

Importance of Inductive Algorithms

Steven Liu

Abdullatif Jarkas

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.

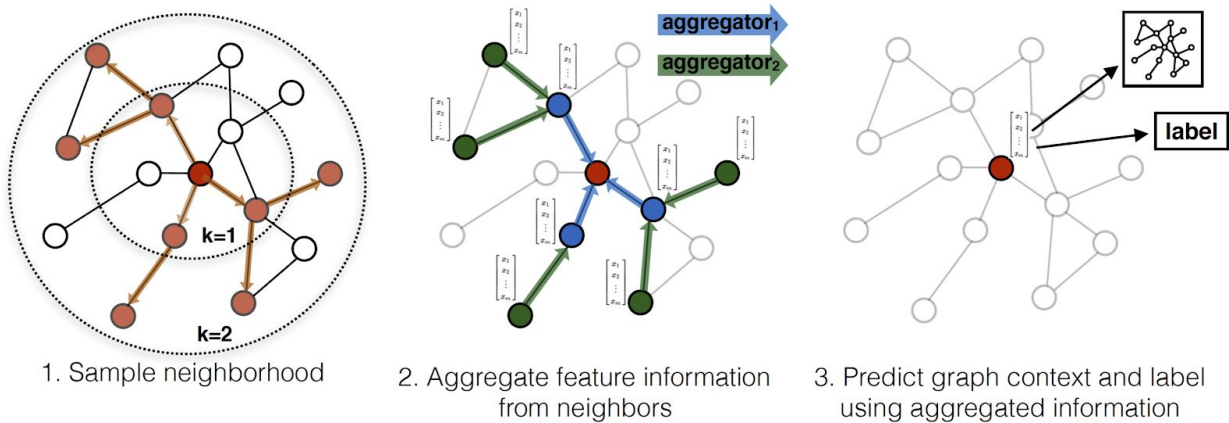
Methods

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 $n \times n$. Thus the feature matrix would be $n \times f$, 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.

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

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 .



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 $\mathbf{W}^k, \forall k \in \{1, \dots, K\}$; non-linearity σ ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, \dots, K\}$; 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  $k = 1 \dots K$  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      $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k))$ 
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 \mathbf{b} , concatenated with the pooling of the neighborhood for each node in \mathbf{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, \mathbf{h} , with \mathbf{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.

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

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 .

$$Y^{(k+1)} = D^{-1} A Y^{(k)},$$

$$y_i^{(k+1)} = y_i^{(0)}, \forall i \leq m.$$

These are the two steps of the propagation rule. First we will use the normalized A to propagate labels to their neighbors. 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:

$$A^* = \arg \min_A L_{lpa}(A)$$

$$= \arg \min_A \frac{1}{m} \sum_{v_a: a \leq m} J(\hat{y}_a^{lpa}, y_a),$$

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

$$W^* = \arg \min_W L_{gcn}(W, A^*)$$

$$= \arg \min_W \frac{1}{m} \sum_{v_a: a \leq m} J(\hat{y}_a^{gcn}, y_a),$$

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

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

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

Here, we will be benchmarking the three methods (GCN, GCN-LPA, and GraphSage) on the Cora and Arxiv datasets and attempt to compare them with the results from the research paper. Our training is based on an 80-10-10 split, in which 80% of the data is used trained and 10% of the data is used for each, the validation and the test set. Assuming a learning rate of 0.01 and Epochs set to 100, we develop the following results. Test Accuracies (100 Epochs, LR = 0.01)

Test Accuracies (Epochs=100, LR = 0.01)

Method	Cora	OGBSample
GCN	75.375%	72.5%
GCN-LPA	85.13%	N/A (Memory Error)
GraphSAGE	50.7%	N/A (Memory Error)

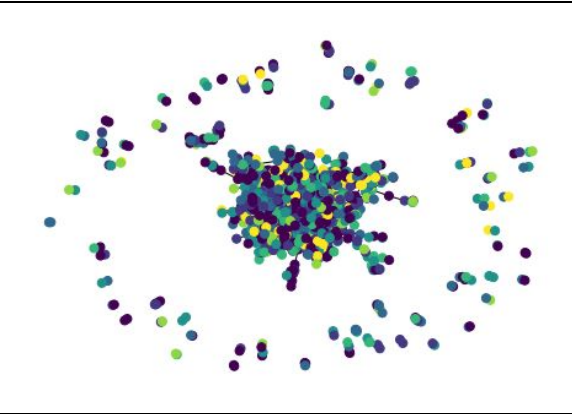
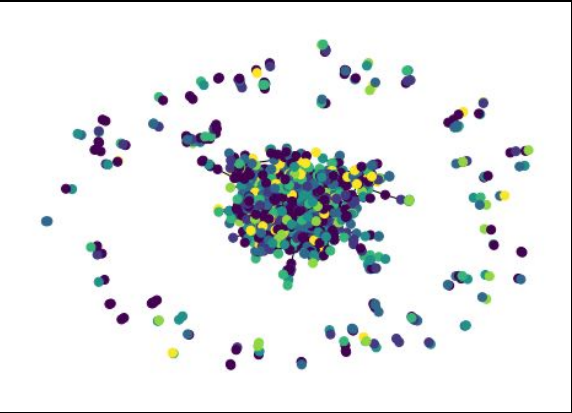
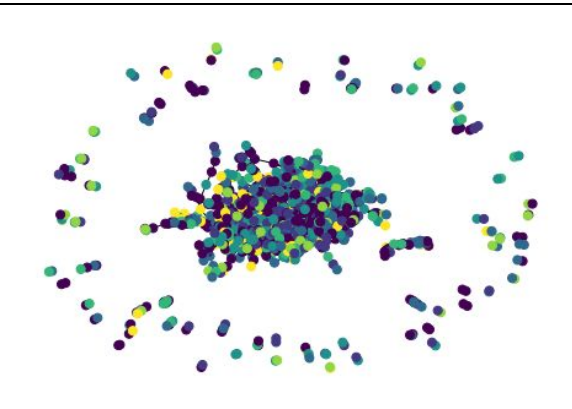
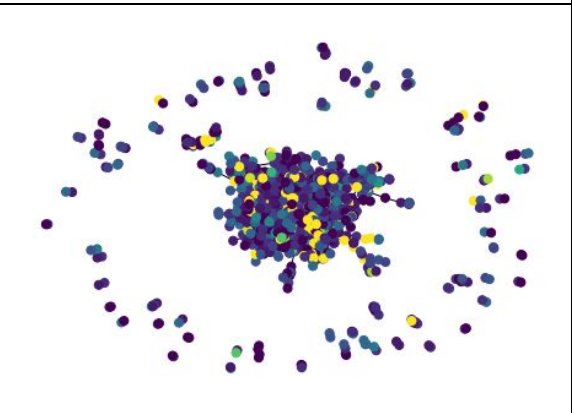
- **Cora Dataset**

