Principal Component Analysis

The goal of this question is to build a conceptual understanding of dimensionality reduction using PCA and implement it on a toy dataset. You'll only have to use numpy and matplotlib for this question.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
```

In [7]:

```
# (a) Load data (features)
def load_data():
    filename = "/home/akshay/Downloads/MAIL/24787HW4-F21/q1-data/features.npy"
    data_ = np.load(filename)
    mean_data = np.mean(data_, axis = 0)
    std_ = np.std(data_, axis = 0)
    data = (data_ - mean_data) / std_
    return data
```

In [3]:

```
# (b) Perform eigen decomposition and return eigen pairs in desecending order of eigen values
def eigendecomp(X):
    covariance = (1 / X.shape[0]) * (X.T @ X)
    e, v = np.linalg.eig(covariance)
    sort_idx = np.argsort(e)
    sort_idx = sort_idx[::-1]
    sorted_eig_vals = e[sort_idx]
    sorted_eig_vecs = v[sort_idx]
    return (sorted_eig_vals, sorted_eig_vecs)
```

In [56]:

```
# (c) Evaluate using variance_explained as the metric
def eval(sorted_eig_vals):
    sum_eig = np.sum(sorted_eig_vals)
    for k in range(1, sorted_eig_vals.shape[0]+1, 1):
        print("k:", k, np.round(np.sum(sorted_eig_vals[:k]) / sum_eig, 3), "Eigen Values: ", sorte
#np.round(np.sum(sorted_eig_vals[:k]) / sum_eig, 3),
```

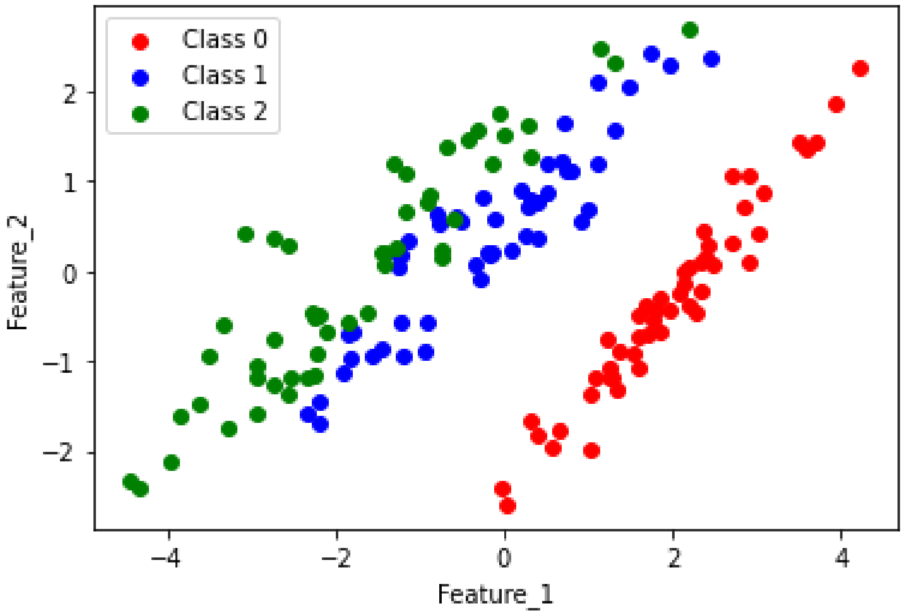
In [57]:

```python
# (d) Visualize after projecting to 2-D space
def viz(sorted_eig_vals, sorted_eig_vecs, data):
    projected = data @ sorted_eig_vecs[:,:2]
    filename = "/home/akshay/Downloads/MAIL/24787HW4-F21/q1-data/labels.npy"
    labels = np.load(filename, allow_pickle=True)
    class_0 = projected[np.where(labels == 0)]
    class_1 = projected[np.where(labels == 1)]
    class_2 = projected[np.where(labels == 2)]
    fig, ax = plt.subplots(1)
    ax.scatter(class_0[:,0], class_0[:,1], color='r', label="Class 0")
    ax.scatter(class_1[:,0], class_1[:,1], color='b', label="Class 1")
    ax.scatter(class_2[:,0], class_2[:,1], color='g', label="Class 2")
    ax.legend()
    ax.set_xlabel("Feature_1")
    ax.set_ylabel("Feature_2")
```

In [58]:

```python
def main():
    data = load_data()
    sorted_eig_vals, sorted_eig_vecs = eigendecomp(data)
    eval(sorted_eig_vals)
    viz(sorted_eig_vals, sorted_eig_vecs, data)

