hw 6

Chen Yuanteng

3039725444

1. Debugging DNNs.

(a)

potential reasons:

(1) Overfitting: 56-layer model is more complex and deeper, might be more prone to overfitting the training data, while the smaller 20-layer model may have better generalization capabilities.

(2) Vanishing or exploding gradients:
The deeper network architecture could lead to the problem of vanishing or exploding gradients.

(3). maybe training data is limited, it needs more data to train a model as deep and large as 56-layer model.

To mitigate this problem:

(1). Add regularization and dropout layer. to reduce the risk of overfitting.

(2) Add residual connections. which can aid information to flow in deeper network and alleviate the issue of vanishing or exploding gradient.

(b)

the model with layer normalization will not pass the test.

Because layer normalization computes the mean and var across the spatial dimensions for each individual sample in the batch.

In the provided gradient accumulation algorithm, the model accumulates gradients and updates the parameters every accumulated steps.

However, since layer normalization compute statistics per sample. the accumulated gradients from different samples within the same effective batch would have different mean and var values.

(c). in for loop, optimizer.zero_grad()

should be implemented after optimizer.step().

Otherwise, gradients would be accumulated

during every (inputs, label).

2. Tensor. Rematerialization.

(a) to compute activation of layer-9,
we need to compute activations of 6, 7, 8
first, so it needs 4 fwd in total
So to compute activations of 6, 7, 8, 9.
it needs 4+3+2+1 = 10, same as layer 1-4
∴ 2×10 = 20 fwd in total.

(b). when computing activations of 6, 7, 8, 9.
4 Loadmem are necessary,
to compute 1, 2, 3, 4. another 4 Loadmem
are needed. So 2×4 = 8 Loadmem in total.

(c)
during a single backward:
in tensor rematerialization.
20·2ms + 8·1ms = 48ms.
in storing all activations on this disk:
10·(time of Loaddisk) = 48ms.
time of Loaddisk = 48ms

3. Graph Dynamics

(a). $G_{t+1} = A \cdot G_t \quad G_0 = E$

$\therefore G_K = A^K$ and j-th node in G_K is the j-th row of G_K.

$\therefore$ the output of the j-th node at layer K in this network $\rightarrow A^K_j$

(b). using induction:

$L_0(i,j) = [i=j]$

$L_1(i,j)$ is the definition of matrix A

assume $L_h(i,j) = [A_h]_{i,j} \quad (h \geq 1)$

try to verify $L_{h+1}(i,j) = [A_{h+1}]_{i,j}$

from node i to node j with distance $= h+1$

$\Leftrightarrow$ from node i to node X with distance $= h$ + from node X to node j with an edge.

$\therefore L_{h+1}(i,j) = \sum_{X \in V(j)} L_h(i,X)$

$\therefore L_{h+1}(i,j) = \sum_{x=1}^{n} L_h(i,x) \cdot A_{x,j}$

$\because L_h(i,j) = [A_h]_{i,j}$.

$\therefore L_{h+1}(i,j) = \sum_{x=1}^{n} [A_h]_{i,x} A_{x,j} = [A_{h+1}]_{i,j}$

cc). in the matrix, each row $\Leftrightarrow$ each node

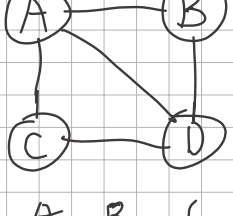
the graph is a set of nodes:

$$V_j = \sum_{i \in V(j)} \cdot V_i$$

cd:. replace sum aggregation with max aggregation:

outputs of node j at layer k $=$

- 1 — there is a path from i to j with length k
- 0 — otherwise



	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

$= A$

$$A^2 = \begin{matrix} 3. & 1 & 1 & 2. \\ 1 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 2 & 1 & 1 & 2 \end{matrix}$$

University of California, Berkeley Fall 2022, EECS 182/282

The power of the graph perspective in clustering

Dependencies

In [15]:
```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"

import numpy as np
import matplotlib.pyplot as plt

from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

from scipy.linalg import svd
from scipy.spatial import distance
from sklearn.preprocessing import normalize

import math
```

Helper Functions

In [16]:
```
def show_data_results(X, num_plot=2, y_pred=None, cmap='jet'):
    if num_plot==1:
        plt.scatter(X[:, 0], X[:, 1])
        plt.title ("input data")
    elif num_plot==2:
        try:
            assert y_pred is not None
            fig = plt.figure(figsize=(10, 3))
            ax1 = fig.add_subplot(121)
            ax1.scatter(X[:, 0], X[:, 1])
            ax1.set(xticks=[],yticks=[],title ="input data")

            ax2 = fig.add_subplot(122)
            ax2.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=cmap)
            ax2.set(xticks=[], yticks=[], title ="clustered data")
        except:
            print('y_pred is required for 2 plots')
```

Q.1. (Given)

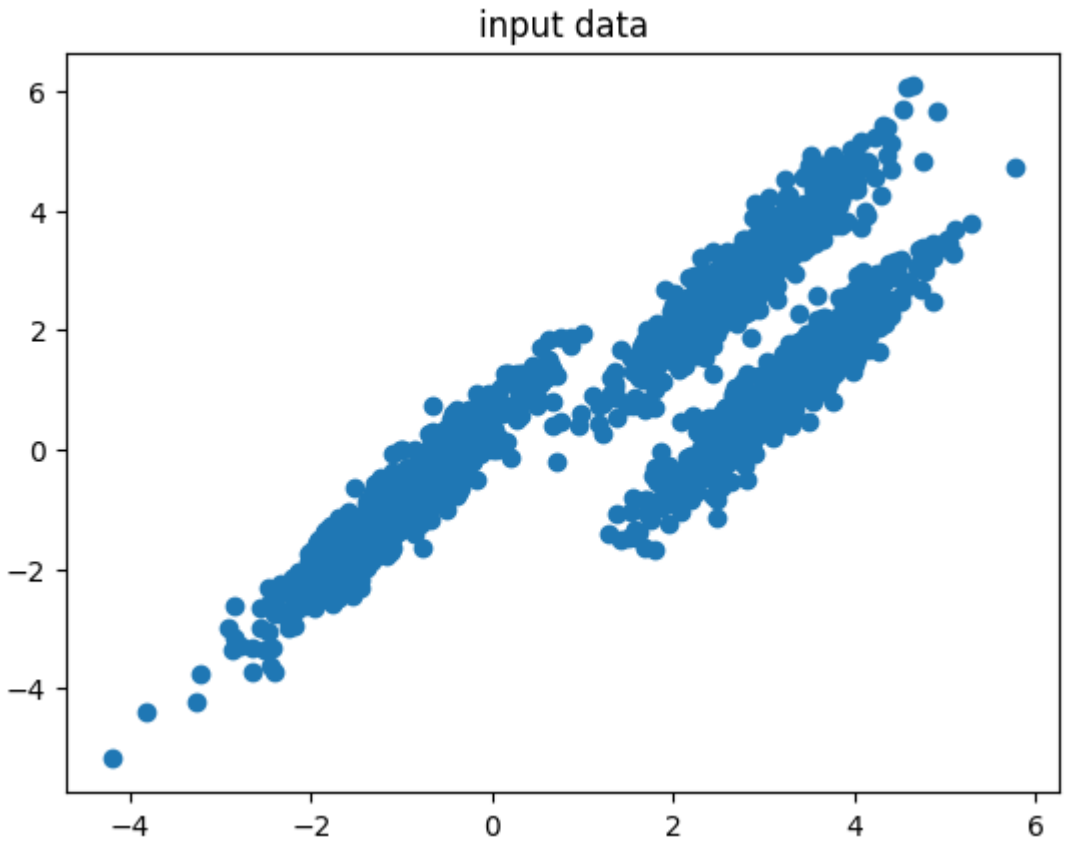
Please, read https://en.wikipedia.org/wiki/K-means_clustering (https://en.wikipedia.org/wiki/K-means_clustering) about the Kmeans algorithm.

In this problem, we will show how interperting a dataset as a graph may result is obtaining an elegant clustering solution. We have an input dataset that we wish to cluster in 3 aparant classes.

We provide the synthetic dataset of 2000 points described below where the T_matrix is just a 2D transformation matrix:

In [17]:
```
T_matrix, seed = [[-0.60834549, -0.63667341], [0.40887718, 0.85253229]], 170
X_orig, y_orig = make_blobs(n_samples=2000, random_state=seed)
X = np.dot(X_orig, T_matrix)
```

In [18]:
```
show_data_results(X, num_plot=1)
```

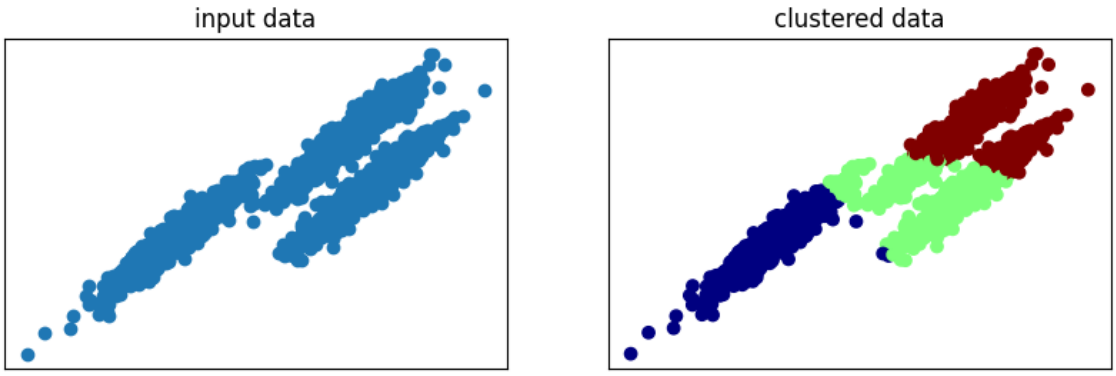


Using the the Kmeans algorithm implementation of sklearn, show your attempt to cluster this dataset into 3 classes in one luine of code.

Solution (Given)

In [19]:
```
y_pred = KMeans(n_clusters=3, random_state=seed).fit_predict(X)
```

In [20]:
```
show_data_results(X, 2, y_pred)
```



Q.2.

Comment on the output the the KMeans algorithm? Did it work? If so explain why, if not, explain not not.

Solution

Type *Markdown* and LaTeX: α^2

Q.3.

Let's now interprete every single point in the provided dataset as a node in a graph. Our goal is to find a way to relate every node in the graph is such way that they points that closer together maintain that relationship while points that are far are explicitely identified.

lots of points points are closed top each other and kmaeans is missing it. representing as the graph unveils the relation.

One way to capture such relationship between points (nodes) in a graph is through the Adjacency matrix. Typically, a simple adjacency matrix between nodes of and indiredted graph is given by:

$$A_{i,j} = \begin{cases} 1 : \text{if there is an edge between node i and node j,} \\ 0 : \text{otherwise.} \end{cases}$$

In this probem, we will use the weighted distances betweeen points instead as a similary measure. Write a function that takes in the input dataset and some coeficient gamma which returns the adjacency matrix A.

$$A_{i,j} = e^{\gamma ||x_i - x_j||^2}$$

where x_i and x_j represent each point in the provided dataset. You may find the $dictance$ module from $scipy.spatial$ useful.

In [21]:
```
print(X.shape)
```

```
(2000, 2)
```

Solution

In [40]:
```
def get_adjacency_matrix(gamma, X):
    # fill in your code here
    # adjacency_matrix = ?

    #num = X.shape[0]
    #adjacency_matrix = np.zeros((num,num))

    #for i in range(num):
    #    for j in range(i, num):
    #        dis = math.exp(-gamma * np.square(distance.euclidean(X[i],X[j])))
    #        adjacency_matrix[i][j] = dis
    #        adjacency_matrix[j][i] = dis

    adjacency_matrix = np.exp(-gamma * distance.cdist(X,X,metric='sqeuclidean'))

    return adjacency_matrix
```

In [41]:
```
adj = get_adjacency_matrix(gamma=1, X=X)
print(adj)
```

```
[[1.00000000e+00 1.16130639e-02 5.62548235e-10 ... 6.92521915e-09
  3.80974378e-15 6.19953663e-02]
 [1.16130639e-02 1.00000000e+00 4.22486809e-16 ... 3.40414939e-15
  1.31605674e-22 1.36491384e-02]
 [5.62548235e-10 4.22486809e-16 1.00000000e+00 ... 8.49932392e-01
  2.60842144e-01 7.64214433e-18]
 ...
 [6.92521915e-09 3.40414939e-15 8.49932392e-01 ... 1.00000000e+00
  1.26977418e-01 2.67051116e-16]
 [3.80974378e-15 1.31605674e-22 2.60842144e-01 ... 1.26977418e-01
  1.00000000e+00 1.22127633e-24]
 [6.19953663e-02 1.36491384e-02 7.64214433e-18 ... 2.67051116e-16
  1.22127633e-24 1.00000000e+00]]
```

In [26]:
```
adj.shape
```

Out[26]: (2000, 2000)

In [32]:
```
compare = np.exp(-1 * distance.cdist(X,X,metric='sqeuclidean'))
compare.shape
```

Out[32]: (2000, 2000)

In [33]:
```
compare
```

Out[33]:
```
array([[1.00000000e+00, 1.16130639e-02, 5.62548235e-10, ...,
        6.92521915e-09, 3.80974378e-15, 6.19953663e-02],
       [1.16130639e-02, 1.00000000e+00, 4.22486809e-16, ...,
        3.40414939e-15, 1.31605674e-22, 1.36491384e-02],
       [5.62548235e-10, 4.22486809e-16, 1.00000000e+00, ...,
        8.49932392e-01, 2.60842144e-01, 7.64214433e-18],
       ...,
       [6.92521915e-09, 3.40414939e-15, 8.49932392e-01, ...,
        1.00000000e+00, 1.26977418e-01, 2.67051116e-16],
       [3.80974378e-15, 1.31605674e-22, 2.60842144e-01, ...,
        1.26977418e-01, 1.00000000e+00, 1.22127633e-24],
       [6.19953663e-02, 1.36491384e-02, 7.64214433e-18, ...,
        2.67051116e-16, 1.22127633e-24, 1.00000000e+00]])
```

Q. 4.

The degree matrix of an undirected graph is a diagonal matrix which contains information about the degree of each vertex. In other word, it contains the number of edges attached to each vertex and it is given by:

$$D_{i,j} = \begin{cases} deg(v_i) : \text{if i} == \text{j}, \\ 0 : \text{otherwise.} \end{cases}$$

where the degree $\deg(v_i)$ of a vertex counts the number of times an edge terminates at that vertex. Note that in the traditional definition of the adjacency matrix, this boils down to the diagonal matrix in which element along the diagonals are column-wise sum of the adjacency matrix. Using the same idea, write a function that takes in the adjacency matrix as argument and returns the inverse square root of degree matrix.

