v = b$ or $u = b$, $v = a$. Do the same for edge e_2. Call the randomly selected endpoint w and the other endpoint x.
3. Perform the rewiring. In the graph, replace the directed edges $e_1 = (a, b)$ and $e_2 = (c, d)$ with

the directed edges (u, w) and (v, x) *as long as this results in a simple network (no self-loops or multi-edges)*. If the result is not a simple network, reject this rewiring and return to step 1; otherwise, keep the newly swapped edges and return to step 1.

Test your implementation and verify that you did not alter the number of nodes or edges on the original network. Run your edge rewiring implementation for 10,000 iterations on the power grid network. Every 100 iterations, calculate the average clustering coefficient of the rewired network. Then plot the average clustering coefficient as a function of the number of iterations. Briefly comment on the curve that represents the model being rewired by explaining its shape.

3.2 Exact Subgraph Enumeration (ESU) [15 points]

As we saw in class (Lecture 5), we have covered several metrics at node level (such as degree, PageRank score and node clustering) as well as at graph level (diameter, size of components, graph clustering coefficient). It would be useful to have something that characterized the graph at the mesoscopic level, that is, somewhere in between looking at the whole graph and looking at a single node. Looking at subgraphs of the network seems intuitive and will prove useful. Motifs are defined as recurrent, significant patterns of interconnections. They can help us understand how a network works and will help us predict the reaction of a network in a given situation.

We can think of motifs as structurally identical subgraphs that occur frequently within a graph. It may be the case that a motif of interest appears several time in the graph, with overlapping nodes. Even though instances of the motif may overlap, we count each appearance separately.

We wish to be able to count occurrences of certain subgraphs and compare with the number of occurrences in a null model, so that we identify the patterns that occur more frequently in a given graph and that as such, may be identified as motifs. But, as we saw in lecture, finding size-k motifs in graphs is a complex computational challenge, since it involves enumerating all possible size-k connected subgraphs and counting the number of occurrences of each subgraph type. This is an NP-hard problem, since graph isomorphism is NP-complete! But dont worry! We'll tackle a subproblem.

Implement the ESU algorithm we covered in Lecture 5 to find all possible 3-subgraphs. We have provided starter code which takes care of the graph isomorphism matching and counting problem. Although, in general, this problem is NP-hard, for $k = 3$ we can exhaustively check all $3! = 6$ possible permutations on node labelings to find to which of the 13 possible 3-subgraphs (shown in Figure 4) has been identified. All you need to do is implement ESU as a recursive algorithm that upon finding a 3-subgraph, counts it using our provided isomorphism counting implementation.

Test your implementation against the provided `esu_test.txt` dataset. Load the file as a `PNGraph`. In Figure 5 you can see the graph we are loading. Then, run with the `verbose` option set to `True` and report the output of the run.

3.3 Detecting motifs on real data [10 points]

Now that we are equiped with a null model and a motif counting algorithm, we will proceed to detect motifs on real world data.

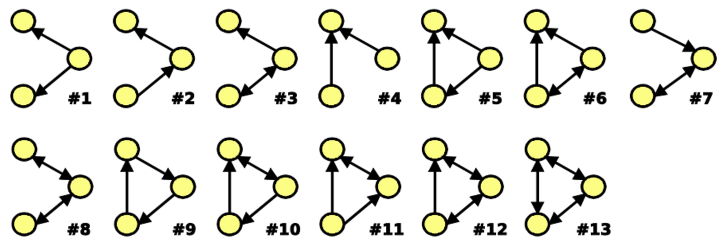

Figure 4: All possible non-isomorphic directed 3-subgraphs

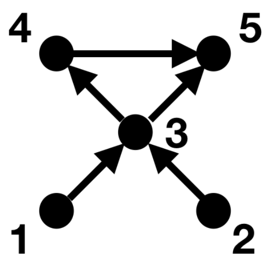

Figure 5: The provided graph to test ESU against. Note that this graph has 6 connected 3-subgraphs

Load the provided `USpowergrid_n4941.txt` dataset as a `PNGraph`. Compute the motif frequencies using your ESU implementation from the previous exercise. Now, sample 10 instances from the configuration model using this graph a starting point. Note that depending on your implementation you may need to reload the graph to generate each configuration model instance. Compute the motif frequencies on each of the 10 samples random graphs and store the results. You may use a `numpy` array of size 10×13 to store your motif counts. Compute the per-motif mean and standard deviation for the motif counts on these 10 runs and use it to compute the Z scores. Recall:

$$Z_i = \frac{N^{(i)} - \bar{N}^{(i)}_{\text{sampled}}}{\text{std}(N^{(i)}_{\text{sampled}})}$$

where $N^{(i)}$ is the number of times motif i appeared in the original graph, $\bar{N}^{(i)}_{\text{sampled}}$ is the mean number of times motif i appeared in the 10 sampled graphs and $\text{std}(N^{(i)}_{\text{sampled}})$ is the standard deviation of the sampled motif frequencies for motif i.

Plot the Z scores against their respective motif index and comment (1-2 sentences) on the results.

Repeat the above process for the `email-Eu-core.txt` dataset also loaded as a `PNGraph`. Report the obtained Z score plot along with comments as before.

Our reference implementation of the solution runs in about 10 minutes for the `USpowergrid` dataset and about 2.5 hours for the `email-Eu` dataset.

What to submit

Page 8: • A plot of the average clustering coefficient as a function of the iteration number for the rewiring algorithm. A brief comment on this plot (1–2 sentences).

Page 9: • The output of a verbose run of the ESU algorithm against on the `esu_test.txt` network.

Page 10: • Two plots of Z scores along with their respective comments (1–2 sentences per plot).

4 Spectral clustering [25 points – Alex Haigh]

This question derives a spectral clustering algorithm that we then use to analyze a real-world dataset. These algorithms use eigenvectors of matrices associated with the graph. We will post a handout on graph clustering on Piazza and the course website that will be helpful for solving this question.

Let's first discuss some notation:

- Let $G = (V, E)$ be a simple (that is, self- or multi- edges) undirected, connected graph with $n = |V|$ and $m = |E|$.
- A is the adjacency matrix of the graph G, i.e., A_{ij} is equal to 1 if $(i, j) \in E$ and equal to 0 otherwise.
- D is the diagonal matrix of degrees: $D_{ii} = \sum_j A_{ij}$ = the degree of node i.
- We define the *graph Laplacian* of G by $L = D - A$.

For a set of nodes $S \subset V$, we will measure the quality of S as a cluster with a "cut" value and a "volume" value. We define the cut of the set S to be the number of edges that have one end point in S and one end point in the complement set $\bar{S} = V \backslash S$:

$$\text{cut}(S) = \sum_{i \in S, j \in \bar{S}} A_{ij}.$$

Note that the cut is symmetric in the sense that $\text{cut}(S) = \text{cut}(\bar{S})$. The *volume* of S is simply the sum of degrees of nodes in S:

$$\text{vol}(S) = \sum_{i \in S} d_i,$$

where d_i is the degree of node i.

4.1 A Spectral Algorithm for Normalized Cut Minimization: Foundations [10 points]

We will try to find a set S with a small normalized cut value:

$$\text{NCUT}(S) = \frac{\text{cut}(S)}{\text{vol}(S)} + \frac{\text{cut}(\bar{S})}{\text{vol}(\bar{S})} \tag{3}$$

Intuitively, a set S with a small normalized cut value must have few edges connecting to the rest of the graph (making the numerators small) as well as some balance in the size of the clusters (making the denominators large).

Define the assignment vector x for some set of nodes S such that

$$x_i = \begin{cases} \sqrt{\frac{\text{vol}(\bar{S})}{\text{vol}(S)}} & i \in S \\ -\sqrt{\frac{\text{vol}(S)}{\text{vol}(\bar{S})}} & i \in \bar{S} \end{cases} \tag{4}$$

Prove the following properties:

(i) $L = \sum_{(i,j)\in E}(e_i - e_j)(e_i - e_j)^T$, where e_k is an n-dimensional column vector with a 1 at position k and 0's elsewhere. Note that we aren't summing over the entire adjacency matrix and only count each edge once.

(ii) $x^T Lx = \sum_{(i,j)\in E}(x_i - x_j)^2$

(iii) $x^T Lx = c \cdot \text{NCUT}(S)$ for some constant c (in terms of the problem parameters).

(iv) $x^T De = 0$, where e is the vector of all ones.

(v) $x^T Dx = 2m$.

4.2 Normalized Cut Minimization: Solving for the Minimizer [8 points]

Since $x^T Dx$ is just a constant ($2m$), we can formulate the normalized cut minimization problem in the following way:

$$\begin{aligned} \underset{S\subset V,\, x\in\mathbb{R}^n}{\text{minimize}} \quad & \frac{x^T Lx}{x^T Dx} \\ \text{subject to} \quad & x^T De = 0,\ x^T Dx = 2m,\ x \text{ as in Equation 4} \end{aligned} \tag{5}$$

The constraint that x takes the form of Equation 4 makes the optimization problem NP-hard. We will instead use the "relax and round" technique where we relax the problem to make the optimization problem tractable and then round the relaxed solution back to a feasible point for the original problem. Our relaxed problem will eliminate the constraint that x take the form of Equation 4 which leads to the following relaxed problem:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & \frac{x^T Lx}{x^T Dx} \\ \text{subject to} \quad & x^T De = 0,\ x^T Dx = 2m \end{aligned} \tag{6}$$

Show that the minimizer of Equation 6 is $D^{-1/2}$v, where v is the eigenvector corresponding to the second smallest eigenvalue of the *normalized graph Laplacian* $\tilde{L} = D^{-1/2}LD^{-1/2}$. Finally, to round the solution back to a feasible point in the original problem, we can take the indices of all positive entries of the eigenvector to be the set S and the indices of all negative entries to be $\bar{S}$.

Hint 1: Make the substitution $z = D^{1/2}x$.

Hint 2: Note that e *is the eigenvector corresponding to the smallest eigenvalue of* L.

Hint 3: The normalized graph Laplacian $\tilde{L}$ *is symmetric, so its eigenvectors are orthonormal and form a basis for* $\mathbb{R}^n$. *This means we can write any vector* x *as a linear combination of orthonormal eigenvectors of* $\tilde{L}$.

4.3 Relating Modularity to Cuts and Volumes [7 points]

In Problem 2, we presented the modularity of a graph clustering in the context of the Louvain Algorithm. Modularity actually relates to cuts and volumes as well. Let's consider a partitioning of our graph A into 2 clusters, and let $y \in \{1, -1\}^n$ be an assignment vector for a set S:

$$y_i = \begin{cases} 1 & \text{if } i \in S \\ -1 & \text{if } i \in \bar{S} \end{cases} \tag{7}$$

Then, the *modularity* of the assignment y is

$$Q(y) = \frac{1}{2m} \sum_{1 \le i,j \le n} \left[A_{ij} - \frac{d_i d_j}{2m} \right] I_{y_i = y_j}. \tag{8}$$

Let y be the assignment vector in Equation 7. Prove that

$$Q(y) = \frac{1}{2m} \left(-2 \cdot \text{cut}(S) + \frac{1}{m} \text{vol}(S) \cdot \text{vol}(\bar{S}) \right) \tag{9}$$

Thus, maximizing modularity is really just minimizing the sum of the cut and the negative product of the partition's volumes. As a result, we can use spectral algorithms similar to the one derived in parts 1-2 in order to find a clustering that maximizes modularity. While this might provide an intuitively "better" clustering after inspection than the Louvain Algorithm, spectral algorithms are computationally intensive on large graphs, and would only partition the graph into 2 clusters.

Note: You only need to prove the relationship between modularity and cuts; you do *not* need to derive the actual spectral algorithm.

What to submit

Page 11:	• Proofs of the five properties.
Page 12:	• Proof of the minimizer.
Page 13:	• Proof of the relationship between modularity and cuts and volumes.

Information sheet
CS224W: Analysis of Networks

Assignment Submission Fill in and include this information sheet with each of your assignments. This page should be the last page of your submission. Assignments are due at 11:59pm and are always due on a Thursday. All students (SCPD and non-SCPD) must submit their homeworks via Gradescope (`http://www.gradescope.com`). Students can typeset or scan their homeworks. Make sure that you answer each (sub-)question on a separate page. That is, one answer per page regardless of the answer length. Students also need to upload their code at `http://snap.stanford.edu/submit`. Put all the code for a single question into a single file and upload it. Please do not put any code in your Gradescope submissions.

Late Homework Policy Each student will have a total of *two* free late periods. *Homework are due on Thursdays at 11:59pm PDT and one late period expires on the following Monday at 11:59pm PDT*. Only one late period may be used for an assignment. Any homework received after 11:59pm PDT on the Monday following the homework due date will receive no credit. Once these late periods are exhausted, any assignments turned in late will receive no credit.

Honor Code We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down their solutions independently i.e., each student must understand the solution well enough in order to reconstruct it by him/herself. Students should clearly mention the names of all the other students who were part of their discussion group. Using code or solutions obtained from the web (github/google/previous year solutions, etc.) is considered an honor code violation. We check all the submissions for plagiarism. We take the honor code very seriously and expect students to do the same.

Your name: ______________________________

Email: ____________________ **SUID:** ______________

Discussion Group: ______________________________

I acknowledge and accept the Honor Code.

(Signed) ______________________________