CS224W - Colab 1 2020

May 24, 2022

1 CS224W - Colab 1

In this Colab, we will write a full pipeline for **learning node embeddings**. We will go through the following 3 steps.

To start, we will load a classic graph in network science, the Karate Club Network. We will explore multiple graph statistics for that graph.

We will then work together to transform the graph structure into a PyTorch tensor, so that we can perform machine learning over the graph.

Finally, we will finish the first learning algorithm on graphs: a node embedding model. For simplicity, our model here is simpler than DeepWalk / node2vec algorithms taught in the lecture. But it's still rewarding and challenging, as we will write it from scratch via PyTorch.

Now let's get started!

Note: Make sure to **sequentially run all the cells**, so that the intermediate variables / packages will carry over to the next cell

2 1 Graph Basics

To start, we will load a classic graph in network science, the Karate Club Network. We will explore multiple graph statistics for that graph.

2.1 Setup

We will heavily use NetworkX in this Colab.

[1]:
```
import networkx as nx
```

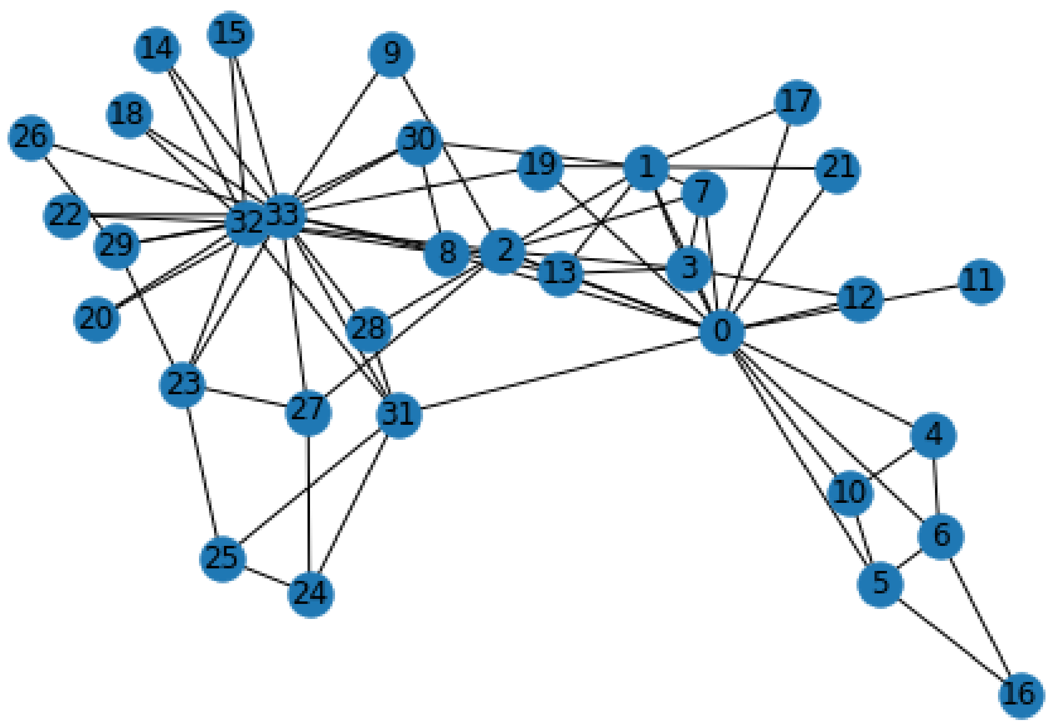
2.2 Zachary's karate club network

The Karate Club Network is a graph describes a social network of 34 members of a karate club and documents links between members who interacted outside the club.

[2]:
```
G = nx.karate_club_graph()

# G is an undirected graph
type(G)
```

[2]: networkx.classes.graph.Graph

[3]:
```
# Visualize the graph
nx.draw(G, with_labels = True)
```



2.3 Question 1: What is the average degree of the karate club network? (5 Points)

[4]:
```
def average_degree(num_edges, num_nodes):
  # TODO: Implement this function that takes number of edges
  # and number of nodes, and returns the average node degree of
  # the graph. Round the result to nearest integer (for example
  # 3.3 will be rounded to 3 and 3.7 will be rounded to 4)

  avg_degree = 0

  ############# Your code here ############
  avg_degree = round(num_edges/num_nodes)

  #########################################

  return avg_degree
```

```
num_edges = G.number_of_edges()
num_nodes = G.number_of_nodes()
avg_degree = average_degree(num_edges, num_nodes)
print("num_edges: {}".format(num_edges))
print("num_nodes: {}".format(num_nodes))
print("Average degree of karate club network is {}".format(avg_degree))
```

```
num_edges: 78
num_nodes: 34
Average degree of karate club network is 2
```

2.4 Question 2: What is the average clustering coefficient of the karate club network? (5 Points)

[5]:
```
def average_clustering_coefficient(G):
  # TODO: Implement this function that takes a nx.Graph
  # and returns the average clustering coefficient. Round
  # the result to 2 decimal places (for example 3.333 will
  # be rounded to 3.33 and 3.7571 will be rounded to 3.76)

  avg_cluster_coef = 0

  ############# Your code here ############
  ## Note:
  ## 1: Please use the appropriate NetworkX clustering function
  avg_cluster_coef = round(nx.average_clustering(G),2)
  #########################################

  return avg_cluster_coef

avg_cluster_coef = average_clustering_coefficient(G)
print("Average clustering coefficient of karate club network is {}".
 ↪format(avg_cluster_coef))
```

```
Average clustering coefficient of karate club network is 0.57
```

2.5 Question 3: What is the PageRank value for node 0 (node with id 0) after one PageRank iteration? (5 Points)

Please complete the code block by implementing the PageRank equation: $r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$

[6]:
```
node_id = 0
print("Node {} has degree {}".format(node_id, G.degree[node_id]))
for neighbour in G.neighbors(node_id):
  print("Neighbour {} has {} neighbours".format(neighbour,G.degree[neighbour]))
#print("Node {} has {} neighbours".format(node_id, G.degree[node_id]))
```

```
#print("Pagerank for node {} is {}".format(node_id,nx.pagerank(G)))
# print("Node {} has {} out_edges and {} in_edges".format(node_id,G.
↪in_edges(node_id),G.in_edges(node_id)))
```

```
Node 0 has degree 16
Neighbour 1 has 9 neighbours
Neighbour 2 has 10 neighbours
Neighbour 3 has 6 neighbours
Neighbour 4 has 3 neighbours
Neighbour 5 has 4 neighbours
Neighbour 6 has 4 neighbours
Neighbour 7 has 4 neighbours
Neighbour 8 has 5 neighbours
Neighbour 10 has 3 neighbours
Neighbour 11 has 1 neighbours
Neighbour 12 has 2 neighbours
Neighbour 13 has 5 neighbours
Neighbour 17 has 2 neighbours
Neighbour 19 has 3 neighbours
Neighbour 21 has 2 neighbours
Neighbour 31 has 6 neighbours
```

[7]:

```
def one_iter_pagerank(G, beta, r0, node_id):
  # TODO: Implement this function that takes a nx.Graph, beta, r0 and node id.
  # The return value r1 is one interation PageRank value for the input node.
  # Please round r1 to 2 decimal places.

  r1 = 0

  ############# Your code here ############
  ## Note:
  ## 1: You should not use nx.pagerank
  num_nodes = G.number_of_nodes()

  r1 = (1-beta)/num_nodes

  for neighbour in G.neighbors(node_id):
    r1 = r1 + beta*(r0/G.degree[neighbour])

  r1=round(r1,2)
  #########################################

  return r1

beta = 0.8
r0 = 1 / G.number_of_nodes()
node = 0
```

```
r1 = one_iter_pagerank(G, beta, r0, node)
print("The PageRank value for node 0 after one iteration is {}".format(r1))
```

```
The PageRank value for node 0 after one iteration is 0.13
```

2.6 Question 4: What is the (raw) closeness centrality for the karate club network node 5? (5 Points)

The equation for closeness centrality is $c(v) = \frac{1}{\sum_{u \neq v} \text{shortest path length between } u \text{ and } v}$

[8]:
```
def closeness_centrality(G, node=5):
  # TODO: Implement the function that calculates closeness centrality
  # for a node in karate club network. G is the input karate club
  # network and node is the node id in the graph. Please round the
  # closeness centrality result to 2 decimal places.

  closeness = 0

  ## Note:
  ## 1: You can use networkx closeness centrality function.
  ## 2: Notice that networkx closeness centrality returns the normalized
  ## closeness directly, which is different from the raw (unnormalized)
  ## one that we learned in the lecture.
  closeness = nx.closeness_centrality(G)[node]
  closeness = round(closeness,2)
  #########################################

  return closeness

node = 5
closeness = closeness_centrality(G, node=node)
print("The karate club network has closeness centrality {}".format(closeness))
```

```
The karate club network has closeness centrality 0.38
```

[9]:
```
nx.closeness_centrality(G)[5]
```

[9]: 0.38372093023255816

3 2 Graph to Tensor

We will then work together to transform the graph G into a PyTorch tensor, so that we can perform machine learning over the graph.

3.1 Setup

Check if PyTorch is properly installed

[10]:
```python
import torch
print(torch.__version__)
```

```
1.10.2+cu102
```

3.2 PyTorch tensor basics

We can generate PyTorch tensor with all zeros, ones or random values.

[11]:
```python
# Generate 3 x 4 tensor with all ones
ones = torch.ones(3, 4)
print(ones)

# Generate 3 x 4 tensor with all zeros
zeros = torch.zeros(3, 4)
print(zeros)

# Generate 3 x 4 tensor with random values on the interval [0, 1)
random_tensor = torch.rand(3, 4)
print(random_tensor)

# Get the shape of the tensor
print(ones.shape)
```

```
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
tensor([[0.5719, 0.7805, 0.5706, 0.4343],
        [0.9959, 0.6000, 0.4333, 0.3053],
        [0.5905, 0.6266, 0.7656, 0.1364]])
torch.Size([3, 4])
```

PyTorch tensor contains elements for a single data type, the `dtype`.

[12]:
```python
# Create a 3 x 4 tensor with all 32-bit floating point zeros
zeros = torch.zeros(3, 4, dtype=torch.float32)
print(zeros.dtype)

# Change the tensor dtype to 64-bit integer
zeros = zeros.type(torch.long)
print(zeros.dtype)
```

```
torch.float32
torch.int64
```

3.3 Question 5: Getting the edge list of the karate club network and transform it into torch.LongTensor. What is the torch.sum value of pos_edge_index tensor? (10 Points)

[13]:
```
def graph_to_edge_list(G):
  # TODO: Implement the function that returns the edge list of
  # an nx.Graph. The returned edge_list should be a list of tuples
  # where each tuple is a tuple representing an edge connected
  # by two nodes.

  edge_list = []

  ############# Your code here ############
  for edge in G.edges():
    edge_list.append(edge)
  #########################################

  return edge_list

def edge_list_to_tensor(edge_list):
  # TODO: Implement the function that transforms the edge_list to
  # tensor. The input edge_list is a list of tuples and the resulting
  # tensor should have the shape [2 x len(edge_list)].

  edge_index = torch.tensor([])

  ############# Your code here ############
  edge_index = torch.tensor(edge_list)
  edge_index = torch.transpose(edge_index,0,1)
  #########################################

  return edge_index

pos_edge_list = graph_to_edge_list(G)
pos_edge_index = edge_list_to_tensor(pos_edge_list)
print("The pos_edge_index tensor has shape {}".format(pos_edge_index.shape))
print("The pos_edge_index tensor has sum value {}".format(torch.
 ↪sum(pos_edge_index)))
```

```
The pos_edge_index tensor has shape torch.Size([2, 78])
The pos_edge_index tensor has sum value 2535
```

[14]:
```
pos_edge_index = edge_list_to_tensor(pos_edge_list)
print("The pos_edge_index tensor has shape {}".format(pos_edge_index.shape))

pos_edge_index
#pos_edge_list = graph_to_edge_list(G)
```

```
#pos_edge_list
```

```
The pos_edge_index tensor has shape torch.Size([2, 78])
```

[14]:
```
tensor([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,
          1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  4,
          4,  5,  5,  5,  6,  8,  8,  8,  9, 13, 14, 14, 15, 15, 18, 18, 19, 20,
         20, 22, 22, 23, 23, 23, 23, 23, 24, 24, 24, 25, 26, 26, 27, 28, 28, 29,
         29, 30, 30, 31, 31, 32],
        [ 1,  2,  3,  4,  5,  6,  7,  8, 10, 11, 12, 13, 17, 19, 21, 31,  2,  3,
          7, 13, 17, 19, 21, 30,  3,  7,  8,  9, 13, 27, 28, 32,  7, 12, 13,  6,
         10,  6, 10, 16, 16, 30, 32, 33, 33, 33, 32, 33, 32, 33, 32, 33, 33, 32,
         33, 32, 33, 25, 27, 29, 32, 33, 25, 27, 31, 31, 29, 33, 33, 31, 33, 32,
         33, 32, 33, 32, 33, 33]])
```

3.4 Question 6: Please implement following function that samples negative edges. Then you will answer which edges (edge_1 to edge_5) can be negative ones in the karate club network? (10 Points)

[15]:
```
import random
num_nodes=10
list1=[]



def generate_edge(number_nodes):
  node1=random.randrange(0,number_nodes)
  node2=random.randrange(node1,number_nodes) # node 2 is always bigger than␣
 ↪node 1
  if node2 is node1:
    node2=random.randrange(node1,number_nodes) # node 2 is always bigger than␣
 ↪node 1
  edge = (node1,node2)

  return edge

counter=0
while counter < num_nodes:

    this_edge = generate_edge(num_nodes)

 #   print("list1: {}".format(list1))
 #   print("this edge: {}".format(this_edge))
    #nodes=num_nodes

    while this_edge in list1:
      this_edge = generate_edge(num_nodes)
```

```
    list1.append(this_edge)
    counter+=1

print(list1)
```

```
[(3, 7), (9, 9), (0, 5), (2, 9), (5, 5), (4, 9), (0, 9), (0, 8), (1, 8), (6, 8)]
```

[16]:
```
list2 = [(9, 2), (0, 9), (5, 1), (2, 6), (4, 5), (8, 4), (2, 7), (0, 2), (9,␣
 ↪0), (9, 2)]

item=(9, 0) # check code will do as expected
if item in list2:
  print("{} is in the list".format(item))
#no_duplicate_list = list(dict.fromkeys(list2))
#print(no_duplicate_list)
```

```
(9, 0) is in the list
```

[17]:
```
import random

def sample_negative_edges(G, num_neg_samples):
  # TODO: Implement the function that returns a list of negative edges.
  # The number of sampled negative edges is num_neg_samples. You do not
  # need to consider the corner case when the number of possible negative edges
  # is less than num_neg_samples. It should be ok as long as your␣
 ↪implementation
  # works on the karate club network. In this implementation, self loop should
  # not be considered as either a positive or negative edge. Also, notice that
  # the karate club network is an undirected graph, if (0, 1) is a positive
  # edge, do you think (1, 0) can be a negative one?

  neg_edge_list = []

  ############# Your code here ############
  list1=[]

  def generate_edge(number_nodes):
    node1=random.randrange(0,number_nodes)
    node2=random.randrange(node1,number_nodes) # node 2 is always bigger than␣
 ↪node 1
    while node2 is node1:
      node1=random.randrange(0,number_nodes)
      node2=random.randrange(node1,number_nodes) # node 2 is always bigger than␣
 ↪node 1
    edge = (node1,node2)

    return edge
```

```
  counter=0
  while counter < num_neg_samples:
    num_nodes = G.number_of_nodes()
    this_edge = generate_edge(num_nodes)

    #print("list1: {}".format(list1))
    #print("this edge: {}".format(this_edge))
    #nodes=num_nodes

    while this_edge in list1:
      this_edge = generate_edge(num_nodes)

    list1.append(this_edge)
    counter+=1

  neg_edge_list=list1
  #########################################

  return neg_edge_list

# Sample 78 negative edges
neg_edge_list = sample_negative_edges(G, len(pos_edge_list))

# Transform the negative edge list to tensor
neg_edge_index = edge_list_to_tensor(neg_edge_list)
print("The neg_edge_index tensor has shape {}".format(neg_edge_index.shape))

# Which of following edges can be negative ones?
edge_1 = (7, 1)
edge_2 = (1, 33)
edge_3 = (33, 22)
edge_4 = (0, 4)
edge_5 = (4, 2)

############ Your code here ###########
## Note:
## 1: For each of the 5 edges, print whether it can be negative edge
# neg_edge_list.append((7,1)) ## for testing negative edge function check is␣
↪working
print(neg_edge_list)
print("Number of unique edges: {}".format(len(set(neg_edge_list))))
edges_given = [edge_1, edge_2, edge_3, edge_4, edge_5]
for edge in edges_given:
  print("Evaluating edge {}".format(edge))
  if edge in neg_edge_list:
    print("{} is a negative edge".format(edge))
```

```
############################################
```

```
The neg_edge_index tensor has shape torch.Size([2, 78])
[(2, 13), (3, 16), (4, 30), (9, 14), (18, 26), (10, 25), (19, 30), (15, 29),
(20, 25), (20, 31), (17, 26), (15, 30), (15, 21), (14, 26), (24, 29), (14, 23),
(15, 24), (2, 5), (27, 29), (2, 30), (3, 31), (23, 26), (27, 32), (8, 17), (29,
32), (11, 27), (20, 33), (23, 31), (1, 7), (8, 12), (11, 30), (8, 11), (13, 18),
(17, 28), (1, 32), (9, 22), (28, 33), (20, 32), (26, 33), (29, 31), (13, 23),
(23, 24), (12, 18), (1, 14), (8, 28), (2, 19), (24, 32), (15, 25), (11, 13),
(32, 33), (22, 24), (19, 27), (28, 29), (6, 28), (25, 31), (4, 22), (5, 10),
(30, 31), (3, 27), (14, 25), (10, 29), (1, 8), (23, 33), (23, 28), (6, 26), (9,
31), (16, 31), (5, 31), (25, 32), (8, 14), (20, 30), (12, 20), (10, 19), (11,
31), (1, 5), (0, 31), (29, 33), (23, 30)]
Number of unique edges: 78
Evaluating edge (7, 1)
Evaluating edge (1, 33)
Evaluating edge (33, 22)
Evaluating edge (0, 4)
Evaluating edge (4, 2)
```

Answer: None of the 5 edges are negative edges

4 3 Node Emebedding Learning

Finally, we will finish the first learning algorithm on graphs: a node embedding model.

4.1 Setup

[18]:
```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

print(torch.__version__)
```

```
1.10.2+cu102
```

To write our own node embedding learning methods, we'll heavily use the `nn.Embedding` module in PyTorch. Let's see how to use `nn.Embedding`:

[19]:
```
# Initialize an embedding layer
# Suppose we want to have embedding for 4 items (e.g., nodes)
# Each item is represented with 8 dimensional vector

emb_sample = nn.Embedding(num_embeddings=4, embedding_dim=8)
print('Sample embedding layer: {}'.format(emb_sample))
```

```
Sample embedding layer: Embedding(4, 8)
```

We can select items from the embedding matrix, by using Tensor indices

[20]:
```python
# Select an embedding in emb_sample
id = torch.LongTensor([1])
print(emb_sample(id))

# Select multiple embeddings
ids = torch.LongTensor([1, 3])
print(emb_sample(ids))

# Get the shape of the embedding weight matrix
shape = emb_sample.weight.data.shape
print(shape)

# Overwrite the weight to tensor with all ones
emb_sample.weight.data = torch.ones(shape)

# Let's check if the emb is indeed initilized
ids = torch.LongTensor([0, 3])
print(emb_sample(ids))
```

```
tensor([[-1.7170,  0.2966,  0.0219,  1.5743,  0.3510,  1.7183, -2.7950,
-0.9427]],
       grad_fn=<EmbeddingBackward0>)
tensor([[-1.7170,  0.2966,  0.0219,  1.5743,  0.3510,  1.7183, -2.7950,
-0.9427],
        [ 0.3910,  0.1231, -1.8800, -0.2816,  0.5195,  1.0568,  0.0414,
0.7390]],
       grad_fn=<EmbeddingBackward0>)
torch.Size([4, 8])
tensor([[1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1.]], grad_fn=<EmbeddingBackward0>)
```

[21]:
```python
id = torch.LongTensor([1])
print(emb_sample(id))
print(id)
ids = torch.LongTensor([0, 3])
print(ids)

emb_sample = nn.Embedding(num_embeddings=4, embedding_dim=8)
print(emb_sample(torch.LongTensor([2])))
```

```
tensor([[1., 1., 1., 1., 1., 1., 1., 1.]], grad_fn=<EmbeddingBackward0>)
tensor([1])
tensor([0, 3])
tensor([[ 0.3164, -0.1295,  0.4061,  1.4841,  0.2454, -1.3059, -1.3507,
-0.0216]],
       grad_fn=<EmbeddingBackward0>)
```

Now, it's your time to create node embedding matrix for the graph we have! - We want to have **16 dimensional** vector for each node in the karate club network. - We want to initalize the matrix under **uniform distribution**, in the range of $[0, 1)$. We suggest you using `torch.rand`.

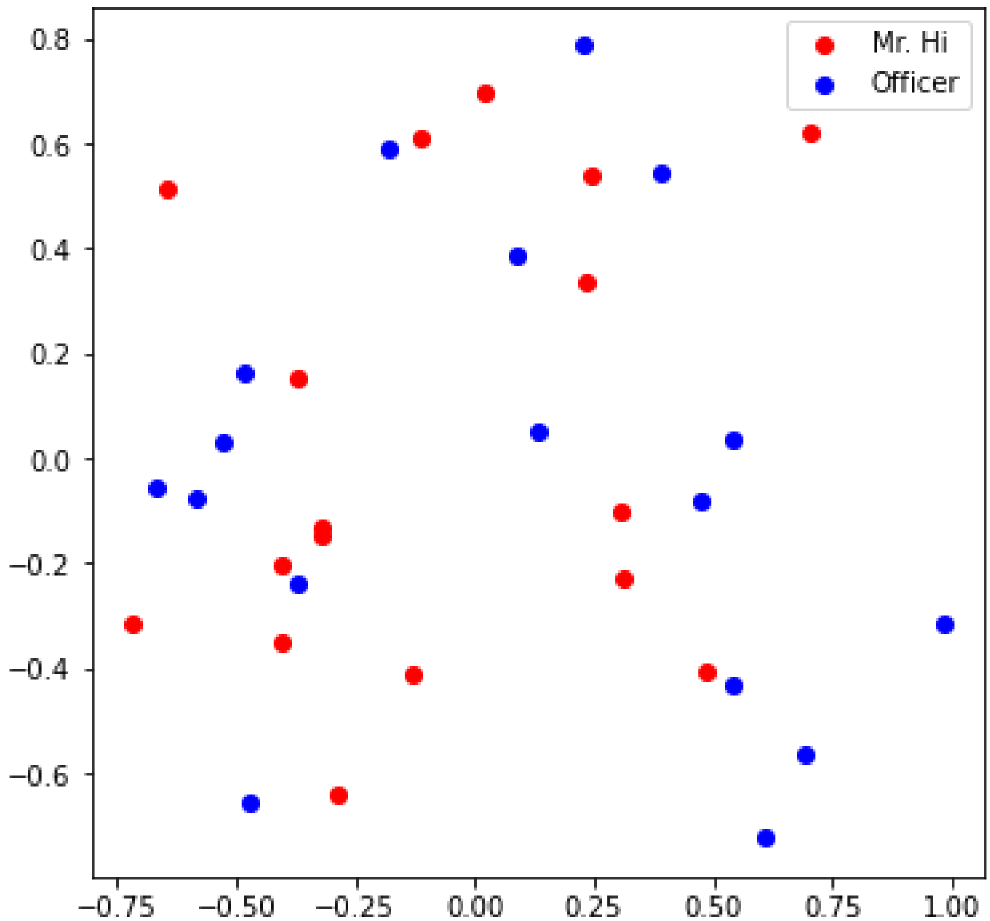
[22]:
```python
# Please do not change / reset the random seed
torch.manual_seed(1)

def create_node_emb(num_node=34, embedding_dim=16):
  # TODO: Implement this function that will create the node embedding matrix.
  # A torch.nn.Embedding layer will be returned. You do not need to change
  # the values of num_node and embedding_dim. The weight matrix of returned
  # layer should be initialized under uniform distribution.

  emb = None

  ############# Your code here ############
  emb = nn.Embedding(num_node, embedding_dim)
  shape = emb.weight.data.shape
  emb.weight.data = torch.rand(shape)

  #########################################

  return emb

emb = create_node_emb()
ids = torch.LongTensor([0, 3])
ids_1 = torch.LongTensor([1, 3])
ids_2 = torch.LongTensor([2, 3])

# Print the embedding layer
print("Embedding: {}".format(emb))

# An example that gets the embeddings for node 0 and 3
print(emb(ids))
print(emb(torch.LongTensor([0])))
#print(emb(ids_1))
#print(emb(ids_2))
```

```
Embedding: Embedding(34, 16)
tensor([[0.2114, 0.7335, 0.1433, 0.9647, 0.2933, 0.7951, 0.5170, 0.2801, 0.8339,
         0.1185, 0.2355, 0.5599, 0.8966, 0.2858, 0.1955, 0.1808],
        [0.7486, 0.6546, 0.3843, 0.9820, 0.6012, 0.3710, 0.4929, 0.9915, 0.8358,
         0.4629, 0.9902, 0.7196, 0.2338, 0.0450, 0.7906, 0.9689]],
       grad_fn=<EmbeddingBackward0>)
tensor([[0.2114, 0.7335, 0.1433, 0.9647, 0.2933, 0.7951, 0.5170, 0.2801, 0.8339,
         0.1185, 0.2355, 0.5599, 0.8966, 0.2858, 0.1955, 0.1808]],
       grad_fn=<EmbeddingBackward0>)
```

4.2 Visualize the initial node embeddings

One good way to understand an embedding matrix, is to visualize it in a 2D space. Here, we have implemented an embedding visualization function for you. We first do PCA to reduce the dimensionality of embeddings to a 2D space. Then visualize each point, colored by the community it belongs to.

[23]:
```python
def visualize_emb(emb):
  X = emb.weight.data.numpy()
  pca = PCA(n_components=2)
  components = pca.fit_transform(X)
  plt.figure(figsize=(6, 6))
  club1_x = []
  club1_y = []
  club2_x = []
  club2_y = []
  for node in G.nodes(data=True):
    if node[1]['club'] == 'Mr. Hi':
      club1_x.append(components[node[0]][0])
      club1_y.append(components[node[0]][1])
    else:
      club2_x.append(components[node[0]][0])
      club2_y.append(components[node[0]][1])
  plt.scatter(club1_x, club1_y, color="red", label="Mr. Hi")
  plt.scatter(club2_x, club2_y, color="blue", label="Officer")
  plt.legend()
  plt.show()

# Visualize the initial random embeddding
visualize_emb(emb)
```



4.3 Question 7: Training the embedding! What is the best performance you can get? Please report both the best loss and accuracy on Gradescope. (20 Points)