if __name__ == "__main__":
    main()
```

```
k: 1 0.589 Eigen Values:  [4.71136968]
k: 2 0.874 Eigen Values:  [4.71136968 2.2805474 ]
k: 3 0.97 Eigen Values:  [4.71136968 2.2805474  0.77173111]
k: 4 0.996 Eigen Values:  [4.71136968 2.2805474  0.77173111 0.20281175]
k: 5 1.0 Eigen Values:  [4.71136968 2.2805474  0.77173111 0.20281175 0.03354006]
k: 6 1.0 Eigen Values:  [4.71136968e+00 2.28054740e+00 7.71731109e-01 2.02811748e-01
 3.35400649e-02 5.53596026e-16]
k: 7 1.0 Eigen Values:  [4.71136968e+00 2.28054740e+00 7.71731109e-01 2.02811748e-01
 3.35400649e-02 5.53596026e-16 3.03086151e-16]
k: 8 1.0 Eigen Values:  [ 4.71136968e+00  2.28054740e+00  7.71731109e-01  2.02811748e-01
  3.35400649e-02  5.53596026e-16  3.03086151e-16 -6.82293804e-16]
```



(e1): If the number of features is 1000 and the number of data points is 10, what will be the dimension of your covariance matrix? Can you suggest what can be changed to improve the performance?

The covariance matrix will be 1000*1000 dimension.

To improve the performance

1. We should increase the number of data collected.
2. We should use PCA to reduce the number of features, preferably to 1 or 2. This will remove the redundant features

(e2): Assume you have a dataset with the original dimensionality as 2 and you have to reduce it to 1. Provide a sample scatter plot of the original data (less than 10 datapoints) where PCA might produce misleading results. You can plot it by hand and then take a picture. In the next cell, switch to Markdown mode and use the command: `![title]()`

The following x+y is a dataset of 9 data with 2 dimensionality, if we make it 1 dimension the data at one end will be misclassified.

In [48]:

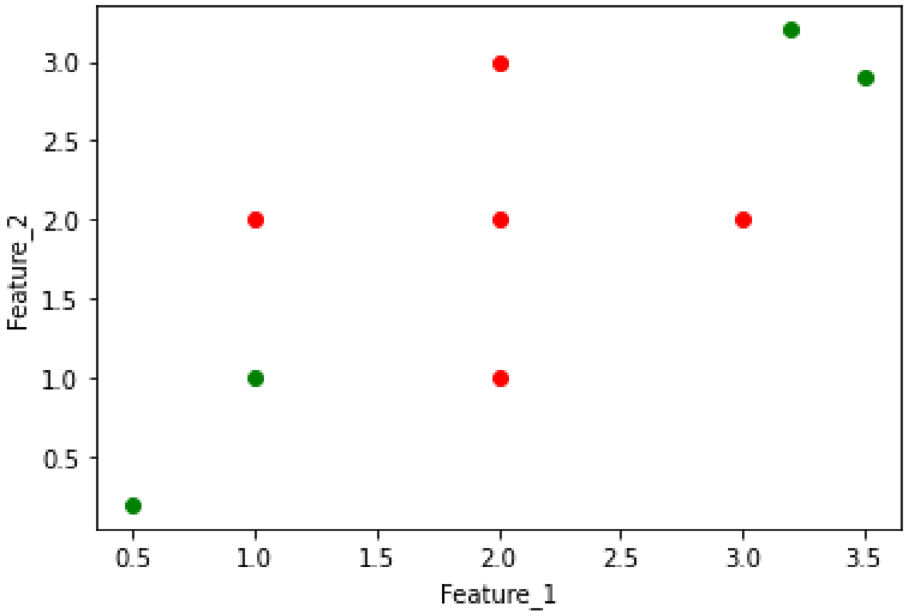
```
x = np.asarray([[2,2], [2,1], [1,2], [2,3], [3,2]])
y = np.asarray([[1,1], [3.2,3.2], [3.5,2.9], [.5,.2]])
print(x[:,0], x[:,1])
fig, ax = plt.subplots(1)
ax.scatter(x[:,0], x[:,1], color='r', label="Class_1")
ax.scatter(y[:,0], y[:,1], color='g', label="Class_2")
```

```python
ax.set_xlabel("Feature_1")
ax.set_ylabel("Feature_2")
```

```
[2 2 1 2 3] [2 1 2 3 2]
```

Out[48]:

```
Text(0, 0.5, 'Feature_2')
```



In [49]:

```
/bin/bash: -c: line 0: syntax error near unexpected token `<'
/bin/bash: -c: line 0: `[title](<your_plot_file_path>)'
```

In [ ]:

This problem was adapted from Professor Farimani's paper. If you are interested in learning more, you can read it here.

In [3]:
```
import sklearn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
import matplotlib
from matplotlib.colors import ListedColormap
import random
from sklearn.ensemble import RandomForestClassifier
```

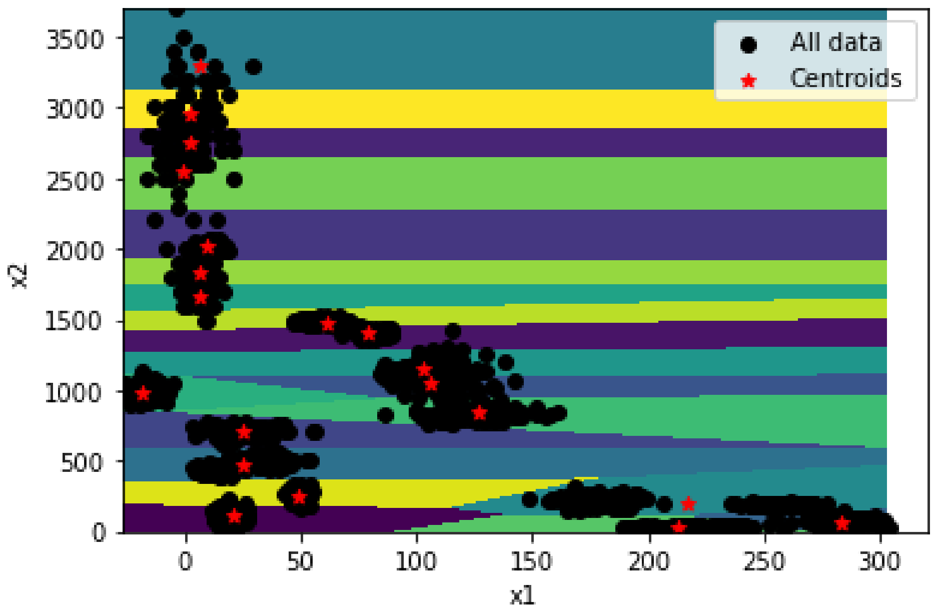
In [4]:
```
data = pd.read_csv("/home/akshay/Downloads/MAIL/HW4/q2-data/data.csv")
```

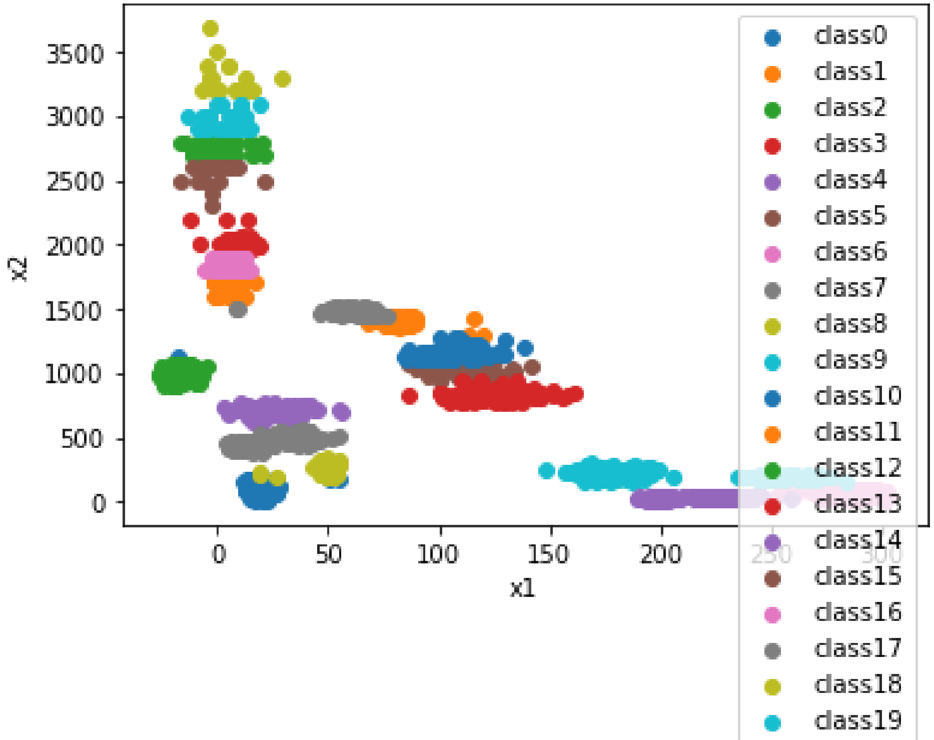
In [5]:
```
# (a)
# data preprocessing
def data_preprocessing():
    data = pd.read_csv("/home/akshay/Downloads/MAIL/HW4/q2-data/data.csv")
    x = np.zeros((0,2), dtype=np.float64)
    y = np.zeros((2000,1), dtype=np.float64)

    for i in range(2,42,2):
        curr_data = data.iloc[:,i-2:i]
        x = np.concatenate([x, np.asarray(curr_data)],axis=0)
        y[int((i-2)*100/2):int((i*100)/2),0] = (i-2)/2

    X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3, )
    return x, y, X_train, X_test, y_train, y_test

x, y, X_train, X_test, y_train, y_test = data_preprocessing()
csv_rows = np.concatenate([x, y], axis=1)
#pd.DataFrame(csv_rows).to_csv("/home/akshay/Downloads/MAIL/HW4/q2_ordered.csv")
```

In [5]:
```
# (b)
# k-means
```

```python
def kmeans(X_train, X_test, y_train, y_test, data):
    #defining kmeans and training the data
    kmeans = KMeans(20)
    kmeans.fit(X_train)

    #predicting clusters of train and test
    train_pred = kmeans.predict(X_train)
    test_pred = kmeans.predict(X_test)
    centroids = kmeans.cluster_centers_

    predicted_classes = []

    #storing the prediction by classes, after predicting the results using the trained kmeans
    #both training and testing are predicted
    for i in range(20):
        class_i_train = X_train[np.where(train_pred == i)]
        class_i_test = X_test[np.where(test_pred == i)]
        class_i = np.concatenate([class_i_train, class_i_test], axis=0)
        predicted_classes.append(class_i)

    # generating a meshgrid
    h = .9
    x_min, x_max = data[:, 0].min(), data[:, 0].max()
    y_min, y_max = data[:, 1].min(), data[:, 1].max()

    x1, y1 = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    xx = x1.reshape(-1,1)
    yy = y1.reshape(-1,1)

    #predicting values of the meshgrid
    mesh_data = np.concatenate([xx,yy],axis = 1)
    predicted_mesh = kmeans.predict(mesh_data)
    predicted_mesh = predicted_mesh.reshape(x1.shape)


    fig, ax = plt.subplots(1)
    #plotting the decision boundaries on all data
    ax.pcolormesh(x1,y1,predicted_mesh, shading='auto')
    ax.scatter(data[:,0], data[:,1], color=(0,0,0), label="All data")
    ax.scatter(centroids[:,0], centroids[:,1], color='r', label="Centroids", marker='*')
```

```
    ax.set_xlabel("x1")
    ax.set_ylabel("x2")
    ax.legend()

    #plotting different classes with different colors according to the assigned kmeans cluster
    i = 0
    fig2, ax2 = plt.subplots(1)
    #plotting the
    for class_ in predicted_classes:
        #rgb = np.random.rand(3,)
        ax2.scatter(class_[:,0], class_[:,1], label="class"+str(i))
        i += 1

    ax2.set_xlabel("x1")
    ax2.set_ylabel("x2")
    ax2.legend()

kmeans(X_train, X_test, y_train, y_test, x)
```



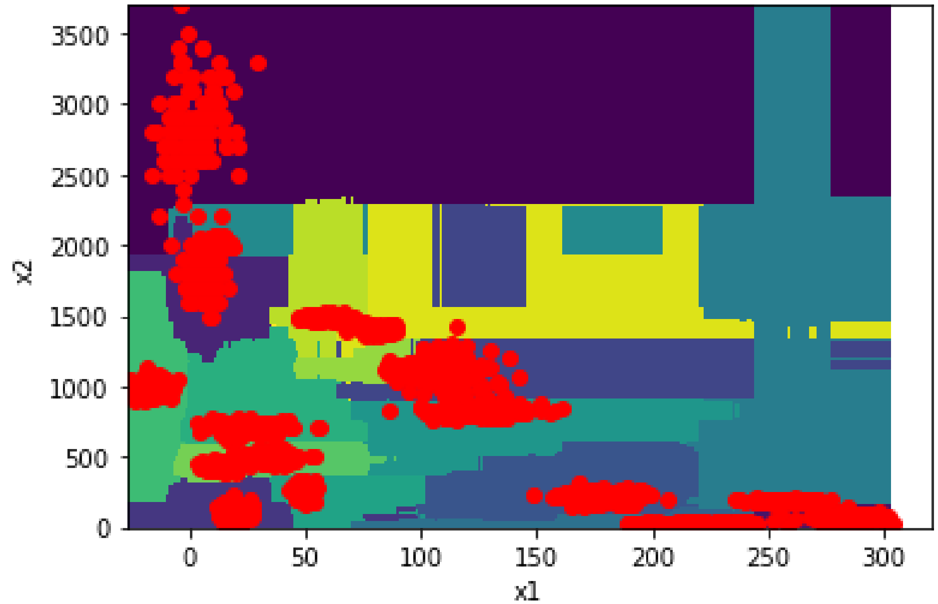


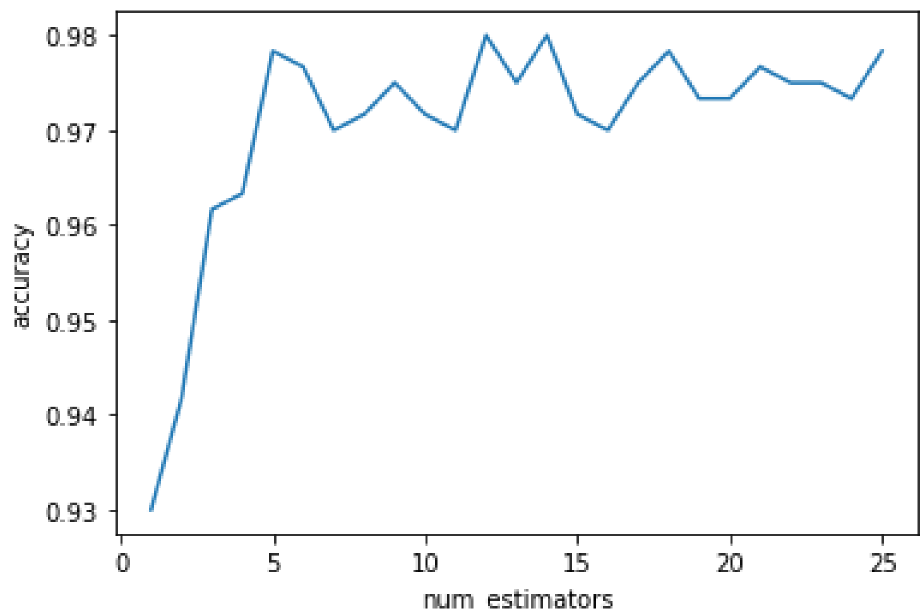
In [6]:

```
# (c)
# random forest
def randomForest(X_train, X_test, y_train, y_test, data):
    randomForest = RandomForestClassifier()
    randomForest.fit(X_train, y_train.squeeze())
    print("train accuracy:", randomForest.score(X_train, y_train.squeeze()))
    print("test accuracy:", randomForest.score(X_test, y_test))

    h = .9
    x_min, x_max = data[:, 0].min(), data[:, 0].max()
    y_min, y_max = data[:, 1].min(), data[:, 1].max()

    x1, y1 = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    xx = x1.reshape(-1,1)
    yy = y1.reshape(-1,1)

    #predicting values of the meshgrid
    mesh_data = np.concatenate([xx,yy],axis = 1)
```

```python
    predicted_mesh = randomForest.predict(mesh_data)
    predicted_mesh = predicted_mesh.reshape(x1.shape)

    fig, ax = plt.subplots(1)
    #plotting the decsision boundaries
    ax.pcolormesh(x1,y1,predicted_mesh, shading='auto')
    ax.scatter(data[:,0], data[:,1], color='r')
    ax.set_xlabel("x1")
    ax.set_ylabel("x2")

randomForest(X_train, X_test, y_train, y_test, x)

def accuracy_vs_estimators(X_train, X_test, y_train, y_test, x):
    score = []
    for i in range(1, 26, 1):
        #append the accuracy on the test dataset to the list
        randomForest = RandomForestClassifier(n_estimators=i)
        randomForest.fit(X_train, y_train.squeeze())
        y_t_sq = y_test.squeeze()
        score.append(randomForest.score(X_test, y_t_sq))

    num_estimators = np.arange(1,26,1)
    fig1, ax2 = plt.subplots(1)
    ax2.plot(num_estimators, score)
    ax2.set_ylabel("accuracy")
    ax2.set_xlabel("num_estimators")

accuracy_vs_estimators(X_train, X_test, y_train, y_test, x)
```

```
train accuracy: 1.0
test accuracy: 0.9783333333333334
```


3500
3000
2500
2000
1500
1000
500
0
x2
0
50
100
150
200
250
300
x1


0.98
0.97
0.96
0.95
0.94
0.93
accuracy
0
5
10
15
20
25
num_estimators

(d)

Analysis

1. The decision boundary for kmeans is piece-wise linear, while the random forest is parallel to x, or y-axis.
2. Random forest gives 100 accuarcy on the training set, while a little less on the test set. We can say that random forest does over-fitting in some cases.
3. The accuracy of random forest is way better than k-means because it uses the labels(supervised learning).
4. And as the number of estimators increase the test accuracy seems to increase for the random forest, because it starts to generalize by using a large number of trees.
5. Large number of trees(num_estimators) makes the random forest algorithm slow, both in time and space complexity

In [ ]:

```

```

In [5]:

```python
#importing all the libraries
import torch
import pandas as pd
import numpy as np
from torch.utils.data import DataLoader, Dataset
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import normalize
```

In [6]:

```python
#inheriting from the torch CLASS
class MyDataset(Dataset):
    def __init__(self, x, y):
        super(MyDataset, self).__init__()
        x = torch.from_numpy(x)
        y = torch.from_numpy(y)
        self.x = x.type(torch.float32)
        self.y = y.type(torch.LongTensor)

    def __getitem__(self, idx):
        sample_data = {'input': self.x[idx], 'output': self.y[idx]}
        return sample_data

    def __len__(self):
        return len(self.x)
```

In [7]:

```python
# creating model with 1 hidden layer
class Model(nn.Module):
    def __init__(self):
        super(Model,self).__init__()
        self.model =  nn.Sequential(nn.Linear(2,64),
                          nn.ReLU(),
                          nn.Linear(64,32),
                          nn.ReLU(),
                          nn.Linear(32,20))
```

```python
    def forward(self, x):
        return self.model(x)
```

In [8]:

```python
# returns the number of correctly predicted outputs
def accuracy(predictions, labels):
    _, predicted = torch.max(predictions, dim=1)
    accuracy = torch.sum(predicted == labels).item()
    return accuracy
```

In [12]:

```python
if __name__ == '__main__':
    #load the ordered data from q2
    data = pd.read_csv("/home/akshay/Downloads/MAIL/HW4/q2_ordered.csv")
    data = data.iloc[:, 1:4]
    data = np.asarray(data)
    x = data[:, 0:2]
    y = data[:, 2]

    #normalize the data
    x = normalize(x, axis=0, norm='max')
    train_X, test_X, train_y, test_y = train_test_split(x, y, test_size=0.2, random_state=4)

    #create the torch dataset
    dataset = MyDataset(train_X, train_y)
    loss_function = nn.CrossEntropyLoss()
    model = Model()
    #using Adam optimizer
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    print("Training model...")
    for i in range(100):
        data_loader = DataLoader(dataset, batch_size=64, shuffle=True, num_workers=8)
        epoch_loss = 0
        no_data = 0
        total_acc = 0
        for batch_data in data_loader:
            input_data, target_data = batch_data['input'], batch_data['output']
            predicted_data = model(input_data)
            loss = loss_function(predicted_data, target_data)
            optimizer.zero_grad()
```

```
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item() * input_data.shape[0]
            no_data += input_data.shape[0]
            total_acc += accuracy(predicted_data, target_data)

        print("Epoch [{}], train_loss: {:.4f}, train_acc: {:.4f}".format(i, epoch_loss/no_data, to

    print("Training Completed...")
```

```
Training model...
Epoch [0], train_loss: 2.9841, train_acc: 5.0625
Epoch [1], train_loss: 2.9359, train_acc: 9.3750
Epoch [2], train_loss: 2.8582, train_acc: 19.6875
Epoch [3], train_loss: 2.7334, train_acc: 16.6875
Epoch [4], train_loss: 2.5637, train_acc: 22.3750
Epoch [5], train_loss: 2.3760, train_acc: 29.9375
Epoch [6], train_loss: 2.1903, train_acc: 39.2500
Epoch [7], train_loss: 2.0129, train_acc: 44.5625
Epoch [8], train_loss: 1.8486, train_acc: 47.1875
Epoch [9], train_loss: 1.7020, train_acc: 48.7500
Epoch [10], train_loss: 1.5729, train_acc: 56.6875
Epoch [11], train_loss: 1.4647, train_acc: 69.8750
Epoch [12], train_loss: 1.3728, train_acc: 63.9375
Epoch [13], train_loss: 1.2944, train_acc: 70.0625
Epoch [14], train_loss: 1.2249, train_acc: 69.4375
Epoch [15], train_loss: 1.1590, train_acc: 82.8125
Epoch [16], train_loss: 1.1043, train_acc: 73.6875
Epoch [17], train_loss: 1.0544, train_acc: 79.6250
Epoch [18], train_loss: 1.0084, train_acc: 82.4375
Epoch [19], train_loss: 0.9655, train_acc: 82.1250
Epoch [20], train_loss: 0.9241, train_acc: 86.1250
Epoch [21], train_loss: 0.8905, train_acc: 80.7500
Epoch [22], train_loss: 0.8628, train_acc: 79.6250
Epoch [23], train_loss: 0.8223, train_acc: 84.3750
Epoch [24], train_loss: 0.7901, train_acc: 88.6250
Epoch [25], train_loss: 0.7631, train_acc: 88.1250
Epoch [26], train_loss: 0.7348, train_acc: 91.0625
Epoch [27], train_loss: 0.7108, train_acc: 87.6875
Epoch [28], train_loss: 0.6828, train_acc: 90.1875
Epoch [29], train_loss: 0.6617, train_acc: 90.0625
Epoch [30], train_loss: 0.6382, train_acc: 91.9375
```

```
Epoch [31], train_loss: 0.6198, train_acc: 91.2500
Epoch [32], train_loss: 0.5980, train_acc: 92.0625
Epoch [33], train_loss: 0.5796, train_acc: 92.2500
Epoch [34], train_loss: 0.5590, train_acc: 94.2500
Epoch [35], train_loss: 0.5458, train_acc: 92.6250
Epoch [36], train_loss: 0.5260, train_acc: 93.8750
Epoch [37], train_loss: 0.5105, train_acc: 91.6250
Epoch [38], train_loss: 0.5000, train_acc: 91.9375
Epoch [39], train_loss: 0.4836, train_acc: 93.1250
Epoch [40], train_loss: 0.4710, train_acc: 91.6250
Epoch [41], train_loss: 0.4563, train_acc: 93.5625
Epoch [42], train_loss: 0.4417, train_acc: 94.8125
Epoch [43], train_loss: 0.4288, train_acc: 93.4375
Epoch [44], train_loss: 0.4203, train_acc: 93.5625
Epoch [45], train_loss: 0.4119, train_acc: 94.0625
Epoch [46], train_loss: 0.4015, train_acc: 93.7500
Epoch [47], train_loss: 0.3910, train_acc: 94.3125
Epoch [48], train_loss: 0.3810, train_acc: 94.2500
Epoch [49], train_loss: 0.3768, train_acc: 93.3750
Epoch [50], train_loss: 0.3640, train_acc: 94.9375
Epoch [51], train_loss: 0.3521, train_acc: 94.8750
Epoch [52], train_loss: 0.3492, train_acc: 94.1250
Epoch [53], train_loss: 0.3451, train_acc: 94.5000
Epoch [54], train_loss: 0.3333, train_acc: 94.3125
Epoch [55], train_loss: 0.3258, train_acc: 94.1875
Epoch [56], train_loss: 0.3203, train_acc: 95.0000
Epoch [57], train_loss: 0.3177, train_acc: 94.8125
Epoch [58], train_loss: 0.3062, train_acc: 95.3125
Epoch [59], train_loss: 0.3024, train_acc: 94.5000
Epoch [60], train_loss: 0.2937, train_acc: 95.2500
Epoch [61], train_loss: 0.2902, train_acc: 94.8750
Epoch [62], train_loss: 0.2881, train_acc: 94.5625
Epoch [63], train_loss: 0.2870, train_acc: 95.3125
Epoch [64], train_loss: 0.2766, train_acc: 95.6875
Epoch [65], train_loss: 0.2700, train_acc: 95.1250
Epoch [66], train_loss: 0.2678, train_acc: 95.0625
Epoch [67], train_loss: 0.2654, train_acc: 95.0625
Epoch [68], train_loss: 0.2608, train_acc: 95.1875
Epoch [69], train_loss: 0.2609, train_acc: 94.8125
Epoch [70], train_loss: 0.2543, train_acc: 94.7500
Epoch [71], train_loss: 0.2495, train_acc: 95.1250
Epoch [72], train_loss: 0.2474, train_acc: 94.9375
```

```
Epoch [73], train_loss: 0.2445, train_acc: 95.2500
Epoch [74], train_loss: 0.2395, train_acc: 95.1875
Epoch [75], train_loss: 0.2355, train_acc: 95.3750
Epoch [76], train_loss: 0.2361, train_acc: 94.7500
Epoch [77], train_loss: 0.2316, train_acc: 94.8750
Epoch [78], train_loss: 0.2254, train_acc: 95.3125
Epoch [79], train_loss: 0.2236, train_acc: 95.4375
Epoch [80], train_loss: 0.2214, train_acc: 95.4375
Epoch [81], train_loss: 0.2197, train_acc: 95.8125
Epoch [82], train_loss: 0.2165, train_acc: 95.5625
Epoch [83], train_loss: 0.2114, train_acc: 95.6250
Epoch [84], train_loss: 0.2134, train_acc: 95.0000
Epoch [85], train_loss: 0.2072, train_acc: 95.6250
Epoch [86], train_loss: 0.2055, train_acc: 95.5625
Epoch [87], train_loss: 0.2070, train_acc: 95.0625
Epoch [88], train_loss: 0.2026, train_acc: 95.8125
Epoch [89], train_loss: 0.1997, train_acc: 95.8750
Epoch [90], train_loss: 0.2029, train_acc: 95.1250
Epoch [91], train_loss: 0.1990, train_acc: 95.6875
Epoch [92], train_loss: 0.1932, train_acc: 95.8125
Epoch [93], train_loss: 0.1953, train_acc: 95.1875
Epoch [94], train_loss: 0.1918, train_acc: 96.1250
Epoch [95], train_loss: 0.1893, train_acc: 95.9375
Epoch [96], train_loss: 0.1864, train_acc: 95.5000
Epoch [97], train_loss: 0.1857, train_acc: 95.4375
Epoch [98], train_loss: 0.1865, train_acc: 95.5625
Epoch [99], train_loss: 0.1856, train_acc: 95.3125
Training Completed...
```

In [13]:

```
#testing
print("Testing the model...")
dataset = MyDataset(test_X, test_y)
model.eval()

test_dataloader = DataLoader(dataset, batch_size=128, shuffle=True, num_workers=8)
accurate_pred = 0
total_no = 0

for batch_data in test_dataloader:
    input_data, target_data = batch_data['input'], batch_data['output']
    predicted_data = model(input_data)
    accurate_pred += accuracy(predicted_data, target_data)
```

```
        total_no += input_data.shape[0]

    print("Test_Acc: {:.4f}".format(accurate_pred/total_no))
```

```
Testing the model...
Test_Acc: 0.9475
```

HW-4
Assignment Report

Akshay Antony
akshayan@andrew.cmu.edu

Q1 :
b:

- Eigenvalues obtained
- 4.71136968e+00
- 2.28054740e+00
- 7.71731109e-01
- 2.02811748e-01
- 3.35400649e-02
- 5.53596026e-16
- 3.03086151e-16
- -6.82293804e-16

c:

- k: 1, Eigen Values: [4.71136968]
- k: 2, Eigen Values: [4.71136968, 2.2805474]
- k: 3, Eigen Values: [4.71136968, 2.2805474, 0.77173111]
- k: 4, Eigen Values: [4.71136968, 2.2805474, 0.77173111, 0.20281175]
- k: 5, Eigen Values: [4.71136968, 2.2805474, 0.77173111, 0.20281175, 0.03354006]
- k: 6, Eigen Values: [4.71136968, 2.2805474, 0.77173110, 0. 202811748, 3.35400649e-02, 5.53596026e-16]
- k: 7, Eigen Values: [4.71136968, 2.28054740, 7.71731109e-01, 2.02811748e-01
  3.35400649e-02, 5.53596026e-16, 3.03086151e-16]
- k: 8, Eigen Values: [4.71136968, 2.28054740, 7.71731109e-01, 2.02811748e-01
  3.35400649e-02, 5.53596026e-16, 3.03086151e-16, 6.82293804e-16]

d:

- I will choose k=3 because the ratio of the sum of eigenvalues to the total eigenvalues is 97%, which will store 97% of the variation in the data.

Q4

Q.4) All calculations are shown In a Jupyter-NB.

Steps

1) Did forward propagation by adding a column of value 1, to the start of input ~~to~~ which is equivalent to bias

2) do $\left[\text{alpha} @ (\text{Input}^T)\right] = a$

3) $z = \text{sigmoid}(a)$, value of 1 is inserted at first position of z. {bias}

4) $b = [\text{beta} @ z]$

5) $y = \text{softmax}(b)$

6) $\text{Loss} = -y_2 \times \log(\hat{y}_2) = \varepsilon$

7) Back propagation.

eg:- $\frac{\partial \varepsilon}{\partial \beta_{2,1}} = \frac{\partial \varepsilon}{\partial \hat{y}_2} \times \frac{\partial \hat{y}_2}{\partial b_2} \times \frac{\partial b_2}{\partial z_1}$

This made to a matrix form

$$\left[\frac{\partial \varepsilon}{\partial \beta_{i,j}}\right] = \underset{1\times 1}{\left[\frac{\partial \varepsilon}{\partial \hat{y}_2}\right]} \times \underset{3\times 1}{\begin{bmatrix} \partial \hat{y}_2/\partial b_1 \\ \partial \hat{y}_2/\partial b_2 \\ \partial \hat{y}_2/\partial b_3 \end{bmatrix}} \times \underset{3\times 5}{\begin{bmatrix} 1 & z_1 & z_2 & z_3 & z_4 \\ 1 & z_1 & z_2 & z_3 & z_4 \\ 1 & z_1 & z_2 & z_3 & z_4 \end{bmatrix}}$$

first column = bias

8) update $\beta_{i,j} = \beta_{i,j} - \eta \times \frac{\partial \varepsilon}{\partial \beta_{i,j}}$

9) For $\frac{\partial \varepsilon}{\partial \alpha_{1,1}} = \frac{\partial \varepsilon}{\partial \hat{y}_2} \times \left(\frac{\partial \hat{y}_2}{\partial b_i} \times \frac{\partial b_i}{\partial z_1}\right) \frac{\partial z_1}{\partial a_1} \times \frac{\partial a_1}{\partial \alpha_{1,1}}$ ⇒ derivative of sigmoid

$$\left\{\frac{\partial \hat{y}_2}{\partial b_1} \times \frac{\partial b_1}{\partial z_1} + \frac{\partial \hat{y}_2}{\partial b_2} \times \frac{\partial b_2}{\partial z_1} + \frac{\partial \hat{y}_2}{\partial b_3} \times \frac{\partial b_3}{\partial z_1}\right\}$$

$$\left[\frac{\partial \varepsilon}{\partial \alpha_{i,j}}\right] = \underset{1\times 1}{\left[\partial \varepsilon/\partial \hat{y}_2\right]} \times \underset{4\times 1}{\begin{bmatrix} \sum_{i=1}^{3} \frac{\partial \hat{y}_2}{\partial b_i} \times \frac{\partial b_i}{\partial z_1} \\ \sum_{i=1}^{3} \frac{\partial \hat{y}_2}{\partial b_i} \times \frac{\partial b_i}{\partial z_2} \\ \sum_{i=1}^{3} \frac{\partial \hat{y}_2}{\partial b_i} \times \frac{\partial b_i}{\partial z_3} \\ \sum_{i=1}^{3} \frac{\partial \hat{y}_2}{\partial b_i} \times \frac{\partial b_i}{\partial z_4} \end{bmatrix}} \times \underset{4\times 1}{\begin{bmatrix} \partial z_1/\partial a_1 \\ \partial z_2/\partial a_2 \\ \partial z_3/\partial a_3 \\ \partial z_4/\partial a_4 \end{bmatrix}} \times \underset{4\times 7}{\begin{bmatrix} 1 & x_1 & x_2 & x_3 & x_4 \\ '' & '' \\ '' & '' \\ '' & '' \end{bmatrix}}$$

(1×1)

Final Answers.

1) a) $a_1^{(1)} = 2$, $z_1^{(1)} = 0.8808$

1) b) $a_3^{(1)} = 8$, $z_3^{(1)} = 0.9997$

1) c) $b_2^{(1)} = 3.643$,

d) $\hat{y}_2 = 0.2615$

e) I will predict it to class $\boxed{3}$

f) 1.3412

2) a) 1.6504

b) 0.3918

c) 2.0

d) .9986

e) Class 2, with softmax value = 1.000

JUPYTER NOTEBOOK PHOTO ON NEXT PAGE

In [1]:
```
import numpy as np
```

1. Defining the weight matrix, where each of the first columns are biases=1.
2. Input matrix is also appended by 1 at the start to show the bias.

In [2]:
```
inp = np.asarray([1, 1, 1, 0, 0, 1, 1], dtype=np.float32)
alpha = np.asarray([[1, 1, 2, -3, 0, 1, -3],
                    [1, 3, 1, 2, 1, 0, 2],
                    [1, 2, 2, 2, 2, 2, 1],
                    [1, 1, 0, 2, 1, -2, 2]], dtype = np.float32)
beta = np.asarray([[1, 1, 2, -2, 1],
                   [1, 1, -1, 1, 2],
                   [1, 3, 1, -1, 1]], dtype = np.float32)
```

In [3]:
```
a = alpha @ inp.T
a
```

Out[3]:
```
array([2., 7., 8., 2.], dtype=float32)
```

In [4]:
```
z = 1. / (1 + np.exp(-a))
z
z = np.insert(z, 0, 1)
z
```

Out[4]:
```
array([1.        , 0.880797  , 0.999089  , 0.99966466, 0.880797  ],
      dtype=float32)
```

In [5]:
```
b = beta @ z.T
b
```

Out[5]:
```
array([2.7604427, 3.6429667, 4.5226126], dtype=float32)
```

In [6]:
```
y = np.exp(b) / np.sum(np.exp(b))
```

```
y
```

Out[6]: `array([0.10820103, 0.26152113, 0.6302779 ], dtype=float32)`

1. From the above arrays we can see that the predicted class is 3.
2. The answers are reported separately

In [7]:

```
loss = - np.log(y[1])
loss
```

Out[7]: `1.3412402`

$$\frac{\partial loss}{\partial y} * \frac{\partial y}{\partial b} * \frac{\partial b}{\partial \beta} = \frac{\partial loss}{\partial \beta} \tag{1}$$

\ db_dbeta represents the derivative of b w.r.t beta matrix. For the sake of matrix manipulation it is repeated 3 times to make 3 rows \ dy_db represensts derivative of y w.r.t to b, which is the derivative of the softmax \ dloss_dy represents derivative of total loss w.r.t to y, Only 1 derivative is non-zero rest are zero.

In [8]:

```
db_dbeta = np.asarray([[1,0.880797,.990889,.999665,.880797],
                       [1,0.880797,.990889,.999665,.880797],
                       [1,0.880797,.990889,.999665,.880797]])

dy_db = np.asarray([[-.028297],
                    [.193128],
                    [-.164831]])

dloss_dy = np.asarray([[-3.823785]])

dloss_dbeta = db_dbeta * dy_db * dloss_dy
dloss_dbeta
```

Out[8]:

```
array([[ 0.10820164,  0.09530368,  0.10721582,  0.1081654 ,  0.09530368],
       [-0.73847995, -0.65045092, -0.73175166, -0.73823256, -0.65045092],
       [ 0.63027831,  0.55514724,  0.62453584,  0.63006716,  0.55514724]])
```

dloss_dbeta represents: $\partial loss/\partial\beta$ \ by gradient descent we change β -= $\partial loss/\partial\beta$ * lr \ lr = 1

In [9]:
```
beta -= dloss_dbeta
beta
```

Out[9]:
```
array([[ 0.8917984 ,  0.90469635,  1.8927842 , -2.1081655 ,  0.90469635],
       [ 1.73848   ,  1.650451  , -0.26824835,  1.7382326 ,  2.650451  ],
       [ 0.36972168,  2.4448528 ,  0.37546417, -1.6300671 ,  0.44485277]],
      dtype=float32)
```

To find derivatives w.r.t alpha \

$$\frac{\partial loss}{\partial y} * \frac{\partial y}{\partial b} * \frac{\partial b}{\partial z} * \frac{\partial z}{\partial a} * \frac{\partial a}{\partial \alpha} = \frac{\partial loss}{\partial \alpha} \tag{2}$$

\ calculates this value (∂y/∂b)∗(∂b/∂z) for z1, z2, z3, z4 by using appropriate b's, and is made into the matrix dy_db_dz

In [10]:
```
dy_db_dz1 = np.asarray([-y[0]* y[1]*1, y[1]*(1-y[1])*1, -y[1]*y[2]*3])
dy_db_dz2 = np.asarray([-y[0]* y[1]*2, y[1]*(1-y[1])*-1, -y[1]*y[2]*1])
dy_db_dz3 = np.asarray([-y[0]* y[1]*-2, y[1]*(1-y[1])*1, -y[1]*y[2]*-1])
dy_db_dz4 = np.asarray([-y[0]* y[1]*1, y[1]*(1-y[1])*2, -y[1]*y[2]*1])
dy_db_dz = np.asarray([[np.sum(dy_db_dz1)], [np.sum(dy_db_dz2)], [np.sum(dy_db_dz3)], [np.sum(dy_d
dy_db_dz
```

Out[10]:
```
array([[-0.32966198],
       [-0.41455252],
       [ 0.41455252],
       [ 0.19312782]])
```

Calculating (∂z/∂a), which is the derivative of sigmoid for z1, z2, z3, z4 wrt corresponding a's and is made into a 4*1, matrix dz_da

In [11]:
```
dz_da = np.asarray([[z[1]*(1-z[1])], [z[2]*(1-z[2])], [z[3]*(1-z[3])], [z[4]*(1-z[4])]])
dz_da
```

Out[11]:
```
array([[0.10499362],
       [0.00091017],
```

```
       [0.00033522],
       [0.10499362]])
```

To get (∂a/∂alpha) the input matrix is repeated 4 times. And all the derivative matrices are multiplied

In [12]:
```
da_dalpha = np.asarray([[1, 1, 1, 0, 0, 1, 1],
                        [1, 1, 1, 0, 0, 1, 1],
                        [1, 1, 1, 0, 0, 1, 1],
                        [1, 1, 1, 0, 0, 1, 1]], dtype = np.float32)

dloss_dalpha = (dy_db_dz * dz_da) * da_dalpha * dloss_dy
dloss_dalpha
```

Out[12]:
```
array([[ 0.1323504 ,  0.1323504 ,  0.1323504 ,  0.        ,  0.        ,
         0.1323504 ,  0.1323504 ],
       [ 0.00144276,  0.00144276,  0.00144276,  0.        ,  0.        ,
         0.00144276,  0.00144276],
       [-0.00053138, -0.00053138, -0.00053138, -0.        , -0.        ,
        -0.00053138, -0.00053138],
       [-0.07753561, -0.07753561, -0.07753561, -0.        , -0.        ,
        -0.07753561, -0.07753561]])
```

Doing the gradient descent on alpha and updating

In [13]:
```
alpha -= dloss_dalpha
alpha
```

Out[13]:
```
array([[ 8.6764961e-01,  8.6764961e-01,  1.8676496e+00, -3.0000000e+00,
         0.0000000e+00,  8.6764961e-01, -3.1323504e+00],
       [ 9.9855721e-01,  2.9985573e+00,  9.9855721e-01,  2.0000000e+00,
         1.0000000e+00, -1.4427608e-03,  1.9985572e+00],
       [ 1.0005314e+00,  2.0005314e+00,  2.0005314e+00,  2.0000000e+00,
         2.0000000e+00,  2.0005314e+00,  1.0005314e+00],
       [ 1.0775356e+00,  1.0775356e+00,  7.7535614e-02,  2.0000000e+00,
         1.0000000e+00, -1.9224644e+00,  2.0775356e+00]], dtype=float32)
```

Final Prediction which is 2.

In [14]:
```
a = (alpha @ inp.T)
z = a / (1 + np.exp(-a))
z = np.insert(z, 0, 1)
```

```
b = beta @ z.T
y = np.exp(b) / np.sum(np.exp(b))
y
```

Out[14]:
```
array([6.669451e-10, 1.000000e+00, 8.454082e-13], dtype=float32)
```

In [ ]: