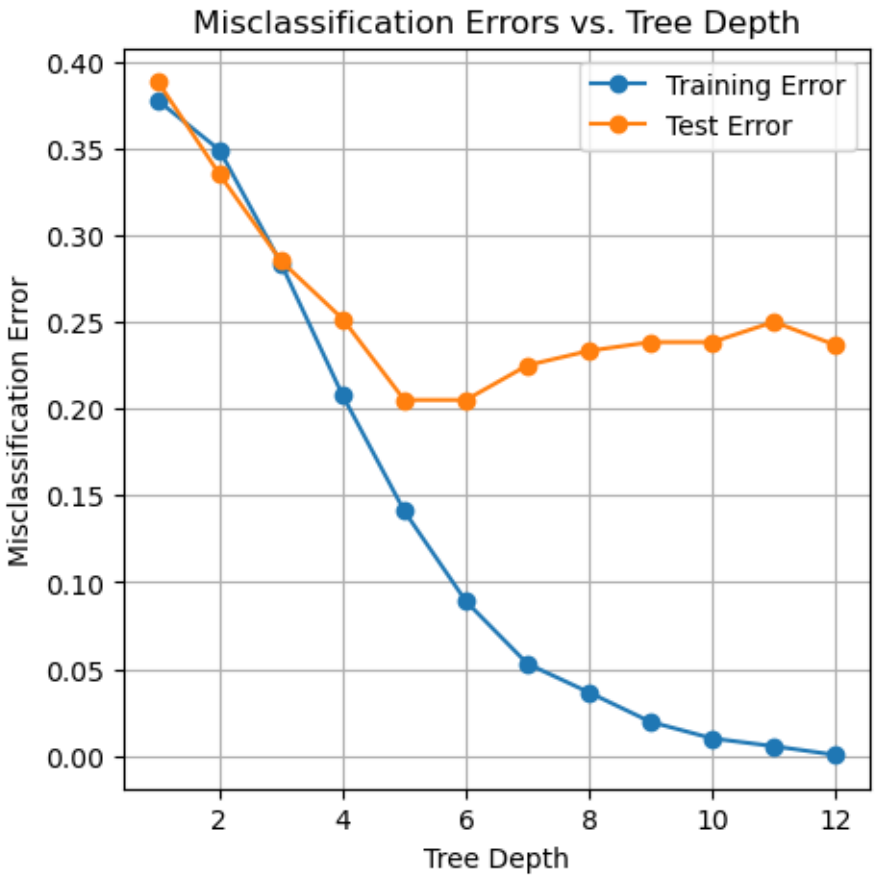
ml-assignment-1

September 15, 2023

[16]:
```
#QN - 1
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

X_madelon_train=np.loadtxt("madelon_train.data")
y_madelon_train=np.loadtxt("madelon_train.labels")
X_madelon_test=np.loadtxt("madelon_valid.data")
y_madelon_test=np.loadtxt("madelon_valid.labels")


madelon_train_errors = []
madelon_test_errors = []


madelon_max_depths = list(range(1, 13))


for depth in madelon_max_depths:
    clf = DecisionTreeClassifier(max_depth=depth, random_state=42)
    clf.fit(X_madelon_train, y_madelon_train)


    train_pred = clf.predict(X_madelon_train)
    test_pred = clf.predict(X_madelon_test)


    train_error = 1.0 - accuracy_score(y_madelon_train, train_pred)
    test_error = 1.0 - accuracy_score(y_madelon_test, test_pred)


    madelon_train_errors.append(train_error)
    madelon_test_errors.append(test_error)


min_test_error = min(madelon_test_errors)
```

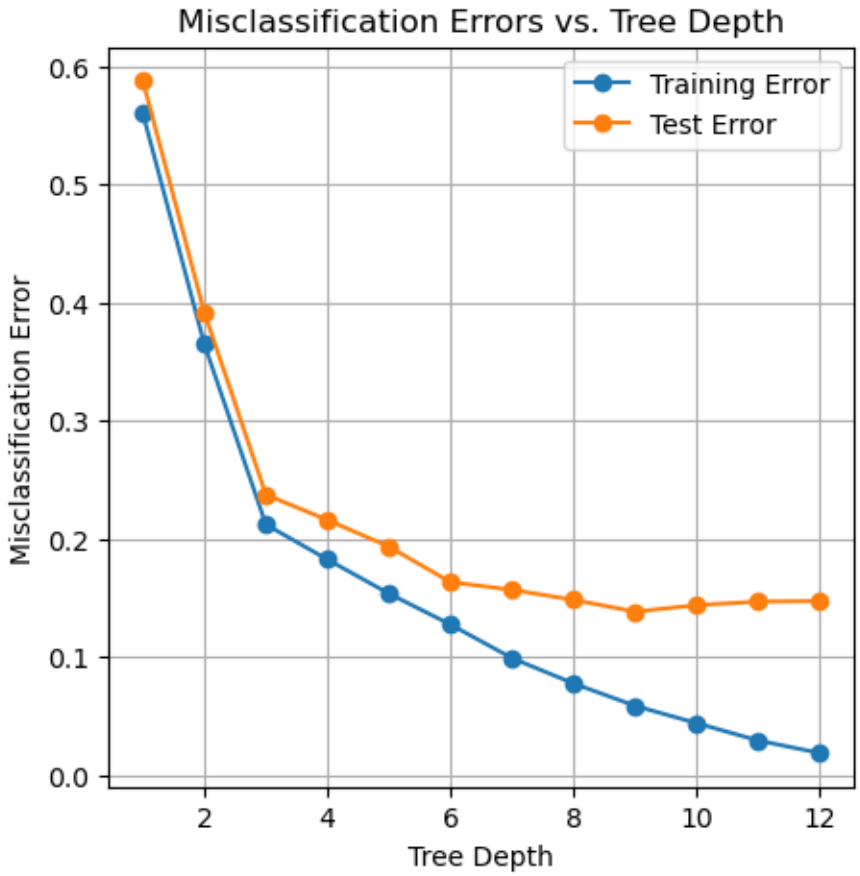
```
best_depth = madelon_max_depths[madelon_test_errors.index(min_test_error)]


plt.figure(figsize=(5, 5))
plt.plot(madelon_max_depths, madelon_train_errors, label='Training Error',␣
 ↪marker='o')
plt.plot(madelon_max_depths, madelon_test_errors, label='Test Error',␣
 ↪marker='o')
plt.xlabel('Tree Depth')
plt.ylabel('Misclassification Error')
plt.title('Misclassification Errors vs. Tree Depth')
plt.legend()
plt.grid(True)


plt.show()
print(f"Minimum Test Error: {min_test_error:.4f}")
print(f"Depth for Minimum Test Error: {best_depth}")
```



```
Minimum Test Error: 0.2050
Depth for Minimum Test Error: 5
```

[11]:
```
# QN - 2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

X_madelon_train=np.loadtxt("X.dat")
y_madelon_train=np.loadtxt("Y.dat")
X_madelon_test=np.loadtxt("Xtest.dat")
y_madelon_test=np.loadtxt("Ytest.dat")


madelon_train_errors = []
madelon_test_errors = []


madelon_max_depths = list(range(1, 13))


for depth in madelon_max_depths:
    clf = DecisionTreeClassifier(max_depth=depth, random_state=42)
    clf.fit(X_madelon_train, y_madelon_train)


    train_pred = clf.predict(X_madelon_train)
    test_pred = clf.predict(X_madelon_test)


    train_error = 1.0 - accuracy_score(y_madelon_train, train_pred)
    test_error = 1.0 - accuracy_score(y_madelon_test, test_pred)


    madelon_train_errors.append(train_error)
    madelon_test_errors.append(test_error)


min_test_error = min(madelon_test_errors)
best_depth = madelon_max_depths[madelon_test_errors.index(min_test_error)]


plt.figure(figsize=(5, 5))
plt.plot(madelon_max_depths, madelon_train_errors, label='Training Error',␣
 ↪marker='o')
```

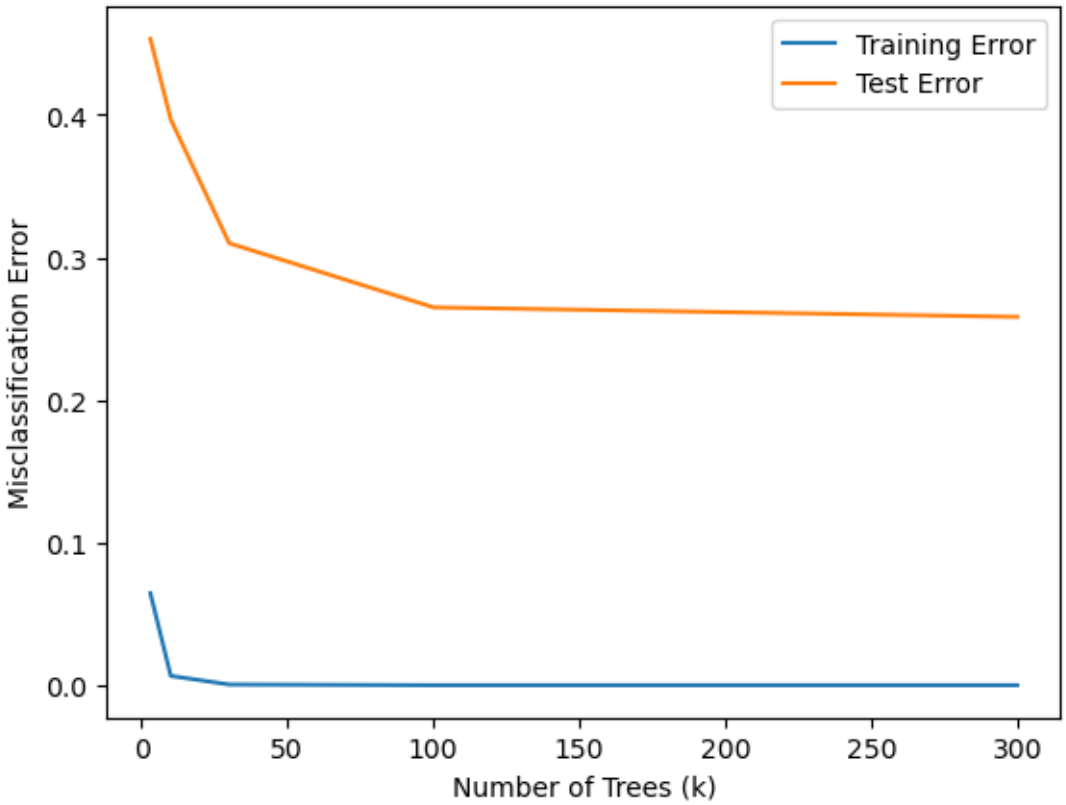
```
plt.plot(madelon_max_depths, madelon_test_errors, label='Test Error',␣
 ↪marker='o')
plt.xlabel('Tree Depth')
plt.ylabel('Misclassification Error')
plt.title('Misclassification Errors vs. Tree Depth')
plt.legend()
plt.grid(True)


plt.show()
print(f"Minimum Test Error: {min_test_error:.4f}")
print(f"Depth for Minimum Test Error: {best_depth}")
```




```
Minimum Test Error: 0.1385
Depth for Minimum Test Error: 9
```

[22]:
```
# QN - 3
import numpy as np
import matplotlib.pyplot as plt
```

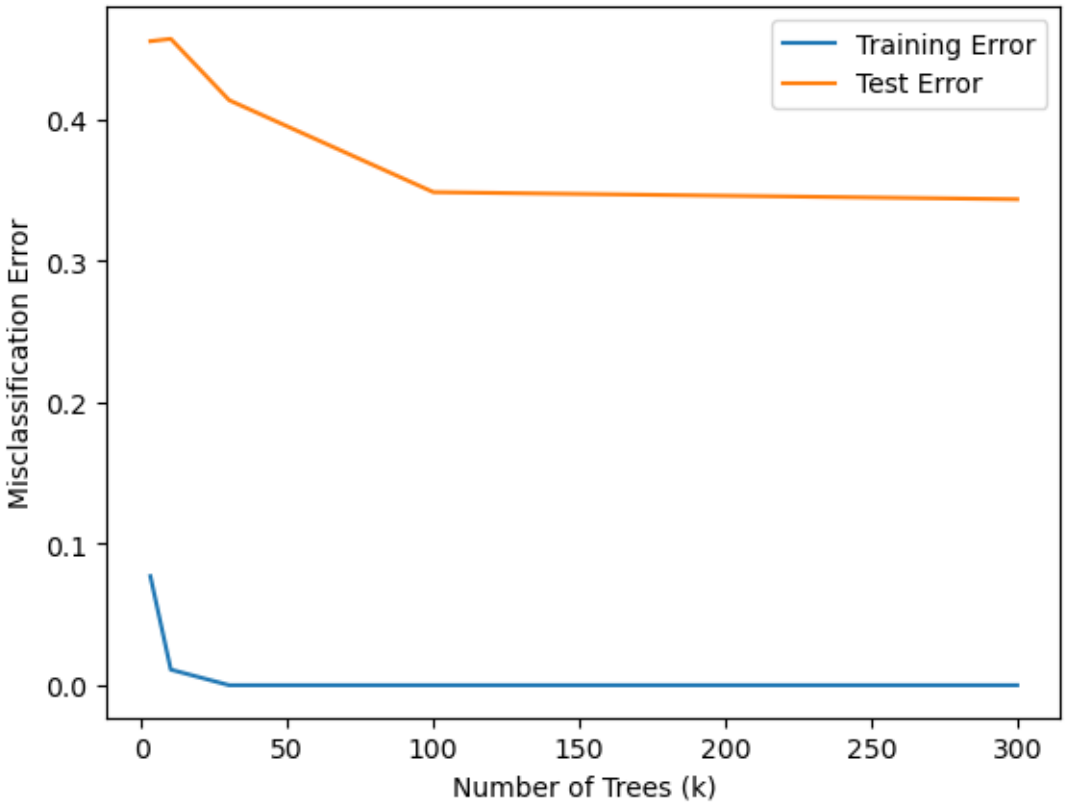
```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score



# Split the dataset into training and test sets (adjust the split ratio)
X_train=np.loadtxt("madelon_train.data")
y_train=np.loadtxt("madelon_train.labels")
X_test=np.loadtxt("madelon_valid.data")
y_test=np.loadtxt("madelon_valid.labels")

# Values of k (number of trees) to experiment with
k_values = [3, 10, 30, 100, 300]

# Initialize arrays to store misclassification errors
train_errors = []
test_errors = []

# Train random forests with different numbers of trees
for k in k_values:
    # Calculate the number of features for random subset ( √500 features)
    clf = RandomForestClassifier(n_estimators=k, max_features=int(np.sqrt(500)))
    clf.fit(X_train, y_train)
    train_pred = clf.predict(X_train)
    test_pred = clf.predict(X_test)
    train_error = 1 - accuracy_score(y_train, train_pred)
    test_error = 1 - accuracy_score(y_test, test_pred)
    train_errors.append(train_error)
    test_errors.append(test_error)

# Plot the misclassification errors vs. number of trees (k)
plt.figure()
plt.plot(k_values, train_errors, label='Training Error')
plt.plot(k_values, test_errors, label='Test Error')
plt.xlabel('Number of Trees (k)')
plt.ylabel('Misclassification Error')
plt.legend()
plt.show()
# Report errors
import pandas as pd

error_table = pd.DataFrame({'Number of Trees (k)': k_values, 'Training Error':␣
 ↪train_errors, 'Test Error': test_errors})
print(error_table)
```



	Number of Trees (k)	Training Error	Test Error
0	3	0.0645	0.453333
1	10	0.0065	0.396667
2	30	0.0005	0.310000
3	100	0.0000	0.265000
4	300	0.0000	0.258333

[18]:
```
# QN - 4
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score



# Split the dataset into training and test sets (adjust the split ratio)
X_train=np.loadtxt("madelon_train.data")
y_train=np.loadtxt("madelon_train.labels")
X_test=np.loadtxt("madelon_valid.data")
```

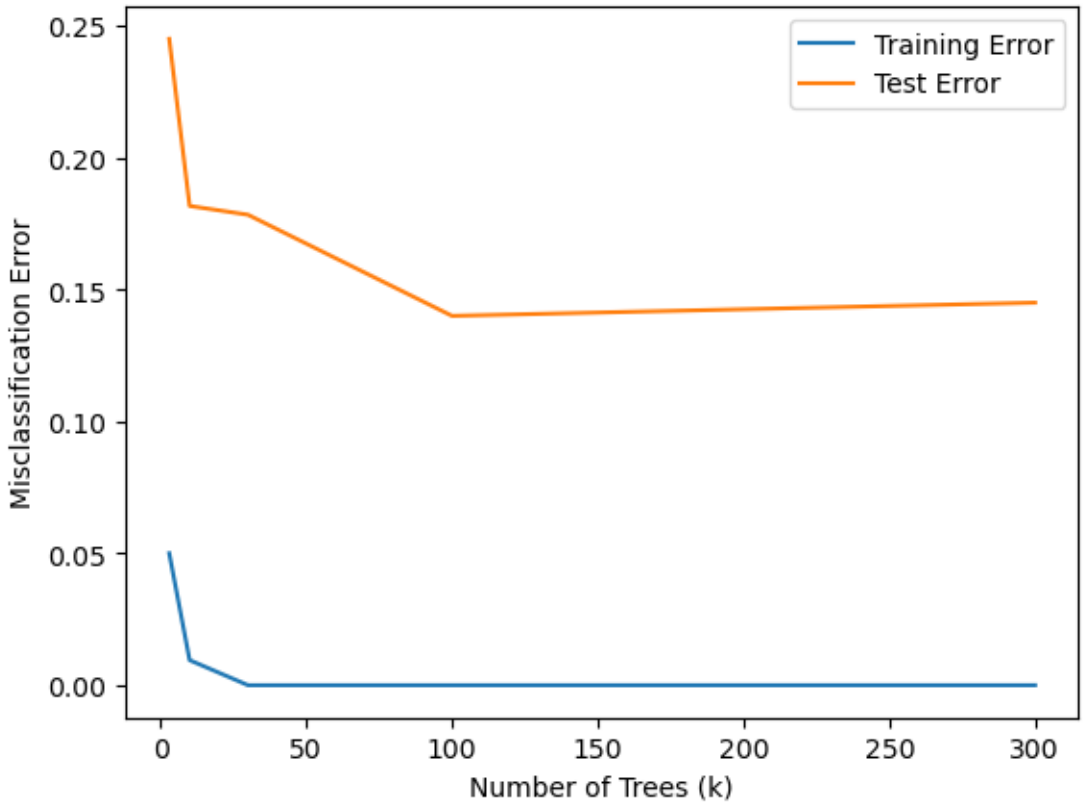
```
y_test=np.loadtxt("madelon_valid.labels")

# Values of k (number of trees) to experiment with
k_values = [3, 10, 30, 100, 300]

# Initialize arrays to store misclassification errors
train_errors = []
test_errors = []

# Train random forests with different numbers of trees
for k in k_values:
    max_features = int(np.log(X_train.shape[1]))
    clf = RandomForestClassifier(n_estimators=k, max_features=max_features,␣
 ↪random_state=42)
    clf.fit(X_train, y_train)
    train_pred = clf.predict(X_train)
    test_pred = clf.predict(X_test)
    train_error = 1 - accuracy_score(y_train, train_pred)
    test_error = 1 - accuracy_score(y_test, test_pred)
    train_errors.append(train_error
    test_errors.append(test_error)

# Plot the misclassification errors vs. number of trees (k)
plt.figure()
plt.plot(k_values, train_errors, label='Training Error')
plt.plot(k_values, test_errors, label='Test Error')
plt.xlabel('Number of Trees (k)')
plt.ylabel('Misclassification Error')
plt.legend()
plt.show()
# Report errors
import pandas as pd

error_table = pd.DataFrame({'Number of Trees (k)': k_values, 'Training Error':␣
 ↪train_errors, 'Test Error': test_errors})
print(error_table)
```



	Number of Trees (k)	Training Error	Test Error
0	3	0.077	0.455000
1	10	0.011	0.456667
2	30	0.000	0.413333
3	100	0.000	0.348333
4	300	0.000	0.343333

[7]:
```
#QN - 5
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

X_train=np.loadtxt("madelon_train.data")
y_train=np.loadtxt("madelon_train.labels")
X_test=np.loadtxt("madelon_valid.data")
y_test=np.loadtxt("madelon_valid.labels")

# Define k values
```

```
k_values = [3, 10, 30, 100, 300]

train_errors = []
test_errors = []

# Train random forest
for k in k_values:
    clf = RandomForestClassifier(n_estimators=k, max_features=None)
    clf.fit(X_train, y_train)
    y_train_pred = clf.predict(X_train)
    y_test_pred = clf.predict(X_test)

    # misclassification errors
    train_error = 1 - accuracy_score(y_train, y_train_pred)
    test_error = 1 - accuracy_score(y_test, y_test_pred)

    train_errors.append(train_error)
    test_errors.append(test_error)

# Plot errors vs. number of trees
plt.figure()
plt.plot(k_values, train_errors, label='Training Error')
plt.plot(k_values, test_errors, label='Test Error')
plt.xlabel('Number of Trees (k)')
plt.ylabel('Misclassification Error')
plt.legend()
plt.show()

# Report errors
import pandas as pd

error_table = pd.DataFrame({'Number of Trees (k)': k_values, 'Training Error':␣
 ↪train_errors, 'Test Error': test_errors})
print(error_table)
```



	Number of Trees (k)	Training Error	Test Error
0	3	0.0500	0.245000
1	10	0.0095	0.181667
2	30	0.0000	0.178333
3	100	0.0000	0.140000
4	300	0.0000	0.145000

assignment-2-ml

September 22, 2023

[5]:
```
#Qn-1 Ass 2
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split


data = pd.read_csv("abalone.csv")
X = data.iloc[:, :7]
y = data.iloc[:, 7]


train_value_mse = []
test_value_mse = []

for i in range(1, 20):
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15,␣
 ↪random_state=None)
Desired_random_state = 42
y_bar = np.mean(y_train)
y_train_pre = np.full_like(y_train, y_bar)
y_test_pre = np.full_like(y_test, y_bar)
train_mse = mean_squared_error(y_train, y_train_pre)
test_mse = mean_squared_error(y_test, y_test_pre)
train_value_mse.append(train_mse)
test_value_mse.append(test_mse)

Average_mse_train = np.mean(train_value_mse)
Average_mse_test = np.mean(test_value_mse)

print(f"Average Training MSE (Null Model): {Average_mse_train}")
print(f"Average Test MSE (Null Model): {Average_mse_test}")
```

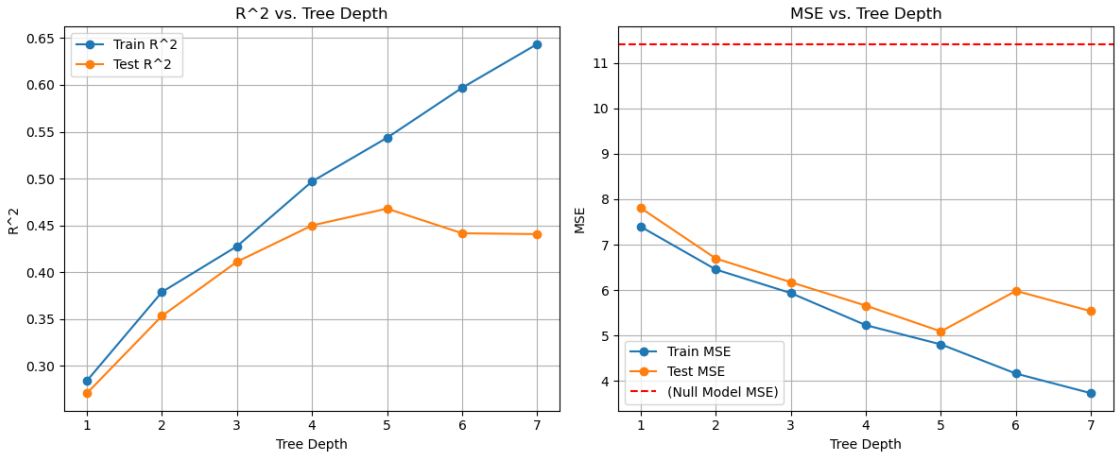
```
Average Training MSE (Null Model): 11.415891800507184
Average Test MSE (Null Model): 10.368421052631579
```

[27]:
```
#Q2 - Ass 2
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error, r2_score


data = pd.read_csv("abalone.csv")
X = data.iloc[:, :7]
y = data.iloc[:, 7]


train_r2_values = []
test_r2_values = []
train_mse_values = []
test_mse_values = []

for i in range(20):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15,␣
 ↪random_state=None)
    model = Ridge(.0001)
    model.fit(X_train, y_train)
    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)


    train_r2 = r2_score(y_train, y_train_pred)
    test_r2 = r2_score(y_test, y_test_pred)
    train_mse = mean_squared_error(y_train, y_train_pred)
    test_mse = mean_squared_error(y_test, y_test_pred)

    train_r2_values.append(train_r2)
    test_r2_values.append(test_r2)
    train_mse_values.append(train_mse)
    test_mse_values.append(test_mse)

average_train_r2 = np.mean(train_r2_values)
average_test_r2 = np.mean(test_r2_values)
average_train_mse = np.mean(train_mse_values)
average_test_mse = np.mean(test_mse_values)

print(f"Average Training R^2: {average_train_r2}")
print(f"Average Test R^2: {average_test_r2}")
print(f"Average Training MSE: {average_train_mse}")
print(f"Average Test MSE: {average_test_mse}")
```

```
Average Training R^2: 0.5292980962569417
Average Test R^2: 0.5134661705756753
Average Training MSE: 4.884065694414113
Average Test MSE: 5.0660220784098255
```

[29]:
```python
#QN-3 Ass 2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, r2_score

data = pd.read_csv("abalone.csv")
X = data.iloc[:, :7]
y = data.iloc[:, 7]
train_r2_values = []
test_r2_values = []
train_mse_values = []
test_mse_values = []

max_depth_values = range(1, 8)
null_model_mse = 11.41

for max_depth in max_depth_values:
    split_train_r2 = []
    split_test_r2 = []
    split_train_mse = []
    split_test_mse = []


    for i in range(1,8):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
 ↪15, random_state=None)


        model = DecisionTreeRegressor(max_depth=max_depth)
        model.fit(X_train, y_train)

        y_train_pred = model.predict(X_train)
        y_test_pred = model.predict(X_test)
        train_r2 = r2_score(y_train, y_train_pred)
        test_r2 = r2_score(y_test, y_test_pred)
        train_mse = mean_squared_error(y_train, y_train_pred)
        test_mse = mean_squared_error(y_test, y_test_pred)


        split_train_r2.append(train_r2)
```

```
        split_test_r2.append(test_r2)
        split_train_mse.append(train_mse)
        split_test_mse.append(test_mse)

    average_train_r2 = np.mean(split_train_r2)
    average_test_r2 = np.mean(split_test_r2)
    average_train_mse = np.mean(split_train_mse)
    average_test_mse = np.mean(split_test_mse)
    train_r2_values.append(average_train_r2)
    test_r2_values.append(average_test_r2)
    train_mse_values.append(average_train_mse)
    test_mse_values.append(average_test_mse)


plt.figure(figsize=(12, 5))

# Plot 1:
plt.subplot(1, 2, 1)
plt.plot(max_depth_values, train_r2_values, label='Train R^2', marker='o')
plt.plot(max_depth_values, test_r2_values, label='Test R^2', marker='o')
plt.xlabel('Tree Depth')
plt.ylabel('R^2')
plt.title('R^2 vs. Tree Depth')
plt.legend()
plt.grid(True)

# Plot 2
plt.subplot(1, 2, 2)
plt.plot(max_depth_values, train_mse_values, label='Train MSE', marker='o')
plt.plot(max_depth_values, test_mse_values, label='Test MSE', marker='o')
plt.axhline(y=null_model_mse, color='r', linestyle='--', label='(Null Model␣
 ↪MSE)')
plt.xlabel('Tree Depth')
plt.ylabel('MSE')
plt.title('MSE vs. Tree Depth')
plt.legend()

plt.tight_layout()
plt.grid(True)
plt.show()
```



[1]:
```
#QN-4 Ass - 2
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score


data = pd.read_csv("abalone.csv")
X = data.iloc[:, :7]
y = data.iloc[:, 7]
train_r2_values = {}
test_r2_values = {}
train_mse_values = {}
test_mse_values = {}


num_splits = 20
n_trees_values = [10, 30, 100, 300]

for n_trees in n_trees_values:
    train_r2_values[n_trees] = []
    test_r2_values[n_trees] = []
    train_mse_values[n_trees] = []
    test_mse_values[n_trees] = []

    for _ in range(num_splits):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
 ↪15, random_state=None)
        model = RandomForestRegressor(n_estimators=n_trees)
        model.fit(X_train, y_train)
```

```
        y_train_pred = model.predict(X_train)
        y_test_pred = model.predict(X_test)
        train_r2 = r2_score(y_train, y_train_pred)
        test_r2 = r2_score(y_test, y_test_pred)
        train_mse = mean_squared_error(y_train, y_train_pred)
        test_mse = mean_squared_error(y_test, y_test_pred)


        train_r2_values[n_trees].append(train_r2)
        test_r2_values[n_trees].append(test_r2)
        train_mse_values[n_trees].append(train_mse)
        test_mse_values[n_trees].append(test_mse)

# Calculate the average training and test R^2 and MSE
average_train_r2 = {n_trees: np.mean(values) for n_trees, values in␣
 ↪train_r2_values.items()}
average_test_r2 = {n_trees: np.mean(values) for n_trees, values in␣
 ↪test_r2_values.items()}
average_train_mse = {n_trees: np.mean(values) for n_trees, values in␣
 ↪train_mse_values.items()}
average_test_mse = {n_trees: np.mean(values) for n_trees, values in␣
 ↪test_mse_values.items()}

# Print  results
for n_trees in n_trees_values:
    print(f"Number of Trees: {n_trees}")
    print(f"Average Training R^2: {average_train_r2[n_trees]}")
    print(f"Average Test R^2: {average_test_r2[n_trees]}")
    print(f"Average Training MSE: {average_train_mse[n_trees]}")
    print(f"Average Test MSE: {average_test_mse[n_trees]}")
```

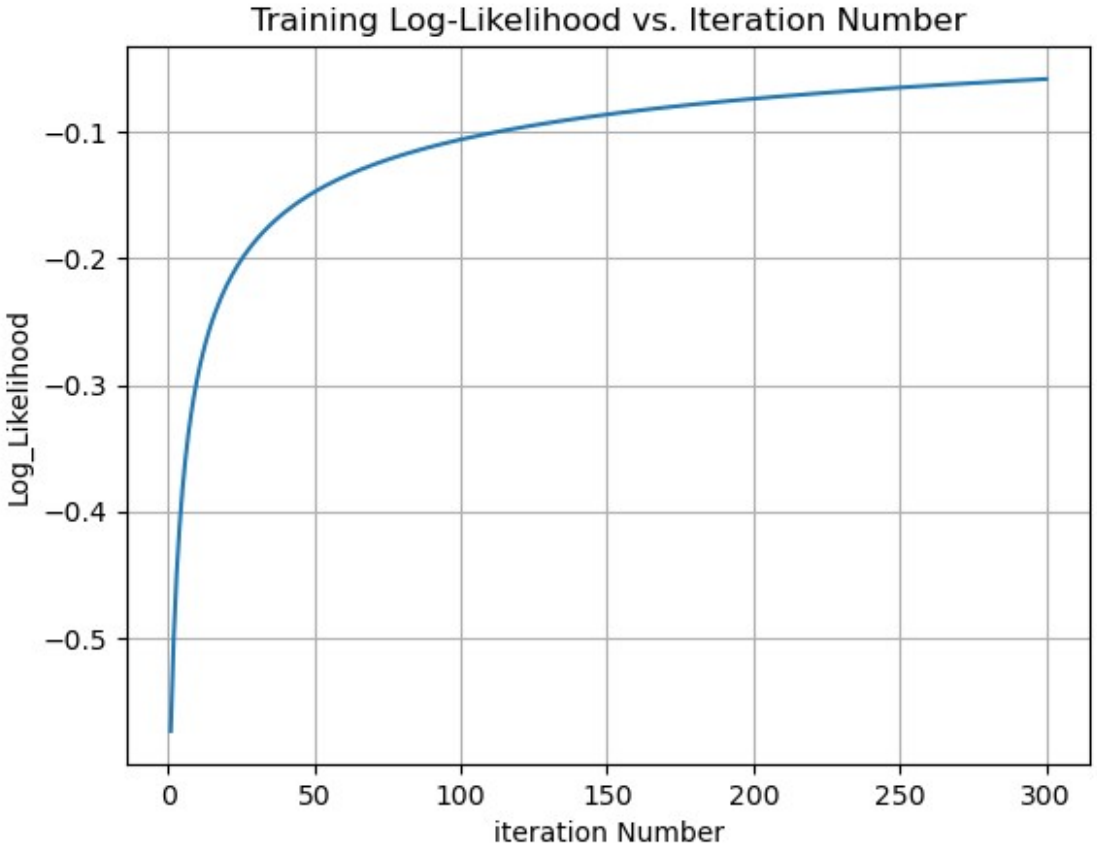
```
Number of Trees: 10
Average Training R^2: 0.9115249291694234
Average Test R^2: 0.49593157790368714
Average Training MSE: 0.9187903634826713
Average Test MSE: 5.226442583732057
Number of Trees: 30
Average Training R^2: 0.9289790026939777
Average Test R^2: 0.5241574285298057
Average Training MSE: 0.7348919100842177
Average Test MSE: 5.036190590111643
Number of Trees: 100
Average Training R^2: 0.9351874293165541
Average Test R^2: 0.539235360081955
Average Training MSE: 0.6762639630881939
Average Test MSE: 4.659414035087719
Number of Trees: 300
```

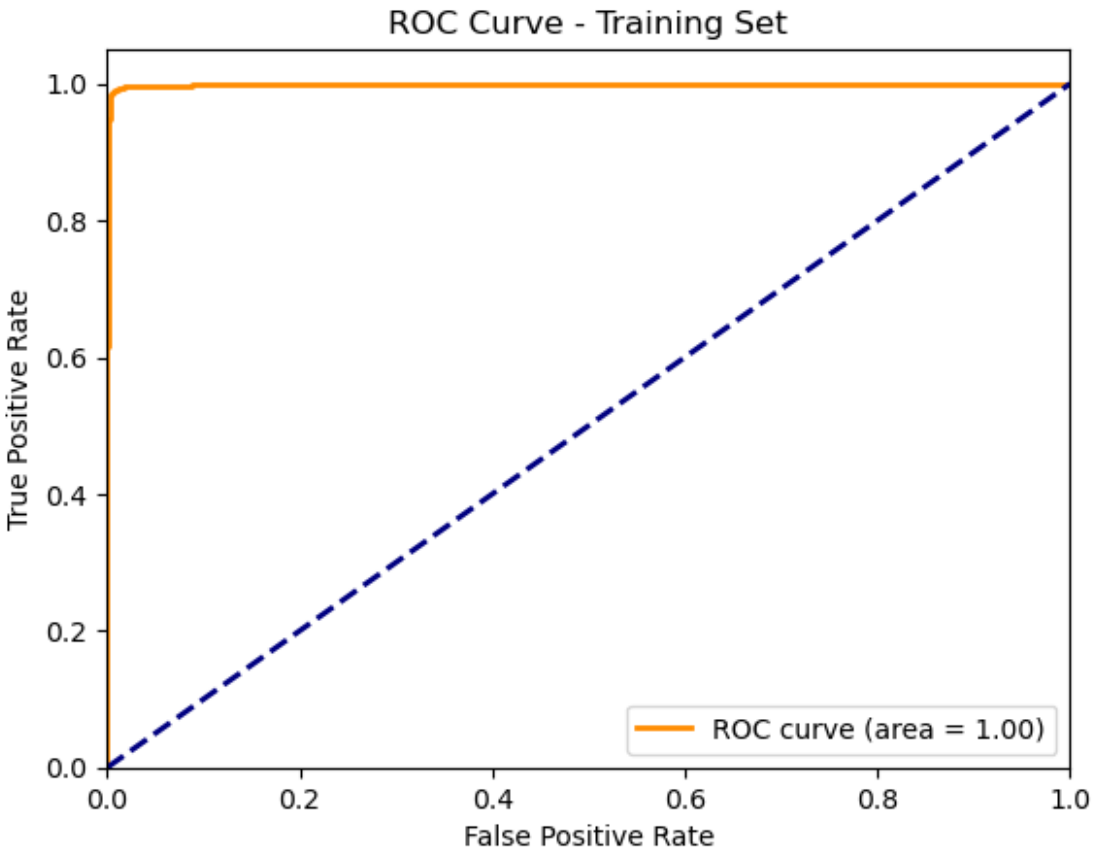
```
Average Training R^2: 0.936779614358402
Average Test R^2: 0.5536244840598114
Average Training MSE: 0.6535232486459409
Average Test MSE: 4.750831537302853
```

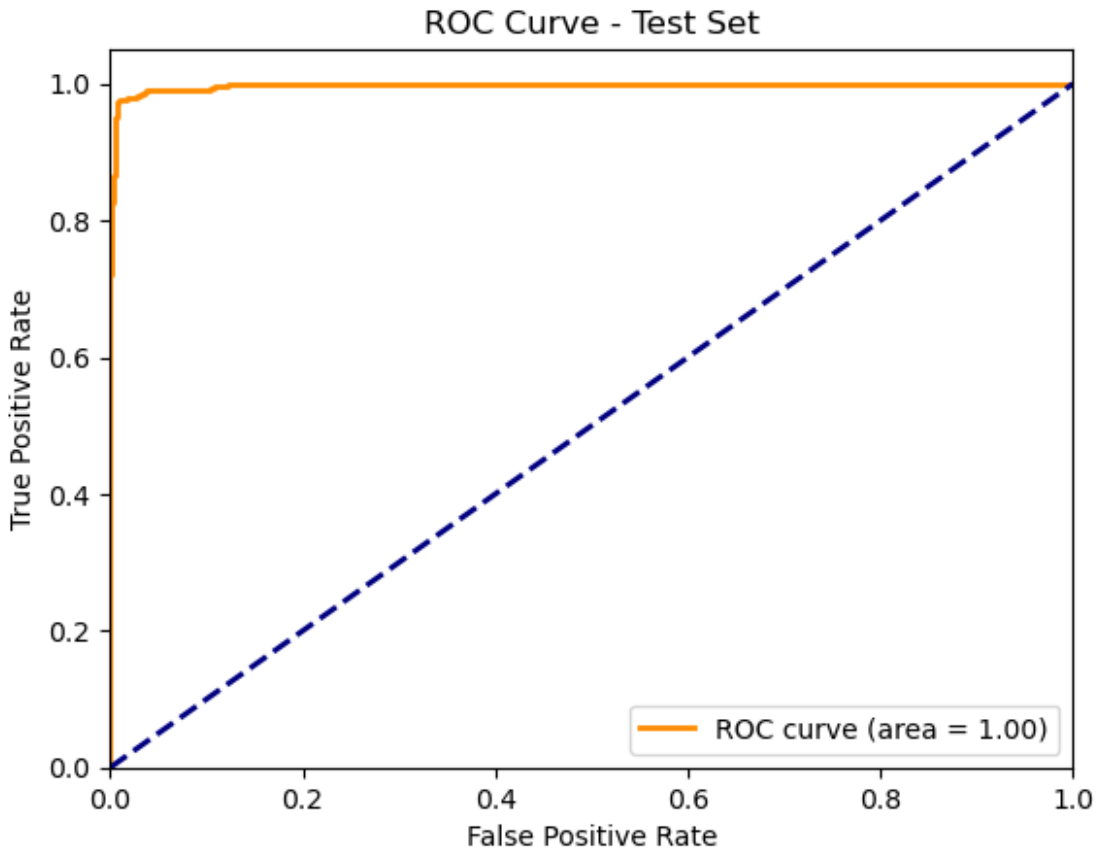
```
# QN - 1

# Here imp required librabries for gis dataset
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, confusion_matrix

# Load the data
A_training =np.loadtxt('gisette_train.data')
B_training =np.loadtxt('gisette_train.labels')
A_testing =np.loadtxt('gisette_valid.data')
B_testing =np.loadtxt('gisette_valid.labels')

# feature normalization
std_val = np.std(A_training, axis = 0)
mask_dataset=(std_val !=0)
A_training=A_training[:, mask_dataset]
m_value=np.mean(A_training, axis=0)
t_value=np.std(A_training, axis=0)

# feature standardization
A_training=(A_training-m_value)/t_value
A_testing=A_testing[:, mask_dataset]
A_testing=(A_testing-m_value)/t_value

A_training=np.insert(A_training, 0, 1, axis=1)
A_testing=np.insert(A_testing, 0, 1, axis=1)

B_training[B_training== -1] = 0
B_testing[B_testing== -1] = 0

#def sigmoid fn
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def log_likelihood(X, y, w_zeros):
    N = len(X)
    y_pred = sigmoid(X.dot(w_zeros))
    return np.sum(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred)) /
N


# def gradient fn
def logistic_regression(X, y, iteration_value, learn_rate_value,
reg_lambda_value):
    N, D = X.shape
    w_zeros = np.zeros(D)
    log_likelihoods = []
```

```
    for i in range(iteration_value):
        y_pred = sigmoid(X.dot(w_zeros))
        grad = X.T.dot(y - y_pred) / N - reg_lambda_value * w_zeros
        w_zeros += learn_rate_value * grad
        ll = log_likelihood(X, y, w_zeros)
        log_likelihoods.append(ll)

    return w_zeros, log_likelihoods

# dec variable
iteration_value = 300
learn_rate_value =0.01
reg_lambda_value = 0.0001


w_zeros, log_likelihoods = logistic_regression(A_training, B_training,
iteration_value, learn_rate_value, reg_lambda_value)

# plot the graph
plt.figure()
plt.plot(range(1, iteration_value + 1), log_likelihoods)
plt.xlabel('iteration Number')
plt.ylabel('Log_Likelihood')
plt.title('Training Log-Likelihood vs. Iteration Number')
plt.grid(True)
plt.show()

def misclassification_error(X, y, w_zeros):
    y_pred = sigmoid(X.dot(w_zeros))
    y_pred_binary = (y_pred > 0.5).astype(int)
    return np.mean(y_pred_binary != y)

training_error = misclassification_error(A_training, B_training,
w_zeros)
testing_error = misclassification_error(A_testing, B_testing, w_zeros)


def plot_roc_curve(X, y, w_zeros, title):
    y_scores = X.dot(w_zeros)
    fpr, tpr, _ = roc_curve(y, y_scores)
    roc_auc = auc(fpr, tpr)
#plot the graphs
    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
(area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
```

```
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()
# plot the curve
plot_roc_curve(A_training, B_training, w_zeros, 'ROC Curve - Training
Set')
plot_roc_curve(A_testing, B_testing, w_zeros, 'ROC Curve - Test Set')

# def the error
training_error = misclassification_error(A_training, B_training,
w_zeros)
testing_error = misclassification_error(A_testing, B_testing, w_zeros)

print(f"Training Misclass Error: {training_error:.4f}")
print(f"Test Misclass Error: {testing_error:.4f}")
```




ROC Curve - Training Set
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 1.00)

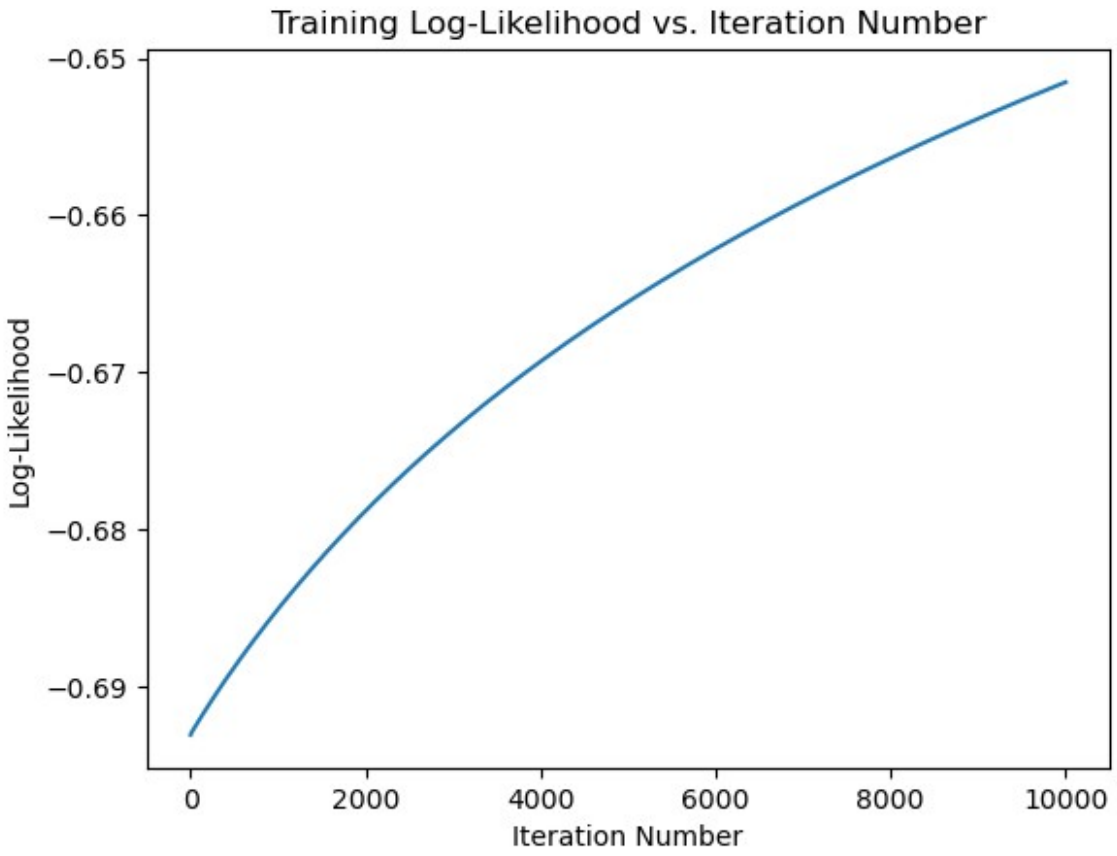



```
Training Misclass Error: 0.0093
Test Misclass Error: 0.0200
```

```
# QN - 1(b)

#imp the lib req
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, confusion_matrix

# import the dataset required
X_train=np.loadtxt('X.dat')
y_train=np.loadtxt('Y.dat')
X_test=np.loadtxt('Xtest.dat')
y_test=np.loadtxt('Ytest.dat')

# feature norm
std=np.std(X_train, axis=0)
mask=(std !=0)

#mean error and stad error cal

X_train=X_train[:, mask]
```

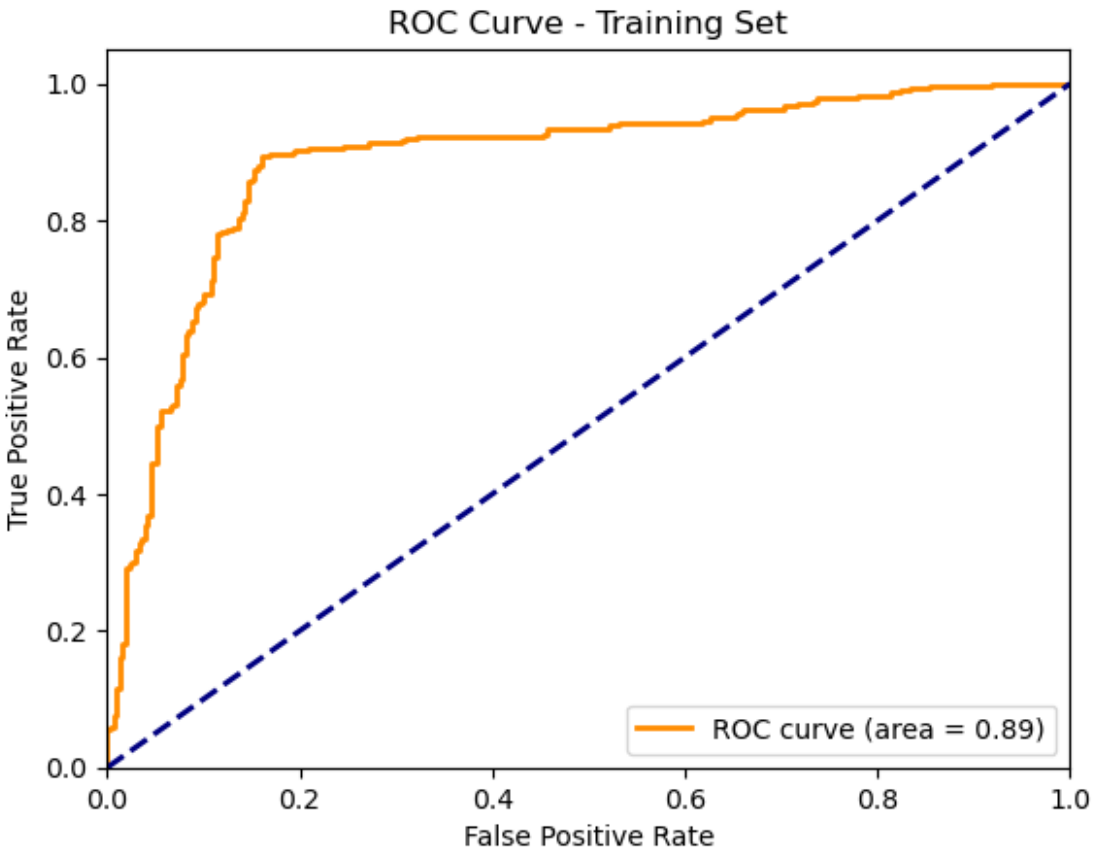
```
mean=np.mean(X_train, axis=0)
true_std=np.std(X_train, axis=0)


# Stand the feature and dif dataset
X_train=(X_train-mean)/true_std
X_test=X_test[:, mask]
X_test=(X_test-mean)/true_std

X_train=np.insert(X_train, 0, 1, axis=1)
X_test=np.insert(X_test, 0, 1, axis=1)

y_train[y_train== -1] = 0
y_test[y_test== -1] = 0


#def sigmoid fn
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# log fn def
def log_likelihood(X, y, w):
    N = len(X)
    y_pred = sigmoid(X.dot(w))
    return np.sum(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred)) /
N

# log_reg defining
def logistic_regression(X, y, iterations, learning_rate, reg_lambda):
    N, D = X.shape
    w = np.zeros(D)
    log_likelihoods = []

    for i in range(iterations):
        y_pred = sigmoid(X.dot(w))
        gradient = X.T.dot(y - y_pred) / N - reg_lambda * w
        w += learning_rate * gradient
        ll = log_likelihood(X, y, w)
        log_likelihoods.append(ll)

    return w, log_likelihoods

# variable dec required
iterations = 10000
learning_rate = 0.01
reg_lambda = 0.0001

w, log_likelihoods = logistic_regression(X_train, y_train, iterations,
learning_rate, reg_lambda)
```

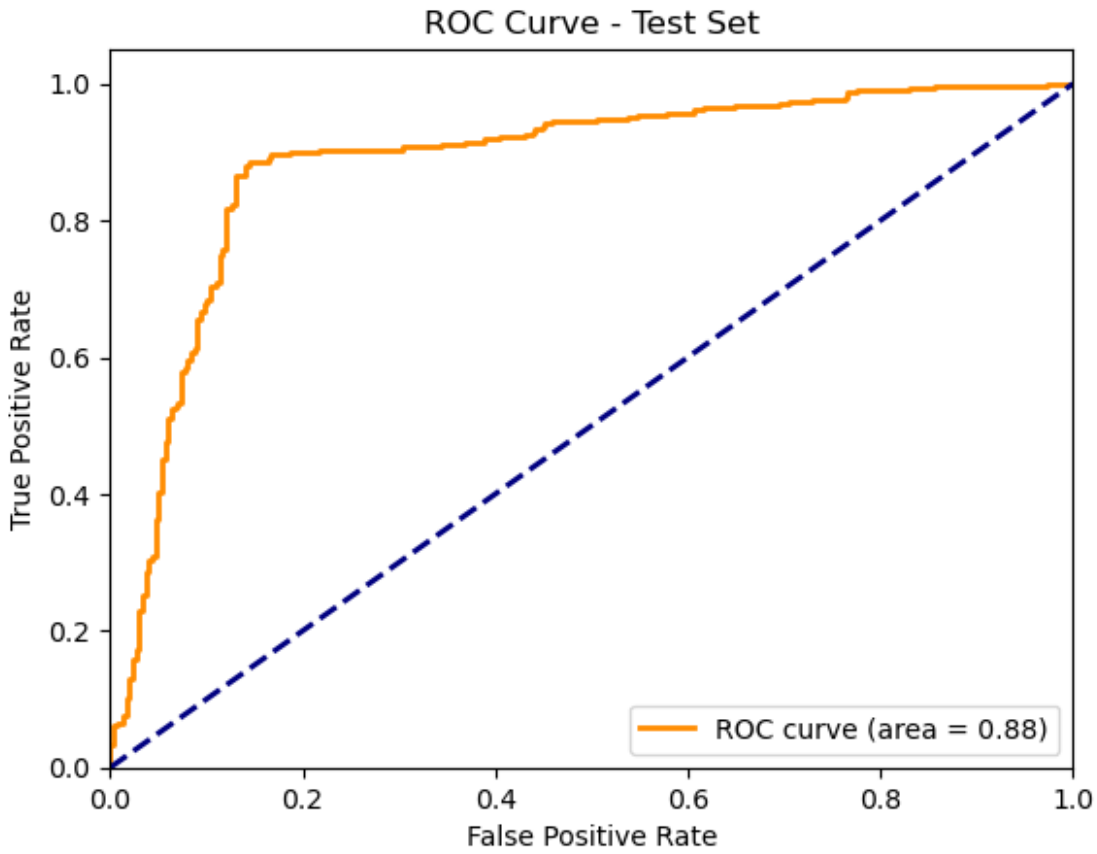
```
#plotting the graph req

plt.figure()
plt.plot(range(1, iterations + 1), log_likelihoods)
plt.xlabel('Iteration Number')
plt.ylabel('Log-Likelihood')
plt.title('Training Log-Likelihood vs. Iteration Number')
plt.show()
```




```
#check for misclass error btw train and test

def misclassification_error(X, y, w):
    y_pred = sigmoid(X.dot(w))
    y_pred_binary = (y_pred > 0.5).astype(int)
    return np.mean(y_pred_binary != y)

train_error = misclassification_error(X_train, y_train, w)
test_error = misclassification_error(X_test, y_test, w)

# printing error
print(f"Training Misclassification Error: {train_error:.4f}")
print(f"Test Misclassification Error: {test_error:.4f}")
```

```
Training Misclassification Error: 0.3647
Test Misclassification Error: 0.3828

# plotting graph for ROC curve
def plot_roc_curve(X, y, w, title):
    y_scores = X.dot(w)
    fpr, tpr, _ = roc_curve(y, y_scores)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
(area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()


plot_roc_curve(X_train, y_train, w, 'ROC Curve - Training Set')
plot_roc_curve(X_test, y_test, w, 'ROC Curve - Test Set')
```