[24]:

```
from torch.optim import SGD

def accuracy(pred, label):
  # TODO: Implement the accuracy function. This function takes the
  # pred tensor (the resulting tensor after sigmoid) and the label
  # tensor (torch.LongTensor). Predicted value greater than 0.5 will
  # be classified as label 1. Else it will be classified as label 0.
  # The returned accuracy should be rounded to 4 decimal places.
  # For example, accuracy 0.82956 will be rounded to 0.8296.

  accu = 0.0

  ############# Your code here ############
```

```
  # convert prediction to class prediction
  #pred_class = pred.clone()
  #pred_class[pred>0.5] = 1
  #pred_class[pred<=0.5] = 0
  pred_class = torch.where(pred>0.5,1,0)
#  print("pred_class: {}".format(pred_class))
  # find number of correct predictions
  correct_pred = torch.eq(pred_class,label)
  correct_predictions = float(torch.sum(correct_pred))
#  print("number of correct predictions: {}".format(correct_predictions))
  num_labels = torch.numel(label)*1
#  print("number of predictions: {}".format(num_labels))

  accu = round(correct_predictions/num_labels,4)
  #accu = round(78/156,4)

  #########################################

  return accu

def train(emb, loss_fn, sigmoid, train_label, train_edge):
  # TODO: Train the embedding layer here. You can also change epochs and
  # learning rate. In general, you need to implement:
  # (1) Get the embeddings of the nodes in train_edge
  # (2) Dot product the embeddings between each node pair
  # (3) Feed the dot product result into sigmoid
  # (4) Feed the sigmoid output into the loss_fn
  # (5) Print both loss and accuracy of each epoch
  # (as a sanity check, the loss should decrease during training)

  epochs = 500
  learning_rate = 0.1

  optimizer = SGD(emb.parameters(), lr=learning_rate, momentum=0.9)

  for i in range(epochs):

    ############# Your code here ############
    # 0. zero parameter gradients
    optimizer.zero_grad()

    # 1. get embeddings of nodes in train_edge
    #train_num_nodes = train_edge.dim()
    #num_nodes = train_edge.size(dim=1)
    #num_nodes = torch.max(train_edge)
    #embs = create_node_emb(num_nodes)
```

```
    # 1b. create list of embeddings
    id1 = torch.LongTensor(train_edge[0])
    id2 = torch.LongTensor(train_edge[1])

    embs_1 = emb(id1)
    embs_2 = emb(id2)

    # 2. dot product of embedding node pairs
    #dot_product = torch.dot(embs_1,embs_2)
##    print("embs_1: dimensions: {} \n{}".format(embs_1.shape,embs_1))
##    print("embs_2: dimensions: {} \n{}".format(embs_2.shape,embs_2))

    product = torch.mul(embs_1,embs_2)
    dot_product = product.sum(dim=1)
    #dot_product = torch.matmul(embs_1,embs_2.T)
    #dot_product = torch.mm(embs_1,embs_2)

##    print("dot_product: dimensions: {} \n{}".format(dot_product.
 ↪shape,dot_product))

    # 3. feed dot product result into a sigmoid
    predictions = sigmoid(dot_product)
##    print("predictions: {}".format(predictions))
    # 4. Feed the sigmoid output into the loss_fn

    loss = loss_fn(predictions,train_label)
    accu = accuracy(predictions, train_label)

    loss.backward()   # derive gradients
    optimizer.step()  # update parameters based on gradients

    # 5. Print both loss and accuracy of each epoch
    print("Epoch {} --- loss:{}  accuracy:{}".format(i, loss, accu))
#    print("".format(loss))
#    print("".format(accu))

    ########################################

loss_fn = nn.BCELoss()
sigmoid = nn.Sigmoid()

# Generate the positive and negative labels
pos_label = torch.ones(pos_edge_index.shape[1], )
neg_label = torch.zeros(neg_edge_index.shape[1], )

# Concat positive and negative labels into one tensor
train_label = torch.cat([pos_label, neg_label], dim=0)
```

```python
# Concat positive and negative edges into one tensor
# Since the network is very small, we do not split the edges into val/test sets
train_edge = torch.cat([pos_edge_index, neg_edge_index], dim=1)

train(emb, loss_fn, sigmoid, train_label, train_edge)
```

```
/home/arch/anaconda3/lib/python3.8/site-packages/torch/autograd/__init__.py:154:
UserWarning: CUDA initialization: CUDA unknown error - this may be due to an
incorrectly set up environment, e.g. changing env variable CUDA_VISIBLE_DEVICES
after program start. Setting the available devices to be zero. (Triggered
internally at  ../c10/cuda/CUDAFunctions.cpp:112.)
  Variable._execution_engine.run_backward(
```

```
Epoch 0 --- loss:2.0070252418518066  accuracy:0.5
Epoch 1 --- loss:1.9928604364395142  accuracy:0.5
Epoch 2 --- loss:1.9661840200424194  accuracy:0.5
Epoch 3 --- loss:1.928676724433899  accuracy:0.5
Epoch 4 --- loss:1.8820090293884277  accuracy:0.5
Epoch 5 --- loss:1.827805757522583  accuracy:0.5
Epoch 6 --- loss:1.7676297426223755  accuracy:0.5
Epoch 7 --- loss:1.7029595375061035  accuracy:0.5
Epoch 8 --- loss:1.6351782083511353  accuracy:0.5
Epoch 9 --- loss:1.5655635595321655  accuracy:0.5
Epoch 10 --- loss:1.4952772855758667  accuracy:0.5
Epoch 11 --- loss:1.4253580570220947  accuracy:0.5
Epoch 12 --- loss:1.3567143678665161  accuracy:0.5
Epoch 13 --- loss:1.290120005607605  accuracy:0.5
Epoch 14 --- loss:1.2262096405029297  accuracy:0.5
Epoch 15 --- loss:1.1654787063598633  accuracy:0.5
Epoch 16 --- loss:1.1082876920700073  accuracy:0.5
Epoch 17 --- loss:1.054869532585144  accuracy:0.5
Epoch 18 --- loss:1.0053397417068481  accuracy:0.5
Epoch 19 --- loss:0.9597120881080627  accuracy:0.5064
Epoch 20 --- loss:0.9179134964942932  accuracy:0.5128
Epoch 21 --- loss:0.879802405834198  accuracy:0.5128
Epoch 22 --- loss:0.845185399055481  accuracy:0.5128
Epoch 23 --- loss:0.8138340711593628  accuracy:0.5128
Epoch 24 --- loss:0.7854991555213928  accuracy:0.5128
Epoch 25 --- loss:0.7599229216575623  accuracy:0.5256
Epoch 26 --- loss:0.7368482947349548  accuracy:0.5449
Epoch 27 --- loss:0.7160273194313049  accuracy:0.5513
Epoch 28 --- loss:0.6972247362136841  accuracy:0.5641
Epoch 29 --- loss:0.6802225112915039  accuracy:0.5705
Epoch 30 --- loss:0.6648202538490295  accuracy:0.5705
Epoch 31 --- loss:0.6508365273475647  accuracy:0.5897
Epoch 32 --- loss:0.6381081342697144  accuracy:0.6026
Epoch 33 --- loss:0.6264894604682922  accuracy:0.6154
```

```
Epoch 34 --- loss:0.6158508658409119  accuracy:0.6474
Epoch 35 --- loss:0.6060777902603149  accuracy:0.6667
Epoch 36 --- loss:0.597069263458252  accuracy:0.6795
Epoch 37 --- loss:0.5887362360954285  accuracy:0.6795
Epoch 38 --- loss:0.5810003876686096  accuracy:0.6923
Epoch 39 --- loss:0.5737928748130798  accuracy:0.6923
Epoch 40 --- loss:0.5670533776283264  accuracy:0.6987
Epoch 41 --- loss:0.5607286691665649  accuracy:0.7051
Epoch 42 --- loss:0.5547724366188049  accuracy:0.7179
Epoch 43 --- loss:0.5491436123847961  accuracy:0.7244
Epoch 44 --- loss:0.5438061952590942  accuracy:0.7244
Epoch 45 --- loss:0.538728654384613  accuracy:0.7244
Epoch 46 --- loss:0.5338830351829529  accuracy:0.7244
Epoch 47 --- loss:0.5292448401451111  accuracy:0.7372
Epoch 48 --- loss:0.5247924327850342  accuracy:0.7436
Epoch 49 --- loss:0.520506739616394  accuracy:0.7436
Epoch 50 --- loss:0.5163707733154297  accuracy:0.7436
Epoch 51 --- loss:0.5123697519302368  accuracy:0.7436
Epoch 52 --- loss:0.5084903240203857  accuracy:0.75
Epoch 53 --- loss:0.5047208666801453  accuracy:0.75
Epoch 54 --- loss:0.5010510087013245  accuracy:0.7564
Epoch 55 --- loss:0.4974713921546936  accuracy:0.7564
Epoch 56 --- loss:0.4939739406108856  accuracy:0.7628
Epoch 57 --- loss:0.49055129289627075  accuracy:0.7628
Epoch 58 --- loss:0.4871968626976013  accuracy:0.7628
Epoch 59 --- loss:0.4839048981666565  accuracy:0.7692
Epoch 60 --- loss:0.480670303106308  accuracy:0.7692
Epoch 61 --- loss:0.477488249540329  accuracy:0.7692
Epoch 62 --- loss:0.47435468435287476  accuracy:0.7756
Epoch 63 --- loss:0.47126585245132446  accuracy:0.7949
Epoch 64 --- loss:0.46821844577789307  accuracy:0.8077
Epoch 65 --- loss:0.46520936489105225  accuracy:0.8141
Epoch 66 --- loss:0.4622359573841095  accuracy:0.8141
Epoch 67 --- loss:0.4592958092689514  accuracy:0.8141
Epoch 68 --- loss:0.45638665556907654  accuracy:0.8205
Epoch 69 --- loss:0.4535066783428192  accuracy:0.8205
Epoch 70 --- loss:0.4506538510322571  accuracy:0.8269
Epoch 71 --- loss:0.4478267729282379  accuracy:0.8269
Epoch 72 --- loss:0.44502389430999756  accuracy:0.8333
Epoch 73 --- loss:0.4422440230846405  accuracy:0.8333
Epoch 74 --- loss:0.43948569893836975  accuracy:0.8333
Epoch 75 --- loss:0.4367481768131256  accuracy:0.8397
Epoch 76 --- loss:0.4340302348136902  accuracy:0.8397
Epoch 77 --- loss:0.43133115768432617  accuracy:0.8462
Epoch 78 --- loss:0.4286501407623291  accuracy:0.8526
Epoch 79 --- loss:0.4259864091873169  accuracy:0.8526
Epoch 80 --- loss:0.4233393967151642  accuracy:0.859
Epoch 81 --- loss:0.4207085967063904  accuracy:0.859
```

```
Epoch 82 --- loss:0.41809332370758057  accuracy:0.859
Epoch 83 --- loss:0.4154932498931885  accuracy:0.8654
Epoch 84 --- loss:0.4129081070423126  accuracy:0.8718
Epoch 85 --- loss:0.41033729910850525  accuracy:0.8718
Epoch 86 --- loss:0.4077807068824768  accuracy:0.8718
Epoch 87 --- loss:0.40523800253868103  accuracy:0.8718
Epoch 88 --- loss:0.4027090072631836  accuracy:0.8782
Epoch 89 --- loss:0.40019354224205017  accuracy:0.8782
Epoch 90 --- loss:0.3976913392543793  accuracy:0.8782
Epoch 91 --- loss:0.3952024281024933  accuracy:0.8782
Epoch 92 --- loss:0.3927266597747803  accuracy:0.8782
Epoch 93 --- loss:0.39026400446891785  accuracy:0.8782
Epoch 94 --- loss:0.38781434297561646  accuracy:0.8782
Epoch 95 --- loss:0.38537776470184326  accuracy:0.8782
Epoch 96 --- loss:0.3829541802406311  accuracy:0.8782
Epoch 97 --- loss:0.38054367899894714  accuracy:0.8846
Epoch 98 --- loss:0.3781462609767914  accuracy:0.8846
Epoch 99 --- loss:0.375762015581131  accuracy:0.891
Epoch 100 --- loss:0.3733910322189331  accuracy:0.891
Epoch 101 --- loss:0.37103334069252014  accuracy:0.891
Epoch 102 --- loss:0.36868906021118164  accuracy:0.891
Epoch 103 --- loss:0.36635836958885193  accuracy:0.891
Epoch 104 --- loss:0.3640412986278534  accuracy:0.891
Epoch 105 --- loss:0.36173805594444275  accuracy:0.8974
Epoch 106 --- loss:0.35944864153862  accuracy:0.8974
Epoch 107 --- loss:0.357173353433609  accuracy:0.8974
Epoch 108 --- loss:0.35491225123405457  accuracy:0.8974
Epoch 109 --- loss:0.3526654839515686  accuracy:0.8974
Epoch 110 --- loss:0.35043323040008545  accuracy:0.8974
Epoch 111 --- loss:0.34821563959121704  accuracy:0.8974
Epoch 112 --- loss:0.3460128605365753  accuracy:0.8974
Epoch 113 --- loss:0.3438250720500946  accuracy:0.8974
Epoch 114 --- loss:0.34165239334106445  accuracy:0.8974
Epoch 115 --- loss:0.3394949734210968  accuracy:0.8974
Epoch 116 --- loss:0.337352991104126  accuracy:0.8974
Epoch 117 --- loss:0.3352265954017639  accuracy:0.8974
Epoch 118 --- loss:0.33311590552330017  accuracy:0.8974
Epoch 119 --- loss:0.3310210406780243  accuracy:0.8974
Epoch 120 --- loss:0.328942209482193  accuracy:0.8974
Epoch 121 --- loss:0.3268794119358063  accuracy:0.8974
Epoch 122 --- loss:0.324832946062088  accuracy:0.9038
Epoch 123 --- loss:0.32280275225639343  accuracy:0.9038
Epoch 124 --- loss:0.320789098739624  accuracy:0.9103
Epoch 125 --- loss:0.3187919855117798  accuracy:0.9103
Epoch 126 --- loss:0.3168115019798279  accuracy:0.9103
Epoch 127 --- loss:0.31484782695770264  accuracy:0.9103
Epoch 128 --- loss:0.31290093064308167  accuracy:0.9103
Epoch 129 --- loss:0.310971021652217  accuracy:0.9103
```

```
Epoch 130 --- loss:0.3090580105781555  accuracy:0.9103
Epoch 131 --- loss:0.3071620762348175  accuracy:0.9103
Epoch 132 --- loss:0.30528321862220764  accuracy:0.9167
Epoch 133 --- loss:0.3034214675426483  accuracy:0.9167
Epoch 134 --- loss:0.3015768527984619  accuracy:0.9167
Epoch 135 --- loss:0.2997494339942932  accuracy:0.9167
Epoch 136 --- loss:0.2979392111301422  accuracy:0.9167
Epoch 137 --- loss:0.2961461842060089  accuracy:0.9167
Epoch 138 --- loss:0.2943703532218933  accuracy:0.9167
Epoch 139 --- loss:0.2926117777824402  accuracy:0.9167
Epoch 140 --- loss:0.29087033867836  accuracy:0.9167
Epoch 141 --- loss:0.2891460359096527  accuracy:0.9167
Epoch 142 --- loss:0.28743889927864075  accuracy:0.9167
Epoch 143 --- loss:0.28574883937835693  accuracy:0.9167
Epoch 144 --- loss:0.28407585620880127  accuracy:0.9167
Epoch 145 --- loss:0.28241991996765137  accuracy:0.9167
Epoch 146 --- loss:0.2807808816432953  accuracy:0.9167
Epoch 147 --- loss:0.27915874123573303  accuracy:0.9167
Epoch 148 --- loss:0.2775534689426422  accuracy:0.9167
Epoch 149 --- loss:0.2759649455547333  accuracy:0.9167
Epoch 150 --- loss:0.2743930220603943  accuracy:0.9167
Epoch 151 --- loss:0.27283769845962524  accuracy:0.9167
Epoch 152 --- loss:0.27129894495010376  accuracy:0.9167
Epoch 153 --- loss:0.2697765529155731  accuracy:0.9167
Epoch 154 --- loss:0.2682705223560333  accuracy:0.9167
Epoch 155 --- loss:0.26678067445755005  accuracy:0.9167
Epoch 156 --- loss:0.2653069496154785  accuracy:0.9167
Epoch 157 --- loss:0.26384925842285156  accuracy:0.9167
Epoch 158 --- loss:0.2624073922634125  accuracy:0.9167
Epoch 159 --- loss:0.26098138093948364  accuracy:0.9167
Epoch 160 --- loss:0.25957101583480835  accuracy:0.9167
Epoch 161 --- loss:0.25817617774009705  accuracy:0.9167
Epoch 162 --- loss:0.25679677724838257  accuracy:0.9167
Epoch 163 --- loss:0.255432665348053  accuracy:0.9167
Epoch 164 --- loss:0.2540837228298187  accuracy:0.9167
Epoch 165 --- loss:0.25274986028671265  accuracy:0.9167
Epoch 166 --- loss:0.2514308989048004  accuracy:0.9167
Epoch 167 --- loss:0.25012674927711487  accuracy:0.9167
Epoch 168 --- loss:0.2488372027873993  accuracy:0.9167
Epoch 169 --- loss:0.2475622445344925  accuracy:0.9167
Epoch 170 --- loss:0.24630165100097656  accuracy:0.9167
Epoch 171 --- loss:0.24505534768104553  accuracy:0.9167
Epoch 172 --- loss:0.2438231259584427  accuracy:0.9167
Epoch 173 --- loss:0.24260489642620087  accuracy:0.9167
Epoch 174 --- loss:0.2414005696773529  accuracy:0.9167
Epoch 175 --- loss:0.2402098923921585  accuracy:0.9167
Epoch 176 --- loss:0.23903289437294006  accuracy:0.9167
Epoch 177 --- loss:0.2378692626953125  accuracy:0.9167
```

```
Epoch 178 --- loss:0.23671898245811462  accuracy:0.9167
Epoch 179 --- loss:0.23558185994625092  accuracy:0.9167
Epoch 180 --- loss:0.23445777595043182  accuracy:0.9167
Epoch 181 --- loss:0.2333465963602066  accuracy:0.9167
Epoch 182 --- loss:0.2322482019662857  accuracy:0.9167
Epoch 183 --- loss:0.2311624437570572  accuracy:0.9167
Epoch 184 --- loss:0.23008915781974792  accuracy:0.9167
Epoch 185 --- loss:0.22902828454971313  accuracy:0.9167
Epoch 186 --- loss:0.22797958552837372  accuracy:0.9167
Epoch 187 --- loss:0.22694304585456848  accuracy:0.9167
Epoch 188 --- loss:0.2259184569120407  accuracy:0.9167
Epoch 189 --- loss:0.22490569949150085  accuracy:0.9167
Epoch 190 --- loss:0.22390463948249817  accuracy:0.9167
Epoch 191 --- loss:0.2229151576757431  accuracy:0.9167
Epoch 192 --- loss:0.2219371497631073  accuracy:0.9167
Epoch 193 --- loss:0.22097046673297882  accuracy:0.9167
Epoch 194 --- loss:0.22001495957374573  accuracy:0.9167
Epoch 195 --- loss:0.21907053887844086  accuracy:0.9167
Epoch 196 --- loss:0.21813705563545227  accuracy:0.9167
Epoch 197 --- loss:0.21721439063549042  accuracy:0.9167
Epoch 198 --- loss:0.21630237996578217  accuracy:0.9167
Epoch 199 --- loss:0.21540100872516632  accuracy:0.9167
Epoch 200 --- loss:0.21451006829738617  accuracy:0.9167
Epoch 201 --- loss:0.21362945437431335  accuracy:0.9167
Epoch 202 --- loss:0.21275904774665833  accuracy:0.9167
Epoch 203 --- loss:0.21189869940280914  accuracy:0.9167
Epoch 204 --- loss:0.21104836463928223  accuracy:0.9167
Epoch 205 --- loss:0.21020789444446564  accuracy:0.9167
Epoch 206 --- loss:0.20937713980674744  accuracy:0.9167
Epoch 207 --- loss:0.20855602622032166  accuracy:0.9167
Epoch 208 --- loss:0.20774440467357635  accuracy:0.9167
Epoch 209 --- loss:0.20694221556186676  accuracy:0.9167
Epoch 210 --- loss:0.20614929497241974  accuracy:0.9167
Epoch 211 --- loss:0.20536555349826813  accuracy:0.9167
Epoch 212 --- loss:0.20459087193012238  accuracy:0.9167
