CS246: Mining Massive Data Sets **Winter 2024**

Homework 4

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

1 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.

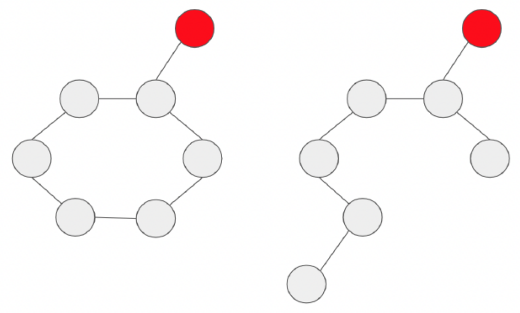
These are the key steps in each layer of a GNN:

- 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

$$M\left(h_v^k\right) =$$

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 $M\left(h_v^k\right)$, aggregation function $h_{N(v)}^{k+1}$ and update function h_v^{k+1}.

2 Decision Tree Learning (20 points)

In this problem, we want to construct a decision tree to find out if a person will enjoy beer.

Definitions. Let there be k binary-valued attributes in the data.

We pick an attribute that maximizes the gain at each node:

$$G = I(D) - (I(D_L) + I(D_R)); \tag{1}$$

where D is the given dataset, and D_L and D_R are the sets on left and right hand-side branches after division. Ties may be broken arbitrarily.

There are three commonly used impurity measures used in binary decision trees: Entropy, Gini index, and Classification Error. In this problem, we use Gini index and define $I(D)$ as follows[1]:

$$I(D) = |D| \times \left(1 - \sum_i p_i^2\right),$$

where:

- $|D|$ is the number of items in D;
- $1 - \sum_i p_i^2$ is the gini index;
- p_i is the probability distribution of the items in D, or in other words, p_i is the fraction of items that take value $i \in \{+, -\}$. Put differently, p_+ is the fraction of positive items and p_- is the fraction of negative items in D.

Note that this intuitively has the feel that the more evenly-distributed the numbers are, the smaller the $\sum_i p_i^2$, and the larger the impurity.

(a) [10 Points]

Let $k = 3$. We have three binary attributes that we could use: "likes wine", "likes running" and "likes pizza". Suppose the following:

- There are 100 people in sample set, 40 of whom like beer and 60 who don't.

[1] As an example, if D has 10 items, with 4 positive items (*i.e.* 4 people who enjoy beer), and 6 negative items (*i.e.* 6 who do not), we have $I(D) = 10 \times (1 - (0.16 + 0.36))$.

- Out of the 100 people, 50 like wine; out of those 50 people who like wine, 20 like beer.
- Out of the 100 people, 30 like running; out of those 30 people who like running, 20 like beer.
- Out of the 100 people, 80 like pizza; out of those 80 people who like pizza, 30 like beer.

Task: What are the values of G (defined in Equation 1) for wine, running and pizza attributes? Which attribute would you use to split the data at the root if you were to maximize the gain G using the gini index metric defined above?

(b) [10 Points]

Let's consider the following example:

- There are 100 attributes with binary values $a_1, a_2, a_3, \ldots, a_{100}$.
- Let there be one example corresponding to each possible assignment of 0's and 1's to the values $a_1, a_2, a_3 \ldots, a_{100}$. (Note that this gives us 2^{100} training examples.)
- Let the values taken by the target variable y depend on the values of a_1 for 99% of the datapoints. More specifically, of all the datapoints where $a_1 = 1$, let 99% of them are labeled $+$. Similarly, of all the datapoints where $a_1 = 0$, let 99% of them are labeled with $-$. (Assume that the values taken by y depend on $a_2, a_3, \ldots, a_{100}$ for fewer than 99% of the datapoints.)
- Assume that we build a complete binary decision tree (*i.e.*, we use values of all attributes).

Task: Explain **what this decision tree will look like.** Be sure to mention the height of the tree and any significant splitting patterns. (A 2-3 line explanation will suffice.) Also, in 2-3 sentences, identify **what the desired decision tree for this situation should look like to avoid overfitting, and why.** (The desired decision tree isn't necessarily a complete binary decision tree)

What to submit

(i) Values of G for wine, running and pizza attributes. [part (a)]

(ii) The attribute you would use for splitting the data at the root. [part (a)]

(iii) Explain what the decision tree looks like in the described setting. Explain how a decision tree should look like to avoid overfitting. (1-2 lines each) [part (b)]

3 Clustering Data Streams (20 points)

Introduction. In this problem, we study an approach for clustering massive data streams. We will study a framework for turning an approximate clustering algorithm into one that can work on data streams, *i.e.*, one which needs a small amount of memory and a small number of (actually, just one) passes over the data. As the instance of the clustering problem, we will focus on the k-means problem.

Definitions. Before going into further details, we need some definitions:

- The function $d : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}^+$ denotes the Euclidean distance:

$$d(x, y) = ||x - y||_2.$$

- For any $x \in \mathbb{R}^p$ and $T \subset \mathbb{R}^p$, we define:

$$d(x, T) = \min_{z \in T}\{d(x, z)\}.$$

- Having subsets $S, T \subset \mathbb{R}^p$, and a weight function $w : S \to \mathbb{R}^+$, we define:

$$\text{cost}_w(S, T) = \sum_{x \in S} w(x) d(x, T)^2.$$

- Finally, if for all $x \in S$ we have $w(x) = 1$, we simply denote $\text{cost}_w(S, T)$ by $\text{cost}(S, T)$.

Reminder: k-means clustering. The k-means clustering problem is as follows: given a subset $S \subset \mathbb{R}^p$, and an integer k, find the set T (with $|T| = k$), which minimizes $\text{cost}(S, T)$. If a weight function w is also given, the k-means objective would be to minimize $\text{cost}_w(S, T)$, and we call the problem the weighted k-means problem.

Strategy for clustering data streams. We assume we have an algorithm ALG which is an α-approximate weighted k-means clustering algorithm (for some $\alpha > 1$). In other words, given any $S \subset \mathbb{R}^p$, $k \in \mathbb{N}$, and a weight function w, ALG returns a set $T \subset \mathbb{R}^p$, $|T| = k$, such that:

$$\text{cost}_w(S, T) \leq \alpha \min_{|T'|=k}\{\text{cost}_w(S, T')\}.$$

We will see how we can use ALG as a building block to make an algorithm for the k-means problem on data streams.

The basic idea here is that of divide and conquer: if S is a huge set that does not fit into main memory, we can read a portion of it that does fit into memory, solve the problem on this subset (*i.e.*, do a clustering on this subset), record the result (*i.e.*, the cluster centers and some corresponding weights, as we will see), and then read a next portion of S which is again small enough to fit into memory, solve the problem on this part, record the result, etc. At the end, we will have to combine the results of the partial problems to construct a solution for the main big problem (*i.e.*, clustering S).

To formalize this idea, we consider the following algorithm, which we denote as ALGSTR:

- Partition S into ℓ parts $S_1, \ldots, S_\ell$.
- For each $i = 1$ to ℓ, run ALG on S_i to get a set of k centers $T_i = \{t_{i1}, t_{i2}, \ldots, t_{ik}\}$, and assume $\{S_{i1}, S_{i2}, \ldots, S_{ik}\}$ is the corresponding clustering of S_i (*i.e.*, $S_{ij} = \{x \in S_i \,|\, d(x, t_{ij}) < d(x, t_{ij'}) \; \forall j' \neq j, 1 \leq j' \leq k\}$).
- Let $\widehat{S} = \bigcup_{i=1}^{\ell} T_i$, and define weights $w(t_{ij}) = |S_{ij}|$.
- Run ALG on $\widehat{S}$ with weights w, to get k centers T.
- Return T.

Now, we analyze this algorithm. Assuming $T^* = \{t_1^*, \ldots, t_k^*\}$ to be the optimal k-means solution for S (that is, $T^* = \text{argmin}_{|T'|=k}\{\text{cost}(S, T')\}$), we would like to compare $\text{cost}(S, T)$ (where T is returned by ALGSTR) with $\text{cost}(S, T^*)$.

A small fact might be useful in the analysis below: for any $(a, b) \in \mathbb{R}^+$ we have:

$$(a + b)^2 \leq 2a^2 + 2b^2.$$

(a) [5pts]

First, we show that the cost of the final clustering can be bounded in terms of the total cost of the intermediate clusterings:

Task: Prove that:

$$\text{cost}(S, T) \leq 2 \cdot \text{cost}_w(\widehat{S}, T) + 2 \sum_{i=1}^{\ell} \text{cost}(S_i, T_i).$$

Hint: You might want to use Triangle Inequality for Euclidean distance d.

(b) [5pts]

So, to bound the cost of the final clustering, we can bound the terms on the right hand side of the inequality in part (a). Intuitively speaking, we expect the second term to be small compared to $\text{cost}(S, T^*)$, because T^* only uses k centers to represent the data set (S), while the T_i's, in total, use $k\ell$ centers to represent the same data set (and $k\ell$ is potentially much bigger than k). We show this formally:

Task: Prove that:

$$\sum_{i=1}^{\ell} \mathrm{cost}(S_i, T_i) \le \alpha \cdot \mathrm{cost}(S, T^*).$$

(c) [10pt]

Prove that ALGSTR is a $(4\alpha^2 + 6\alpha)$-approximation algorithm for the k-means problem.

Task: Prove that:

$$\mathrm{cost}(S, T) \le (4\alpha^2 + 6\alpha) \cdot \mathrm{cost}(S, T^*).$$

Hint: You might want to first prove two useful facts, which help bound the first term on the right hand side of the inequality in part (a):

$$\mathrm{cost}_w(\widehat{S}, T) \le \alpha \cdot \mathrm{cost}_w(\widehat{S}, T^*).$$

$$\mathrm{cost}_w(\widehat{S}, T^*) \le 2 \sum_{i=1}^{\ell} \mathrm{cost}(S_i, T_i) + 2 \cdot \mathrm{cost}(S, T^*).$$

Additional notes: We have shown above that ALGSTR is a $(4\alpha^2 + 6\alpha)$-approximation algorithm for the k-means problem. Clearly, $4\alpha^2 + 6\alpha > \alpha$, so ALGSTR has a somewhat worse approximation guarantee than ALG (with which we started). However, ALGSTR is better suited for the streaming application, as not only it takes just one pass over the data, but also it needs a much smaller amount of memory.

Assuming that ALG needs $\Theta(n)$ memory to work on an input set S of size n (note that just representing S in memory will need $\Omega(n)$ space), if we partitioning S into $\sqrt{n/k}$ equal parts, ALGSTR can work with only $O(\sqrt{nk})$ memory. (Like in the rest of the problem, k represents the number of clusters per partition.)

Note that for typical values of n and k, assuming $k \ll n$, we have $\sqrt{nk} \ll n$. For instance, with $n = 10^6$, and $k = 100$, we have $\sqrt{nk} = 10^4$, which is 100 times smaller than n.

What to submit

(a) Proof that $\mathrm{cost}(S, T) \le 2 \cdot \mathrm{cost}_w(\widehat{S}, T) + 2 \sum_{i=1}^{\ell} \mathrm{cost}(S_i, T_i)$.

(b) Proof that $\sum_{i=1}^{\ell} \mathrm{cost}(S_i, T_i) \le \alpha \cdot \mathrm{cost}(S, T^*)$.

(c) Proof that $\mathrm{cost}(S, T) \le (4\alpha^2 + 6\alpha) \cdot \mathrm{cost}(S, T^*)$.

4 Data Streams (30 points)

In this problem, we study an approach to approximating the frequency of occurrences of different items in a data stream. Assume $S = \langle a_1, a_2, \dots, a_t \rangle$ is a data stream of items from the set $\{1, 2, \dots, n\}$. Assume for any $1 \le i \le n$, $F[i]$ is the number of times i has appeared in S. We would like to have good approximations of the values $F[i]$ ($1 \le i \le n$) at all times.

A simple way to do this is to just keep the counts for each item $1 \le i \le n$ separately. However, this will require $\mathcal{O}(n)$ space, and in many applications (e.g., think online advertising and counts of user's clicks on ads) this can be prohibitively large. We see in this problem that it is possible to approximate these counts using a much smaller amount of space. To do so, we consider the algorithm explained below.

Strategy. The algorithm has two parameters $\delta, \epsilon > 0$. It picks $\lceil \log \frac{1}{\delta} \rceil$ independent hash functions:

$$\forall j \in \left[\!\left[1; \left\lceil \log \frac{1}{\delta} \right\rceil \right]\!\right], \quad h_j : \{1, 2, \dots, n\} \to \{1, 2, \dots, \left\lceil \frac{e}{\epsilon} \right\rceil\},$$

where log denotes natural logarithm. Also, it associates a count $c_{j,x}$ to any $1 \le j \le \lceil \log \frac{1}{\delta} \rceil$ and $1 \le x \le \lceil \frac{e}{\epsilon} \rceil$. In the beginning of the stream, all these counts are initialized to 0. Then, upon arrival of each a_k ($1 \le k \le t$), each of the counts $c_{j,h_j(a_k)}$ ($1 \le j \le \lceil \log \frac{1}{\delta} \rceil$) is incremented by 1.

For any $1 \le i \le n$, we define $\tilde{F}[i] = \min_j \{c_{j,h_j(i)}\}$. We will show that $\tilde{F}[i]$ provides a good approximation to $F[i]$.

Memory cost. Note that this algorithm only uses $\mathcal{O}\left(\frac{1}{\epsilon} \log \frac{1}{\delta}\right)$ space.

Properties. A few important properties of the algorithm presented above:

- For any $1 \le i \le n$:
$$\tilde{F}[i] \ge F[i].$$
- For any $1 \le i \le n$ and $1 \le j \le \lceil \log(\frac{1}{\delta}) \rceil$:
$$\mathbb{E}\left[c_{j,h_j(i)}\right] \le F[i] + \frac{\epsilon}{e}(t - F[i]).$$

(a) [10 Points]

Prove that:

$$\Pr\left[\tilde{F}[i] \le F[i] + \epsilon t\right] \ge 1 - \delta.$$

Hint: Use Markov inequality and the property of independence of hash functions.

Based on the proof in part (a) and the properties presented earlier, it can be inferred that $\tilde{F}[i]$ is a good approximation of $F[i]$ for any item i such that $F[i]$ is not very small (compared to t). In many applications (*e.g.*, when the values $F[i]$ have a heavy-tail distribution), we are indeed only interested in approximating the frequencies for items which are not too infrequent. We next consider one such application.

(b) [20 Points]

Warning. This implementation question requires substantial computation time Python implementation reported to take 15min - 1 hour. Therefore, we advise you to start early.

Dataset. The dataset in **q4/data** contains the following files:

1. `words_stream.txt` Each line of this file is a number, corresponding to the ID of a word in the stream. As a sanity check, there should be around 196 million lines.
2. `counts.txt` Each line is a pair of numbers separated by a tab. The first number is an ID of a word and the second number is its associated exact frequency count in the stream.
3. `words_stream_tiny.txt` and `counts_tiny.txt` are smaller versions of the dataset above that you can use for debugging your implementation.
4. `hash_params.txt` Each line is a pair of numbers separated by a tab, corresponding to parameters a and b which you may use to define your own hash functions (See explanation below).

Instructions. Implement the algorithm and run it on the dataset with parameters $\delta = e^{-5}, \epsilon = e \times 10^{-4}$. (Note: with this choice of δ you will be using 5 hash functions - the 5 pairs (a, b) that you'll need for the hash functions are in `hash_params.txt`). Then for each distinct word i in the dataset, compute the relative error $E_r[i] = \frac{\tilde{F}[i] - F[i]}{F[i]}$ and plot these values as a function of the exact word frequency $\frac{F[i]}{t}$. (**You do not have to implement the algorithm in Spark.**)

The plot should use a logarithm scale both for the x and the y axes, and there should be ticks to allow reading the powers of 10 (e.g. 10^{-1}, 10^0, 10^1 etc...). The plot should have a title, as well as the x and y axes. The exact frequencies $F[i]$ should be read from the counts file. Note that words of low frequency can have a very large relative error. That is not a bug in your implementation, but just a consequence of the bound we proved in question (a).

Answer the following question by reading values from your plot: What is an approximate condition on a word frequency in the document to have a relative error below $1 = 10^0$?

Hash functions. You may use the following hash function (see example pseudo-code), with $p = 123457$, a and b values provided in the hash params file and `n_buckets` (which is equivalent to $\lceil \frac{e}{\epsilon} \rceil$) chosen according to the specification of the algorithm. In the provided file, each line gives you a, b values to create one hash function.

```
# Returns hash(x) for hash function given by parameters a, b, p and n_buckets
def hash_fun(a, b, p, n_buckets, x):
    y = x [modulo] p
    hash_val = (a*y + b) [modulo] p
    return hash_val [modulo] n_buckets
```

Note: This hash function implementation produces outputs of value from 0 to (`n_buckets` − 1), which is different from our specification in the **Strategy** part. You can either keep the range as $\{0, ..., \texttt{n\_buckets} - 1\}$, or add 1 to the hash result so the value range becomes $\{1, ..., \texttt{n\_buckets}\}$, as long as you stay consistent within your implementation.

What to submit

(i) Proof that $\Pr[\tilde{F}[i] \leq F[i] + \epsilon t] \geq 1 - \delta$. [part (a)]

(ii) Log-log plot of the relative error as a function of the frequency. Answer for which word frequencies is the relative error below 1. (Note that `words_stream.txt` is a large file, and it's recommended that the codes be run on a local PC instead of Google Colab) [part (b)]

(iii) Submit the code on Gradescope. [part (b)]