Solution

In [27]:
```
def get_degree_matrix(adjacency_matrix):
    # fill in your code here
    # degree_matrix = ?
    degree_matrix = np.diag(np.sum(adjacency_matrix, axis=0))
    return degree_matrix
```

Type *Markdown* and LaTeX: α^2

Q. 5.

Using γ = 7.5, compute the symmetrically normalized adjacency matrix A, degree matrix D and the matrix $M = D^{-1/2} A D^{-1/2}$

Solution

In [46]:
```
# adjacency_matrix = ?
# degree_matrix = ?
# M = ?
adjacency_matrix = get_adjacency_matrix(7.5, X)
degree_matrix = get_degree_matrix(adjacency_matrix)
inv = np.linalg.inv(np.sqrt(degree_matrix))
M = inv @ adjacency_matrix @ inv
```

In [47]:
```
M
```

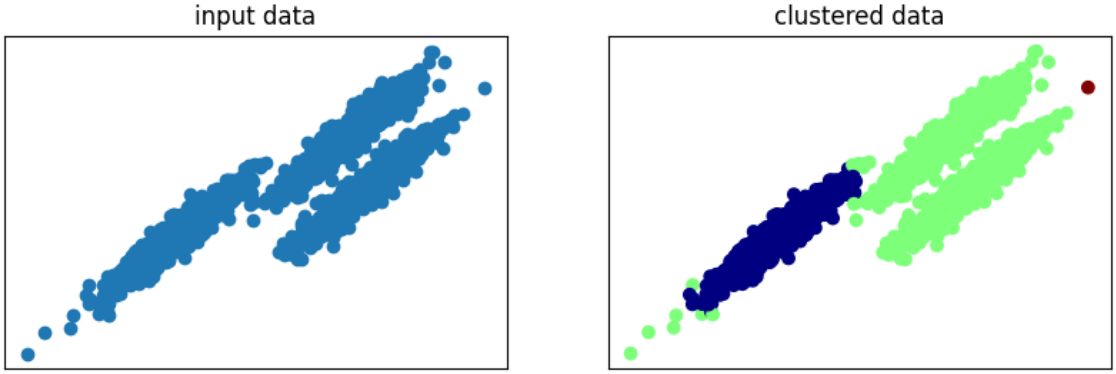
Out[47]:
```
array([[1.56324335e-002, 5.95177042e-017, 5.28958398e-072, ...,
        8.42495130e-064, 1.55115214e-110, 1.17671220e-011],
       [5.95177042e-017, 2.40471459e-002, 7.66196907e-118, ...,
        5.08041331e-111, 2.09900856e-166, 1.71705334e-016],
       [5.28958398e-072, 7.66196907e-118, 1.00096744e-002, ...,
        3.13253209e-003, 7.24379279e-007, 4.52154623e-131],
       ...,
       [8.42495130e-064, 5.08041331e-111, 3.13253209e-003, ...,
        1.12358359e-002, 3.46875976e-009, 1.80186978e-119],
       [1.55115214e-110, 2.09900856e-166, 7.24379279e-007, ...,
        3.46875976e-009, 2.97745821e-002, 8.29831903e-182],
       [1.17671220e-011, 1.71705334e-016, 4.52154623e-131, ...,
        1.80186978e-119, 8.29831903e-182, 1.15325532e-002]])
```

Q. 6.

Using SVD decomposition, select the first 3 vectors in the matrix U and perform the same KMeans clustering used above on them. What do you observe? Did it work? If so explain why, if not, explain not not.

Solution

In [48]:
```
u, s, vh = np.linalg.svd(M)
svd_y_pred = KMeans(n_clusters=3, random_state=seed).fit_predict(u[:, :3])
show_data_results(X, 2, svd_y_pred)
```



In [50]:
```
u[:, :3].shape
```

Out[50]:
```
(2000, 3)
```

Q.7.

Now lets think of the Adjacency obtained above as the transition Matrix in of a Markov Chain.To do so, A needs to be a proper stochastic matrix which means that the sum of the element in each column must add up to 1. Write a function that takes in the matrix M and returns M_stachastic, the stochastic version of M; compute the stochastic matrix

Solution

In [51]:
```
def stochastic_matrix_converter(M):
    # fill in your code here
    # degree_matrix = ?
    M_stoch = M / np.sum(M, axis=0)
    return M_stoch
```

In [52]:
```
M_stoch = stochastic_matrix_converter(M)
M_stoch
```

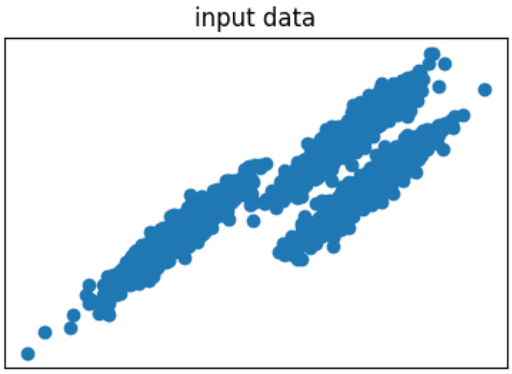
Out[52]:
```
array([[1.63135023e-002, 6.90398945e-017, 4.96294146e-072, ...,
        8.26245010e-064, 1.79736241e-110, 1.14711464e-011],
       [6.21107523e-017, 2.78944298e-002, 7.18882696e-118, ...,
        4.98242185e-111, 2.43217864e-166, 1.67386472e-016],
       [5.52003887e-072, 8.88780144e-118, 9.39155672e-003, ...,
        3.07211154e-003, 8.39358087e-007, 4.40781688e-131],
       ...,
       [8.79200687e-064, 5.89322461e-111, 2.93909189e-003, ...,
        1.10191181e-002, 4.01934683e-009, 1.75654779e-119],
       [1.61873223e-110, 2.43482727e-166, 6.79647391e-007, ...,
        3.40185402e-009, 3.45006229e-002, 8.08959343e-182],
       [1.22797882e-011, 1.99176335e-016, 4.24233159e-131, ...,
        1.76711516e-119, 9.61548928e-182, 1.12424777e-002]])
```

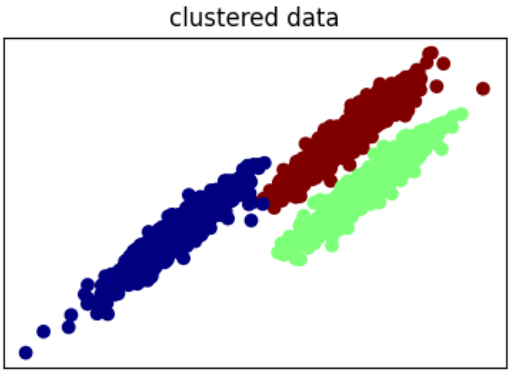
Q.8

Now, let's investigate how could we have made the matrix M work directly in our original interpretation. To do this, normalize those 3 vectors first before performing the clustering.

Solution

In [53]:
```
u, s, vh = np.linalg.svd(M_stoch)
svd_y_pred = KMeans(n_clusters=3, random_state=seed).fit_predict(u[:, :3])
show_data_results(X, 2, svd_y_pred)
```





In [ ]:

Watching SGD in action with constant step sizes

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA
```

In [2]:
```
def compute_loss_avg(X, y, w):
    return (1/X.shape[0])*LA.norm(X@w - y, ord=2)**2
```

In [6]:
```
def SGD_update(X, y, w, eta):
    return w - (2.0 * eta/X.shape[0])*(X.transpose()@(X@w - y))
```

Part (1). Under-parameterized ($n > d$) Noiseless ($\sigma = 0$) Regime

Generate data

In [3]:
```
# Generate data
np.random.seed(0)

# Set number of samples
N = 2000
# Set the dimension
d = 200

# Generate data matrix X_train
X_train = np.random.randn(N, d)
# Generate ground truth w_star
w_star = np.random.randn(d, 1)

# Generate outputs y_train
y_train = X_train @ w_star
# Set mini batch size
batch_size = 64
# Set step size
eta = 0.01
# Set number of iterations
N_iteration = 10000

# Evaluate the largest and smallest eigenvalue
_, s, _ = np.linalg.svd(X_train/np.sqrt(N))
print('largest eigenvalue: ', s[0]**2)
print('smallest eigenvalue: ', s[-1]**2)
```

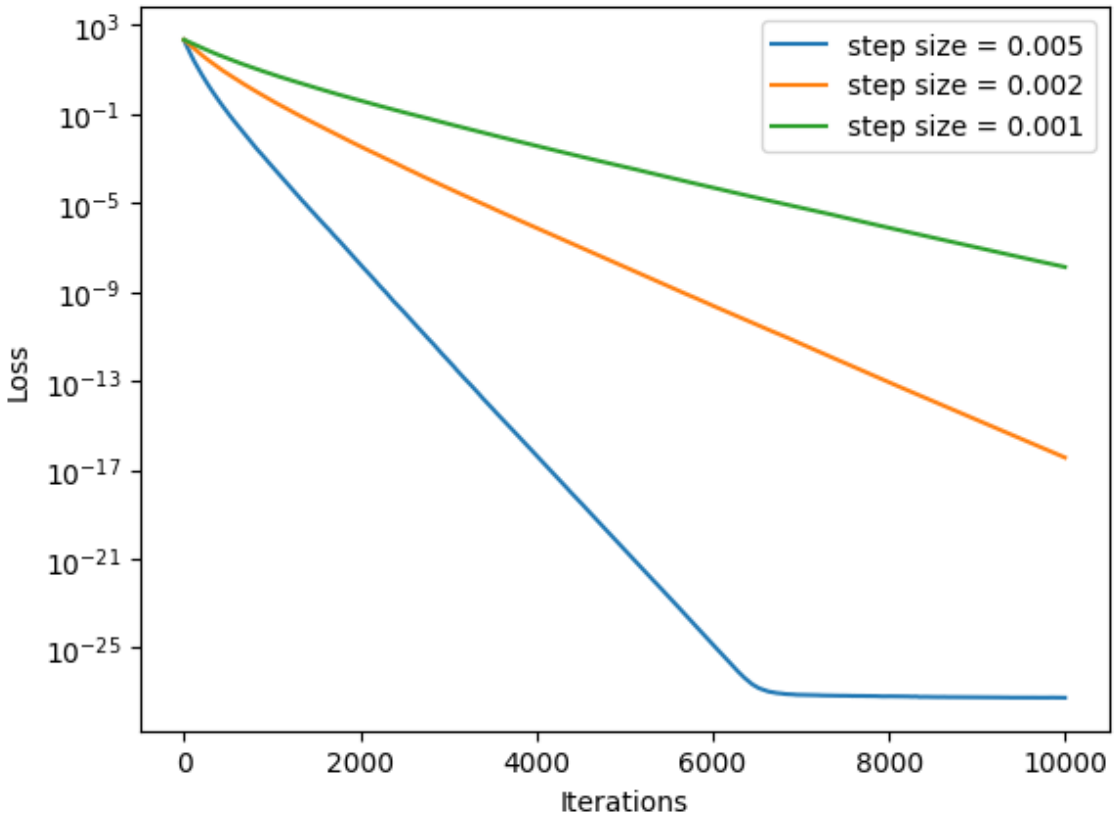
```
largest eigenvalue:  1.7095023515203154
smallest eigenvalue:  0.47581318789985916
```

In [4]:
```
BatchSizeList = [1, 64, 128]
EtaList = [0.005, 0.002, 0.001]
```

Study the effect of step size η

In [7]:
```
w_init = (np.random.randn(d, 1)) * 0.0

Losses = []
batch_size = 64
for eta in EtaList:
    loss = []
    w = w_init.copy()
    for i in range(N_iteration):
        random_index = np.random.choice(N, batch_size)
        X_i = X_train[random_index, :]
        y_i = y_train[random_index]
        w = SGD_update(X_i, y_i, w, eta)
        loss.append(compute_loss_avg(X_train, y_train, w))
    Losses.append(loss)
```

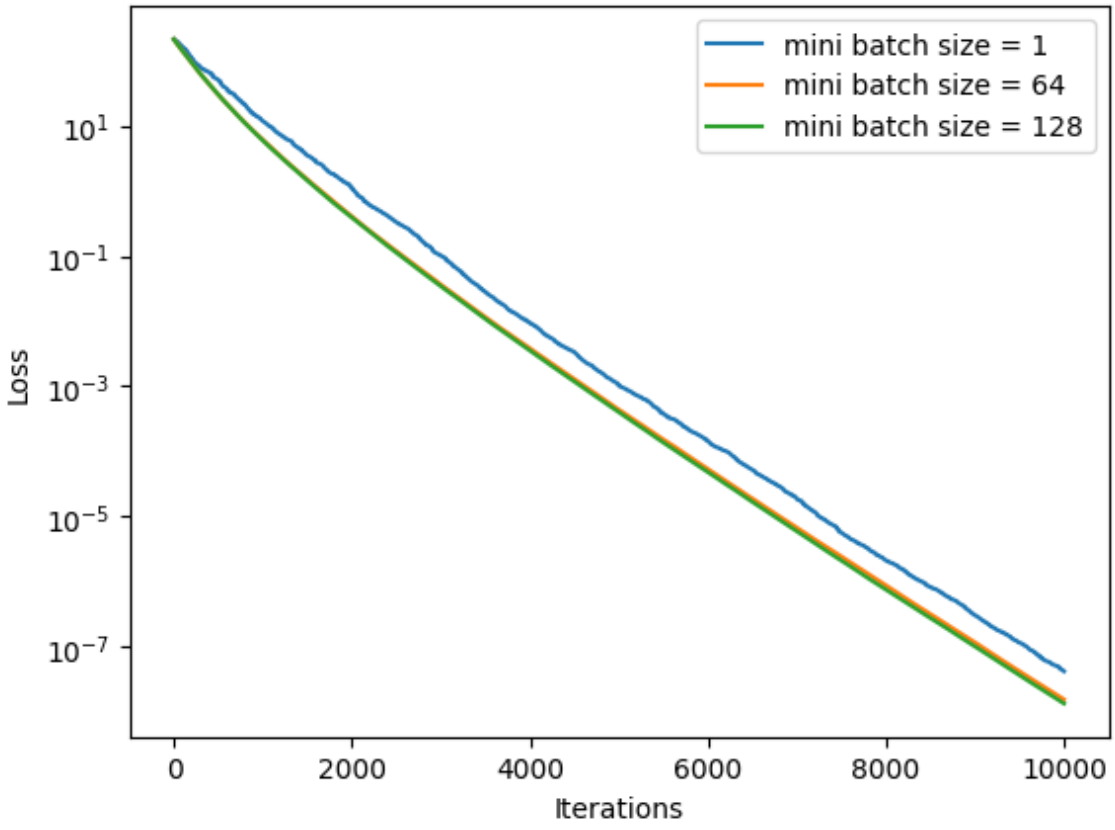
In [8]:
```
plt.figure()
idx = 0
for eta in EtaList:
    plt.semilogy(range(N_iteration), Losses[idx], label = 'step size = {}'.format(et
    idx += 1
plt.axis('tight')
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Study the effect of mini batch size $|\mathcal{S}_t|$