Epoch 213 --- loss:0.20382516086101532  accuracy:0.9167
Epoch 214 --- loss:0.20306828618049622  accuracy:0.9167
Epoch 215 --- loss:0.2023201435804367  accuracy:0.9167
Epoch 216 --- loss:0.20158065855503082  accuracy:0.9167
Epoch 217 --- loss:0.200849711894989  accuracy:0.9167
Epoch 218 --- loss:0.20012718439102173  accuracy:0.9167
Epoch 219 --- loss:0.1994129866361618  accuracy:0.9167
Epoch 220 --- loss:0.19870701432228088  accuracy:0.9167
Epoch 221 --- loss:0.1980091780424118  accuracy:0.9167
Epoch 222 --- loss:0.19731934368610382  accuracy:0.9167
Epoch 223 --- loss:0.19663743674755096  accuracy:0.9167
Epoch 224 --- loss:0.19596336781978607  accuracy:0.9167
Epoch 225 --- loss:0.1952970176935196  accuracy:0.9167
```

```
Epoch 226 --- loss:0.19463831186294556  accuracy:0.9167
Epoch 227 --- loss:0.19398713111877441  accuracy:0.9167
Epoch 228 --- loss:0.19334344565868378  accuracy:0.9167
Epoch 229 --- loss:0.19270706176757812  accuracy:0.9167
Epoch 230 --- loss:0.19207794964313507  accuracy:0.9167
Epoch 231 --- loss:0.19145603477954865  accuracy:0.9167
Epoch 232 --- loss:0.1908411979675293  accuracy:0.9167
Epoch 233 --- loss:0.19023333489894867  accuracy:0.9167
Epoch 234 --- loss:0.189632385969162  accuracy:0.9167
Epoch 235 --- loss:0.1890382617712021  accuracy:0.9167
Epoch 236 --- loss:0.188450887799263  accuracy:0.9167
Epoch 237 --- loss:0.18787012994289398  accuracy:0.9167
Epoch 238 --- loss:0.18729595839977264  accuracy:0.9167
Epoch 239 --- loss:0.18672825396060944  accuracy:0.9167
Epoch 240 --- loss:0.18616695702075958  accuracy:0.9167
Epoch 241 --- loss:0.18561196327209473  accuracy:0.9167
Epoch 242 --- loss:0.18506325781345367  accuracy:0.9167
Epoch 243 --- loss:0.1845206767320633  accuracy:0.9167
Epoch 244 --- loss:0.18398417532444  accuracy:0.9167
Epoch 245 --- loss:0.18345369398593903  accuracy:0.9167
Epoch 246 --- loss:0.1829291433095932  accuracy:0.9167
Epoch 247 --- loss:0.18241041898727417  accuracy:0.9167
Epoch 248 --- loss:0.18189752101898193  accuracy:0.9167
Epoch 249 --- loss:0.18139027059078217  accuracy:0.9167
Epoch 250 --- loss:0.18088869750499725  accuracy:0.9167
Epoch 251 --- loss:0.18039266765117645  accuracy:0.9167
Epoch 252 --- loss:0.17990213632583618  accuracy:0.9167
Epoch 253 --- loss:0.17941702902317047  accuracy:0.9167
Epoch 254 --- loss:0.17893727123737335  accuracy:0.9167
Epoch 255 --- loss:0.17846278846263885  accuracy:0.9167
Epoch 256 --- loss:0.177993506193161  accuracy:0.9167
Epoch 257 --- loss:0.17752937972545624  accuracy:0.9167
Epoch 258 --- loss:0.17707034945487976  accuracy:0.9167
Epoch 259 --- loss:0.1766163408756256  accuracy:0.9167
Epoch 260 --- loss:0.17616726458072662  accuracy:0.9167
Epoch 261 --- loss:0.17572307586669922  accuracy:0.9167
Epoch 262 --- loss:0.175283744931221  accuracy:0.9167
Epoch 263 --- loss:0.17484916746616364  accuracy:0.9167
Epoch 264 --- loss:0.17441928386688232  accuracy:0.9167
Epoch 265 --- loss:0.1739940643310547  accuracy:0.9167
Epoch 266 --- loss:0.17357340455055237  accuracy:0.9167
Epoch 267 --- loss:0.17315728962421417  accuracy:0.9167
Epoch 268 --- loss:0.17274561524391174  accuracy:0.9167
Epoch 269 --- loss:0.17233838140964508  accuracy:0.9167
Epoch 270 --- loss:0.17193551361560822  accuracy:0.9167
Epoch 271 --- loss:0.17153693735599518  accuracy:0.9167
Epoch 272 --- loss:0.171142578125  accuracy:0.9167
Epoch 273 --- loss:0.17075243592262268  accuracy:0.9167
```

```
Epoch 274 --- loss:0.17036642134189606  accuracy:0.9167
Epoch 275 --- loss:0.16998448967933655  accuracy:0.9167
Epoch 276 --- loss:0.16960659623146057  accuracy:0.9167
Epoch 277 --- loss:0.16923268139362335  accuracy:0.9167
Epoch 278 --- loss:0.1688627004623413  accuracy:0.9167
Epoch 279 --- loss:0.16849659383296967  accuracy:0.9167
Epoch 280 --- loss:0.16813431680202484  accuracy:0.9167
Epoch 281 --- loss:0.16777583956718445  accuracy:0.9167
Epoch 282 --- loss:0.16742107272148132  accuracy:0.9167
Epoch 283 --- loss:0.16707001626491547  accuracy:0.9167
Epoch 284 --- loss:0.16672258079051971  accuracy:0.9167
Epoch 285 --- loss:0.16637875139713287  accuracy:0.9167
Epoch 286 --- loss:0.16603846848011017  accuracy:0.9167
Epoch 287 --- loss:0.16570168733596802  accuracy:0.9167
Epoch 288 --- loss:0.16536834836006165  accuracy:0.9167
Epoch 289 --- loss:0.16503845155239105  accuracy:0.9167
Epoch 290 --- loss:0.16471192240715027  accuracy:0.9167
Epoch 291 --- loss:0.16438870131969452  accuracy:0.9167
Epoch 292 --- loss:0.164068803191185  accuracy:0.9167
Epoch 293 --- loss:0.16375213861465454  accuracy:0.9167
Epoch 294 --- loss:0.16343864798545837  accuracy:0.9167
Epoch 295 --- loss:0.16312836110591888  accuracy:0.9167
Epoch 296 --- loss:0.1628211885690689  accuracy:0.9167
Epoch 297 --- loss:0.16251710057258606  accuracy:0.9167
Epoch 298 --- loss:0.16221608221530914  accuracy:0.9167
Epoch 299 --- loss:0.1619180291891098  accuracy:0.9167
Epoch 300 --- loss:0.16162298619747162  accuracy:0.9167
Epoch 301 --- loss:0.16133086383342743  accuracy:0.9167
Epoch 302 --- loss:0.16104164719581604  accuracy:0.9167
Epoch 303 --- loss:0.16075529158115387  accuracy:0.9167
Epoch 304 --- loss:0.16047176718711853  accuracy:0.9167
Epoch 305 --- loss:0.16019102931022644  accuracy:0.9167
Epoch 306 --- loss:0.1599130481481552  accuracy:0.9167
Epoch 307 --- loss:0.15963777899742126  accuracy:0.9167
Epoch 308 --- loss:0.1593652367591858  accuracy:0.9167
Epoch 309 --- loss:0.15909531712532043  accuracy:0.9167
Epoch 310 --- loss:0.1588280200958252  accuracy:0.9167
Epoch 311 --- loss:0.15856333076953888  accuracy:0.9167
Epoch 312 --- loss:0.1583011895418167  accuracy:0.9167
Epoch 313 --- loss:0.15804161131381989  accuracy:0.9167
Epoch 314 --- loss:0.15778449177742004  accuracy:0.9167
Epoch 315 --- loss:0.15752986073493958  accuracy:0.9167
Epoch 316 --- loss:0.1572776734828949  accuracy:0.9167
Epoch 317 --- loss:0.15702787041664124  accuracy:0.9167
Epoch 318 --- loss:0.1567804515361786  accuracy:0.9167
Epoch 319 --- loss:0.15653538703918457  accuracy:0.9167
Epoch 320 --- loss:0.1562926471233368  accuracy:0.9167
Epoch 321 --- loss:0.15605217218399048  accuracy:0.9167
```

```
Epoch 322 --- loss:0.15581399202346802  accuracy:0.9167
Epoch 323 --- loss:0.15557803213596344  accuracy:0.9167
Epoch 324 --- loss:0.15534427762031555  accuracy:0.9167
Epoch 325 --- loss:0.15511274337768555  accuracy:0.9167
Epoch 326 --- loss:0.15488334000110626  accuracy:0.9167
Epoch 327 --- loss:0.15465609729290009  accuracy:0.9167
Epoch 328 --- loss:0.15443091094493866  accuracy:0.9167
Epoch 329 --- loss:0.15420785546302795  accuracy:0.9167
Epoch 330 --- loss:0.1539868265390396  accuracy:0.9167
Epoch 331 --- loss:0.15376783907413483  accuracy:0.9167
Epoch 332 --- loss:0.1535508632659912  accuracy:0.9167
Epoch 333 --- loss:0.15333586931228638  accuracy:0.9167
Epoch 334 --- loss:0.15312279760837555  accuracy:0.9167
Epoch 335 --- loss:0.1529117375612259  accuracy:0.9167
Epoch 336 --- loss:0.15270252525806427  accuracy:0.9167
Epoch 337 --- loss:0.15249525010585785  accuracy:0.9167
Epoch 338 --- loss:0.15228982269763947  accuracy:0.9167
Epoch 339 --- loss:0.1520862579345703  accuracy:0.9167
Epoch 340 --- loss:0.15188449621200562  accuracy:0.9167
Epoch 341 --- loss:0.15168456733226776  accuracy:0.9167
Epoch 342 --- loss:0.15148639678955078  accuracy:0.9167
Epoch 343 --- loss:0.15129001438617706  accuracy:0.9167
Epoch 344 --- loss:0.15109536051750183  accuracy:0.9167
Epoch 345 --- loss:0.15090243518352509  accuracy:0.9167
Epoch 346 --- loss:0.15071119368076324  accuracy:0.9167
Epoch 347 --- loss:0.1505216509103775  accuracy:0.9167
Epoch 348 --- loss:0.15033377707004547  accuracy:0.9167
Epoch 349 --- loss:0.15014754235744476  accuracy:0.9167
Epoch 350 --- loss:0.14996293187141418  accuracy:0.9167
Epoch 351 --- loss:0.14977993071079254  accuracy:0.9167
Epoch 352 --- loss:0.14959852397441864  accuracy:0.9167
Epoch 353 --- loss:0.1494186818599701  accuracy:0.9167
Epoch 354 --- loss:0.1492403894662857  accuracy:0.9167
Epoch 355 --- loss:0.14906364679336548  accuracy:0.9167
Epoch 356 --- loss:0.14888840913772583  accuracy:0.9167
Epoch 357 --- loss:0.14871467649936676  accuracy:0.9167
Epoch 358 --- loss:0.14854243397712708  accuracy:0.9167
Epoch 359 --- loss:0.148371621966362  accuracy:0.9167
Epoch 360 --- loss:0.1482023000717163  accuracy:0.9167
Epoch 361 --- loss:0.14803440868854523  accuracy:0.9167
Epoch 362 --- loss:0.14786791801452637  accuracy:0.9167
Epoch 363 --- loss:0.14770282804965973  accuracy:0.9167
Epoch 364 --- loss:0.1475391387939453  accuracy:0.9167
Epoch 365 --- loss:0.14737685024738312  accuracy:0.9167
Epoch 366 --- loss:0.14721587300300598  accuracy:0.9167
Epoch 367 --- loss:0.1470562368631363  accuracy:0.9167
Epoch 368 --- loss:0.14689794182777405  accuracy:0.9167
Epoch 369 --- loss:0.14674095809459686  accuracy:0.9167
```

```
Epoch 370 --- loss:0.14658525586128235  accuracy:0.9167
Epoch 371 --- loss:0.1464308500289917  accuracy:0.9167
Epoch 372 --- loss:0.14627772569656372  accuracy:0.9167
Epoch 373 --- loss:0.14612582325935364  accuracy:0.9167
Epoch 374 --- loss:0.14597520232200623  accuracy:0.9167
Epoch 375 --- loss:0.14582578837871552  accuracy:0.9167
Epoch 376 --- loss:0.1456775963306427  accuracy:0.9167
Epoch 377 --- loss:0.1455305963754654  accuracy:0.9167
Epoch 378 --- loss:0.1453847885131836  accuracy:0.9167
Epoch 379 --- loss:0.1452401578426361  accuracy:0.9167
Epoch 380 --- loss:0.14509667456150055  accuracy:0.9167
Epoch 381 --- loss:0.1449543684720993  accuracy:0.9167
Epoch 382 --- loss:0.1448131799697876  accuracy:0.9167
Epoch 383 --- loss:0.144673153758049  accuracy:0.9167
Epoch 384 --- loss:0.1445341855287552  accuracy:0.9167
Epoch 385 --- loss:0.1443963497877121  accuracy:0.9167
Epoch 386 --- loss:0.14425961673259735  accuracy:0.9167
Epoch 387 --- loss:0.14412394165992737  accuracy:0.9167
Epoch 388 --- loss:0.14398933947086334  accuracy:0.9167
Epoch 389 --- loss:0.14385581016540527  accuracy:0.9167
Epoch 390 --- loss:0.1437232792377472  accuracy:0.9167
Epoch 391 --- loss:0.14359183609485626  accuracy:0.9167
Epoch 392 --- loss:0.14346139132976532  accuracy:0.9167
Epoch 393 --- loss:0.14333197474479675  accuracy:0.9167
Epoch 394 --- loss:0.1432035267353058  accuracy:0.9167
Epoch 395 --- loss:0.1430761069059372  accuracy:0.9167
Epoch 396 --- loss:0.1429496556520462  accuracy:0.9167
Epoch 397 --- loss:0.14282415807247162  accuracy:0.9167
Epoch 398 --- loss:0.14269964396953583  accuracy:0.9167
Epoch 399 --- loss:0.14257608354091644  accuracy:0.9167
Epoch 400 --- loss:0.14245343208312988  accuracy:0.9167
Epoch 401 --- loss:0.14233176410198212  accuracy:0.9167
Epoch 402 --- loss:0.1422109752893448  accuracy:0.9167
Epoch 403 --- loss:0.14209111034870148  accuracy:0.9167
Epoch 404 --- loss:0.141972154378891  accuracy:0.9167
Epoch 405 --- loss:0.14185409247875214  accuracy:0.9167
Epoch 406 --- loss:0.14173689484596252  accuracy:0.9167
Epoch 407 --- loss:0.14162060618400574  accuracy:0.9167
Epoch 408 --- loss:0.141505166888237  accuracy:0.9167
Epoch 409 --- loss:0.1413905769586563  accuracy:0.9167
Epoch 410 --- loss:0.14127686619758606  accuracy:0.9167
Epoch 411 --- loss:0.14116397500038147  accuracy:0.9167
Epoch 412 --- loss:0.14105193316936493  accuracy:0.9167
Epoch 413 --- loss:0.14094069600105286  accuracy:0.9167
Epoch 414 --- loss:0.14083027839660645  accuracy:0.9167
Epoch 415 --- loss:0.1407206803560257  accuracy:0.9167
Epoch 416 --- loss:0.14061188697814941  accuracy:0.9167
Epoch 417 --- loss:0.1405038833618164  accuracy:0.9167
```

```
Epoch 418 --- loss:0.14039663970470428  accuracy:0.9167
Epoch 419 --- loss:0.14029018580913544  accuracy:0.9167
Epoch 420 --- loss:0.14018452167510986  accuracy:0.9167
Epoch 421 --- loss:0.1400795876979828  accuracy:0.9167
Epoch 422 --- loss:0.139975443482399  accuracy:0.9167
Epoch 423 --- loss:0.13987202942371368  accuracy:0.9167
Epoch 424 --- loss:0.13976933062076569  accuracy:0.9167
Epoch 425 --- loss:0.13966739177703857  accuracy:0.9167
Epoch 426 --- loss:0.13956616818904877  accuracy:0.9167
Epoch 427 --- loss:0.13946568965911865  accuracy:0.9167
Epoch 428 --- loss:0.13936588168144226  accuracy:0.9167
Epoch 429 --- loss:0.13926680386066437  accuracy:0.9167
Epoch 430 --- loss:0.1391684114933014  accuracy:0.9167
Epoch 431 --- loss:0.13907073438167572  accuracy:0.9167
Epoch 432 --- loss:0.13897369801998138  accuracy:0.9167
Epoch 433 --- loss:0.13887737691402435  accuracy:0.9167
Epoch 434 --- loss:0.13878171145915985  accuracy:0.9167
Epoch 435 --- loss:0.13868670165538788  accuracy:0.9167
Epoch 436 --- loss:0.13859236240386963  accuracy:0.9167
Epoch 437 --- loss:0.1384986788034439  accuracy:0.9167
Epoch 438 --- loss:0.13840563595294952  accuracy:0.9167
Epoch 439 --- loss:0.13831323385238647  accuracy:0.9167
Epoch 440 --- loss:0.13822147250175476  accuracy:0.9167
Epoch 441 --- loss:0.13813035190105438  accuracy:0.9167
Epoch 442 --- loss:0.13803981244564056  accuracy:0.9167
Epoch 443 --- loss:0.13794991374015808  accuracy:0.9167
Epoch 444 --- loss:0.13786064088344574  accuracy:0.9167
Epoch 445 --- loss:0.13777194917201996  accuracy:0.9167
Epoch 446 --- loss:0.13768385350704193  accuracy:0.9167
Epoch 447 --- loss:0.13759636878967285  accuracy:0.9167
Epoch 448 --- loss:0.13750946521759033  accuracy:0.9167
Epoch 449 --- loss:0.13742314279079437  accuracy:0.9167
Epoch 450 --- loss:0.13733741641044617  accuracy:0.9167
Epoch 451 --- loss:0.13725225627422333  accuracy:0.9167
Epoch 452 --- loss:0.13716764748096466  accuracy:0.9167
Epoch 453 --- loss:0.13708360493183136  accuracy:0.9167
Epoch 454 --- loss:0.13700011372566223  accuracy:0.9167
Epoch 455 --- loss:0.13691718876361847  accuracy:0.9167
Epoch 456 --- loss:0.13683480024337769  accuracy:0.9167
Epoch 457 --- loss:0.13675297796726227  accuracy:0.9167
Epoch 458 --- loss:0.13667166233062744  accuracy:0.9167
Epoch 459 --- loss:0.1365908980369568  accuracy:0.9167
Epoch 460 --- loss:0.13651065528392792  accuracy:0.9167
Epoch 461 --- loss:0.13643090426921844  accuracy:0.9167
Epoch 462 --- loss:0.13635170459747314  accuracy:0.9167
Epoch 463 --- loss:0.13627301156520844  accuracy:0.9167
Epoch 464 --- loss:0.1361948400735855  accuracy:0.9167
Epoch 465 --- loss:0.13611716032028198  accuracy:0.9167
```

```
Epoch 466 --- loss:0.13603997230529785  accuracy:0.9167
Epoch 467 --- loss:0.1359632909297943  accuracy:0.9167
Epoch 468 --- loss:0.13588711619377136  accuracy:0.9167
Epoch 469 --- loss:0.13581140339374542  accuracy:0.9167
Epoch 470 --- loss:0.13573616743087769  accuracy:0.9167
Epoch 471 --- loss:0.13566142320632935  accuracy:0.9167
Epoch 472 --- loss:0.1355871558189392  accuracy:0.9167
Epoch 473 --- loss:0.1355133354663849  accuracy:0.9167
Epoch 474 --- loss:0.13544002175331116  accuracy:0.9167
Epoch 475 --- loss:0.13536715507507324  accuracy:0.9167
Epoch 476 --- loss:0.13529473543167114  accuracy:0.9167
Epoch 477 --- loss:0.13522276282310486  accuracy:0.9167
Epoch 478 --- loss:0.13515126705169678  accuracy:0.9167
Epoch 479 --- loss:0.13508018851280212  accuracy:0.9167
Epoch 480 --- loss:0.1350095570087433  accuracy:0.9167
Epoch 481 --- loss:0.13493937253952026  accuracy:0.9167
Epoch 482 --- loss:0.13486960530281067  accuracy:0.9167
Epoch 483 --- loss:0.1348002851009369  accuracy:0.9167
Epoch 484 --- loss:0.13473139703273773  accuracy:0.9167
Epoch 485 --- loss:0.1346629410982132  accuracy:0.9167
Epoch 486 --- loss:0.1345948725938797  accuracy:0.9167
Epoch 487 --- loss:0.13452723622322083  accuracy:0.9167
Epoch 488 --- loss:0.13446000218391418  accuracy:0.9167
Epoch 489 --- loss:0.13439320027828217  accuracy:0.9167
Epoch 490 --- loss:0.1343267858028412  accuracy:0.9167
Epoch 491 --- loss:0.13426080346107483  accuracy:0.9167
Epoch 492 --- loss:0.13419517874717712  accuracy:0.9167
Epoch 493 --- loss:0.13412997126579285  accuracy:0.9167
Epoch 494 --- loss:0.13406513631343842  accuracy:0.9167
Epoch 495 --- loss:0.13400070369243622  accuracy:0.9167
Epoch 496 --- loss:0.13393665850162506  accuracy:0.9167
Epoch 497 --- loss:0.13387301564216614  accuracy:0.9167
Epoch 498 --- loss:0.13380973041057587  accuracy:0.9167
Epoch 499 --- loss:0.13374683260917664  accuracy:0.9167
```

Best loss: 0.1337 Best accuracy: 0.9167

4.3.1 Start of Debugging

[25]:
```python
# test dot product on a toy tensor

#train_edge = torch.cat([pos_edge_index, neg_edge_index], dim=1)
print(train_edge.shape)
#print(train_edge.size(dim=1))
print("test[0]:{}".format(test[0]))
print("test[1]:{}".format(test[1]))
product = torch.dot(test[0],test[1])
print("torch.dot: {}".format(product))
```

```
tensor_dot = torch.mul(test[0],test[1])
print("torch.mul: {}".format(tensor_dot))

sigmoid_output = sigmoid(tensor_dot)
print("sigmoid_output: {}".format(sigmoid_output))
print("rounded sigmoid_output: {}".format(round))
#train_label = torch.cat([pos_label, neg_label], dim=0)
#print("train_label: {}".format(train_label))
#loss_fn_output = loss_fn(sigmoid_output)
#print("loss_fn_output: {}".format(loss_fn_output))
```

```
torch.Size([2, 156])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-25-f1c1b20c5c12> in <module>
      4 print(train_edge.shape)
      5 #print(train_edge.size(dim=1))
----> 6 print("test[0]:{}".format(test[0]))
      7 print("test[1]:{}".format(test[1]))
      8 product = torch.dot(test[0],test[1])

NameError: name 'test' is not defined
```

[ ]:
```
#print("embeddings for nodes 0,1,2 & 3 {}".format(emb(torch.
↪LongTensor([0,1,2,3]))))
print("emb.shape {}".format(emb.weight.data.shape))

print(pos_label.shape)
#print(pos_edge_index.shape[1])
#print("pos_edge_index \n{}".format(pos_edge_index))
#print("neg_edge_index \n{}".format(neg_edge_index))
```

[ ]:
```
#print("pos_edge_index: \n{}".format(pos_edge_index))
#print("neg_edge_index: \n{}".format(neg_edge_index))
train_edge = torch.cat([pos_edge_index, neg_edge_index], dim=1)
print("train_edge.shape {}".format(train_edge.shape))
print(train_edge)

print("train_label.shape {}".format(train_label.shape))
```

[ ]:
```
### development functions whilst testing ###

#print(train_edge.size(dim=1))
#max = torch.argmax(train_edge,dim=1)
```

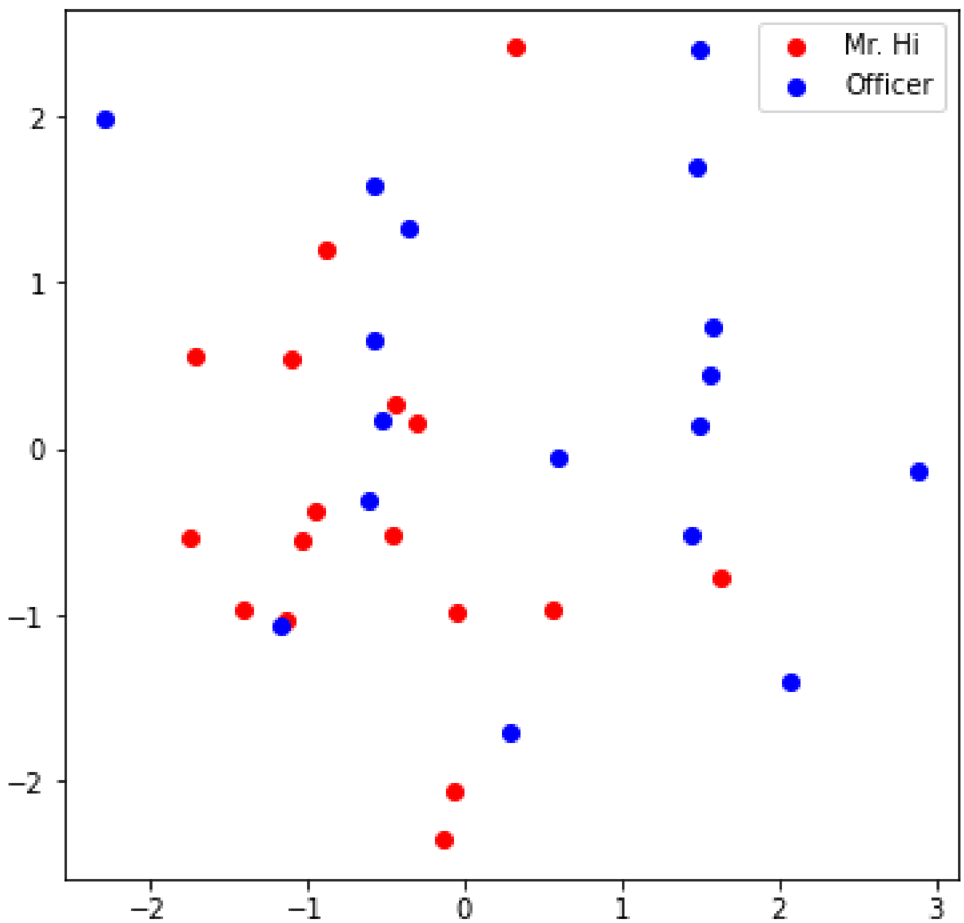
```
#max = torch.max(train_edge,dim=1)
max = torch.max(train_edge)
print("Max index value: {}".format(max))

num_nodes = torch.max(train_edge)+1
embs = create_node_emb(num_nodes)
id1 = torch.LongTensor(train_edge[0])
id2 = torch.LongTensor(train_edge[1])
#print("id1: {}".format(id1))
#print("id2: {}".format(id2))
print("train_edge: {}".format(train_edge))

#print("embs: {}".format(embs))
print("embs(id1).shape is {}".format(embs(id1).shape))
print("train edge embeddings 1: {}".format(embs(id1)))

print("embs(id2).shape is {}".format(embs(id2).shape))
print("train edge embeddings 2: {}".format(embs(id2)))

print("embeddings for nodes 0,1,2 & 3: \n{}".format(embs(torch.
 ↪LongTensor([0,1,2,3]))))
##print("train_edge 2: {}".format(train_edge[0]))
#train_node_nodes = train_edge.size(dim=1)
#print(train_node_nodes)
#print("train_edge.shape".format(train_edge.shape))
#print(pos_edge_index)
#print(pos_edge_list)
```

4.3.2 End of Debugging

4.4 Visualize the final node embeddings

Visualize your final embedding here! You can visually compare the figure with the previous embedding figure. After training, you should oberserve that the two classes are more evidently separated. This is a great sanitity check for your implementation as well.

[26]:
```
# Visualize the final learned embedding
visualize_emb(emb)
```



5 Submission

In order to get credit, you must go submit your answers on Gradescope.