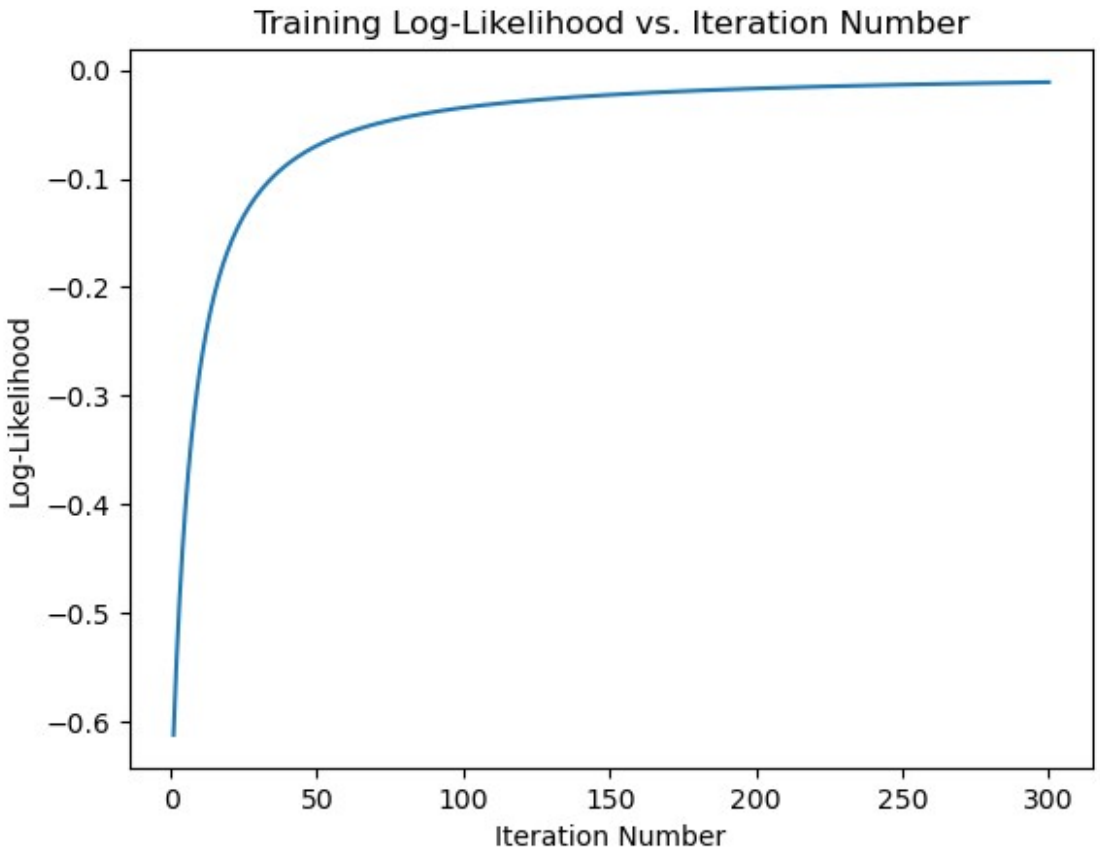

ROC Curve - Training Set
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 0.89)



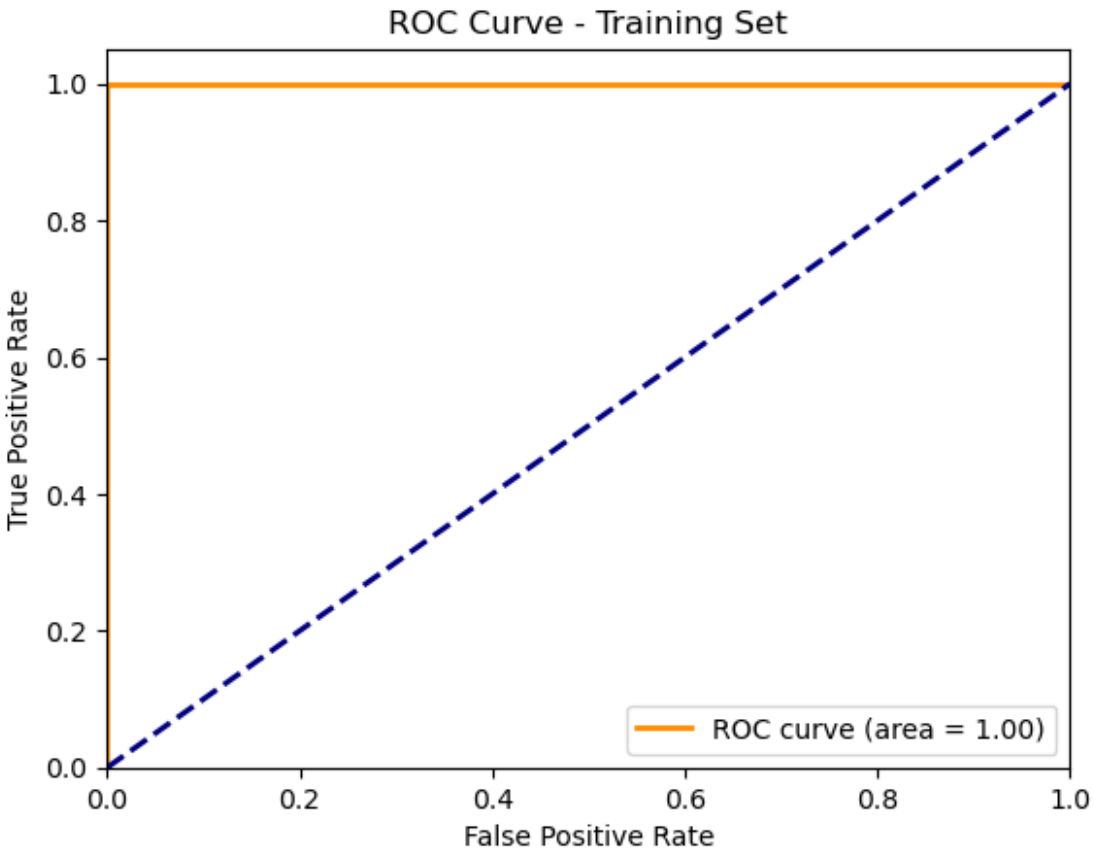

```
# QN - 1 (c)
#imp the lib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, confusion_matrix

#import Dataset of dex
M_training=np.genfromtxt('dexter_train.csv', delimiter=',')
N_training=np.loadtxt('dexter_train.labels')
M_testing=np.genfromtxt('dexter_valid.csv' , delimiter=',')
N_testing=np.loadtxt('dexter_valid.labels')


# Normalize features
std_value=np.std(M_training, axis=0)
data_masking=(std_value !=0)


# calculation of mean value and standard daviation
M_training=M_training[:, data_masking]
mean_value=np.mean(M_training, axis=0)
```

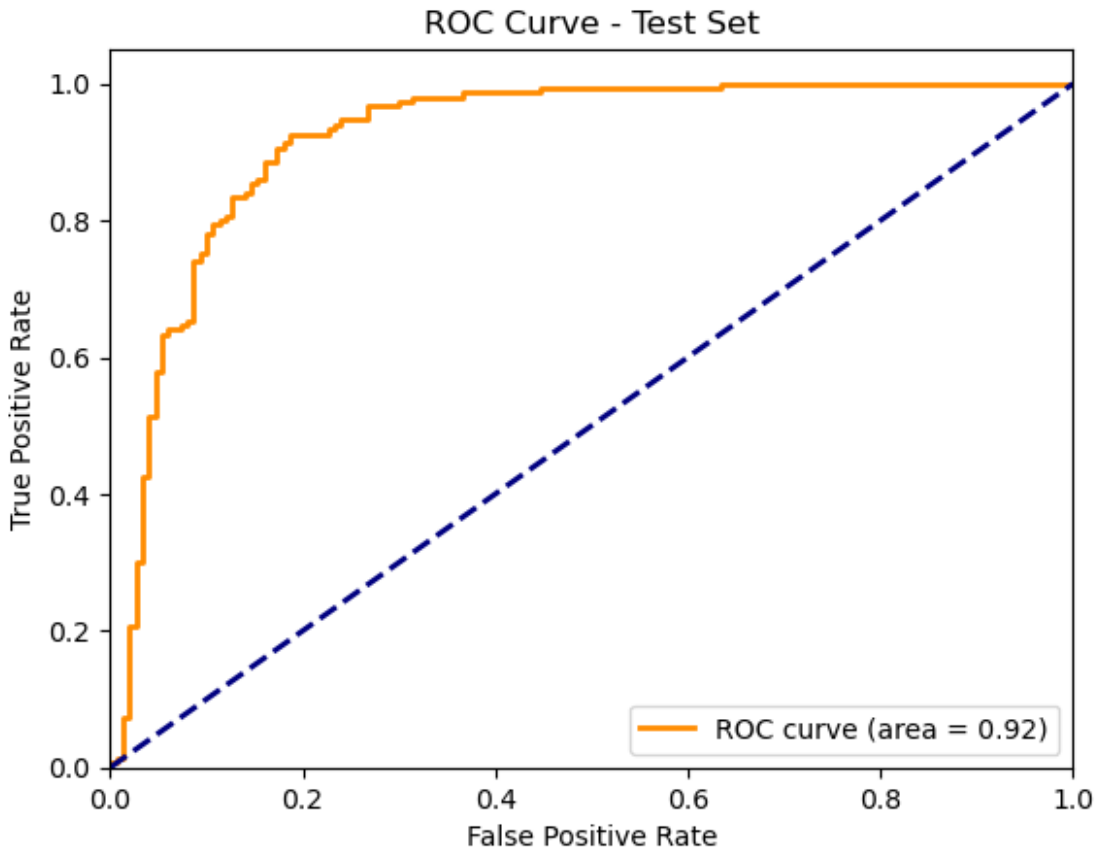
```
t_std_value=np.std(M_training, axis=0)
# Standardize Features
M_training=(M_training-mean_value)/t_std_value
M_testing=M_testing[:, data_masking]
M_testing=(M_testing-mean_value)/t_std_value

M_training=np.insert(M_training, 0, 1, axis=1)
M_testing=np.insert(M_testing, 0, 1, axis=1)

N_training[N_training== -1] = 0
N_testing[N_testing== -1] = 0


# Def fn for sigmoid
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def log_likelihood(X, y, w):
    N = len(X)
    y_pred = sigmoid(X.dot(w))
    return np.sum(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred)) /
N


# Gradient Fn

def logistic_regression(X, y, iteration_value, learning_rate_value,
reg_lambda_value):
    N, D = X.shape
    w = np.zeros(D)
    log_likelihoods = []

    for i in range(iteration_value):
        y_pred = sigmoid(X.dot(w))
        grad_value = X.T.dot(y - y_pred) / N - reg_lambda_value* w
        w += learning_rate_value * grad_value
        ll = log_likelihood(X, y, w)
        log_likelihoods.append(ll)

    return w, log_likelihoods

# declaring variables
iteration_value = 300
learning_rate_value = 0.01
reg_lambda_value= 0.0001

w, log_likelihoods = logistic_regression(M_training, N_training,
iteration_value, learning_rate_value, reg_lambda_value)

# Graphing the plot
```

```
plt.figure()
plt.plot(range(1, iteration_value + 1), log_likelihoods)
plt.xlabel('Iteration Number')
plt.ylabel('Log-Likelihood')
plt.title('Training Log-Likelihood vs. Iteration Number')
plt.show()

def misclassification_error(X, y, w):
    y_pred = sigmoid(X.dot(w))
    y_pred_binary = (y_pred > 0.5).astype(int)
    return np.mean(y_pred_binary != y)

error_training = misclassification_error(M_training, N_training, w)
error_testing = misclassification_error(M_testing, N_testing, w)

print(f"Training Misclass Error: {error_training:.4f}")
print(f"Test Misclass Error: {error_testing:.4f}")

def plot_roc_curve(X, y, w, title):
    y_scores = X.dot(w)
    fpr, tpr, _ = roc_curve(y, y_scores)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
(area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()

plot_roc_curve(M_training, N_training, w, 'ROC Curve - Training Set')
plot_roc_curve(M_testing, N_testing, w, 'ROC Curve - Test Set')
```




```
Training Misclass Error: 0.0000
Test Misclass Error: 0.1400
```

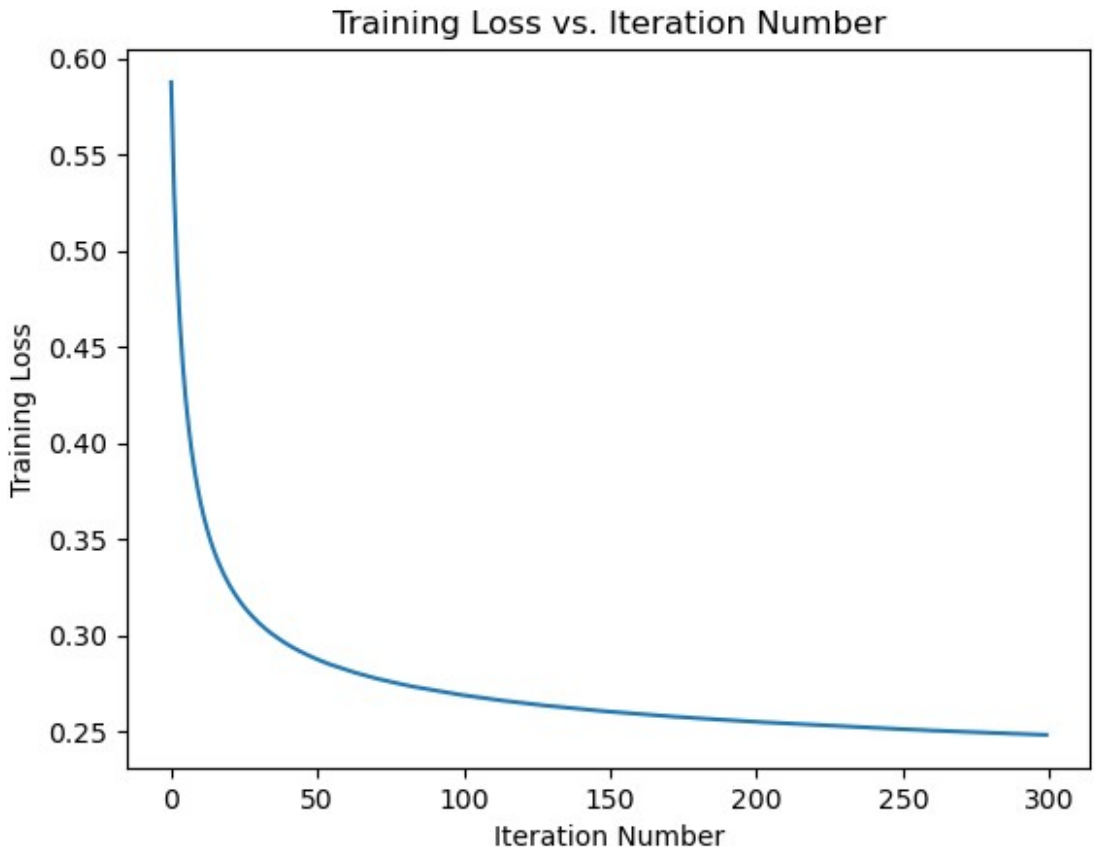

ROC Curve - Training Set
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 1.00)




```
#QNo - 2 (a)

# imp Lib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, confusion_matrix,
accuracy_score, roc_auc_score

# imp Dataset
C_traingis=np.loadtxt('gisette_train.data')
D_traingis=np.loadtxt('gisette_train.labels')
C_testgis=np.loadtxt('gisette_valid.data')
D_testgis=np.loadtxt('gisette_valid.labels')


# feature norm
stdvalue=np.std(C_traingis, axis=0)
masking_dataset=(stdvalue !=0)

# Mean and Stan dev Cal
C_traingis=C_traingis[:, masking_dataset]
mean_val=np.mean(C_traingis, axis=0)
```

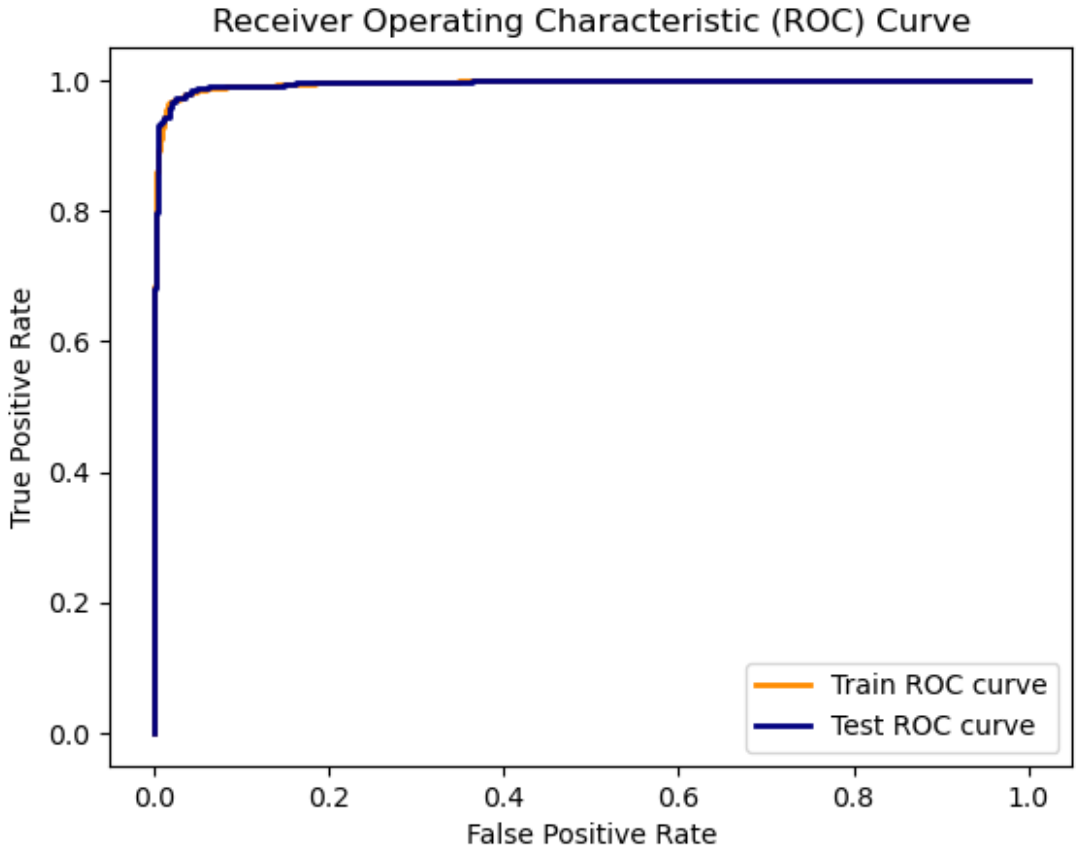
```
t_std_val=np.std(C_traingis, axis=0)

C_traingis=(C_traingis-mean_val)/t_std_val
C_testgis=C_testgis[:, masking_dataset]
C_testgis=(C_testgis-mean_val)/t_std_val


C_traingis=np.insert(C_traingis, 0, 1, axis=1)
C_testgis=np.insert(C_testgis, 0, 1, axis=1)


D_traingis[D_traingis== -1] = 0
D_testgis[D_testgis== -1] = 0
w_zeros = np.zeros(C_traingis.shape[1])

# norm data defining
def normalize_data(C_traingis, C_testgis):
    mean_val = np.mean(C_traingis, axis=0)
    stdvalue = np.std(C_traingis, axis=0)
    C_traingis_norm = (C_traingis - mean_val) / stdvalue
    C_testgis_norm = (C_testgis - mean_val) / stdvalue
    return C_traingis_norm, C_testgis_norm

# Logistic Loss define
def l1_penalized_logistic_loss(X, y, w_zeros, lambda_val):
    N = len(y)
    z = np.dot(X, w_zeros)
    sigmoid = 1 / (1 + np.exp(-z))
    loss = -np.sum(y * np.log(sigmoid) + (1 - y) * np.log(1 -
sigmoid)) / N
    regularization = lambda_val * np.sum(np.abs(w_zeros))
    return loss + regularization

# def grad
def gradient_descent(X, y, w_zeros, lambda_val, learning_rate,
number_iteration_val):
    N = len(y)
    loss_history = []

    for i in range(number_iteration_val):
        z = np.dot(X, w_zeros)
        sigmoid = 1 / (1 + np.exp(-z))
        gradient = np.dot(X.T, (sigmoid - y)) / N + lambda_val *
np.sign(w_zeros)
        w_zeros -= learning_rate * gradient

        # Calculate and store the loss
        loss = l1_penalized_logistic_loss(X, y, w_zeros, lambda_val)
        loss_history.append(loss)
```

```
    return w_zeros, loss_history


# variable dec
number_iteration_val = 300
learning_rate = 0.01
lambda_val = 0.01

w_zeros, loss_history = gradient_descent(C_traingis, D_traingis,
w_zeros, lambda_val, learning_rate, number_iteration_val)



# graping plot
plt.plot(range(number_iteration_val), loss_history)
plt.xlabel('Iteration Number')
plt.ylabel('Training Loss')
plt.title('Training Loss vs. Iteration Number')
plt.show()

# misclass error def

predict_training = (np.dot(C_traingis, w_zeros) > 0).astype(int)
predict_testing = (np.dot(C_testgis, w_zeros) > 0).astype(int)

error_training = 1 - accuracy_score(D_traingis, predict_training)
error_testing = 1 - accuracy_score(D_testgis, predict_testing)

print("Misclass Error on the training Set:", error_training)
print("Misclass Error on the test Set:", error_testing)
```




```
Misclass Error on the training Set: 0.02966666666666662
Misclass Error on the test Set: 0.030000000000000027

# QNo - 2(b)

# def prob

train_probabilities = 1 / (1 + np.exp(-np.dot(C_train, w_zeros)))
test_probabilities = 1 / (1 + np.exp(-np.dot(C_test, w_zeros)))

fpr_train, tpr_train, _ = roc_curve(D_train, train_probabilities)
fpr_test, tpr_test, _ = roc_curve(D_test, test_probabilities)

# plot the graph

plt.figure()
plt.plot(fpr_train, tpr_train, color='darkorange', lw=2, label='Train
ROC curve')
plt.plot(fpr_test, tpr_test, color='navy', lw=2, label='Test ROC
curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
```

```
plt.legend(loc='lower right')
plt.show()
```




```
# QNo - 2(c)

# non- zero ent def
nonzeros = np.sum(w != 0)
values_more_than_lambda = np.sum(np.abs(w) > lambda_val)

# print val
print("Num of Nonzero entries in w:", nonzeros)
print("Num of Val in w satisfy |wi| > λ:", values_more_than_lambda)
```

```
Num of Nonzero entries in w: 7752
Num of Val in w satisfy |wi| > λ: 1806
```

```
# Question No - 1
# first thing first try importing all the relevent libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, roc_curve, auc

# then load the required dataset
gis_trn_dt=np.loadtxt('gisette_train.data')
gis_trn_lbl=np.loadtxt('gisette_train.labels')
gis_val_dt=np.loadtxt('gisette_valid.data')
gis_val_lbl=np.loadtxt('gisette_valid.labels')

# Now will do feature normalization
std= np.std(gis_trn_dt, axis=0)
NonZeroMask = (std!=0)
# add masking for non zero std values
gis_trn_dt=gis_trn_dt[:,NonZeroMask]
mean=np.mean(gis_trn_dt, axis=0)
newStd=np.std(gis_trn_dt, axis=0)
# apply mask for gis dataset to get mean and std val
gis_trn_dt=(gis_trn_dt-mean)/newStd
gis_val_dt=gis_val_dt[:,NonZeroMask]
gis_val_dt=(gis_val_dt-mean)/newStd

# Process dataset by add bias terminology, and convert label
gis_trn_dt=np.insert(gis_trn_dt,0,1,axis=1)
gis_val_dt=np.insert(gis_val_dt,0,1,axis=1)
gis_trn_lbl[gis_trn_lbl==0]=-1
gis_val_lbl[gis_val_lbl==0]=-1

#  Now will check mean & variance val for training and test data set
print("Train mean: ", np.mean(gis_trn_dt))
print("Train variance: ", np.var(gis_trn_dt))
print("Train mean: ", np.mean(gis_val_dt))
print("Train variance: ", np.var(gis_val_dt))

#  parameter initialization as per needs
iteration=100
# threshold val will change a lot
thresholds = [0.1, 0.15, 0.2]
w=np.zeros(gis_trn_dt.shape[1])

trn_missclassif_errors=[]
val_misclassif_errors=[]
trn_missclassif_errors_30=[]
features=[]
fpr_train_list=[]
tpr_train_list=[]
roc_auc_train_list=[]
```

```
fpr_valid_list=[]
tpr_valid_list=[]
roc_auc_valid_list=[]

# Continuously adjust weight val while applying L1 regularization,and
also monitoring and recording both misclass error and the features
that have been chosen or selected.
for lambda_ in thresholds:
    for i in range(iteration):
        # For training data and weight will check dot prod
        dot = np.sum(gis_trn_dt * w, axis=1)
         # adding gradient val
        gradient = np.sum((gis_trn_lbl / ( 1 + np.exp(gis_trn_lbl *
dot))) * (gis_trn_dt).T, axis=1)
        w += gradient * (1 / gis_trn_dt.shape[0])
        w[np.absolute(w) <= lambda_] = 0

        print(i,"weight is: ", np.sum(w != 0))
        dot = np.sum(gis_trn_dt * w, axis=1)

        y_pred_train = ((dot >= 0) == gis_trn_lbl)
        misclass_error_train = 1 - accuracy_score(gis_trn_lbl,
y_pred_train)
        # try diff lambda val
        if(lambda_ == 0.2):
            trn_missclassif_errors_30.append(misclass_error_train)

    feature = np.sum(w != 0)
    features.append(feature)
    print("for lambda of: ",lambda_," feature is: ", feature)

    #Cal misclas error, ROC curve, and AUC score for train and valid
data sets
    trn_missclassif_errors.append(misclass_error_train)

    dot_valid = np.sum(gis_val_dt * w, axis=1)
    y_pred_valid=((dot_valid>=0)==gis_val_lbl)
    misclass_error_valid=1 - accuracy_score(gis_val_lbl,y_pred_valid)
    val_misclassif_errors.append(misclass_error_valid)

    fpr_train, tpr_train, _=roc_curve(gis_trn_lbl,1/(1+ np.exp(-dot)))
    roc_auc_train=auc(fpr_train,tpr_train)
    fpr_train_list.append(fpr_train)
    tpr_train_list.append(tpr_train)
    roc_auc_train_list.append(roc_auc_train)

    fpr_valid, tpr_valid, _=roc_curve(gis_val_lbl,1/(1+ np.exp(-
dot_valid)))
    roc_auc_valid=auc(fpr_valid,tpr_valid)
```

```
    fpr_valid_list.append(fpr_valid)
    tpr_valid_list.append(tpr_valid)
    roc_auc_valid_list.append(roc_auc_valid)

    w=np.zeros_like(w)
# check the trn mis class error and val mis class error
print("Features selected: ",features)
print("Train misclassification errors: ", trn_missclassif_errors)
print("Validation misclassification errors: ", val_misclassif_errors)

#Create and display three subplots: 1) Misclassification Error vs.
Iterations for 30 features, 2) Misclassification Error vs. Selected
Features for both Train and Valid, and 3) ROC curves with AUC scores
for Train and Valid sets
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(range(iteration),trn_missclassif_errors_30,
label="Train",color='blue')
plt.xlabel('Iterations')
plt.ylabel('Miss Class Error')
plt.title('30 Feature: Iterations vs Misclassification Error')
plt.grid()
plt.legend()

plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(features,trn_missclassif_errors,marker="o",label="Train",colo
r='darkslategray')
plt.plot(features,val_misclassif_errors,marker="o",label="Valid",color
='deepskyblue')
plt.xlabel('Features')
plt.ylabel('Miss Class Error')
plt.title('Selected Features vs Misclassification Error')
plt.grid()
plt.legend()

plt.figure(figsize=(12,5))
plt.subplot(1,2,2)
plt.plot(fpr_train_list[-1],tpr_train_list[-
1],color='darkred',lw=4,label=f'Training Set
(AUC={roc_auc_train_list[-1]:.2f})')
plt.plot(fpr_valid_list[-1],tpr_valid_list[-
1],color='darkgreen',lw=2,label=f'Validation Set
(AUC={roc_auc_valid_list[-1]:.2f})')
plt.xlabel('False Positive rate')
plt.ylabel('True Positive rate')
plt.title('Reciever Operating Characterisitcs (ROC) Curve')
plt.grid()
plt.legend()
```

```
Train mean:  0.0002017756255044388
Train variance:  0.9999999592865941
Train mean:  0.0062998654925201245
Train variance:  1.0634350583552006
0 weight is:  338
1 weight is:  269
2 weight is:  218
3 weight is:  178
4 weight is:  156
5 weight is:  146
6 weight is:  144
7 weight is:  133
8 weight is:  129
9 weight is:  122
10 weight is:  115
11 weight is:  112
12 weight is:  110
13 weight is:  105
14 weight is:  102
15 weight is:  99
16 weight is:  96
17 weight is:  95
18 weight is:  94
19 weight is:  93
20 weight is:  92
21 weight is:  92
22 weight is:  92
23 weight is:  92
24 weight is:  92
25 weight is:  92
26 weight is:  90
27 weight is:  88
28 weight is:  87
29 weight is:  86
30 weight is:  86
31 weight is:  86
32 weight is:  84
33 weight is:  83
34 weight is:  83
35 weight is:  81
36 weight is:  80
37 weight is:  80
38 weight is:  79
39 weight is:  79
40 weight is:  79
41 weight is:  79
42 weight is:  78
43 weight is:  76
44 weight is:  76
45 weight is:  76
```

```
46 weight is:  76
47 weight is:  76
48 weight is:  76
49 weight is:  76
50 weight is:  76
51 weight is:  76
52 weight is:  76
53 weight is:  76
54 weight is:  76
55 weight is:  76
56 weight is:  76
57 weight is:  76
58 weight is:  76
59 weight is:  76
60 weight is:  76
61 weight is:  74
62 weight is:  74
63 weight is:  73
64 weight is:  72
65 weight is:  72
66 weight is:  72
67 weight is:  72
68 weight is:  72
69 weight is:  72
70 weight is:  72
71 weight is:  71
72 weight is:  71
73 weight is:  71
74 weight is:  71
75 weight is:  71
76 weight is:  71
77 weight is:  71
78 weight is:  71
79 weight is:  71
80 weight is:  71
81 weight is:  71
82 weight is:  71
83 weight is:  71
84 weight is:  71
85 weight is:  71
86 weight is:  71
87 weight is:  71
88 weight is:  70
89 weight is:  70
90 weight is:  70
91 weight is:  70
92 weight is:  70
93 weight is:  70
94 weight is:  70
95 weight is:  70
```

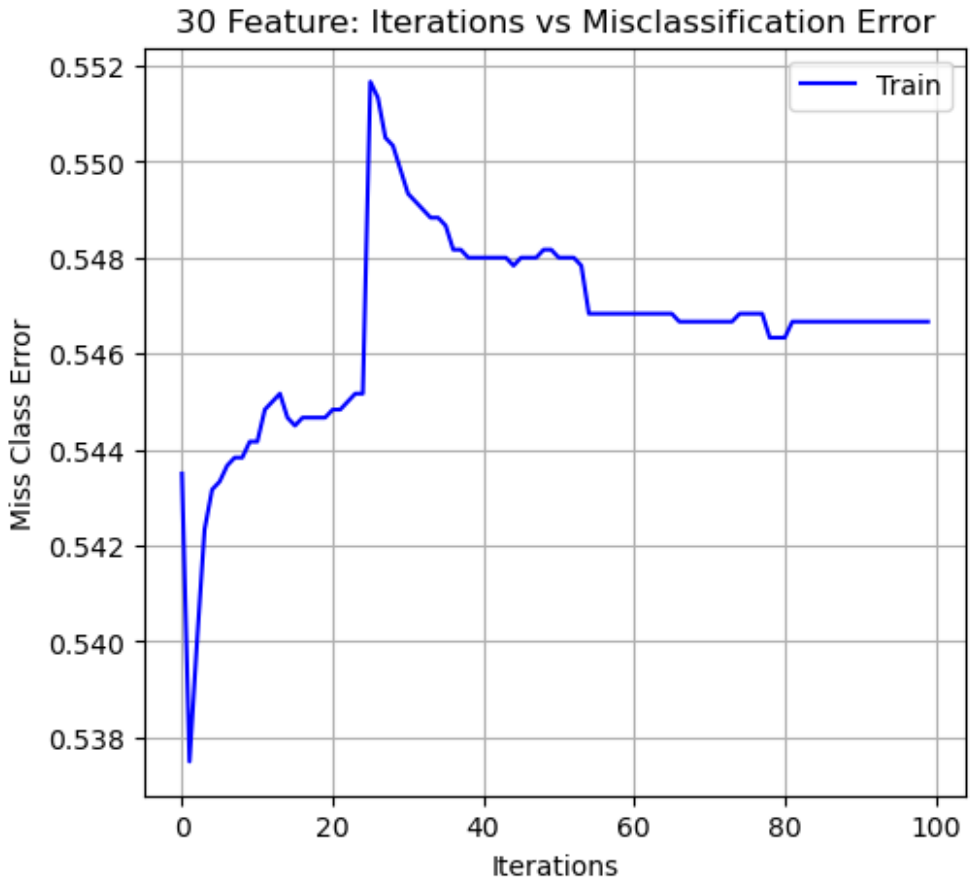
```
96 weight is:  70
97 weight is:  69
98 weight is:  69
99 weight is:  69
for lambda of:  0.1  feature is:  69
0 weight is:  156
1 weight is:  99
2 weight is:  62
3 weight is:  31
4 weight is:  27
5 weight is:  25
6 weight is:  25
7 weight is:  25
8 weight is:  24
9 weight is:  23
10 weight is:  23
11 weight is:  23
12 weight is:  23
13 weight is:  22
14 weight is:  22
15 weight is:  22
16 weight is:  22
17 weight is:  22
18 weight is:  22
19 weight is:  22
20 weight is:  22
21 weight is:  20
22 weight is:  20
23 weight is:  20
24 weight is:  20
25 weight is:  20
26 weight is:  20
27 weight is:  19
28 weight is:  18
29 weight is:  18
30 weight is:  18
31 weight is:  18
32 weight is:  18
33 weight is:  18
34 weight is:  18
35 weight is:  18
36 weight is:  18
37 weight is:  18
38 weight is:  18
39 weight is:  18
40 weight is:  18
41 weight is:  18
42 weight is:  18
43 weight is:  18
44 weight is:  18
```

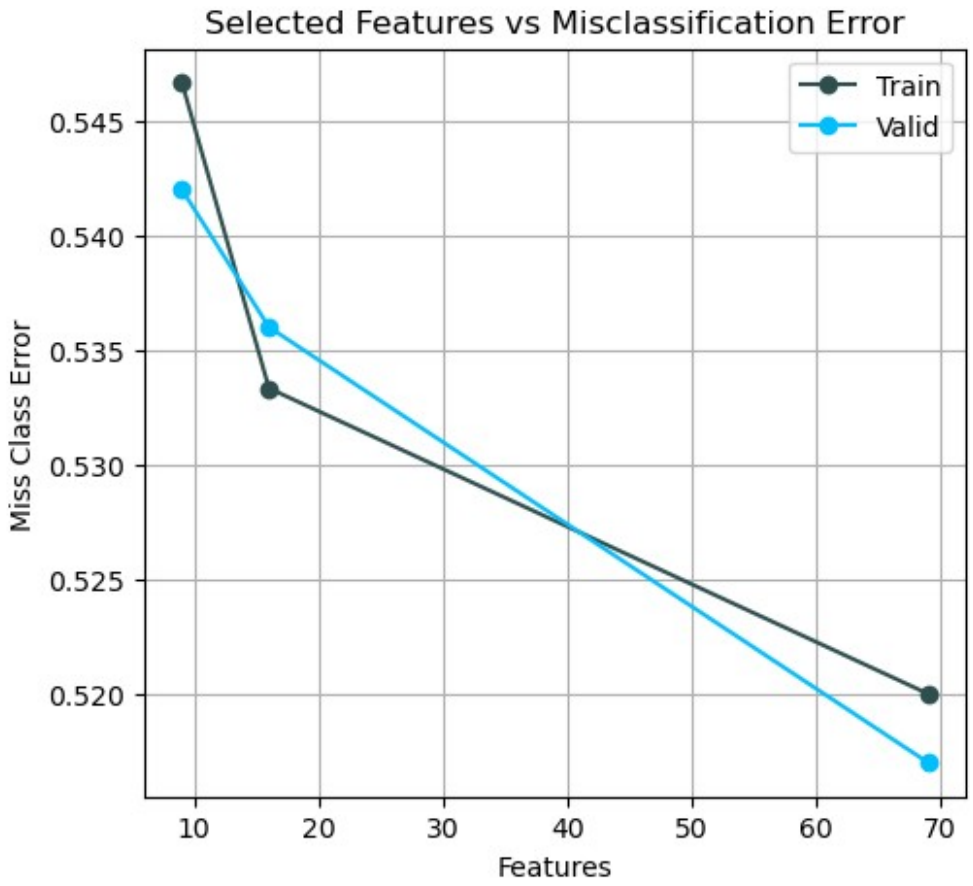
```
45 weight is:  18
46 weight is:  18
47 weight is:  18
48 weight is:  18
49 weight is:  18
50 weight is:  18
51 weight is:  18
52 weight is:  18
53 weight is:  18
54 weight is:  18
55 weight is:  18
56 weight is:  18
57 weight is:  18
58 weight is:  18
59 weight is:  18
60 weight is:  18
61 weight is:  18
62 weight is:  18
63 weight is:  18
64 weight is:  18
65 weight is:  18
66 weight is:  18
67 weight is:  18
68 weight is:  18
69 weight is:  18
70 weight is:  18
71 weight is:  18
72 weight is:  18
73 weight is:  18
74 weight is:  18
75 weight is:  18
76 weight is:  17
77 weight is:  17
78 weight is:  17
79 weight is:  17
80 weight is:  17
81 weight is:  17
82 weight is:  17
83 weight is:  17
84 weight is:  17
85 weight is:  17
86 weight is:  17
87 weight is:  17
88 weight is:  16
89 weight is:  16
90 weight is:  16
91 weight is:  16
92 weight is:  16
93 weight is:  16
94 weight is:  16
```

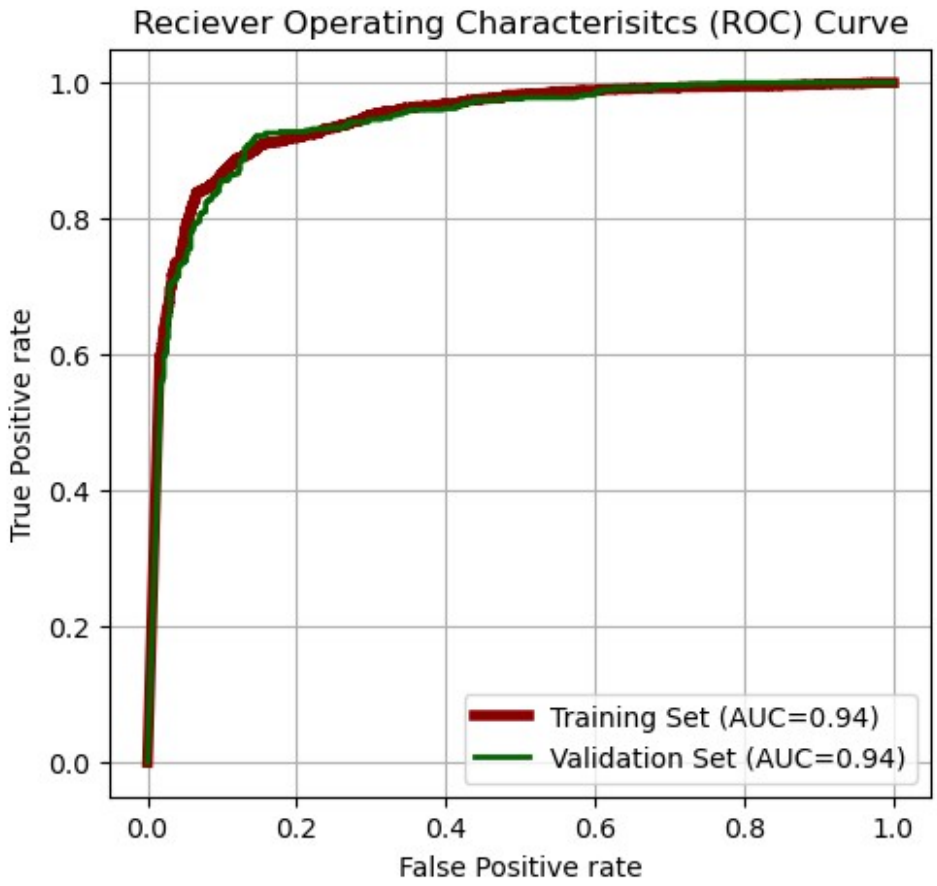
```
95 weight is:  16
96 weight is:  16
97 weight is:  16
98 weight is:  16
99 weight is:  16
for lambda of:  0.15  feature is:  16
0 weight is:  73
1 weight is:  35
2 weight is:  14
3 weight is:  11
4 weight is:  11
5 weight is:  11
6 weight is:  11
7 weight is:  11
8 weight is:  11
9 weight is:  11
10 weight is:  11
11 weight is:  11
12 weight is:  11
13 weight is:  11
14 weight is:  11
15 weight is:  11
16 weight is:  11
17 weight is:  11
18 weight is:  11
19 weight is:  11
20 weight is:  11
21 weight is:  11
22 weight is:  10
23 weight is:  10
24 weight is:  10
25 weight is:  9
26 weight is:  9
27 weight is:  9
28 weight is:  9
29 weight is:  9
30 weight is:  9
31 weight is:  9
32 weight is:  9
33 weight is:  9
34 weight is:  9
35 weight is:  9
36 weight is:  9
37 weight is:  9
38 weight is:  9
39 weight is:  9
40 weight is:  9
41 weight is:  9
42 weight is:  9
43 weight is:  9
```

```
44 weight is:  9
45 weight is:  9
46 weight is:  9
47 weight is:  9
48 weight is:  9
49 weight is:  9
50 weight is:  9
51 weight is:  9
52 weight is:  9
53 weight is:  9
54 weight is:  9
55 weight is:  9
56 weight is:  9
57 weight is:  9
58 weight is:  9
59 weight is:  9
60 weight is:  9
61 weight is:  9
62 weight is:  9
63 weight is:  9
64 weight is:  9
65 weight is:  9
66 weight is:  9
67 weight is:  9
68 weight is:  9
69 weight is:  9
70 weight is:  9
71 weight is:  9
72 weight is:  9
73 weight is:  9
74 weight is:  9
75 weight is:  9
76 weight is:  9
77 weight is:  9
78 weight is:  9
79 weight is:  9
80 weight is:  9
81 weight is:  9
82 weight is:  9
83 weight is:  9
84 weight is:  9
85 weight is:  9
86 weight is:  9
87 weight is:  9
88 weight is:  9
89 weight is:  9
90 weight is:  9
91 weight is:  9
92 weight is:  9
93 weight is:  9
```

```
94 weight is:  9
95 weight is:  9
96 weight is:  9
97 weight is:  9
98 weight is:  9
99 weight is:  9
for lambda of:  0.2  feature is:  9
Features selected:  [69, 16, 9]
Train misclassification errors:  [0.52, 0.5333333333333333,
0.5466666666666666]
Validation misclassification errors:  [0.517, 0.536, 0.542]

<matplotlib.legend.Legend at 0x154f3010a90>
```




Selected Features vs Misclassification Error
Train
Valid
0.545
0.540
0.535
0.530
0.525
0.520
Miss Class Error
10
20
30
40
50
60
70
Features




```
#Create a DataFrame to display the results including lambda values,
selected features, train misclassification errors, and test
misclassification errors
Final_Results = pd.DataFrame({
    'Lambda': thresholds,
    'Features': features,
    'Train Misclass Error': trn_missclassif_errors,
    'Test Misclass Error': val_misclassif_errors
})
print(Final_Results)
```

```
   Lambda  Features  Train Misclass Error  Test Misclass Error
0    0.10        69              0.520000                0.517
1    0.15        16              0.533333                0.536
2    0.20         9              0.546667                0.542
```

```
# Queation No - 2
# add the libraries needed for this data
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, accuracy_score
```

```
# export and add dexter dataset for further cal
dex_train_dt=np.genfromtxt('dexter_train.csv', delimiter=',')
dex_train_lb=np.loadtxt('dexter_train.labels')
dex_test_dt=np.genfromtxt('dexter_valid.csv', delimiter=',')
dex_test_lb=np.loadtxt('dexter_valid.labels')

# now norm the feature
std=np.std(dex_train_dt,axis=0)

# data masking imp and taking mean and var of trn and tst data
data_masking=(std != 0)
dex_train_dt=dex_train_dt[:, data_masking]
val_mean = np.mean(dex_train_dt,axis=0)
std_true_val = np.std(dex_train_dt,axis=0)

dex_train_dt = (dex_train_dt-val_mean)/std_true_val
dex_test_dt = dex_test_dt[:,data_masking]
dex_test_dt = (dex_test_dt-val_mean)/std_true_val

dex_train_dt = np.insert(dex_train_dt,0,1, axis=1)
dex_test_dt = np.insert(dex_test_dt,0,1, axis=1)

dex_train_lb[dex_train_lb == 0] = -1
dex_test_lb[dex_test_lb == 0] = -1

# param initialize as per need
iteraions_val = 100
# give threshold val will chg a lot
val_threshold = [0.141, 0.098, 0.0712, 0.0523, 0.0468]
w = np.zeros(dex_train_dt.shape[1])

# initilize vals
train_misclass_errors=[]
valid_misclass_errors=[]
train_misclass_errors_30=[]
features=[]
fpr_train_list=[]
tpr_train_list=[]
roc_auc_train_list=[]
fpr_valid_list=[]
tpr_valid_list=[]
roc_auc_valid_list=[]

# Continuously adjust weight val while applying L1 regularization,and
also monitoring and recording both misclass error and
for val_lambda in val_threshold:
    for i in range(iteraions_val):
        # For training data and weight will check dot prod
        dot=np.sum(dex_train_dt * w, axis=1)
        # do grad
```

```
        gradient=np.sum((dex_train_lb/
(1+np.exp(dex_train_lb*dot)))*(dex_train_dt).T, axis=1)

        w += gradient * (1/ dex_train_dt.shape[0])
        w[np.absolute(w) <= val_lambda]=0

        print(i, "weight is:",np.sum(w !=0))

        dot = np.sum(dex_train_dt*w, axis=1)

        y_pred_train = ((dot>=0)== dex_train_lb)
        misclass_error_train = 1 - accuracy_score(dex_train_lb,
y_pred_train)
        # try diff lambda val
        if(val_lambda ==  0.141):
            train_misclass_errors_30.append(misclass_error_train)
            # diff feature selection
    feature = np.sum(w != 0)
    features.append(feature)
    # pt lam val and ft
    print('for lambda of :', val_lambda, 'features is:', feature)
    train_misclass_errors.append(misclass_error_train)

    dot_valid = np.sum(dex_test_dt * w, axis=1)
    y_pred_valid = ((dot_valid >= 0)== dex_test_lb)
    misclass_error_valid = 1 - accuracy_score(dex_test_lb,
y_pred_valid)
    valid_misclass_errors.append(misclass_error_valid)

    fpr_train, tpr_train, _ = roc_curve(dex_train_lb, 1/ (1+ np.exp(-
dot)))
    roc_auc_train = auc(fpr_train, tpr_train)
    fpr_train_list.append(fpr_train)
    tpr_train_list.append(tpr_train)
    roc_auc_train_list.append(roc_auc_train)

    fpr_valid, tpr_valid, _ = roc_curve(dex_test_lb, 1/(1 + np.exp(-
dot_valid)))
    roc_auc_valid = auc(fpr_valid, tpr_valid)
    fpr_valid_list.append(fpr_valid)
    tpr_valid_list.append(tpr_valid)
    roc_auc_valid_list.append(roc_auc_valid)

    w=np.zeros_like(w)
# give trn and tst misclass errors
print('features selected:',features)
print('train misclassification error:', train_misclass_errors)
print('valid misclassification error:', valid_misclass_errors)
```

```
# present 3 diff type of graph and check if correct or not
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(range(iteraions_val), train_misclass_errors_30,
label="Train")
plt.xlabel('iteraions_val')
plt.ylabel('Miss Class Error')
plt.title('30 Feature: iteraions_val vs Miss Classification Error')
plt.grid()
plt.legend()

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(features, train_misclass_errors, marker="o", label="Train")
plt.plot(features, valid_misclass_errors, marker="o", label="Test")
plt.xlabel('Features')
plt.ylabel('Miss class Error')
plt.title('Selected Features vs Miss Classification Error')
plt.grid()
plt.legend()

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(fpr_train_list[-1], tpr_train_list[-1], color='blue', lw=2,
label=f'Training Set (AUC = {roc_auc_train_list[-1]:.2f})')
plt.plot(fpr_valid_list[-1], tpr_valid_list[-1], color='darkorange',
lw=2,
label=f'Validation Set (AUC = {roc_auc_valid_list[-1]:.2f})')


plt.xlabel('False Positive Rate')
plt.ylabel('True positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()

plt.show()


0 weight is: 13
1 weight is: 13
2 weight is: 13
3 weight is: 13
4 weight is: 13
5 weight is: 13
6 weight is: 12
7 weight is: 12
8 weight is: 12
9 weight is: 12
10 weight is: 12
11 weight is: 12
```

```
12 weight is: 12
13 weight is: 12
14 weight is: 12
15 weight is: 12
16 weight is: 12
17 weight is: 12
18 weight is: 12
19 weight is: 12
20 weight is: 12
21 weight is: 12
22 weight is: 12
23 weight is: 12
24 weight is: 12
25 weight is: 12
26 weight is: 12
27 weight is: 12
28 weight is: 12
29 weight is: 12
30 weight is: 12
31 weight is: 12
32 weight is: 12
33 weight is: 12
34 weight is: 12
35 weight is: 12
36 weight is: 12
37 weight is: 12
38 weight is: 12
39 weight is: 12
40 weight is: 12
41 weight is: 12
42 weight is: 11
43 weight is: 11
44 weight is: 11
45 weight is: 11
46 weight is: 11
47 weight is: 11
48 weight is: 11
49 weight is: 11
50 weight is: 11
51 weight is: 11
52 weight is: 11
53 weight is: 11
54 weight is: 11
55 weight is: 11
56 weight is: 11
57 weight is: 11
58 weight is: 11
59 weight is: 11
60 weight is: 11
```

```
61 weight is: 11
62 weight is: 11
63 weight is: 11
64 weight is: 11
65 weight is: 11
66 weight is: 11
67 weight is: 11
68 weight is: 11
69 weight is: 11
70 weight is: 11
71 weight is: 11
72 weight is: 11
73 weight is: 11
74 weight is: 11
75 weight is: 11
76 weight is: 11
77 weight is: 11
78 weight is: 11
79 weight is: 11
80 weight is: 11
81 weight is: 11
82 weight is: 11
83 weight is: 11
84 weight is: 11
85 weight is: 11
86 weight is: 11
87 weight is: 11
88 weight is: 11
89 weight is: 11
90 weight is: 11
91 weight is: 11
92 weight is: 11
93 weight is: 11
94 weight is: 11
95 weight is: 11
96 weight is: 11
97 weight is: 11
98 weight is: 11
99 weight is: 11
for lambda of : 0.141 features is: 11
0 weight is: 40
1 weight is: 38
2 weight is: 37
3 weight is: 36
4 weight is: 36
5 weight is: 36
6 weight is: 34
7 weight is: 34
8 weight is: 34
```

```
9 weight is: 34
10 weight is: 34
11 weight is: 34
12 weight is: 33
13 weight is: 33
14 weight is: 33
15 weight is: 33
16 weight is: 33
17 weight is: 33
18 weight is: 33
19 weight is: 33
20 weight is: 33
21 weight is: 33
22 weight is: 33
23 weight is: 33
24 weight is: 33
25 weight is: 33
26 weight is: 33
27 weight is: 33
28 weight is: 33
29 weight is: 33
30 weight is: 33
31 weight is: 33
32 weight is: 33
33 weight is: 33
34 weight is: 32
35 weight is: 32
36 weight is: 32
37 weight is: 32
38 weight is: 32
39 weight is: 32
40 weight is: 32
41 weight is: 32
42 weight is: 32
43 weight is: 32
44 weight is: 32
45 weight is: 32
46 weight is: 32
47 weight is: 32
48 weight is: 32
49 weight is: 32
50 weight is: 32
51 weight is: 32
52 weight is: 32
53 weight is: 32
54 weight is: 32
55 weight is: 32
56 weight is: 32
57 weight is: 32
```

```
58 weight is: 32
59 weight is: 31
60 weight is: 31
61 weight is: 30
62 weight is: 30
63 weight is: 30
64 weight is: 30
65 weight is: 30
66 weight is: 30
67 weight is: 30
68 weight is: 30
69 weight is: 30
70 weight is: 30
71 weight is: 30
72 weight is: 30
73 weight is: 30
74 weight is: 30
75 weight is: 30
76 weight is: 30
77 weight is: 30
78 weight is: 30
79 weight is: 30
80 weight is: 30
81 weight is: 30
82 weight is: 30
83 weight is: 30
84 weight is: 30
85 weight is: 30
86 weight is: 30
87 weight is: 30
88 weight is: 30
89 weight is: 30
90 weight is: 30
91 weight is: 30
92 weight is: 30
93 weight is: 30
94 weight is: 30
95 weight is: 30
96 weight is: 30
97 weight is: 30
98 weight is: 30
99 weight is: 30
for lambda of : 0.098 features is: 30
0 weight is: 123
1 weight is: 119
2 weight is: 119
3 weight is: 118
4 weight is: 117
5 weight is: 117
```

```
6 weight is: 117
7 weight is: 115
8 weight is: 114
9 weight is: 114
10 weight is: 113
11 weight is: 113
12 weight is: 113
13 weight is: 112
14 weight is: 111
15 weight is: 110
16 weight is: 110
17 weight is: 109
18 weight is: 109
19 weight is: 109
20 weight is: 106
21 weight is: 105
22 weight is: 105
23 weight is: 104
24 weight is: 104
25 weight is: 104
26 weight is: 104
27 weight is: 103
28 weight is: 103
29 weight is: 102
30 weight is: 102
31 weight is: 101
32 weight is: 101
33 weight is: 101
34 weight is: 101
35 weight is: 101
36 weight is: 101
37 weight is: 101
38 weight is: 101
39 weight is: 101
40 weight is: 101
41 weight is: 101
42 weight is: 100
43 weight is: 100
44 weight is: 100
45 weight is: 100
46 weight is: 100
47 weight is: 100
48 weight is: 100
49 weight is: 100
50 weight is: 100
51 weight is: 100
52 weight is: 100
53 weight is: 100
54 weight is: 100
```

```
55 weight is: 100
56 weight is: 100
57 weight is: 100
58 weight is: 100
59 weight is: 100
60 weight is: 100
61 weight is: 100
62 weight is: 100
63 weight is: 100
64 weight is: 100
65 weight is: 100
66 weight is: 100
67 weight is: 100
68 weight is: 100
69 weight is: 100
70 weight is: 100
71 weight is: 100
72 weight is: 100
73 weight is: 100
74 weight is: 100
75 weight is: 100
76 weight is: 100
77 weight is: 100
78 weight is: 100
79 weight is: 100
80 weight is: 100
81 weight is: 100
82 weight is: 100
83 weight is: 100
84 weight is: 100
85 weight is: 100
86 weight is: 100
87 weight is: 99
88 weight is: 99
89 weight is: 99
90 weight is: 99
91 weight is: 99
92 weight is: 99
93 weight is: 99
94 weight is: 99
95 weight is: 99
96 weight is: 99
97 weight is: 99
98 weight is: 99
99 weight is: 99
for lambda of : 0.0712 features is: 99
0 weight is: 345
1 weight is: 340
2 weight is: 331
```

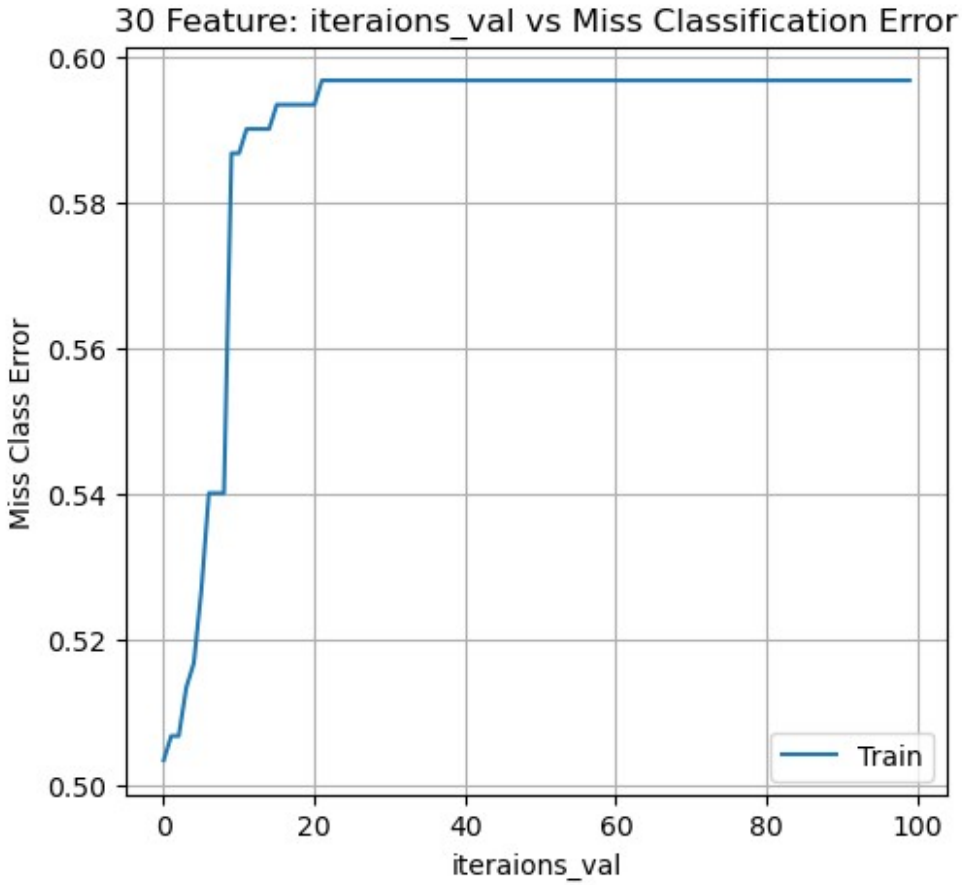
```
3 weight is: 326
4 weight is: 323
5 weight is: 322
6 weight is: 322
7 weight is: 321
8 weight is: 321
9 weight is: 321
10 weight is: 321
11 weight is: 320
12 weight is: 320
13 weight is: 320
14 weight is: 320
15 weight is: 320
16 weight is: 306
17 weight is: 306
18 weight is: 305
19 weight is: 304
20 weight is: 303
21 weight is: 303
22 weight is: 303
23 weight is: 303
24 weight is: 302
25 weight is: 302
26 weight is: 302
27 weight is: 302
28 weight is: 302
29 weight is: 301
30 weight is: 301
31 weight is: 301
32 weight is: 301
33 weight is: 301
34 weight is: 300
35 weight is: 300
36 weight is: 300
37 weight is: 300
38 weight is: 300
39 weight is: 300
40 weight is: 300
41 weight is: 300
42 weight is: 300
43 weight is: 300
44 weight is: 300
45 weight is: 300
46 weight is: 300
47 weight is: 300
48 weight is: 300
49 weight is: 300
50 weight is: 300
51 weight is: 300
```

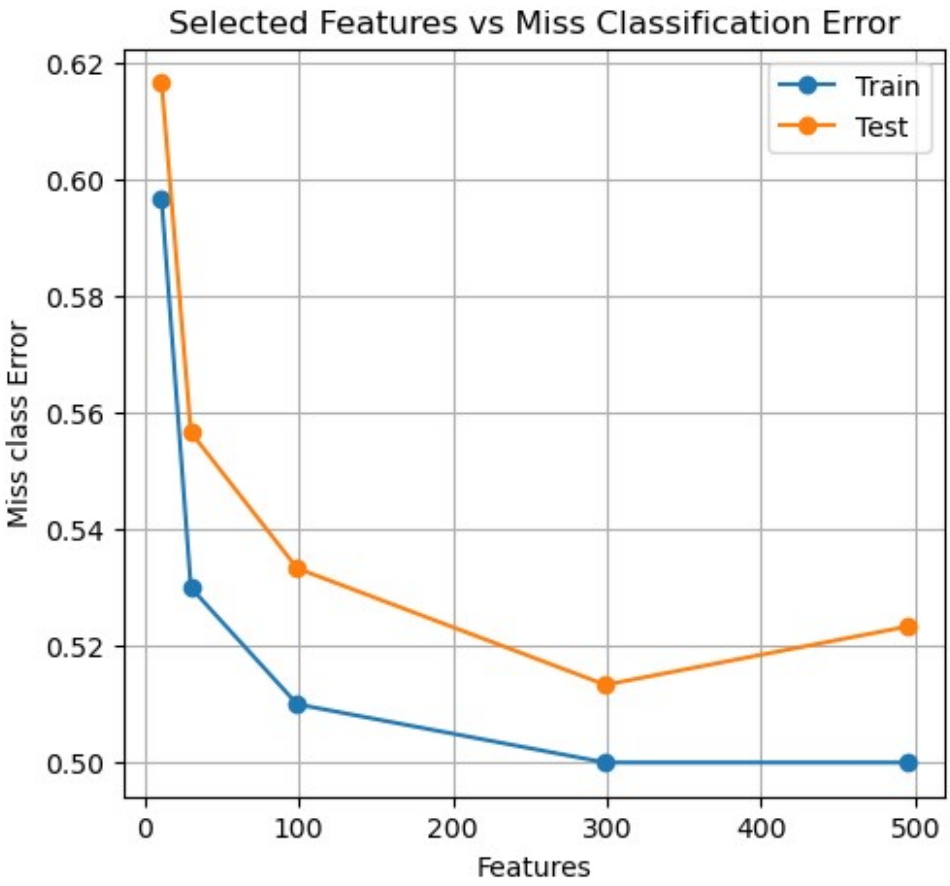
```
52 weight is: 300
53 weight is: 300
54 weight is: 300
55 weight is: 300
56 weight is: 300
57 weight is: 300
58 weight is: 300
59 weight is: 300
60 weight is: 300
61 weight is: 300
62 weight is: 300
63 weight is: 300
64 weight is: 300
65 weight is: 300
66 weight is: 300
67 weight is: 300
68 weight is: 300
69 weight is: 300
70 weight is: 300
71 weight is: 300
72 weight is: 300
73 weight is: 300
74 weight is: 300
75 weight is: 300
76 weight is: 300
77 weight is: 300
78 weight is: 300
79 weight is: 300
80 weight is: 300
81 weight is: 299
82 weight is: 299
83 weight is: 299
84 weight is: 299
85 weight is: 299
86 weight is: 299
87 weight is: 299
88 weight is: 299
89 weight is: 299
90 weight is: 299
91 weight is: 299
92 weight is: 299
93 weight is: 299
94 weight is: 299
95 weight is: 299
96 weight is: 299
97 weight is: 299
98 weight is: 299
99 weight is: 299
for lambda of : 0.0523 features is: 299
```

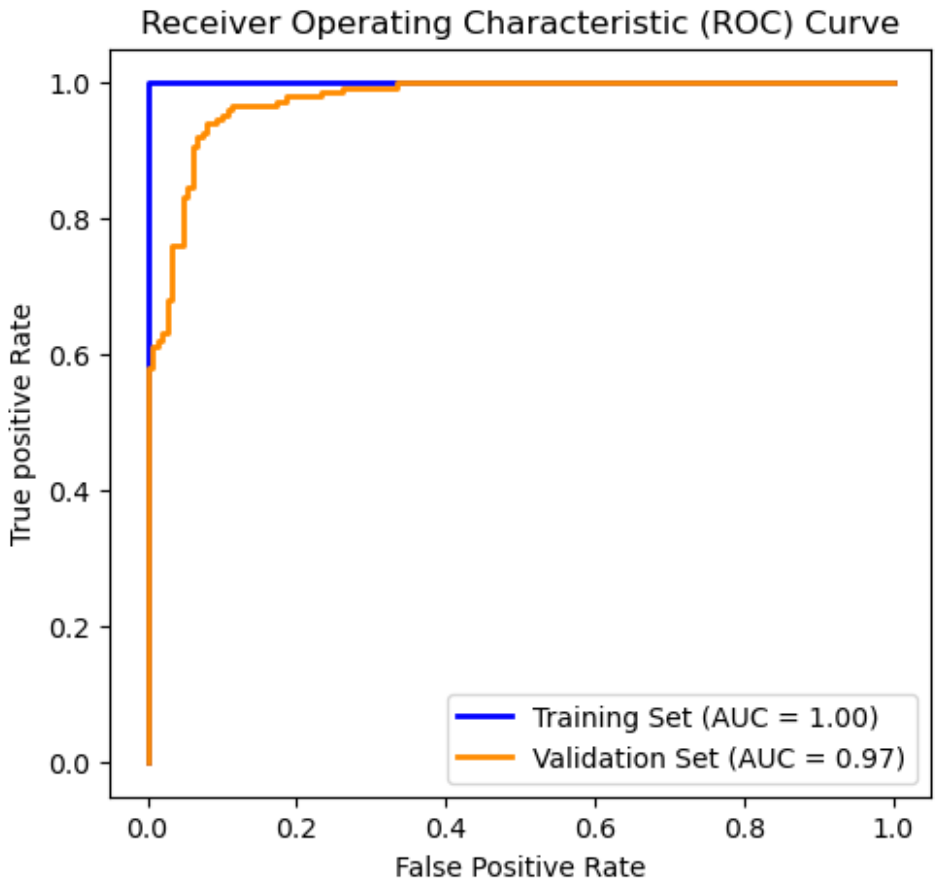
```
0 weight is: 566
1 weight is: 537
2 weight is: 521
3 weight is: 514
4 weight is: 508
5 weight is: 506
6 weight is: 503
7 weight is: 501
8 weight is: 500
9 weight is: 500
10 weight is: 500
11 weight is: 500
12 weight is: 500
13 weight is: 500
14 weight is: 500
15 weight is: 500
16 weight is: 500
17 weight is: 500
18 weight is: 500
19 weight is: 500
20 weight is: 500
21 weight is: 500
22 weight is: 500
23 weight is: 500
24 weight is: 500
25 weight is: 500
26 weight is: 500
27 weight is: 500
28 weight is: 499
29 weight is: 499
30 weight is: 499
31 weight is: 499
32 weight is: 499
33 weight is: 499
34 weight is: 499
35 weight is: 499
36 weight is: 498
37 weight is: 498
38 weight is: 498
39 weight is: 498
40 weight is: 497
41 weight is: 497
42 weight is: 497
43 weight is: 497
44 weight is: 497
45 weight is: 497
46 weight is: 497
47 weight is: 497
48 weight is: 496
```

```
49 weight is: 496
50 weight is: 496
51 weight is: 496
52 weight is: 496
53 weight is: 496
54 weight is: 496
55 weight is: 496
56 weight is: 496
57 weight is: 496
58 weight is: 496
59 weight is: 496
60 weight is: 496
61 weight is: 496
62 weight is: 496
63 weight is: 496
64 weight is: 496
65 weight is: 496
66 weight is: 496
67 weight is: 496
68 weight is: 496
69 weight is: 496
70 weight is: 496
71 weight is: 496
72 weight is: 496
73 weight is: 496
74 weight is: 496
75 weight is: 496
76 weight is: 496
77 weight is: 496
78 weight is: 496
79 weight is: 496
80 weight is: 496
81 weight is: 496
82 weight is: 496
83 weight is: 496
84 weight is: 496
85 weight is: 496
86 weight is: 496
87 weight is: 496
88 weight is: 496
89 weight is: 495
90 weight is: 495
91 weight is: 495
92 weight is: 495
93 weight is: 495
94 weight is: 495
95 weight is: 495
96 weight is: 495
97 weight is: 495
98 weight is: 495
```

```
99 weight is: 495
for lambda of : 0.0468 features is: 495
features selected: [11, 30, 99, 299, 495]
train misclassification error: [0.5966666666666667, 0.53, 0.51, 0.5,
0.5]
valid misclassification error: [0.6166666666666667,
0.5566666666666666, 0.5333333333333333, 0.5133333333333333,
0.5233333333333333]
```




Selected Features vs Miss Classification Error
Train
Test
Miss class Error
Features
0.62
0.60
0.58
0.56
0.54
0.52
0.50
0
100
200
300
400
500




```
#Create a DataFrame to display the results including lambda values,
selected features, train misclassification errors, and test
misclassification errors
final_result = pd.DataFrame({
    'Lambda': thresholds,
    'Features': features,
    'Train Misclass Error': train_misclass_errors,
    'Test Misclass Error': train_misclass_errors
})
print(final_result)
```

	Lambda	Features	Train Misclass Error	Test Misclass Error
0	0.029796	11	0.596667	0.596667
1	0.024500	30	0.530000	0.530000
2	0.017750	99	0.510000	0.510000
3	0.007500	299	0.500000	0.500000
4	0.000200	495	0.500000	0.500000

```
# Question No - 3
# imp all the libraries needed for this perticular query
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, accuracy_score


# insert madelon dataset for further use
mad_train_dt=np.loadtxt('madelon_train.data')
mad_train_lbs=np.loadtxt('madelon_train.labels')
mad_tst_dt=np.loadtxt('madelon_valid.data')
mad_tst_lbs=np.loadtxt('madelon_valid.labels')

# norm the req features
std=np.std(mad_train_dt,axis=0)

# data masking is imp
dat_mask=(std != 0)
# apply mask for madelon dataset to get mean and std val
mad_train_dt=mad_train_dt[:, dat_mask]
val_mean = np.mean(mad_train_dt,axis=0)
true_std = np.std(mad_train_dt,axis=0)

mad_train_dt = (mad_train_dt-val_mean)/true_std
mad_tst_dt = mad_tst_dt[:,dat_mask]
mad_tst_dt = (mad_tst_dt-val_mean)/true_std

 #Process dataset by add bias terminology, and convert label
mad_train_dt = np.insert(mad_train_dt,0,1, axis=1)
mad_tst_dt = np.insert(mad_tst_dt,0,1, axis=1)

mad_train_lbs[mad_train_lbs == 0] = -1
mad_tst_lbs[mad_tst_lbs == 0] = -1

# check mean and var for train and test set
print("train mean", np.mean(mad_train_dt))
print('train variance', np.var(mad_train_dt))
print('test mean', np.mean(mad_tst_dt))
print('test variance', np.var(mad_tst_dt))

# add iterate val and give threshold val as per rule
iteration_value = 100
thresholds = [0.029795977, 0.0245, 0.01775, 0.0075, 0.0002]
w = np.zeros(mad_train_dt.shape[1])

train_misclass_errors=[]
valid_misclass_errors=[]
train_misclass_errors_30=[]
features=[]
fpr_train_list=[]
tpr_train_list=[]
roc_auc_train_list=[]
fpr_valid_list=[]
```

```
tpr_valid_list=[]
roc_auc_valid_list=[]

# Continuously adjust weight val while applying L1 regularization,and
also monitoring and recording both misclass error and the features
that have been chosen or selected.
for val_lambda in thresholds:
    for i in range(iteration_value):
        # For training data and weight will check dot prod
        dot=np.sum(mad_train_dt * w, axis=1)
        # adding gradient val
        gradient=np.sum((mad_train_lbs/
(1+np.exp(mad_train_lbs*dot)))*(mad_train_dt).T, axis=1)

        w += gradient * (1/ mad_train_dt.shape[0])
        w[np.absolute(w) <= val_lambda]=0

        print(i, "weight is:",np.sum(w !=0))

        dot = np.sum(mad_train_dt*w, axis=1)

        y_pred_train = ((dot>=0)== mad_train_lbs)
        misclass_error_train = 1 - accuracy_score(mad_train_lbs,
y_pred_train)
        # try diff lambda val
        if(val_lambda == 0.029795977):
            train_misclass_errors_30.append(misclass_error_train)
    feature = np.sum(w != 0)
    features.append(feature)
    print('for lambda of :', val_lambda, 'features is:', feature)
         #Cal misclas error, ROC curve, and AUC score for train and
valid data sets
    train_misclass_errors.append(misclass_error_train)

    dot_valid = np.sum(mad_tst_dt * w, axis=1)
    y_pred_valid = ((dot_valid >= 0)== mad_tst_lbs)
    misclass_error_valid = 1 - accuracy_score(mad_tst_lbs,
y_pred_valid)
    valid_misclass_errors.append(misclass_error_valid)

    fpr_train, tpr_train, _ = roc_curve(mad_train_lbs, 1/ (1+ np.exp(-
dot)))
    roc_auc_train = auc(fpr_train, tpr_train)
    fpr_train_list.append(fpr_train)
    tpr_train_list.append(tpr_train)
    roc_auc_train_list.append(roc_auc_train)

    fpr_valid, tpr_valid, _ = roc_curve(mad_tst_lbs, 1/(1 + np.exp(-
dot_valid)))
```

```
    roc_auc_valid = auc(fpr_valid, tpr_valid)
    fpr_valid_list.append(fpr_valid)
    tpr_valid_list.append(tpr_valid)
    roc_auc_valid_list.append(roc_auc_valid)

    w=np.zeros_like(w)
 # check trn mean and var and then check for weight
print('features selected:',features)
print('train misclassification error:', train_misclass_errors)
print('valid misclassification error:', valid_misclass_errors)

#Create and display three subplots: Misclassification Error vs.
Iterations for 30 features
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(range(iteration_value), train_misclass_errors_30,
label="Train")
plt.xlabel('Iteration_value')
plt.ylabel('Miss Class Error')
plt.title('30 Feature: Iteration_value vs Miss Classification Error')
plt.grid()
plt.legend()

# Misclassification Error vs. Selected Features for both Train and
Valid
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(features, train_misclass_errors, marker="o", label="Train")
plt.plot(features, valid_misclass_errors, marker="o", label="Test")
plt.xlabel('Features')
plt.ylabel('Miss class Error')
plt.title('Selected Features vs Miss Classification Error')
plt.grid()
plt.legend()

# ROC curves with AUC scores for Train and Valid sets
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(fpr_train_list[-1], tpr_train_list[-1], color='blue', lw=2,
label=f'Training Set (AUC = {roc_auc_train_list[-1]:.2f})')
plt.plot(fpr_valid_list[-1], tpr_valid_list[-1], color='darkorange',
lw=2,
label=f'Validation Set (AUC = {roc_auc_valid_list[-1]:.2f})')


plt.xlabel('False Positive Rate')
plt.ylabel('True positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
```

```
plt.show()
```

```
train mean 0.0019960079840318605
train variance 0.9999960159521275
test mean 0.0008493208619512199
test variance 1.004451295877802
0 weight is: 14
1 weight is: 16
2 weight is: 13
3 weight is: 13
4 weight is: 13
5 weight is: 13
6 weight is: 10
7 weight is: 10
8 weight is: 10
9 weight is: 10
10 weight is: 10
11 weight is: 10
12 weight is: 10
13 weight is: 10
14 weight is: 10
15 weight is: 10
16 weight is: 10
17 weight is: 10
18 weight is: 10
19 weight is: 10
20 weight is: 10
21 weight is: 10
22 weight is: 10
23 weight is: 10
24 weight is: 10
25 weight is: 10
26 weight is: 10
27 weight is: 10
28 weight is: 10
29 weight is: 10
30 weight is: 10
31 weight is: 10
32 weight is: 10
33 weight is: 10
34 weight is: 10
35 weight is: 10
36 weight is: 10
37 weight is: 10
38 weight is: 10
39 weight is: 10
40 weight is: 10
41 weight is: 10
42 weight is: 10
```

```
43 weight is: 9
44 weight is: 9
45 weight is: 9
46 weight is: 9
47 weight is: 9
48 weight is: 8
49 weight is: 8
50 weight is: 8
51 weight is: 8
52 weight is: 8
53 weight is: 8
54 weight is: 8
55 weight is: 8
56 weight is: 8
57 weight is: 8
58 weight is: 8
59 weight is: 8
60 weight is: 8
61 weight is: 8
62 weight is: 8
63 weight is: 8
64 weight is: 8
65 weight is: 8
66 weight is: 8
67 weight is: 8
68 weight is: 8
69 weight is: 8
70 weight is: 8
71 weight is: 8
72 weight is: 8
73 weight is: 8
74 weight is: 8
75 weight is: 8
76 weight is: 8
77 weight is: 8
78 weight is: 8
79 weight is: 8
80 weight is: 8
81 weight is: 8
82 weight is: 8
83 weight is: 8
84 weight is: 8
85 weight is: 8
86 weight is: 8
87 weight is: 8
88 weight is: 8
89 weight is: 8
90 weight is: 8
91 weight is: 8
```

```
92 weight is: 8
93 weight is: 8
94 weight is: 8
95 weight is: 8
96 weight is: 8
97 weight is: 8
98 weight is: 8
99 weight is: 8
for lambda of : 0.029795977 features is: 8
0 weight is: 28
1 weight is: 31
2 weight is: 31
3 weight is: 32
4 weight is: 32
5 weight is: 32
6 weight is: 32
7 weight is: 32
8 weight is: 31
9 weight is: 31
10 weight is: 31
11 weight is: 31
12 weight is: 31
13 weight is: 31
14 weight is: 31
15 weight is: 31
16 weight is: 31
17 weight is: 31
18 weight is: 31
19 weight is: 31
20 weight is: 31
21 weight is: 31
22 weight is: 31
23 weight is: 31
24 weight is: 31
25 weight is: 31
26 weight is: 31
27 weight is: 31
28 weight is: 31
29 weight is: 31
30 weight is: 31
31 weight is: 31
32 weight is: 31
33 weight is: 31
34 weight is: 31
35 weight is: 31
36 weight is: 31
37 weight is: 31
38 weight is: 31
39 weight is: 31
```

```
40 weight is: 31
41 weight is: 31
42 weight is: 31
43 weight is: 31
44 weight is: 31
45 weight is: 31
46 weight is: 31
47 weight is: 31
48 weight is: 31
49 weight is: 31
50 weight is: 31
51 weight is: 31
52 weight is: 31
53 weight is: 31
54 weight is: 30
55 weight is: 30
56 weight is: 30
57 weight is: 30
58 weight is: 30
59 weight is: 30
60 weight is: 30
61 weight is: 30
62 weight is: 30
63 weight is: 30
64 weight is: 30
65 weight is: 30
66 weight is: 30
67 weight is: 30
68 weight is: 30
69 weight is: 30
70 weight is: 30
71 weight is: 30
72 weight is: 30
73 weight is: 30
74 weight is: 30
75 weight is: 30
76 weight is: 30
77 weight is: 30
78 weight is: 30
79 weight is: 30
80 weight is: 30
81 weight is: 30
82 weight is: 30
83 weight is: 30
84 weight is: 30
85 weight is: 30
86 weight is: 30
87 weight is: 30
88 weight is: 30
```

```
89 weight is: 30
90 weight is: 30
91 weight is: 30
92 weight is: 30
93 weight is: 30
94 weight is: 30
95 weight is: 30
96 weight is: 30
97 weight is: 30
98 weight is: 29
99 weight is: 29
for lambda of : 0.0245 features is: 29
0 weight is: 80
1 weight is: 90
2 weight is: 91
3 weight is: 92
4 weight is: 92
5 weight is: 93
6 weight is: 94
7 weight is: 94
8 weight is: 94
9 weight is: 93
10 weight is: 92
11 weight is: 93
12 weight is: 93
13 weight is: 93
14 weight is: 93
15 weight is: 94
16 weight is: 95
17 weight is: 96
18 weight is: 96
19 weight is: 96
20 weight is: 96
21 weight is: 96
22 weight is: 97
23 weight is: 97
24 weight is: 97
25 weight is: 97
26 weight is: 97
27 weight is: 97
28 weight is: 97
29 weight is: 97
30 weight is: 97
31 weight is: 97
32 weight is: 97
33 weight is: 97
34 weight is: 97
35 weight is: 97
36 weight is: 97
```

```
37 weight is: 97
38 weight is: 97
39 weight is: 97
40 weight is: 97
41 weight is: 97
42 weight is: 97
43 weight is: 97
44 weight is: 97
45 weight is: 97
46 weight is: 97
47 weight is: 97
48 weight is: 97
49 weight is: 97
50 weight is: 97
51 weight is: 97
52 weight is: 97
53 weight is: 97
54 weight is: 97
55 weight is: 97
56 weight is: 97
57 weight is: 97
58 weight is: 97
59 weight is: 97
60 weight is: 97
61 weight is: 97
62 weight is: 97
63 weight is: 97
64 weight is: 97
65 weight is: 97
66 weight is: 97
67 weight is: 97
68 weight is: 97
69 weight is: 96
70 weight is: 96
71 weight is: 96
72 weight is: 96
73 weight is: 96
74 weight is: 96
75 weight is: 96
76 weight is: 96
77 weight is: 96
78 weight is: 96
79 weight is: 96
80 weight is: 95
81 weight is: 96
82 weight is: 96
83 weight is: 96
84 weight is: 96
85 weight is: 96
```

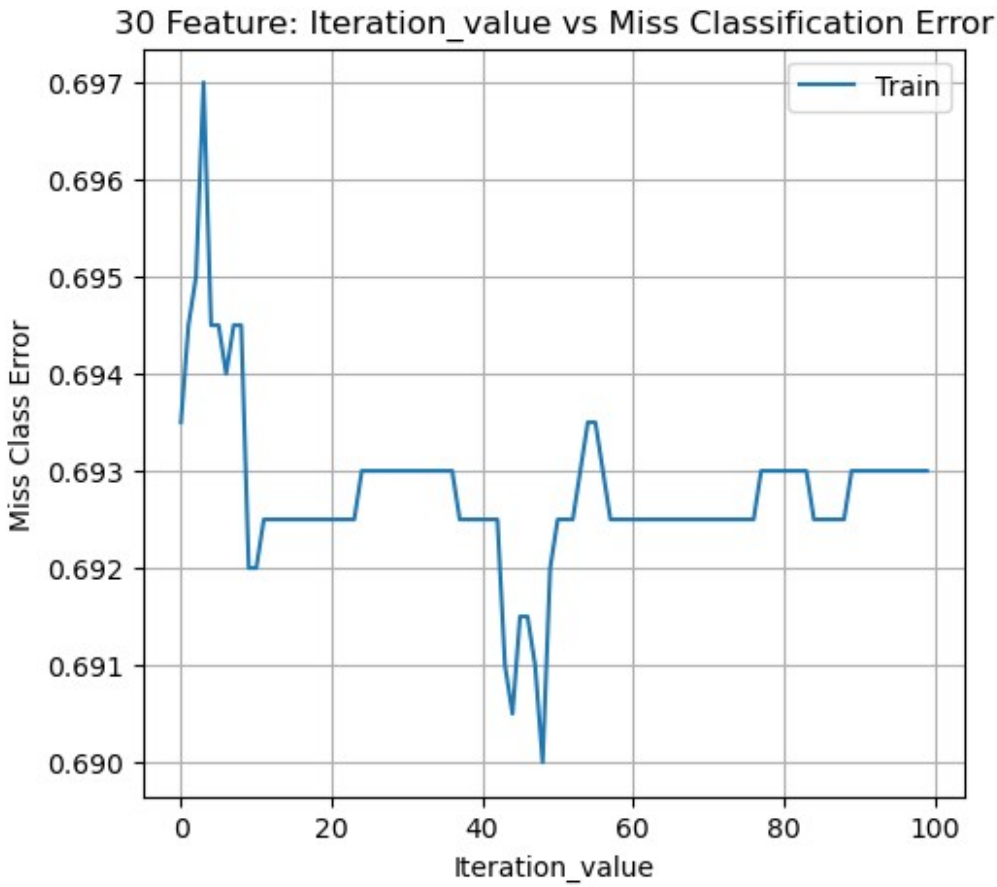
```
86 weight is: 96
87 weight is: 96
88 weight is: 97
89 weight is: 97
90 weight is: 97
91 weight is: 97
92 weight is: 97
93 weight is: 97
94 weight is: 97
95 weight is: 97
96 weight is: 97
97 weight is: 97
98 weight is: 97
99 weight is: 97
for lambda of : 0.01775 features is: 97
0 weight is: 264
1 weight is: 290
2 weight is: 298
3 weight is: 304
4 weight is: 304
5 weight is: 306
6 weight is: 303
7 weight is: 303
8 weight is: 306
9 weight is: 306
10 weight is: 303
11 weight is: 304
12 weight is: 305
13 weight is: 304
14 weight is: 304
15 weight is: 302
16 weight is: 302
17 weight is: 302
18 weight is: 302
19 weight is: 302
20 weight is: 301
21 weight is: 301
22 weight is: 300
23 weight is: 300
24 weight is: 299
25 weight is: 298
26 weight is: 300
27 weight is: 300
28 weight is: 300
29 weight is: 300
30 weight is: 300
31 weight is: 300
32 weight is: 300
33 weight is: 299
```

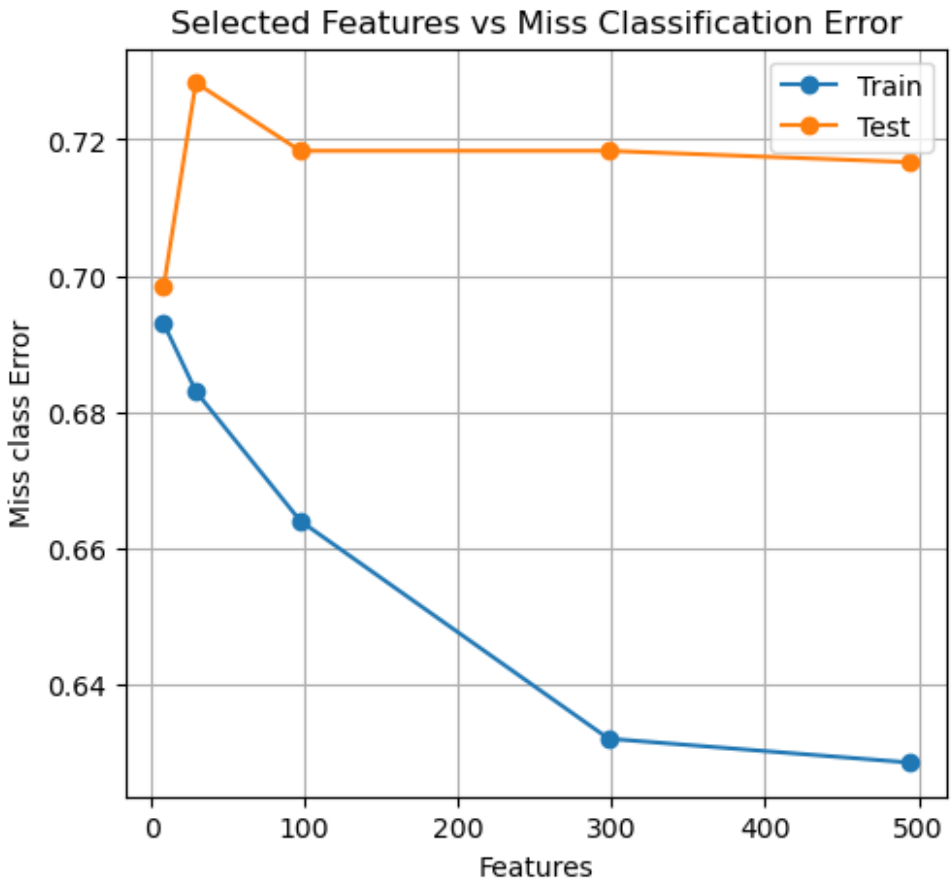
```
34 weight is: 298
35 weight is: 298
36 weight is: 298
37 weight is: 298
38 weight is: 297
39 weight is: 297
40 weight is: 297
41 weight is: 297
42 weight is: 297
43 weight is: 297
44 weight is: 298
45 weight is: 298
46 weight is: 298
47 weight is: 298
48 weight is: 298
49 weight is: 298
50 weight is: 299
51 weight is: 299
52 weight is: 299
53 weight is: 299
54 weight is: 300
55 weight is: 300
56 weight is: 300
57 weight is: 299
58 weight is: 299
59 weight is: 299
60 weight is: 299
61 weight is: 299
62 weight is: 299
63 weight is: 299
64 weight is: 299
65 weight is: 298
66 weight is: 297
67 weight is: 297
68 weight is: 297
69 weight is: 297
70 weight is: 297
71 weight is: 297
72 weight is: 297
73 weight is: 297
74 weight is: 297
75 weight is: 297
76 weight is: 297
77 weight is: 297
78 weight is: 297
79 weight is: 297
80 weight is: 297
81 weight is: 297
82 weight is: 297
```

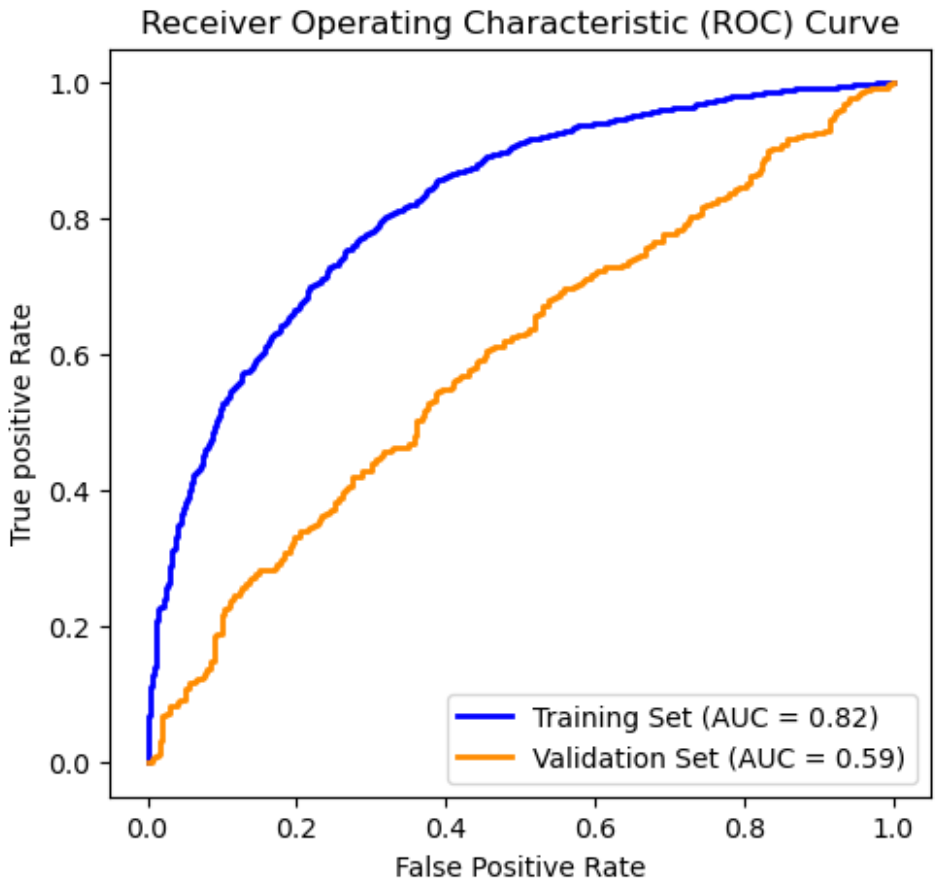
```
83 weight is: 297
84 weight is: 297
85 weight is: 297
86 weight is: 297
87 weight is: 297
88 weight is: 296
89 weight is: 296
90 weight is: 296
91 weight is: 296
92 weight is: 297
93 weight is: 297
94 weight is: 297
95 weight is: 297
96 weight is: 297
97 weight is: 297
98 weight is: 298
99 weight is: 298
for lambda of : 0.0075 features is: 298
0 weight is: 493
1 weight is: 497
2 weight is: 498
3 weight is: 497
4 weight is: 497
5 weight is: 496
6 weight is: 497
7 weight is: 498
8 weight is: 498
9 weight is: 500
10 weight is: 499
11 weight is: 497
12 weight is: 500
13 weight is: 499
14 weight is: 499
15 weight is: 500
16 weight is: 496
17 weight is: 499
18 weight is: 498
19 weight is: 497
20 weight is: 499
21 weight is: 499
22 weight is: 500
23 weight is: 499
24 weight is: 500
25 weight is: 498
26 weight is: 500
27 weight is: 499
28 weight is: 499
29 weight is: 498
30 weight is: 498
```

```
31 weight is: 498
32 weight is: 497
33 weight is: 496
34 weight is: 496
35 weight is: 497
36 weight is: 497
37 weight is: 497
38 weight is: 497
39 weight is: 498
40 weight is: 498
41 weight is: 498
42 weight is: 498
43 weight is: 497
44 weight is: 497
45 weight is: 497
46 weight is: 498
47 weight is: 498
48 weight is: 499
49 weight is: 499
50 weight is: 498
51 weight is: 497
52 weight is: 497
53 weight is: 497
54 weight is: 496
55 weight is: 496
56 weight is: 495
57 weight is: 495
58 weight is: 495
59 weight is: 495
60 weight is: 495
61 weight is: 494
62 weight is: 494
63 weight is: 493
64 weight is: 494
65 weight is: 494
66 weight is: 494
67 weight is: 494
68 weight is: 494
69 weight is: 493
70 weight is: 493
71 weight is: 493
72 weight is: 493
73 weight is: 493
74 weight is: 493
75 weight is: 493
76 weight is: 493
77 weight is: 493
78 weight is: 492
79 weight is: 492
```

```
80 weight is: 493
81 weight is: 493
82 weight is: 493
83 weight is: 493
84 weight is: 493
85 weight is: 493
86 weight is: 493
87 weight is: 493
88 weight is: 493
89 weight is: 493
90 weight is: 493
91 weight is: 492
92 weight is: 493
93 weight is: 494
94 weight is: 494
95 weight is: 494
96 weight is: 494
97 weight is: 494
98 weight is: 494
99 weight is: 494
for lambda of : 0.0002 features is: 494
features selected: [8, 29, 97, 298, 494]
train misclassification error: [0.6930000000000001, 0.683,
0.6639999999999999, 0.632, 0.6285000000000001]
valid misclassification error: [0.6983333333333333,
0.7283333333333333, 0.7183333333333333, 0.7183333333333333,
0.7166666666666667]
```


30 Feature: Iteration_value vs Miss Classification Error
Train
Miss Class Error
0.697
0.696
0.695
0.694
0.693
0.692
0.691
0.690
0
20
40
60
80
100
Iteration_value


Selected Features vs Miss Classification Error
Train
Test
0.72
0.70
0.68
0.66
0.64
Miss class Error
0
100
200
300
400
500
Features




```
#Create a DataFrame to display the results including lambda values,
selected features, train misclassification errors, and test
misclassification errors
final_result = pd.DataFrame({
    'Lambda': thresholds,
    'Features': features,
    'Train Misclass Error': train_misclass_errors,
    'Test Misclass Error': train_misclass_errors
})
print(final_result)
```

```
     Lambda  Features  Train Misclass Error  Test Misclass Error
0  0.029796         8                0.6930               0.6930
1  0.024500        29                0.6830               0.6830
2  0.017750        97                0.6640               0.6640
3  0.007500       298                0.6320               0.6320
4  0.000200       494                0.6285               0.6285
```

```python
# ML ASS QNO - 1, 2, 3
# importing those installed libraries
import pandas as pd
import numpy as np
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import random
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
import zipfile


# first normalizing the function and then take mean for training
dataset and returing normalized data
def normalize_train(X):
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    X_normalized = (X - mean) / std
    return X_normalized, mean, std

# first normalizing the function and then take mean for testing
dataset and returing normalized data
def normalize_test(X, mean, std):
    return (X - mean) / std

# adding multiple variable and functions and defining them by diff
names to get the proper idea
def L_D(beta, Y, X, s):
    N = len(Y)
    # adding the functions
    sum_term = np.sum(np.log(1 + np.exp(-Y * np.dot(X, beta))))
    regularization = s * np.linalg.norm(beta)**2
    return (1/N) * sum_term + regularization
# giving threshold
def soft_thresholding(beta, threshold):
    return np.sign(beta) * np.maximum(np.abs(beta) - threshold, 0)

# defining grad for functions
def proximal_gradient_descent(Y, X, beta, s, mu, N_iter, k):
    step_size = 1/mu
    # looping btw all items to assign beta and masking the data
    for _ in range(N_iter):
        gradient = -np.dot(X.T, Y / (1 + np.exp(Y * np.dot(X, beta))))
        beta_temp = beta - step_size * gradient
        beta = soft_thresholding(beta_temp, s*step_size)
        #  obtain k values
        idx = np.argsort(np.abs(beta))[-k:]
        mask = np.ones(beta.shape[0], dtype=bool)
```

```
        mask[idx] = False
        # assigning mask for beta
        beta[mask] = 0
    return beta

# assigning different names to train and test data set to identify
X_train = pd.read_csv('gisette_train.data', delim_whitespace=True,
header=None)
Y_train = pd.read_csv('gisette_train.labels', header=None)
X_test = pd.read_csv('gisette_valid.data', delim_whitespace=True,
header=None)
Y_test = pd.read_csv('gisette_valid.labels', header=None)

Y_test.replace(-1, 0, inplace = True)
Y_train.replace(-1,0,inplace = True)
X_test.replace(-1, 0, inplace = True)

Y_train = Y_train.values.flatten()
Y_test = Y_test.values.flatten()

# assigning required lib for this perticular code
import os
import numpy as np
import dask.dataframe as dd
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, roc_curve
from scipy.special import expit
import matplotlib.pyplot as plt

#  assigning required values for configuration param
CONFIG_ITERATIONS = 300
CONFIG_PARAM1 = 200
CONFIG_PARAM2 = 0.0001

# defining aux function
def auxiliary_function(val):
   # checking  for value to add sign
    if val < 0:
        return np.log(1+np.exp(val))
    else:
        return np.log(1+np.exp(-val))

    # adding vectorzed fns
vectorized_function = np.vectorize(auxiliary_function)

# defining schedule fns
def function_schedule(param_a: int,
                      param_b: int,
```

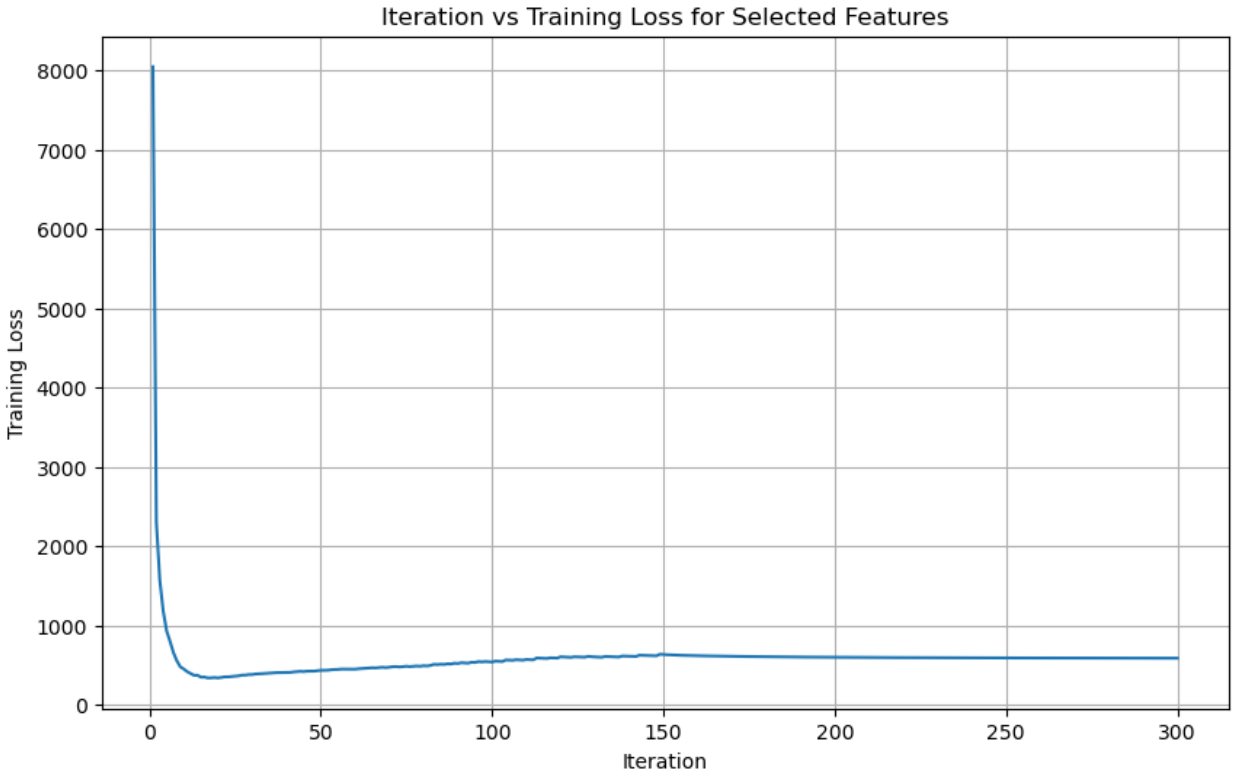
```
                        param_c : float = CONFIG_PARAM1,
                        param_d : int = CONFIG_ITERATIONS) -> list:
    # Provides calculated outputs depending on the specified values.
    computed_values = [round(param_b+(param_a-param_b)*np.max([0,
(param_d-2*i)/(param_d+2*i*param_c)])) for i in range(param_d)]
    return computed_values

class GenericFeatureSelector():
# A versatile feature selection class.
    def __init__(self,
                 sequence: list,
                 learning_rate: float,
                 config_iterations: int = CONFIG_ITERATIONS,
                 reg_param: float = CONFIG_PARAM2):
        self.sequence = sequence
        self.learning_rate = learning_rate
        self.reg_param = reg_param
        self.config_iterations = config_iterations
        self.coefficients = None
        self.error_log = []
#  defining compute loss fns
    def _compute_loss(self, coef, data_x, data_y):
        coef_data = np.matmul(coef, data_x.T)
        new_data_y = np.where(coef_data < 0, data_y, data_y-1)
        return -np.dot(new_data_y, coef_data)
+vectorized_function(coef_data).sum()+\
                self.reg_param*np.dot(coef, coef)
# adding compute grad fns
    def _compute_gradient(self, coef, data_x, data_y):
        data_coef = np.matmul(data_x, coef)
        gradient = -np.matmul(data_x.T, data_y-expit(data_coef)) +
2*self.reg_param*coef
        return gradient

    def fit(self, data_x, data_y):
        data_y = data_y.squeeze()
        coef_init = np.zeros(data_x.shape[1])
        selected_data = range(data_x.shape[1])
        for i in range(self.config_iterations):
            coef_init, selected_data =
self.update_coefficients(coef_init, data_x, data_y, selected_data, i)
            self.log_errors(coef_init, data_x, data_y, selected_data)
        return self
# getting coeff values
    def update_coefficients(self, coef_init, data_x, data_y,
selected_data, i):
        coef_init[selected_data] = coef_init[selected_data] -
self.learning_rate * self._compute_gradient(coef_init[selected_data],
data_x[:,selected_data], data_y)
```

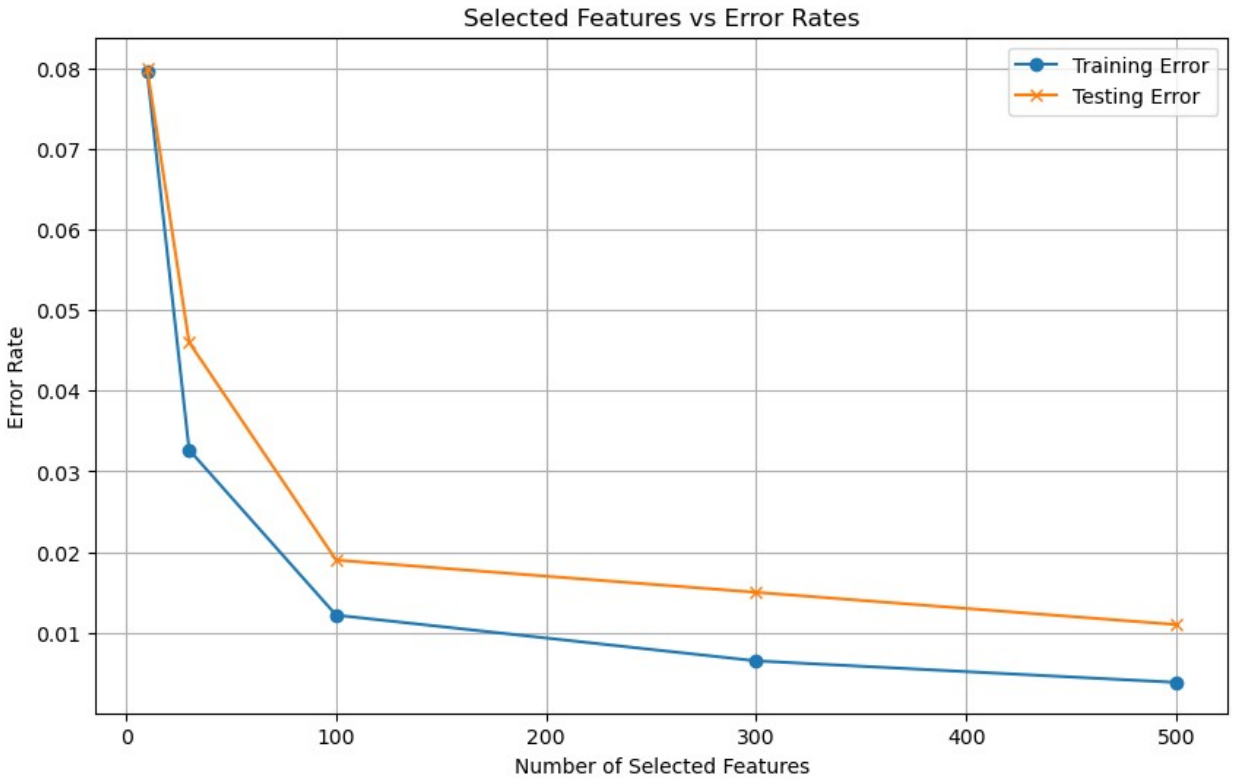
```
        selected_data = np.abs(coef_init).argsort()[-
(self.sequence[i]):]
        self.coefficients = np.zeros(data_x.shape[1])
        self.coefficients[selected_data] = coef_init[selected_data]
        coef_init = self.coefficients
        return coef_init, selected_data
# getting logging error if any
    def log_errors(self, coef_init, data_x, data_y, selected_data):

self.error_log.append(self._compute_loss(coef_init[selected_data],
data_x[:,selected_data], data_y))
# adding pred values and fns
    def predict(self, data_x):
        data_coef = np.matmul(data_x, self.coefficients)
        return np.where(
            expit(data_coef) > 0.5,
            1,
            0
        )

    def predict_proba(self, data_x):
        data_coef = np.matmul(data_x, self.coefficients)
        return expit(data_coef)
# adding bias fns
class AppendBias():
# Adds columns containing 1s to the dataset.
    def fit(self, data_x, data_y = None):
        return self

    def transform(self, data_x):
        return np.hstack([np.ones((data_x.shape[0], 1)), data_x])

class MainTaskHandler():
    # The central controller for tasks and calculations.

    def __init__(self,
                 training_data_x: pd.DataFrame,
                 training_data_y: pd.DataFrame,
                 testing_data_x: pd.DataFrame,
                 testing_data_y: pd.DataFrame,
                 learning_rate=0.02) -> None:

#Calculates properties using both training and testing data.
        feature_counts = [500, 300, 100, 30, 10]
        nonzero_counts = []
        training_error_log = []
        testing_error_log = []

        for count in feature_counts:
            pipeline = self.initialize_pipeline(training_data_x,
```

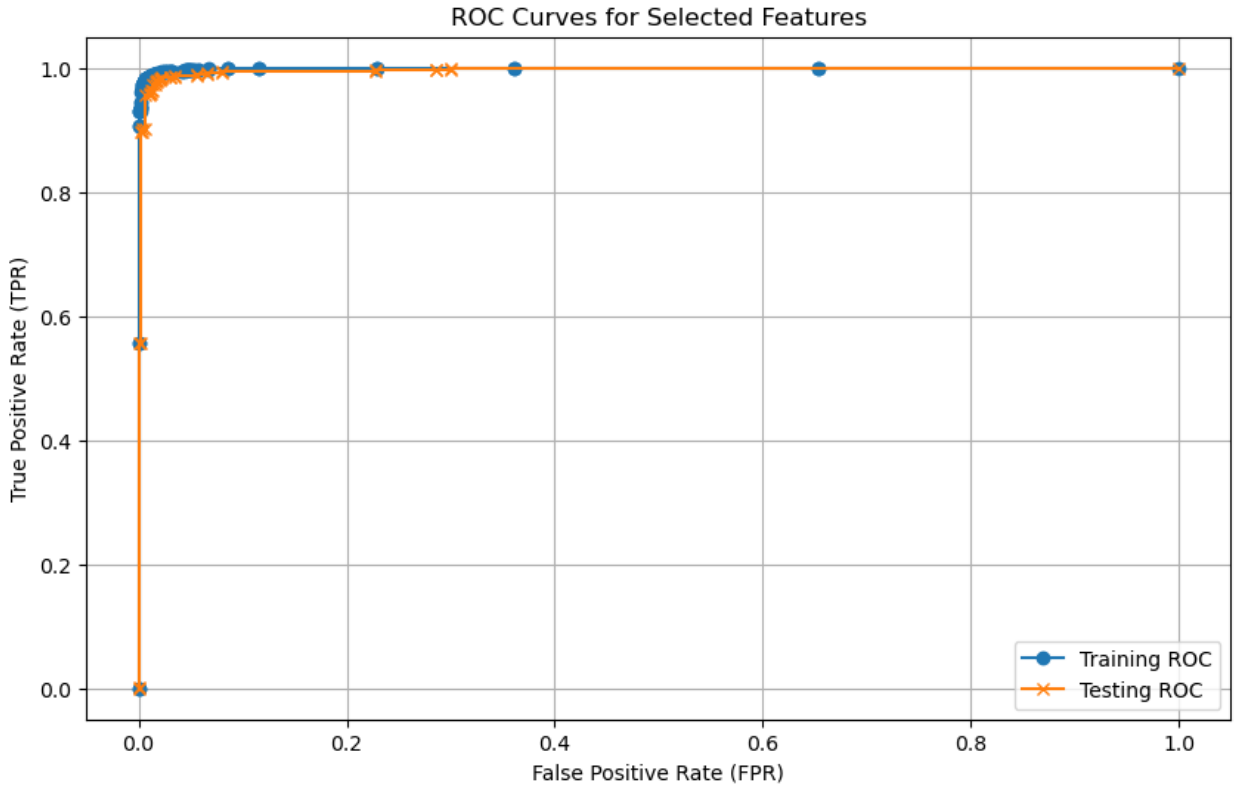
```
training_data_y, count, learning_rate)

nonzero_counts.append(np.count_nonzero(pipeline['selector'].coefficien
ts))

            training_error, testing_error =
self.compute_errors(pipeline, training_data_x, training_data_y,
testing_data_x, testing_data_y)
            training_error_log.append(training_error)
            testing_error_log.append(testing_error)

            self.handle_special_cases(pipeline, count,
training_data_x, testing_data_x)

        self.classification_summary = [list(x) for x in
zip(feature_counts, nonzero_counts, training_error_log,
testing_error_log)]

    def initialize_pipeline(self, data_x, data_y, count,
learning_rate):
        computed_sequence = function_schedule(data_x.shape[1]+1,
count)
        pipeline = Pipeline([
            ('scaler', StandardScaler()),
            ('bias_appender', AppendBias()),
            ('selector', GenericFeatureSelector(computed_sequence,
learning_rate=learning_rate))
        ])
        pipeline.fit(data_x, data_y)
        return pipeline

    def compute_errors(self, pipeline, training_data_x,
training_data_y, testing_data_x, testing_data_y):
        training_predicted_y = pipeline.predict(training_data_x)
        training_error = 1 - accuracy_score(training_data_y,
training_predicted_y)

        testing_predicted_y = pipeline.predict(testing_data_x)
        testing_error = 1 - accuracy_score(testing_data_y,
testing_predicted_y)

        return training_error, testing_error

    def handle_special_cases(self, pipeline, count, training_data_x,
testing_data_x):
        if count == 30:
            self.training_loss_sequence =
pipeline['selector'].error_log
        if count == 100:
            self.training_prob_predictions =
```

```
pipeline.predict_proba(training_data_x)
            self.test_prob_predictions =
pipeline.predict_proba(testing_data_x)
# defining  a function to plot Iteration vs Training Loss for Selected
Features in grid format
    def plot_training_loss(self):
        plt.figure(figsize=(10,6))
        plt.plot(range(1, CONFIG_ITERATIONS+1),
self.training_loss_sequence)
        plt.title('Iteration vs Training Loss for Selected Features')
        plt.xlabel('Iteration')
        plt.ylabel('Training Loss')
        plt.grid(True)
        plt.show()

# Creating a function to depict the relationship between Selected
Features vs Error Rates for Chosen Attributes in a grid layout.
    def plot_error_rates(self):
        plt.figure(figsize=(10,6))
        nonzero_values  = [item[1] for item in
self.classification_summary]
        training_errors = [item[2] for item in
self.classification_summary]
        testing_errors  = [item[3] for item in
self.classification_summary]
        plt.plot(nonzero_values, training_errors, marker='o',
label='Training Error')
        plt.plot(nonzero_values, testing_errors, marker='x',
label='Testing Error')
        plt.title('Selected Features vs Error Rates')
        plt.xlabel('Number of Selected Features')
        plt.ylabel('Error Rate')
        plt.legend()
        plt.grid(True)
        plt.show()
# Creating a function
    def generate_summary_table(self):
        nonzero_values  = [item[1] for item in
self.classification_summary]
        training_errors = [item[2] for item in
self.classification_summary]
        testing_errors  = [item[3] for item in
self.classification_summary]
        table_data = {
            'Selected Features': nonzero_values,
            'Training Errors': training_errors,
            'Testing Errors': testing_errors
        }
        summary_df = pd.DataFrame(table_data)
```

```
        return summary_df

    def display_roc_curves(self, training_labels, testing_labels):
        train_fpr, train_tpr, _ = roc_curve(training_labels,
self.training_prob_predictions)
        test_fpr, test_tpr, _ = roc_curve(testing_labels,
self.test_prob_predictions)

        #  plotting a graph to visualize the relationship for ROC
Curves for Selected Features for the chosen features in a grid format.
        plt.figure(figsize=(10, 6))
        plt.plot(train_fpr, train_tpr, marker='o', label='Training
ROC')
        plt.plot(test_fpr, test_tpr, marker='x', label='Testing ROC')
        plt.xlabel('False Positive Rate (FPR)')
        plt.ylabel('True Positive Rate (TPR)')
        plt.legend()
        plt.title('ROC Curves for Selected Features')
        plt.grid(True)
        plt.show()


# Initializing the main task handler with the specified parameters
task_handler = MainTaskHandler(training_data_x=X_train,
                               training_data_y=Y_train,
                               testing_data_x=X_test,
                               testing_data_y=Y_test,
                               learning_rate=0.95/X_train.shape[0])
# Plotting the training loss for analysis
task_handler.plot_training_loss()
# Plotting the error rates for further evaluation
task_handler.plot_error_rates()
# Generating a summary table for an overview of the results
summary_df = task_handler.generate_summary_table()
# Displaying ROC curves for training and testing data labels
task_handler.display_roc_curves(training_labels=Y_train,
testing_labels=Y_test)
#Display the summary DataFrame
summary_df
```


Iteration vs Training Loss for Selected Features
Training Loss
Iteration
8000
7000
6000
5000
4000
3000
2000
1000
0
0
50
100
150
200
250
300

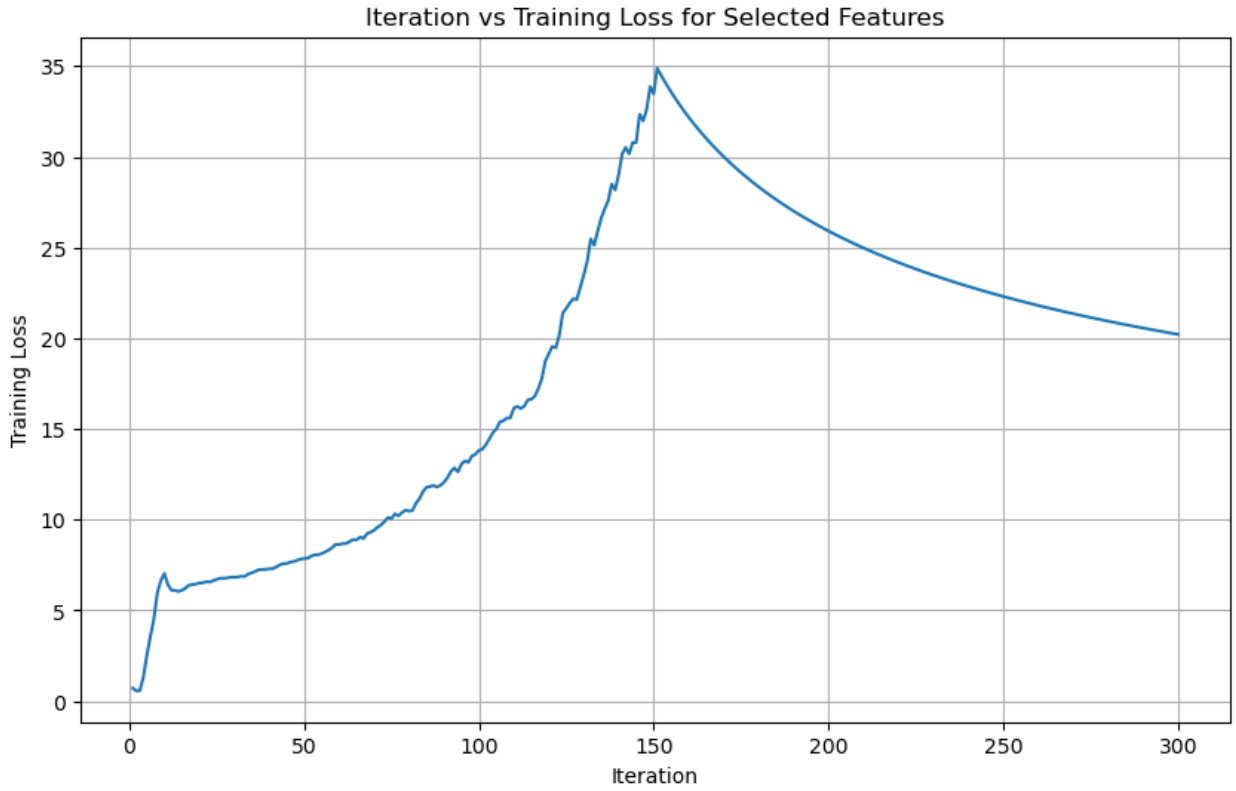

Selected Features vs Error Rates
Training Error
Testing Error
Error Rate
Number of Selected Features
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
100
200
300
400
500

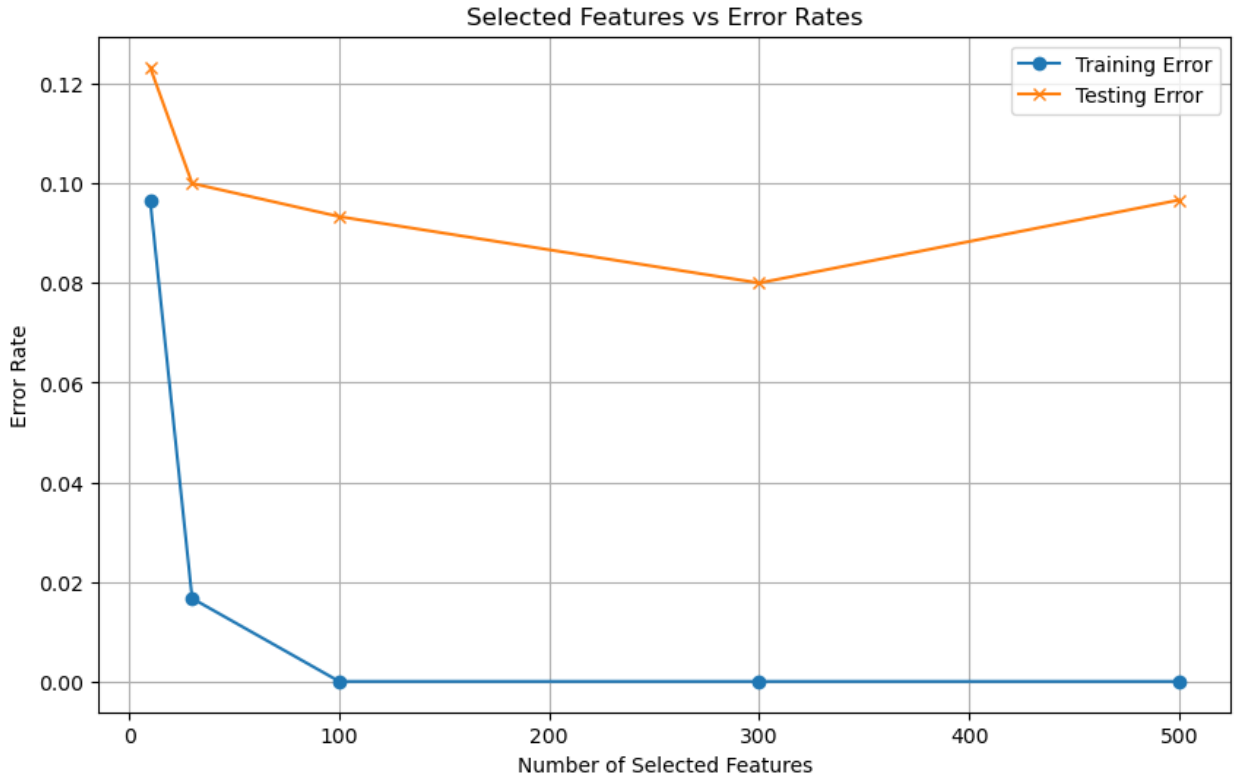


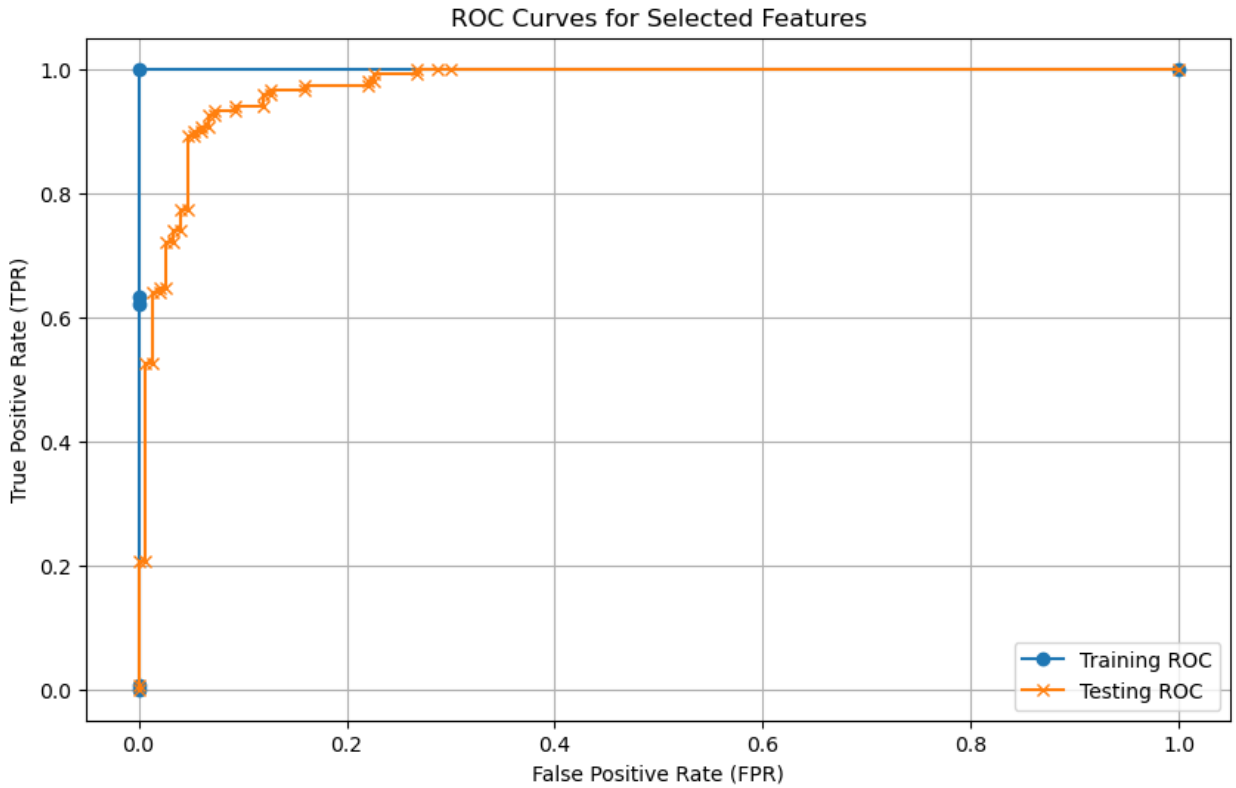
	Selected Features	Training Errors	Testing Errors
0	500	0.003833	0.011
1	300	0.006500	0.015
2	100	0.012167	0.019
3	30	0.032667	0.046
4	10	0.079667	0.080

```
# adding datset of dex for code
X_train_dexter = pd.read_csv('dexter_train.csv', header = None)
X_test_dexter = pd.read_csv('dexter_valid.csv', header = None)
Y_train_dexter = pd.read_csv('dexter_train.labels',
delim_whitespace=True, header=None )
Y_test_dexter =
pd.read_csv('dexter_valid.labels',delim_whitespace=True, header=None)

Y_test_dexter.replace(-1, 0, inplace = True)
Y_train_dexter.replace(-1,0,inplace = True)

Y_train_dexter = Y_train_dexter.values.flatten()
Y_test_dexter = Y_test_dexter.values.flatten()

# Initializing the main task handler with the specified parameters
task_handler = MainTaskHandler(training_data_x=X_train_dexter,
                               training_data_y=Y_train_dexter,
                               testing_data_x=X_test_dexter,
                               testing_data_y=Y_test_dexter,
```

```
learning_rate=0.5/X_train_dexter.shape[0])
# Plotting the training loss for analysis
task_handler.plot_training_loss()
task_handler.plot_error_rates()
# Generating a summary table for an overview of the results
summary_df = task_handler.generate_summary_table()
# Display ROC (Receiver Operating Characteristic) curves for both
training and testing data.
task_handler.display_roc_curves(training_labels=Y_train_dexter,
testing_labels=Y_test_dexter)
## Display the summary DataFrame
summary_df
```

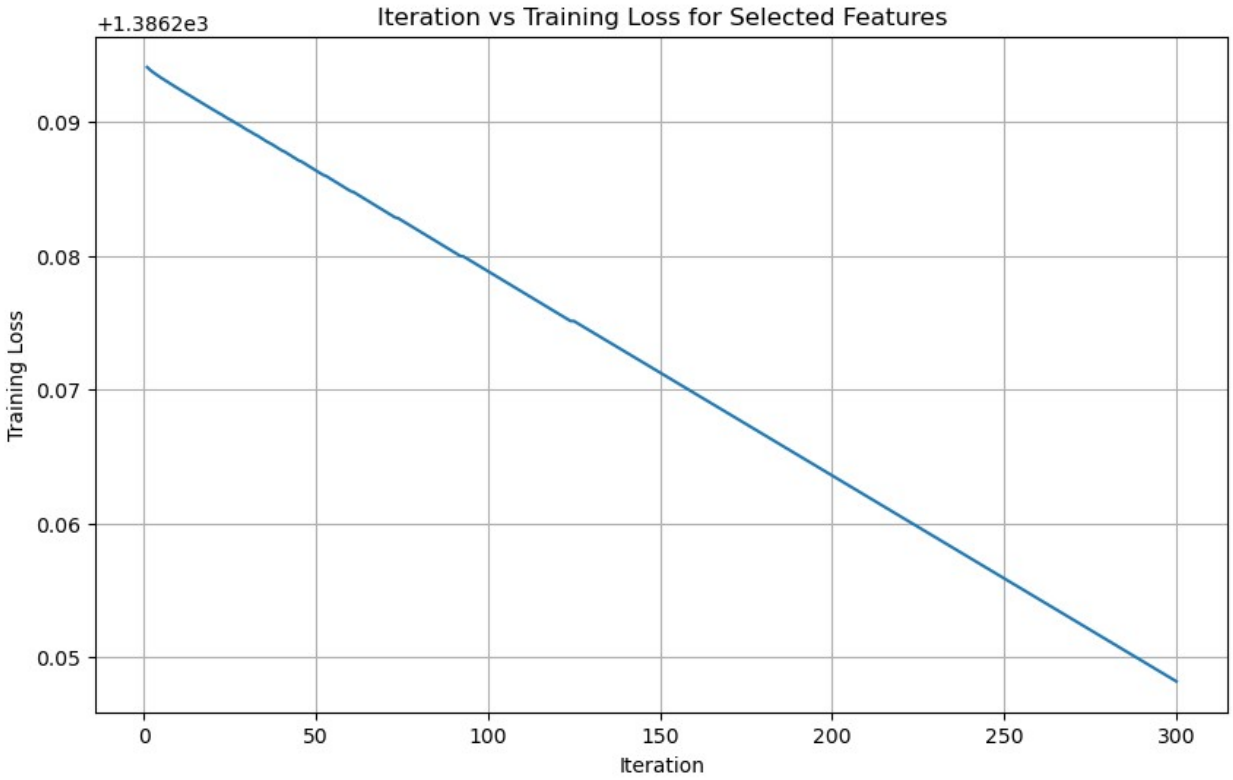


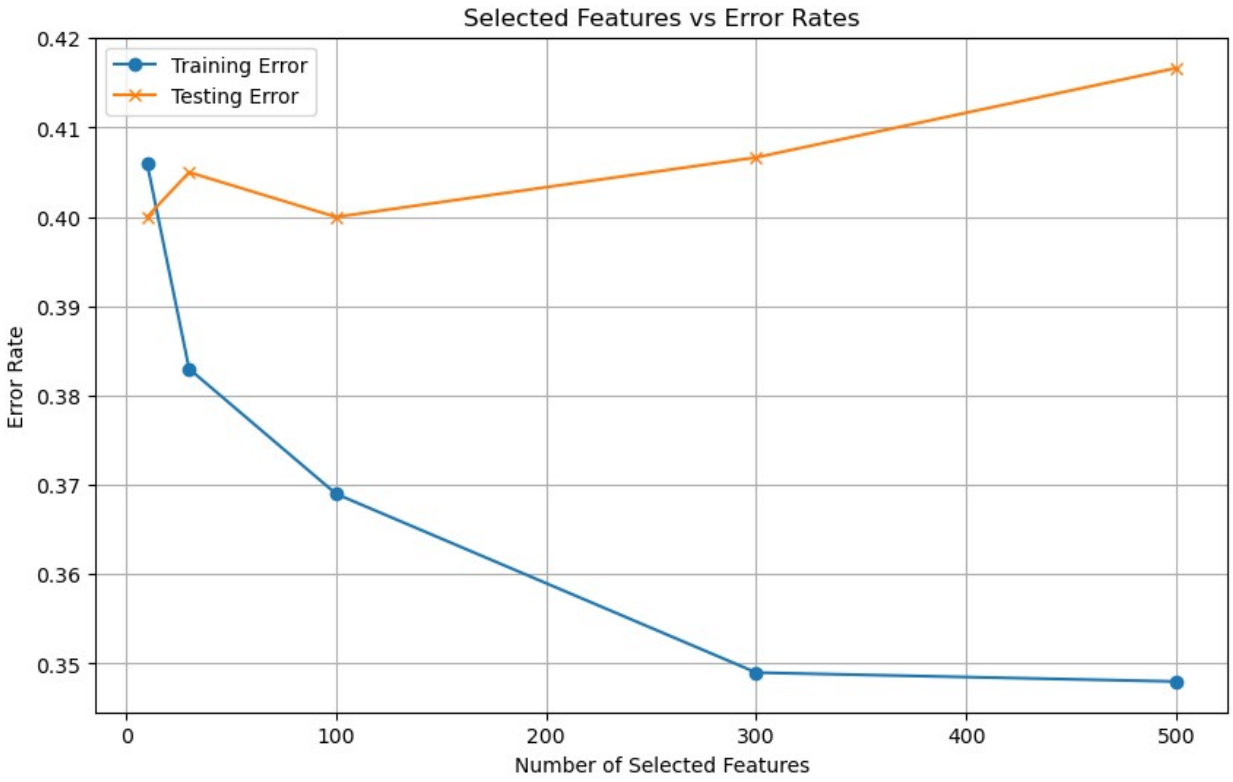

Selected Features vs Error Rates
Training Error
Testing Error
Error Rate
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
100
200
300
400
500
Number of Selected Features

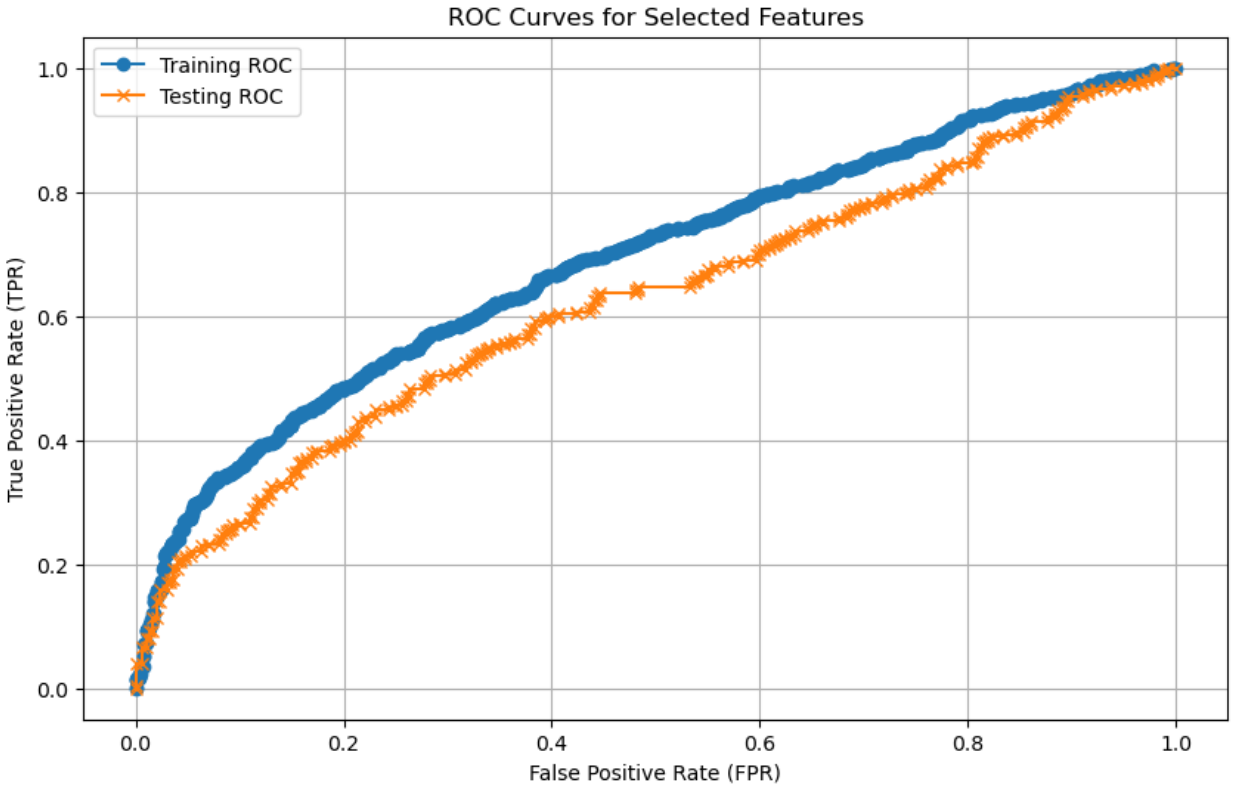

ROC Curves for Selected Features
True Positive Rate (TPR)
1.0
0.8
0.6
0.4
0.2
0.0
Training ROC
Testing ROC
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate (FPR)

```
   Selected Features  Training Errors  Testing Errors
0                500         0.000000        0.096667
1                300         0.000000        0.080000
2                100         0.000000        0.093333
3                 30         0.016667        0.100000
4                 10         0.096667        0.123333
```

```
# Reading the training and testing data for madelon dataset
X_train_madelon = pd.read_csv('madelon_train.data',
delim_whitespace=True, header = None)
X_test_madelon = pd.read_csv('madelon_valid.data',
delim_whitespace=True, header = None)
Y_train_madelon = pd.read_csv('madelon_train.labels',
delim_whitespace=True, header=None )
Y_test_madelon =
pd.read_csv('madelon_valid.labels',delim_whitespace=True, header=None)
# Replacing -1 values with 0 in both the testing and training label
Y_test_madelon.replace(-1, 0, inplace = True)
Y_train_madelon.replace(-1,0,inplace = True)

# Flattening the training and testing labels for madelon dataset

Y_train_madelon = Y_train_madelon.values.flatten()
Y_test_madelon = Y_test_madelon.values.flatten()

# Initializing the main task handler with the specified parameters
task_handler = MainTaskHandler(training_data_x=X_train_madelon,
                               training_data_y=Y_train_madelon,
                               testing_data_x=X_test_madelon,
                               testing_data_y=Y_test_madelon,

learning_rate=0.000001/X_train_madelon.shape[0])
# # Plotting the training loss for analysis
task_handler.plot_training_loss()
task_handler.plot_error_rates()
# Generating a summary table for an overview of the results
summary_df = task_handler.generate_summary_table()
# Display ROC (Receiver Operating Characteristic) curves for both
training and testing data.

task_handler.display_roc_curves(training_labels=Y_train_madelon,
testing_labels=Y_test_madelon)
#Display the summary DataFrame containing task performance metrics
summary_df
```


Iteration vs Training Loss for Selected Features
+1.3862e3
0.09
0.08
0.07
0.06
0.05
Training Loss
0
50
100
150
200
250
300
Iteration


Selected Features vs Error Rates
Training Error
Testing Error
0.42
0.41
0.40
0.39
0.38
0.37
0.36
0.35
Error Rate
0
100
200
300
400
500
Number of Selected Features



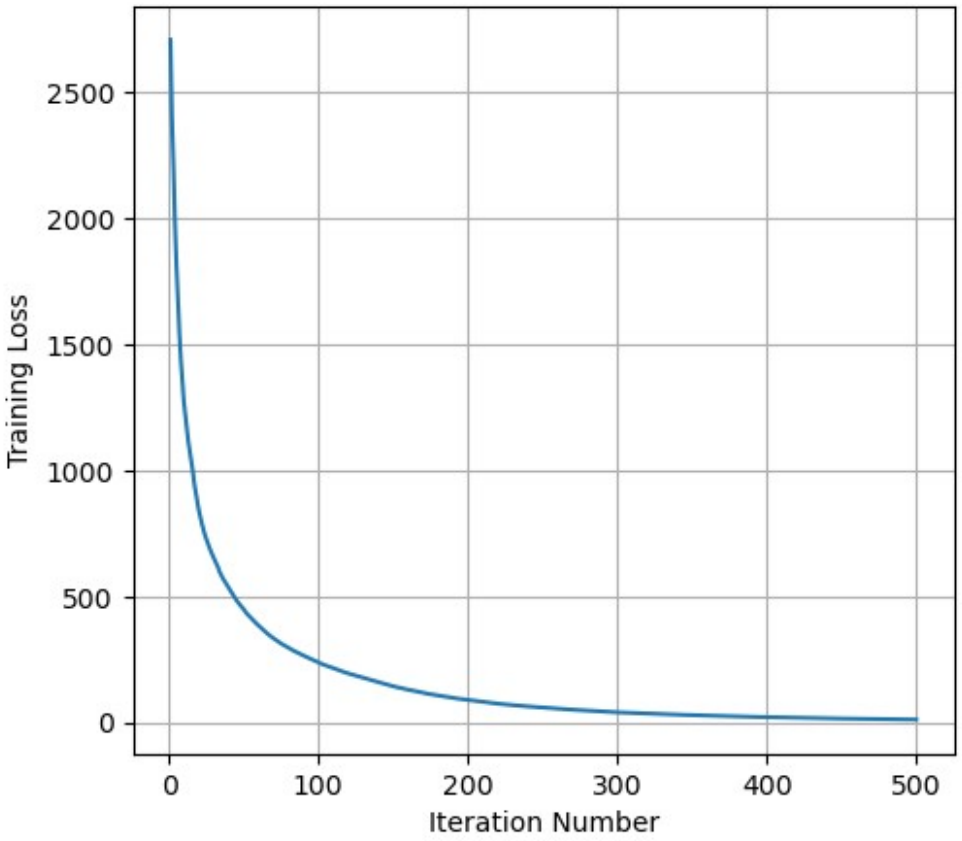
	Selected Features	Training Errors	Testing Errors
0	500	0.348	0.416667
1	300	0.349	0.406667
2	100	0.369	0.400000
3	30	0.383	0.405000
4	10	0.406	0.400000

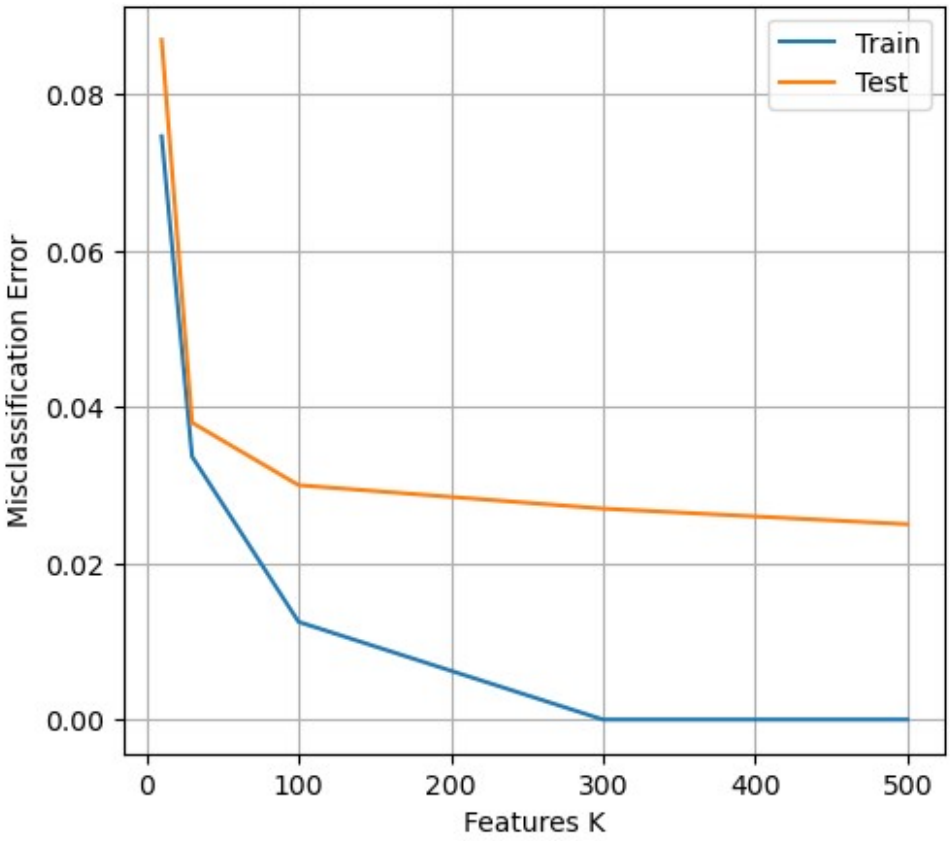
```
# Question No - 1
#Import necessary libraries
import pandas as pd
from sklearn.metrics import roc_curve,roc_auc_score,accuracy_score
import numpy as np
import matplotlib.pyplot as plt

#Load the training and validation data
x = np.loadtxt('gisette_train.data')
y = np.loadtxt('gisette_train.labels')
xt = np.loadtxt('gisette_valid.data')
yt = np.loadtxt('gisette_valid.labels')

#Data preprocessing and initializng variables
ones_train = np.ones(x.shape[0])
ones_test = np.ones(xt.shape[0])

traindata=np.insert(x,0,ones_train,axis=1)
testdata=np.insert(xt,0,ones_test,axis=1)

N=traindata.shape[0]
M=traindata.shape[1]

loss=[]
error_train=[]
error_test=[]
results_list = []

k=[10,30,100,300,500]

#Iterative Logistic Regression to train a Logitboost classifier with
various values of k, and collect training and test errors for each k
for l in range(len(k)):
    # Initialize variables and arrays
    x=traindata
    xt=testdata
    β=np.zeros(M)
 # Iterate through the specified range for k
    for i in range(0,k[l]):
        h=np.dot(x,β)
        p=1.0/(1.0+np.exp(-2*h))
        w=p*(1.0-p)
        z=0.5*(y+1)-p
        z[w==0]=0
        z[w!=0]=z[w!=0]/w[w!=0]


         # Initialize arrays and perform computations
        coeff=np.zeros((2,M-1))
        newloss=np.zeros(M-1)
```

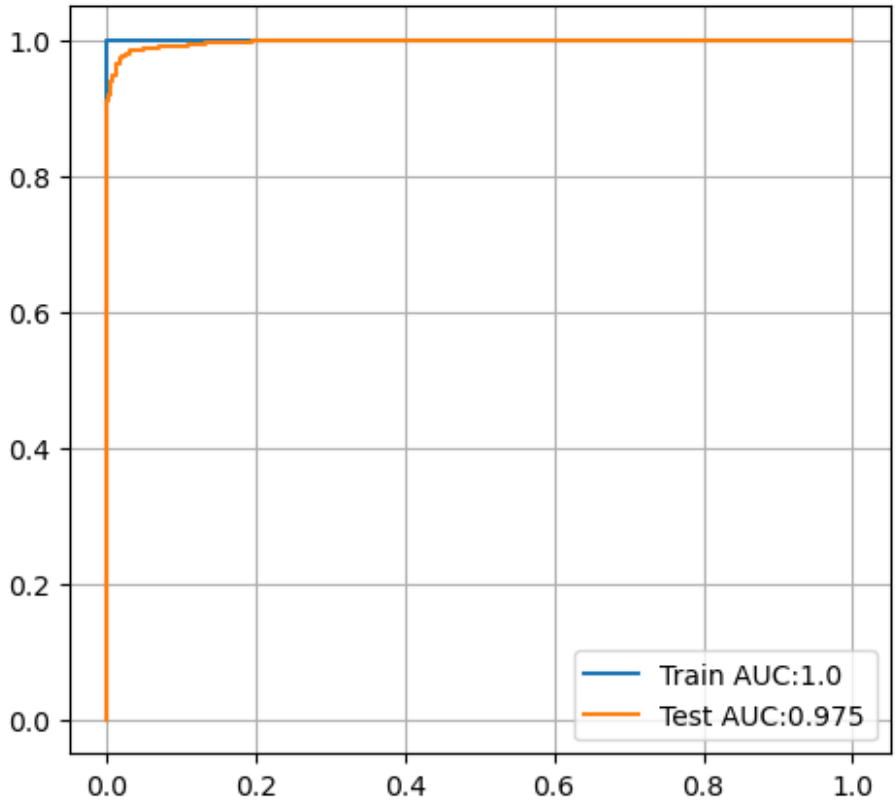
```
        for j in range(0,M-1):
            xj = x[:,j+1]
            a=np.sum(w)
            b=np.sum(w*xj)
            c=np.sum(w*xj**2)
            d=np.sum(w*z)
            e=np.sum(w*xj*z)

             # Compute βj based on specific conditions
            if(a*c-b**2)==0:
                βj=np.array([d/a,0])
            else:
                βj=np.array([c*d-b*e,a*e-b*d])/(a*c-b**2)
            hj=h+0.5*(βj[0]+βj[1]*xj)
            lossj=np.sum(np.log(1+np.exp(-2*y*hj)))
            coeff[:,j]=βj
            newloss[j]=lossj

        minloss=np.argmin(newloss)
        β[0]=β[0]+0.5*coeff[0,minloss]
        β[minloss+1]=β[minloss+1]+0.5*coeff[1][minloss]

        # Record loss if k is 300
        if k[l]==300:
            loss.append(newloss[minloss])

    p=np.dot(x,β)
    pred=np.where(p>0.0,1,-1)
    err=1-np.mean(pred==y)
    error_train.append(err)

    p_test=np.dot(xt,β)
    pred_test=np.where(p_test>0.0,1,-1)
    err_test=1-np.mean(pred_test==yt)
    error_test.append(err_test)
     # Append results to the results_list
    results_list.append({"k": k[l], "Training Error": err, "Test
Error": err_test})
   # Concatenate the results into a DataFrame
results_df = pd.concat([pd.DataFrame(result, index=[0]) for result in
results_list], ignore_index=True)

#Show and present outcomes: Inaccuracies in classification
print("Misclassification Errors:")
print(results_df)
# Display the plot btw iteration vs train loss
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(range(1,501),loss)
```

```
plt.xlabel('Iteration Number')
plt.ylabel('Training Loss')
plt.grid()
plt.show()
# show the plot btw features vs misclass error
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(k,error_train,label='Train')
plt.plot(k,error_test,label='Test')
plt.xlabel('Features K')
plt.ylabel('Misclassification Error')
plt.legend()
plt.grid()
plt.show()
# Plot the ROC curve for test data and label the AUC
train_fpr,train_tpr,_=roc_curve(y,p)
test_fpr,test_tpr,_=roc_curve(yt,p_test)
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(train_fpr,train_tpr,label="Train
AUC:"+str(roc_auc_score(y,pred)))
plt.plot(test_fpr,test_tpr,label="Test
AUC:"+str(roc_auc_score(yt,pred_test)))
plt.legend()
plt.grid()
plt.show()
```

```
Misclassification Errors:
     k  Training Error  Test Error
0   10        0.074667       0.087
1   30        0.033667       0.038
2  100        0.012500       0.030
3  300        0.000000       0.027
4  500        0.000000       0.025
```


2500
2000
1500
1000
500
0
Training Loss
0
100
200
300
400
500
Iteration Number


Train
Test
0.08
0.06
0.04
0.02
0.00
Misclassification Error
0
100
200
300
400
500
Features K

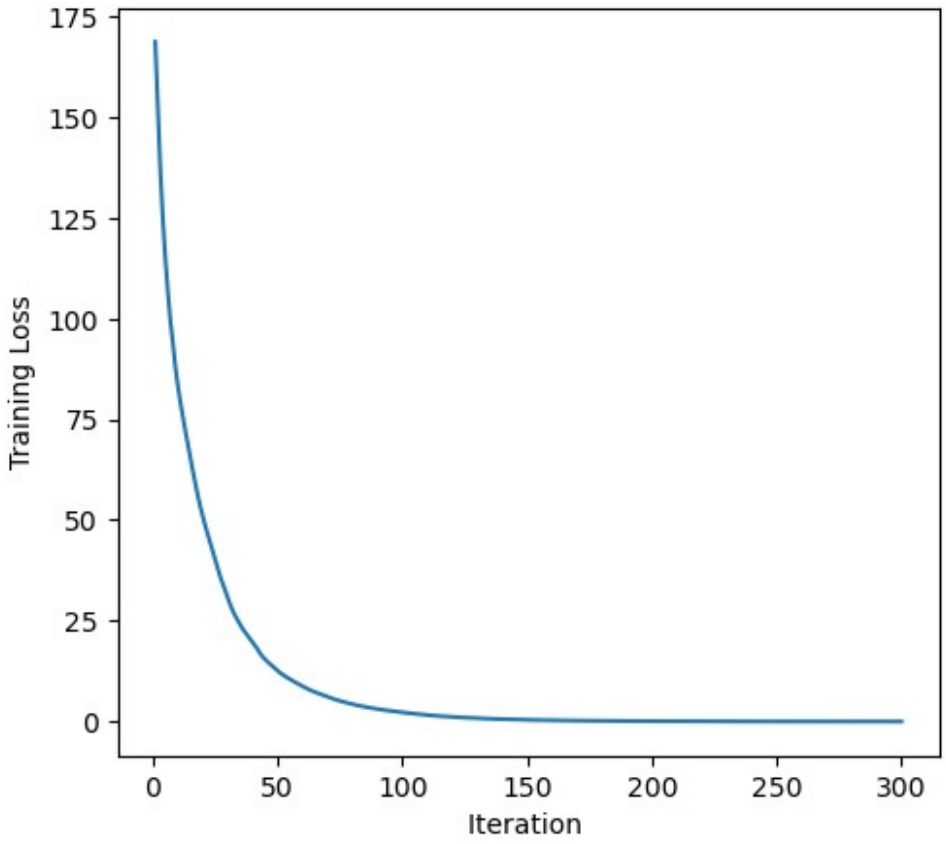

1.0
0.8
0.6
0.4
0.2
0.0
Train AUC:1.0
Test AUC:0.975
0.0
0.2
0.4
0.6
0.8
1.0

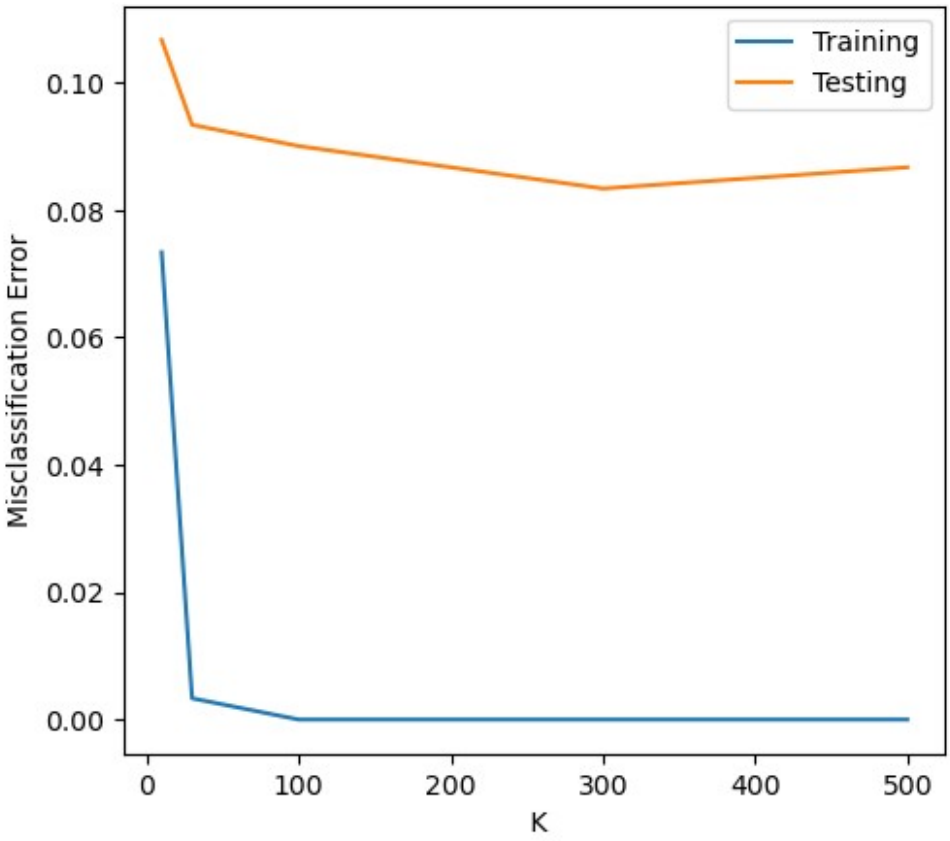
```
# Question - 2
#Import necessary libraries
import pandas as pd
from sklearn.metrics import roc_curve,roc_auc_score,accuracy_score
import numpy as np
import matplotlib.pyplot as plt

#Load the train vs test data
x = np.genfromtxt('dexter_train.csv', delimiter=',')
y = np.loadtxt('dexter_train.labels')
xt = np.genfromtxt('dexter_valid.csv' , delimiter=',')
yt = np.loadtxt('dexter_valid.labels')

#Data preprocessing and initializng variables
ones_train = np.ones(x.shape[0])
ones_test = np.ones(xt.shape[0])

traindata=np.insert(x,0,ones_train,axis=1)
testdata=np.insert(xt,0,ones_test,axis=1)

N=traindata.shape[0]
M=traindata.shape[1]

loss=[]
error_train=[]
error_test=[]
results_list = []

k=[10,30,100,300,500]

##Iterative Logistic Regression to train a Logitboost classifier with
various values of k, and collect training and test errors for each k
for l in range(len(k)):
    # Initialize variables and arrays
    x=traindata
    xt=testdata
    β=np.zeros(M)
 # Iterate through the specified range for k
    for i in range(0,k[l]):
        h=np.dot(x,β)
        p=1.0/(1.0+np.exp(-2*h))
        w=p*(1.0-p)
        z=0.5*(y+1)-p
        z[w==0]=0
        z[w!=0]=z[w!=0]/w[w!=0]
        # Initialize arrays and perform computations
        coeff=np.zeros((2,M-1))
        newloss=np.zeros(M-1)
        # Compute βj based on specific conditions
        for j in range(0,M-1):
```

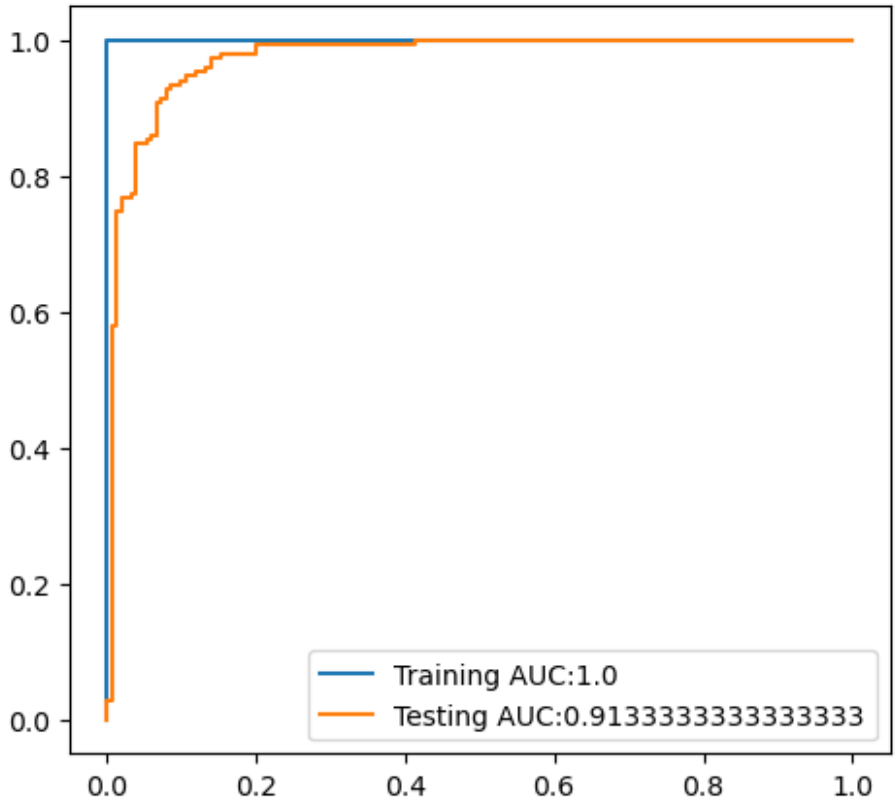
```
            xj = x[:,j+1]
            a=np.sum(w)
            b=np.sum(w*xj)
            c=np.sum(w*xj**2)
            d=np.sum(w*z)
            e=np.sum(w*xj*z)
            if(a*c-b**2)==0:
                βj=np.array([d/a,0])
            else:
                βj=np.array([c*d-b*e,a*e-b*d])/(a*c-b**2)
            hj=h+0.5*(βj[0]+βj[1]*xj)
            lossj=np.sum(np.log(1+np.exp(-2*y*hj)))
            coeff[:,j]=βj
            newloss[j]=lossj

        minloss=np.argmin(newloss)
        β[0]=β[0]+0.5*coeff[0,minloss]
        β[minloss+1]=β[minloss+1]+0.5*coeff[1][minloss]
       # Record loss if k is 300
        if k[l]==300:
            loss.append(newloss[minloss])

    p=np.dot(x,β)
    pred=np.where(p>0.0,1,-1)
    err=1-np.mean(pred==y)
    error_train.append(err)

    p_test=np.dot(xt,β)
    pred_test=np.where(p_test>0.0,1,-1)
    err_test=1-np.mean(pred_test==yt)
    error_test.append(err_test)
 # Append results to the results_list
    results_list.append({"k": k[l], "Training Error": err, "Test
Error": err_test})
   # Concatenate the results into a DataFrame
results_df = pd.concat([pd.DataFrame(result, index=[0]) for result in
results_list], ignore_index=True)

#Show and present outcomes: Inaccuracies in classification
print("Misclassification Error:")
print(results_df)
# Display the plot btw iteration vs train loss
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(range(1,301),loss)
plt.xlabel('Iteration ')
plt.ylabel('Training Loss')
plt.show()
# show the plot btw features vs misclass error
plt.figure(figsize=(12,5))
```

```
plt.subplot(1,2,1)
plt.plot(k,error_train,label='Training')
plt.plot(k,error_test,label='Testing')
plt.xlabel(' K')
plt.ylabel('Misclassification Error')
plt.legend()
plt.show()
# Plot the ROC curve for test data and label the AUC
train_fpr,train_tpr,_=roc_curve(y,p)
test_fpr,test_tpr,_=roc_curve(yt,p_test)
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(train_fpr,train_tpr,label="Training
AUC:"+str(roc_auc_score(y,pred)))
plt.plot(test_fpr,test_tpr,label="Testing
AUC:"+str(roc_auc_score(yt,pred_test)))
plt.legend()
plt.show()
```

```
Misclassification Error:
     k  Training Error  Test Error
0   10        0.073333    0.106667
1   30        0.003333    0.093333
2  100        0.000000    0.090000
3  300        0.000000    0.083333
4  500        0.000000    0.086667
```


175
150
125
100
75
50
25
0
Training Loss
0
50
100
150
200
250
300
Iteration


Training
Testing
0.10
0.08
0.06
0.04
0.02
0.00
Misclassification Error
0
100
200
300
400
500
K

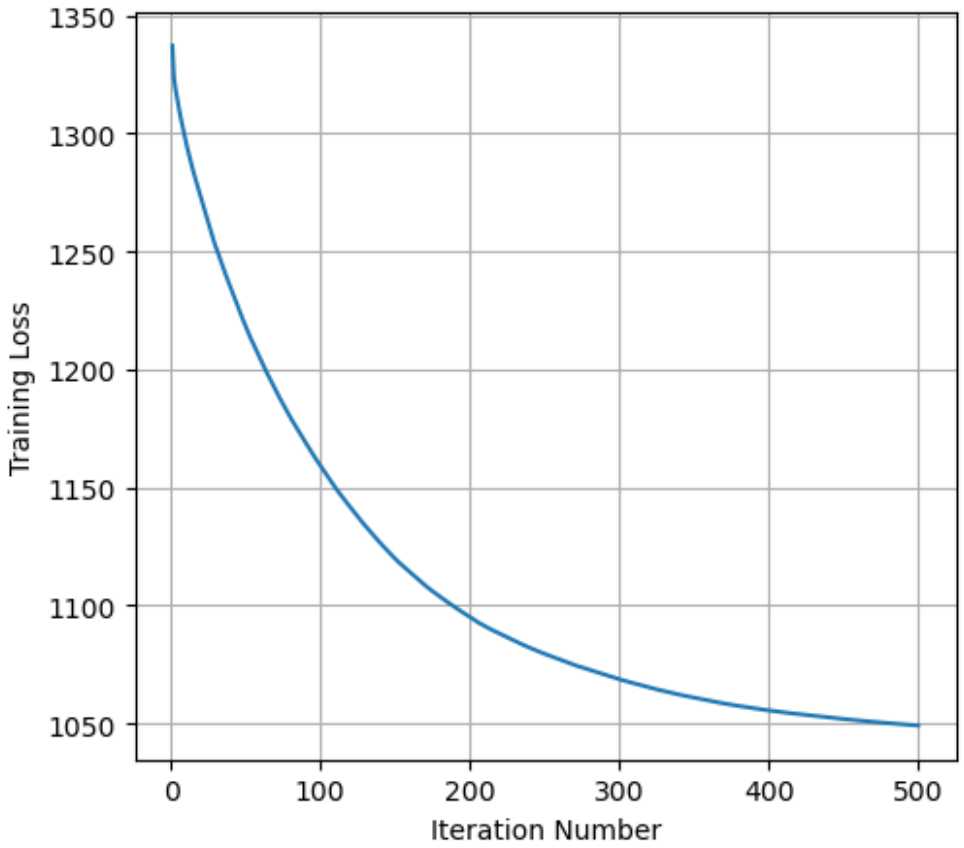

1.0
0.8
0.6
0.4
0.2
0.0
Training AUC:1.0
Testing AUC:0.9133333333333333
0.0
0.2
0.4
0.6
0.8
1.0

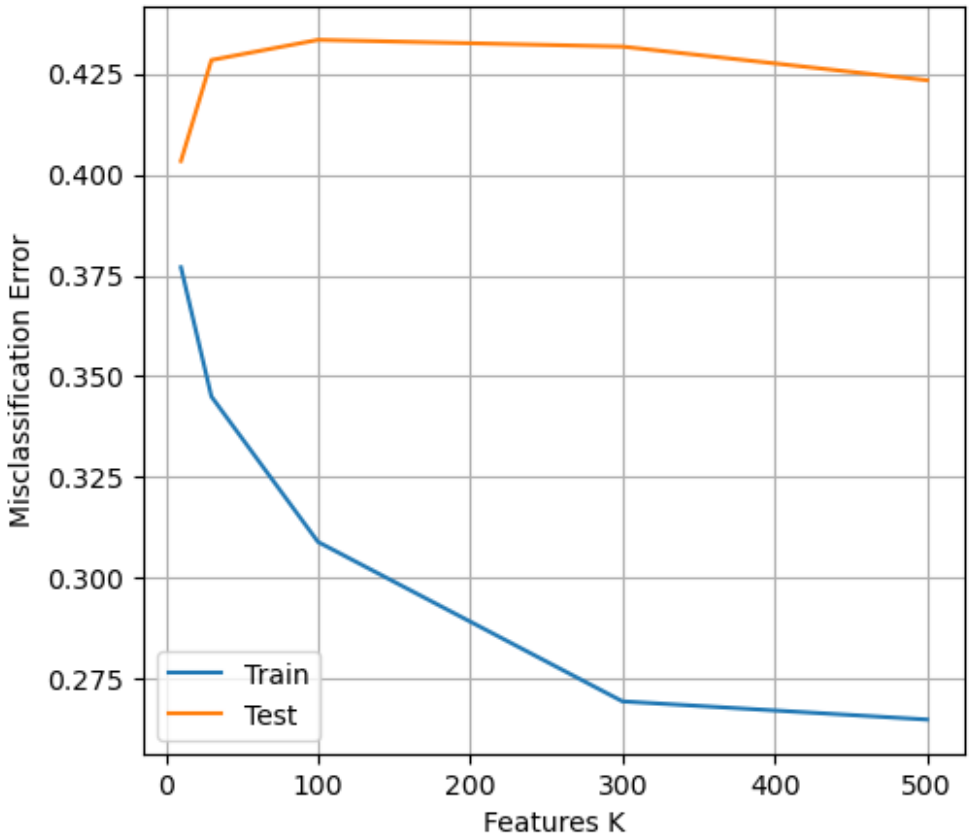
```
# Question - 3
#Import necessary libraries
import pandas as pd
from sklearn.metrics import roc_curve,roc_auc_score,accuracy_score
import numpy as np
import matplotlib.pyplot as plt

#Load the training and validation data
x = np.loadtxt('madelon_train.data')
y = np.loadtxt('madelon_train.labels')
xt = np.loadtxt('madelon_valid.data')
yt = np.loadtxt('madelon_valid.labels')

#Data preprocessing and initializng variables
ones_train = np.ones(x.shape[0])
ones_test = np.ones(xt.shape[0])

traindata=np.insert(x,0,ones_train,axis=1)
testdata=np.insert(xt,0,ones_test,axis=1)

N=traindata.shape[0]
M=traindata.shape[1]

loss=[]
error_train=[]
error_test=[]
results_list = []

k=[10,30,100,300,500]

#Iterative Logistic Regression to train a Logitboost classifier with
various values of k, and collect training and test errors for each k
for l in range(len(k)):
    # Initialize variables and arrays
    x=traindata
    xt=testdata
    β=np.zeros(M)
 # Iterate through the specified range for k
    for i in range(0,k[l]):
        h=np.dot(x,β)
        p=1.0/(1.0+np.exp(-2*h))
        w=p*(1.0-p)
        z=0.5*(y+1)-p
        z[w==0]=0
        z[w!=0]=z[w!=0]/w[w!=0]

         # Initialize arrays and perform computations
        coeff=np.zeros((2,M-1))
        newloss=np.zeros(M-1)
```

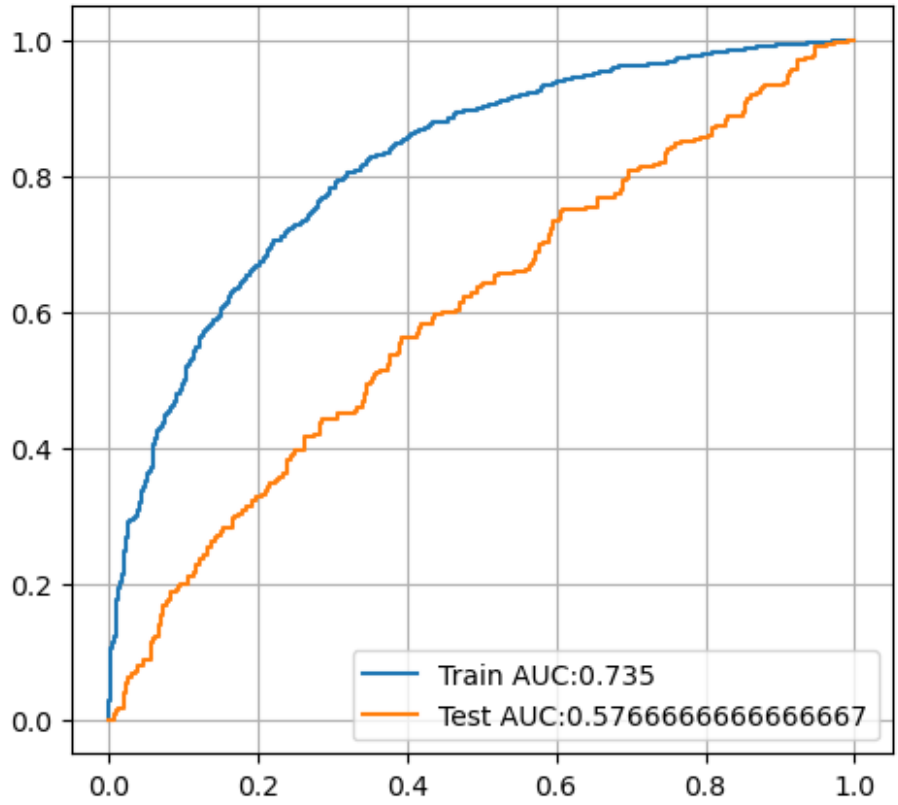
```
        for j in range(0,M-1):
            xj = x[:,j+1]
            a=np.sum(w)
            b=np.sum(w*xj)
            c=np.sum(w*xj**2)
            d=np.sum(w*z)
            e=np.sum(w*xj*z)

             # Compute βj based on specific conditions
            if(a*c-b**2)==0:
                βj=np.array([d/a,0])
            else:
                βj=np.array([c*d-b*e,a*e-b*d])/(a*c-b**2)
            hj=h+0.5*(βj[0]+βj[1]*xj)
            lossj=np.sum(np.log(1+np.exp(-2*y*hj)))
            coeff[:,j]=βj
            newloss[j]=lossj

        minloss=np.argmin(newloss)
        β[0]=β[0]+0.5*coeff[0,minloss]
        β[minloss+1]=β[minloss+1]+0.5*coeff[1][minloss]
        # Record loss if k is 300
        if k[l]==300:
            loss.append(newloss[minloss])

    p=np.dot(x,β)
    pred=np.where(p>0.0,1,-1)
    err=1-np.mean(pred==y)
    error_train.append(err)

    p_test=np.dot(xt,β)
    pred_test=np.where(p_test>0.0,1,-1)
    err_test=1-np.mean(pred_test==yt)
    error_test.append(err_test)
     # Append results to the results_list
    results_list.append({"k": k[l], "Training Error": err, "Test
Error": err_test})
   # Concatenate the results into a DataFrame
results_df = pd.concat([pd.DataFrame(result, index=[0]) for result in
results_list], ignore_index=True)

#Show and present outcomes: Inaccuracies in classification
print("Misclassification Errors:")
print(results_df)
# Display the plot btw iteration vs train loss
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(range(1,501),loss)
plt.xlabel('Iteration Number')
plt.ylabel('Training Loss')
```

```
plt.grid()
plt.show()
# show the plot btw features vs misclass error
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(k,error_train,label='Train')
plt.plot(k,error_test,label='Test')
plt.xlabel('Features K')
plt.ylabel('Misclassification Error')
plt.legend()
plt.grid()
plt.show()
# Plot the ROC curve for test data and label the AUC
train_fpr,train_tpr,_=roc_curve(y,p)
test_fpr,test_tpr,_=roc_curve(yt,p_test)
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(train_fpr,train_tpr,label="Train
AUC:"+str(roc_auc_score(y,pred)))
plt.plot(test_fpr,test_tpr,label="Test
AUC:"+str(roc_auc_score(yt,pred_test)))
plt.legend()
plt.grid()
plt.show()
```

```
Misclassification Errors:
     k  Training Error  Test Error
0   10          0.3770    0.403333
1   30          0.3450    0.428333
2  100          0.3090    0.433333
3  300          0.2695    0.431667
4  500          0.2650    0.423333
```

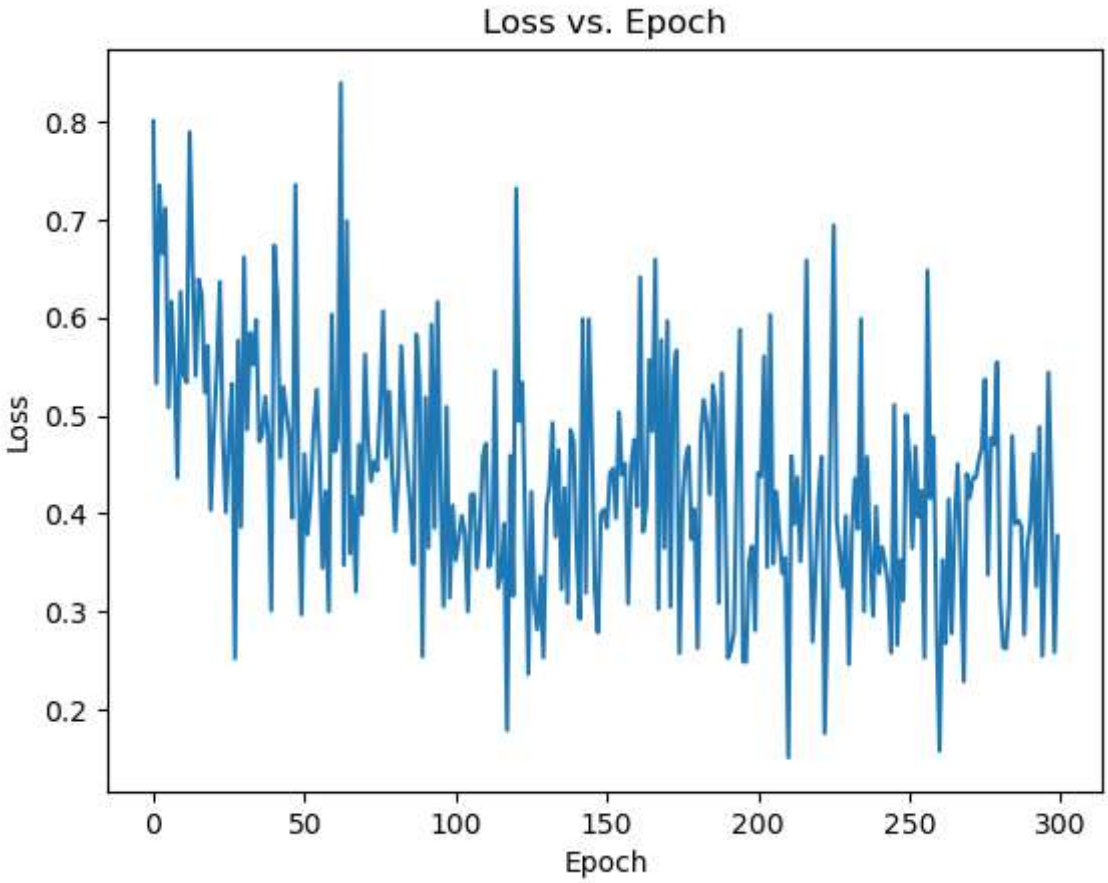

1350
1300
1250
1200
1150
1100
1050
Training Loss
0
100
200
300
400
500
Iteration Number

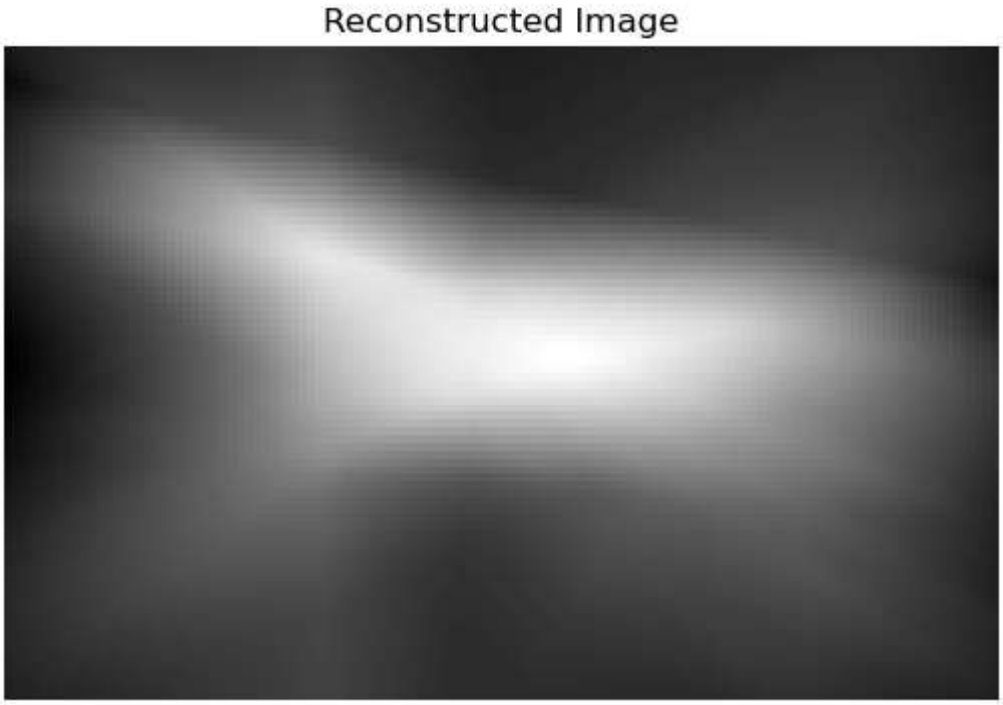

0.425
0.400
0.375
0.350
0.325
0.300
0.275
Misclassification Error
Train
Test
0
100
200
300
400
500
Features K


1.0
0.8
0.6
0.4
0.2
0.0
Train AUC:0.735
Test AUC:0.5766666666666667
0.0
0.2
0.4
0.6
0.8
1.0

In [11]:
```
# AML - 7(a)
# attach required Libraries
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Load picture of horse
img = Image.open("horse025b.png")
img_array = np.asarray(img)
X = []
y = []

# Create X, y data points from the image
for i in range(img_array.shape[1]): # rows
    for j in range(img_array.shape[0]): # columns
        X.append([i, j])
        y.append(img_array[j, i])

X = np.array(X)
y = np.array(y)

# Normalize X and y
X_mean, X_std = np.mean(X, axis=0), np.std(X, axis=0)
y_mean, y_std = np.mean(y), np.std(y)
X = (X - X_mean) / X_std
y = (y - y_mean) / y_std

import torch
import torch.nn as nn

# Define a simple neural network model
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(2, 128)
        self.fc2 = nn.Linear(128, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = SimpleNN().to(device)

criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.003)

# Convert data to tensors
X_tensor = torch.FloatTensor(X).to(device)
y_tensor = torch.FloatTensor(y).view(-1, 1).to(device)

losses = []

for epoch in range(300):
    permutation = torch.randperm(X_tensor.size()[0])
```

```
    for i in range(0, X_tensor.size()[0], 64):
        indices = permutation[i:i+64]
        batch_x, batch_y = X_tensor[indices], y_tensor[indices]

        # Forward pass
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)

        # Backward pass and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    losses.append(loss.item())

    if (epoch+1) % 100 == 0:
        for g in optimizer.param_groups:
            g['lr'] = g['lr'] / 2

# Plot the loss
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss vs. Epoch')
plt.show()

#reconstruct the image
reconstructed_img = np.zeros((84, 128))  # Note the changed shape
with torch.no_grad():
    for i in range(128):
        for j in range(84):
            coord = torch.FloatTensor([(i - X_mean[0]) / X_std[0], (j - X_mean[1]) / X
            pixel_value = model(coord).item() * y_std + y_mean
            reconstructed_img[j, i] = pixel_value  # Swap the indices here

#plot the graph
plt.imshow(reconstructed_img, cmap='gray')
plt.title('Reconstructed Image')
plt.axis('off')
plt.show()
```

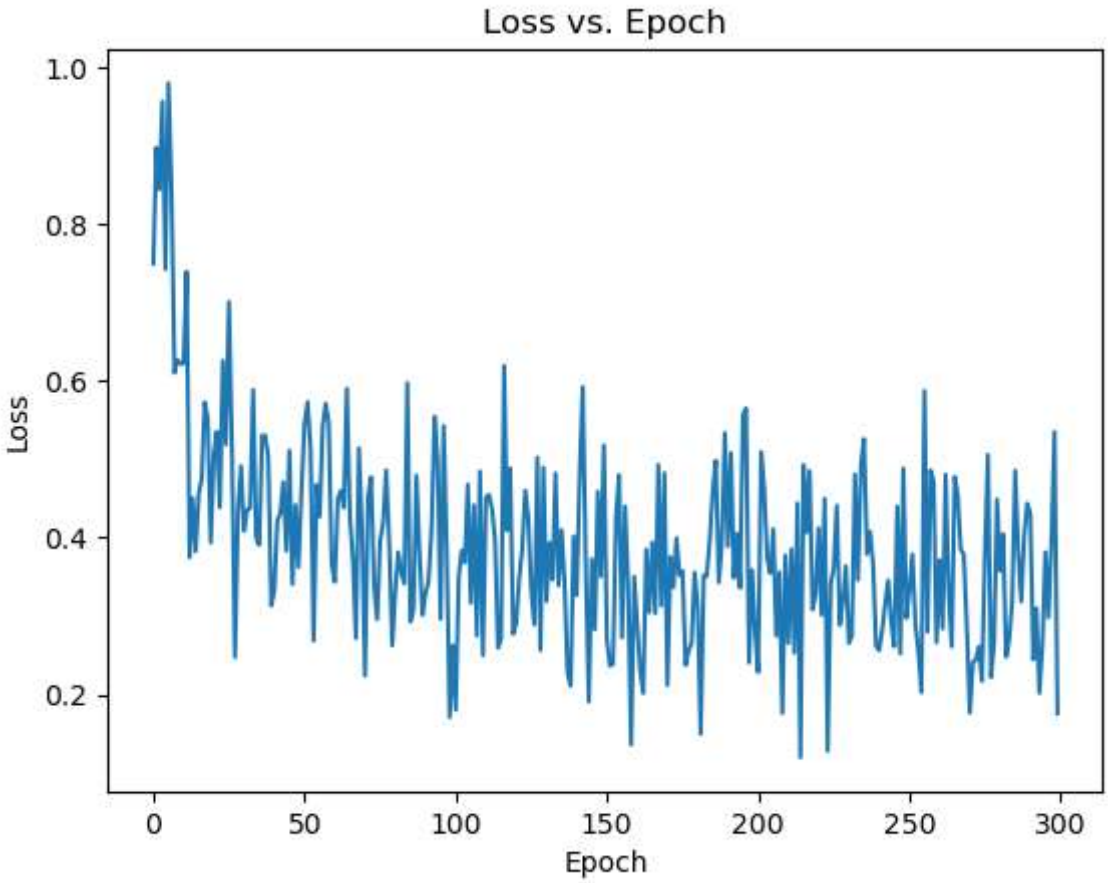


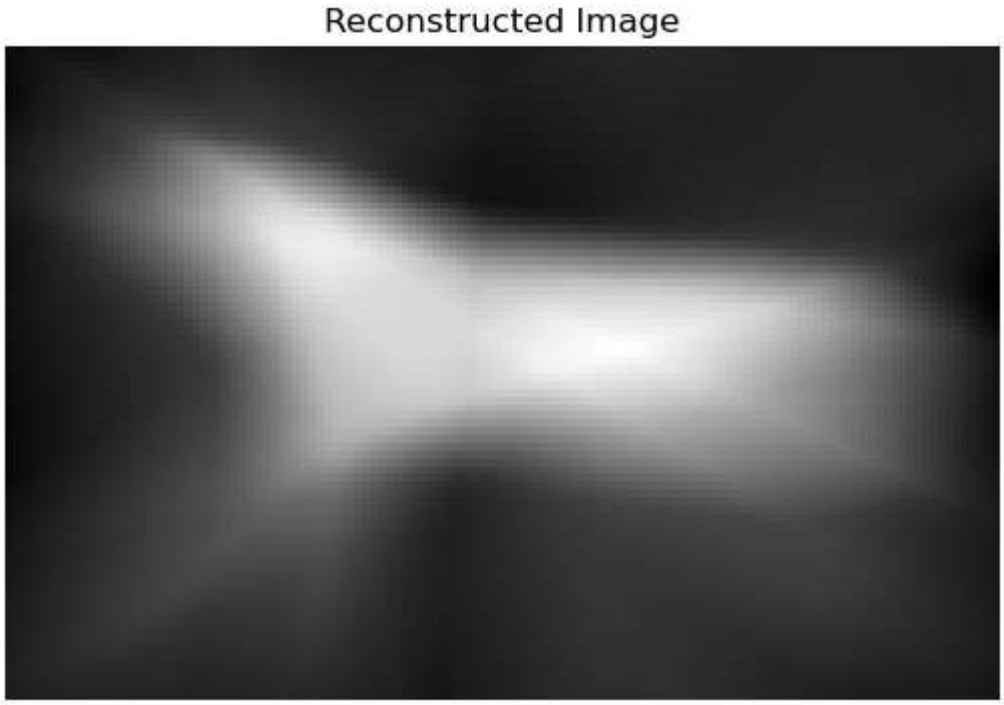


In [7]:
```
# AML - 7(b)

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import torch
import torch.nn as nn
```

```
# Load the image and convert it to arrays
img = Image.open("horse025b.png")
img_array = np.asarray(img)

# Initialize empty lists for data points
X = []
y = []

# Create X, y data points from the image
for i in range(img_array.shape[1]):
    for j in range(img_array.shape[0]):
        X.append([i, j])
        y.append(img_array[j, i])

# Convert lists to numpy arrays
X = np.array(X)
y = np.array(y)

# Normalize X and y
X_mean, X_std = np.mean(X, axis=0), np.std(X, axis=0)
y_mean, y_std = np.mean(y), np.std(y)

X = (X - X_mean) / X_std
y = (y - y_mean) / y_std

# Define the Neural Network with Two Hidden Layers

class TwoLayerNN(nn.Module):
    def __init__(self):
        super(TwoLayerNN, self).__init__()
        self.fc1 = nn.Linear(2, 32)
        self.fc2 = nn.Linear(32, 128)
        self.fc3 = nn.Linear(128, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x


# Training Loop
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = TwoLayerNN().to(device)

criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.003)

X_tensor = torch.FloatTensor(X).to(device)
y_tensor = torch.FloatTensor(y).view(-1, 1).to(device)

losses = []
# Training the model
for epoch in range(300):
    permutation = torch.randperm(X_tensor.size()[0])

    for i in range(0, X_tensor.size()[0], 64):
```

```
        indices = permutation[i:i+64]
        batch_x, batch_y = X_tensor[indices], y_tensor[indices]
     # forward pass and optimize
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
# Backward pass and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    losses.append(loss.item())

    if (epoch+1) % 100 == 0:
        for g in optimizer.param_groups:
            g['lr'] = g['lr'] / 2
#plot the graph
plt.figure()
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss vs. Epoch')
plt.show()
# add the image which is reconstructed
reconstructed_img = np.zeros((84, 128))  # Note the changed shape
with torch.no_grad():
    for i in range(128):
        for j in range(84):
            coord = torch.FloatTensor([(i - X_mean[0]) / X_std[0], (j - X_mean[1]) / X
            pixel_value = model(coord).item() * y_std + y_mean
            reconstructed_img[j, i] = pixel_value  # Swap the indices here

# Plot the reconstructed image
plt.imshow(reconstructed_img, cmap='gray')
plt.title('Reconstructed Image')
plt.axis('off')
plt.show()
```

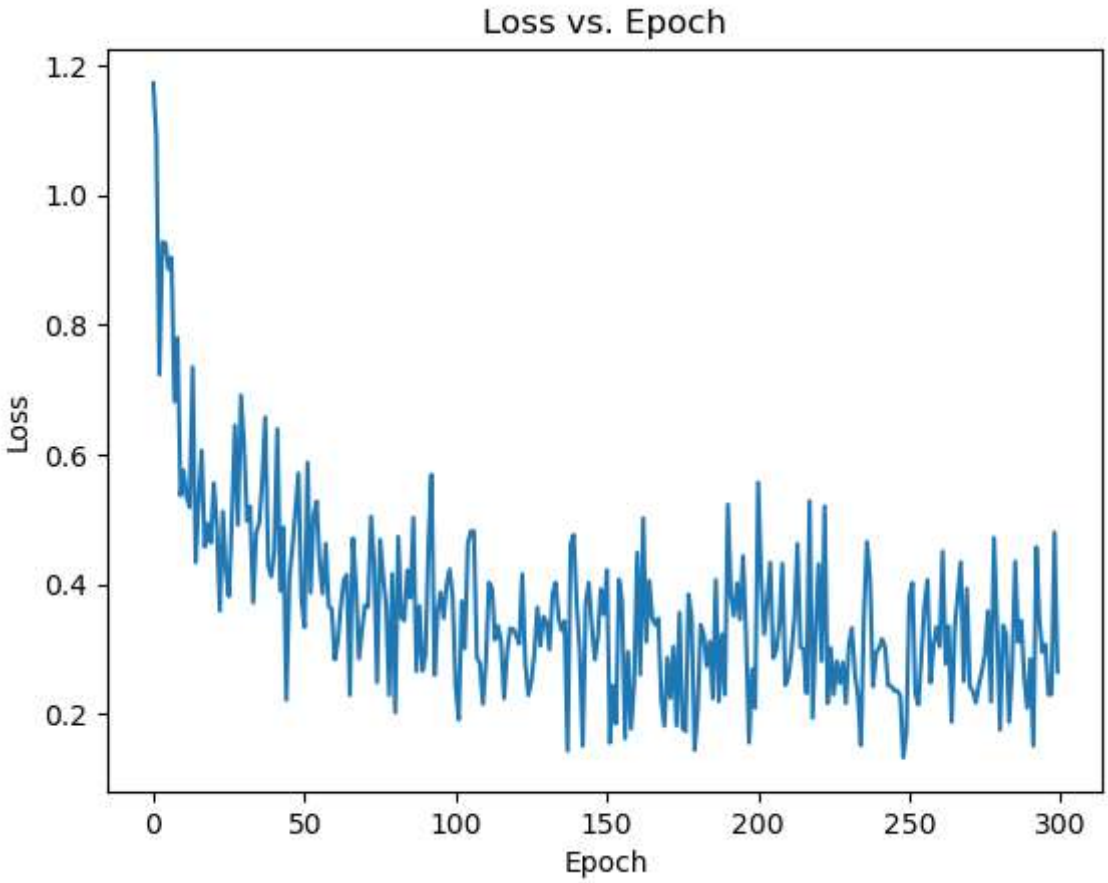


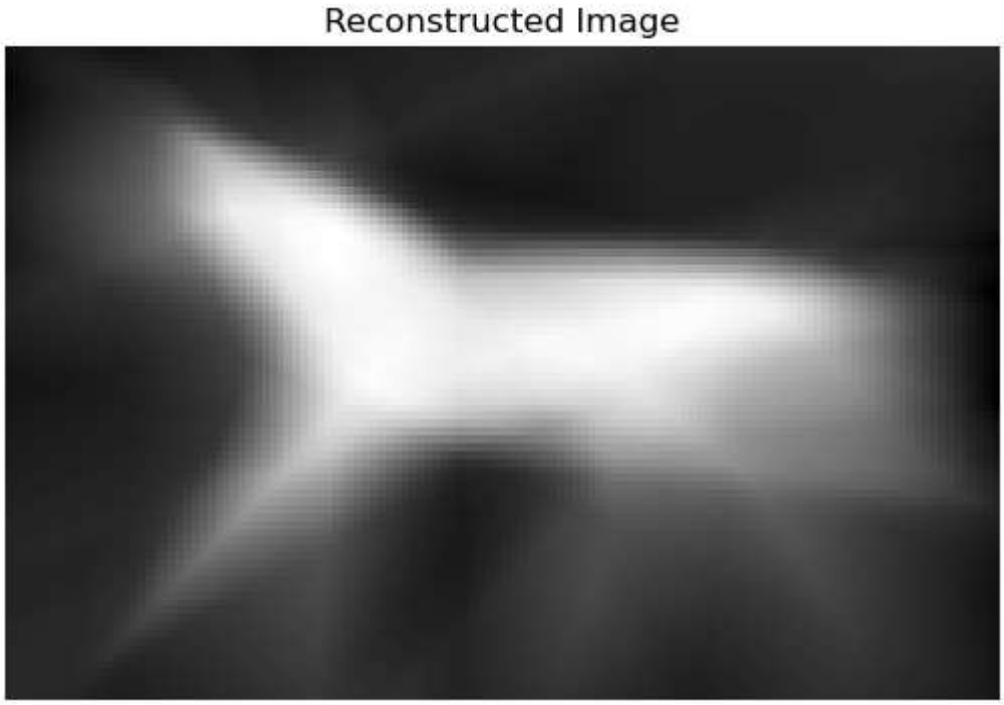


In [8]:
```
# AML - 7(c)

# required libraries for this code
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import torch
import torch.nn as nn
```

```
# Define the Four-Layer Neural Network
class ThreeLayerNN(nn.Module):
    def __init__(self):
        super(ThreeLayerNN, self).__init__()
        self.fc1 = nn.Linear(2, 32)
        self.fc2 = nn.Linear(32, 64)
        self.fc3 = nn.Linear(64, 128)
        self.fc4 = nn.Linear(128, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.relu(x)
        x = self.fc4(x)
        return x
# Load and Process Image

img = Image.open("horse025b.png")
img_array = np.asarray(img)

X = []
y = []

for i in range(img_array.shape[1]):
    for j in range(img_array.shape[0]):
        X.append([i, j])
        y.append(img_array[j, i])

X = np.array(X)
y = np.array(y)

X_mean, X_std = np.mean(X, axis=0), np.std(X, axis=0)
y_mean, y_std = np.mean(y), np.std(y)

X = (X - X_mean) / X_std
y = (y - y_mean) / y_std



device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = ThreeLayerNN().to(device)

criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.003)

X_tensor = torch.FloatTensor(X).to(device)
y_tensor = torch.FloatTensor(y).view(-1, 1).to(device)

losses = []
# Training the model
for epoch in range(300):
    permutation = torch.randperm(X_tensor.size()[0])

    for i in range(0, X_tensor.size()[0], 64):
        indices = permutation[i:i+64]
```

```
        batch_x, batch_y = X_tensor[indices], y_tensor[indices]

        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    losses.append(loss.item())

    if (epoch+1) % 100 == 0:
        for g in optimizer.param_groups:
            g['lr'] = g['lr'] / 2
            # plot the loss
plt.figure()
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss vs. Epoch')
plt.show()

# reconstruct the horse image
reconstructed_img = np.zeros((84, 128))  # Note the changed shape
with torch.no_grad():
    for i in range(128):
        for j in range(84):
            coord = torch.FloatTensor([(i - X_mean[0]) / X_std[0], (j - X_mean[1]) / X
            pixel_value = model(coord).item() * y_std + y_mean
            reconstructed_img[j, i] = pixel_value  # Swap the indices here

plt.imshow(reconstructed_img, cmap='gray')
plt.title('Reconstructed Image')
plt.axis('off')
plt.show()
```

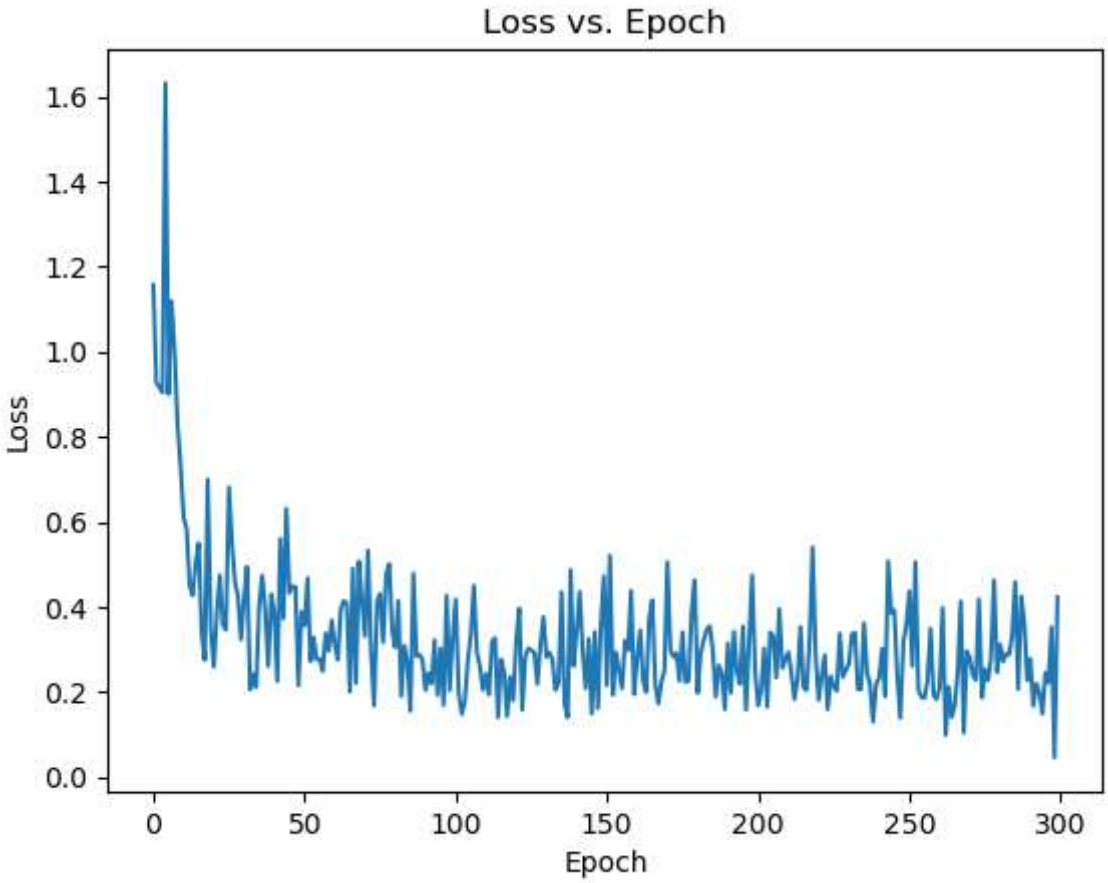


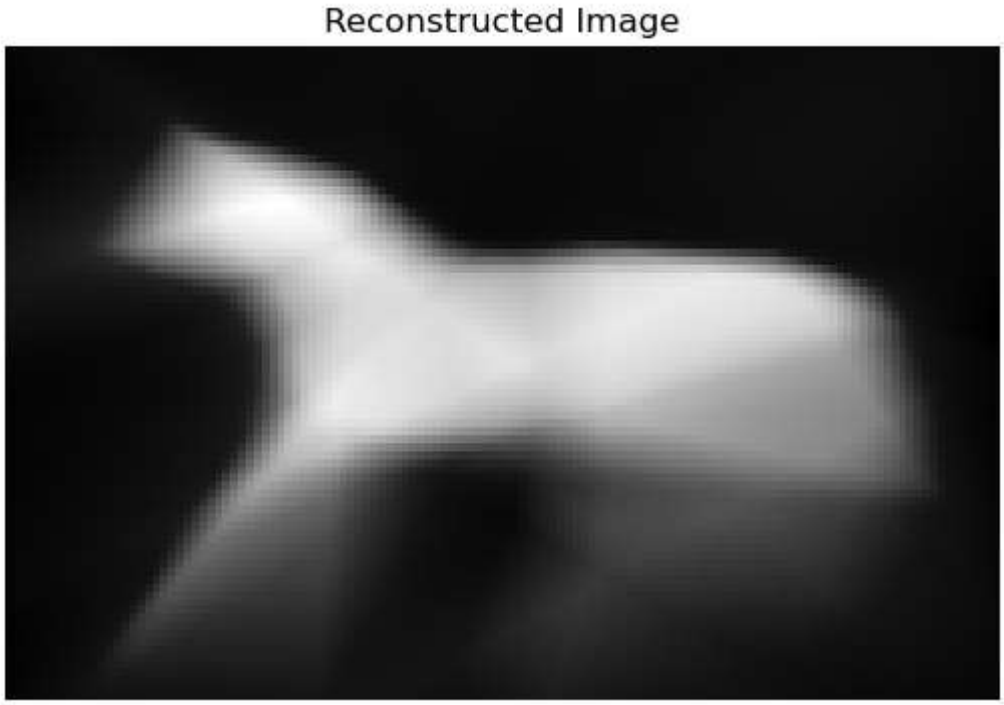


In [9]:
```
# AML 7(d)
# add the libraries which is required
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import torch
import torch.nn as nn
```

```
# Define the Four-Layer Neural Network
class FourLayerNN(nn.Module):
    def __init__(self):
        super(FourLayerNN, self).__init__()
        self.fc1 = nn.Linear(2, 32)
        self.fc2 = nn.Linear(32, 64)
        self.fc3 = nn.Linear(64, 128)
        self.fc4 = nn.Linear(128, 128)
        self.fc5 = nn.Linear(128, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.relu(x)
        x = self.fc4(x)
        x = self.relu(x)
        x = self.fc5(x)
        return x

    # Load and Process Image
img = Image.open("horse025b.png")
img_array = np.asarray(img)

X = []
y = []

for i in range(img_array.shape[1]):
    for j in range(img_array.shape[0]):
        X.append([i, j])
        y.append(img_array[j, i])

X = np.array(X)
y = np.array(y)

X_mean, X_std = np.mean(X, axis=0), np.std(X, axis=0)
y_mean, y_std = np.mean(y), np.std(y)

X = (X - X_mean) / X_std
y = (y - y_mean) / y_std

# Training Loop
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = FourLayerNN().to(device)

criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.003)

X_tensor = torch.FloatTensor(X).to(device)
y_tensor = torch.FloatTensor(y).view(-1, 1).to(device)

losses = []

for epoch in range(300):
    permutation = torch.randperm(X_tensor.size()[0])
```

```
    for i in range(0, X_tensor.size()[0], 64):
        indices = permutation[i:i+64]
        batch_x, batch_y = X_tensor[indices], y_tensor[indices]

        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    losses.append(loss.item())

    if (epoch+1) % 100 == 0:
        for g in optimizer.param_groups:
            g['lr'] = g['lr'] / 2
plt.figure()
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss vs. Epoch')
plt.show()

reconstructed_img = np.zeros((84, 128))  # Note the changed shape
with torch.no_grad():
    for i in range(128):
        for j in range(84):
            coord = torch.FloatTensor([(i - X_mean[0]) / X_std[0], (j - X_mean[1]) / X
            pixel_value = model(coord).item() * y_std + y_mean
            reconstructed_img[j, i] = pixel_value  # Swap the indices here

            #plot the reconstructed image of the horse
plt.imshow(reconstructed_img, cmap='gray')
plt.title('Reconstructed Image')
plt.axis('off')
plt.show()
```


Loss vs. Epoch
Loss
Epoch
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300


Reconstructed Image

In [ ]:

APPLIED MACHINE LEARNING ASSIGNMENT - 8

Question _ 1(a)

In [2]:
```
# Add required libraries
import numpy as np
```

In [3]:
```
# Loaing the required dataset
X = np.loadtxt("hmm_pb1.csv", delimiter=",")

# adding required variable
# Initial state probabilities
pi = np.array([0.5,0.5])
# State transition probabilities
A = np.array([[0.95,0.05],[0.05,0.95]])
# Emission probabilities
B = np.array([[1/6,1/6,1/6,1/6,1/6,1/6],[1/10,1/10,1/10,1/10,1/10,1/2]])
# printing the values
print(pi,"\n\n",A,"\n\n",B)
```

```
[0.5 0.5]

 [[0.95 0.05]
 [0.05 0.95]]

 [[0.16666667 0.16666667 0.16666667 0.16666667 0.16666667 0.16666667]
 [0.1        0.1        0.1        0.1        0.1        0.5       ]]
```

In [4]:
```
# VITERBI FUNCTION
def viterbi(X, pi, A, B):
    # Initialize matrices for storing values and pointers
    C = np.zeros([2, X.shape[0]])  # Stores the values
    ptr = np.zeros([2, X.shape[0]-1])  # Stores the pointers
    PTR = np.zeros(X.shape[0])  # Stores the final path
    # Iterate through the sequence
    for i in range(X.shape[0]):
        # For the first observation
        if (i == 0):
            C[:,i] = np.log(B[:, int(X[i]-1)]*pi)
        else:
            # For subsequent observations
            C[:,i] = np.log(B[:, int(X[i]-1)]) + np.max(np.log(A)+C[:,i-1], axis=1)
            ptr[:,i-1] = np.argmax(np.log(A)+C[:,i-1], axis=1)

    # Backtrack for finding the most likely sequences
    for j in range(PTR.shape[0]-1, -1, -1):
        if (j == PTR.shape[0]-1):
            PTR[j] = np.argmax(C[:, X.shape[0]-1])
        else:
            PTR[j] = ptr[int(PTR[j+1]), j]

    return PTR
```

In [5]:
```
# Run the Viterbi algorithm
# Adding 1 to convert states to 1 and 2
Y = viterbi(X, pi, A, B) + 1
# printing the result values
print("\nY (1 = 'FAIR' AND 2 = 'LAODED')\n\n", Y)
```

```
Y (1 = 'FAIR' AND 2 = 'LAODED')

 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]
```

Question - 1(b)

In [7]:
```
# FORWARD FUNCTION
def forward(X, pi, A, B):
    # Initialize the alpha matrix

    alpha = np.zeros([2, X.shape[0]])

    # Iterate through the sequence
    for i in range(X.shape[0]):
        # For the first observation
        if (i == 0):
            # Initial alpha values
            alpha[:,i] = B[:, int(X[i])-1] * pi
        else:
            # For subsequent observations
            alpha[:,i] = B[:,int(X[i])-1] * np.matmul(alpha[:,i-1], A) # Update alpha values
            alpha[:,i] = alpha[:,i] / np.sum(alpha[:,i]) # Norm the alpha values

    return alpha
```

In [8]:
```
# BACKWARD FUNCTION
def backward(X, pi, A, B):
     # Initialize the beta matrix
    beta = np.zeros([2, X.shape[0]])


    # Iterate through the sequence in reverse
    for i in range(X.shape[0]-1, -1, -1):
         # For the last observation
        if ( i == X.shape[0]-1):
            # Set the beta values to 1 for the last observation
            beta[:,i] = np.ones(2)
        else:
            # Update beta values
            beta[:,i] = np.matmul(beta[:,i+1]*B[:, int(X[i+1]-1)], A.T)

    return beta
```

In [9]:
```
# Calculate the ratio of alpha values
my_alpha = forward(X, pi, A, B)[0,134] / forward(X, pi, A, B)[1,134]
# Print the values of alpha for time step 135
print("ALPHA 1-135:", forward(X, pi, A, B)[0,134])
print("ALPHA 2-135:", forward(X, pi, A, B)[1,134])
# Print the ratio of alpha values
print("\nALPHA 1-135 / ALPHA 2-135 =", my_alpha)
```

```
ALPHA 1-135: 0.5810349174411512
ALPHA 2-135: 0.41896508255884873

ALPHA 1-135 / ALPHA 2-135 = 1.3868337520932614
```

In [10]:
```
# Calculate the ratio of beta values
my_beta = backward(X, pi, A, B)[0,134] / backward(X, pi, A, B)[1,134]
# Print the values of beta for time step 135
print("BETA 1-135:", backward(X, pi, A, B)[0,134])
print("BETA 2-135:", backward(X, pi, A, B)[1,134])
# Print the ratio of beta values
print("\nBETA 1-135 / BETA 2-135 =", my_beta)
```

```
BETA 1-135: 1.2224053078693514e-96
BETA 2-135: 1.4190400700799443e-96

BETA 1-135 / BETA 2-135 = 0.8614311418284931
```

Question - 2

In [1]:
```
# loading the required libraries
import numpy as np
```

In [2]:
```
# adding required dataset values
X = np.loadtxt("hmm_pb1.csv", delimiter=",")

# adding required variables
pi = np.array([0.5,0.5])
A = np.array([[0.95,0.05],[0.05,0.95]])
B = np.array([[1/6,1/6,1/6,1/6,1/6,1/6],[1/10,1/10,1/10,1/10,1/10,1/2]])

# printing the values
print(pi,"\n\n",A,"\n\n",B)
```

```
[0.5 0.5]

 [[0.95 0.05]
 [0.05 0.95]]

 [[0.16666667 0.16666667 0.16666667 0.16666667 0.16666667 0.16666667]
 [0.1        0.1        0.1        0.1        0.1        0.5       ]]
```

In [3]:
```
# FORWARD FUNCTION
def forward(X, A, B, pi):
    # Initialize the alpha matrix
    alpha = np.zeros([X.shape[0], 2])
     # Iterate through the sequence
    for i in range(X.shape[0]):
         # For the first observation
        if (i == 0):

            alpha[i,:] = B[int(X[i]-1),:] * pi # Initial alpha values
        else:
            # For subsequent observations
            alpha[i,:] = B[int(X[i])-1,:] * np.matmul(alpha[i-1,:], A) # updating the alpha values
            alpha[i,:] = alpha[i,:] / np.sum(alpha[i,:]) # Normalizing the alpha values

    return alpha
```

In [4]:
```
# BACKWARD FUNCTION
def backward(X, A, B, pi):
    # Initialize the beta matrix
    beta = np.zeros([X.shape[0], 2])
     # Iterate through the sequence in reverse
    for i in range(X.shape[0]-1,-1,-1):
        # For the last observation
        if ( i == X.shape[0]-1):
            # Set the beta values to 1 for the last observation
            beta[i,:] = np.ones(2)
        else:
            # For subsequent observations
            beta[i,:] = np.matmul(beta[i+1,:]*B[int(X[i+1]-1),:], A.T) # updating the beta values
            beta[i,:] = beta[i,:] / np.sum(beta[i,:]) # Normalizing the beta values

    return beta
```

In [5]:
```
# Baum-Welch E-step function
def Baum_Welch_E_Step(A, B, alpha, beta):
    # Initialize matrices for storing kesi and gamma values
    kesi = np.zeros([alpha.shape[0]-1, 2, 2])
    gamma = np.zeros([alpha.shape[0], 2])
     # Iterate through the sequence
    for i in range(alpha.shape[0]-1):
        pro4 = beta[i+1,:] * B[int(X[i+1]-1),:]
        pro3 = np.matmul(np.expand_dims(alpha[i,:],axis=1), np.expand_dims(pro4,axis=1).T) * A
        kesi[i,:] = (1.0*pro3) / np.sum(pro3)  # Calculate kesi values
        gamma[i,:] = (1.0 * alpha[i,:] * beta[i,:]) / np.sum(alpha[i,:]*beta[i,:]) # Calculate gamma value
        pro5 = np.sum(alpha[alpha.shape[0]-1,:] * beta[alpha.shape[0]-1,:])

    gamma[alpha.shape[0]-1,:] = (1.0 * alpha[alpha.shape[0]-1,:] * beta[alpha.shape[0]-1,:]) / pro5

    return kesi, gamma
```

In [6]:
```
# Baum-Welch M-step function
def Baum_Welch_M_Step(kesi, gamma):
    # Update initial state probabilities
    pi = gamma[0,:]
    # Update state transition matrix A
    A = np.sum(kesi,axis=0) / np.expand_dims(np.sum(gamma[:-1,:],axis=0),axis=1)
     # Initialize matrix B
    B = np.zeros([6,2])
    # Iterate through the range of B
    for i in range(B.shape[0]):
        pro6 = gamma * np.expand_dims(np.where(X == (i+1),1,0),axis=1)
        B[i,:] = (1.0 * np.sum(pro6,axis=0)) / np.sum(gamma,axis=0) # Update emission matrix B

    return pi, A, B
```

In [7]:
```
# Set the convergence criterion and initialize the iteration counter
# Convergence criterion
criteria = 100
# Iteration counter
i = 0
```

In [8]:
```
# Iterate while the convergence criterion is met
while(criteria > 10**-5):
    # Perform the backward and forward algorithms
    beta = backward(X, A, B, pi)
    alpha = forward(X, A, B, pi)
    # Store the current pi, A, and B values
    my_pi, my_A, my_B = pi, A, B
     # Execute the Baum-Welch E-step
    kesi, gamma = Baum_Welch_E_Step(A, B, alpha, beta)
    # Execute the Baum-Welch M-step
    pi, A, B = Baum_Welch_M_Step(kesi, gamma)
    # Adjust small values in pi
    pi[pi < 10**(-100)] = 10 ** (-100)
    # Calculate the convergence criterion
    criteria = np.max(np.abs(my_pi-pi) + np.max(np.abs(my_A-A) + np.max(np.abs(my_B-B))))
    # Increment the iteration counter
    i += 1
```

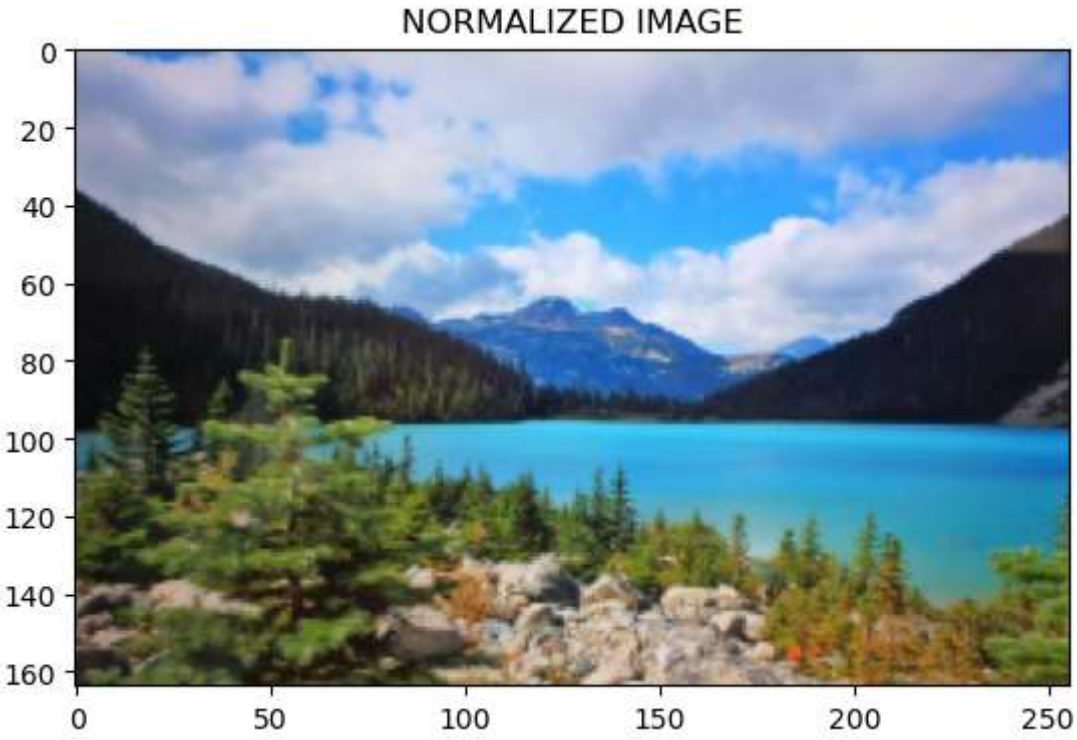
In [9]:
```
# Print the updated pi, A, and B values
 # Print the updated initial state probabilities
print("pi =", pi)
 # Print the updated state transition matrix
print("\nA =", A)
# Print the updated emission matrix
print("\nB =", B)
```

```
pi = [1.e-100 1.e+100]

A = [[0.70872775 0.29127225]
 [0.01192257 0.98807743]]

B = [[0.09529734 0.20085006]
 [0.11215521 0.20567704]
 [0.07141696 0.1933959 ]
 [0.58099717 0.10540076]
 [0.09678099 0.09305344]]
```

Question 1a

In [1]:
```
# IMPORT IMPORTANT LIBRARIES FOR THIS PROBLEM
import numpy as np
from PIL import Image
from scipy.sparse import lil_matrix, csr_matrix
from scipy.sparse.linalg import svds
from sklearn.cluster import SpectralClustering
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
```

In [2]:
```
# LOADING THE GIVEN SCENERY IMAGE
image = Image.open('scene256.jpg')

# Change the dimensions of the image to 164x256.
resized_image = image.resize((256,164))

# Convert the image to a NumPy array and perform normalization.
norm_image = np.array(resized_image) / 256.0

# Show the normalized image.
plt.imshow(norm_image)
plt.title('NORMALIZED IMAGE')
plt.show()
```



In [3]:
```
#Building the affinity matrix.

def construct_affinity_matrix(image, sigma=0.03):
    rows, cols, _ = image.shape
    num_pixels = rows * cols

    # Initialize the affinity matrix with zeros.
    affinity_matrix = lil_matrix((num_pixels, num_pixels))


#Translate 2D coordinates into a 1D index.
    def to_index(r, c):
        return r * cols + c

    # CREATE AFFINITY MATRIX
    for row in range(rows):
        for col in range(cols):
            index = to_index(row, col)
            if row + 1 < rows:
                bottom_index = to_index(row + 1, col)
                difference = image[row, col] - image[row + 1, col]
                affinity_matrix[index, bottom_index] = np.exp(-np.linalg.n
            if col + 1 < cols:
                right_index = to_index(row, col + 1)
                difference = image[row, col] - image[row, col + 1]
                affinity_matrix[index, right_index] = np.exp(-np.linalg.no

    return affinity_matrix

# DISPLAY CONSTRUCTED AFFINITY MATRIX
affinity_matrix = construct_affinity_matrix(norm_image)
affinity_matrix.shape
```

Out[3]: (41984, 41984)

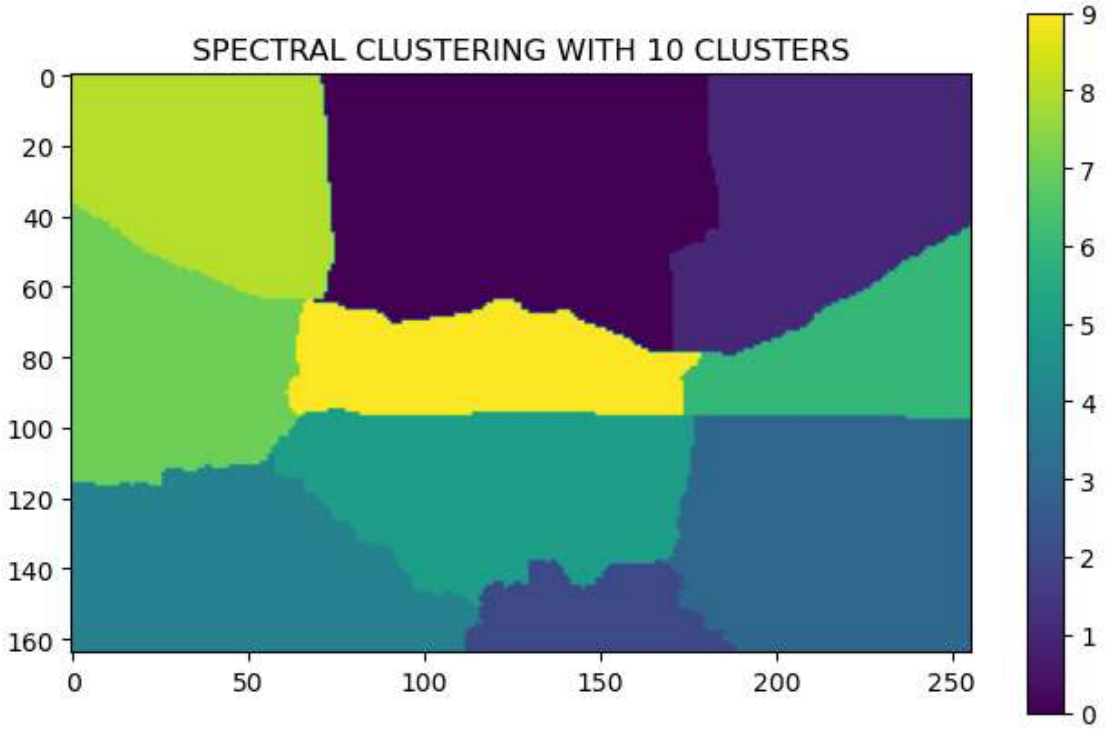
In [4]:
```
#Construct a sparse affinity matrix.
sparse_affinity_matrix = csr_matrix(affinity_matrix)

# Generate the Laplacian matrix using a sparse matrix representation.
degree_matrix = sparse_affinity_matrix.sum(axis=1).A1
degree_matrix = csr_matrix(np.diag(degree_matrix))
laplacian_matrix = degree_matrix - sparse_affinity_matrix
```

In [5]:
```
# Quantity of clusters.
num_clusters = 10

# Calculate eigenvectors, necessary for clustering, using sparse SVD.
a_, b_, eigenvectors = svds(laplacian_matrix, k=num_clusters)
```

In [6]:
```
# Employ the eigenvectors for clustering purposes.
kmeans = KMeans(n_clusters=num_clusters, n_init=10)
```

In [7]:
```
# Ensuring symmetry in the affinity matrix.
symmetric_matrix = 0.5 * (sparse_affinity_matrix + sparse_affinity_matrix.

# Performing spectral clustering.
spectral_model = SpectralClustering(n_clusters=num_clusters, affinity='pre
labels = spectral_model.fit_predict(symmetric_matrix)

# Reshaping the labels to visualize an image with dimensions 164x256.
labels_image = labels.reshape(164, 256)
```

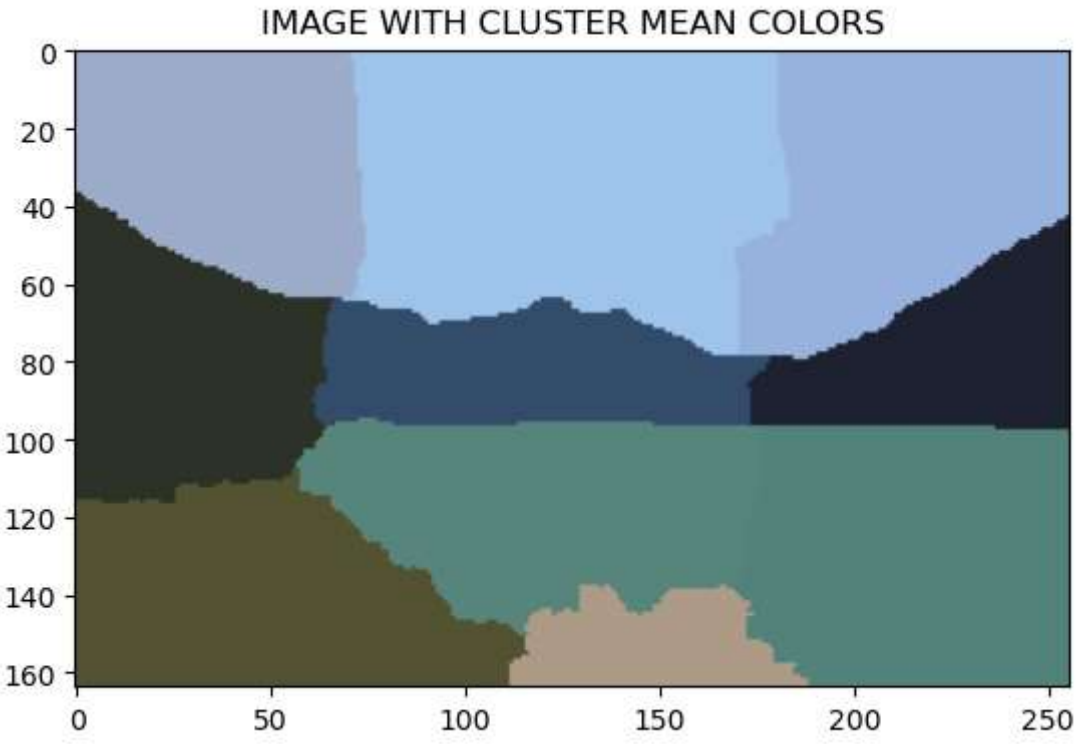
In [8]:
```

# Visualize the clustering outcome.
plt.figure(figsize=(8, 5))
plt.imshow(labels_image, cmap='viridis')
plt.title('SPECTRAL CLUSTERING WITH 10 CLUSTERS')
plt.colorbar()
plt.show()
```



Question 1b

In [9]:
```
# Transform the normalized image into a NumPy array.
original_image = np.array(norm_image)

# Array for storing the new image.
new_image = np.zeros_like(original_image)

# Compute the average color for each cluster.
for i in range(num_clusters):

    # Pixels associated with the current cluster.
    cluster_pixels = original_image[labels.reshape(164, 256) == i]

    # Compute the average color for those pixels.
    mean_color = np.mean(cluster_pixels, axis=0)

    # Assign the mean color to the corresponding pixels in the new image.
    new_image[labels.reshape(164, 256) == i] = mean_color
```

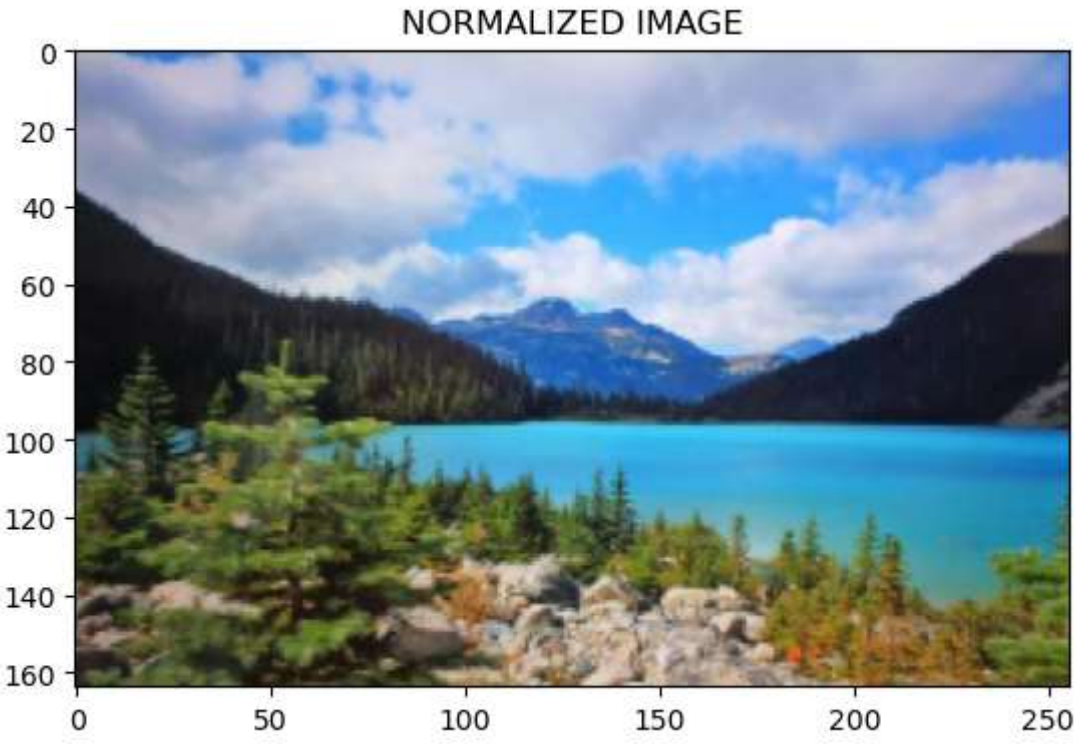
In [10]:
```
# Show the new image.
plt.imshow(new_image)
plt.title('IMAGE WITH CLUSTER MEAN COLORS')
plt.show()
```



Question 1c

In [12]:
```
# Import the necessary libraries.
import numpy as np
from PIL import Image
from scipy.sparse import lil_matrix, csr_matrix
from scipy.sparse.linalg import svds
from sklearn.cluster import SpectralClustering
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
```

In [13]:
```
# LOAD THE IMAGE
image = Image.open('scene256.jpg')

# Adjust the image dimensions to 164x256.
resized_image = image.resize((256,164))

# Convert the image to a NumPy array and perform normalization.
norm_image = np.array(resized_image) / 256.0

# Show the normalized image.
plt.imshow(norm_image)
plt.title('NORMALIZED IMAGE')
plt.show()
```



In [14]:
```
# Constructing the affinity matrix.
def construct_affinity_matrix(image, sigma=0.03):
    rows, cols, _ = image.shape
    num_pixels = rows * cols

    # Initialize the affinity matrix with zeros.
    affinity_matrix = lil_matrix((num_pixels, num_pixels))

    # Transform 2D coordinates into a 1D index.
    def to_index(r, c):
        return r * cols + c

    # Build the affinity matrix.
    for row in range(rows):
        for col in range(cols):
            index = to_index(row, col)
            if row + 1 < rows:
                bottom_index = to_index(row + 1, col)
                difference = image[row, col] - image[row + 1, col]
                affinity_matrix[index, bottom_index] = np.exp(-np.linalg.n
            if col + 1 < cols:
                right_index = to_index(row, col + 1)
                difference = image[row, col] - image[row, col + 1]
                affinity_matrix[index, right_index] = np.exp(-np.linalg.no

    return affinity_matrix

# Create the affinity matrix.
affinity_matrix = construct_affinity_matrix(norm_image)
affinity_matrix.shape
```

Out[14]: (41984, 41984)

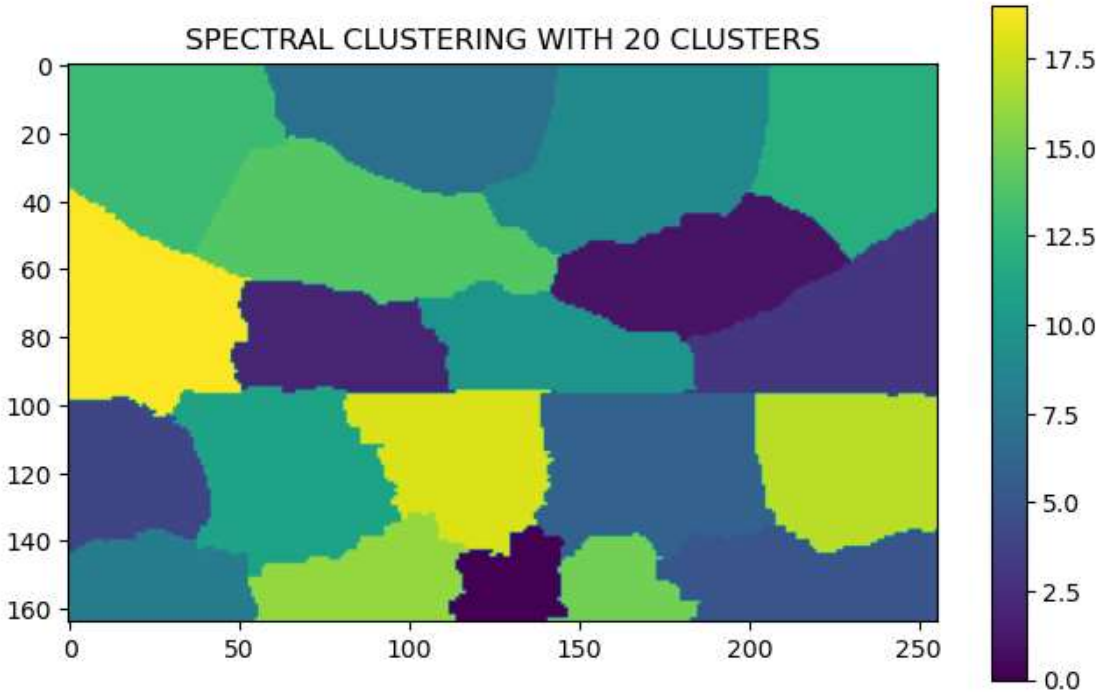
In [15]:
```
# Generate a sparse affinity matrix.
sparse_affinity_matrix = csr_matrix(affinity_matrix)

# Generate the Laplacian matrix using a sparse matrix representation.
degree_matrix = sparse_affinity_matrix.sum(axis=1).A1
degree_matrix = csr_matrix(np.diag(degree_matrix))
laplacian_matrix = degree_matrix - sparse_affinity_matrix
```

In [16]:
```
# Quantity of clusters.
num_clusters = 20

# Calculate eigenvectors, necessary for clustering, using sparse SVD.
a_, b_, eigenvectors = svds(laplacian_matrix, k=num_clusters)
```

In [17]:
```
# Utilize the eigenvectors for clustering purposes.
kmeans = KMeans(n_clusters=num_clusters, n_init=10)
```

In [18]:
```
# Ensuring symmetry in the affinity matrix.
symmetric_matrix = 0.5 * (sparse_affinity_matrix + sparse_affinity_matrix.

# Performing spectral clustering.
spectral_model = SpectralClustering(n_clusters=num_clusters, affinity='pre
labels = spectral_model.fit_predict(symmetric_matrix)

# Reshape the labels for displaying an image with dimensions 164x256.
labels_image = labels.reshape(164, 256)
```

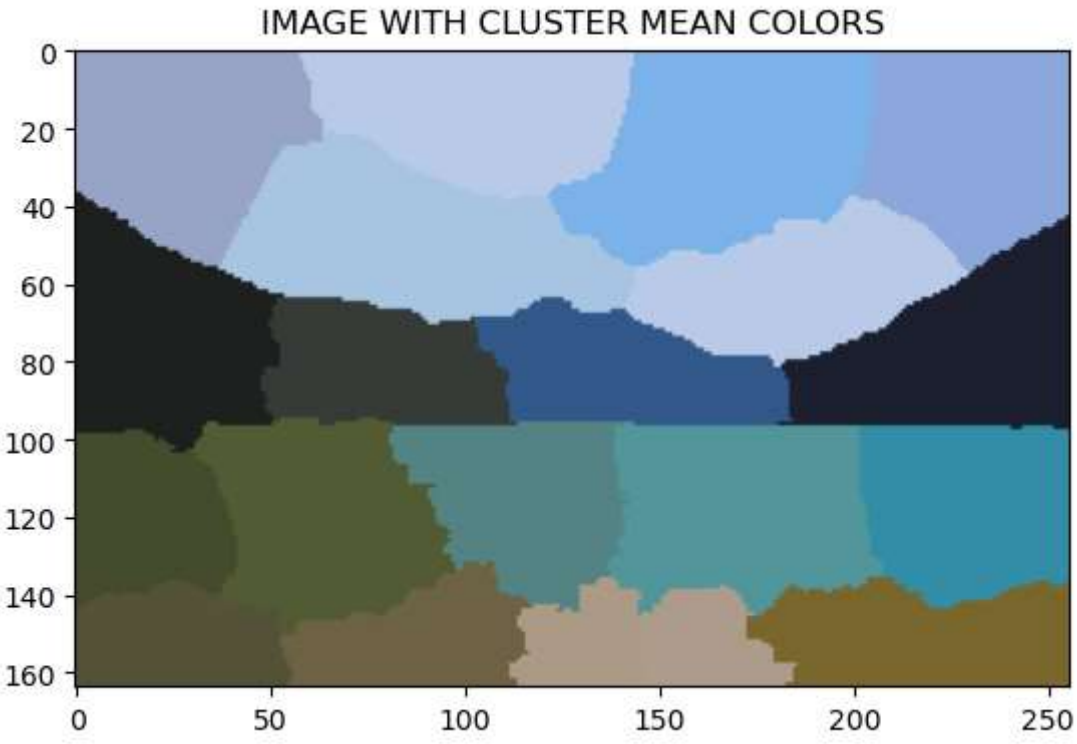
In [19]:
```
# Visualize the clustering outcome.
plt.figure(figsize=(8, 5))
plt.imshow(labels_image, cmap='viridis')
plt.title('SPECTRAL CLUSTERING WITH 20 CLUSTERS')
plt.colorbar()
plt.show()
```



Question 1d

In [20]:
```
# Transform the normalized image into a NumPy array.
original_image = np.array(norm_image)

# Array for storing the new image.
new_image = np.zeros_like(original_image)

# Compute the average color for each cluster.
for i in range(num_clusters):

    # Pixels associated with the current cluster.
    cluster_pixels = original_image[labels.reshape(164, 256) == i]

    # Compute the mean color for those pixels.
    mean_color = np.mean(cluster_pixels, axis=0)

    # Assign the mean color to the corresponding pixels in the new image.
    new_image[labels.reshape(164, 256) == i] = mean_color
```

In [21]:
```
# Show the new image.
plt.imshow(new_image)
plt.title('IMAGE WITH CLUSTER MEAN COLORS')
plt.show()
```



AML Assignment 10 - Kirti Katiyar

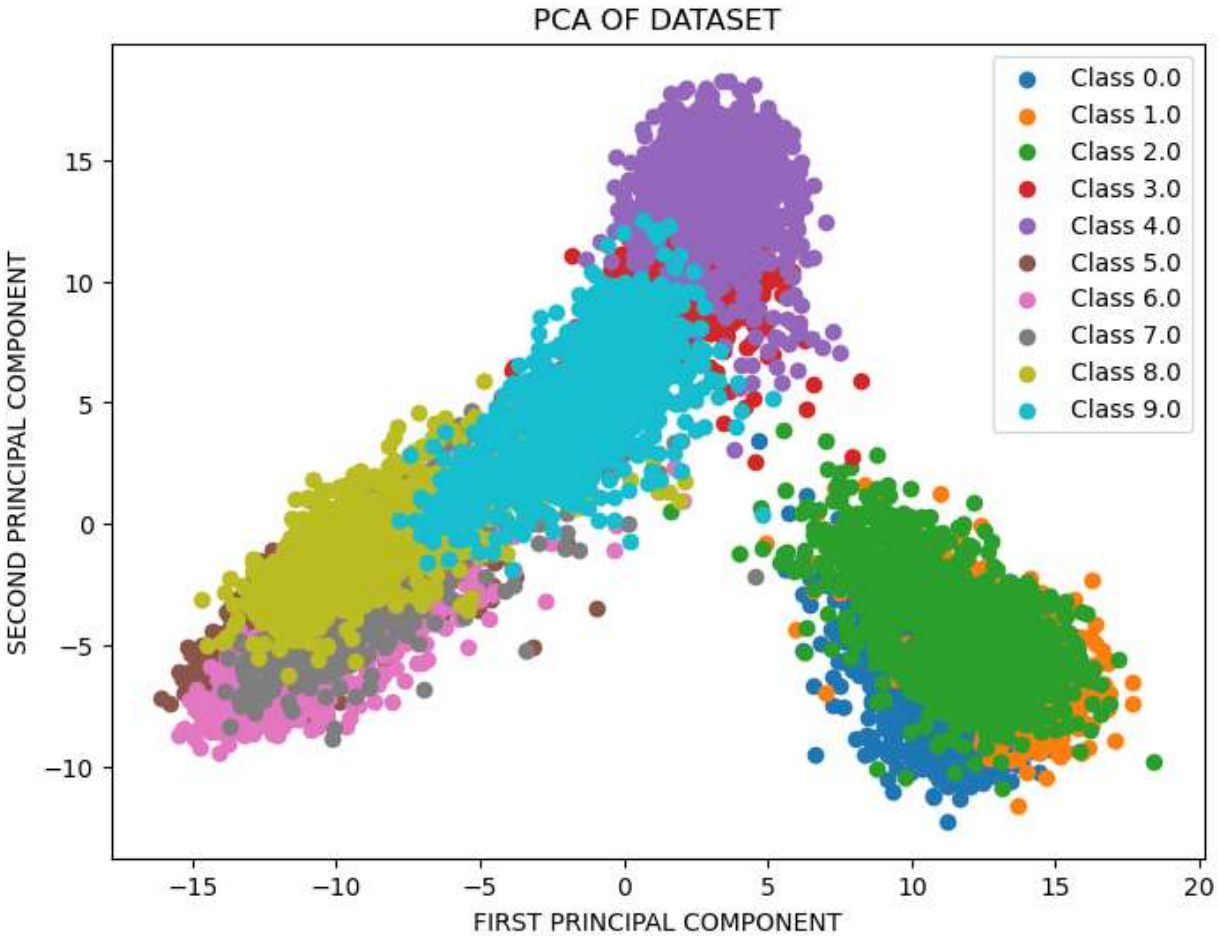
Question 1a

In [50]:
```
# IMPORT REQUIRED LIBRARIES
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from scipy.io import loadmat
from scipy.spatial.distance import cdist
from sklearn.cluster import KMeans
from sklearn.metrics.cluster import contingency_matrix
from scipy.optimize import linear_sum_assignment
from sklearn.metrics import adjusted_rand_score
from scipy.io import loadmat
```

In [51]:
```
# LOAD DATA
data = loadmat('data_clust.mat')
X = data['x']
y = data['y']

# ENSURING y IS 1D ARRAY
y = y.squeeze()
```

In [52]:
```
# PERFORMING PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

In [53]:
```
# PLOT
plt.figure(figsize=(8, 6))
unique_classes = np.unique(y)
for cls in unique_classes:
    plt.scatter(X_pca[y == cls, 0], X_pca[y == cls, 1], label=f'Class {cls}')
plt.xlabel('FIRST PRINCIPAL COMPONENT')
plt.ylabel('SECOND PRINCIPAL COMPONENT')
plt.title('PCA OF DATASET')
plt.legend()
plt.show()
```



Question 1b

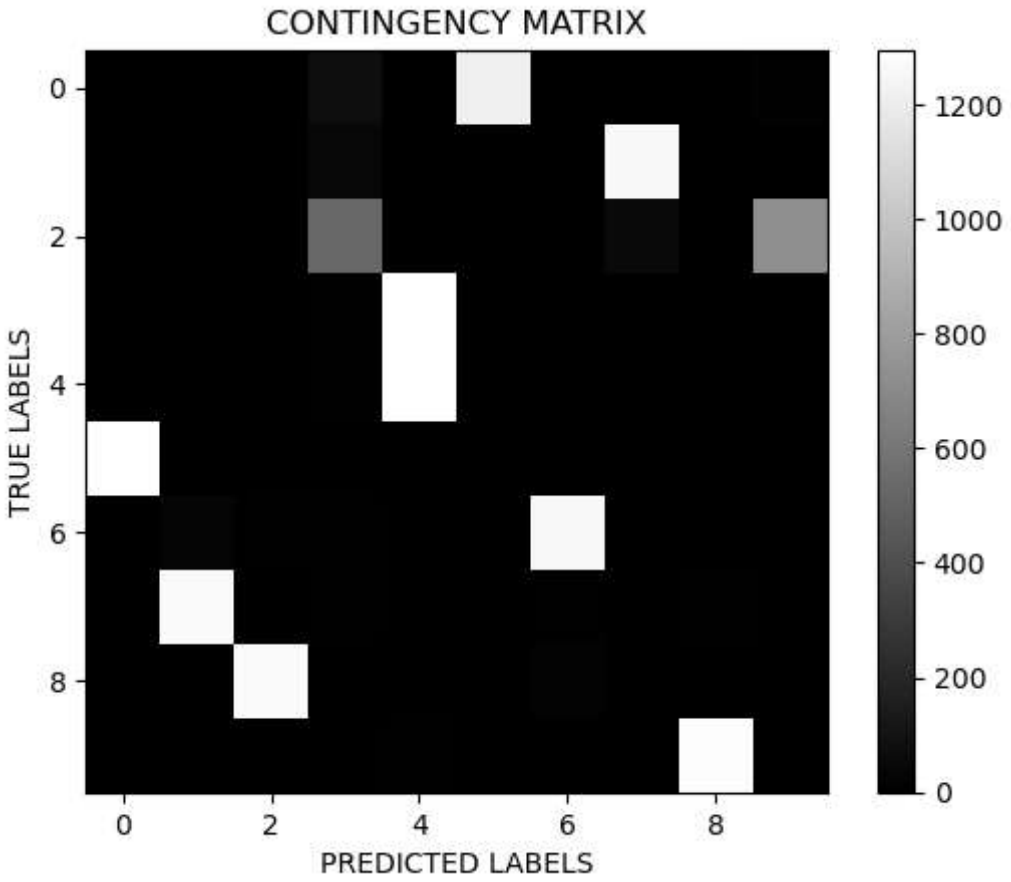
In [54]:
```
# IMPORT REQUIRED LIBRARIES
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics.cluster import contingency_matrix
from scipy.io import loadmat
```

In [55]:
```
# LOAD DATA
data = loadmat('data_clust.mat')
X = data['x']
y = data['y']

# ENSURING y IS 1D ARRAY
y = y.squeeze()
```

In [56]:
```
# PERFORM K-MEANS CLUSTERING
kmeans = KMeans(n_clusters=10, init='random', n_init=1)
y_pred = kmeans.fit_predict(X)
```

In [57]:
```
# CONTINGENCY MATRIX
cont_matrix = contingency_matrix(y, y_pred)
```

In [58]:
```
# CONTINGENCY MATRIX AS A GRAYSCALE IMAGE
plt.imshow(cont_matrix, cmap='gray', interpolation='nearest')
plt.colorbar()
plt.title('CONTINGENCY MATRIX')
plt.xlabel('PREDICTED LABELS')
plt.ylabel('TRUE LABELS')
plt.show()
```



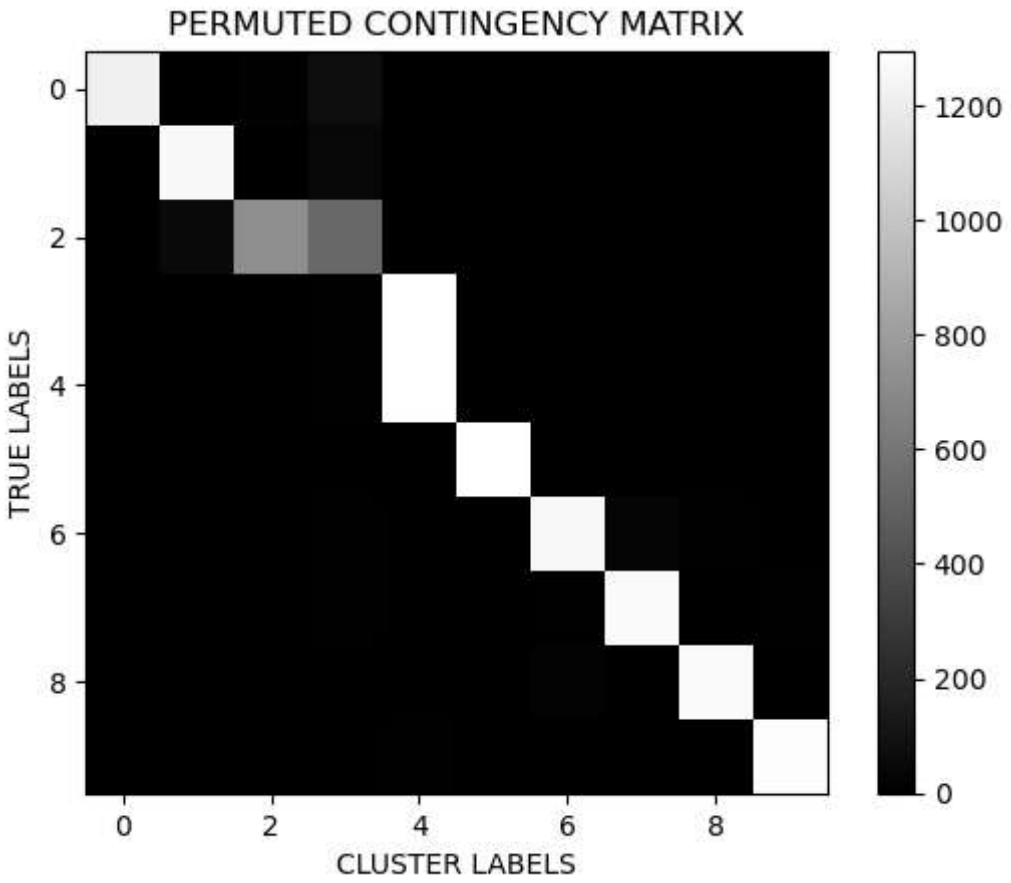
Question 1c

In [59]:
```
# IMPORT EXTRA REQUIRED LBRARIES
from scipy.optimize import linear_sum_assignment
```

In [60]:
```
# SOLVING LINEAR SUM ASSIGNMENT PROBLEM
row_ind, col_ind = linear_sum_assignment(-cont_matrix)
```

In [61]:
```
# APPLYING PERMUTATION TO THE COLUMNS OF THE CONTINGENCY MATRIX
permuted_matrix = cont_matrix[:, col_ind]
```

In [62]:
```
# DISPLAY PERMUTED CONTINGENCY MATRIX
plt.imshow(permuted_matrix, cmap='gray', interpolation='nearest')
plt.colorbar()
plt.title('PERMUTED CONTINGENCY MATRIX')
plt.xlabel('CLUSTER LABELS')
plt.ylabel('TRUE LABELS')
plt.show()
```

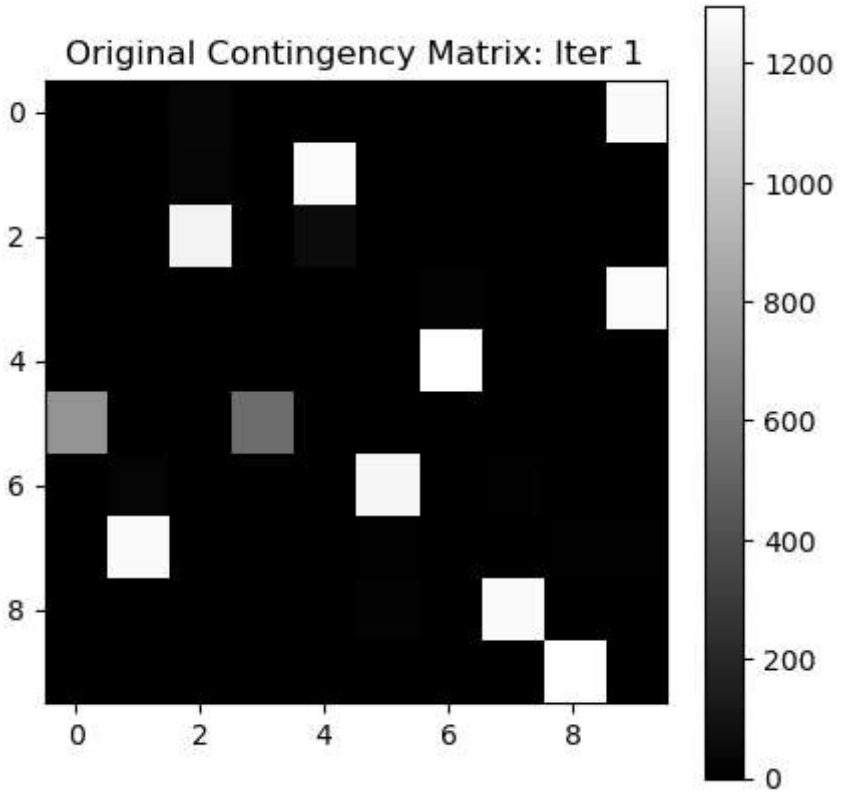


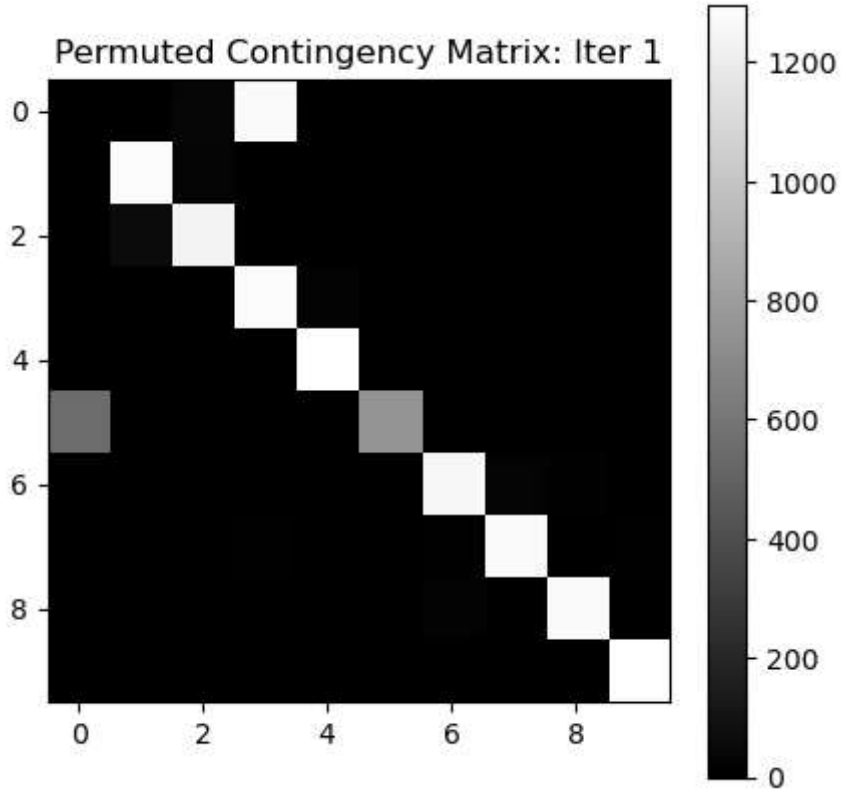
Question 1d

In [63]:
```
# LISTS TO STORE RESULTS
random_accuracies = []
random_adjusted_rand_scores = []
```

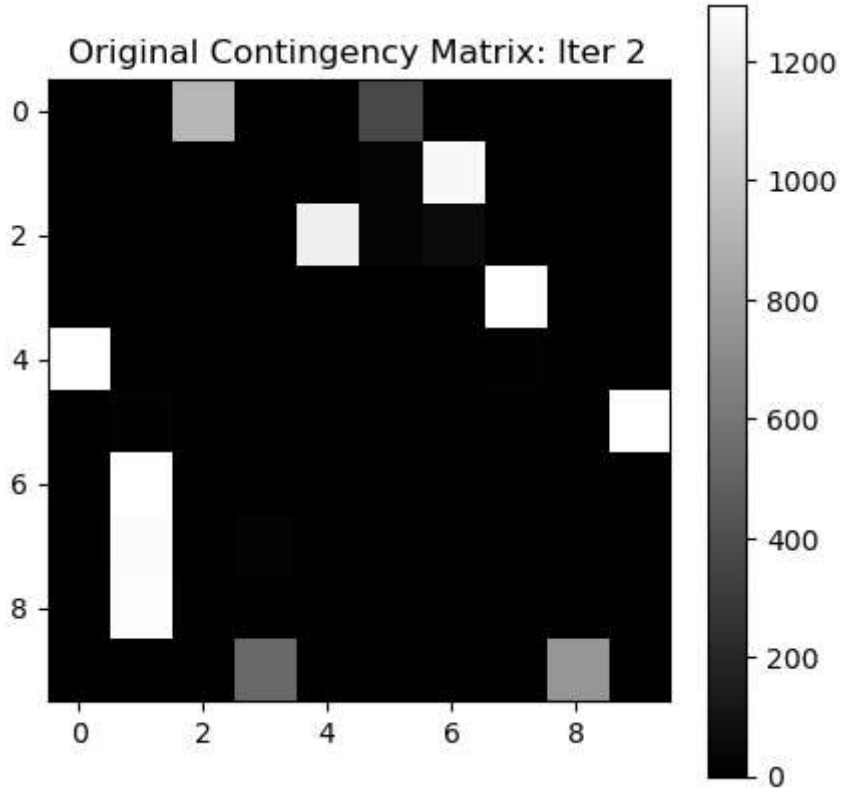
In [64]:
```
# REPEATING THE WHOLE PROCESS 5 TIMES

for i in range(5):
    # PERFORMING K-MEANS CLUSTERING
    kmeans = KMeans(n_clusters=10, init='random', n_init=1, random_state=i)
    y_pred = kmeans.fit_predict(X)

    # COMPUTE CONTINGENCY MATRIX
    cont_matrix = contingency_matrix(y, y_pred)

    # DISPLAY THE ORIGINAL CONTINGENCY MATRIX
    plt.figure(figsize=(5, 5))
    plt.imshow(cont_matrix, cmap='gray', interpolation='nearest')
    plt.title(f'Original Contingency Matrix: Iter {i+1}')
    plt.colorbar()
    plt.show()

    # LINEAR SUM ASSIGNMENT PROBLEM
    row_ind, col_ind = linear_sum_assignment(-cont_matrix)

    # APPLYING PERMUTATION
    permuted_matrix = cont_matrix[:, col_ind]
```

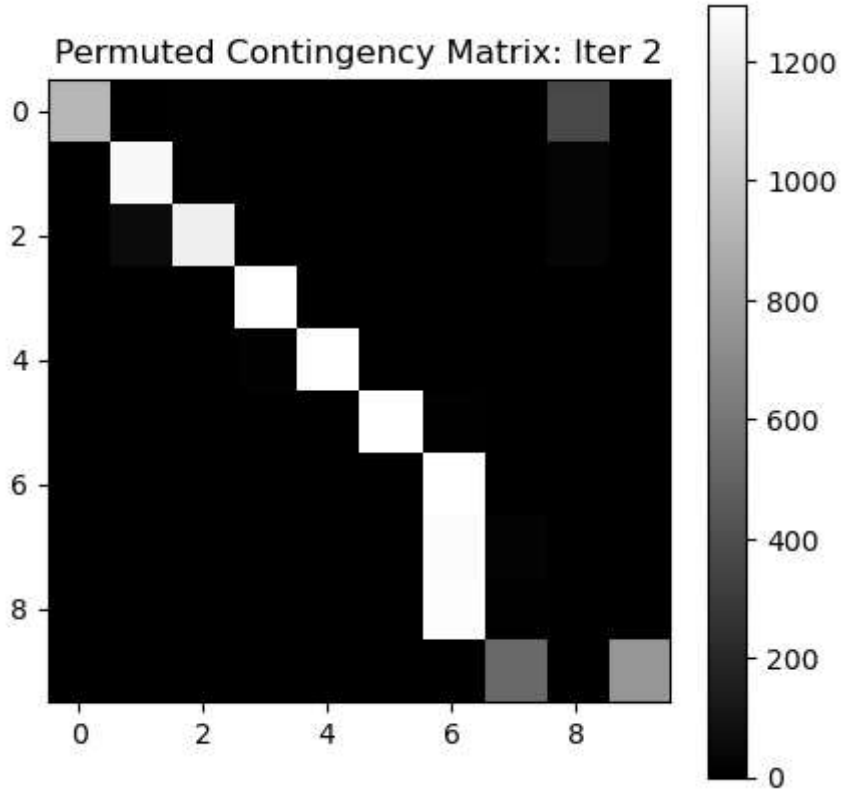
```
    # DISPLAY CONTINGENCY MATRIX
    plt.figure(figsize=(5, 5))
    plt.imshow(permuted_matrix, cmap='gray', interpolation='nearest')
    plt.title(f'Permuted Contingency Matrix: Iter {i+1}')
    plt.colorbar()
    plt.show()

    # CALCULATE ACCURACY
    random_accuracy = np.sum(np.diag(permuted_matrix)) / np.sum(permuted_matrix)
    random_accuracies.append(random_accuracy)

    # CALCULATE ADJUSTED RAND INDEX
    random_ari = adjusted_rand_score(y, y_pred)
    random_adjusted_rand_scores.append(random_ari)

    print(f"ITERATION {i+1}: ACCURACY = {random_accuracy:.4f}, ADJUSTED RAND INDEX = {
```



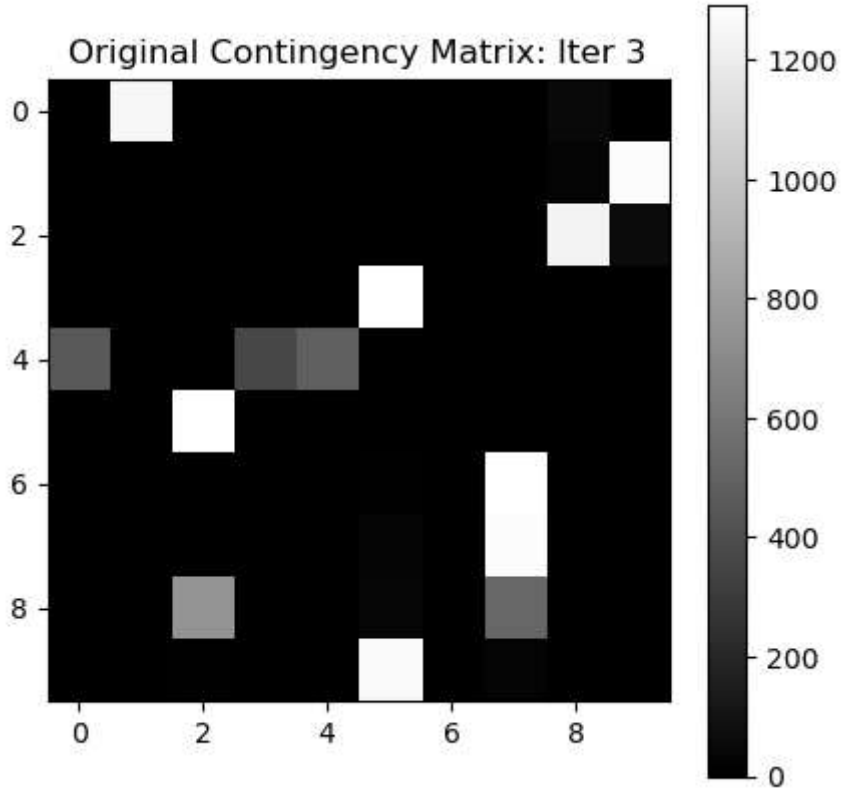


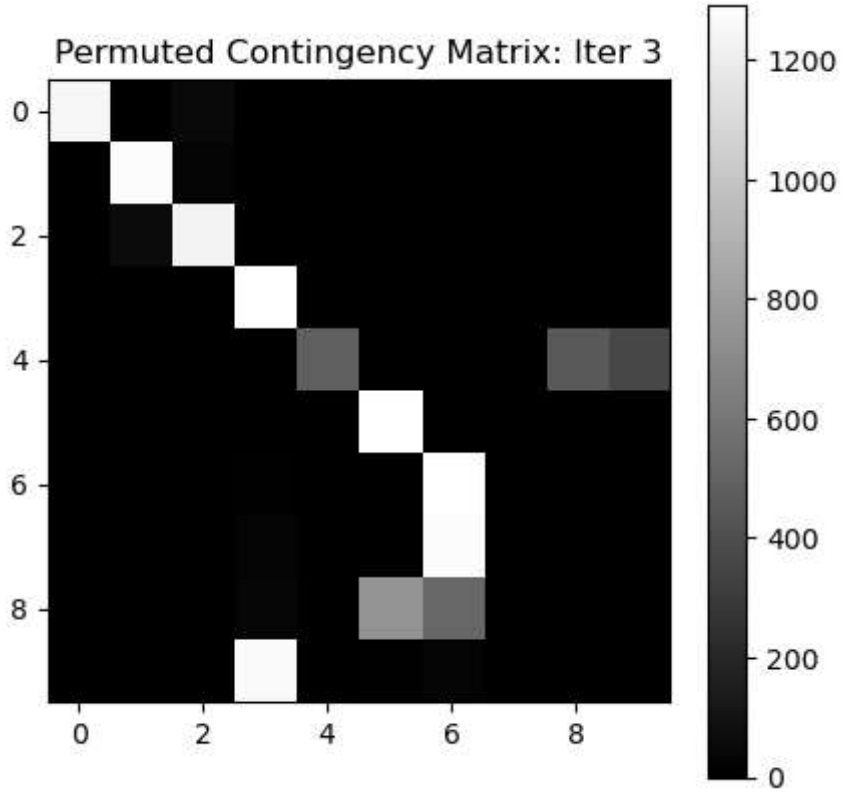

```
ITERATION 1: ACCURACY = 0.8392, ADJUSTED RAND INDEX = 0.8308
```



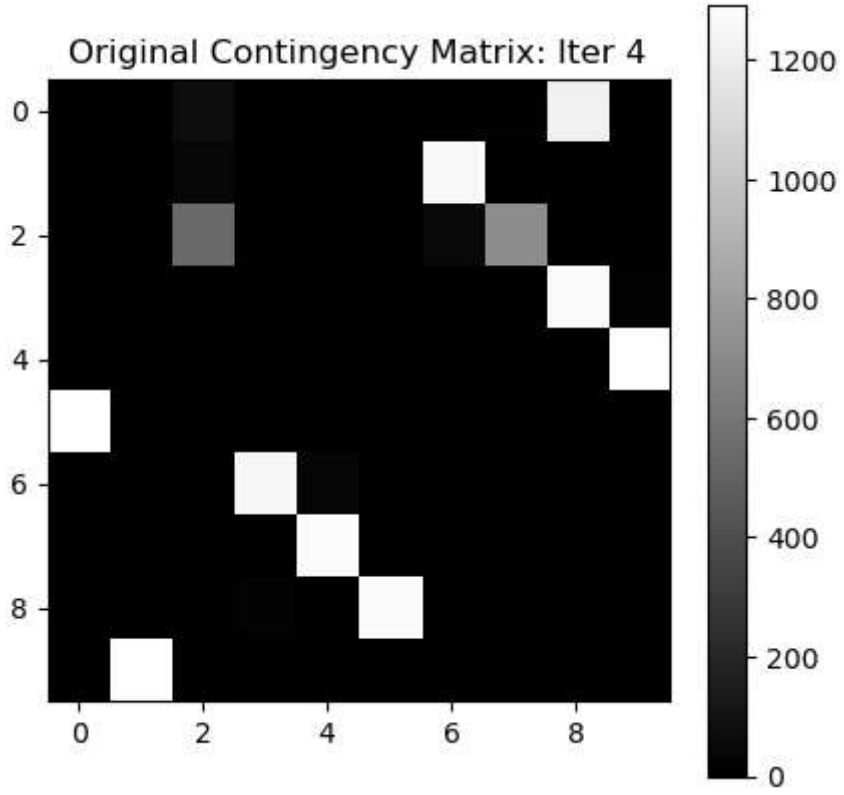


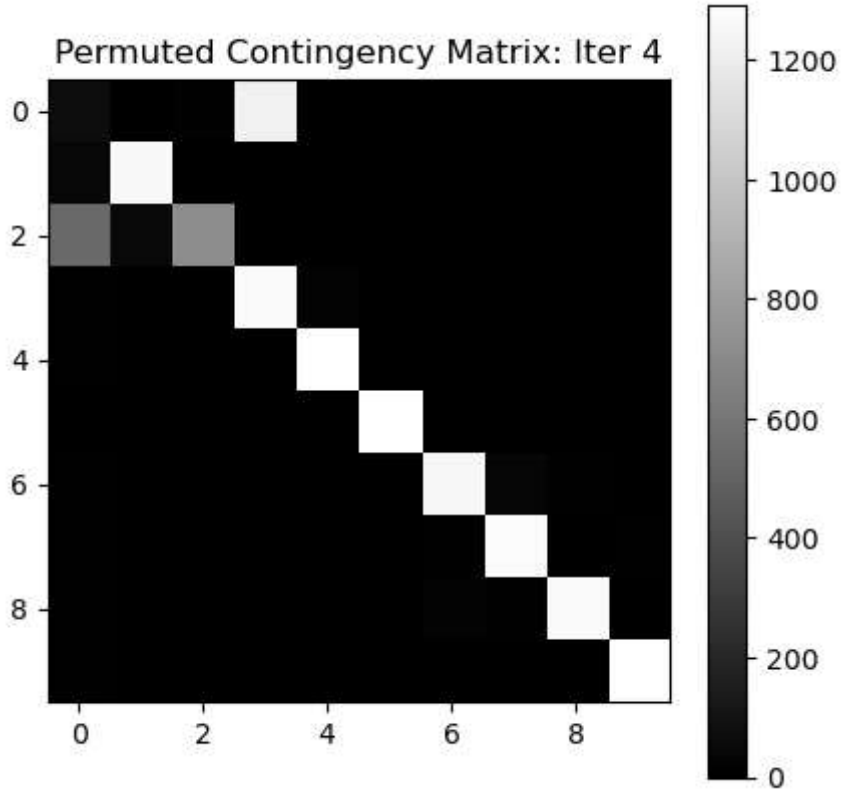

```
ITERATION 2: ACCURACY = 0.7189, ADJUSTED RAND INDEX = 0.6647
```



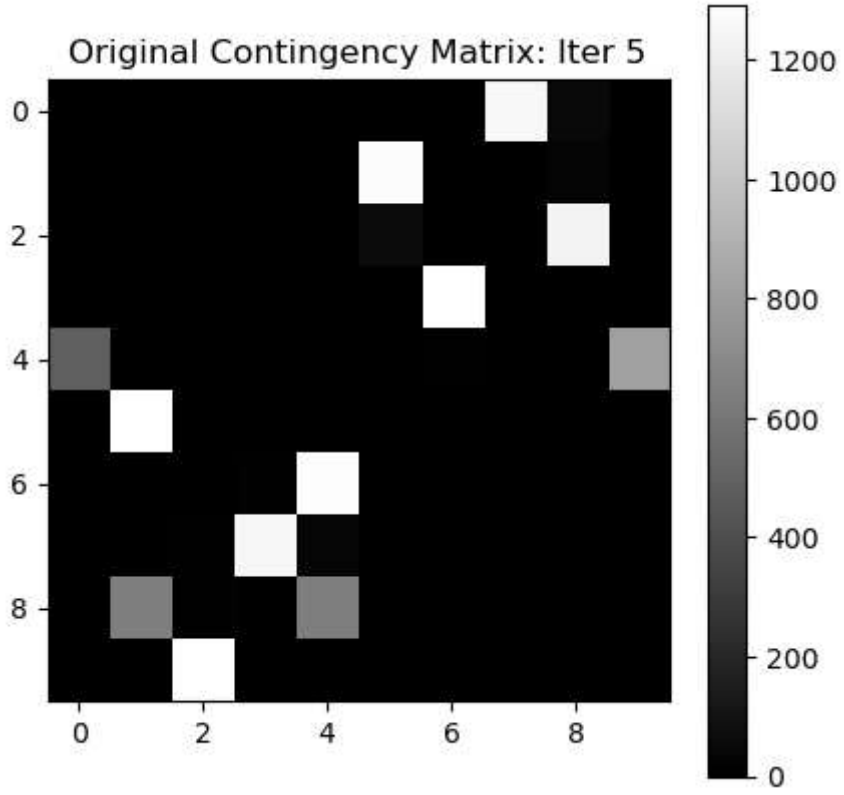


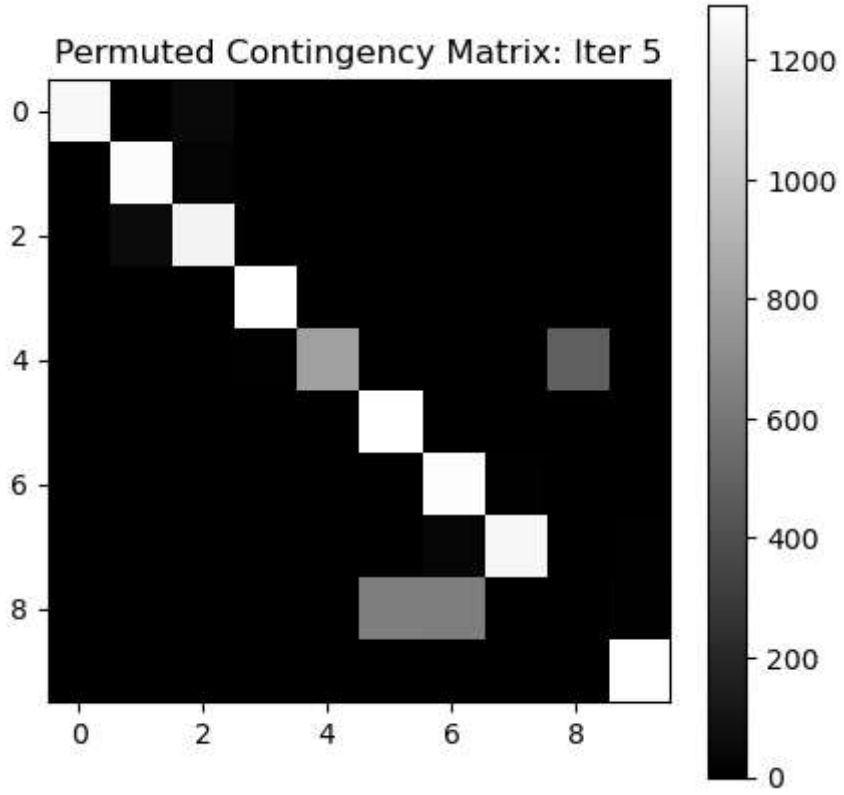

```
ITERATION 3: ACCURACY = 0.6245, ADJUSTED RAND INDEX = 0.6141
```






```
ITERATION 4: ACCURACY = 0.8432, ADJUSTED RAND INDEX = 0.8311
```






```
ITERATION 5: ACCURACY = 0.8437, ADJUSTED RAND INDEX = 0.8013
```

In [65]:
```
# DISPLAY OVERALL RESULTS
print("AVERAGE ACCURACY:", np.mean(random_accuracies))
print("AVERAGE ADJUSTED RAND INDEX:", np.mean(random_adjusted_rand_scores))
```

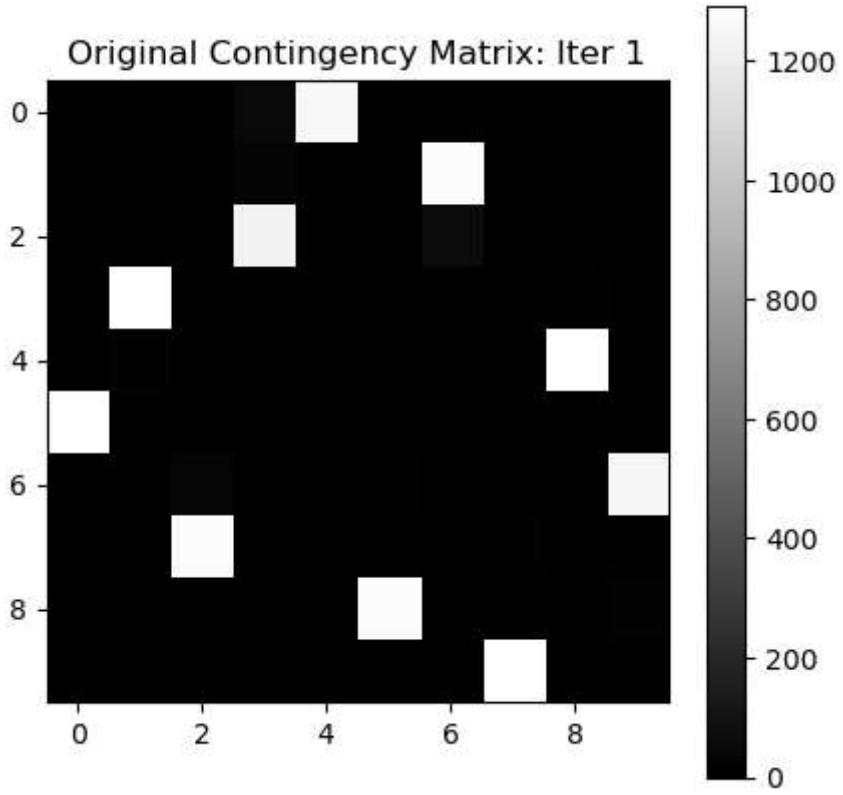
```
AVERAGE ACCURACY: 0.7739230769230769
AVERAGE ADJUSTED RAND INDEX: 0.7483869833733245
```

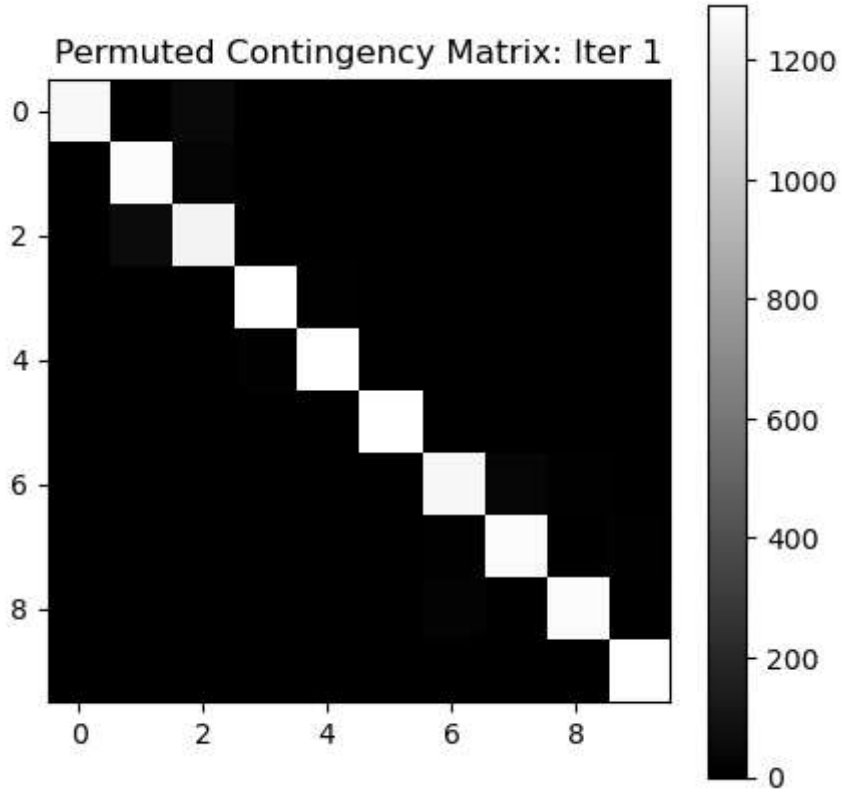
Question 1e

In [66]:
```
# LISTS TO STORE RESULTS
kmeans_accuracies = []
kmeans_adjusted_rand_scores = []
```

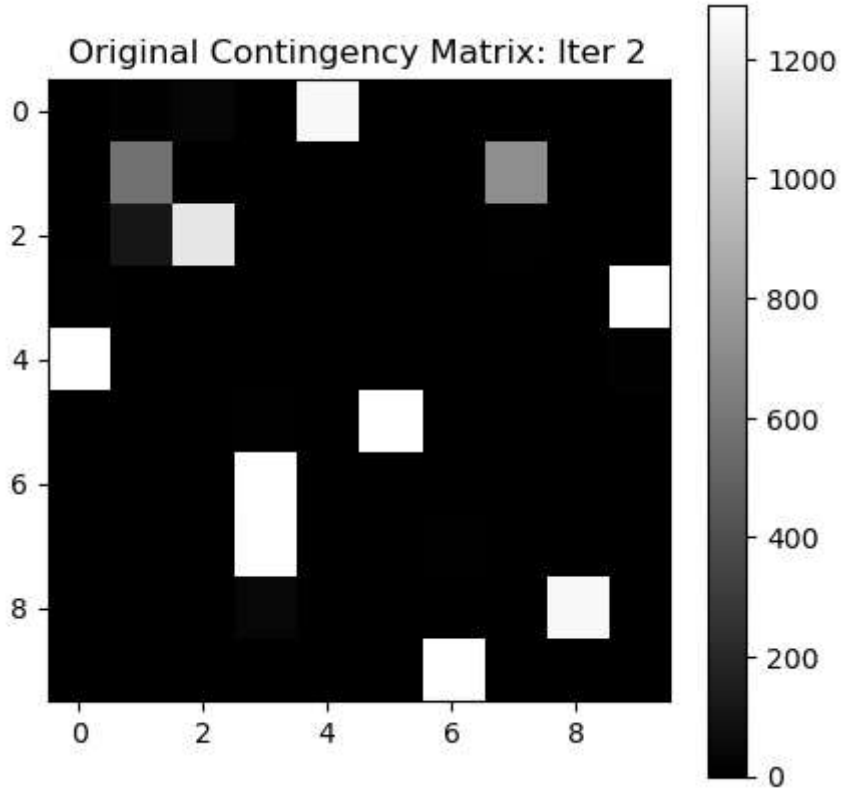
In [67]:
```
# REPEATING THE WHOLE PROCESS 5 TIMES

for i in range(5):
    # PERFORMING K-MEANS CLUSTERING
    kmeans = KMeans(n_clusters=10, init='k-means++', n_init=1, random_state=i)
    y_pred = kmeans.fit_predict(X)

    # COMPUTE CONTINGENCY MATRIX
    cont_matrix = contingency_matrix(y, y_pred)

    # DISPLAY THE ORIGINAL CONTINGENCY MATRIX
    plt.figure(figsize=(5, 5))
    plt.imshow(cont_matrix, cmap='gray', interpolation='nearest')
    plt.title(f'Original Contingency Matrix: Iter {i+1}')
    plt.colorbar()
    plt.show()
```

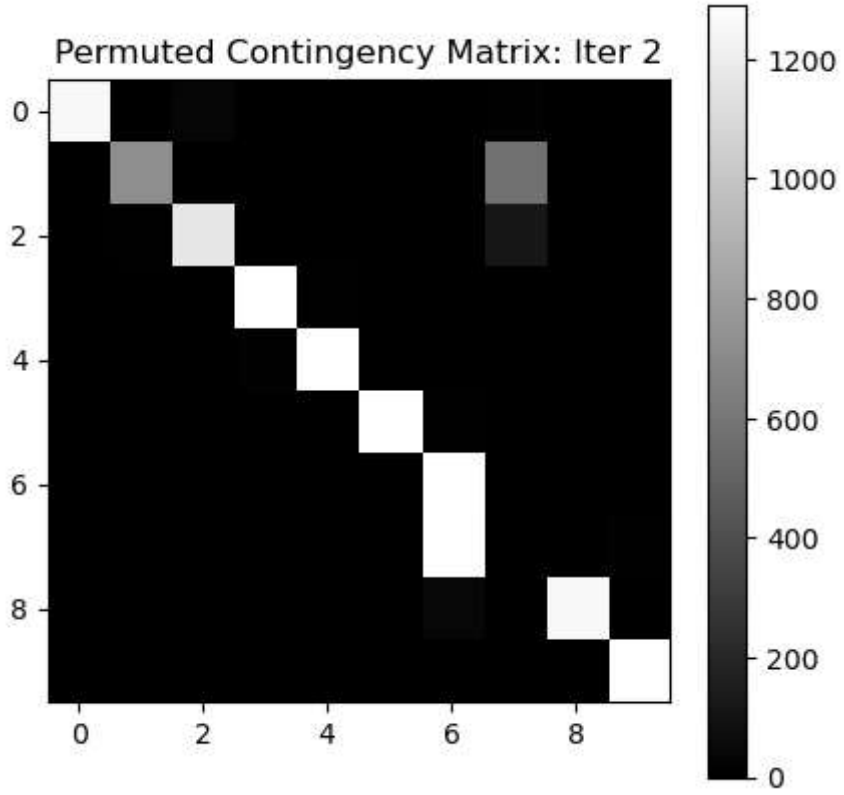
```
    # LINEAR SUM ASSIGNMENT PROBLEM
    row_ind, col_ind = linear_sum_assignment(-cont_matrix)

    # APPLYING PERMUTATION
    permuted_matrix = cont_matrix[:, col_ind]

    # DISPLAY CONTINGENCY MATRIX
    plt.figure(figsize=(5, 5))
    plt.imshow(permuted_matrix, cmap='gray', interpolation='nearest')
    plt.title(f'Permuted Contingency Matrix: Iter {i+1}')
    plt.colorbar()
    plt.show()

    # CALCULATE ACCURACY
    kmeans_accuracy = np.sum(np.diag(permuted_matrix)) / np.sum(permuted_matrix)
    kmeans_accuracies.append(kmeans_accuracy)

    # CALCULATE ADJUSTED RAND INDEX
    kmeans_ari = adjusted_rand_score(y, y_pred)
    kmeans_adjusted_rand_scores.append(kmeans_ari)

    print(f"ITERATION {i+1}: ACCURACY = {kmeans_accuracy:.4f}, ADJUSTED RAND INDEX = {
```



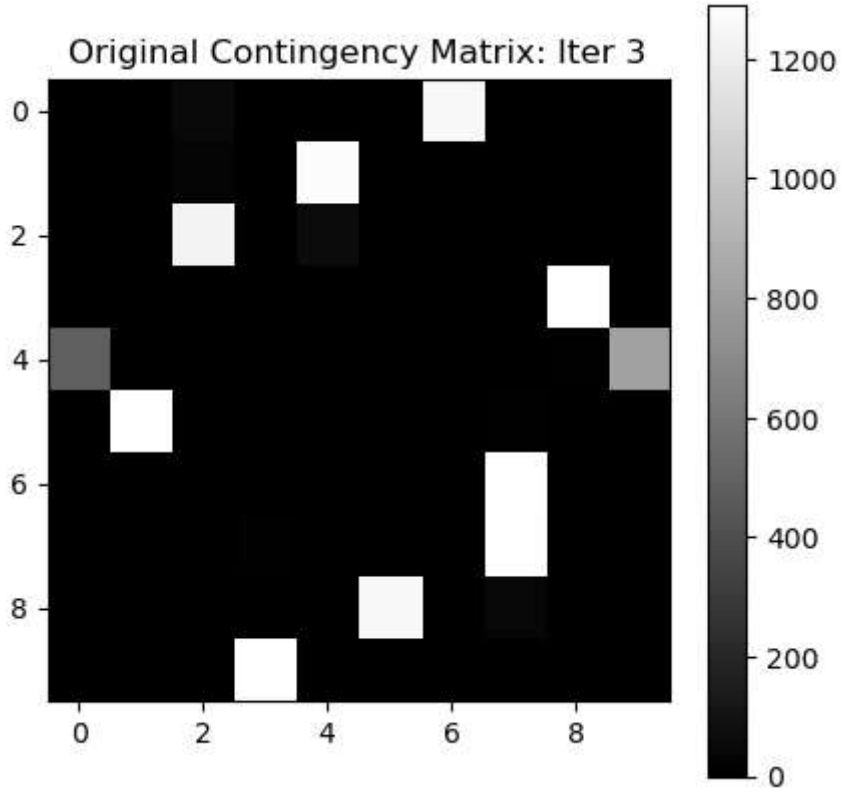


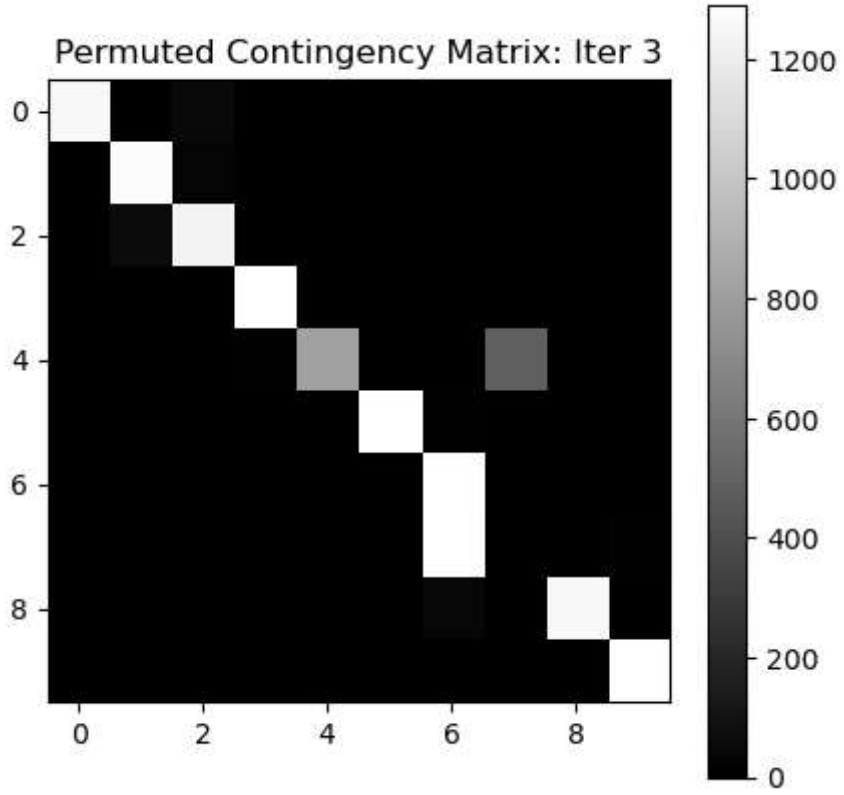

```
ITERATION 1: ACCURACY = 0.9780, ADJUSTED RAND INDEX = 0.9523
```



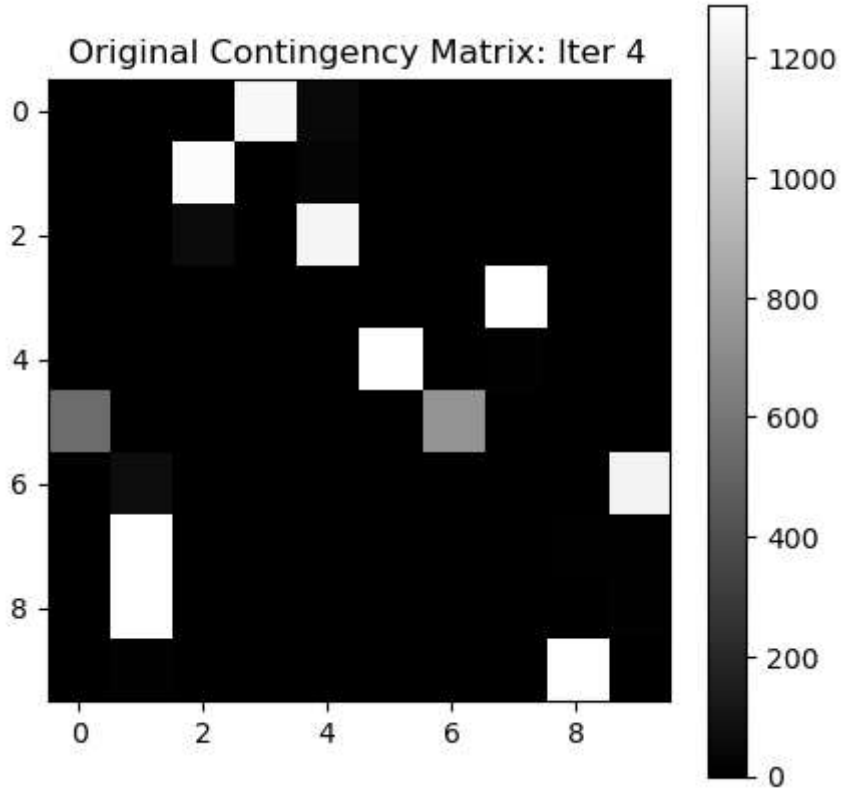


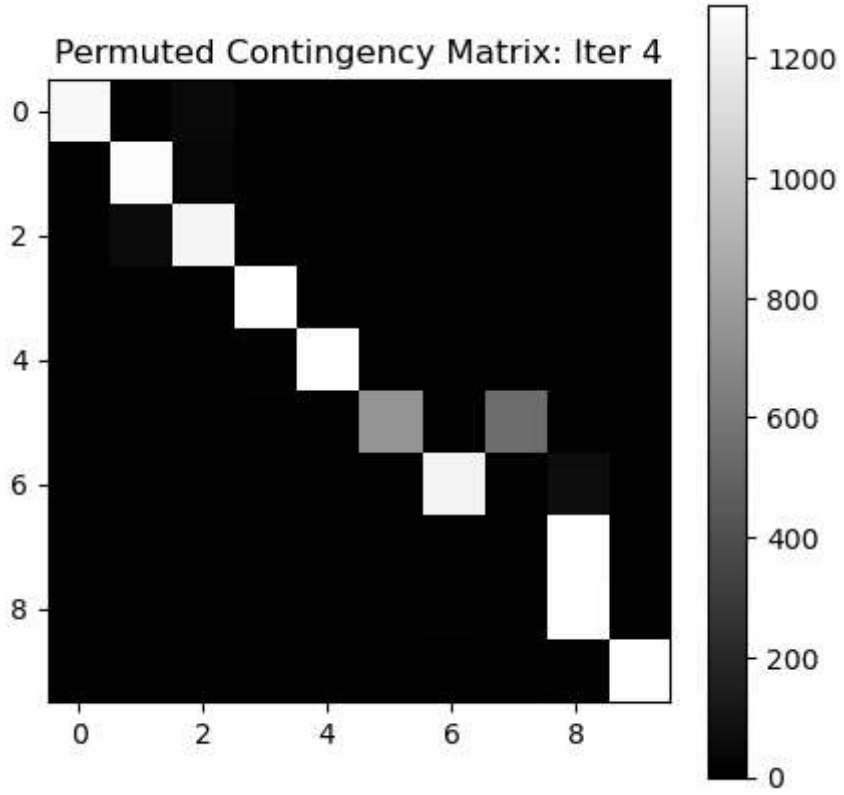

```
ITERATION 2: ACCURACY = 0.8342, ADJUSTED RAND INDEX = 0.8275
```



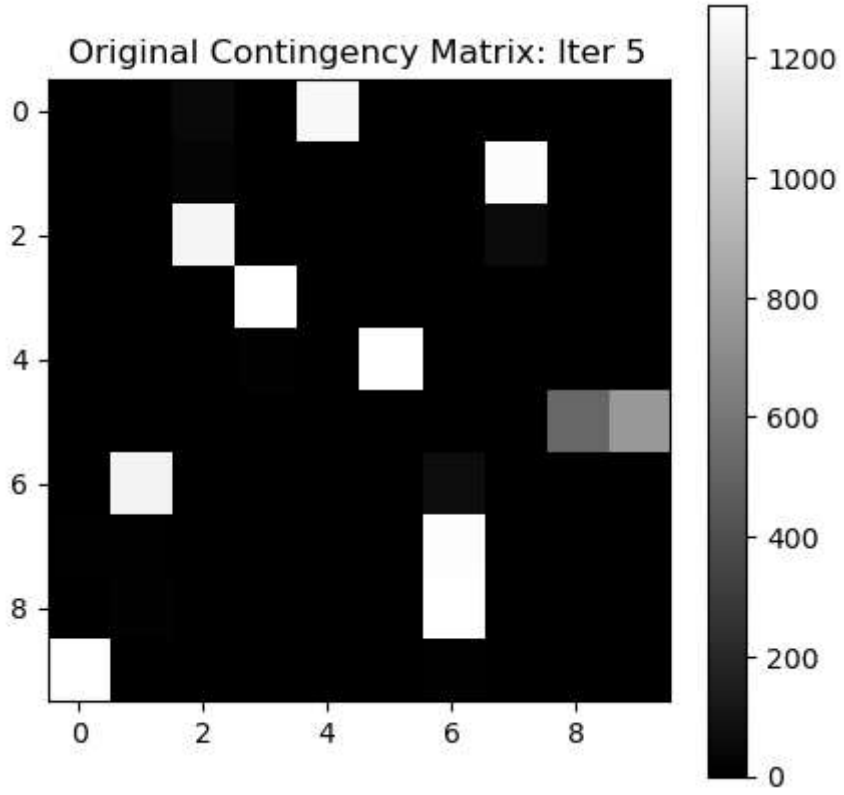


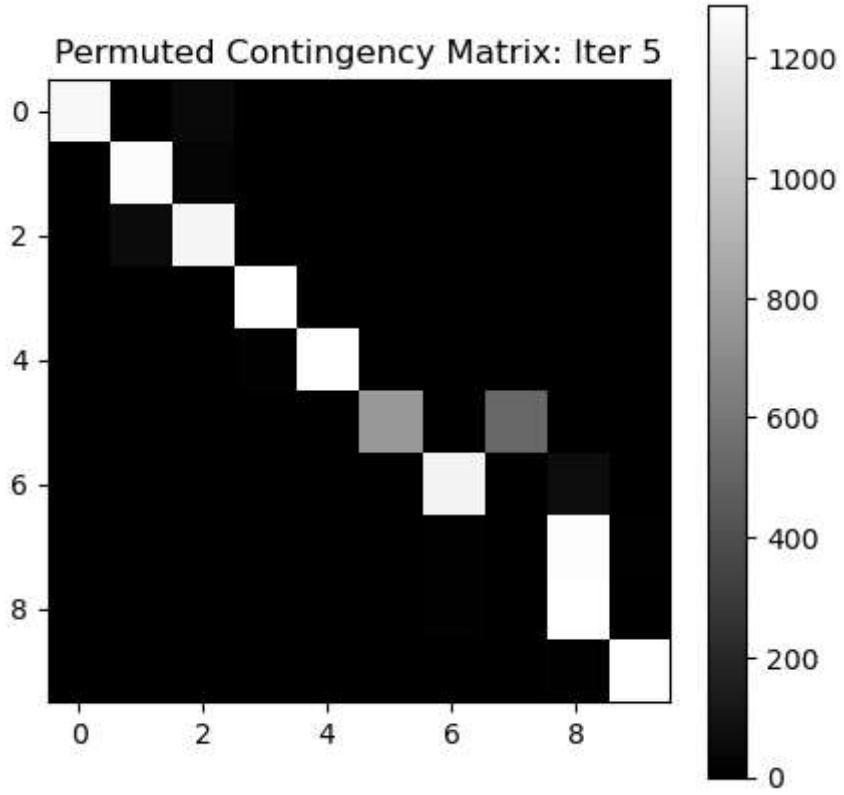

```
ITERATION 3: ACCURACY = 0.8448, ADJUSTED RAND INDEX = 0.8307
```






```
ITERATION 4: ACCURACY = 0.8365, ADJUSTED RAND INDEX = 0.8208
```






```
ITERATION 5: ACCURACY = 0.8388, ADJUSTED RAND INDEX = 0.8212
```

In [68]:
```
# DISPLAY OVERALL RESULTS
print("AVERAGE ACCURACY:", np.mean(kmeans_accuracies))
print("AVERAGE ADJUSTED RAND INDEX:", np.mean(kmeans_adjusted_rand_scores))
```

```
AVERAGE ACCURACY: 0.8664769230769231
AVERAGE ADJUSTED RAND INDEX: 0.8505128738880735
```

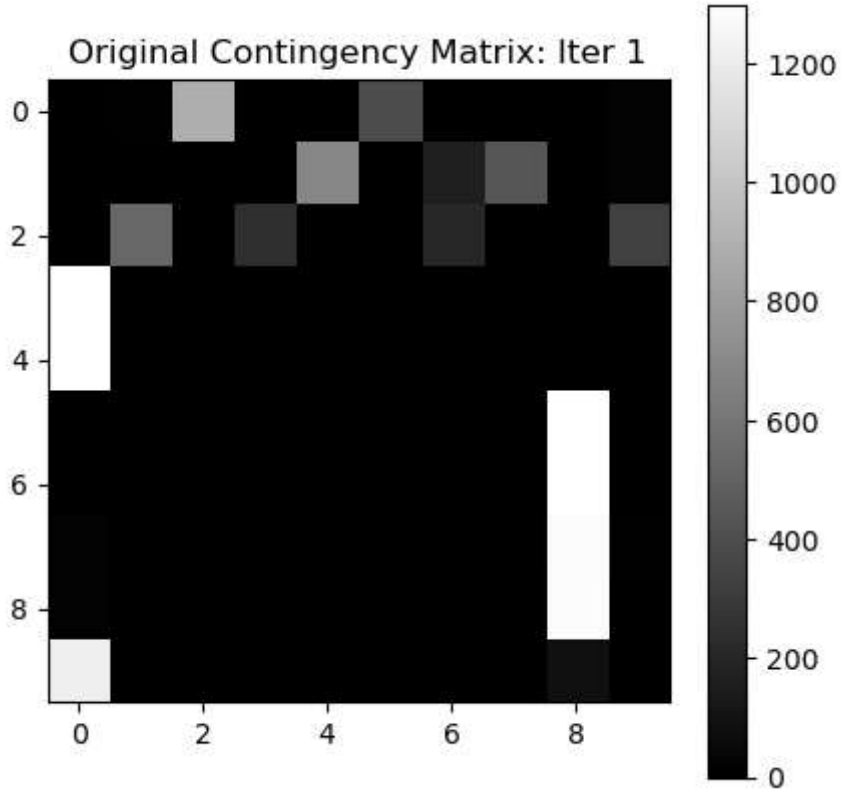
Question 1f

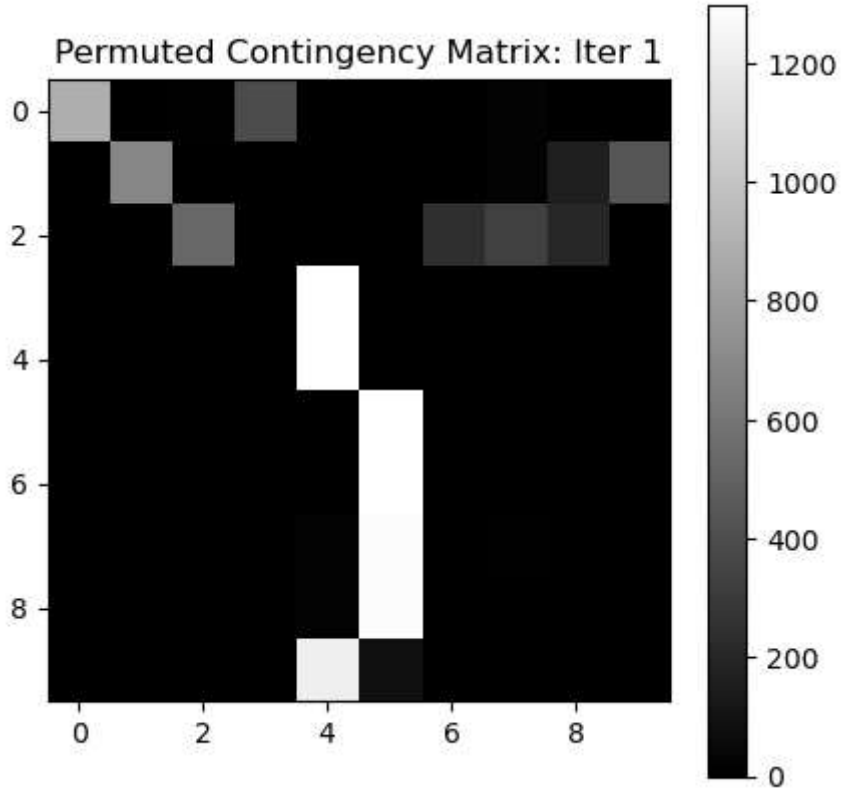
In [69]:
```
# DEFINE FARTHEST POINT INITIALIZATION FUNCTION
def farthest_points_initialization(X, k):
    # FIRST CENTER IS CHOSEN RANDOMLY
    n_samples, n_features = X.shape
    centers = np.empty((k, n_features))
    center_idx = np.random.choice(n_samples)
    centers[0] = X[center_idx]

    # REMAINING CENTERS
    for i in range(1, k):
        # COMPUTE DISTANCES FROM EACH POINT TO THE NEAREST CENTER
        distances = cdist(X, centers[:i], 'euclidean')
        min_distances = np.min(distances, axis=1)

        # NEXT CENTER - POINT THAT MAXIMIZES THE MIN DISTANCE TO THE EXISTING CENTERS
        centers[i] = X[np.argmax(min_distances)]

    return centers

def farthest_points_initialization(X, k):
    n_samples, n_features = X.shape
```

```
    centers = np.empty((k, n_features))
    center_idx = np.random.choice(n_samples)
    centers[0] = X[center_idx]

    for i in range(1, k):
        distances = cdist(X, centers[:i], 'euclidean')
        min_distances = np.min(distances, axis=1)
        centers[i] = X[np.argmax(min_distances)]

    return centers
```

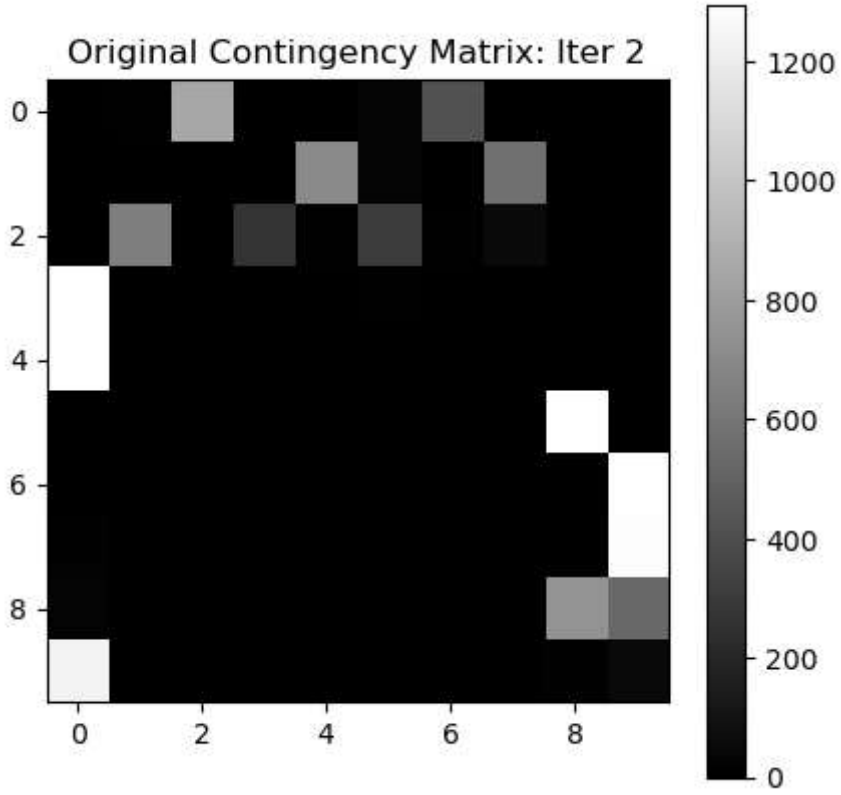
In [70]:
```
# LISTS TO STORE RESULTS
farthest_init_accuracies = []
farthest_init_adjusted_rand_scores = []
```

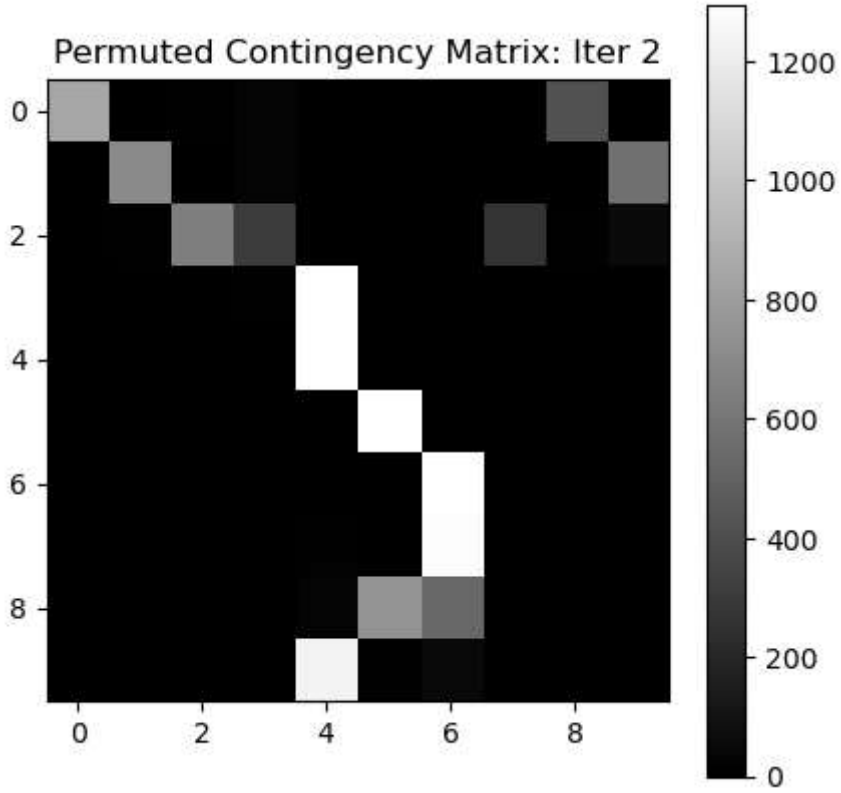
In [71]:
```
# REPEATING THE WHOLE PROCESS 5 TIMES

for i in range(5):
    # INITIALIZE CENTERS USING FARTHEST POINTS METHOD
    initial_centers = farthest_points_initialization(X, k=10)
    kmeans = KMeans(n_clusters=10, init=initial_centers, n_init=1)
    y_pred = kmeans.fit_predict(X)

    # COMPUTE CONTINGENCY MATRIX
    cont_matrix = contingency_matrix(y, y_pred)

    # DISPLAY THE ORIGINAL CONTINGENCY MATRIX
    plt.figure(figsize=(5, 5))
    plt.imshow(cont_matrix, cmap='gray', interpolation='nearest')
    plt.title(f'Original Contingency Matrix: Iter {i+1}')
    plt.colorbar()
    plt.show()

    # LINEAR SUM ASSIGNMENT PROBLEM
    row_ind, col_ind = linear_sum_assignment(-cont_matrix)

    # APPLYING PERMUTATION
    permuted_matrix = cont_matrix[:, col_ind]

    # DISPLAY CONTINGENCY MATRIX
    plt.figure(figsize=(5, 5))
    plt.imshow(permuted_matrix, cmap='gray', interpolation='nearest')
    plt.title(f'Permuted Contingency Matrix: Iter {i+1}')
    plt.colorbar()
    plt.show()

    # CALCULATE ACCURACY
    farthest_init_accuracy = np.sum(np.diag(permuted_matrix)) / np.sum(permuted_matrix
    farthest_init_accuracies.append(farthest_init_accuracy)

    # CALCULATE ADJUSTED RAND INDEX
    farthest_init_ari = adjusted_rand_score(y, y_pred)
    farthest_init_adjusted_rand_scores.append(farthest_init_ari)

    print(f"ITERATION {i+1}: ACCURACY = {farthest_init_accuracy:.4f}, ADJUSTED RAND IN
```



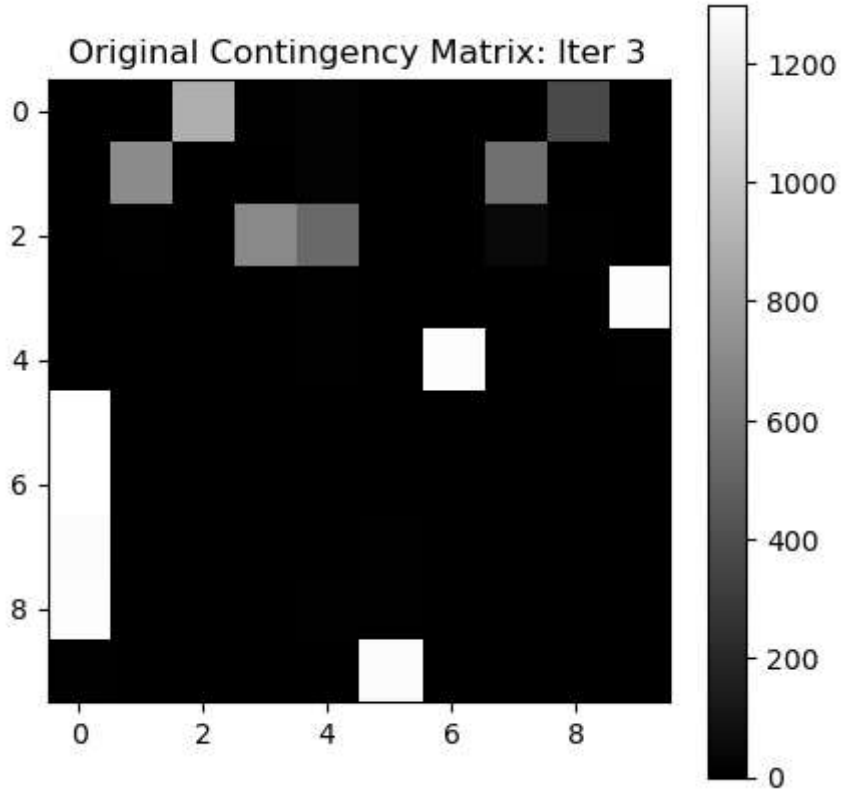


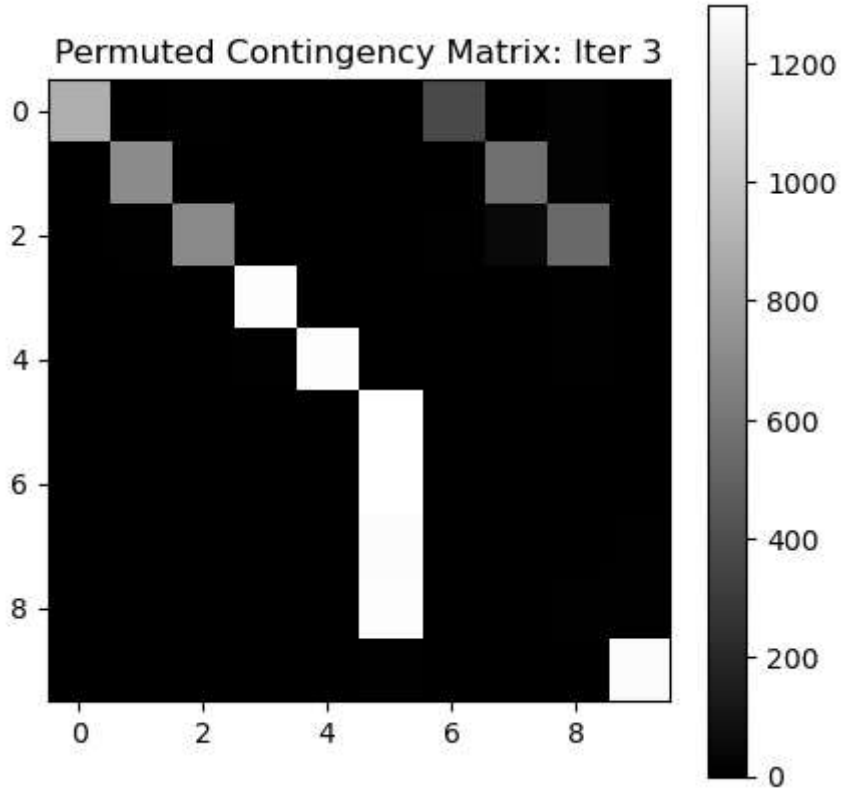

```
ITERATION 1: ACCURACY = 0.3613, ADJUSTED RAND INDEX = 0.3480
```



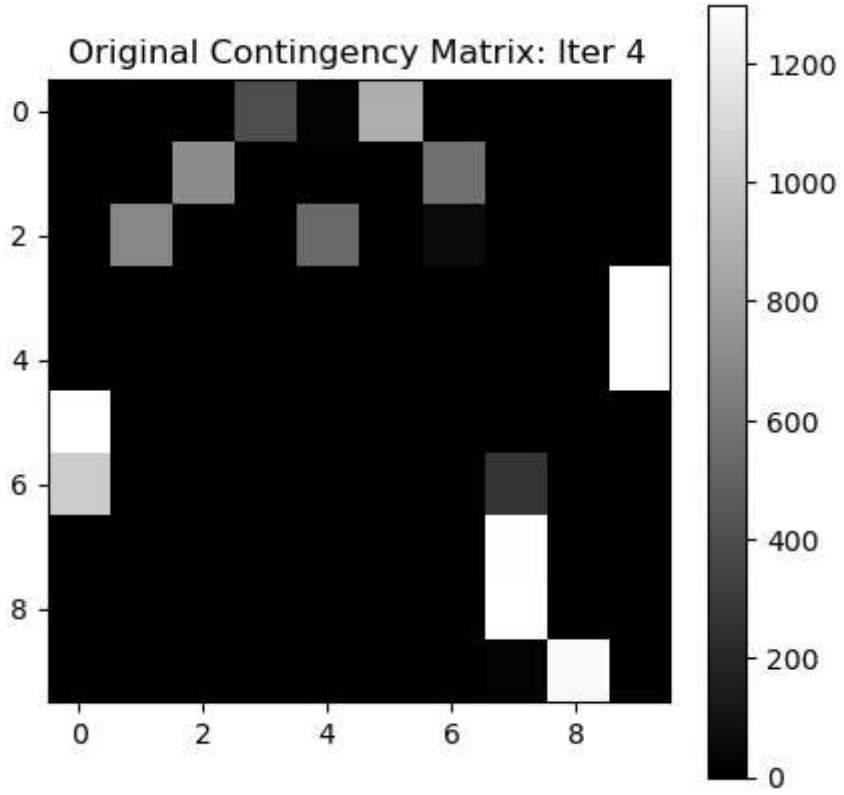


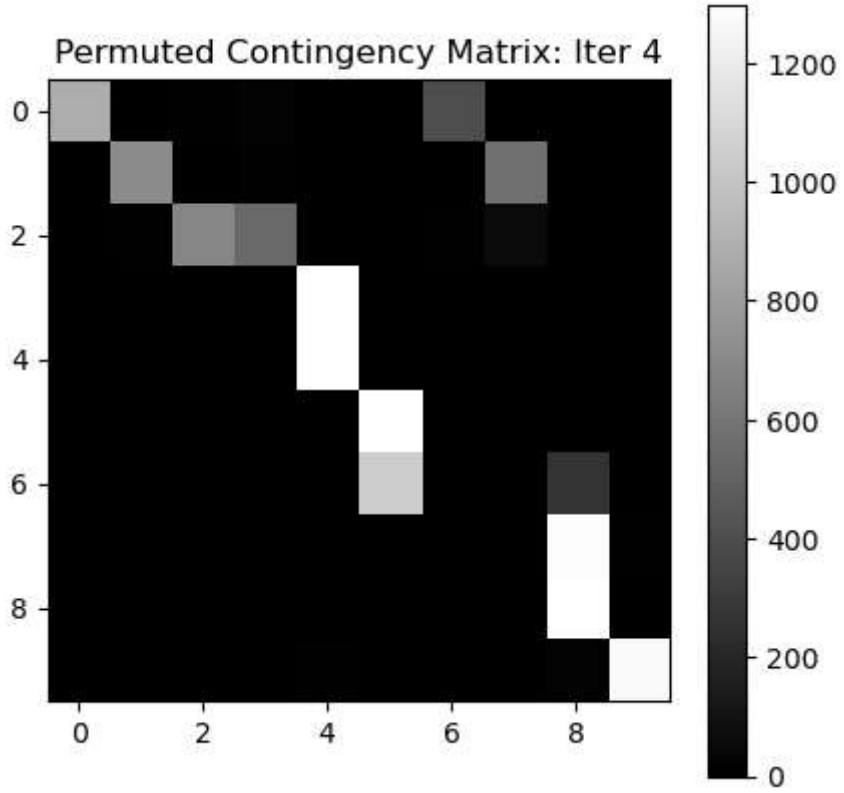

```
ITERATION 2: ACCURACY = 0.4678, ADJUSTED RAND INDEX = 0.4665
```



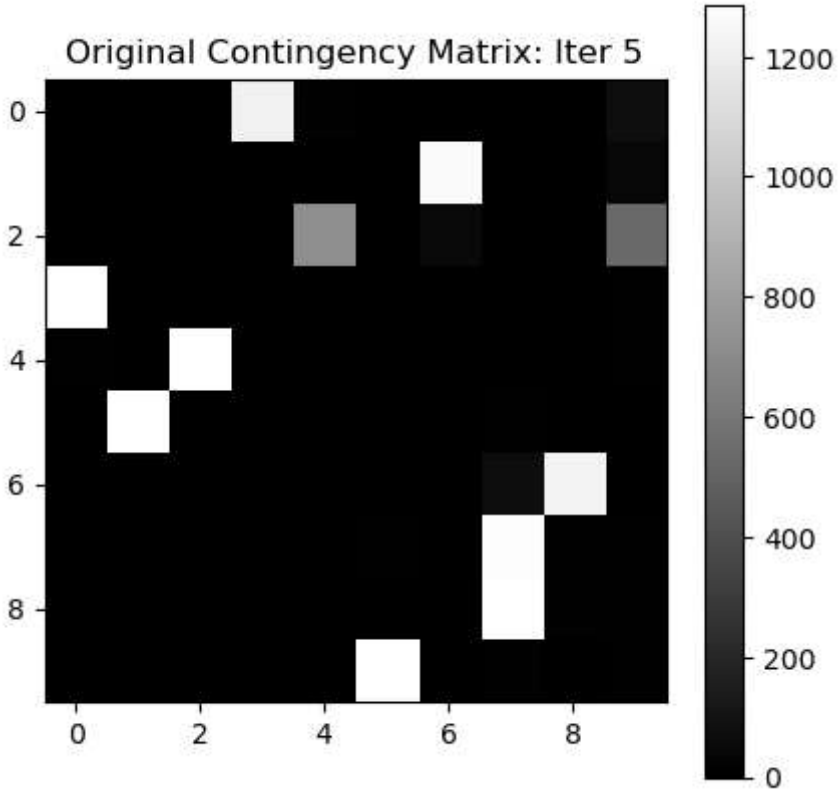


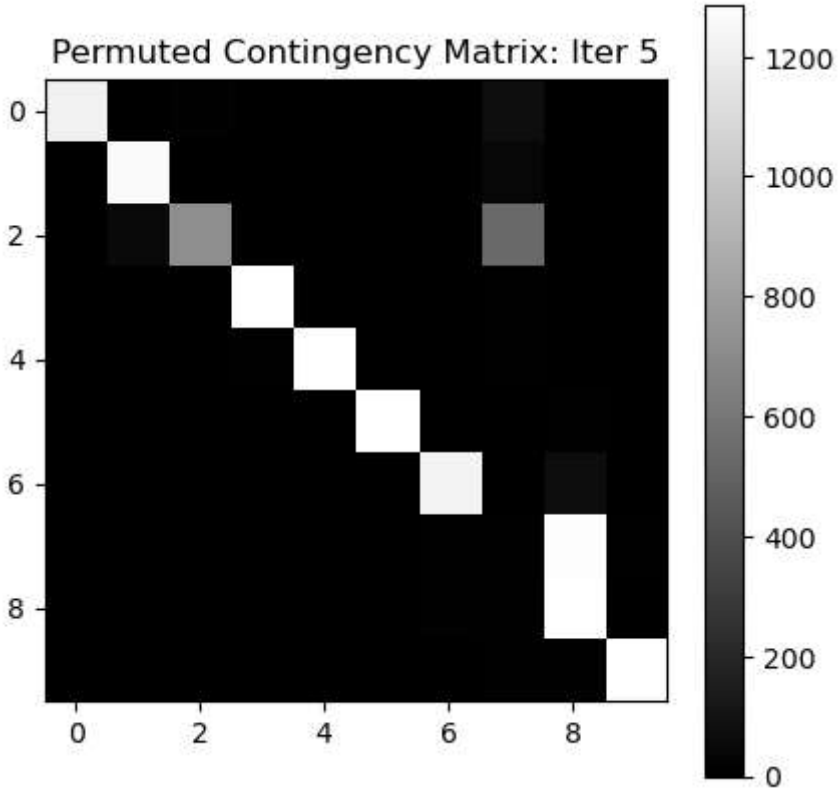

```
ITERATION 3: ACCURACY = 0.5730, ADJUSTED RAND INDEX = 0.4796
```






```
ITERATION 4: ACCURACY = 0.5714, ADJUSTED RAND INDEX = 0.6071
```






```
ITERATION 5: ACCURACY = 0.8335, ADJUSTED RAND INDEX = 0.8184
```

In [72]:
```
# DISPLAY OVERALL RESULTS
print("AVERAGE ACCURACY:", np.mean(farthest_init_accuracies))
print("AVERAGE ADJUSTED RAND INDEX:", np.mean(farthest_init_adjusted_rand_scores))
```

```
AVERAGE ACCURACY: 0.5614
AVERAGE ADJUSTED RAND INDEX: 0.5439399873074628
```

Question 1g

In [73]:
```
# TABLE REPORT
results_table = {
    'INITIALIZATION': ['RANDOM', 'K-MEANS++', 'FARTHEST POINT'],
    'AVERAGE ACCURACY': [np.mean(random_accuracies), np.mean(kmeans_accuracies), np.me
    'AVERAGE ARI': [np.mean(random_adjusted_rand_scores), np.mean(kmeans_adjusted_rand
}
```

In [74]:
```
# PRINT THE TABLE
print(f"{'INITILIZATION':<20} {'AVERAGE ACCURACY':<20} {'AVERAGE ARI':<20}")
for i in range(len(results_table['INITIALIZATION'])):
    print(f"{results_table['INITIALIZATION'][i]:<20} {results_table['AVERAGE ACCURACY'
```

```
INITILIZATION        AVERAGE ACCURACY     AVERAGE ARI
RANDOM               0.7739               0.7484
K-MEANS++            0.8665               0.8505
FARTHEST POINT       0.5614               0.5439
```

In [75]:
```
# DETERMINING WHICH METHOD OBTAINS LARGEST AVG ACCURACY AND AVG ARI
best_accuracy_index = np.argmax(results_table['AVERAGE ACCURACY'])
best_ari_index = np.argmax(results_table['AVERAGE ARI'])
best_accuracy_method = results_table['INITIALIZATION'][best_accuracy_index]
best_ari_method = results_table['INITIALIZATION'][best_ari_index]

print(f"LARGEST AVG ACCURACY SCORE: {best_accuracy_method}")
print(f"LARGEST AVG ARI: {best_ari_method}")
```

```
LARGEST AVG ACCURACY SCORE: K-MEANS++
LARGEST AVG ARI: K-MEANS++
```

Applied Machine Learning Assignment 11 (Kirti Katiyar)

In [1]:
```
# Import necessary libraries
import os  # For operating system related functionalities
import numpy as np  # For numerical operations
from PIL import Image  # For image handling
from sklearn.decomposition import PCA  # For Principal Component Analysis
import matplotlib.pyplot as plt  # For plotting graphs
```

Question 1a

In [2]:
```
# Path to the folder containing the face images
faces_folder_path = 'C:/Users/user/Desktop/Applied Machine Learning/Faces'

# List to store valid face images
valid_faces_images = []
```

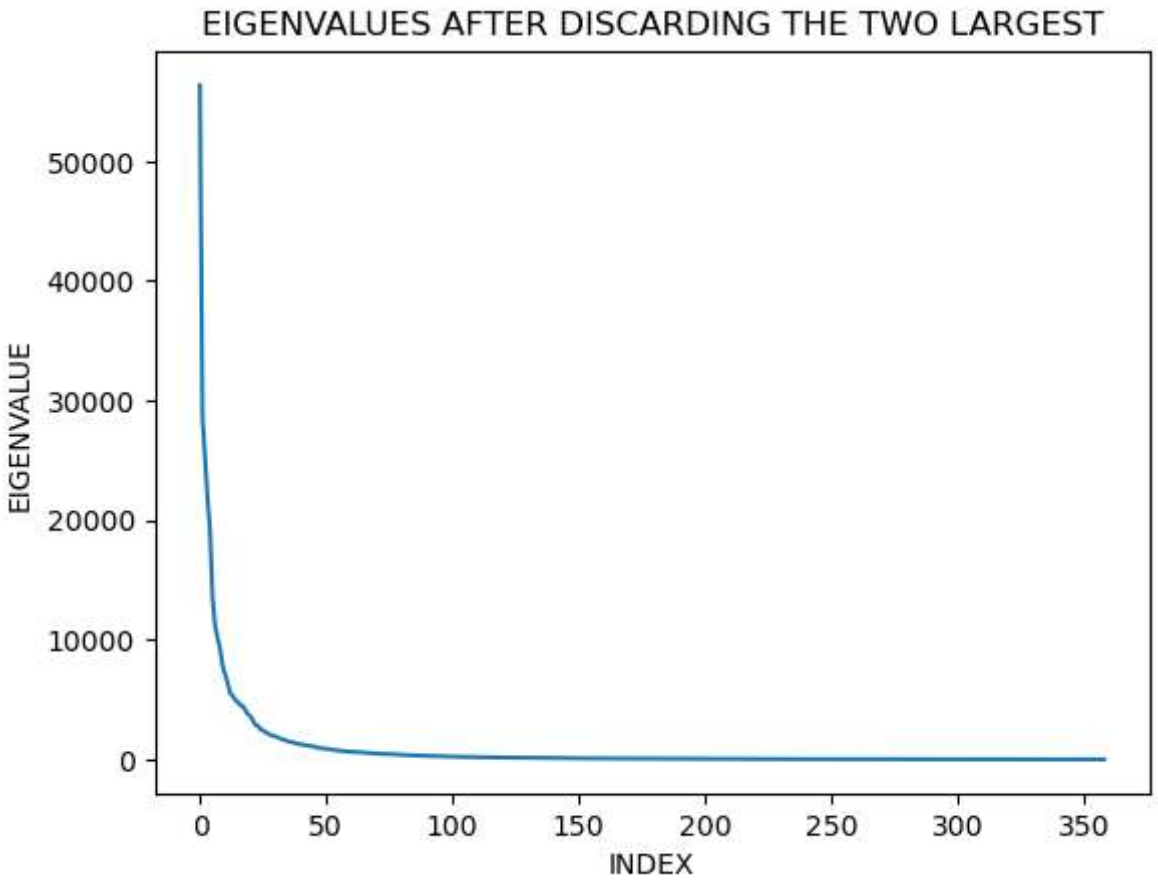
In [3]:
```
# Iterate through each file in the faces folder
for filename in os.listdir(faces_folder_path):
    # Check if the file is a PGM image
    if filename.endswith('.pgm'):
        try:
            # Get the path to the face image
            face_img_path = os.path.join(faces_folder_path, filename)
             # Open the face image using PIL
            face_img = Image.open(face_img_path)
            # Append valid face images to the list
            valid_faces_images.append(face_img)
        except IOError:
            # Handle unreadable face images
            print(f"DISCARDNG UNREADABLE FACE IMAGE: {filename}")
```

In [4]:
```
 # Flatten face images to create a data matrix
faces_data_matrix = np.array([np.array(face_img).flatten() for face_img in valid_faces

#  Standardize the data
faces_data_matrix = faces_data_matrix - np.mean(faces_data_matrix, axis=0)
```

In [5]:
```
# Applying Principal Component Analysis (PCA)
first_pca = PCA()
first_pca.fit(faces_data_matrix)

# Obtaining eigenvalues and excluding the initial two
eigenvalues = first_pca.explained_variance_[2:]

# Arranging eigenvalues in a descending order
sorted_eigenvalues = np.sort(eigenvalues)[::-1]
```

In [6]:
```
# plot the values
plt.plot(sorted_eigenvalues)
```

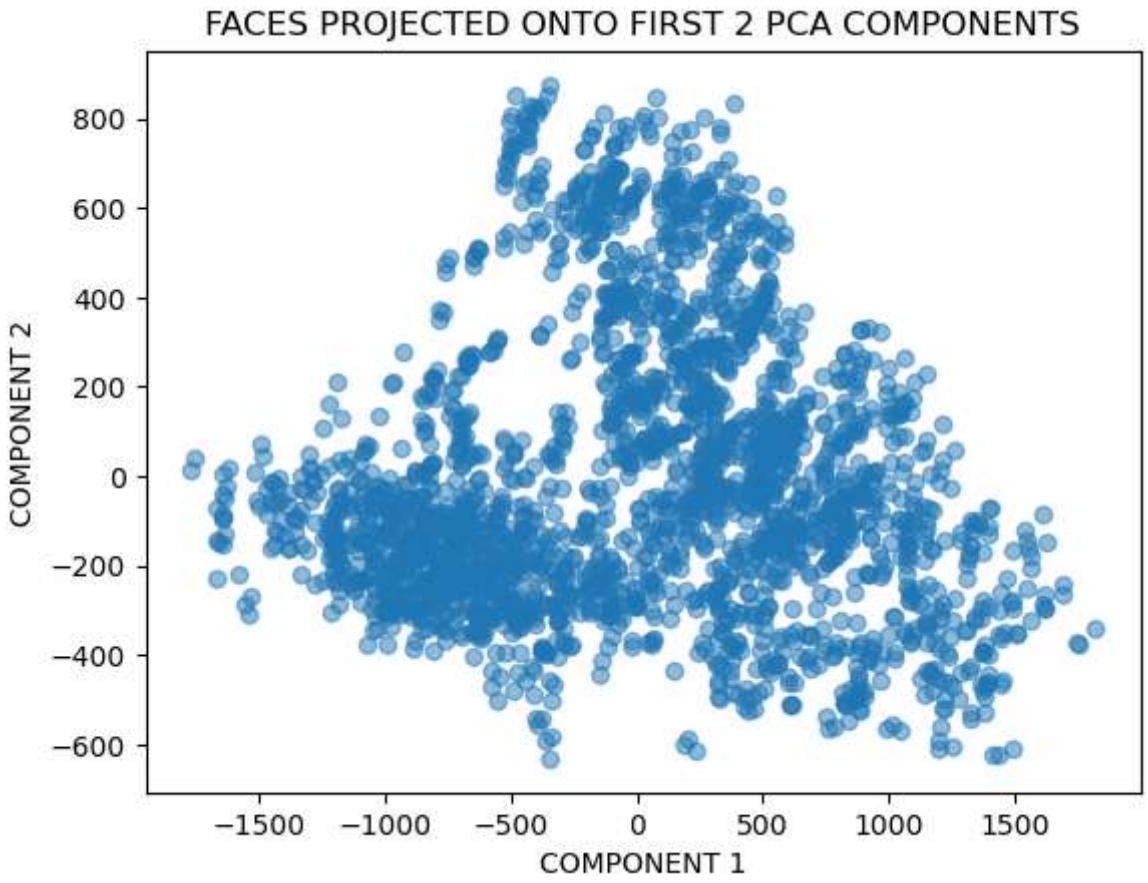
```
plt.title('EIGENVALUES AFTER DISCARDING THE TWO LARGEST')
plt.xlabel('INDEX')
plt.ylabel('EIGENVALUE')
plt.show()
```



Question 1b

In [7]:
```
# add pca
second_pca = PCA(n_components=2)
projected = second_pca.fit_transform(faces_data_matrix)
```

In [8]:
```
# Plot the graph
plt.scatter(projected[:, 0], projected[:, 1], alpha=0.5)
plt.xlabel('COMPONENT 1')
plt.ylabel('COMPONENT 2')
plt.title('FACES PROJECTED ONTO FIRST 2 PCA COMPONENTS')
plt.show()
```



Question 1c

In [9]:
```
# IMAGES DERIVED FROM BACKGROUND DIRECTORY
background_folder_path = 'C:/Users/user/Desktop/Applied Machine Learning/Background'
```

In [10]:
```
# DATA STRUCTURE TO HOLD ALL VALID BACKGROUND IMAGES
valid_background_images = []
```

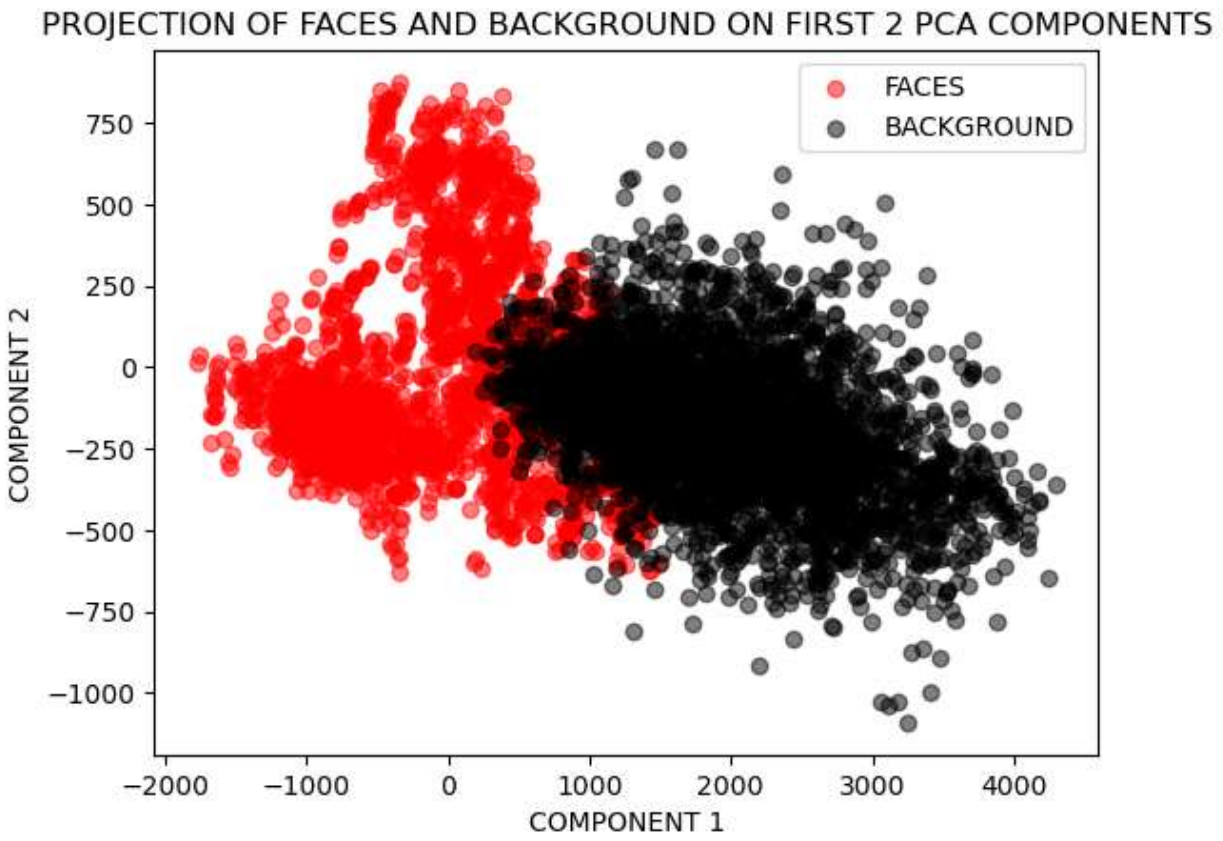
In [11]:
```
# ITERATING THROUGH EACH FILE IN THE BACKGROUND FOLDER
for filename in os.listdir(background_folder_path):
    if filename.endswith('.pgm'):
        try:
            background_img_path = os.path.join(background_folder_path, filename)
            background_img = Image.open(background_img_path)
            valid_background_images.append(background_img)
        except IOError:
            print(f"DISCARDNG UNREADABLE BACKGROUND IMAGE: {filename}")
```

In [12]:
```
# FLATTEN BACKGROUND IMAGES AND CREATE DATA MATRIX
background_data_matrix = np.array([np.array(background_img).flatten() for background_i

# data standardize
background_data_matrix = background_data_matrix - np.mean(faces_data_matrix, axis=0)
```

In [13]:
```
# using pca for face data
third_pca = PCA(n_components=2)
third_pca.fit(faces_data_matrix)
```

```
projected_faces = third_pca.transform(faces_data_matrix)
projected_background = third_pca.transform(background_data_matrix)
```

In [14]:
```
# plot the scatter graph
plt.scatter(projected_faces[:, 0], projected_faces[:, 1], alpha=0.5, color='red', labe
plt.scatter(projected_background[:, 0], projected_background[:, 1], alpha=0.5, color='
plt.xlabel('COMPONENT 1')
plt.ylabel('COMPONENT 2')
plt.title('PROJECTION OF FACES AND BACKGROUND ON FIRST 2 PCA COMPONENTS')
plt.legend()
plt.show()
```



Question 1d

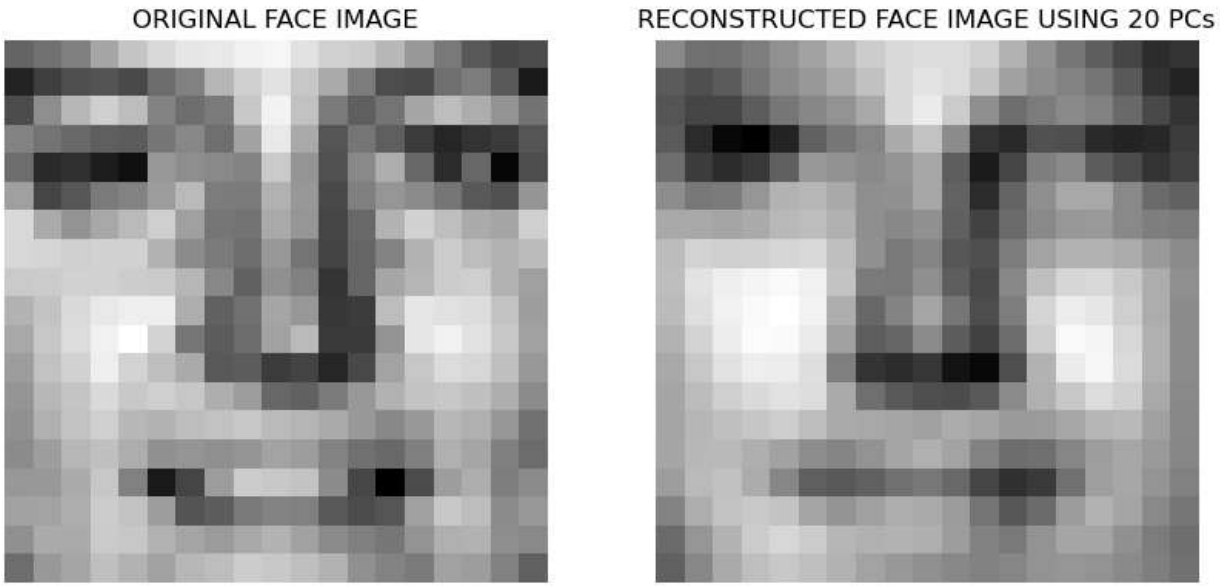
In [15]:
```
# ACCESSING A PARTICULAR FACE IMAGE
face_image_path = 'C:/Users/user/Desktop/Applied Machine Learning/Faces/face00067.pgm'
original_face_img = Image.open(face_image_path)
```

In [16]:
```
# FLATTENING THE IMAGE AND SUBTRACTING THE AVERAGE
face_flat_img = np.array(original_face_img).flatten()
face_flat_img_centered = face_flat_img - np.mean(face_flat_img)
```

In [17]:
```
# APPLYING PCA WITH 20 COMPONENTS (BASED ON THE 20-COMPONENT PCA MODEL FROM PART A)
pca = PCA(n_components=20)
pca.fit(faces_data_matrix)

transformed_face_img = pca.transform([face_flat_img_centered])
reconstructed_face_img = pca.inverse_transform(transformed_face_img)
```

In [18]:
```
# RESTORE THE RECONSTRUCTED FACE IMAGE TO ITS ORIGINAL DIMENSIONS
reconstructed_face_img = reconstructed_face_img.reshape(original_face_img.size)
```

In [19]:
```
# plot the images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(original_face_img, cmap='gray')
axes[0].set_title('ORIGINAL FACE IMAGE')
axes[0].axis('off')

axes[1].imshow(reconstructed_face_img, cmap='gray')
axes[1].set_title('RECONSTRUCTED FACE IMAGE USING 20 PCs')
axes[1].axis('off')

plt.show()
```



Question 1e

In [20]:
```
# ACCESSING A PARTICULAR BACKGROUND IMAGE
background_image_path = 'C:/Users/user/Desktop/Applied Machine Learning/Background/B1_
original_background_img = Image.open(background_image_path)
```

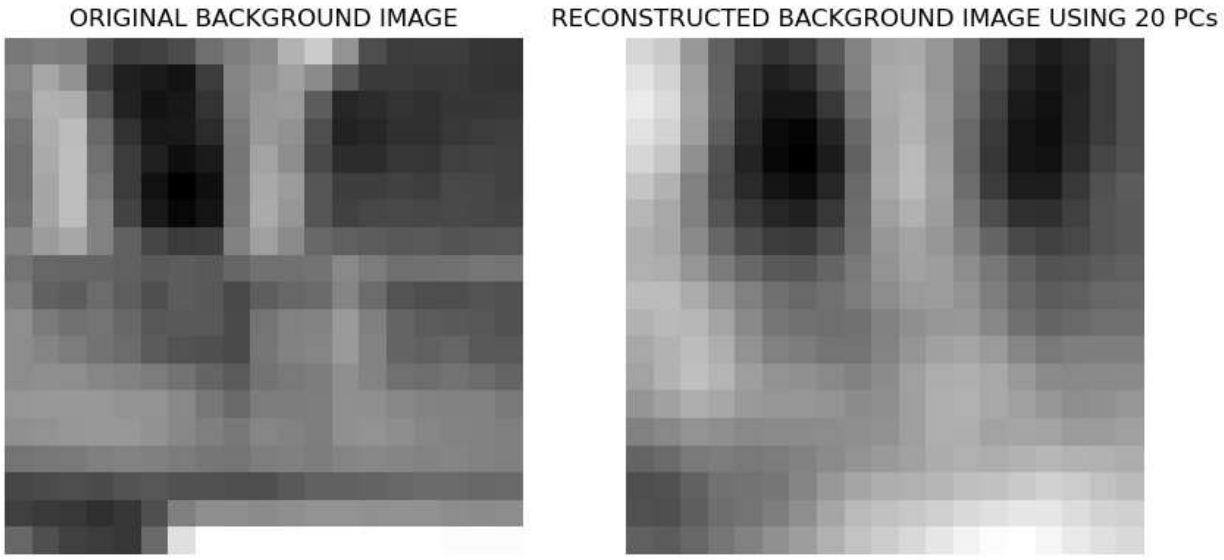
In [21]:
```
# FLATTENING THE IMAGE AND SUBTRACTING THE MEAN
background_flat_img = np.array(original_background_img).flatten()
background_flat_img_centered = background_flat_img - np.mean(background_flat_img)
```

In [22]:
```
# APPLYING PCA WITH 20 COMPONENTS (BASED ON THE PCA MODEL FROM PART A)
pca = PCA(n_components=20)
pca.fit(background_data_matrix)
```

```
transformed_background_img = pca.transform([background_flat_img_centered])
reconstructed_background_img = pca.inverse_transform(transformed_background_img)
```

In [23]:
```
#  RESTORE THE RECONSTRUCTED BACKGROUND IMAGE TO ITS ORIGINAL DIMENSIONS
reconstructed_background_img = reconstructed_background_img.reshape(original_backgrour
```

In [24]:
```
# plot the images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(original_background_img, cmap='gray')
axes[0].set_title('ORIGINAL BACKGROUND IMAGE')
axes[0].axis('off')

axes[1].imshow(reconstructed_background_img, cmap='gray')
axes[1].set_title('RECONSTRUCTED BACKGROUND IMAGE USING 20 PCs')
axes[1].axis('off')

plt.show()
```



Question 1f

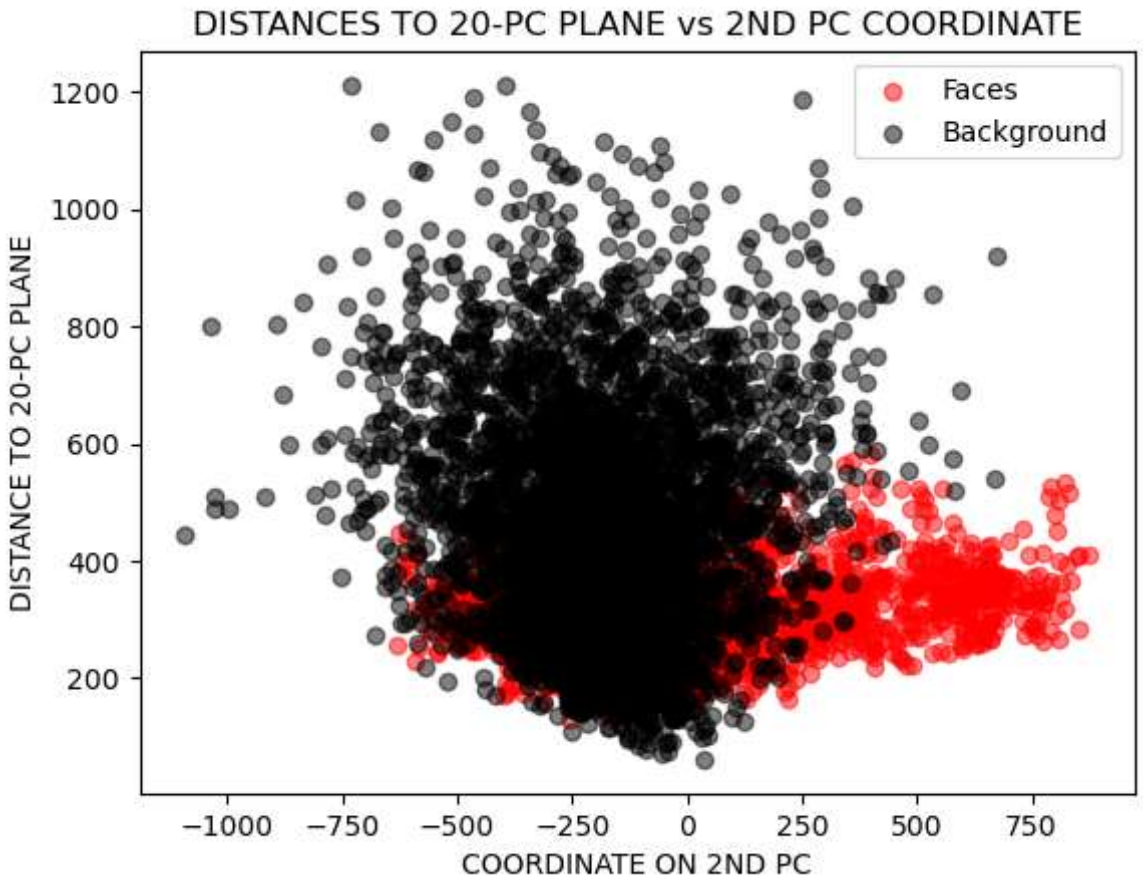
In [25]:
```
# PERFORMING PCA WITH 20 COMPONENTS
fourth_pca_20 = PCA(n_components=20)
fourth_pca_20.fit(faces_data_matrix) # TRAINING ONLY ON FACE DATA

# MAPPING BOTH DATASETS ONTO THE 20 PRINCIPAL COMPONENTS
projected_faces = fourth_pca_20.transform(faces_data_matrix)
projected_background = fourth_pca_20.transform(background_data_matrix)
```

In [26]:
```
# GENERATING IMAGES USING THE INFORMATION FROM THE 20 PRINCIPAL COMPONENTS
reconstructed_faces = fourth_pca_20.inverse_transform(projected_faces)
reconstructed_background = fourth_pca_20.inverse_transform(projected_background)
```

In [27]:
```
# COMPUTING DISTANCES TO THE 20-PRINCIPAL COMPONENT PLANE
distances_faces = np.sqrt(np.sum((faces_data_matrix - reconstructed_faces)**2, axis=1)
distances_background = np.sqrt(np.sum((background_data_matrix - reconstructed_backgrou
```

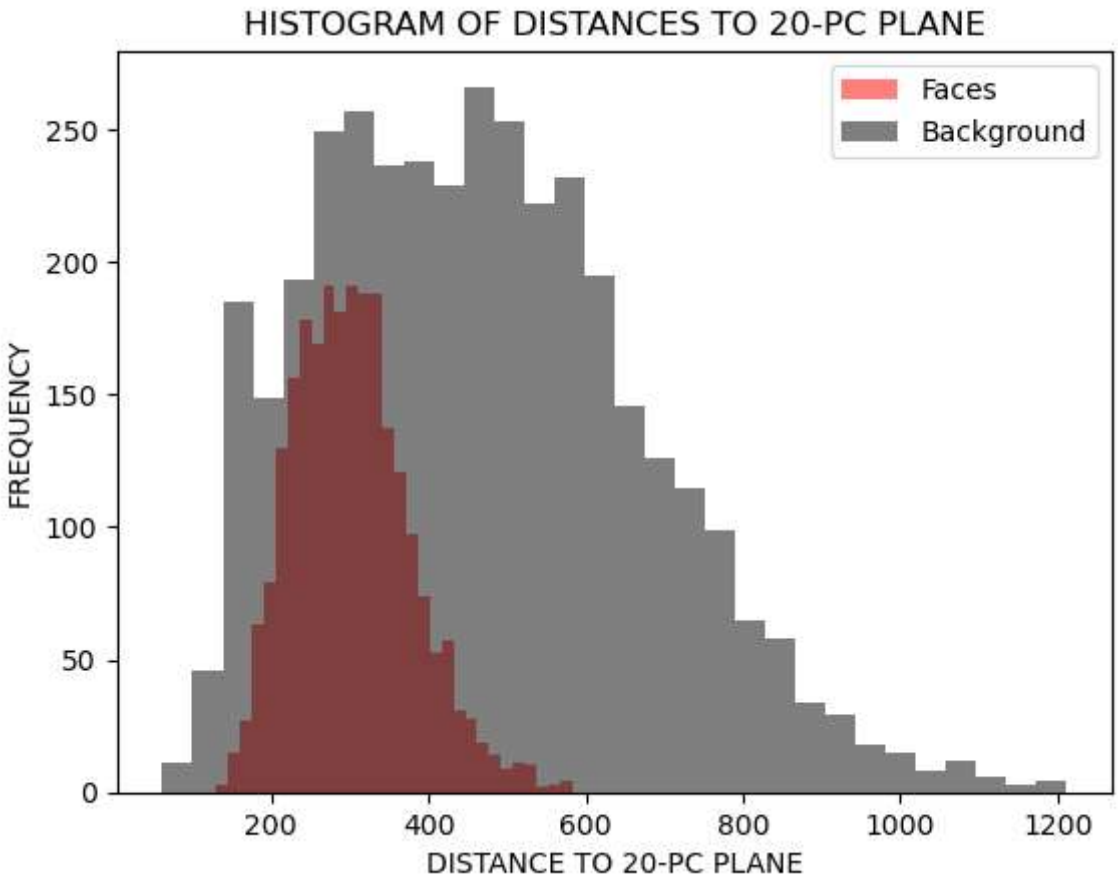
In [28]:
```
# COORDINATES ALONG THE SECOND PRINCIPAL COMPONENT
second_pc_faces = projected_faces[:, 1]
second_pc_background = projected_background[:, 1]
```

In [29]:
```
# plot the scatter images
plt.scatter(second_pc_faces, distances_faces, alpha=0.5, color='red', label='Faces')
plt.scatter(second_pc_background, distances_background, alpha=0.5, color='black', labe
plt.xlabel('COORDINATE ON 2ND PC')
plt.ylabel('DISTANCE TO 20-PC PLANE')
plt.title('DISTANCES TO 20-PC PLANE vs 2ND PC COORDINATE')
plt.legend()
plt.show()
```



Question 1g

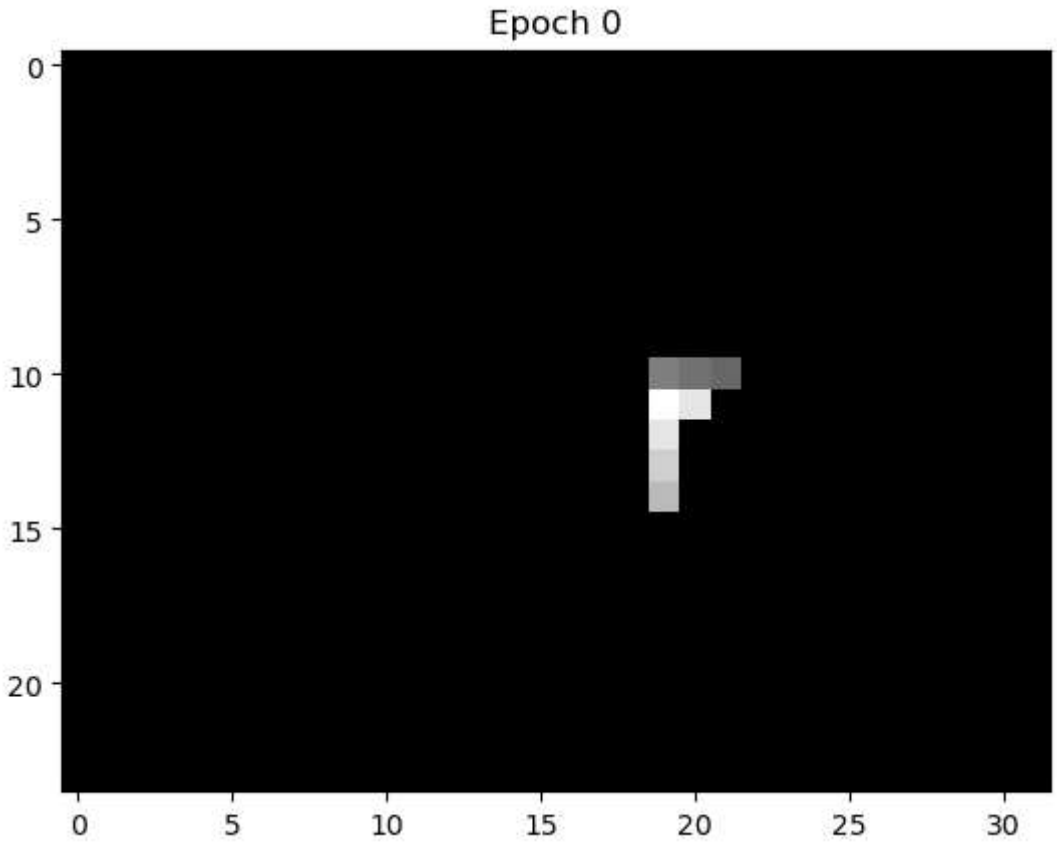
In [30]:
```
# plot the histogram
plt.hist(distances_faces, bins=30, alpha=0.5, color='red', label='Faces')
plt.hist(distances_background, bins=30, alpha=0.5, color='black', label='Background')
plt.xlabel('DISTANCE TO 20-PC PLANE')
plt.ylabel('FREQUENCY')
plt.title('HISTOGRAM OF DISTANCES TO 20-PC PLANE')
plt.legend()
plt.show()
```

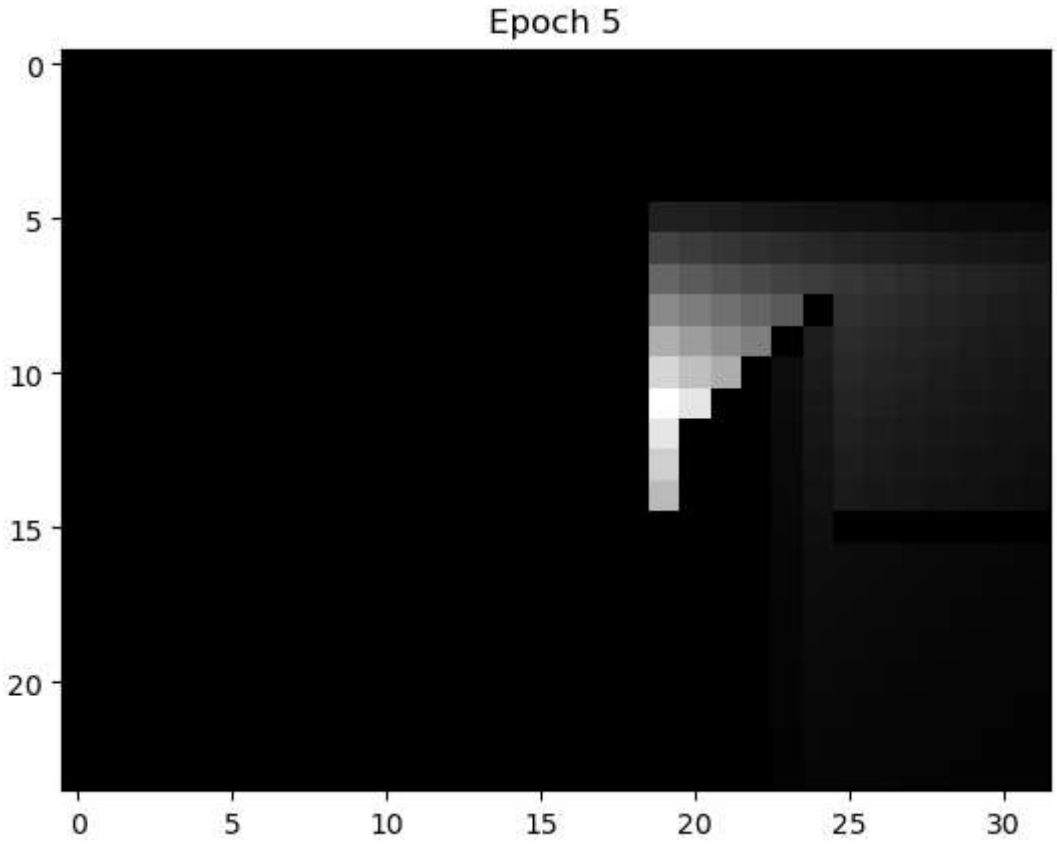

HISTOGRAM OF DISTANCES TO 20-PC PLANE
Faces
Background
FREQUENCY
0
50
100
150
200
250
200
400
600
800
1000
1200
DISTANCE TO 20-PC PLANE

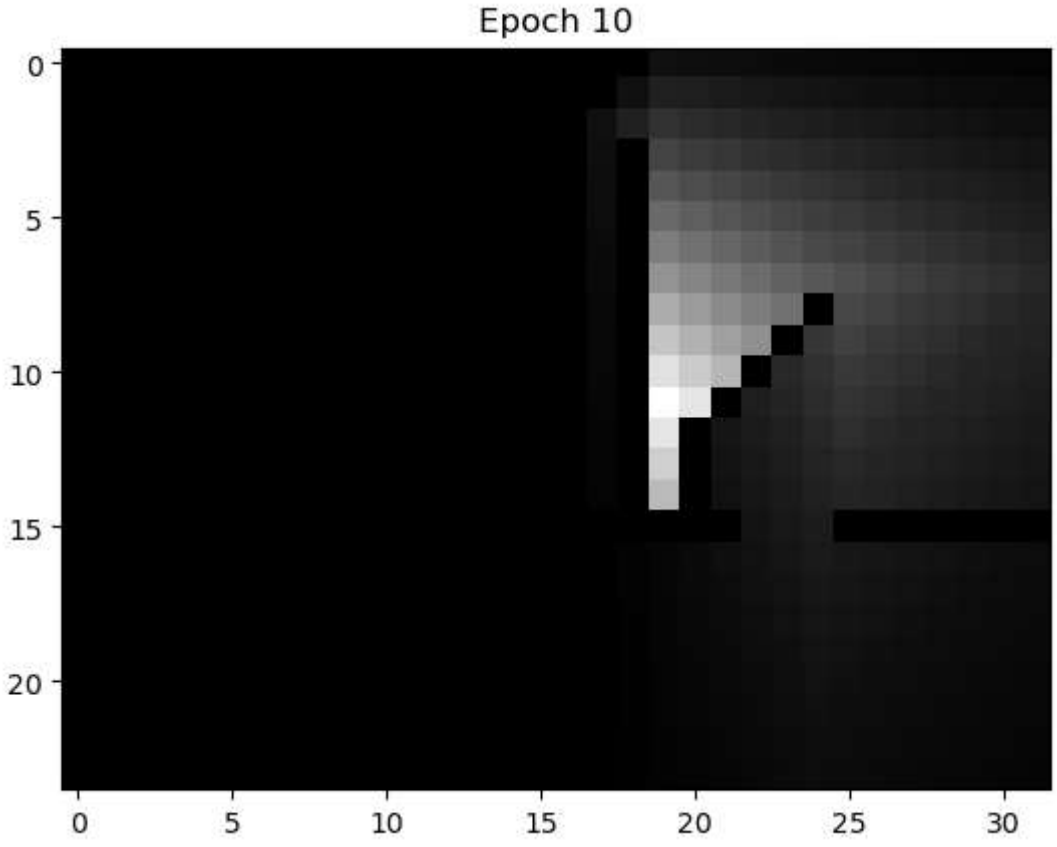
AML Assignment - 12 (Kirti Katiyar)

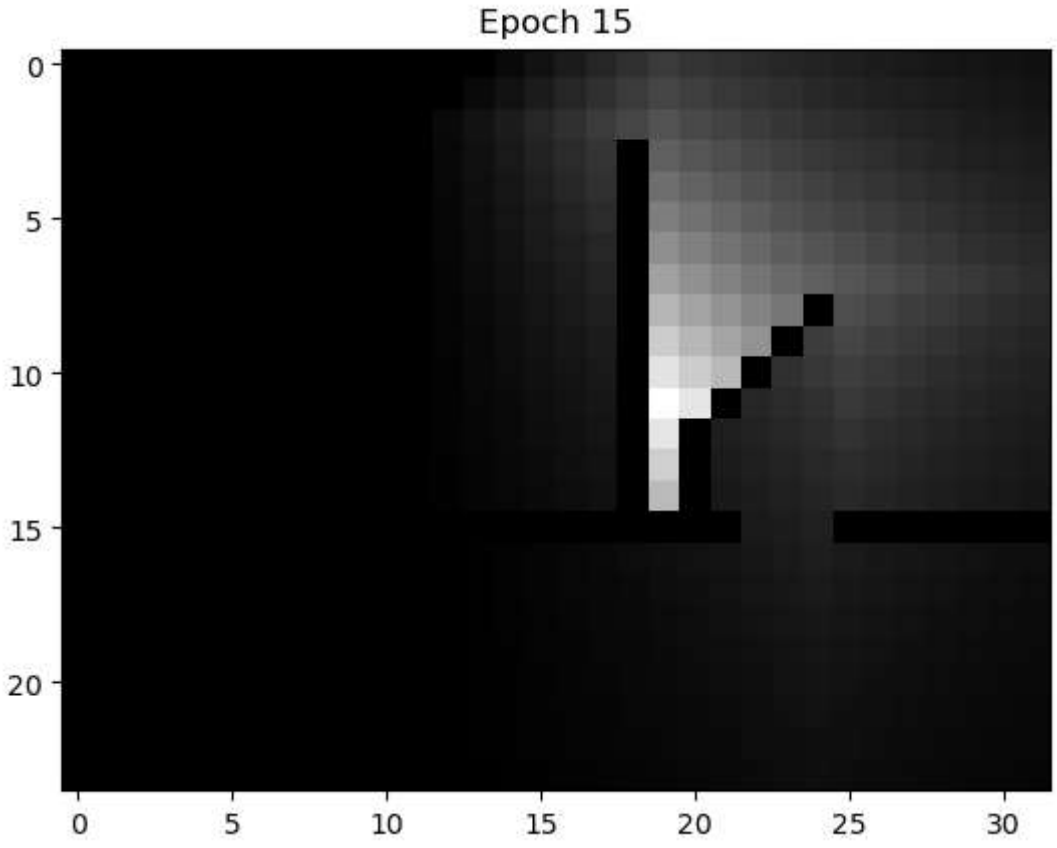
Question - 1

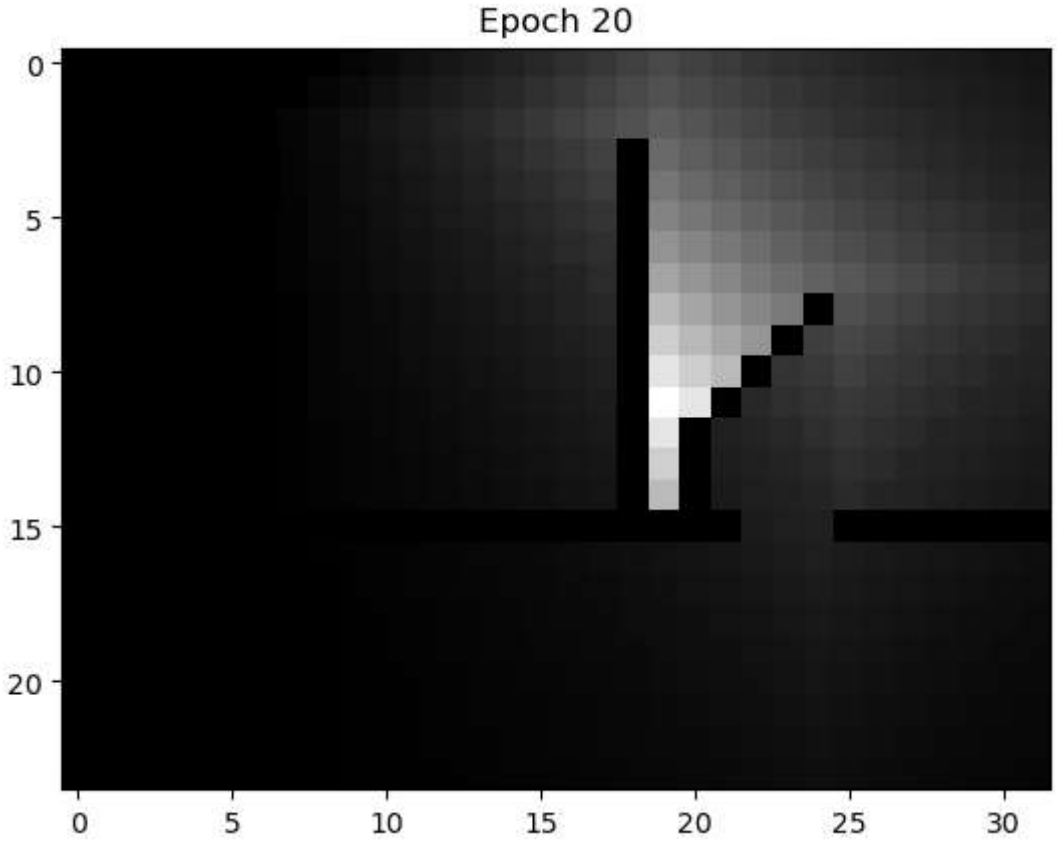
In [7]:
```
# Load the Required Libraries
import numpy as np
import matplotlib.pyplot as plt

#Import the map data stored in the CSV file.
map_data = np.genfromtxt('map_24x32.csv', delimiter=',')

# adding the required Constant values
gamma = 0.9
max_epochs = 50
valid_locations = map_data >= 0

# Set up V(s) using the provided map values within valid locations.
V = np.copy(map_data)
V[~valid_locations] = 0

# Specify the available actions as (left, right, up, down).
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
action_symbols = ['L', 'R', 'U', 'D']

# The iterative process for determining values
for epoch in range(max_epochs):
    for i in range(V.shape[0]):
        for j in range(V.shape[1]):
            if not valid_locations[i, j]:
                continue

            # Compute the Q-values associated with each action.
            Q_values = []
            for action in actions:
                next_i, next_j = i + action[0], j + action[1]
                next_i = max(0, min(V.shape[0] - 1, next_i))  # Ensure next_i is withi
                next_j = max(0, min(V.shape[1] - 1, next_j))  # Ensure next_j is withi
                Q_values.append(V[next_i, next_j])

            # Revise V(s) by applying the Bellman equation.
            V[i, j] = map_data[i, j] + gamma * max(Q_values)

    # Showcase V(s) every 5 epochs.
    if epoch % 5 == 0:
        plt.imshow(V, cmap='gray', interpolation='none')
        plt.title(f'Epoch {epoch}')
        plt.show()
```

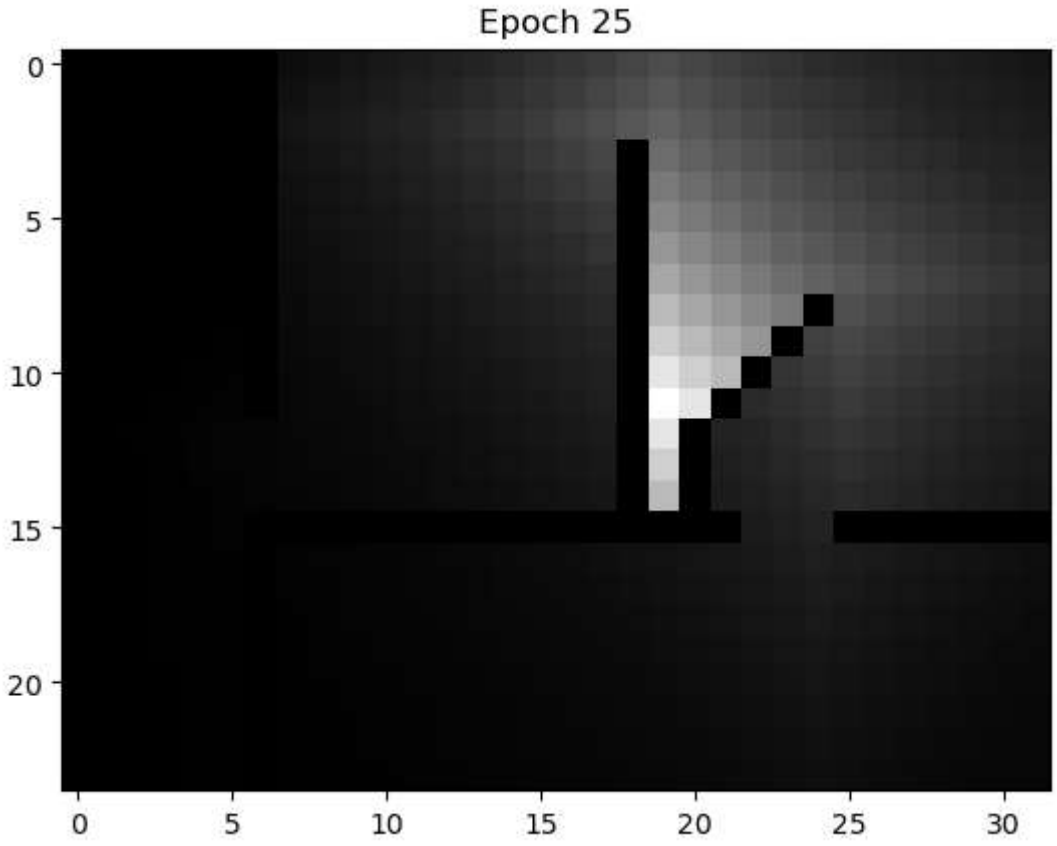

Epoch 0
0
5
10
15
20
0
5
10
15
20
25
30

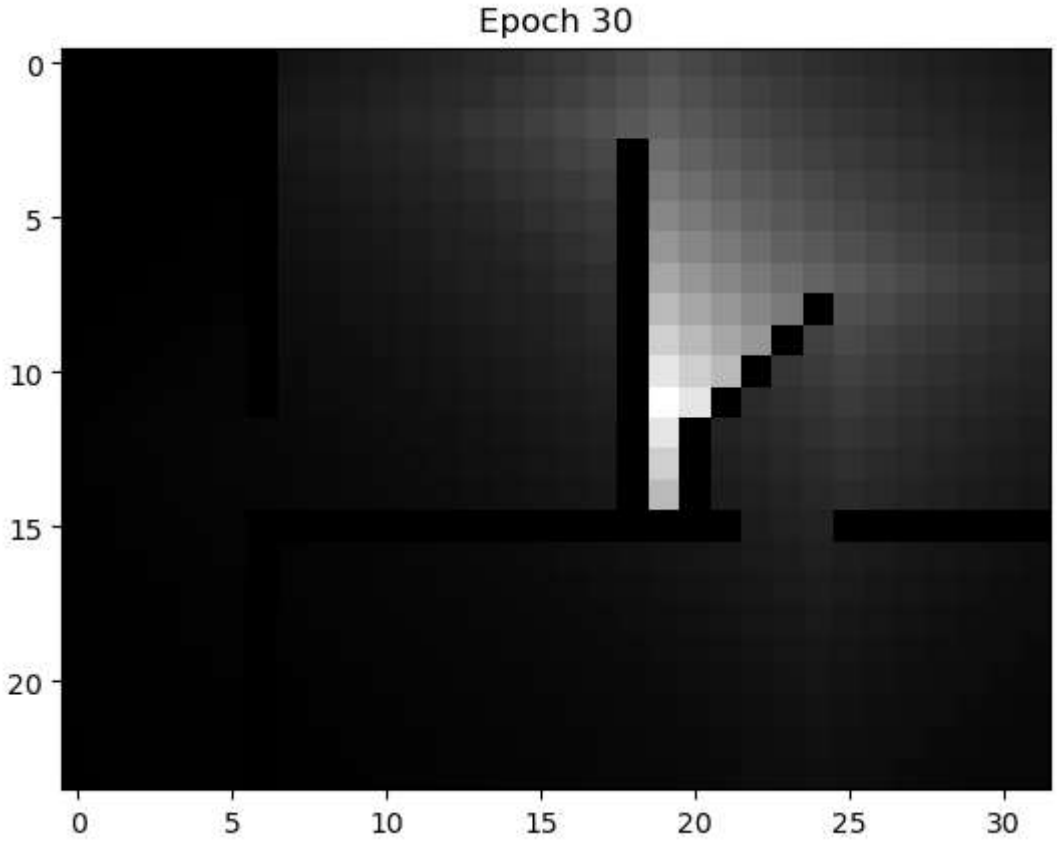

Epoch 5
0
5
10
15
20
0
5
10
15
20
25
30

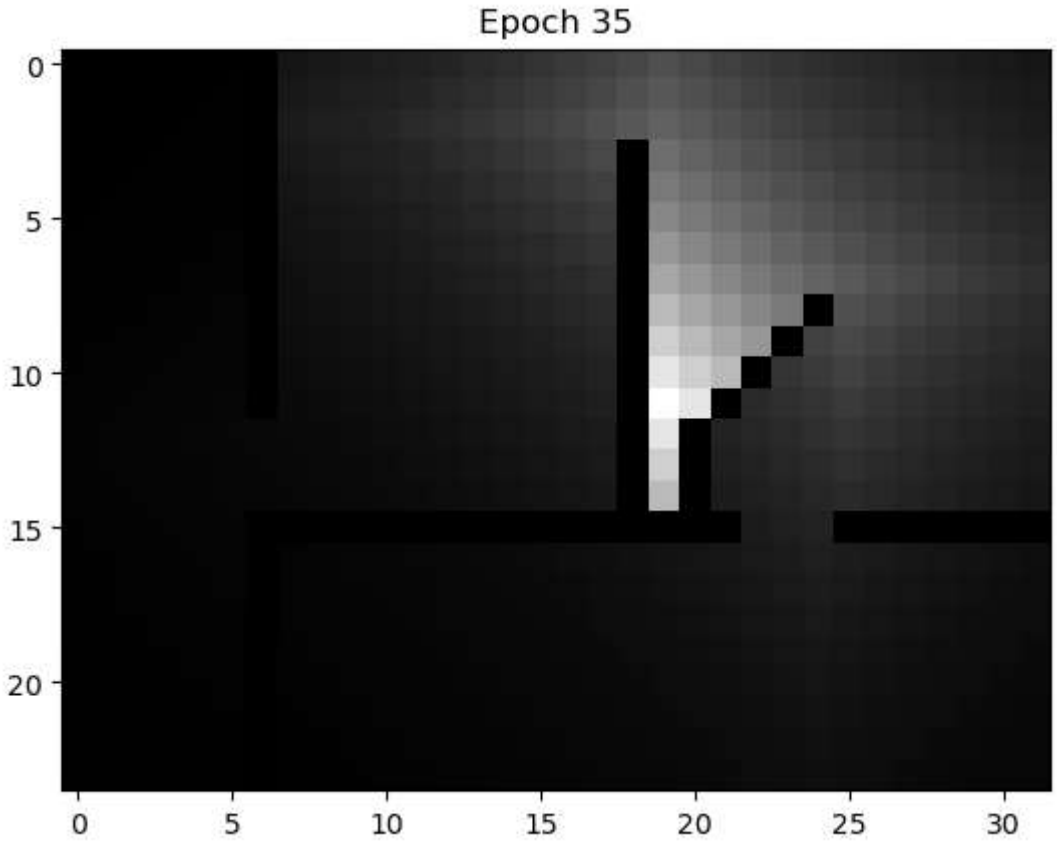

Epoch 10
0
5
10
15
20
0
5
10
15
20
25
30

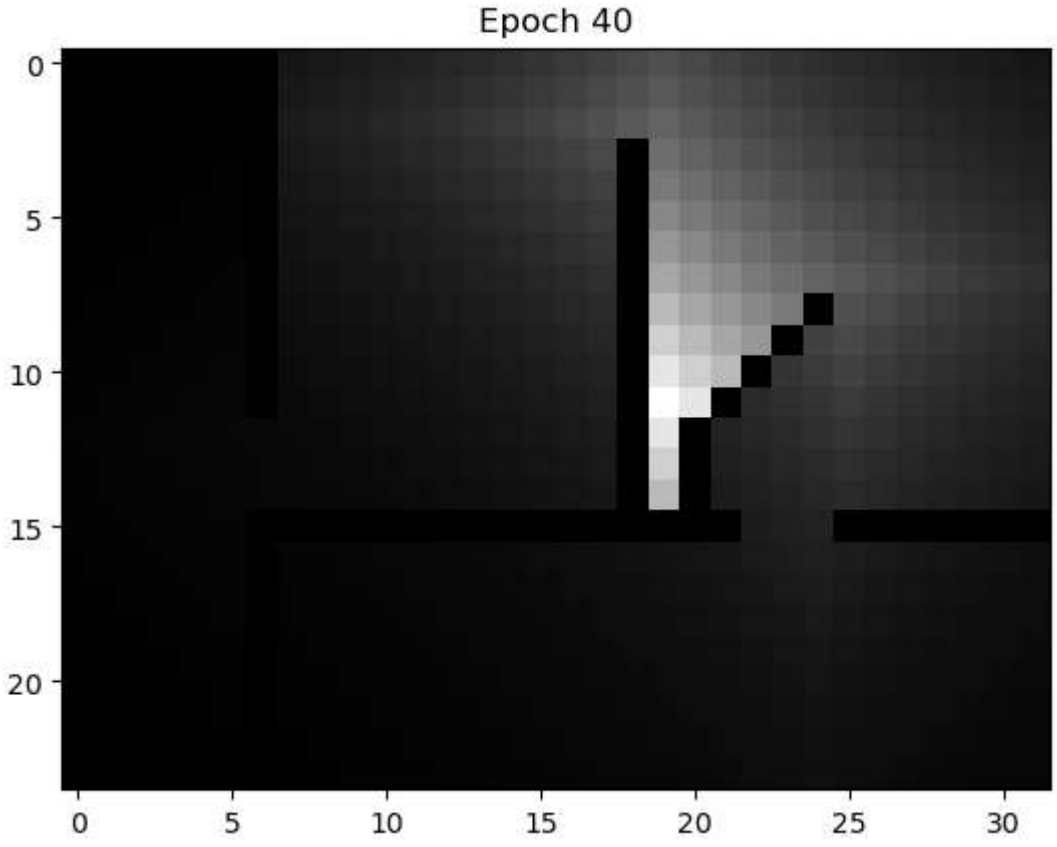

Epoch 15
0
5
10
15
20
0
5
10
15
20
25
30

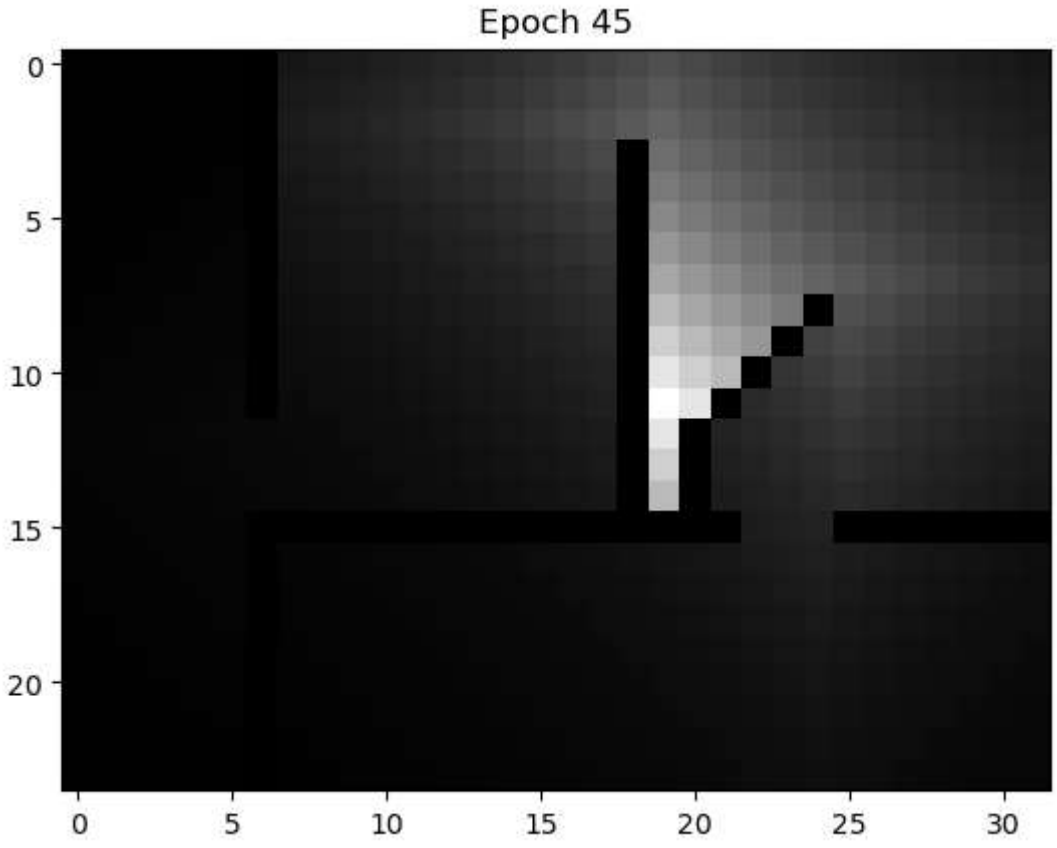

Epoch 20
0
5
10
15
20
0
5
10
15
20
25
30


Epoch 25
0
5
10
15
20
0
5
10
15
20
25
30


Epoch 30
0
5
10
15
20
0
5
10
15
20
25
30


Epoch 35
0
5
10
15
20
0
5
10
15
20
25
30


Epoch 40
0
5
10
15
20
0
5
10
15
20
25
30


Epoch 45
0
5
10
15
20
0
5
10
15
20
25
30

Question - 2

In [3]:
```
# Display the final learned policy as a table the actions at each valid location encoa
policy_table = np.empty_like(map_data, dtype='<U1')
policy_table[~valid_locations] = 'X'  # Mark invalid locations with 'X'

for i in range(V.shape[0]):
    for j in range(V.shape[1]):
        if valid_locations[i, j]:
            # Find the optimal action for every valid position.
            best_action_index = np.argmax([V[max(0, min(V.shape[0] - 1, i + a[0])), ma
            policy_table[i, j] = action_symbols[best_action_index]
# Show the final result in a tabular form
print("Final Policy:")
print(policy_table)
```

```
Final Policy:
[['R' 'R' 'R' 'R' 'R' 'R' 'X' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R'
  'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R'
  'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D'
  'D' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'R' 'R' 'R' 'R' 'R' 'U' 'U' 'U' 'U' 'U' 'U' 'U' 'U']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'R' 'R' 'R' 'R' 'U' 'X' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'R' 'R' 'R' 'U' 'X' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'R' 'R' 'U' 'X' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['R' 'R' 'R' 'R' 'R' 'R' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'R' 'U' 'X' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'L' 'X' 'D' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'L' 'X' 'D' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'L'
  'X' 'L' 'X' 'D' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['D' 'D' 'D' 'D' 'D' 'L' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X'
  'X' 'X' 'X' 'X' 'D' 'D' 'L' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
 ['D' 'D' 'D' 'D' 'D' 'L' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D'
  'D' 'D' 'D' 'D' 'D' 'D' 'L' 'U' 'U' 'U' 'U' 'U' 'U' 'U']
 ['D' 'D' 'D' 'D' 'D' 'L' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D'
  'D' 'D' 'D' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['D' 'D' 'D' 'D' 'D' 'L' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D'
  'D' 'D' 'D' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['D' 'D' 'D' 'D' 'D' 'L' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D'
  'D' 'D' 'D' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['D' 'D' 'D' 'D' 'D' 'L' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D'
  'D' 'D' 'D' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['D' 'D' 'D' 'D' 'D' 'L' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D'
  'D' 'D' 'D' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['D' 'D' 'D' 'D' 'D' 'L' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D'
  'D' 'D' 'D' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
 ['D' 'D' 'D' 'D' 'D' 'L' 'X' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D' 'D'
  'D' 'D' 'D' 'D' 'D' 'D' 'L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']]
```