Importance of Inductive Algorithms

Steven Liu, 2020

Introduction

The graph structure is a powerful data structure that is able to capture the relationship between real world objects. For example, in sports, we can see that a node would be a team, and teams would have features like players. An edge can be the relationship between two teams, and the edge can contain features that represent the results in recorded matches. The relationship is extremely important and useful because it should be obvious that how well a team performs also depends on the team they are playing against. Since a graph structure captures this, graph machine learning tasks are able to produce more accurate results than other data structures. The graph structure is a concept that can apply to many different domains such as social networks, molecular research, and more.

A broad problem among graph data analysis has been the scalability and generalizability of large-scale graphs. Dimensional reduction has been a useful technique when dealing with many situations when there are too many features. To handle nodes with many features, many algorithms have designed an embedding framework which makes it easier to perform machine learning on large inputs. However these frameworks often are transductive, only able to generate embeddings for a single fixed graph, thus are unable to efficiently generalize to unseen nodes and cannot generalize across different graphs. GraphSAGE, an inductive framework, has been designed to combat this as an embedding framework that can take advantage of features to generalize representations on unseen data. With GraphSAGE, we should now be able to efficiently use feature information in order to perform machine learning tasks in large-scale graphs. LPA-GCN is a node classification that combines the ideas of LPA and GCN. It takes

advantage of node labels across edges of the graph as well as node feature information which will lead to better classification performance.

What does better performance mean though? There is an issue with current benchmarks for graph data analysis. Many graph datasets are extremely small relative to real world datasets thus the algorithms developed to perform well on these small datasets are not scalable to the large real world datasets. Not only this, but there is also not a universal procedure to follow when performing and comparing experimental results. There are many studies that can take advantage of a graph structure which leads these studies to use their own data splits, metrics, and cross-validations, which makes it difficult to compare the resulting performance across other studies. This makes the world desperate for a benchmark that is able to compare results from various domain studies. OGB is a benchmark with the goal to develop a diverse set of challenging and realistic datasets that are large-scale, diverse, and allow machine learning tasks such as node, link, and graph predictions. This benchmark supports further research in graph machine learning by presenting an specific and extensive benchmark analysis for each dataset.

Following this benchmark, we will be comparing the results obtained from applying GraphSAGE, GCN-LPA, and a classic GCN to the Cora and OGB-arXiv datasets. The Cora and OGB-arXiv are citation datasets, thus are often split into two files, a content and a cites. The contents file contains the features of each paper, it could be something like the word frequency of each paper. Then we have the cites file which shows the citation of every paper. We can see that a citation dataset will create a directed graph where each paper will be a node, each citation will be an edge between papers, and the node features can be accessed through the contents file. What makes the Cora and OGB-arXiv different is the size of them. Cora consists of 2708 papers, 5429 links, 7 labels and 1433 features that indicate the presence of a word from a

dictionary. On the other hand, we have the OGB-arXiv which contains 169343 papers, 1166243 links, 40 labels, and 120 features. The objective in each of these datasets to be able to predict the labels, which is the primary subject area of the paper. Cora consists of more features, while OGB-arXiv has more nodes and edges. Based on OGB, Cora is a small-scale dataset and is easily solved with GNNs, while OGB-arXiv is a small-scale real-world dataset which will be more challenging to solve. Another flaw with the cora dataset is that it has a 42% node leak of information between features and labels.

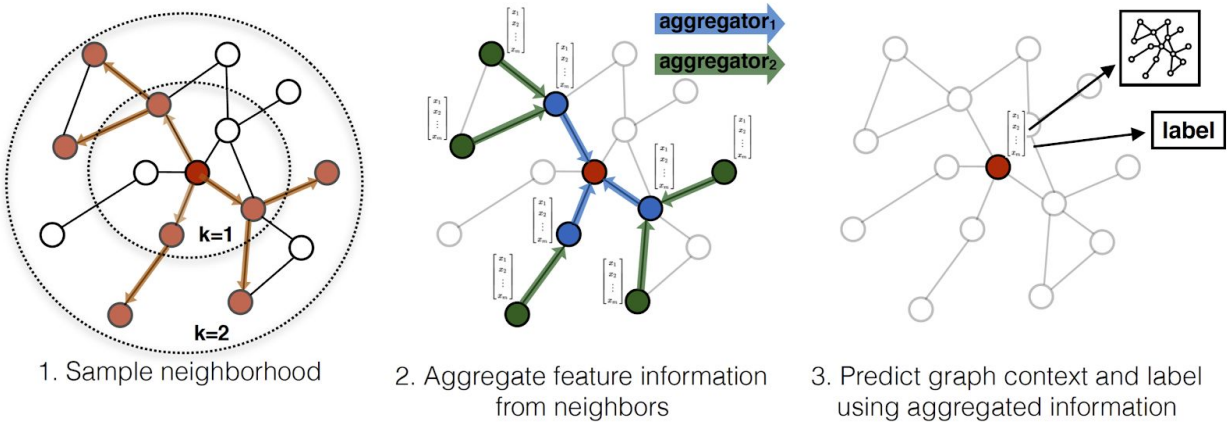
Methods

When implementing the traditional GCN, the GCN model is initialized with the number of nodes, n, number of hidden layers, *l*, and the number of classes, c. Inside the GCN model, we have the GCN layers which are initialized with an adjacency matrix of structure of the graph, A and the feature matrix, X. The shape of the adjacency matrix should be n x n. Thus the feature matrix would be n x f, where f is the number of features. We will use ReLU as the activation function, σ, of these layers and use a softmax to make the classification. To train and test the model, we will need to call the forward method of the GCN model, which will take an adjacency matrix of node edges, and a feature matrix. There is a parameter in the GCN forward method to specify whether you want to use Kipf & Welling's normalization of the adjacency matrix, A, or to leave it unnormalized.

$$f(H^{(l)}, A) = \sigma\left(AH^{(l)}W^{(l)}\right) \qquad f(H^{(l)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

The left formula shows how the layer will be computed without normalizing A, while the right shows how it will be computed with normalizing A. The model will use cross entropy loss to tune the weight parameters in each GCN layer.

GraphSAGE will create batches of neighborhoods and aggregate the information from these neighborhoods into a new feature matrix. The forward method will take in a batch of nodes, b and the depth size, k to develop a neighborhood. The model will then use b and k to create a subsample of A, an adjacency matrix that defines the neighborhood of every node in batch, b.



1. Sample neighborhood

2. Aggregate feature information from neighbors

3. Predict graph context and label using aggregated information

The batch of nodes, b and subsample of A will be then used as input to the GraphSAGE aggregators, mean and pooling that can also be specified as a parameter in the forward method.

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

1 $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
2 **for** $k = 1...K$ **do**
3 **for** $v \in \mathcal{V}$ **do**
4 $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
5 $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
6 **end**
7 $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
8 **end**
9 $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

The mean aggregator layer will output a feature matrix that contains the average of neighborhood features. The pooling aggregator layer will output the original feature matrix b, concatenated with the pooling of the neighborhood for each node in b. The forward algorithm here should return a feature matrix for each node. In order to make the algorithm more efficient, we can use matrix multiplication instead of a for loop in line 3.We will multiply the matrix of the last iteration, h, with A. This results in a matrix that contains the sum of the features of every node. We can tweak this updated matrix depending on which aggregator we decide to use.

What makes the GraphSAGE loss function interesting is that it only requires the output of the GraphSAGE model as input.

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right)$$

The loss function is calculated based on the similarity between the input, thus it doesn't need to know the true label of a node. It is also possible to use stochastic gradient descent as the loss function. The parameters that will be tuned are the weights of the aggregator layers.

In order to implement LPA-GCN, we need to understand the implementation of LPA and GCN. We will make the assumption that two connected nodes are likely to have the same label,

thus it propagates labels iteratively along the edges. LPA will require the label distribution for each node, and A.

$$Y^{(k+1)} = D^{-1} A\, Y^{(k)},$$
$$y_i^{(k+1)} = y_i^{(0)},\ \forall\, i \le m.$$

These are the two steps of the propagation rule. First we will use the normalized A to propagate labels to their numbers. Then we will reset labeled nodes to the initial values.

When unifying LPA and GCN, we are unable to use Kipf & Welling's normalization, instead we use the one shown in the formula above. Thus when we apply the GCN layer to our LPA-GCN model, it will look like:

$$X^{(k+1)} = \sigma\left(D^{-1} A X^{(k)} W^{(k)}\right).$$

With LPA, we can learn the optimal edge weight, A* by minimizing the predicted labels of LPA:

$$\begin{aligned} A^* &= \arg\min_A L_{lpa}(A) \\ &= \arg\min_A \frac{1}{m} \sum_{v_a : a \le m} J(\hat{y}_a^{lpa}, y_a), \end{aligned}$$

The same can be done to find the optimal weight matrix, W*.

$$\begin{aligned} W^* &= \arg\min_W L_{gcn}(W, A^*) \\ &= \arg\min_W \frac{1}{m} \sum_{v_a : a \le m} J(\hat{y}_a^{gcn}, y_a), \end{aligned}$$

With A*, we will also be able to find the optimal D, D*. Then our optimized and normalized GCN layer will look like:

$$X^{(k+1)} = \sigma(D^{*-1}A^*X^{(k)}W^{(k)}),\ k = 0, 1, \cdots, K-1.$$

Where W will be replaced with the optimal W, W*. Theoretically, this is an improved GCN model and we would expect better performance results.

Results

I will be testing the three models above on the Cora and OGB data sets. In the event that the OGB data set takes too much memory, I will use a sample of it to test the results. Unable to apply my models to the large dataset, I decided to use it on a sample with about the same number of rows as the Cora data set. I would expect similar results.

We will use the results from 100 epochs:

Test Accuracies in 100 Epochs

Method	Cora	OGBSample ~2000 rows
GCN	25.46%	1.40%
LPA-GCN	11.44%	15.42%
GraphSAGE	20.30%	19.16%

Comparing our results to OGB's results on the OGB data set.

Table 6: **Results for `ogbn-arxiv`.**

Method	Accuracy (%)		
	Training	Validation	**Test**
MLP	63.58±0.81	57.65±0.12	55.50±0.23
NODE2VEC	76.43±0.81	71.29±0.13	70.07±0.13
GCN	78.87±0.66	**73.00**±0.17	**71.74**±0.29
GRAPHSAGE	82.35±1.51	72.77±0.16	71.49±0.27

We can see that the results from my implementation of the models are completely off. I believe that the flaw is in my implementation of the GCN because we can see a huge increase in accuracy after applying the LPA to the GCN. Although my results are extremely off and my models are poor, it is still possible to compare the performance of the three models. We can see the variants of the GCN increase the accuracy of the model which is what we would expect. Surprisingly, the results from both models are different.

Conclusion

From the results in my implementation, I cannot conclude anything. This is because both the Cora and OGB sample data that I tested the performance model on produced extremely different results. I also failed to make my model work on real-world datasets as it was running into memory issues when used on the actual OGB-arxiv data set. There are many ways that this paper could be improved. For example, my GCN's implementation needs to be reviewed so that it could achieve the same performance as the GCN that OGB used in their results.

References

Inductive Representation Learning on Large Graphs.

W.L. Hamilton, R. Ying, and Jure Leskovec *arXiv:1706.02216 [cs.SI]*, 2017.

Unifying Graph Convolutional Neural Networks and Label Propagation.

Hongwei Wang, and Jure Leskovec. *arXiv:2002.06755 [cs.LG]*, 2020.

Open Graph Benchmark: Datasets for Machine Learning on Graphs.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, Jure Leskovec. *arXiv:2005.00687 [cs.LG]*, 2020

Graph Convolutional Networks.

Thomas Kipf. 2016. URL: https://tkipf.github.io/graph-convolutional-networks/