Accesing the Fairness Level of the Training Set of a Blackbox Graph Neural Network Model through Node Masking

Theodora Ko

September 2022

1 Introduction

Ranging from visualising social network service connections amongst people to modelling polypharmacy side effects based on protein interactions, graphs are a useful form of data representation in various different academic fields. Recently, there has been a growing interest in adapting a neural network model on graph format data, or **Graph Neural Networks** (GNNs), and, moreover, interpretation of predictions by GNNs. This paper focuses on machines that perform node classification. It describes an attempt to debias the output data through node masking input data and demonstrates that correcting a training data to reflect the group fairness may lead to higher accuracy. This paper proposes the challenge GNN poses on Explainable Artificial Intelligence (XAI) and definitions used throughout the paper. Then, the paper explains the framework of our approach to the problem and provides pseudocode of the actual codes used in the experiment. The codes were implemented using mainly Pytorch Geometric (PyG) and NetworkX libraries. Also, note that in our approach, although we used Planetoid dataset provided by PyG library and GraphSAGE model, any dataset and any GNN model can be analysed using our proposed framework.

2 Problem Statement & Definitions

The inspiration for this paper rose from understanding the definition of a blackbox model. As useful as artificial intelligence (AI) and machine learning (ML) become in terms of their efficiency in assessing and producing predictions based on dense, big data, their accuracy is often met with scepticism due to their rather unclear data-processing method, the reason ML models are considered a **"black-box"**. GNN in particular, from producing the first set of hyperparameters through the first layer through producing those of the final layer, has a particularly complex data transformation process due to the larger size of an adjacency matrix. They are considered black-box as any GNN model's behaviour cannot be comprehended easily, regardless of having access to its structure and weights. This proposes challenges to expanding explainability of AI over graph data, albeit their applicability.

3 The Proposed Framework

However, although we do not have access to the transformation process within the black box, we do have access to the input and the output dataset, leading to our proposed framework. By manually masking a node in the input training data at a time, we are able to take a look at the influence of a single node in the overall output and, furthermore, access the fairness of our training data set.

3.1 Node Masking

More formally, we will define our approach as such:

Consider a node classification problem where we map some input space X with n many training points from $z_1, \cdots, z_n$ to an output space Y. Given n training points, our blackbox GNN model will produce n outputs that labels each training points with one of k many labels.

Our initial goal is to access the influence of a node to a model's prediction. We define a test as masking all nodes in the training set, one node at a time, and retraining the model and comparing the output accuracy. We record the index of a node that led to an increase in output accuracy. We define such node as an influential node. Then, we will perform the test multiple times to get a final list of influential nodes and perform various tests based on it.

Algorithm 1 `loo_pipeline(model, dataset, data, train_mask, test_mask, which_node, n_epochs)`

Output: compute the output accuracy of a new model trained based on a new training set that masks the given node

`new_model` ← make a copy of `model`
`new_mask` ← Define a new mask by masking `which_node`

`train_acc_list, test_acc_list, loss_list, misclassified, predictions` ← train using `new_mask` `n_epochs` times
`loo_output` ← new_model(data.x, data.edge_index)
`loo_accuracy` ← comput accuracy of the new model
return `loo_output, loo_accuracy`

Algorithm 2 `check_pipeline(dataset)`

Output: find influential points of a datast by performing an individual node masking on all nodes in the training set

```
influential_nodes ← an empty dictionary
train_set ← a list of node indces for nodes in the training set
original_accuracy ← compute the original accuracy by calling train(model, dataset)

for node ∈ train_set do
    new_output, new_accuracy ← Call loo_pipeline(node)
    if original_accuracy <= new_accuracy then
        influential_nodes[node] = new_accuracy
        original_accuracy = new_accuracy
    end if
end for
return influential_nodes
```

Algorithm 3 `repeat_pipelines(model, dataset, data, epoch_size, test_num)`

Output: find influential points of a datast by performing an individual node masking on all nodes in the training set multiple times

`influential_nodes` ← empty dictionary
for `test` ∈ `test_num` **do**
  `new_influential_nodes`= call `check_pipeline(dataset)`
  `influential_nodes[test]` = `new_influential_nodes`
end for
return `influential_nodes`

Once we have collected a list of influential nodes, we ran several different tests to identify common features of the influential nodes such as node similarities and node centralities. The most distinctive feature they had in common was their node centrality indices in terms of their degrees. The following images are the distribution of degree centrality and betweeness centrality of the training set of the Cora dataset.