In [9]:
```
w_init = (np.random.randn(d, 1)) * 0.0
Losses = []
eta = 0.001
for batch_size in BatchSizeList:
    loss = []
    w = w_init.copy()
    for i in range(N_iteration):
        random_index = np.random.choice(N, batch_size)
        X_i = X_train[random_index, :]
        y_i = y_train[random_index]
        w = SGD_update(X_i, y_i, w, eta)
        loss.append(compute_loss_avg(X_train, y_train, w))
    Losses.append(loss)
```

In [10]:
```
plt.figure()
idx = 0
for batch_size in BatchSizeList:
    plt.semilogy(range(N_iteration), Losses[idx], label = 'mini batch size = {}'.for
    idx += 1
plt.axis('tight')
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Part (2). Over-parameterized ($n < d$) Noiseless ($\sigma = 0$) Regime

Generate data

In [11]:
```
# Generate data
np.random.seed(0)

# Set number of samples
N = 500
# Set the dimension
d = 1000

# Generate data matrix X_train
X_train = np.random.randn(N, d)
# Generate ground truth w_star
w_star = np.random.randn(d, 1)

# Generate outputs y_train
y_train = X_train @ w_star
# Set mini batch size
batch_size = 64
# Set step size
eta = 0.001
# Set number of iterations
N_iteration = 50000


# Evaluate the largest and smallest eigenvalue
_, s, _ = np.linalg.svd(X_train/np.sqrt(N))
print('largest eigenvalue: ', s[0]**2)
print('smallest singular value (square): ', s[-1]**2)
```

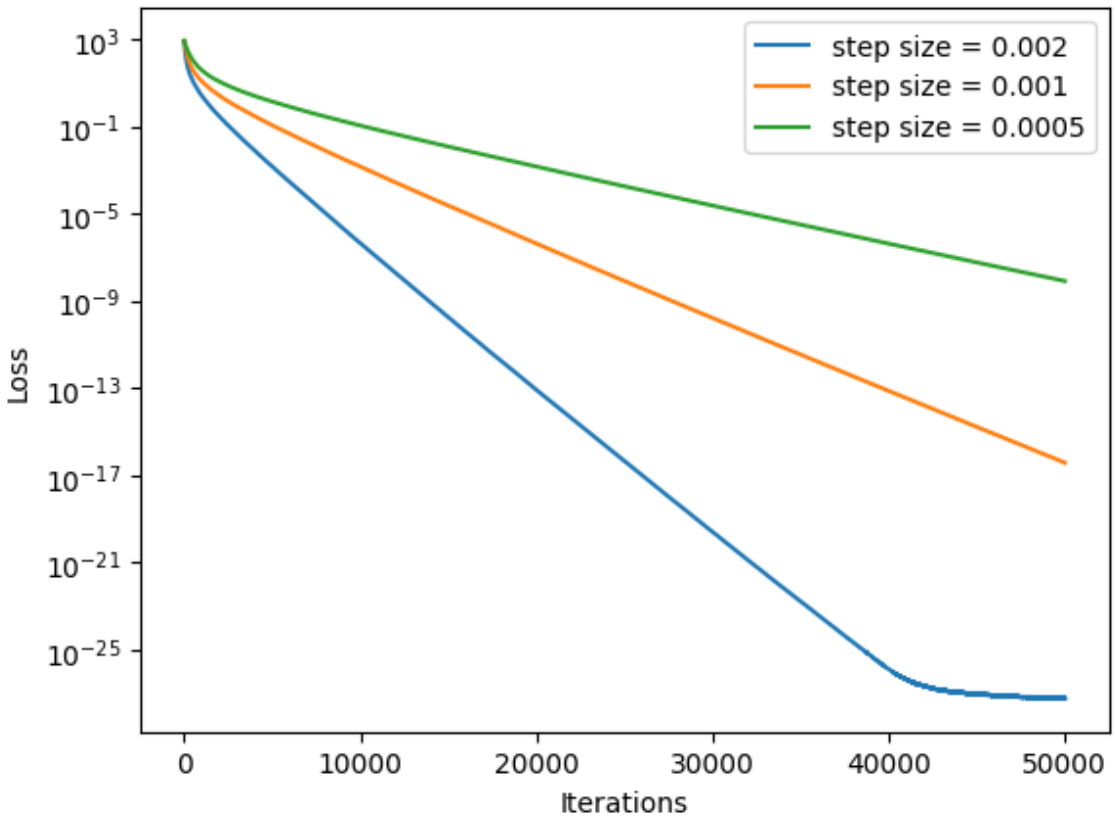
```
largest eigenvalue:  5.771623003224577
smallest singular value (square):  0.17331335068830736
```

In [12]:
```
BatchSizeList = [1, 64, 128]
EtaList = [0.002, 0.001, 0.0005]
```

Study the effect of step size η

In [14]:
```
w_init = (np.random.randn(d, 1)) * 0.0

Losses = []
batch_size = 64
for eta in EtaList:
    loss = []
    w = w_init.copy()
    for i in range(N_iteration):
        random_index = np.random.choice(N, batch_size)
        X_i = X_train[random_index, :]
        y_i = y_train[random_index]
        w = SGD_update(X_i, y_i, w, eta)
        loss.append(compute_loss_avg(X_train, y_train, w))
    Losses.append(loss)
```

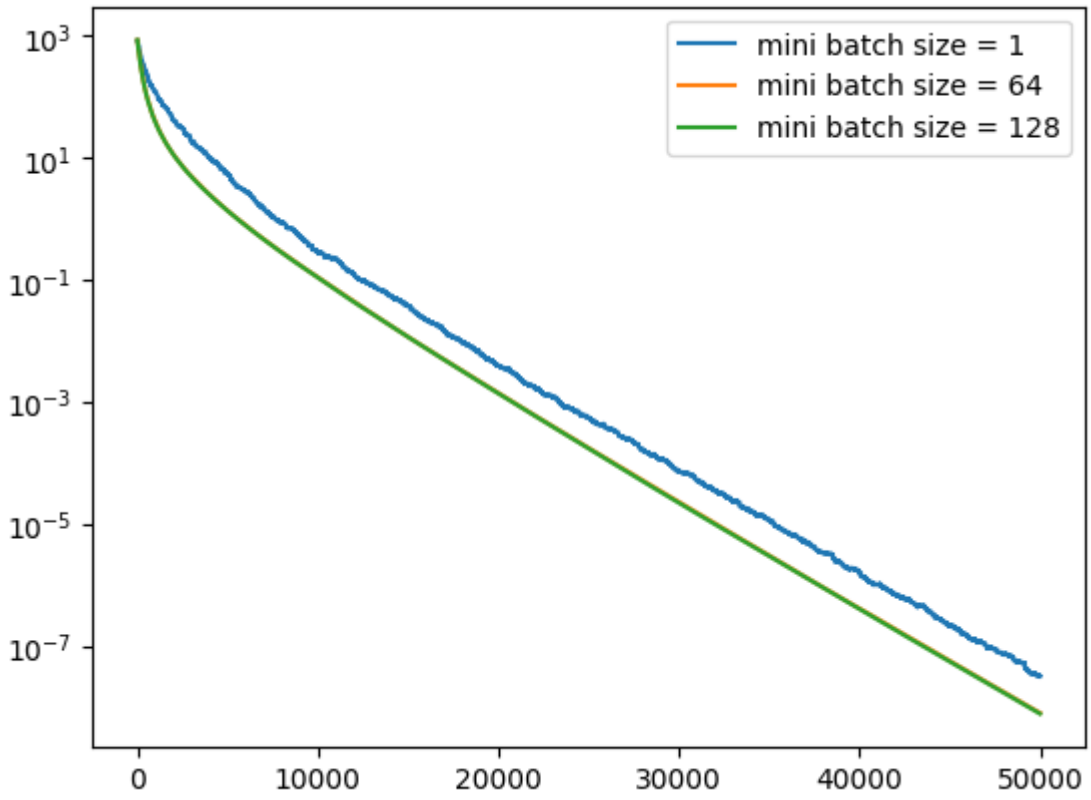
In [15]:
```
plt.figure()
idx = 0
for eta in EtaList:
    plt.semilogy(range(N_iteration), Losses[idx], label = 'step size = {}'.format(et
    idx += 1
plt.axis('tight')
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Study the effect of mini batch size $|S_t|$

In [16]:
```
w_init = (np.random.randn(d, 1)) * 0.0
Losses = []
eta = 0.0005
for batch_size in BatchSizeList:
    loss = []
    w = w_init.copy()
    for i in range(N_iteration):
        random_index = np.random.choice(N, batch_size)
        X_i = X_train[random_index, :]
        y_i = y_train[random_index]
        w = SGD_update(X_i, y_i, w, eta)
        loss.append(compute_loss_avg(X_train, y_train, w))
    Losses.append(loss)
```

In [17]:
```
plt.figure()
idx = 0
for batch_size in BatchSizeList:
    plt.semilogy(range(N_iteration), Losses[idx], label = 'mini batch size = {}'.for
    idx += 1
plt.axis('tight')
plt.legend()
plt.show()
```



Part (3). Over-parameterized ($n < d$) Noise ($\sigma > 0$) Regime

Generate data

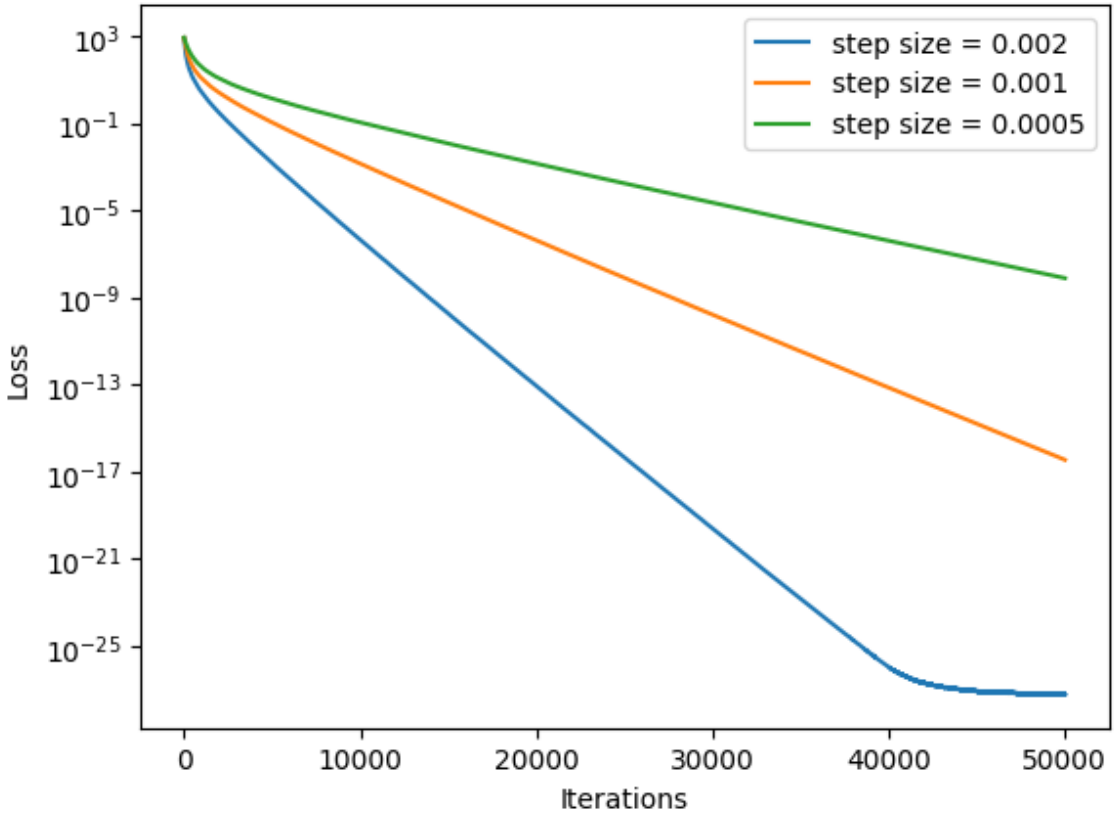
In [18]:
```
# Generate data
np.random.seed(0)

# Set number of samples
N = 500
# Set the dimension
d = 1000

# Generate data matrix X_train
X_train = np.random.randn(N, d)
# Generate ground truth w_star
w_star = np.random.randn(d, 1)

# Generate outputs y_train
y_train = X_train @ w_star + 0.1 * np.random.randn(N, 1)
# Set mini batch size
batch_size = 64
# Set step size
eta = 0.001
# Set number of iterations
N_iteration = 50000


# Evaluate the largest and smallest eigenvalue
X_train = np.concatenate((X_train, 0.1 * np.eye(N)), axis=1)
_, s, _ = np.linalg.svd(X_train/np.sqrt(N))
print('largest eigenvalue: ', s[0]**2)
print('smallest singular value (square): ', s[-1]**2)
```

```
largest eigenvalue:  5.77164300322458
smallest singular value (square):  0.17333335068830757
```

Study the effect of step size η

In [19]:
```
w_init = (np.random.randn(d + N, 1)) * 0.0

Losses = []
batch_size = 64
for eta in EtaList:
    loss = []
    w = w_init.copy()
    for i in range(N_iteration):
        random_index = np.random.choice(N, batch_size)
        X_i = X_train[random_index, :]
        y_i = y_train[random_index]
        w = SGD_update(X_i, y_i, w, eta)
        loss.append(compute_loss_avg(X_train, y_train, w))
    Losses.append(loss)
```

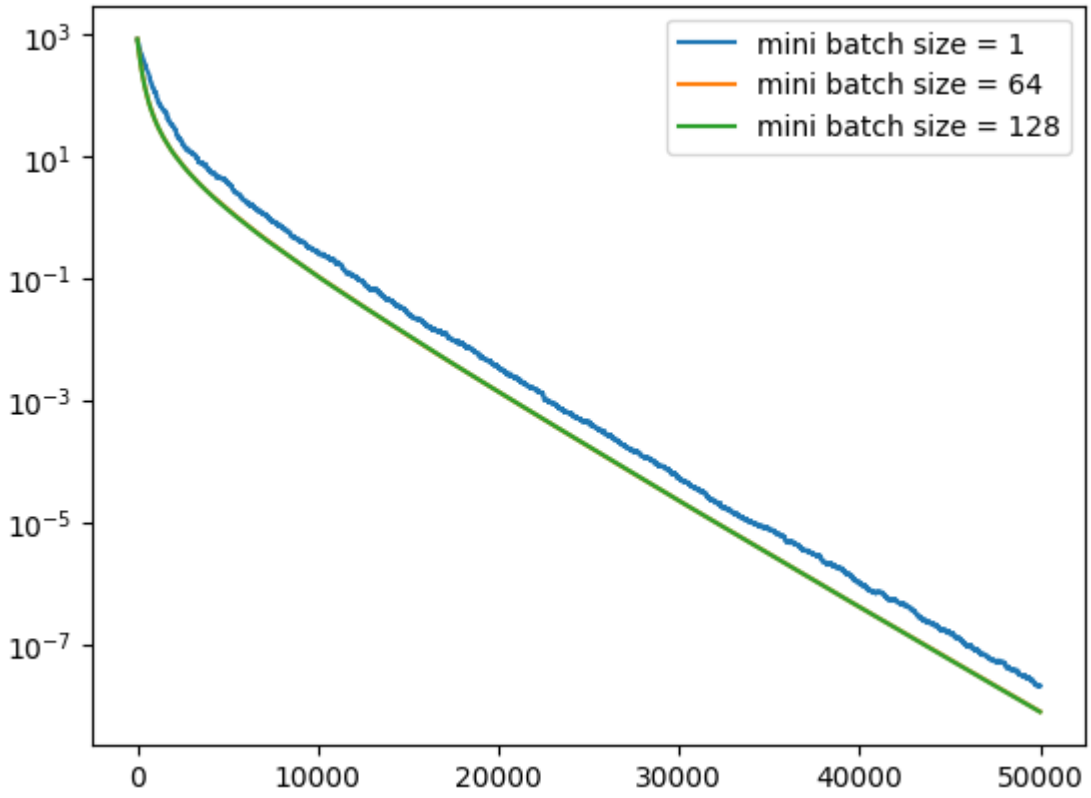
In [20]:
```
plt.figure()
idx = 0
for eta in EtaList:
    plt.semilogy(range(N_iteration), Losses[idx], label = 'step size = {}'.format(et
    idx += 1
plt.axis('tight')
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Study the effect of mini batch size $|\mathcal{S}_t|$

In [21]:
```
w_init = (np.random.randn(d + N, 1)) * 0.0
Losses = []
eta = 0.0005
for batch_size in BatchSizeList:
    loss = []
    w = w_init.copy()
    for i in range(N_iteration):
        random_index = np.random.choice(N, batch_size)
        X_i = X_train[random_index, :]
        y_i = y_train[random_index]
        w = SGD_update(X_i, y_i, w, eta)
        loss.append(compute_loss_avg(X_train, y_train, w))
    Losses.append(loss)
```

In [22]:
```
plt.figure()
idx = 0
for batch_size in BatchSizeList:
    plt.semilogy(range(N_iteration), Losses[idx], label = 'mini batch size = {}'.for
    idx += 1
plt.axis('tight')
plt.legend()
plt.show()
```



Part (4). Under-parameterized ($n > d$) Noise ($\sigma > 0$) Regime

Compare SGD on original ridge regression and feature-augmented regression

In [23]:
```
# Generate data
np.random.seed(0)

# Set number of samples
N = 500
# Set the dimension
d = 50

# Generate data matrix X_train
X_train = np.random.randn(N, d)
# Generate ground truth w_star
w_star = np.random.randn(d, 1)

# Generate outputs y_train
y_train = X_train @ w_star + 0.1 * np.random.randn(N, 1)
y_train_clean = X_train @ w_star
# Set mini batch size
batch_size = 64
# Set step size
eta = 0.001
# Set number of iterations
N_iteration = 500000


alpha = 0.01

w_star = np.linalg.inv(X_train.transpose()@X_train + N * alpha * np.eye(d))@X_train
w_star_clean = np.linalg.inv(X_train.transpose()@X_train)@X_train.transpose()@y_tra


# Evaluate the largest and smallest eigenvalue
X_train_aug = np.concatenate((X_train, np.sqrt(N * alpha) * np.eye(N)), axis=1)
_, s, _ = np.linalg.svd(X_train_aug/np.sqrt(N))
print('largest eigenvalue: ', s[0]**2)
print('smallest singular value (square): ', s[-1]**2)
```

```
largest eigenvalue:  1.6696575471365793
smallest singular value (square):  0.009999999999999981
```

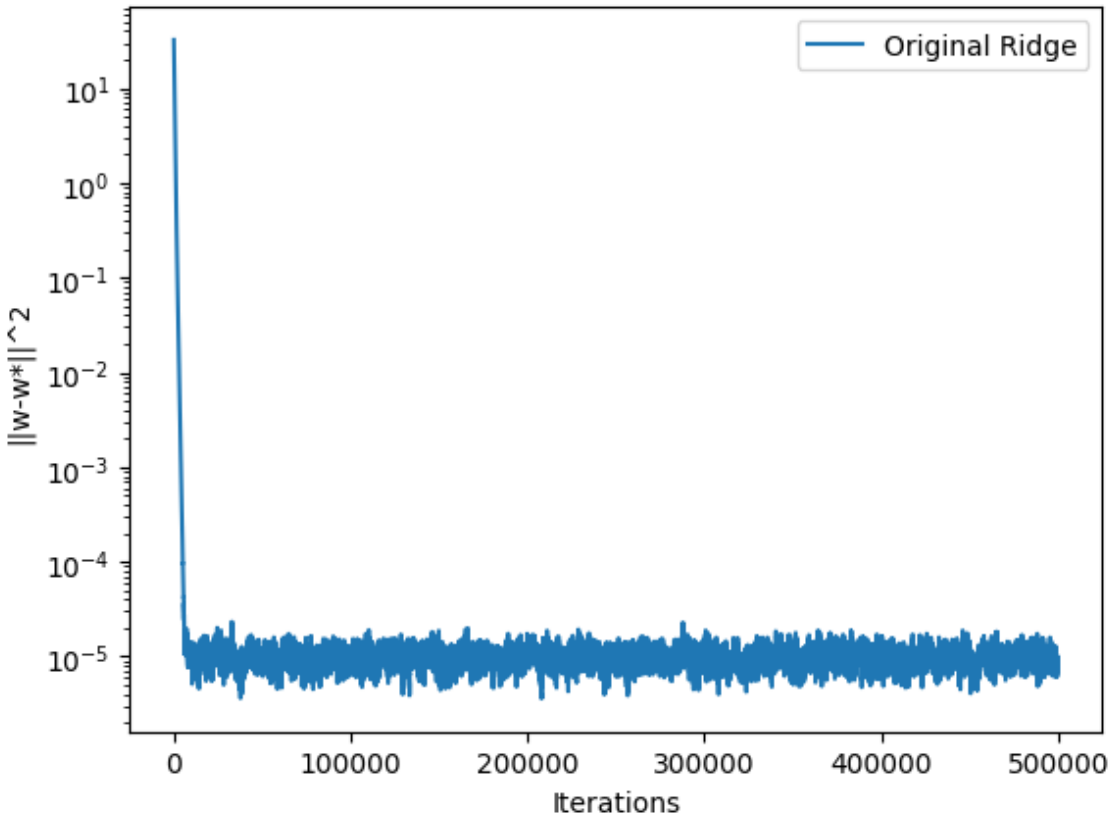
In [24]:
```
def compute_diff_norm(w, w_star):
    return LA.norm(w - w_star, ord=2)**2
```

In [25]:
```
def SGD_update_ridge(X, y, w, eta, alpha = 0.01):
    return w - (2.0 * eta/X.shape[0])*(X.transpose()@(X@w - y)) - 2.0 * eta * alph
```

Run SGD on original ridge regression

In [26]:
```
w_init = (np.random.randn(d, 1)) * 0.0
loss_ridge = []
w = w_init.copy()
for i in range(N_iteration):
    random_index = np.random.choice(N, batch_size)
    X_i = X_train[random_index, :]
    y_i = y_train[random_index]
    w = SGD_update_ridge(X_i, y_i, w, eta, alpha)
    loss_ridge.append(compute_diff_norm(w, w_star))
```

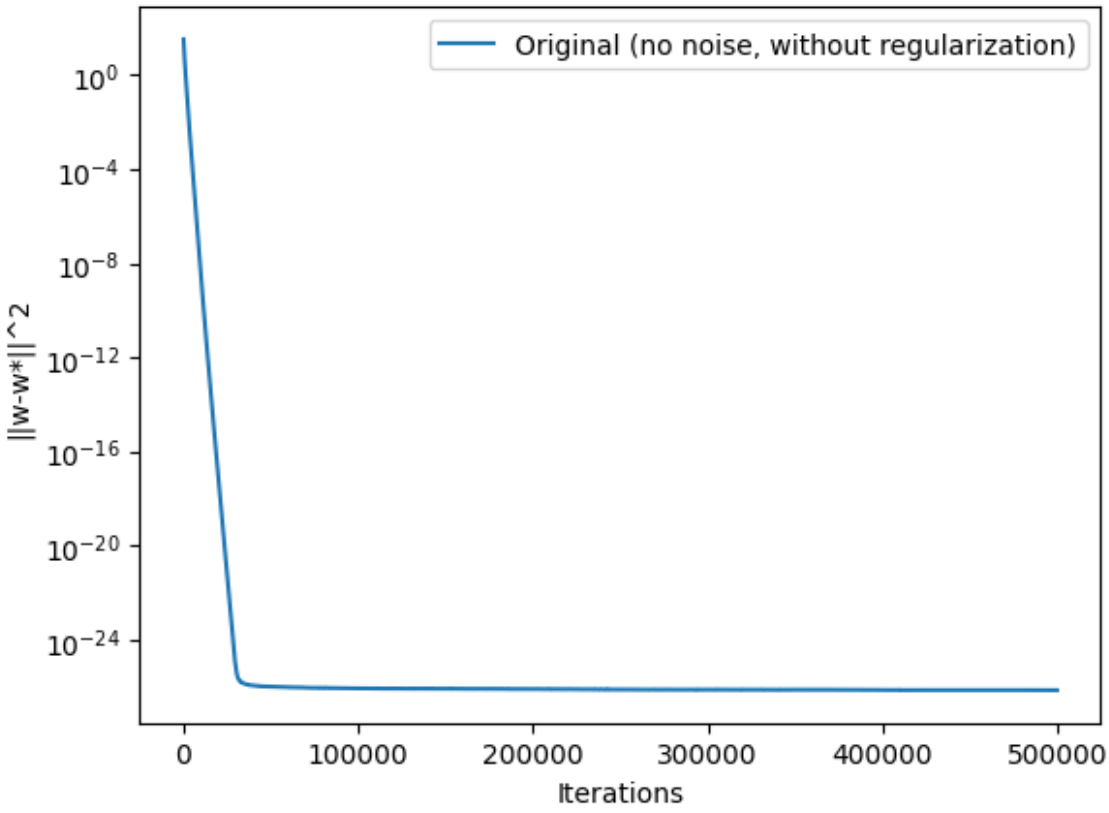
In [27]:
```
plt.figure()
plt.semilogy(range(N_iteration), loss_ridge, label = 'Original Ridge')
plt.axis('tight')
plt.xlabel("Iterations")
plt.ylabel("||w-w*||^2")
plt.legend()
plt.show()
```



Run SGD on original regression (but with no noise in y, i.e., $\sigma = 0.0$)

In [28]:
```
w_init = (np.random.randn(d, 1)) * 0.0
loss_ridge_clean = []
w = w_init.copy()
for i in range(N_iteration):
    random_index = np.random.choice(N, batch_size)
    X_i = X_train[random_index, :]
    y_i = y_train_clean[random_index]
    w = SGD_update(X_i, y_i, w, eta)
    loss_ridge_clean.append(compute_diff_norm(w, w_star_clean))
```

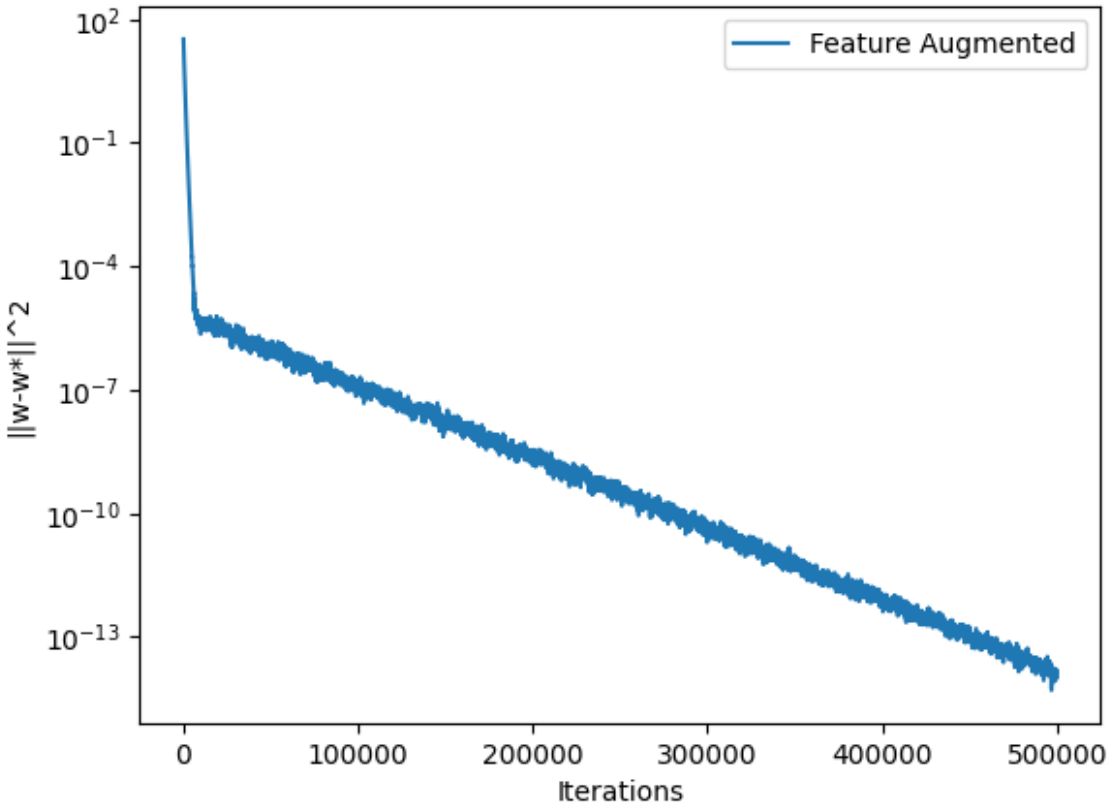
In [29]:
```
plt.figure()
plt.semilogy(range(N_iteration), loss_ridge_clean, label = 'Original (no noise, with
plt.axis('tight')
plt.xlabel("Iterations")
plt.ylabel("||w-w*||^2")
plt.legend()
plt.show()
```



Run SGD on augmented regression

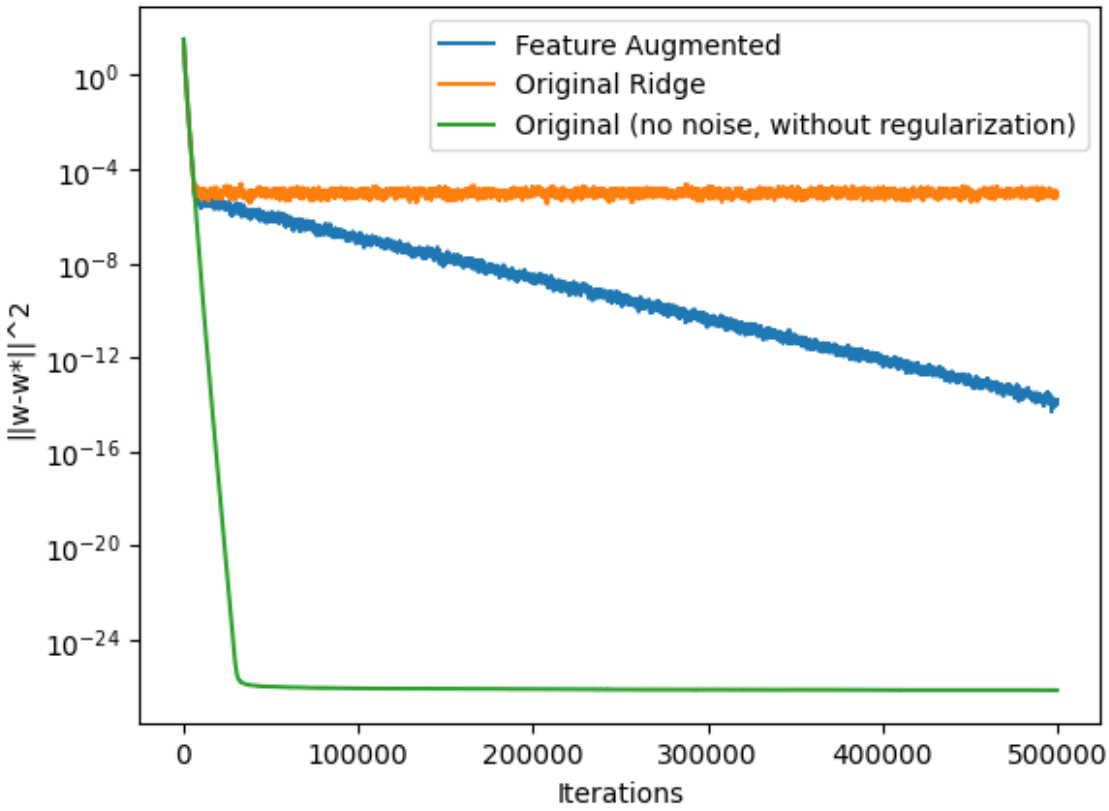
In [30]:
```
w_aug_init = (np.random.randn(d + N, 1)) * 0.0

