A Technical And Simulation Approach on RIM: Reliable Influence-based Active Learning on Graphs

Amir Eskandari

Statistical Learning Theory Course, Fall 2021, Instructor: Prof. Vahid Pourahmadi
Amirkabir University of Technology, Tehran

February 15, 2022

Message passing is the core of most graph models such as Graph Convolutional Network (GCN) and Label Propagation (LP), which usually require a large number of clean labeled data to smooth out the neighborhood over the graph. However, the labeling process can be tedious, costly, and error-prone in practice. "RIM" Algorithm offers a novel algorithm and architecture based on Influence maximization (both quantity and quality) assisted by the message-passing method. The quantity of influence measures how much influence a labeled node does on an unlabeled node in a GRAPH, and quality dealing with honesty/trust of influence toward the right label. In this paper, I don't aim to discuss the detail of the RIM algorithm, but I am willing to interpret details of written source code for algorithm simulation. I use two datasets "Cora" and "Citeseer". Rest of the paper is divided into 3 sections: in the first section, I explain "RIM" briefly, then I explain the details of source code that is used for simulating "RIM", Also in the last section I interpret result of re-simulation. In this paper, I assume readers already is familiar with Graph, Graph Convolution Networks, Label Propagation, and Active Learning.

1 RIM Overview

RIM combined two conceptually separated part that work in an end-to-end manner (Zhang et al., 2021). The first part, for selecting the most influential subset of training nodes in the graph to label by oracle. The second part is just a simple GNN and LP for classifying in semi-supervised setting. RIM innovation embedded in the first part, RIM measure both quality and quantity influence of labeled node in node selection part.

1.1 Node selection

Node selection have several components. RIM, the first framework that considers both the influence quality and influence quantity. At each batch of node selection, RIM first measures the proposed reliable influence quantity and selects a batch of nodes that can maximize the number of activated nodes, and then it updates the influence quality for the next iteration. The above process is repeated until the labeling budget B runs out. In the nexts subsection I introduce RIM components in node selection.

1.1.1 Influence propagation (quantity)

In this section I introduce two concepts that measure quantity of feature/label influence of specific nodes on the rest of the graph. I use equation 1 for measuring feature influence quantity of v_i on v_j after k iteration, which numerator is $\hat{I}_f(v_j, v_i, k) = \|E[\partial X_j^{(k)} / \partial X_i^{(0)}]\|$.

$$I_f(v_j, v_i, k) = \frac{\hat{I}_f(v_j, v_i, k)}{\sum_{v_w \in V} I(v_j, v_w, k)} \tag{1}$$

For measuring label influence quantity I use equation 2. Equation 2 accept one-hot label vector as y.

$$\hat{I}_l(v_j, v_i, k) = \|E[\partial y_j^{(k)} / \partial y_i^{(0)}]\| \tag{2}$$

1.1.2 Influence Quality Estimation

As I said before RIM is the first framework that uses both quantity and quality for labeling. The quality measuring is urgent because the oracle labels nodes with error due to a lot of reasons. Might a specific node's influence quantity be strong but, the influence might be wrong. So RIM should label subset of nodes which are the best in both quantity and quality.

The quality of influence is based on **homophily** properties in networks. It says, if nodes are near or

connected in a graph they are likely to have the same label. In other words, if two nodes are k-hub neighbors the reliability influence increase between them by lowering the k. In equation 3 I use mentioned properties for measuring influence reliability between two nodes. In this equation α is oracle accuracy, c is the number of classes.

$$r_{v_i \to v_j} = \frac{\alpha s}{\alpha s + (1-\alpha)\frac{1-s}{c-1}} \tag{3}$$

The most important and influential component of equation 3 is **s**. It's cosine similarity between v_i and v_j in graph after k iteration by model free GCN and LP. In GCN, RIM use $\hat{X}^{(k+1)} = \hat{D}^{-1}\hat{A}\hat{X}^{(k)}$ as model free massage passing. In the equation, $\hat{X}$ is the $n \times p$ feature matrix n is the number of samples and p is the dimension of samples. In LP, RIM use label matrix $\hat{Y}$ for calculating similarity after k iteration by $\hat{Y}^{(k+1)} = \hat{D}^{-1}\hat{A}\hat{Y}^{(k)}$. $\hat{Y}$ contain one-hot label vectors.