Algorithm 4 `node_analysis(dataset, influential_nodes, task)`

Input: dataset, a list of influential nodes, and task designating which metrics to compute
Output: three lists of node indices in training, validation, and test set

`graph` ← transform the torch dataset to NetworkX type
`dictionary` ← empty dictionary

if `task` == 'node_similarity' **then**
  `sim` ← compute simrank_similarity of the graph
  return `sim`
else if `task` == 'Degree Centrality' **then**
  `dictionary` ← compute degree centrality of all the nodes in the graph
else if `task` == 'Betweeness Centrality' **then**
  `dictionary` ← compute betweeness centrality of all the nodes in the graph
end if

`result` ← empty dictionary
for `node` ∈ `influential_nodes` **do**
  `result[node]` = `dictionary[node]`
end for
return `result`

3.2 Redefining Training Sets

Once we have identified influential nodes whose masking leads to an accuracy improvement, we redefine training sets based on influential nodes and other metrics that we have found by analysing the influential nodes. We have mainly trained the model based on four following sets. Note that Cora dataset has 7 labels and 140 nodes in total for the training set.

Algorithm 5 `designated_split(which_nodes, num_nodes, num_test_nodes)`

Input:

1. which_nodes: designated nodes to be masked for the training set
2. num_test_nodes: size of the test set
3. num_nodes: number of the total nodes in the data

Output: three lists of node indices in training, validation, and test set

`train_mask`, `val_mask`, `test_mask` ← initialise a torch tensor of size `num_nodes` with zeros
`random_nums` ← generate a torch tensor of random numbers from 0 to `num_nodes`
`count` = 0

for i ∈ range(`num_nodes`) **do**
  if `count` == num_test_nodes **then** break
  end if
  if i ∉ `which_nodes` **then**
    `test_mask`[i], `val_mask`[i] = True, False
    `count` += 1
  end if
end for

for `node` ∈ `which_nodes` **do**: `train_mask[node]` = True `val_mask[node]` = False
end for

return `train_mask, val_mask, test_mask`

Algorithm 6 `random_split(num_train_nodes, num_val_nodes, num_test_nodes, num_nodes)`

Input:

1. num_train_nodes: size of the training set
2. num_val_nodes: size of the validation set
3. num_test_nodes: size of the test set
4. num_nodes: number of the total nodes in the data

Output: three lists of node indices in training, validation, and test set

`train_mask`, `val_mask`, `test_mask` ← initialise a torch tensor of size `num_nodes` with zeros
`random_nums` ← generate a torch tensor of random numbers from 0 to `num_nodes`
for i ∈ range(num_train_nodes) **do**
  set train_mask[random_nums[i]] = True
end for
for i ∈ range(num_val_nodes) **do**
  set val_mask[random_nums[num_train_nodes + i]] = True
end for
for i ∈ range(num_test_nodes) **do**
  set test_mask[random_nums[num_val_nodes + num_train_nodes + i]] = True
end for
return `train_mask`, `val_mask`, `test_mask`

Algorithm 7 `generate_manual_data(data, task, num_test_nodes)`

Input:

1. data
2. task: '25quantile', '75quantile', 'random', 'designated'
3. num_test_nodes: number of nodes in the test set
4. other arguments depending on task

Output:

1. if task == 'designated', should give a list of nodes to use for training, validation, and test
2. if task == 'random', should assign the size of the training, validation, and test data set
3. if task == 'quantile', should give a size of the quantile and whether lower/higher than the quantile we are taking

if task == 'random' **then**
  `train, val, test` ← call random_split()
else if task == 'designated' **then**
  `train, val, test` ← call designated_split()
else if task == 'quantile' **then**
  `node_list` ← call find_nodes_given_quantile()
  `train, val, test` ← call designated_split(`node_list`)
end if
`new_data` ← create a torch data using Data(train_mask=train, val_mask=val, test_mask=test),
return `new_data`

3.2.1 Training sets based on node centrality

In terms of node centralities, we redefined the training set using

1. 20 nodes of highest degree centrality per class
2. 20 nodes of lowest degree centrality per class,
3. 20 nodes of highest betweeness centrality per class,
4. 20 nodes of lowest betweeness centrality per class.

Algorithm 8 `sorted_classification(dataset, dictionary)`

Output: returns a dictionary that uses node labels as the keys and a sorted list of tuple (node, centrality value) as the values

```
labels = number of labels
sorted_dictionary ← sort dictionary
    centrality_per_label = {}
for node ∈ labels do: centrality_per_label[node] = {}
    node_classes = dataset[0].y.numpy()
   for i ∈ range(len(sorted_dictionary)) do
      key = node_classes[i]
      centrality_per_label[key][i] = sorted_dictionary[i]
      return centrality_per_label
```

3.2.2 Training sets based on influential nodes

In terms of influential nodes, we redefined the training set using

1. the same training set excluding all influential nodes
2. the same training set excluding influential nodes that appeared more than once in the tests
3. the same training set excluding influential nodes that all tests had in common

3.2.3 Training sets based on node attributes

Furthermore, we have noticed that there is an uneven representation of the label distribution in the training set, leading to redefine the training set using node centralities by reflecting the label ratio in the number of nodes per class and repeating **3.2.1**.

1. nodes of highest degree centrality per class,
2. nodes of lowest degree centrality per class,
3. 20 nodes of highest betweeness centrality per class,
4. 20 nodes of lowest betweeness centrality per class.

Algorithm 9 `centrality_training_Ratio(dataset, target_nodes, target_type, centrality_type, high_or_low)`

```
dictionary, _ ← node_analysis(dataset, target_nodes, centrality_type)
num_nodes ← number of total nodes in the graph
centrality_per_class ← sorted_classification(dataset, dictionary, False)
train_mask ← empty list

for i in centrality_per_class.keys() do
    ratio ← round(len(centrality_per_class[i])/ num_nodes · 140)
    append train_mask with centrality_per_class[:ratio * 140]
    return_x_nodes(centrality_per_class, i, ratio, 'high_or_low)
end for

new ← generate_manual_data(data, 'designated', which_nodes=train_mask, num_test_nodes =
1000)
new_train_acc, new_predictions ← train the model using new

return new_train_acc, new_predictions =0
```

4 Experiment Results & Evaluation

Our experiment gave a range of accuracy improvements at varying degrees. In terms of influential nodes, exclusion of individual or all influential nodes did not lead to significant improvement of the accuracy. However, in terms of node centralities, nodes selected based on degree centralities led to better accuracy than betweenness centrality and, moreover, led to highest accuracy improvement when the training set also reflected the label distribution ratio of the entire input data. We correlate such to how there is a more extreme skewness of the data in terms of degree centralities, leading to our focus on redefining a new training set in terms of node degree centralities.

5 Conclusions

In conclusion, our experiment demonstrates the importance of having a training set that properly reflects the distribution of node labels of the entire input data. By correcting the input data into reflecting the distribution of node labels based on selected metrics, the user would be able to improve the prediction accuracy. This also offers an insight to diagnosing a blackbox GNN model's accuracy such that there is a high possibility for a model with unfair training set to produce a faulty output prediction.

6 Acknowledgement

I would like to acknowledge my internship supervisor, Professor Claire L Donnat, and my internship manager Zachary Rudolph. Without their help, it would have been impossible for me to conduct the research and produce a tangible result.