Replication Paper Accuracies:

Method	Cora
MLP	64.6 ± 1.7
LR	77.3 ± 1.8
LPA	85.3 ± 0.9
GCN	88.2 ± 0.8
GAT	87.7 ± 0.3
JK-Net	89.1 ± 1.2
GraphSAGE	86.8 ± 1.9
GCN-LPA	88.5 ± 1.5

When comparing our accuracies with the results from the replicated paper, *Unifying Graph Convolutional Neural Networks and Label Propagation*, We can see that we under achieved in our results with our GCN, GCN-LPA, and especially GraphSage. This could be the result of our set hyperparameters such as our learning rate, epochs, etc. Through minor tweaks with our hyperparameters we can expect closer results to the paper's results.

Graph Visualization (Cora)

True Labels	Predicted Labels (GCN)
	
Predicted Labels (GCN-LPA)	Predicted Labels (GraphSAGE)
	

- **OGB Arxiv Dataset**

Table 6: Results for ogbn-arxiv.

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

When comparing our results with the results from the benchmark on the left, we had a huge discrepancy with our GCN test accuracy (27% vs. 71%). We predict that we may have developed an issue in how we developed our data ingestion model for the dataset.

Hyperparameter tuning may help marginally but we should not expect a 40% jump just from that.

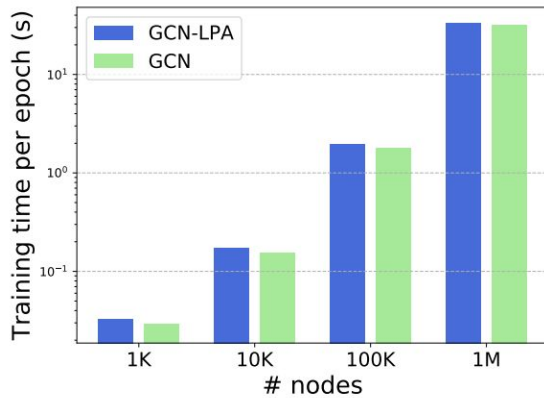
- **Run-Time**

Another metric we took note of was run-time. We calculated our training run time by initiating a timer before our training for each one of our models. Our run-time benchmark is seen in the table below.

Training Run Time (Seconds for 100 Epochs)

Method	Cora	OGBSample
GCN	1.231 seconds	145.89 seconds
GCN-LPA	29.609 seconds	N/A (Memory Error)
GraphSage	452.67	N/A (Memory Error)

Training Run Times of Wang and Leskovec
(GCN vs. GCN-LPA)



When comparing our run times to the results of the paper by Wang and Leskovec, we noticed a great discrepancy. As you can see with our models running on the Cora dataset of 2,708 nodes (quite small), we can see that our GCN ran 24 times quicker than our GCN-LPA Model. And this is excluding loading data. While the paper did reveal GCN to be a faster model, it was quite marginal as seen in the figure to the left.

This probably is due to the way we set up our GCN-LPA's forward function, which is quite computational heavy. Our training code and dataset loading methods were identical to both the GCN and GCN-LPA models, so that could not be the source of the slow run time.

Conclusion

From the results above, we cannot generally conclude the results of the paper which argued that the hybrid GCN-LPA was notably more accurate than its counterpart, the GCN model, while retaining relatively the same training run-time. As we ran into run-time issues with large datasets, such as the OGB Arxiv dataset. Based on our implementation, GraphSAGE and GCN-LPA may not work on large-scale graph networks.

However, when looking strictly at the Cora dataset and model performances on it, we can conclude and agree with the paper that the hybrid GCN-LPA model outperforms the GCN. However, through our work and code, we cannot confirm that the GCN-LPA model is marginally more intensive in training run time, as our results revealed the complete opposite, showing significantly greater training run times in our GCN-LPA model.

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/>