After calculating pair reliability, the normalized reliability of v_i is:

$$r_{v_i} = \sum_{v_i, \hat{y_i} = \hat{y_j}} \hat{r}_{v_i} r_{v_i \to v_j} \tag{4}$$

Which $\hat{r}_{v_i}$ is $\hat{r}_{v_i} = \frac{r_{v_i}}{\sum_{v_q \in V_l, y_j = y_q} r_{vq}}$. Equation 4 is normalized influence quality of v_i on different class label. RIM use r_{v_i} for selecting labeling candidate nodes in the next iteration.

1.1.3 Reliable Influence Quantity Score

Given the influence quality r_{v_i}, the reliable influence quantity score of node v_i on node v_j after k-step feature/label propagation is:

$$Q(v_j, v_i, k) = r_{v_i} I(v_j, v_i, k) \tag{5}$$

where $I(v_j, v_i, k)$ is $I_f(v_j, v_i, k)$ for GCN and $I_I(v_j, v_i, k)$ for LP.

The reliable influence quantity score $Q(v_j, v_i, k)$ is determined by: (1) The influence quality of the labeled influence source node v_i. (2) The feature/label influence score of v_i on v_j after k-step propagation. RIM selection based on quality and quantity is unified by equation 5. Equation 5 is an measure for selecting node labeling candidate by both quality and quantity of influence. As I said before the objective function maximize number of activated node in graph by selecting a subset of node for labeling. Motivated by this, I assume an unlabeled node v_j can be activated if and only if the maximum influence magnitude satisfies:

$$Q(v_j, V_l, k) > \theta \tag{6}$$

where $Q(v_j, V_l, k) = max_{v_i \in V_l} Q(v_j, v_i, k)$ is the maximum influence of V_l on the node v_j, and the threshold θ is a parameter which should be specially tuned for a given dataset.

1.1.4 Node Selection Algorithm

After all, introducing the node selection part components. I can introduce an algorithm. The algorithm is shown below:

Algorithm 1: Batch Node Selection.

Input: Initial labeled set $\mathcal{V}_0$, query batch size b, and labeling accuracy α.
Output: Labeled set $\mathcal{V}_l$

$\mathcal{V}_l = \mathcal{V}_0$;
for $t = 1, 2, \ldots, b$ **do**
  Select the most valuable node $v^* = \arg\max_{v \in \mathcal{V}_{train} \setminus \mathcal{V}_l} F(\mathcal{V}_l \cup \{v\})$;
  Set the influence quality of v^* to the labeling accuracy α;
  Update the labeled set $\mathcal{V}_l = \mathcal{V}_l \cup \{v^*\}$;
Update the influence quality of nodes in $\mathcal{V}_l \setminus \mathcal{V}_0$ according to E.q.4
return $\mathcal{V}_l$

This algorithm solve the objective function below in greedy way:

$$maxF(V_l) = |\sigma(V_l)|, s.t. V_l \subseteq V, |V_l| = B \tag{7}$$

where $\sigma(V_l)$; Given a set of labeled seeds V_l , the activated node set $\sigma(V_l)$ is a subset of nodes in V that can be activated by V_l:

$$\sigma(V_l) = \bigcup_{v \in V, Q(v, V_l, k) > \theta} \{v\}. \tag{8}$$

Summery of algorithm: After measuring both quality and quantity of influence, selection start. Selection executes in mini batches. In each batch quality estimation updates. In each batch, the selection is implemented in a greedy way which means algorithm select the best subset of nodes (size mini batch) that maximize number of activated nodes in current batch. The ultimate goal is to select the best subset of train nodes for labeling that maximize the number of activated nodes in the graph. Finding this subset is an NP problem. As mentioned before RIM employ a greedy algorithm to approximately find the best subset.

1.2 Training

After node selection, the problem summarize to simple semi-supervised classification. A few labeled nodes and graph structures are available so for solving this classification problem. GCN and LP are good choices. The main difference is RIM use reliability score in loss function for training GCN. The intuition is nodes with a higher level of reliability should influence more in the loss function (equation 9). Loss function for GCN is weighted cross entropy, weight by reliability score.

$$L = -\sum_{v_i \in V_l} r_{v_i} \mathbf{y}_i log(\hat{\mathbf{y}}_i) \tag{9}$$

2 Implementation details

The RIM source code is written with python3. The code is available in git-hub[1]. But it just programmed

[1] https://github.com/zwt233/RIM

RIM for GCN. I develope the code also for LP. In addition to I wrote comment and description for functions and codes for better understanding by reader. The source code contain 4 script. **1) graphConvolution 2) utils 3)RIM-GCN 4)RIM-LP**. This 4 scripts work for implementing both LP and GCN based RIM algorithm. For LP based, I run RIM-LP, this script run RIM node selection and classification in an end-to-end manner based on LP. Also RIM-GCN do the same based on the GCN. Original source code available in git-hub just contain 3 script, it doesn't have RIM-LP. In the next subsections I describe each script in detail.