loss_aug = []
w_aug = w_aug_init.copy()
for i in range(N_iteration):
    random_index = np.random.choice(N, batch_size)
    X_i = X_train_aug[random_index, :]
    y_i = y_train[random_index]
    w_aug = SGD_update(X_i, y_i, w_aug, eta)
    loss_aug.append(compute_diff_norm(w_aug[:d], w_star))
```

In [31]:
```
plt.figure()
plt.semilogy(range(N_iteration), loss_aug, label = 'Feature Augmented')
plt.axis('tight')
plt.xlabel("Iterations")
plt.ylabel("||w-w*||^2")
plt.legend()
plt.show()
```



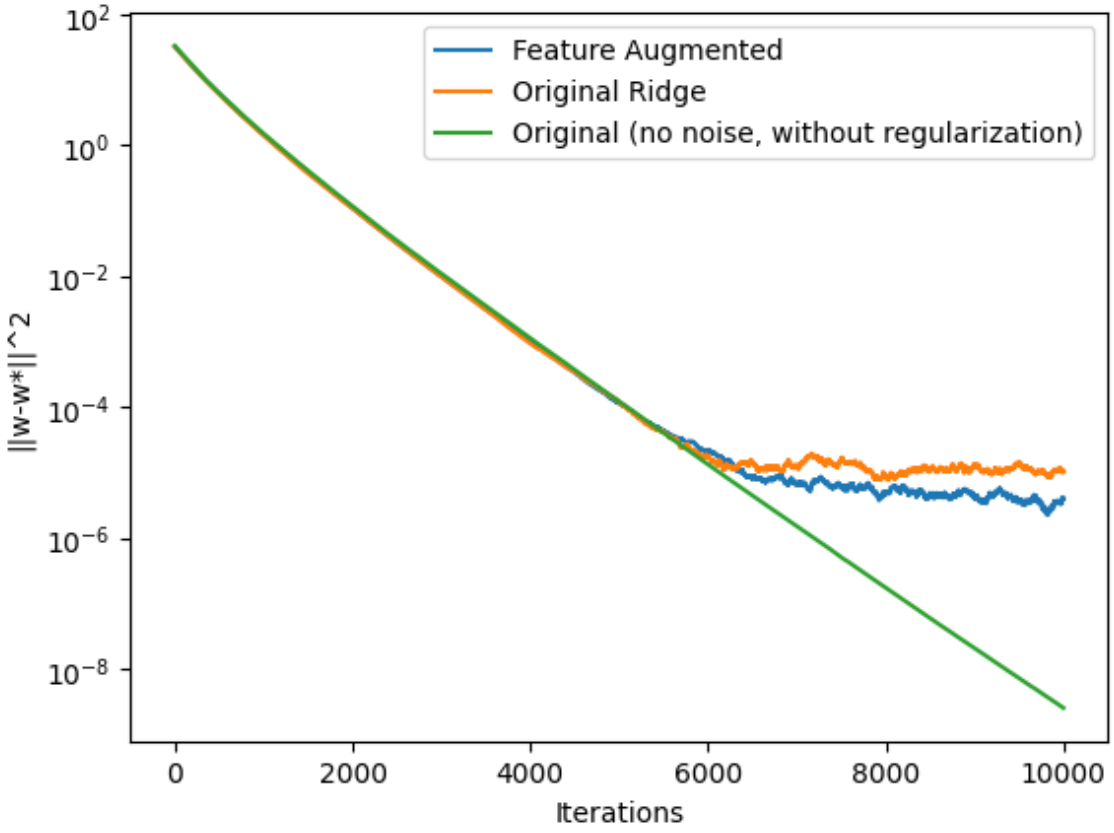
Compare the above three figures

In [32]:
```
plt.figure()
plt.semilogy(range(N_iteration), loss_aug, label = 'Feature Augmented')
plt.semilogy(range(N_iteration), loss_ridge, label = 'Original Ridge')
plt.semilogy(range(N_iteration), loss_ridge_clean, label = 'Original (no noise, with
plt.axis('tight')
plt.xlabel("Iterations")
plt.ylabel("||w-w*||^2")
plt.legend()
plt.show()
```



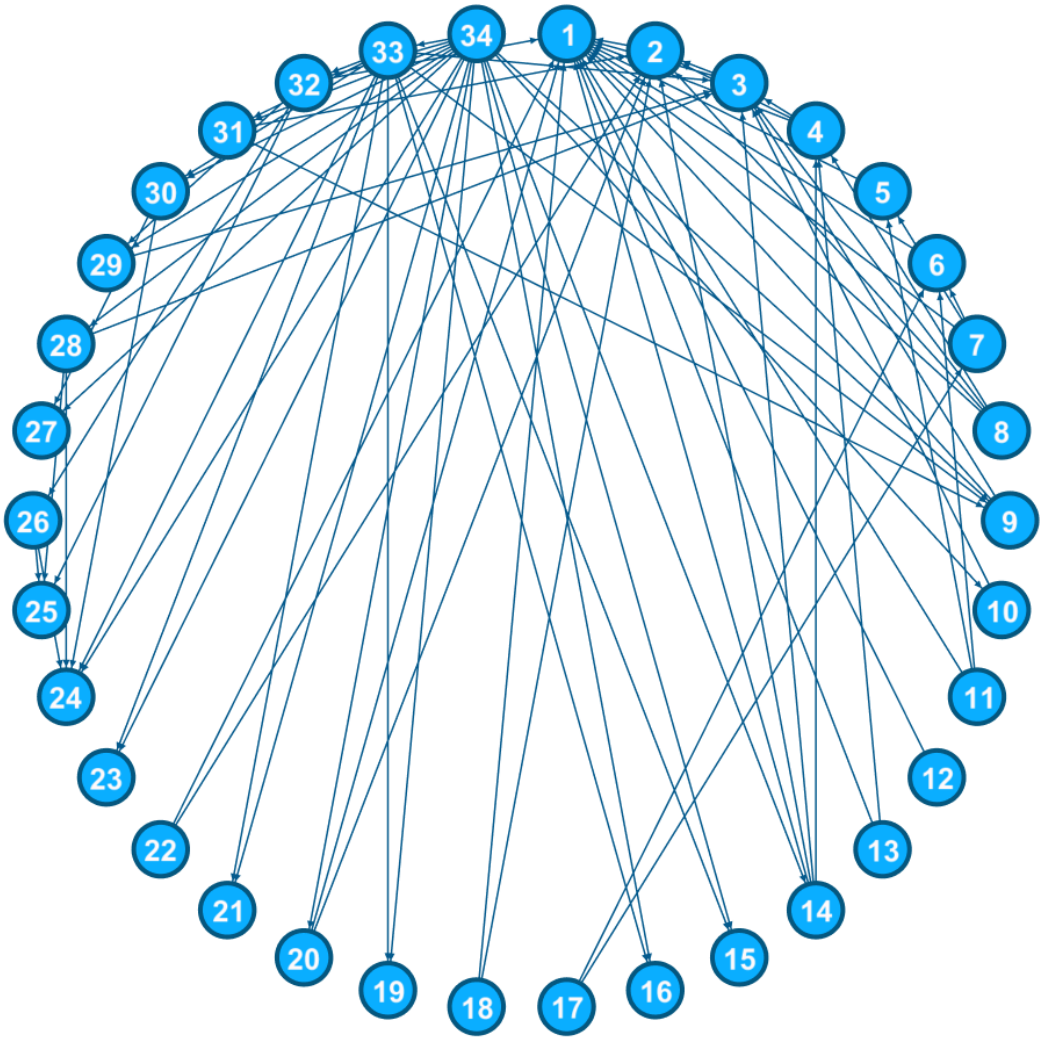
Zoom in Visualization

In [33]:
```
plt.figure()
plt.semilogy(range(N_iteration)[:10000], loss_aug[:10000], label = 'Feature Augmente
plt.semilogy(range(N_iteration)[:10000], loss_ridge[:10000], label = 'Original Ridge
plt.semilogy(range(N_iteration)[:10000], loss_ridge_clean[:10000], label = 'Original
plt.axis('tight')
plt.xlabel("Iterations")
plt.ylabel("||w-w*||^2")
plt.legend()
plt.show()
```



In [ ]:

Zachary's Karate Club



Zachary's karate club (ZKC) is a social network of a university karate club, described in the paper "An Information Flow Model for Conflict and Fission in Small Groups" by Wayne W. Zachary.

The social network captures 34 members of a karate club, documenting links between pairs of members who interacted outside the club.

During the study, a conflict arose between the officer/ administrator ("John A") and the instructor "Mr. Hi", which led to the split of the club into two.

Part of the members formed a new club around Mr. Hi; and the remaining members went with the officer.

Based on collected data Zachary correctly assigned all but one member of the club to the groups they actually joined after the split. You could read more about it here https://en.wikipedia.org/wiki/Zachary%27s_karate_club (https://en.wikipedia.org/wiki/Zachary%27s_karate_club), here

https://www.jstor.org/stable/3629752 (https://www.jstor.org/stable/3629752), and here https://commons.wikimedia.org/wiki/File:Social_Network_Model_of_Relationships_in_the_Kara

Import Dependencies

In [1]:
```
import numpy as np
import networkx as ntwx
from scipy.linalg import sqrtm
import matplotlib.pyplot as plt
from matplotlib import animation
from IPython.display import HTML as web
from networkx.algorithms.community.modularity_max import greedy_modularity_communit