2.1 graphConvolution.py

It's easiest one to explain. It just contains one class which named the same as the script name. This class used for Graph Convolution layer. Equation 10 implemented in this script. The written code for the GCN class is shown in Figure 1. It contains usual methods like forward, reset parameter that needed for all models in Deep Learning. Also attributes like input/output feature dimension are defined in init method.

$$X^{(k+1)} = \sigma(\hat{D}^{-1}\hat{A}X^{(k)}W^{(k)}) \tag{10}$$

```
class GraphConvolution(Module):

    def __init__(self, in_features, out_features, bias=True):
        super(GraphConvolution, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.FloatTensor(in_features, out_features))
        if bias:
            self.bias = Parameter(torch.FloatTensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        stdv = 1. / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-stdv, stdv)
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)

    def forward(self, input, adj):
        support = torch.mm(input, self.weight)
        output = torch.spmm(adj, support)
        if self.bias is not None:
            return output + self.bias
        else:
            return output

    def __repr__(self):
        return self.__class__.__name__ + ' (' \
               + str(self.in_features) + ' -> ' \
               + str(self.out_features) + ')'
```

Figure 1: *CGN Class code*

2.2 utils.py

This script is like a toolbox. Contains 8 functions that I use for implementing RIM. The main function in utils is **load-data**. This function load and transform the raw dataset in the format that we are interested in. Other functions, **parse-index-file**: this function used in load-data. **normalize**: it's clear from the name, normalized data. **accuracy**: this function accepts two arguments, output and label. output is the prediction of the model and label is ground truth label, then function outputs the accuracy for output based on ground-truth labels. **set-seed**: set the randomness seed. **sparse-mx-to-torch-sparse-tensor**: convert a scipy sparse matrix to a torch sparse tensor. **aug-normalized-adjacency**: convert A to $W = D^{-1}A$. In other words,This function normalized the adjacency matrix. Last function **aug-random-walk**.

Due to the length of the codes and also several functions, bringing the whole code is out of the scope of the article. The interested readers can access source code easily and read utils.

2.3 RIM-GCN.py

This script is somehow the "main" script for running GCN based RIM. In this script node selection and classification are implemented in an end-to-end manner. In the GCN-based RIM everything is feature vector oriented; influence quantity and quality computed by feature vectors, Also classification.

This script contains 347 lines, and bringing and explaining all the line is out of the scope of the article. I just highlight key and essential parts.

2.3.1 node selection

In Figure 2 the first line of code is executed to make dataset ready. As discussed in the previous subsection load-data is a function for loading and transforming data. This function only accepts name of the dataset (Cora or Citeseer). load-data outputs adjacency matrix of a graph, that indicates the relationship between nodes, features matrix containing each node's feature vector, and Also index of the train, validation, and test nodes in the graph.

```
adj, features, labels, idx_train, idx_val, idx_test = load_data(dataset="citeseer")

reliability_list = np.ones(num_node)
num_zeros = np.zeros(num_node)
num_ones = np.ones(num_node)
labels = list(labels.cpu())
idx_val = list(idx_val.cpu())
idx_test = list(idx_test.cpu())
idx_avaliable = list()
for i in range(num_node):
    if i not in idx_val and i not in idx_test:
        idx_avaliable.append(i)
```

Figure 2: *Loading data and defining some variables*

Then some variables are defined like reliability-list which contains reliability score of each node. Also, a new index list called idx-available is defined which contains index of training nodes that are not selected yet for labeling. Also some format converting operation are executed.

```
# add noise
label_list=[]
prob_list = np.full((num_class,num_class),(1-oracle_acc)/(num_class-1)).tolist() #Error Probability

for i in range(num_class):
    label_list.append(i) #Just a list contain classes num : [0,1,...,Num_classes - 1]
    prob_list[i][i]=oracle_acc #Put each label trust accuracy to oracle_acc
for idx in idx_avaliable:
    labels[idx]=torch.tensor(random_pick(label_list,prob_list[labels[idx].item()]))
```

Figure 3: *Adding noise to ground truth label*

In Figure 3 I added noise to ground-truth labels. In Figure 2 we saw that load-data outputs labels list. This label list is ground-truth and no labeling mistake happens. In a perfect labeling situation quality measurement is unnecessary. The code written in Figure 3 is for adding noise, the procedure is taking the nodes label and randomly picking a label for nodes by assuming the probability of picking correct is oracle accuracy and picking another label class is $\frac{\text{oracle error}}{c-2}$ where c is the number of classes. This adding noise procedure is just executed for training nodes. validation and test node labels are ground truth label.

In the first line of Figure 4 familiar function called. This function introduce in utils section, this function change input $\hat{A}$ to $\hat{D}^{-1}\hat{A}$.

```
adj = aug_normalized_adjacency(adj)
features = features.cpu()
adj_matrix = torch.FloatTensor(adj.todense()).cpu() #Change Format to Tensor
adj_matrix2 = torch.mm(adj_matrix,adj_matrix).cpu() #A*A
aax_feature = torch.mm(adj_matrix2,features) #A*A*X
aax_feature = np.array(aax_feature.cpu())
adj_matrix2 = np.array(adj_matrix2.cpu())
features = features.cpu()
adj = sparse_mx_to_torch_sparse_tensor(adj).float().cpu()
features_GCN = copy.deepcopy(features)
features_GCN = torch.FloatTensor(features_GCN).cpu()
```

Figure 4: *Compute influence quantity and normalized adjacency matrix*

After normalizing the adjacency matrix I compute influence quantity for all the nodes. The model-free GCN is $X^{(k+1)} = \hat{D}^{-1}\hat{A}X^{(k)}$. For k = 1 it's $X^{(2)} = \hat{D}^{-1}\hat{A}X^{(1)}$ or $X^{(2)} = (\hat{D}^{-1}\hat{A})^2X^{(0)}$. So the influence quantity as introduced in Equation 1 can implement in this situation. The influence quantity is $(\hat{D}^{-1}\hat{A})^2$, which means just normalized adjacency matrix power 2. In forth line of Figure 4 this operation executed.

```
similarity_feature = np.ones((num_node,num_node)) #Similarity Matrix between nodes
for i in range(num_node-1):
    for j in range(i+1,num_node):
        similarity_feature[i][j] = compute_cos_sim(aax_feature[i],aax_feature[j])
        similarity_feature[j][i] = similarity_feature[i][j] #Symetric
dis_range = np.max(similarity_feature) - np.min(similarity_feature)
similarity_feature = (similarity_feature - np.min(similarity_feature))/dis_range #Normilized Similar
```

Figure 5: *Compute similarity matrix*

After computing embedded feature vectors in model-free GCN, I compute a similarity matrix. This similarity matrix components are computed by cosine similarity. This matrix elements will be used as s in Equation 3 in the next parts of the code. Similarity matrix computing is executed by code brought in Figure 5.

Before going to the node selection loop, First I should introduce some functions.

In Figure 6 I bring needed code for Equation 3. The argument is cosine similarity between two nodes. Oracle accuracy and a number of classes are constant for all nodes.

```
def get_reliable_score(similarity):
    return (oracle_acc*similarity)/(oracle_acc*similarity+(1-oracle_acc)*(1-similarity)/(num_class-1))
```

Figure 6: *Equation 3 implementation*

```
def get_activated_node_dense(node,reliable_score,activated_node):
    activated_vector=((adj_matrix2[node]*reliable_score)>th)+0
    #activated_vector=((adj_matrix2[node])>th)+0 #Just Using labeld influence not quality
    activated_vector=activated_vector*activated_node
    count=num_ones.dot(activated_vector)
    return count,activated_vector
```

Figure 7: *Activated nodes count*

In Figure 7 I wrote a function that accepts a specific node's index and its reliability score then outputs associated activated node index and count by labeling this node. As I said before influence quantity is $(AD^{-1})^2$. If the reliability score is multiplied by this influence quantity, the function uses influences quality for node selection, If just uses influence quantity there is no quality consideration for node selection.

```
def get_max_reliable_info_node_dense(idx_used,high_score_nodes,activated_node,train_class,labels):
"""
arguments:
idx_used <-- index of selected nodes for labeling in the previous batches
high_score_nodes <-- index of available nodes for labeling (candidate)
activated_node <-- activated_node (first iteration is ones vactor
                                   then use output of update_reliability)
train_class <-- a dictionary, in keys we have classes and in values
                                   list of labeled nodes in previous batches
labels <-- list labels
"""
    max_ral_node = 0
    max_activated_node = 0
    max_activated_num = 0

    for node in high_score_nodes:
        reliable_score = oracle_acc

        activated_num,activated_node_tmp = get_activated_node_dense(node,reliable_score,
                                                                    activated_node)
        if activated_num > max_activated_num:
            max_activated_num = activated_num
            max_ral_node = node
            max_activated_node = activated_node_tmp

    return max_ral_node,max_activated_node,max_activated_num
```

Figure 8: *Best node selection function*

In Figure 8 the most important function is brought. This function's arguments are the index of labeled nodes and the available node's index for labeling. Selecting the best node among available nodes for labeling is the functionality of this written code. The description for each argument is brought in the function description. The function outputs index of selected node and number and index of activated nodes by labeling this specific node.

```
def update_reliability(idx_used,train_class,labels,num_node):
    """
    Arguments:
    idx_used <-- index of nodes that labeled in previous batches
    train_class <-- a dictionary, in keys we have classe and in
                values list of labeled nodes in previous batches
    labels <-- label of nodes in the hand of oracle (noise added)
    num_node <-- Number of nodes

    """

    activated_node = np.zeros(num_node)
    for node in idx_used:
        reliable_score = 0
        node_label = labels[node].item()
        #-------------This Section Implement Eq 3 and 4 --------------
        if node_label in train_class:
            total_score = 0.0          #training based on their label.
            for tmp_node in train_class[node_label]:
                total_score+=reliability_list[tmp_node] #Total Score: Reliablity
                                #score of all same labled node (dominator of Eq4)
            for tmp_node in train_class[node_label]:
                reliable_score+=reliability_list[tmp_node]
                        *get_reliable_score(similarity_feature[node][tmp_node])
        #-------------------------------------------------------------
        else:
            reliable_score = oracle_acc

        reliability_list[node]=reliable_score #reliabily_list is list contain
                            #reliability score (quality) of each labaled node
        activated_node+=((adj_matrix2[node]*reliable_score)>th)+0 #Eq11
        #activated_vector=((adj_matrix2[node])>th)+0 #Just Using labeld influence not quality
    return np.ones(num_node)-((activated_node>0)+0)
```

Figure 9: *Function for update reliability score of nodes*

The main functionality of the function brought in Figure 9 is updating the reliability score for each node based on currently labeled nodes. This update changes values in reliability-list. Also, this function updates activated nodes by currently labeled nodes and call once in each batch.

```
while True:
    max_ral_node,max_activated_node,max_activated_num = get_max_reliable_info_node_dense(idx_train
                    ,idx_avaliable_temp,activated_node,train_class,labels)
    idx_train.append(max_ral_node) #add selected node for training
    idx_avaliable.remove(max_ral_node) #Remove Selected node form candidate
    idx_avaliable_temp.remove(max_ral_node) #Remove Selected node form candidate
    node_label = labels[max_ral_node].item() #Laibaling selected node
    #-------------------------------------
    if node_label in train_class:
        train_class[node_label].append(max_ral_node)
    else:
        train_class[node_label]=list()
        train_class[node_label].append(max_ral_node)

    count += 1 #Flag of iteration

    if count%batch_size == 0:
        activated_node = update_reliability(idx_train,train_class,labels,num_node)

    activated_node = activated_node - max_activated_node



    if count >= num_coreset or max_activated_num <= 0:
        """
        Stop iteration if we run out of budget
        """
        break
```

Figure 10: *Node Selection Loop*

In Figure 10 I bring written code for node selection phase. This code is a loop that breaks when the budget runs out or candidates can not active any nodes. This loop is executed B (budget) times. In each iteration, one node is selected for labeling. The functions that I introduced before are used here. In each batch, once the node's reliability scores for labeled train nodes are updated (estimate). This code is final step in node selection phase.

2.3.2 training and prediction

After the node selection phase, our output is approximately best subset of training nodes which selected for labeling based on both quality and quantity influence measurement. So our problem summarized to a semi-supervised classification by these labeled nodes.

In this script, I use a two-layer GCN for the classification task. Code of the model's class is shown in Figure 11. In this class two layer GCN (the Convolotional layer taken from graphConvolution script) with relu activation function and dropout are defined.

```
class GCN(nn.Module):
    def __init__(self, nfeat, nhid, nclass, dropout):
        super(GCN, self).__init__()

        self.gc1 = GraphConvolution(nfeat, nhid,bias=True)
        self.gc2 = GraphConvolution(nhid, nclass,bias=True)
        self.dropout = dropout

    def forward(self, x, adj):
        x = F.dropout(x, self.dropout, training=self.training)
        x = F.relu(self.gc1(x, adj))
        x = F.dropout(x, self.dropout, training=self.training)
        x = self.gc2(x, adj)
        return x
```

Figure 11: *Two layer GCN class*

In Figure 12 model and optimizer object Instantiated and in the for loop training execute.

```
model = GCN(nfeat=features_GCN.shape[1],
        nhid=hidden_size,
        nclass=labels.max().item() + 1,
        dropout=0.85)
model.cpu()
optimizer = optim.Adam(model.parameters(),
                       lr=0.05, weight_decay=5e-4)

for epoch in range(400):
    train(epoch,model,record)
```

Figure 12: *Training structure*

```
def train(epoch, model,record):

    model.train()
    optimizer.zero_grad()
    output = model(features_GCN, adj)
    output_ = F.softmax(output,dim=1)
    one_hot_labels = F.one_hot(labels, num_classes=num_class)
    weight_one_hot_labels = torch.mul(one_hot_labels,reliability_list)#Used Reliabilit
    loss_train = my_cross_entropy(output_[idx_train],weight_one_hot_labels[idx_train])
    acc_train = accuracy(output[idx_train], labels[idx_train])
    loss_train.backward()
    optimizer.step()
    model.eval()
    output = model(features_GCN, adj)
    loss_val = F.nll_loss(output[idx_val], labels[idx_val])
    acc_val = accuracy(output[idx_val], labels[idx_val])
    loss_test = F.nll_loss(output[idx_test], labels[idx_test])
    acc_test = accuracy(output[idx_test], labels[idx_test])
    record[acc_val.item()] = acc_test.item()
```

Figure 13: *train function*

For the curious readers, I also bring train function in Figure 13. In train function training is executed in a transductive semi-supervised classification. As I mentioned before the loss function weighted cross-entropy that is weighed by the reliability score of train nodes.

Also in the first lines of RIM-GCN, there are some hyperparameters like oracle accuracy and budget.

2.4 RIM-LP.py

RIM-LP is very similar to RIM-GCN. They are different in three things, First, RIM-LP influence quality and quantity is based on label vector instead of the feature vector. Second, due to this fact, we don't have access to labels at first (before any node selection batch execution) so I can't compute the similarity matrix at first as I did in RIM-GCN so I compute the similarity matrix once in each batch during node selection, in other words, I update similarity matrix in every batch by labeling new labels. The last and obvious difference is, I use LP for prediction (classification) rather than GCN.

In the next subsections, I highlight part of RIM-LP that are different from RIM-GCN.To avoid duplication, I prefer not to mention similar part to RIM-GCN.

2.4.1 Node Selection

In Figure 14 the code for computing influence quantity is brought. The influence quantity computation is independent of label vectors, so I can compute it before any node labeling. As before, in the first line adjacency

```
adj = aug_normalized_adjacency(adj)
adj_matrix = torch.FloatTensor(adj.todense()).cpu()
adj_matrix2 = torch.mm(adj_matrix,adj_matrix).cpu()
adj_matrix2 = np.array(adj_matrix2.cpu())
one_hot_label = F.one_hot(torch.Tensor(labels).to(torch.int64), num_classes=num_class).float()
adj = sparse_mx_to_torch_sparse_tensor(adj).float().cpu()
```

Figure 14: *Influence quantity of label vector*

matrix normalize. And influence quality as before is $(\hat{D}^{-1}\hat{A})^2$. The LP model is $Y^{(k+1)} = \hat{D}^{-1}\hat{A}Y^{(k)}$, for k=1, $Y^{(2)} = \hat{D}^{-1}\hat{A}Y^{(1)}$ or $Y^{(2)} = (\hat{D}^{-1}\hat{A})^2Y^{(0)}$. The derivation of $Y^{(2)} = (\hat{D}^{-1}\hat{A})^2Y^{(0)}$ respect to the $Y^{(0)}$ is $(\hat{D}^{-1}\hat{A})^2$ as I said before. In fifth line of Figure 14 code, labels list converted to one-hot label vector.

```
while True:

    max_ral_node,max_activated_node,max_activated_num = get_max_reliable_info_node_dense(idx_train,
                            idx_avaliable_temp,activated_node,train_class,labels)
    idx_train.append(max_ral_node) #add selected node for training
    idx_avaliable.remove(max_ral_node) #Remove Selected node form candidate
    idx_avaliable_temp.remove(max_ral_node) #Remove Selected node form candidate
    node_label = labels[max_ral_node].item() #Laibaling selected node

    if node_label in train_class:
        train_class[node_label].append(max_ral_node)
    else:
        train_class[node_label]=list()
        train_class[node_label].append(max_ral_node)

    count += 1 #Flag of iteration

    if count%batch_size == 0:

        one_hot_node_label_org = copy.deepcopy(one_hot_label)
        one_hot_node_label = copy.deepcopy(one_hot_label)
        one_hot_node_label[idx_avaliable] = torch.zeros(num_class)
        one_hot_node_label[idx_train] = torch.zeros(num_class)
        one_hot_node_label[idx_test] = torch.zeros(num_class)

        for _ in range(2):
            one_hot_node_label = torch.mm(adj_matrix,one_hot_node_label)
            one_hot_node_label[idx_train] = one_hot_node_label_org[idx_train]

        aax_label = np.array(one_hot_node_label.cpu())

        similarity_label = np.ones((num_node,num_node)) #Similarity Matrix between nodes

        for i in range(num_node-1):
            for j in range(i+1,num_node):
                similarity_label[i][j] = compute_cos_sim(aax_label[i],aax_label[j])
                similarity_label[j][i] = similarity_label[i][j]
        dis_range = np.max(similarity_label) - np.min(similarity_label)
        similarity_label = (similarity_label - np.min(similarity_label))/dis_range

        activated_node = update_reliability(idx_train,train_class,labels,num_node)

    activated_node = activated_node - max_activated_node

    if count >= num_coreset or max_activated_num <= 0:
        break
```

Figure 15: *Node selection phase in LP*

In Figure 15 node selection code for LP is brought. This code is the same as GCN except in one part. Look at the 'if' (if count%batch-size == 0) condition that checks for batch changing. This 'if' is True once in each batch. In the 'if' I compute the similarity matrix as I said before. By careful look can infer that this computing happened after at least one batch because without any labeled nodes computing similarity matrix is meaningless.

At the end of the node selection phase, there is a subset of labeled nodes in training that maximize both influence quality and quantity.

2.4.2 prediction

With an approximately best subset of training labeled nodes, it's obvious afterward is classifying by a semi-supervised method. By an LP model, I classified validation and test node. The code for this task is brought in Figure 16. LP does not need any training.

```
for epoch in range(50):
    predict_label = torch.mm(adj_matrix,predict_label)
    predict_label[idx_train] = one_hot_label[idx_train].clone()


    acc_val = accuracy(predict_label[idx_val], labels[idx_val])
    acc_test = accuracy(predict_label[idx_test], labels[idx_test])
    record[acc_val.item()] = acc_test.item()
```

Figure 16: *LP prediction*

The code brought in Figure 16 is a software implementation of Equation below:

$$Y^{(k+1)} = \hat{D}^{-1}\hat{A}Y^{(k)}, \quad Y_u^{(k+1)} = Y^{(k+1)}, \quad Y_l^{(k+1)} = Y^{(0)} \tag{11}$$

3 Re-Simulation Results

After implementing the Algorithm, it's time for results. First of all, I introduce the datasets. Also describing the splitting distribution is essential for making a good understanding for readers about the problem.

3.1 Datasets

I used two datasets Cora and Citeseer, and didn't use PubMed due to memory shortage. Both Cora and Citeseer are academic paper citation networks. Nodes are bags-of-words of associated papers and edges indicate which papers are cited each other. An overview of datasets are brought in the Table bellow, also splinting setting is reported.

Table 1: *Overview of the Two Datasets*

Dataset	#Nodes	#Features	#Edges	#Classes	#Train/Val/Test	Task type	Description
Cora	2,708	1,433	5,429	7	1,208/500/1,000	Transductive	citation network
Citeseer	3,327	3,703	4,732	6	1,827/500/1,000	Transductive	citation network

3.2 Main Results

In this section, I reported the main results to compare to Table 1 of the original paper. In Table 2 the oracle accuracy for both datasets is 0.7 which means error of labeling is 0.3. The budget size for Cora is 140 and for Citeseer is 120. Also in this table, the result of baselines are brought from the original paper.

In the table, the "Model" is classifier algorithm and the "Methods" is the subset selector algorithm.

Table 2: *The test accuracy (%) on different datasets when labeling accuracy is 0.7*

Model	Methods	Cora	Citeseer
GCN	Random	65.6	56.3
	AGE+	72.5	61.1
	ANRMAB+	72.4	63.4
	GPA+	72.8	63.8
	RIM Paper results	77.9	67.5
	RIM My results	**74.9**	**66.6**
LP	Random	51.7	31.4
	LP-ME+	55.7	35
	LP-MRE+	59.1	41.4
	RIM Paper results	62.4	46.7
	RIM My results	**65.2**	**49.8**

3.3 Influence of labels' budget

I study performance of GCN and LP Model under different labeling budgets. The results are reported in Figure 17 for GCN and Figure 18 for LP. It's clear by increasing the budget size accuracy increases.

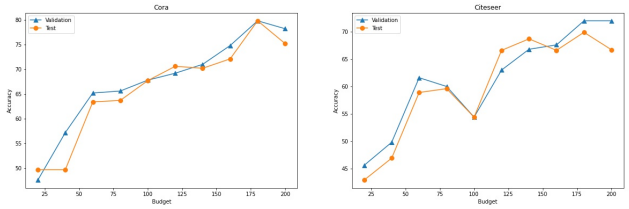

Figure 17: *The test and validation accuracy of GCN model along with different labeling budget when labeling error rate is 0.3*

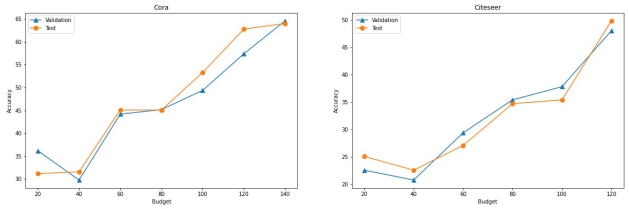

Figure 18: *The test and validation accuracy of LP model along with different labeling budget when labeling error rate is 0.3*

3.4 Influence of oracle accuracy

I study performance of GCN and LP Model under different oracle accuracy (1 - labeling errors). The results for GCN and LP are reported in Figure 19 and Figure 20 respectively. It's interesting how much labeling accuracy is important.

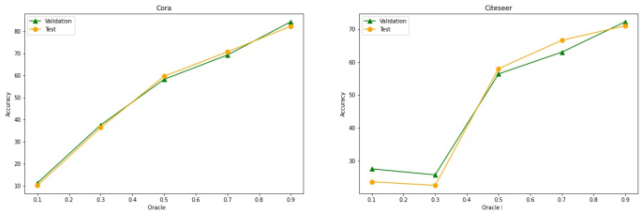

Figure 19: *The test accuracy with different labeling error rate of labeled nodes for GCN*

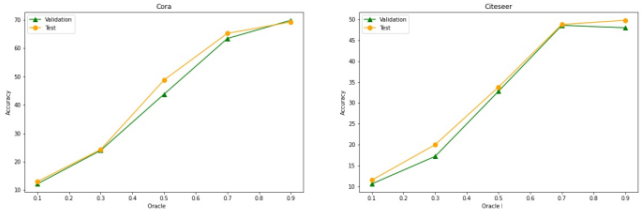

Figure 20: *The test accuracy with different labeling error rate of labeled nodes for LP*

3.5 Influence of quality

I evaluate RIM on GCN while disabling one component at a time when the labeling accuracy is 0.7 and budget size is 140,120 for Cora and Citeseer receptively. I evaluate RIM: (i) without the label reliability score served as the loss weight (called No Reliable Training (RT)); (ii) without the label reliability when selecting the node (called No Reliable Selection (RS));(iii) without both reliable component(called No Reliable Training and Selection (RTS)). Table 3 displays the results of these three settings.

Table 3: *The influence of different components in RIM*

Methods	Cora	△	Citeseer	△
No RT	74.3	-0.5	60.3	-5.1
No RS	63.0	-11.9	59.2	-6
No RTS	64.0	-10.9	60.6	-5.4
RIM My results	**74.9**		**65.2**	

Conclusion

In this paper as the "statistical learning theory" course term-paper, first of all, I tried to give an overview of RIM concepts and ideas. The main purpose of this paper is to indicate my efforts in simulation (programming, testing,...) and understanding the paper. I showed my deep understanding of different aspects of the paper by giving a detailed explanation of the source code, then I reported my results.

Bibliography

Zhang, Wentao et al. (2021). “RIM: Reliable Influence-based Active Learning on Graphs”. In: *Advances in Neural Information Processing Systems* 34.