%matplotlib inline
```

In [2]:
```
def set_seed(seed=100):
    """sets seed"""
    np.random.seed(seed)
```

Processing Helper Functions

In [3]:
```
def get_graph_metadata(graph, key=None):
    """Return metadata obout a graph i.e. number_of_nodes, number_of_edges and node
    if key is None:
        return graph.number_of_nodes(), graph.number_of_edges()
    return graph.number_of_nodes(), graph.number_of_edges(), ntwx.get_node_attribut

def get_xavier_init(input_dim, output_dim):
    """returns xaviers in initializer"""
    std_dev = np.sqrt(5.0 / (input_dim + output_dim))
    return np.random.uniform(-std_dev, std_dev, size=(input_dim, output_dim))

def get_cross_entropy(pred, labels):
    """computes crossentropy between the predictions and the labels"""
    return -np.log(pred)[np.arange(pred.shape[0]), np.argmax(labels, axis=1)]


def normalized_difference_norm(dW, dW_approx):
    """compares the deirivarive of the weight with its apprimation"""
    return np.linalg.norm(dW - dW_approx) / (np.linalg.norm(dW) + np.linalg.norm(dW

def get_colors_labels_and_classes(graph, num_nodes):
    """We applied greedy modurality maximization from the original
    paper https://arxiv.org/pdf/1609.02907.pdf to come up with cluster
    labels for each of the members of the club. We will train our GNN
    to predict these cluster labels."""
    clusters = greedy_modularity_communities(graph)
    unsure_cluster = clusters[-1]
    clusters = clusters[:2]
    clusters[1] = clusters[1].union(unsure_cluster)
    color_lists = np.zeros(num_nodes)
    for i, cluster in enumerate(clusters):
        color_lists[list(cluster)] = i
    classes = np.unique(color_lists).shape[0]
    return color_lists, np.eye(classes)[color_lists.astype(int)], classes

def get_affiliation(club_labels):
    """return the affiation of the karate club members"""
    Mr_Hi, Officer = [], []
    for key, value in club_labels.items():
        if value == 'Mr. Hi':
            Mr_Hi.append(key)
        else:
            Officer.append(key)
    return Mr_Hi, Officer


def fill_diagonal(source_array, diagonal):
    """helps fill element of the source array into a diagonal matrix"""
    copy = source_array.copy()
    np.fill_diagonal(copy, diagonal)
    return copy
```

The original paper on greedy modularity communities maximization could be found here https://journals.aps.org/pre/pdf/10.1103/PhysRevE.70.066111?casa_token=Fqnjw_t-J64AAAAA%3ADmyzj146CDE-UeW_1I6Ifvu40GmCC_goDC4i6IvkYIa9GENKcktHxOgHO5et7Z7xJ3NU1q2Ngt2J6Zs

(https://journals.aps.org/pre/pdf/10.1103/PhysRevE.70.066111?casa_token=Fqnjw_t-
J64AAAAA%3ADmyzj146CDE-

Visualization Helper Function

In [4]:
```
def show_graph(graph,
               label_values_of_nodes,
               label_colors_of_nodes,
               colors_of_edges='black',
               display_window_size=15,
               positions_of_nodes=None,
               cmap='jet'):
    """helps visualize the graph"""
    fig, ax = plt.subplots(figsize=(display_window_size, display_window_size))
    if positions_of_nodes is None:
        # https://networkx.org/documentation/stable/reference/generated/networkx.dra
        positions_of_nodes = ntwx.spring_layout(graph,
                                                k=5/np.sqrt(graph.number_of_nodes()))
        # https://networkx.org/documentation/stable/reference/drawing.html
    ntwx.draw(
        graph,
        positions_of_nodes,
        with_labels=label_values_of_nodes,
        labels=label_values_of_nodes,
        node_color=label_colors_of_nodes,
        ax=ax,
    cmap=cmap,
    edge_color=colors_of_edges)


def plot_training_curves(train_losses, test_losses, accs, grid=False):
    """shows training curves"""
    fig = plt.figure(figsize=(10, 4))
    ax1 = fig.add_subplot(121)
    ax1.plot(np.log10(train_losses), label='train')
    ax1.plot(np.log10(test_losses), label='test')
    ax1.legend()
    if grid:
        ax1.grid()

    ax2 = fig.add_subplot(122)
    ax2.plot(accs, label='acc')
    ax2.set(ylim=[0,1])
    ax2.legend()

    if grid:
        ax2.grid()
```

Gradient Update Helper Function

In [5]:

```
def compute_gradients(param_name, layer, inputs_data, gt_labels, eps=1e-4, weight_d
    """Compute the gradient with respect to a a given parameter in a given layer"""
    gradients_utils = {}
    batch_size = gt_labels.shape[0]
    replica = getattr(layer, param_name).copy()
    replica_flattened = np.asarray(replica).flatten()
    gradients_utils['gradient_values'] = np.zeros(replica_flattened.shape)
    n_parms = replica_flattened.shape[0]
    for ind, param in enumerate(replica_flattened):
        # lower bound cost
        replica_flattened[ind] = param - eps
        temp = replica_flattened.reshape(replica.shape)
        gradients_utils['lower_bound_pred'] = layer.forward_pass(*inputs_data, **{pa
        decay = weight_decay/ 2 * np.sum(replica_flattened ** 2) / batch_size
        lower_cost = np.mean(get_cross_entropy(gradients_utils['lower_bound_pred'],
                                               gt_labels)) + decay
        # upper bound cost
        replica_flattened[ind] = param + eps
        temp = replica_flattened.reshape(replica.shape)
        gradients_utils['upper_bound_pred'] = layer.forward_pass(*inputs_data, **{pa
        decay = weight_decay/ 2 * np.sum(replica_flattened**2) / batch_size

        upper_cost = np.mean(get_cross_entropy(gradients_utils['upper_bound_pred'],
                                               gt_labels)) + decay
        gradients_utils['gradient_values'][ind] = ((upper_cost - lower_cost) / (2 *
        replica_flattened[ind] = param
    return gradients_utils['gradient_values'].reshape(replica.shape)
```

Grad Descent Optimizer Helper Function

In [6]:
```
class Grad_Descent_Optimizer():
    """Performs Gradient Descent"""
    def __init__(self, learning_rate, weight_decay):
        self.learning_rate = learning_rate
        self.weight_decay = weight_decay
        self._y_pred = None
        self._y_true = None
        self._output = None
        self.batch_size = None
        self.nodes_to_be_trained = None

    def __call__(self, y_pred, y_true, nodes_to_be_used_for_training=None):
        self.y_pred = y_pred
        self.y_true = y_true
        self.batch_size = y_pred.shape[0]

        if nodes_to_be_used_for_training is None:
            self.nodes_to_be_used_for_training = np.arange(self.batch_size)
        else:
            self.nodes_to_be_used_for_training = nodes_to_be_used_for_training

    @property
    def out(self, ):
        return self._output

    @out.setter
    def out(self, y):
        self._output = y
```

Set seed

In [7]:
```
set_seed()
```

Implementation Check Helper Function

In [8]:
```
def implementation_check(dW, dW_approx, db, db_approx):
    """This function helps check how correct in your impplementation a layer"""
    try:
        assert normalized_difference_norm(dW, dW_approx) < 1e-7
        assert normalized_difference_norm(db, db_approx) < 1e-7
        print('congrats, your implementation passes the test !!!')
    except:
        print('Not quite there :( yet; your implementation did not pass the test')
```

Training Helper Function

In [9]:
```
def train_test_split(test_nodes, labels):
    return np.array([i for i in range(labels.shape[0]) if i not in test_nodes])

def threshold(arr, threshold_value=0.5):
    """This function treshold the output of a softmax to either be 0 or 1"""
    arr[arr>=threshold_value]=1
    arr[arr<threshold_value]=0
    return arr
```

Node Classification Helper Function

In [10]:
```
class Softmax_Layer():
    """applies a weight multiplication and returns the forward and backward passes s
    so we could use them to compute the gredients.
    Returns: (batch_size, output_dim)"""
    def __init__(self, input_dim, output_dim, name=''):
        self.name = name
        self.cache = {}
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.bias = np.zeros((self.output_dim, 1))
        self.W = get_xavier_init(self.output_dim, self.input_dim)

    def __repr__(self):
        dims = (self.input_dim, self.output_dim)
        if self.name:
            return f"Softmax_Layer: W{'_' + self.name}{dims}"
        else:
            return f"Softmax_Layer: W{'_'+ ''}{dims}"

    def get_softmax(self, x):
        x = x - np.max(x, axis=0, keepdims=True)
        exp_x = np.exp(x)
        return exp_x / np.sum(exp_x, axis=0, keepdims=True)

    def forward_pass(self, X, W=None, bias=None):
        """Returns the softmax of the input X after appliying the weight and biases.
        self.cache['X'] = X.T
        if W is None:
            W = self.W
        if bias is None:
            bias = self.bias

        return self.get_softmax(np.asarray(W @ self.cache['X']) + bias).T # (batch_

    def backward_pass(self, optimizer, need_update=True):
        """mask nodes nodes not revelant for training, and updates optimizer paramet
        training_mask = np.zeros(optimizer.y_pred.shape[0])
        training_mask[optimizer.nodes_to_be_trained] = 1
        training_mask = training_mask.reshape((-1, 1))

        dLoss = np.asarray((optimizer.y_pred - optimizer.y_true))
        dLoss = np.multiply(dLoss, training_mask)

        self.grad = dLoss @ self.W # (batch_size, input_dim)
        optimizer.output = self.grad

        dW = (dLoss.T @ self.cache['X'].T) / optimizer.batch_size # (output_dim, inp
        dbias = np.sum(dLoss.T, axis=1, keepdims=True) / optimizer.batch_size # (ou

        dW_weight_decay = self.W * optimizer.weight_decay / optimizer.batch_size

        if need_update:
            self.W = self.W - (dW + dW_weight_decay) * optimizer.learning_rate
            self.bias = self.bias - dbias.reshape(self.bias.shape) * optimizer.learn

        return dW + dW_weight_decay, dbias.reshape(self.bias.shape)
```

ZKC

We will train a GNN to cluster people in the karate club in such that people who are more likely to associate with either the officer or Mr. Hi will be close together, while the distance beween the 2 classes will be far.

In the original paper titled "Semi-Supervised Classification with Graph Convolutional Networks" that can be found here https://arxiv.org/pdf/1609.02907.pdf (https://arxiv.org/pdf/1609.02907.pdf), the authors framed this as a node-level classification problem on a graph. We will pretend that we only know the affiliation labels for some of the nodes (which we'll call our training set) and we'll predict the affiliation labels for the rest of the nodes (our test set).

We will build a multi-layer Graph Convolutional Network (GCN) with the following layer-wise propagation rule:

$$H^{(l+1)} = \sigma(D^{-1/2}\tilde{A}D^{-1/2}H^{(l)}W^{(l)}) = \sigma(\tilde{A}^{SymNorm}H^{(l)}W^{(l)})$$

where $\tilde{A}$ is the adjacency matrix that we discussed in Homework 4, except that now we add the identity to include self loops at every node for numerical stability. D is the degree matrix as defined in the past, $H^{(l)}$ is the activation matrix of the l-th layer, and W is the weight matrix to be learned. σ is an activation function, in this case tanh.

We used the python module networkx to import the dataset and provided some helper functions to help understand the data as shown below.

In [11]:
```
graph = ntwx.karate_club_graph()
num_nodes, num_edges, club_labels = get_graph_metadata(graph, 'club')
colors, labels, num_classes = get_colors_labels_and_classes(graph, num_nodes)
Mr_Hi_people, Officer_people = get_affiliation(club_labels)
```

Data Inspection

In [12]:
```
print(f'ZKC dataset graph has {num_nodes} nodes and {num_edges} edges')
```

```
ZKC dataset graph has 34 nodes and 78 edges
```

In [13]:
```
print(f'The affiation of each of the 34 members between the officer and Mr. Hi is gi
```

```
The affiation of each of the 34 members between the officer and Mr. Hi is given b
elow:
 {0: 'Mr. Hi', 1: 'Mr. Hi', 2: 'Mr. Hi', 3: 'Mr. Hi', 4: 'Mr. Hi', 5: 'Mr. Hi',
6: 'Mr. Hi', 7: 'Mr. Hi', 8: 'Mr. Hi', 9: 'Officer', 10: 'Mr. Hi', 11: 'Mr. Hi',
12: 'Mr. Hi', 13: 'Mr. Hi', 14: 'Officer', 15: 'Officer', 16: 'Mr. Hi', 17: 'Mr.
Hi', 18: 'Officer', 19: 'Mr. Hi', 20: 'Officer', 21: 'Mr. Hi', 22: 'Officer', 23:
'Officer', 24: 'Officer', 25: 'Officer', 26: 'Officer', 27: 'Officer', 28: 'Offic
er', 29: 'Officer', 30: 'Officer', 31: 'Officer', 32: 'Officer', 33: 'Officer'}
```

In [14]:
```
print(f'nodes/people loyal to Mr. Hi are:\n {Mr_Hi_people}')
```

```
nodes/people loyal to Mr. Hi are:
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 16, 17, 19, 21]
```

In [15]:
```
print(f'nodes/people loyal to the officer are:\n {Officer_people}')
```

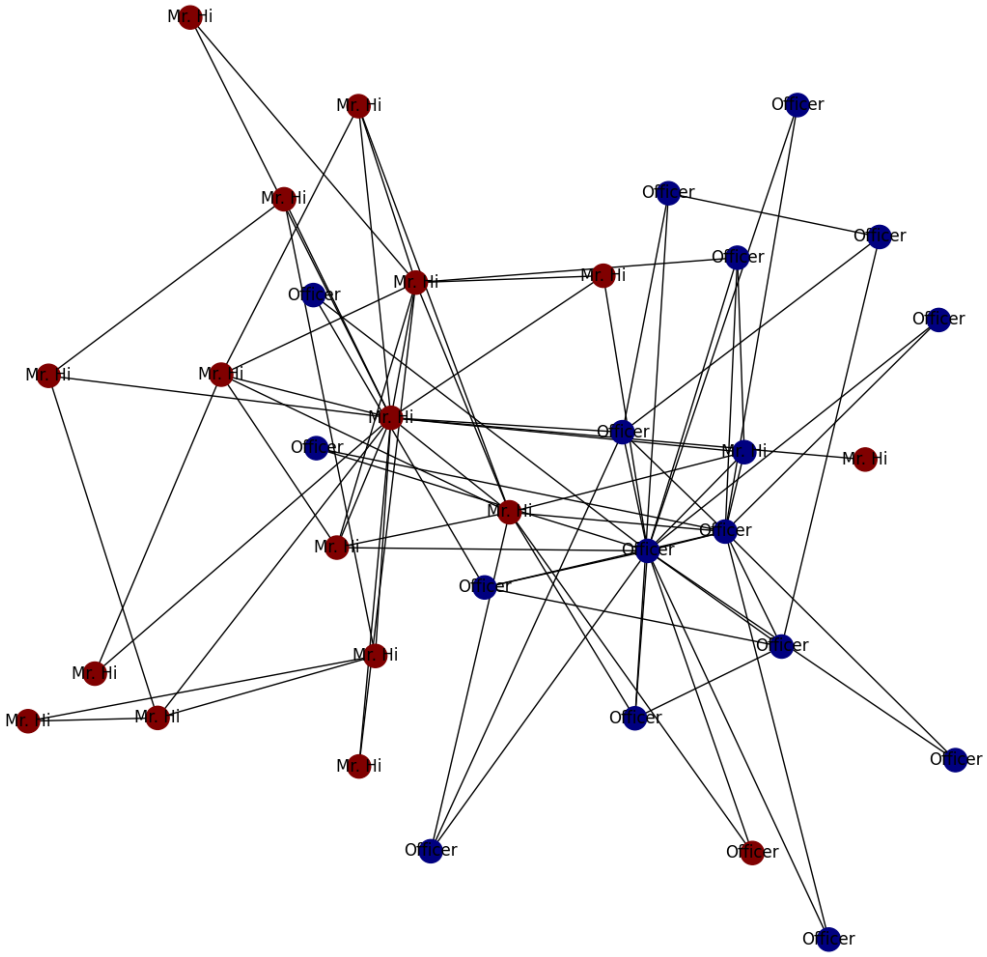
```
nodes/people loyal to the officer are:
 [9, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]
```

In [16]:
```
assert len(Officer_people) + len(Mr_Hi_people) == num_nodes
```

Note that the nodes represent the people and the edges represent the social interaction between the people outside on the club. Also the colors assigned to the nodes as shown below are indepedent of the class since no model has been trained to properly do that.

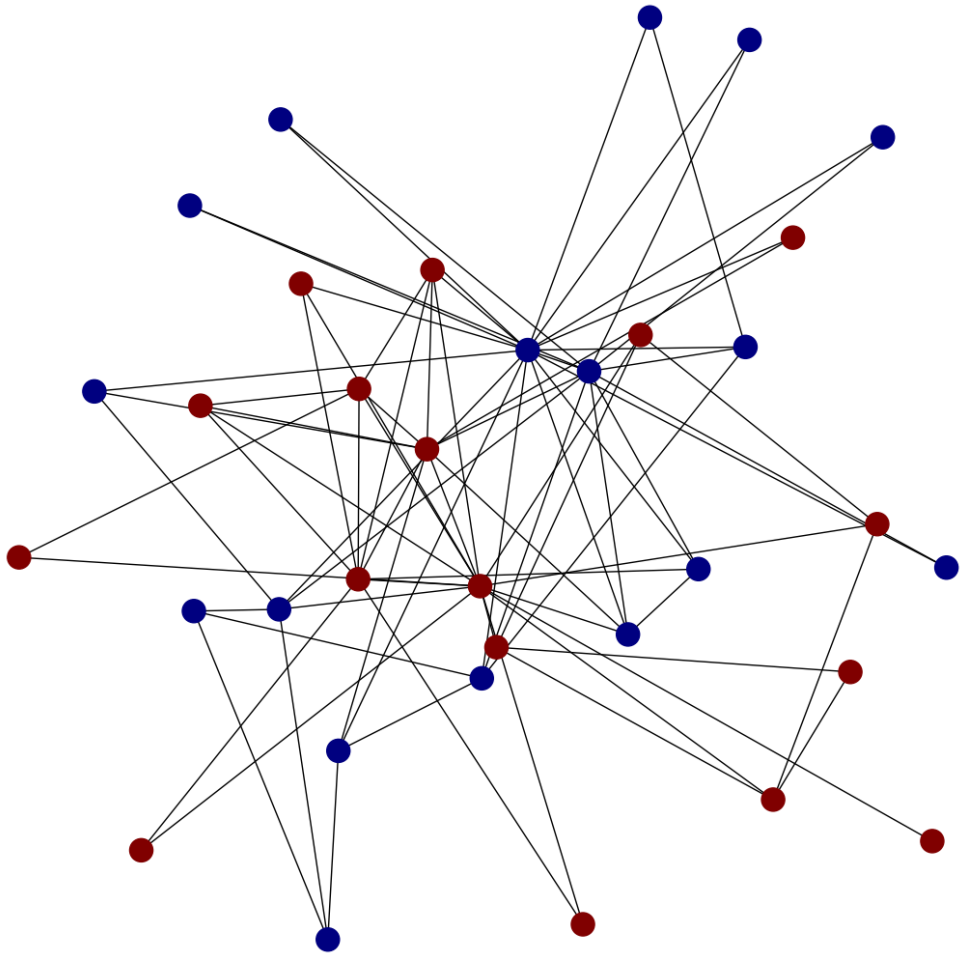
Show Graph with Labels

In [17]:
```
show_graph(graph=graph,
           label_values_of_nodes=club_labels,
           label_colors_of_nodes=colors,
           colors_of_edges='black',
           positions_of_nodes=None)
```



Show Graph without Labels

In [18]:
```
show_graph(graph=graph,
           label_values_of_nodes=None,
           label_colors_of_nodes=colors,
           colors_of_edges='black',
           positions_of_nodes=None,)
```



Note that node labels are obtained using the greedy modurality maximization algorithm descibted in the paper linked above.

Our goal in this problem is therefore to write a simple Graph Neural Network using python to perform node classification. We will also use the node embedding to move nodes with similar classes close to each other. We have provided here the adjacency matrix of the graph.

In [19]:
```
A = ntwx.to_numpy_array(graph)
A
```

Out[19]:
```
array([[0., 1., 1., ..., 1., 0., 0.],
       [1., 0., 1., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 1., 0.],
       ...,
       [1., 0., 0., ..., 0., 1., 1.],
       [0., 0., 1., ..., 1., 0., 1.],
       [0., 0., 0., ..., 1., 1., 0.]])
```

Note that we adjacency matrix does not include the node itself. We want our network to be aware of information about the nodes themselves instead of only the neighborhood, so we add self loops our adjacency matrix. The paper called this $\tilde{A}$.

Q. 1. Compute:

$$\tilde{A} = A_{self-loop} = A + I$$

Where I is the identity matrix which allows us to include self loops of each nodes.

In [21]:
```
np.identity(A.shape[0])
```

Out[21]:
```
array([[1., 0., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 0., 1.]])
```

In [22]:
```
# TODO
A_tild = A + np.identity(A.shape[0])
```

Q.2. Write a function that takes in $\tilde{A}$ as argument and returns the $\tilde{A}^{SymNorm}$ adjacency matrix. You may find the provided function `fill_diagonal` useful, as well as the inverse function `np.linalg.inv` and the matrix square root function `scipy.linalg.sqrtm`.

In [24]:
```
def get_adjacency_matrix(A_tild):

    A_fill = A_tild
    temp = np.diag(np.sum(A_fill, axis=0))
    temp = np.linalg.inv(temp)
    temp = np.sqrt(temp)
    A_tild_symnorm = temp @ A_fill @ temp

    return A_tild_symnorm
```

In [25]:
```
A_hat = get_adjacency_matrix(A_tild)
A_hat
```

Out[25]:
```
array([[0.05882353, 0.0766965 , 0.07312724, ..., 0.09166985, 0.        ,
        0.        ],
       [0.0766965 , 0.1       , 0.09534626, ..., 0.        , 0.        ,
        0.        ],
       [0.07312724, 0.09534626, 0.09090909, ..., 0.        , 0.0836242 ,
        0.        ],
       ...,
       [0.09166985, 0.        , 0.        , ..., 0.14285714, 0.10482848,
        0.08908708],
       [0.        , 0.        , 0.0836242 , ..., 0.10482848, 0.07692308,
        0.06537205],
       [0.        , 0.        , 0.        , ..., 0.08908708, 0.06537205,
        0.05555556]])
```

The other input to our GNN is the graph node matrix X which contains node features. For simplicity, we set X to be the identity matrix because we don't have any node features in this example. In a sense, this will map each node in the graph to a column of learnable parameters in the first layer, resulting in a fully learnable node embedings. In the question below, set the matrix X to be the identity.

Q.3. Generate the feature input matrix X

In [29]:
```
# TODO
X = np.identity(A.shape[0])
```

Single GNN Layer Implementation

We will first implement a single layer GNN. Using the equation provided in the paper that is mentioned above, implement a forward and backward pass for a simple GNN layer.

Note that for *l*=0, H is the input X and $\tilde{A}H$ does the message passing as we have seen in the previous homework and discussion, which is in turn multiplied by a weight matrix W. A non linearity is therefore applied afterward.

In the backward pass, we will apply L2 regularization to the weight matrix W. The regularization term is defined as: $\lambda \sum_{i,j} W_{i,j}^2$ which has gradient: $\lambda 2W$. We will combine $\lambda 2$ in the weight decay parameter optimizer.weight_decay.

Q.4. Complete the #TODO to implement a forward pass that does just that in the class below.

$$H^{(l+1)} = \sigma(D^{-1/2}\tilde{A}D^{-1/2}H^{(l)}W^{(l)}) = \sigma(\tilde{A}^{SymNorm}H^{(l)}W^{(l)})$$

In [26]:

```
class GNN_Layer():
    """process a single GNN layer"""
    def __init__(self, input_dim, output_dim, name=''):
        self.name = name
        self.cache = {}
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.W = get_xavier_init(self.output_dim, self.input_dim)
        self.activation = np.tanh

    def __repr__(self):
        dims = (self.input_dim, self.output_dim)
        if self.name:
            return f"GNN_Layer: W{'_' + self.name}{dims}"
        else:
            return f"GNN_Layer: W{'_'+ ''}{dims}"

    def forward_pass(self, A, X, W=None):
        """A here is the symmetricaly normalized adjacency matrix
        and X is the input to the layer. We cached some values
        to use in the backward pass."""

        self.cache['A'] = A # (batch_size, batch_size)
        # TODO
        self.cache['X'] = X # (batch_size, input_node_feature_dim)

        if W is None:
            W = self.W

        # A : (batch_size, batch_size)
        # X : (batch_size, input_node_feature_dim)
        # W : (hidden_dim, input_dim)

        H = A @ X @ W.T #(batch_size, hidden_dim)
        H = self.activation(H) # [apply activation] (batch_size, hidden_dim)
        self.cache['H'] = H # (batch_size, hidden_dim)
        return H # (batch_size, hidden_dim)

    def backward_pass(self, optimizer, need_update=True):
        # tanh_derivative = 1 - np.tanh(x)**2
        dtanh = 1 - np.square(self.cache['H']) # (batch_size, output_dim)
        # optimizer.output contains the gradient from the next layer. It is multipli
        # so you'll need to divide by optimizer.batch_size to get the average gradie
        d = np.multiply(optimizer.output, dtanh) # (batch_size, output_dim)

        self.grad = (d @ self.W) / optimizer.batch_size  # (batch_size, input_dim)

        optimizer.output = self.grad

        #

        dW = d.T @ self.cache['A'].T @ self.cache['X'] / optimizer.batch_size  # (ou
        dW_weight_decay = optimizer.weight_decay * self.W / optimizer.batch_size

        if need_update: # Use the gradient descent update rule on W. Remember to inc
            self.W = self.W - optimizer.learning_rate * (dW + dW_weight_decay)
```

```
        return dW + dW_weight_decay # (output_dim, input_node_feature_dim)
```

We now test the your implementation

lets's instantiate the GNN_Layer Class and the Softmax_Layer provided in the helper functions to test the gradients.

In [27]:
```
gnn_layer = GNN_Layer(input_dim=num_nodes,
                      output_dim=2,
                      name='gnn_layer_1')

sm_layer = Softmax_Layer(input_dim=2,
                         output_dim=num_classes,
                         name='softmax_layer')

optim = Grad_Descent_Optimizer(learning_rate=0,
                               weight_decay=1.)
```

Q. 5. lets compute the forward passes, uncomment and complete

In [30]:
```
# TODO
gnn_layer_output = gnn_layer.forward_pass(A=A_hat, X=X)

optim(y_pred=sm_layer.forward_pass(X=gnn_layer_output), y_true=labels)
```

lets verify that the layers are properly implemented by looking at the gradients. Note that need update is used only when we are updating the parameters during training. We do not need to do so here since we are just testing our layers.

In [31]:
```
dW_approx = compute_gradients(param_name="W",
                              layer=sm_layer,
                              inputs_data=(gnn_layer_output,),
                              gt_labels=labels,
                              eps=1e-4,
                              weight_decay=optim.weight_decay)

db_approx = compute_gradients(param_name="bias",
                              layer=sm_layer,
                              inputs_data=(gnn_layer_output,),
                              gt_labels=labels,
                              eps=1e-4,
                              weight_decay=optim.weight_decay)
```

In [32]:
```
dW, db = sm_layer.backward_pass(optim, need_update=False)
```

We then get the gradients of Softmax layer.

In [33]:
```
dW, db = sm_layer.backward_pass(optim, need_update=False)
```

We now assert that the true and approximate gradients are very close to each other. If not, something went wrong in our implementation.

In [34]:
```
implementation_check(dW, dW_approx, db, db_approx)
```

```
congrats, your implementation passes the test !!!
```

Q. 6. Now, use the GNN and Softmax layers implemented above to set up our GNN Network.

In [38]:
```
class GNN():
    """This class leverages the GNN layer implemanted above by cascading them into a
    def __init__(self,
                 input_dim,
                 output_dim,
                 hidden_dim,
                 num_layers,):

        self.input_dim = input_dim
        self.output_dim = output_dim
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        self.layers = []
        first_gnn_layer = GNN_Layer(input_dim=self.input_dim,
                                    output_dim=hidden_dim[0],
                                    name='layer_0')

        self.layers.append(first_gnn_layer)

        for layer in range(num_layers - 1):
            gnn_temp = GNN_Layer(input_dim=hidden_dim[layer],
                                 output_dim=hidden_dim[layer + 1],
                                 name=f'layer_{layer}')
            self.layers.append(gnn_temp)


        last_gnn_layer = Softmax_Layer(input_dim=hidden_dim[num_layers-1],
                                output_dim=self.output_dim,
                                name='sm_layer')
        self.layers.append(last_gnn_layer)

    def __repr__(self):
        return '\n'.join([str(layer) for layer in self.layers])

    def embedding(self, A, X):
        H = X
        for layer in self.layers[:-1]:
            H = layer.forward_pass(A, H)

        return H

    def forward_pass(self, A, X):
        H = self.embedding(A, X)
        out = self.layers[-1].forward_pass(H)

        return out
```

Q.7. Let's initialize our model! Uncomment the code bleow and the correct input and output dimensions to initialize the model.

In [39]:
```
# TODO
gnn_model = GNN(
    input_dim=num_nodes,
    output_dim=num_classes,
    num_layers=2,
    hidden_dim=[16, 2],
)
# gnn_model
```

Train/ Test split

We chose nodes 0, 1, 8, 3, 8, 15, 16, 20, 25, 28, and 30 t to be our test nodes and used all the remaining for training.

In [40]:
```
test_nodes = np.array([0, 1, 8, 3, 8, 15, 16, 20, 25, 28, 30])
train_nodes = train_test_split(test_nodes, labels)
```

In [41]:
```
print(f'The nodes that will be used for training are:\n {train_nodes}')
```

```
The nodes that will be used for training are:
 [ 2  4  5  6  7  9 10 11 12 13 14 17 18 19 21 22 23 24 26 27 29 31 32 33]
```

In [42]:
```
print(f'The nodes that will be used for test/val are:\n {test_nodes}')
```

```
The nodes that will be used for test/val are:
 [ 0  1  8  3  8 15 16 20 25 28 30]
```

We now instantiate our optimizer with a learning rate and weight decay for training.

In [43]:
```
training_optim = Grad_Descent_Optimizer(learning_rate=2e-2, weight_decay=2.5e-2)
```

Q.8 Complete the training loop function below. Note that the train loss and test loss are computed over a given set of nodes that is defined by our training and testing set.

In [44]:
```
def train(model,
          A_hat,
          X,
          y_true,
          nodes_to_be_used_for_training,
          nodes_to_be_used_for_testing,
          threshold_value=.5,
          ealy_stopping_steps=60,
          num_epochs=20000):
    """trains our gnn model"""

    accs = []
    training_losses = []
    testing_losses = []
    embeddings = []
    y_preds = []

    minimum_loss = 1e7
    ealy_stopping_counter = 0

    for epoch in range(num_epochs):
        # TODO
        y_pred = model.forward_pass(A_hat, X)
        training_optim(y_pred, y_true, nodes_to_be_used_for_training)

        for layer in reversed(model.layers):
            layer.backward_pass(training_optim, need_update=True)

        embeddings.append(model.embedding(A_hat, X))
        y_preds.append(y_pred)
        acc_temp = (np.argmax(y_pred, axis=1) == np.argmax(y_true, axis=1))[
            [i for i in range(y_true.shape[0]) if i not in nodes_to_be_used_for_tr
        ]
        accs.append(np.mean(acc_temp))

        # loss_temp = ??
        # train_loss_temp = ??
        # test_loss_temp =??

        loss_temp = get_cross_entropy(y_pred, y_true)
        train_loss_temp = np.mean(loss_temp[nodes_to_be_used_for_training])
        test_loss_temp = np.mean(loss_temp[nodes_to_be_used_for_testing])

        training_losses.append(train_loss_temp)
        testing_losses.append(test_loss_temp)

        if test_loss_temp < minimum_loss:
            minimum_loss = test_loss_temp
            ealy_stopping_counter = 0
        else:
            ealy_stopping_counter += 1

        if ealy_stopping_counter > ealy_stopping_steps:
            print("Training Stopped due to Early stopping!")
            break

        if epoch % 100 == 0:
            print(f"epoch #: {epoch+1} \t| train Loss: {train_loss_temp:.3f} \t| tes

    training_losses = np.array(training_losses)
    testing_losses = np.array(testing_losses)
```

```
    y_preds = threshold(np.array(y_preds), threshold_value)
    return training_losses, testing_losses, accs, embeddings, y_preds
```

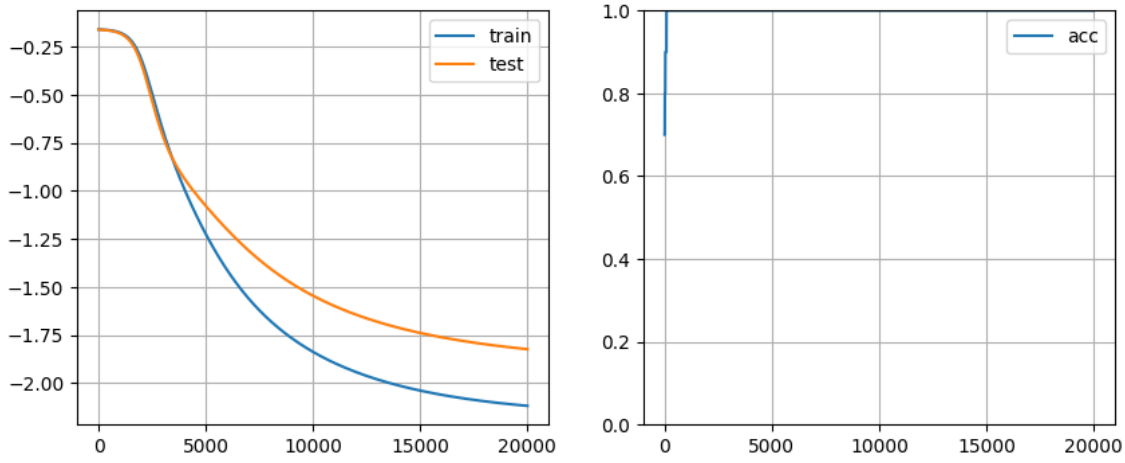
In [45]:
```
training_losses, testing_losses, accs, embeddings, preds = train(model=gnn_model,
                                    A_hat=A_hat,
                                    X=X,
                                    y_true=labels,
                                    nodes_to_be_used_for_training=train_nodes,
                                    nodes_to_be_used_for_testing=test_nodes,
                                    ealy_stopping_steps=50,
                                    num_epochs=20000)
```

```
epoch #: 1      | train Loss: 0.691     | test Loss: 0.689
epoch #: 101    | train Loss: 0.690     | test Loss: 0.688
epoch #: 201    | train Loss: 0.689     | test Loss: 0.687
epoch #: 301    | train Loss: 0.688     | test Loss: 0.686
epoch #: 401    | train Loss: 0.686     | test Loss: 0.684
epoch #: 501    | train Loss: 0.684     | test Loss: 0.681
epoch #: 601    | train Loss: 0.682     | test Loss: 0.678
epoch #: 701    | train Loss: 0.679     | test Loss: 0.675
epoch #: 801    | train Loss: 0.675     | test Loss: 0.670
epoch #: 901    | train Loss: 0.670     | test Loss: 0.665
epoch #: 1001   | train Loss: 0.665     | test Loss: 0.658
epoch #: 1101   | train Loss: 0.657     | test Loss: 0.650
epoch #: 1201   | train Loss: 0.648     | test Loss: 0.639
epoch #: 1301   | train Loss: 0.637     | test Loss: 0.627
epoch #: 1401   | train Loss: 0.624     | test Loss: 0.612
epoch #: 1501   | train Loss: 0.607     | test Loss: 0.593
epoch #: 1601   | train Loss: 0.588     | test Loss: 0.572
epoch #: 1701   | train Loss: 0.566     | test Loss: 0.547
epoch #: 1801   | train Loss: 0.540     | test Loss: 0.519
```

In [46]:
```
print(f'For this toy example, the classification accuracy on the test set is {accs[-
```

For this toy example, the classification accuracy on the test set is 1.0

In [47]:
```
plot_training_curves(training_losses, testing_losses, accs, grid=True)
```



Results

Let's observe the affiliation of people in our test nodes.

In [48]:
```
[club_labels[i] for i in test_nodes]
```

Out[48]:
```
['Mr. Hi',
 'Mr. Hi',
 'Mr. Hi',
 'Mr. Hi',
 'Mr. Hi',
 'Officer',
 'Mr. Hi',
 'Officer',
 'Officer',
 'Officer',
 'Officer']
```

Let's observe the position where our trainined model predict them to be at.

In [49]:
```
embeddings[-1][test_nodes]
```

Out[49]:
```
array([[-0.88604119,  0.07667242],
       [-0.78954779,  0.09252554],
       [ 0.38154912, -0.06526151],
       [-0.82558476,  0.10440181],
       [ 0.38154912, -0.06526151],
       [ 0.65927219, -0.1084209 ],
       [-0.55722073,  0.02027938],
       [ 0.62797918, -0.10975997],
       [ 0.75615907, -0.1749055 ],
       [ 0.37394329, -0.06434665],
       [ 0.49766912, -0.07868414]])
```

Let's observe the GT for values for the test nodes.

In [50]:
```
labels[test_nodes]
```

Out[50]:
```
array([[0., 1.],
       [0., 1.],
       [1., 0.],
       [0., 1.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.]])
```

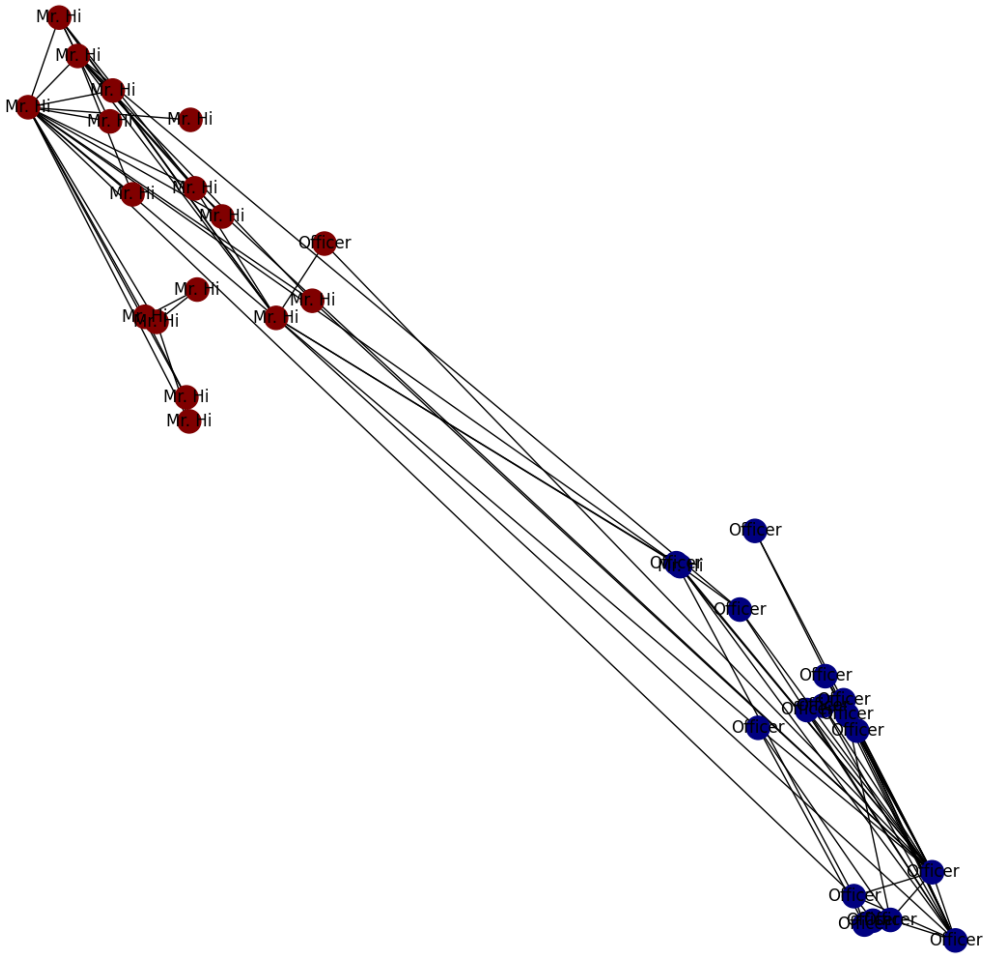
Let's observe the predicted values for them.

In [51]:
```
preds[-1][test_nodes]
```

Out[51]:
```
array([[0., 1.],
       [0., 1.],
       [1., 0.],
       [0., 1.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.]])
```

Finally, lets observe all the data clustered.

In [53]:
```
show_graph(graph=graph,
           label_values_of_nodes=club_labels,
           label_colors_of_nodes=colors,
           colors_of_edges='black',
           positions_of_nodes={i: embeddings[-1][i,:] \
                               for i in range(embeddings[-1].shape[0])},
          )
```



Q. 9. Explain why we obtain a 100% on accuracy on our test set, yet we see in the plot above that 2 samples seem to be misclassified.

In [ ]: