Statistical Computer Vision (ECSE 626)

Assignment-1
Information Theory and Face Recognition

Raghav Mehta
260787488

Q1

February 25, 2019

1 1. Information Theory

```
In [1]: !ls

Color_FERET_Database                   image_folder  Q2.ipynb
ECSE626_assignment1_questions.pdf  Q1.ipynb
```

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        import os
        from scipy import signal
        import scipy

In [3]: np.random.seed(0)

In [4]: folder_path = "image_folder/"
```

1.1 1.1 Entropy of Retina Image

1.1.1 Entropy Helper function

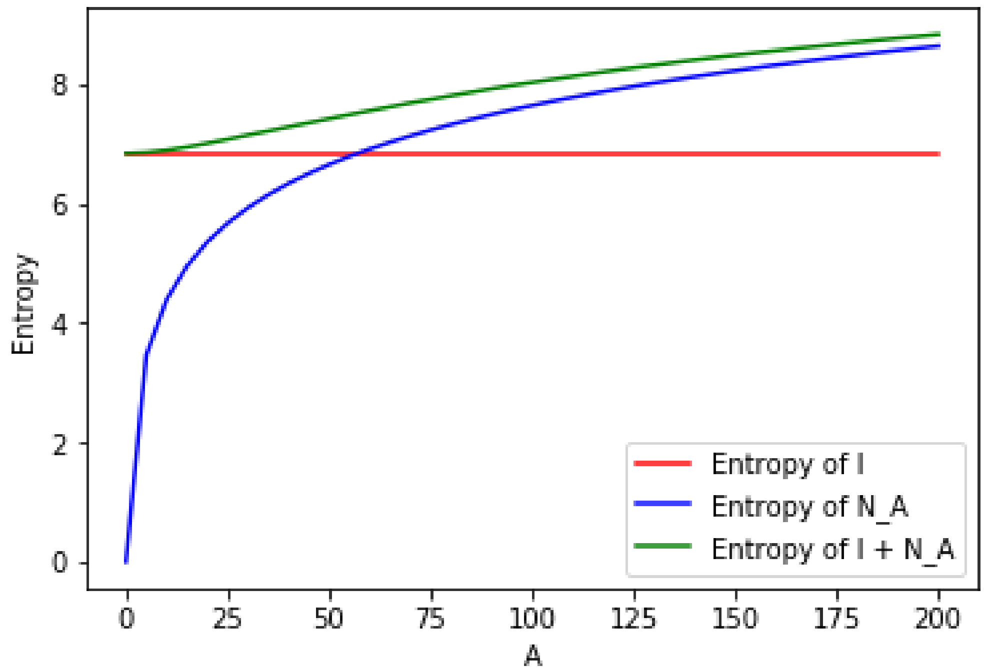
```
In [5]: def entropy_image(I):
            counts = np.histogram(I, bins=I.max()-I.min()+1)[0]
            norm_counts = counts / counts.sum()
            entr = - (norm_counts[norm_counts>0] * np.log2(norm_counts[norm_counts>0])).sum()
            return entr
```

1.1.2 1.1.1 Entropy of the Retina Image

```
In [6]: I = mpimg.imread(os.path.join(folder_path+'img_retina.jpg'))          # read image

In [7]: print("Entropy of the Retina Image I:", entropy_image(I))

Entropy of the Retina Image I: 6.847117961729753
```

1.1.3 1.1.2 Generate Noise Image N_A = U[-A, A]. Vary A from 0 to 200 in steps of 5 and generate different instances of N_A. Compute and plot in a single graph the entropy of: I, N_A, I + N_A as a function of A.

```
In [8]: min_A = 0          # minimum value of A
        max_A = 200        # maximum value of A
        step = 5           # step of increment of A
        entropy_I = []     # list to store entropy of I
        entropy_Na = []    # list to store entropy of N_A
        entropy_I_Na = []  # list to store entropy of I + N_A
        A = []             # list to store values of A

In [9]: for i in range(min_A,max_A+1,step):
            A.append(i)                               # store value of A
            N_A = np.random.randint(-i, i+1, I.shape) # generate N_A, note that +1 is necessary as maximum value is not inclusive in numpy
            entropy_I.append(entropy_image(I))        # calculate and store entropy of I
            entropy_Na.append(entropy_image(N_A))     # calculate and store entropy of N_A
            entropy_I_Na.append(entropy_image(I+N_A)) # calculate and store entropy of I + N_A

In [10]: # convert lists into array
         A = np.array(A)
         entropy_I = np.array(entropy_I)
         entropy_Na = np.array(entropy_Na)
         entropy_I_Na = np.array(entropy_I_Na)

In [11]: plt.plot(A, entropy_I, 'r', label='Entropy of I')
         plt.plot(A, entropy_Na, 'b', label='Entropy of N_A')
         plt.plot(A, entropy_I_Na, 'g', label='Entropy of I + N_A')
         plt.xlabel('A')
         plt.ylabel('Entropy')
         plt.legend()
         plt.show()
```



From this we can observe that entropy of Imaage (I) stays constant with increase in A. This is expected as we are I is unaffected by A.
Entropy of N_A increases with increase in value of A. But this relationship is not increase.
Similarly, we can also observe that with increase in value of A, entropy of I + N_a also increases. But this relationship is also not linear. Entropy of I+N_a is always greater than the entropy of I or entropy of N_a, but it is not a direct sum of them.

1.2 1.2 Mutual Information and KL divergence

1.2.1 Mututal Information helper function

```
In [12]: def MI_AB(A,B):

             epsi = 10e-25

             # calculate joint histogram of A and B
             histAB = np.histogram2d(A.ravel(), B.ravel(), bins=[A.max()-A.min()+1, B.max()-B.min()+1])[0]
             norm_histAB = histAB / histAB.sum()

             # calculate individual histogram of A and B
             histA  = np.histogram(A.ravel(), bins=A.max()-A.min()+1)[0]
             norm_histA = histA / histA.sum()
             histB  = np.histogram(B.ravel(), bins=B.max()-B.min()+1)[0]
             norm_histB = histB / histB.sum()

             # repeat norm_histA and norm_histB for faster computation
             norm_histA_rep = np.repeat(norm_histA[:,np.newaxis],len(norm_histB),axis=-1)
             norm_histB_rep = np.repeat(norm_histB[:,np.newaxis],len(norm_histA),axis=-1).T

             # calculate MI

             MI = ((norm_histAB[(norm_histAB>0) & (norm_histA_rep>0) & (norm_histB_rep>0)]) * np.log2( (norm_histAB[(norm_histAB>0) & (norm_hi

             return MI
```

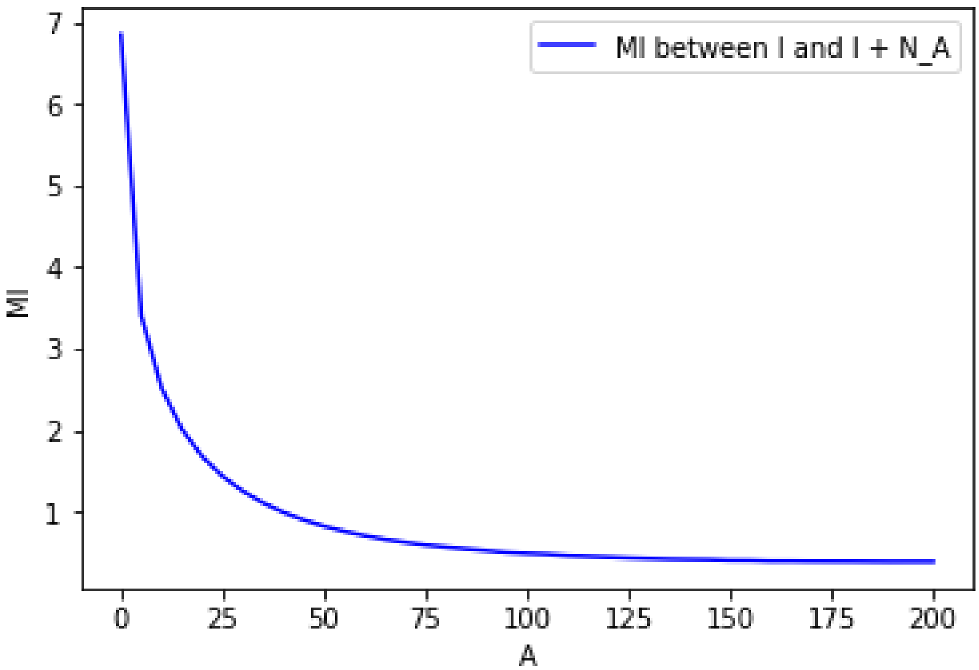
1.2.2 1.2.1 Generate Noise Image N_A, by varying A from 0 to 200 in the steps of 5. Compute and plot the mutual information between I and I + N_A as a function of A.

```
In [13]: min_A = 0          # minimum value of A
         max_A = 200        # maximum value of A
         step = 5           # step of increment of A
         MI_I_I_Na = []     # list to store MI between I and I + N_A
         A = []             # list to store values of A
```

```
In [14]: for i in range(min_A,max_A+1,step):
             A.append(i)                            # store value of A
             N_A = np.random.randint(-i, i+1, I.shape) # generate N_A, note that +1 is necessary as maximum value is not inclusive in numpy
             MI_I_I_Na.append(MI_AB(I, I+N_A))      # calculate and store MI between I and I + N_A
```

In [15]:
```
# convert lists into array
A = np.array(A)
MI_I_I_Na = np.array(MI_I_I_Na)
```

In [16]:
```
plt.plot(A, MI_I_I_Na, 'b', label='MI between I and I + N_A')
plt.xlabel('A')
plt.ylabel('MI')
plt.legend()
plt.show()
```



From the above graph we can see that MI between I and I+N_A decreases with increase in value of A. This shows that as the randomness in the noise increases, MI decreases.

1.2.3 1.2.2 Generate single a single noise image, N_20. Compute the joint entropy of the image pair: H(I, I+N_20). Also verify numerically that: H(I; I+N_A) = H(I) + H(I+N_20) - MI(I; I+N_20)

1.2.4 Joint Entropy helper function

In [17]:
```
def Joint_Entropy(A,B):

    epsi = 10e-25

    # calculate joint histogram of A and B
    histAB = np.histogram2d(A.ravel(), B.ravel(), bins=[A.max()-A.min()+1, B.max()-B.min()+1])[0]
    norm_histAB = histAB / histAB.sum()

    # calculate Joint Entropy
    JE = - (norm_histAB[norm_histAB>0] * np.log2(norm_histAB[norm_histAB>0])).sum()

    return JE
```

In [18]:
```
N_20 = np.random.randint(-20,21, I.shape) # generate N_20, note that +1 is necessary as maximum value is not inclusive in numpy
```

In [19]:
```
Joint_Entropy_I_I_N20 = Joint_Entropy(I, I+N_20) # Joint entropy of I and I + N_20
Entropy_I = entropy_image(I)
Entropy_I_N20 = entropy_image(I+N_20)
MI_I_I_N20 = MI_AB(I, I+N_20)
```

In [20]:
```
print("H(I; I+N20): ", Joint_Entropy_I_I_N20)
print("H(I) + H(I+N20) - MI(I; I+N20): ", Entropy_I + Entropy_I_N20 - MI_I_I_N20)
print("Difference between above two: ", Entropy_I + Entropy_I_N20 - MI_I_I_N20 - Joint_Entropy_I_I_N20)
```

```
H(I; I+N20):  12.182877741459238
H(I) + H(I+N20) - MI(I; I+N20):  12.182877741459237
Difference between above two:  -1.7763568394002505e-15
```

From above, we can see that difference between H(I; I+N20) and H(I) + H(I+N20) - MI(I;I+N20) is almost zero. Thus we varified that they are equivalent.

1.2.5 1.2.3 Compute forward and backward KL divergence between

1.2.6 KL Divergence helper function

In [21]:
```
def parzen_window_density(hist, window_size):
    window = signal.parzen(window_size)
    pwd = np.convolve(hist, window, 'same')
    return pwd
```

In [22]:
```
def KL_divergence(A,B,window_size=5, parzen_window=False, apply_to_A=True, apply_to_B=False):

    epsi = 10e-25

    # calculate range of combined A and B
    total_min = min(A.min(), B.min())
    total_max = max(A.max(), B.max())

    # calculate histA and histB
    histA = np.histogram(A.ravel(), bins=total_max-total_min+1, range=[total_min, total_max])[0]
    if parzen_window and apply_to_A:
        histA = parzen_window_density(histA, window_size)
    norm_histA = histA / histA.sum()

    histB = np.histogram(B.ravel(), bins=total_max-total_min+1, range=[total_min, total_max])[0]
    if parzen_window and apply_to_B:
        histB = parzen_window_density(histB, window_size)
    norm_histB = histB / histB.sum()

    # calculate KL Divergence
    KLD = - (norm_histA[(norm_histA>0) & (norm_histB>0) ] * np.log2( (norm_histB[(norm_histA>0) & (norm_histB>0) ]) / (norm_histA[(n

    return KLD
```

1.2.7 (i) I and the noise N of the same size as I; where pixel intensities of N are drawn from U[0,255]

In [23]:
```
N_U = np.random.randint(0,256,I.shape)
```

In [24]:
```
F_KLD = KL_divergence(I, N_U)
B_KLD = KL_divergence(N_U, I)
print("Forward KL Divergence between I and N_U: ", F_KLD)
print("Backward KL Divergence between I and N_U: ", B_KLD)
```

```
Forward KL Divergence between I and N_U:  1.1519461622825244
Backward KL Divergence between I and N_U:  2.164815111442786
```

From the above values we can say that, I is able to approximate N_U in a better manner than N_U is able to approximate I, as value of Forward KL divergence is smaller compare to Backward KL divergence.

1.2.8 (ii) I and I+N20 from the previous question. Use parzen window filtering on the histogram of I.

In [25]:
```
N_20 = np.random.randint(-20,21,I.shape)
```

No parzen window

In [26]:
```
F_KLD = KL_divergence(I, I+N_20)
B_KLD = KL_divergence(I+N_20, I)
print("Forward KL Divergence between I and N_U: ", F_KLD)
print("Backward KL Divergence between I and N_U: ", B_KLD)
```

```
Forward KL Divergence between I and N_U:  0.034606133956389784
Backward KL Divergence between I and N_U:  0.04133233034511477
```

Parzen window only on I

In [27]:
```
window_size=11
F_KLD = KL_divergence(I, I+N_20, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=False)
B_KLD = KL_divergence(I+N_20, I, window_size=window_size, parzen_window=True, apply_to_A=False, apply_to_B=True)
print("Forward KL Divergence between I and N_U: {}, window_size: {}".format(F_KLD, window_size))
print("Backward KL Divergence between I and N_U: {}, window_size:{}".format(B_KLD,window_size))
```

```
Forward KL Divergence between I and N_U: 0.03237226764984605, window_size: 11
Backward KL Divergence between I and N_U: 0.039395430815245026, window_size:11
```

In [28]:
```
window_size=7
F_KLD = KL_divergence(I, I+N_20, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=False)
B_KLD = KL_divergence(I+N_20, I, window_size=window_size, parzen_window=True, apply_to_A=False, apply_to_B=True)
print("Forward KL Divergence between I and N_U: {}, window_size: {}".format(F_KLD, window_size))
print("Backward KL Divergence between I and N_U: {}, window_size:{}".format(B_KLD,window_size))
```

```
Forward KL Divergence between I and N_U: 0.03337029723344952, window_size: 7
Backward KL Divergence between I and N_U: 0.04079202147553072, window_size:7
```

In [29]:
```
window_size=3
F_KLD = KL_divergence(I, I+N_20, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=False)
B_KLD = KL_divergence(I+N_20, I, window_size=window_size, parzen_window=True, apply_to_A=False, apply_to_B=True)
print("Forward KL Divergence between I and N_U: {}, window_size: {}".format(F_KLD, window_size))
print("Backward KL Divergence between I and N_U: {}, window_size:{}".format(B_KLD,window_size))
```

```
Forward KL Divergence between I and N_U: 0.03433063139989136, window_size: 3
Backward KL Divergence between I and N_U: 0.04224336539981752, window_size:3
```

In [30]:
```
window_size=1
F_KLD = KL_divergence(I, I+N_20, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=False)
B_KLD = KL_divergence(I+N_20, I, window_size=window_size, parzen_window=True, apply_to_A=False, apply_to_B=True)
print("Forward KL Divergence between I and N_U: {}, window_size: {}".format(F_KLD, window_size))
print("Backward KL Divergence between I and N_U: {}, window_size:{}".format(B_KLD,window_size))
```

```
Forward KL Divergence between I and N_U: 0.034606133956389784, window_size: 1
Backward KL Divergence between I and N_U: 0.04133233034511477, window_size:1
```

Parzen window on both I and I+N_20

In [31]:
```
window_size=11
F_KLD = KL_divergence(I, I+N_20, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=True)
B_KLD = KL_divergence(I+N_20, I, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=True)
print("Forward KL Divergence between I and N_U: {}, window_size: {}".format(F_KLD, window_size))
print("Backward KL Divergence between I and N_U: {}, window_size:{}".format(B_KLD,window_size))
```

```
Forward KL Divergence between I and N_U: 0.03271314602735115, window_size: 11
Backward KL Divergence between I and N_U: 0.03973289541762002, window_size:11
```

In [32]:
```
window_size=7
F_KLD = KL_divergence(I, I+N_20, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=True)
B_KLD = KL_divergence(I+N_20, I, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=True)
print("Forward KL Divergence between I and N_U: {}, window_size: {}".format(F_KLD, window_size))
print("Backward KL Divergence between I and N_U: {}, window_size:{}".format(B_KLD,window_size))
```

```
Forward KL Divergence between I and N_U: 0.033366100048884136, window_size: 7
Backward KL Divergence between I and N_U: 0.04063894683680725, window_size:7
```

In [33]:
```
window_size=3
F_KLD = KL_divergence(I, I+N_20, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=True)
B_KLD = KL_divergence(I+N_20, I, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=True)
print("Forward KL Divergence between I and N_U: {}, window_size: {}".format(F_KLD, window_size))
print("Backward KL Divergence between I and N_U: {}, window_size:{}".format(B_KLD,window_size))
```

```
Forward KL Divergence between I and N_U: 0.03433197033858317, window_size: 3
Backward KL Divergence between I and N_U: 0.04211944943796332, window_size:3
```

In [34]:
```
window_size=1
F_KLD = KL_divergence(I, I+N_20, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=True)
B_KLD = KL_divergence(I+N_20, I, window_size=window_size, parzen_window=True, apply_to_A=True, apply_to_B=True)
print("Forward KL Divergence between I and N_U: {}, window_size: {}".format(F_KLD, window_size))
print("Backward KL Divergence between I and N_U: {}, window_size:{}".format(B_KLD,window_size))
```

```
Forward KL Divergence between I and N_U: 0.034606133956389784, window_size: 1
Backward KL Divergence between I and N_U: 0.04133233034511477, window_size:1
```

From the above values we can say that, I is able to approximate I+N_20 in a better manner than I+N_20 is able to approximate I, as value of Forward KL divergence is smaller compare to Backward KL divergence.

We can also observe that as window_size of the parzen window increases, overall Forward and backward KL divergence decreases. This means that PDF as more smooth and affect of noise in I+N_20 decreases.

2 1.3 Registration

2.0.1 1.3.1 Registration on I1_1 with I1_2 and I2_1 with I2_2 using Mutual Information and Mean Suqared Error based similarity metrics

```
In [35]: I1_1 = mpimg.imread(os.path.join(folder_path+'I1_1.png'))          # read image
         I1_2 = mpimg.imread(os.path.join(folder_path+'I1_2.png'))          # read image
         I2_1 = mpimg.imread(os.path.join(folder_path+'I2_1.png'))          # read image
         I2_2 = mpimg.imread(os.path.join(folder_path+'I2_2.png'))          # read image

         I1_1 = (I1_1[:,:,0]*256).astype(int)
         I1_2 = (I1_2[:,:,0]*256).astype(int)
         I2_1 = (I2_1[:,:,0]*256).astype(int)
         I2_2 = (I2_2[:,:,0]*256).astype(int)
```

We will consider the background tobe of zero (0) value.

```
In [36]: def shift(A, num_shift=1, axis=0, pad_value=0):

             if num_shift==0:
                 return A

             else:

                 A_shift = np.ones(A.shape, dtype='int')*pad_value

                 if axis == 0:
                     if num_shift>0: # move down
                         A_shift[num_shift::,:] = A[0:-num_shift,:]
                     else:           # move up
                         A_shift[0:-abs(num_shift),:] = A[abs(num_shift)::,:]
                 else:
                     if num_shift>0: # move right
                         A_shift[:,num_shift::] = A[:,0:-num_shift]
                     else:           # move left
                         A_shift[:,0:-abs(num_shift)] = A[:,abs(num_shift)::]

                 return A_shift
```

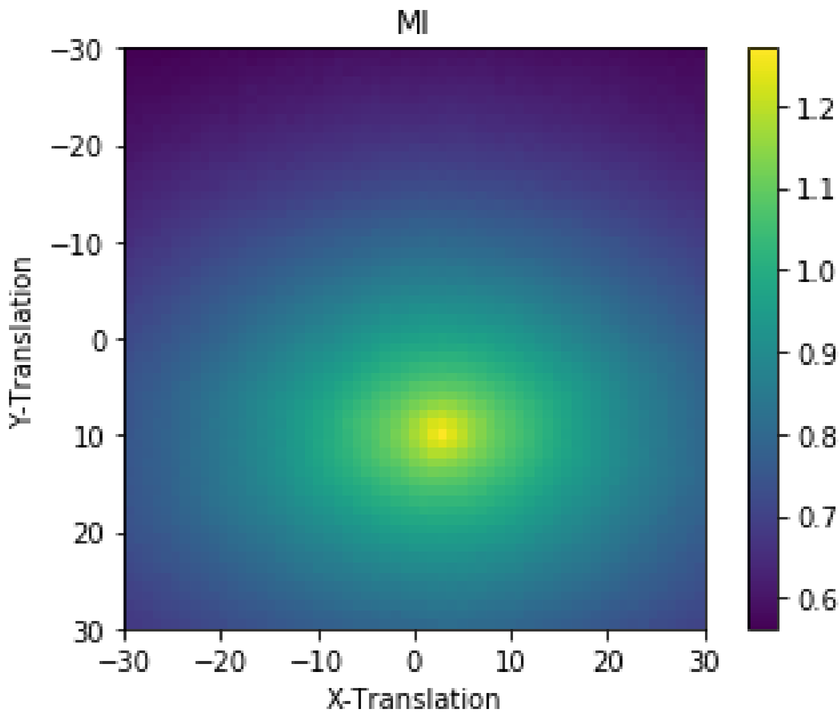
For simplicity we will ristrict our translation values to be in the range of -30:30.

2.0.2 Registration of I1_1 and I1_2

```
In [37]: translation_range = 30
         moving_image = np.copy(I1_1)
         fixed_image = np.copy(I1_2)

         translation_values = np.arange(-translation_range,translation_range+1,1)

         MSE = np.zeros((translation_values.size, translation_values.size))
         MI = np.zeros((translation_values.size, translation_values.size))

         for i,x in enumerate(translation_values):
             for j,y in enumerate(translation_values):
                 mv = shift(moving_image, x, axis=0)
                 mv = shift(mv          , y, axis=1)

                 MSE[i,j] = ((fixed_image - mv)**2).mean()
                 MI[i,j]  = MI_AB(fixed_image, mv)
```

Mutual Information based Registration

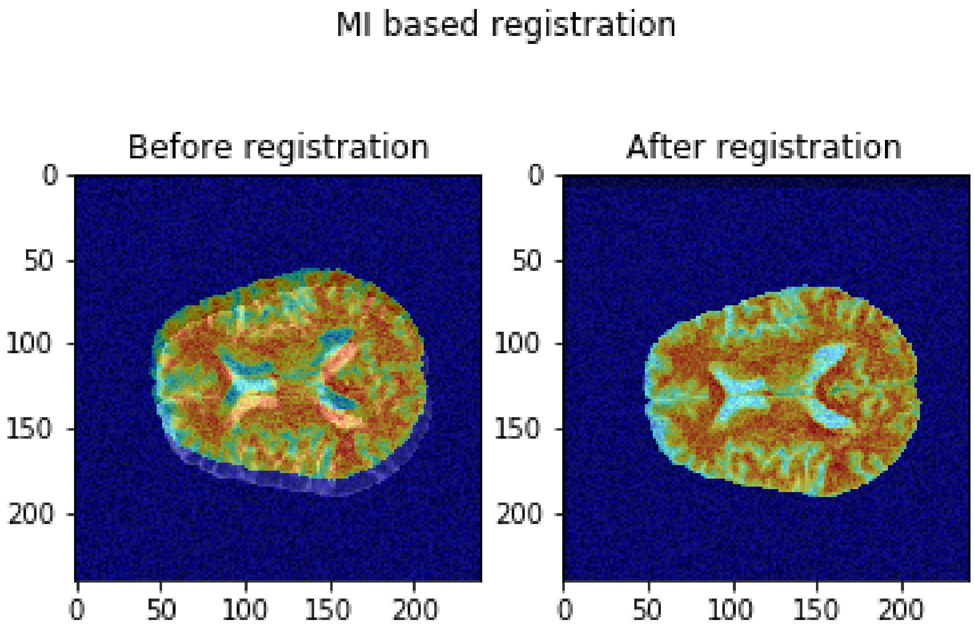
```
In [38]: fig, ax = plt.subplots()
         im = plt.imshow(MI, interpolation='none', extent=[-translation_range,translation_range,translation_range,-translation_range])
         ax.set_xlabel('X-Translation')
         ax.set_ylabel('Y-Translation')
         ax.set_title('MI')
         plt.colorbar()
         plt.show()
```




```
In [39]: # we want to find where MI in maximum
         MI_indices = np.asarray(np.unravel_index(np.argmax(MI, axis=None), MI.shape)) - translation_range
         print('MI based best translation parameters: X-translation = {}, Y-translation = {}'.format(MI_indices[0],MI_indices[1]))
```

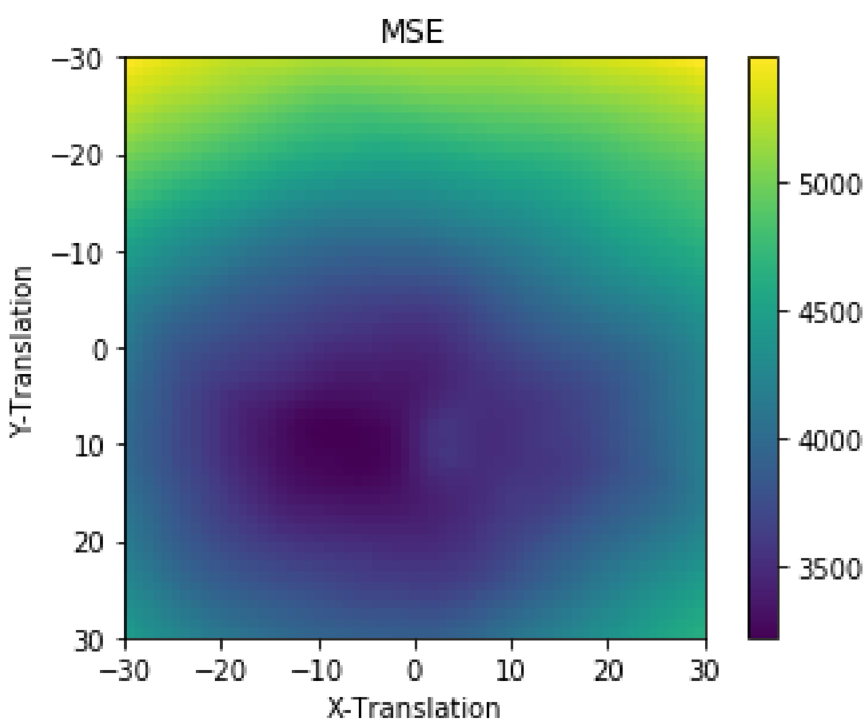
```
MI based best translation parameters: X-translation = 10, Y-translation = 3
```

```
In [40]: plt.subplot(121)
         plt.imshow(I1_2, cmap='gray')
         plt.imshow(I1_1, cmap='jet', alpha=0.5)
         plt.title('Before registration')
         plt.subplot(122)
         plt.imshow(I1_2, cmap='gray')
         mv = shift(I1_1, MI_indices[0], axis=0)
         mv = shift(mv, MI_indices[1], axis=1)
         plt.imshow(mv, cmap='jet', alpha=0.5)
         plt.title('After registration')
         plt.suptitle('MI based registration')
         plt.show()
```



Mean Squared Error Based Registration

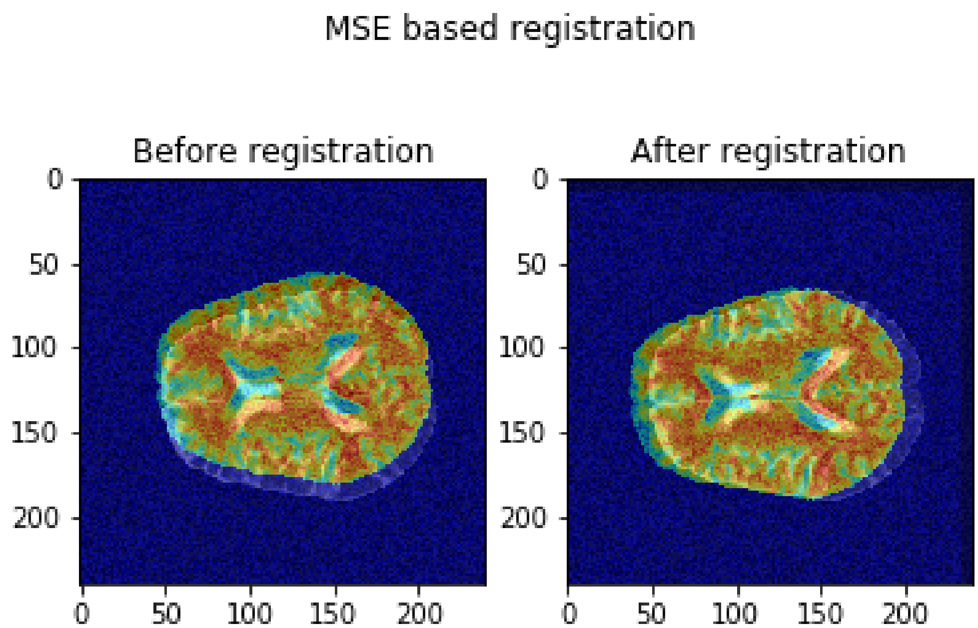
```
In [41]: fig, ax = plt.subplots()
         im = plt.imshow(MSE, interpolation='none', extent=[-translation_range,translation_range,translation_range,-translation_range])
         ax.set_xlabel('X-Translation')
         ax.set_ylabel('Y-Translation')
         ax.set_title('MSE')
         plt.colorbar()
         plt.show()
```




```
In [42]: # we want to find where MSE in minimum
         MSE_indices = np.asarray(np.unravel_index(np.argmin(MSE, axis=None), MSE.shape)) - translation_range
         print('MSE based best translation parameters: X-translation = {}, Y-translation = {}'.format(MSE_indices[0],MSE_indices[1]))
```

```
MSE based best translation parameters: X-translation = 9, Y-translation = -8
```

```
In [43]: plt.subplot(121)
         plt.imshow(I1_2, cmap='gray')
         plt.imshow(I1_1, cmap='jet', alpha=0.5)
         plt.title('Before registration')
         plt.subplot(122)
         plt.imshow(I1_2, cmap='gray')
         mv = shift(I1_1, MSE_indices[0], axis=0)
         mv = shift(mv, MSE_indices[1], axis=1)
         plt.imshow(mv, cmap='jet', alpha=0.5)
         plt.title('After registration')
         plt.suptitle('MSE based registration')
         plt.show()
```



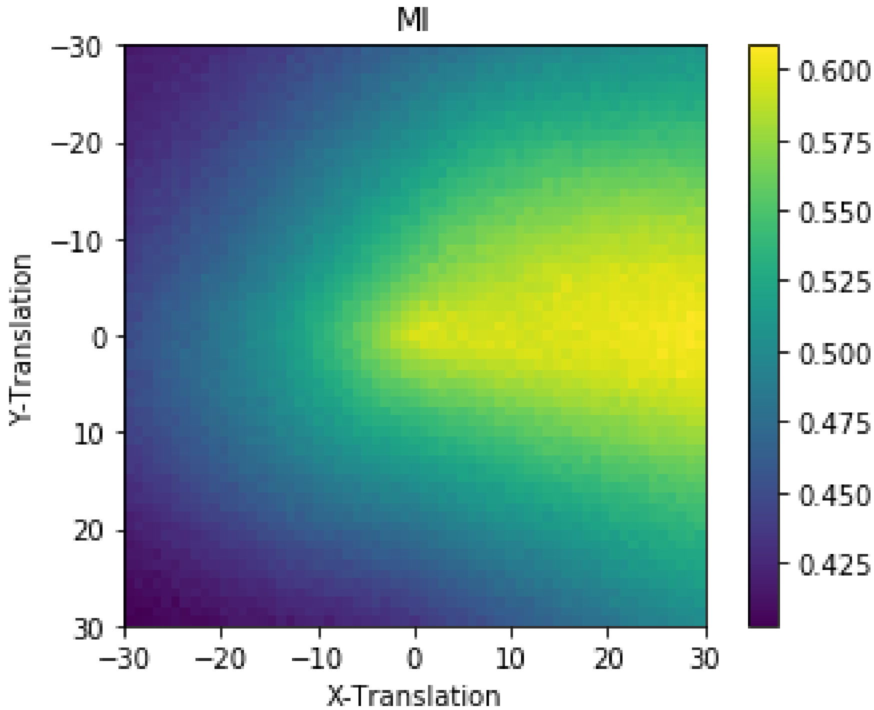
From above, we can see that images (I1_2 and I1_1) are aligned properly after MI based registration while they are not aligned properly after MSE based registration

2.0.3 Registration of I2_1 and I2_2

```
In [44]: translation_range = 30
         moving_image = np.copy(I2_1)
         fixed_image = np.copy(I2_2)

         translation_values = np.arange(-translation_range,translation_range+1,1)

         MSE = np.zeros((translation_values.size, translation_values.size))
         MI = np.zeros((translation_values.size, translation_values.size))

         for i,x in enumerate(translation_values):
             for j,y in enumerate(translation_values):
                 mv = shift(moving_image, x, axis=0)
                 mv = shift(mv          , y, axis=1)

                 MSE[i,j] = ((fixed_image - mv)**2).mean()
                 MI[i,j]  = MI_AB(fixed_image, mv)
```

Mutual Information based Registration

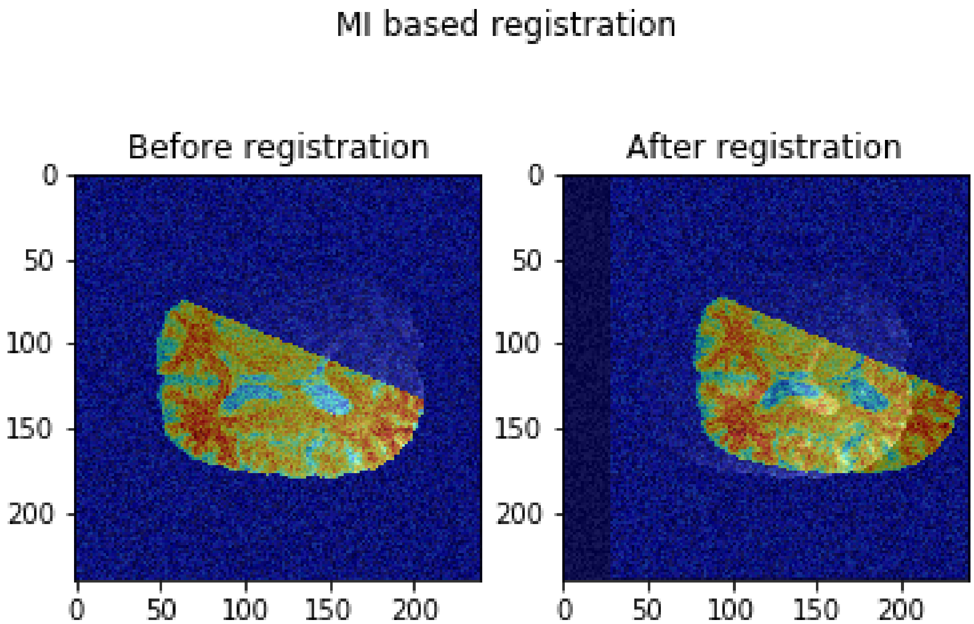
```
In [45]: fig, ax = plt.subplots()
         im = plt.imshow(MI, interpolation='none', extent=[-translation_range,translation_range,translation_range,-translation_range])
         ax.set_xlabel('X-Translation')
         ax.set_ylabel('Y-Translation')
         ax.set_title('MI')
         plt.colorbar()
         plt.show()
```




```
In [46]: # we want to find where MI in maximum
         MI_indices = np.asarray(np.unravel_index(np.argmax(MI, axis=None), MI.shape)) - translation_range
         print('MI based best translation parameters: X-translation = {}, Y-translation = {}'.format(MI_indices[0],MI_indices[1]))
```

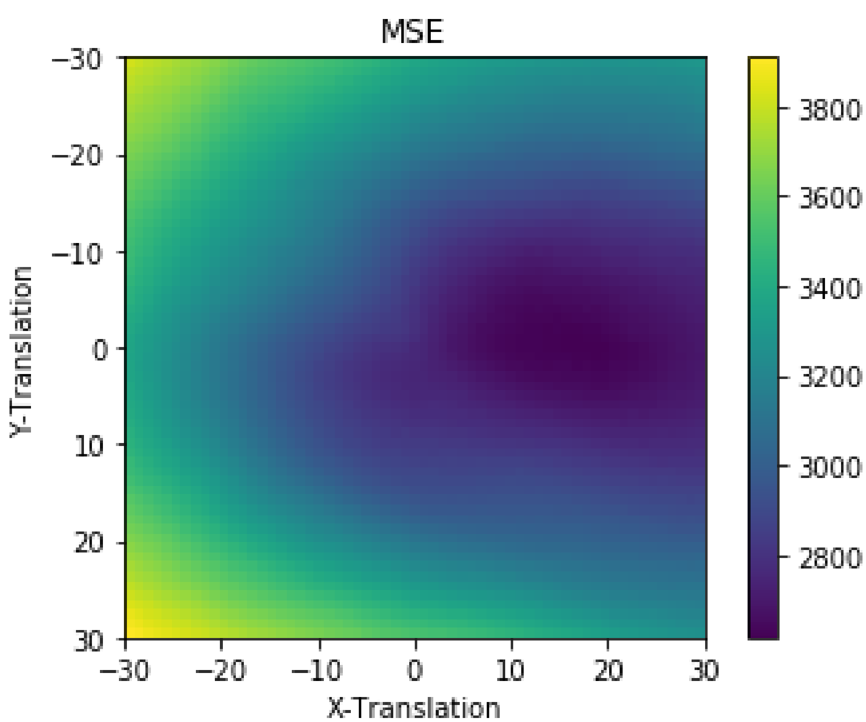
```
MI based best translation parameters: X-translation = -2, Y-translation = 29
```

```
In [47]: plt.subplot(121)
         plt.imshow(I2_2, cmap='gray')
         plt.imshow(I2_1, cmap='jet', alpha=0.5)
         plt.title('Before registration')
         plt.subplot(122)
         plt.imshow(I2_2, cmap='gray')
         mv = shift(I2_1, MI_indices[0], axis=0)
         mv = shift(mv, MI_indices[1], axis=1)
         plt.imshow(mv, cmap='jet', alpha=0.5)
         plt.title('After registration')
         plt.suptitle('MI based registration')
         plt.show()
```



Mean Squared Error Based Registration

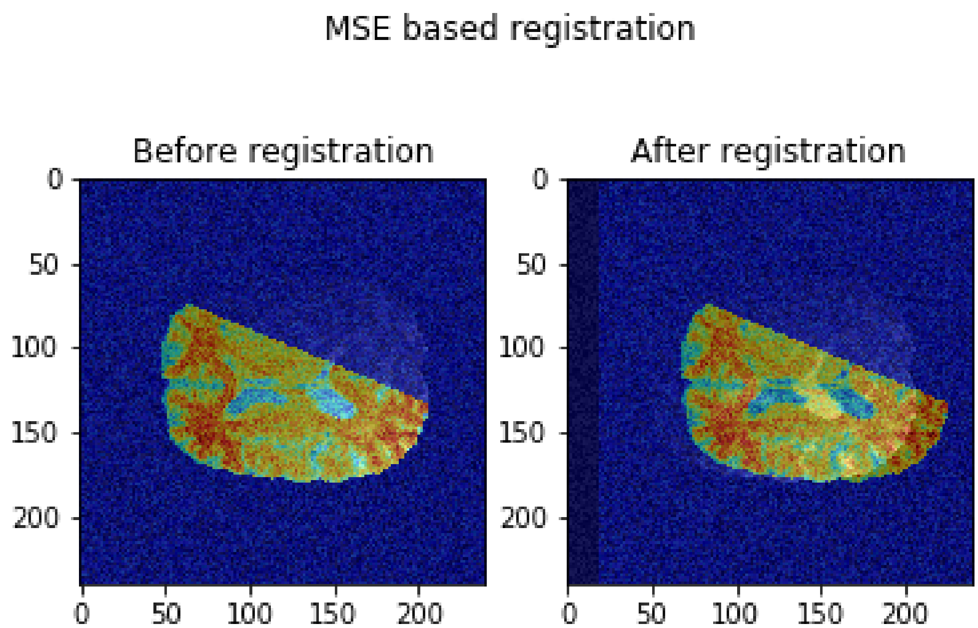
```
In [48]: fig, ax = plt.subplots()
         im = plt.imshow(MSE, interpolation='none', extent=[-translation_range,translation_range,translation_range,-translation_range])
         ax.set_xlabel('X-Translation')
         ax.set_ylabel('Y-Translation')
         ax.set_title('MSE')
         plt.colorbar()
         plt.show()
```




```
In [49]: # we want to find where MSE in minimum
         MSE_indices = np.asarray(np.unravel_index(np.argmin(MSE, axis=None), MSE.shape)) - translation_range
         print('MSE based best translation parameters: X-translation = {}, Y-translation = {}'.format(MSE_indices[0],MSE_indices[1]))
```

```
MSE based best translation parameters: X-translation = 0, Y-translation = 19
```

```
In [50]: plt.subplot(121)
         plt.imshow(I2_2, cmap='gray')
         plt.imshow(I2_1, cmap='jet', alpha=0.5)
         plt.title('Before registration')
         plt.subplot(122)
         plt.imshow(I2_2, cmap='gray')
         mv = shift(I2_1, MSE_indices[0], axis=0)
         mv = shift(mv, MSE_indices[1], axis=1)
         plt.imshow(mv, cmap='jet', alpha=0.5)
         plt.title('After registration')
         plt.suptitle('MSE based registration')
         plt.show()
```



From above, we can see that neither MI or MSE is able to find good translation parameters to register I2_1 and I2_2, as we can clearly see that images are misregistered after registration.

In fact, we can see that images were registered properly before registration so optimal translation parameters from either MI or MSE should have been 0,0. But they fail to give these parameters

2.0.4 1.3.2 Discuss the above results in terms of the assumptions inherent to the metrics. Describe the context in which each metric should be used. Support your arguments with an example or two.

In the previous quesiont, we saw that MI was able to register I1_1 and I1_2 properly, while it was not possible with MSE based metric.

We can say that as MSE is computing pixel-wise squared error, it won't give optimal parameters for registration when moving and fixed images follow different intensity distribution. I.e., I1_1 and I1_2. MSE based reigstration will be able to recover optimal parameters if moving and fixed images follow same intensity distributions. I.e., I1_1 and I2_1 or I1_2 and I2_2.

While MI will be able to give optimal parameters as it relies on marignal entropy and joint entropy. MI based metric tries to find parameters where marginal entropy is maximum and joint entropy is minimum. Due to this MI based registration is able to handle cases where moving and fixed images are from different distributions. I.e. I1_1 and I1_2.

Both these metrics will fail in cases where images are highly corrupted with noise or non-uniform intensity field, Or when both images don't capture the same structure and there is minimum overlap between their captured structure. I.e. I2_1 and I2_2, where we can see that I2_1 is missing top right part of the brain and I2_2 is highehly corrupted by non-uniform intensity field ob top left part.

Q2

February 25, 2019

1 2. Face Recognition

In [1]:
```
import os
from glob import glob
import numpy as np
import random
from random import shuffle
import imageio
import matplotlib.pyplot as plt
from sklearn.preprocessing import normalize
from skimage.transform import resize
from scipy.stats import multivariate_normal
np.random.seed(0)
random.seed(0)
```

In [2]:
```
folder_path = 'Color_FERET_Database/' # main_folder path
```

In [3]:
```
subdirectory_paths = glob(folder_path+'*') # full path of all the sub-directories (subjects)
num_subjects = len(subdirectory_paths) # total number of sub-diretories (subjects)
```

In [4]:
```
train_percentage = 0.8 # train split percentage
total = 0              # total images in a database
total_train = 0        # total images in training set
total_test = 0         # total images in test set

for path in subdirectory_paths: # go through each sub-directory
    total_files = len(glob(path+'/*.ppm')) # read all ppm files in a sub-directory
    total += total_files
    total_train += int(total_files*train_percentage)
    total_test += total_files - int(total_files*train_percentage)

print('Total Train files: {}, Total Test Files: {}'.format(total_train, total_test))
```

```
Total Train files: 1603, Total Test Files: 426
```

Original Image file size is 768x512x3 (RGB). We will convert it into grayscale and downsample it by a factor of 4. So resized file size will be 192x128

In [5]:
```
W = 768
H = 512
downsample_factor = 4
H_d = int(H/downsample_factor)
W_d = int(W/downsample_factor)

X_train = np.zeros((W_d*H_d, total_train))
X_test  = np.zeros((W_d*H_d, total_test))
Y_train = np.zeros(total_train)
Y_test  = np.zeros(total_test)
```

In [6]:
```
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
```

In [7]:
```
train_counter = 0
test_counter = 0

subject_counter = 0
subject_frequency = np.zeros(len(subdirectory_paths))


for path in subdirectory_paths:  # go through each sub-directory

    file_names = glob(path+'/*.ppm')  # read all ppm files in a sub-directory
    shuffle(file_names)               # random shuffle file_names

    # check number of training images for a particular sub-directory (subject)
    train_set_size = int(len(file_names)*train_percentage)
    subject_frequency[subject_counter] = train_set_size
```

```
    for i, file in enumerate(file_names):
        # read, convert-to-gray scale, resize, and normalize the image to 0-1
        img = resize(rgb2gray(imageio.imread(file)), (W_d,H_d), mode='constant') / 255

        # add either to train set or test set
        if i < train_set_size:
            X_train[:, train_counter] = img.ravel()
            Y_train[train_counter] = subject_counter
            train_counter += 1
        else:
            X_test[:, test_counter] = img.ravel()
            Y_test[test_counter] = subject_counter
            test_counter += 1

    subject_counter += 1

subject_frequency /= total_train
```

1.1 2.1 Principal Component Analysis

1.1.1 2.1.1 EigenFaces

The training set can be placed into a matrix X = [x1, x2, . . . , xD] of size NxD, N being the total number of pixels in an image x and D being the total number of training images. Compute the principal components using the snap-shot method. Display the mean face and the first 10 Eigenfaces.

```
In [8]: mean_face = np.mean(X_train, axis=1)
        std_face = np.std(X_train, axis=1)
```

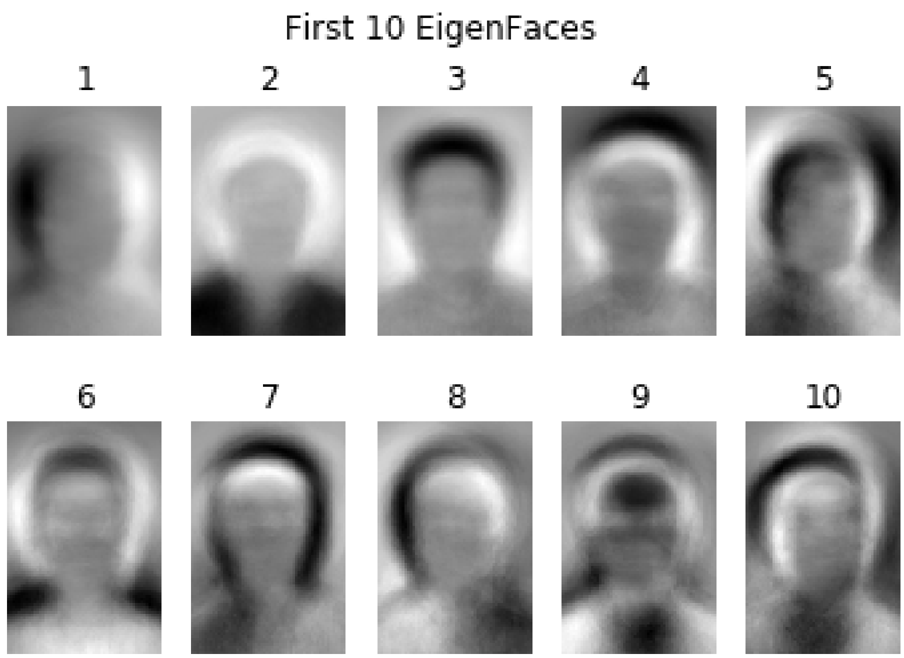
```
In [9]: plt.imshow(mean_face.reshape((W_d,H_d)), cmap='gray')
        plt.axis('off')
        plt.title('Mean Face of Training Dataset')
        plt.show()
```


```
In [10]: X_train = (X_train.T - mean_face).T # subtract mean face from training dataset
         #X_train = (X_train.T / std_face).T # divide by std face from training dataset
```

```
In [11]: C_ = np.dot(X_train.T, X_train)                    # convert into DxD matrix, D = number of examples
         eigen_values, eigen_vectors_V = np.linalg.eig(C_)  # get eigenvalues and eigenvectors for C_

         eigen_vectors = (1/eigen_values) *   np.dot(X_train,eigen_vectors_V) # get eigenvectors for NxN matrix
         eigen_vectors = normalize(eigen_vectors, axis=0) # normalize each eigenvectors to unit norm #   (1/np.sqrt(eigen_values)) *


         # sort eigenvalues and its corresponding eigenvectors in descending order
```

```
idx = eigen_values.argsort()[::-1]
eigen_values = eigen_values[idx]
eigen_vectors = eigen_vectors[:,idx]
eigen_vectors_V = eigen_vectors_V[:,idx]
```

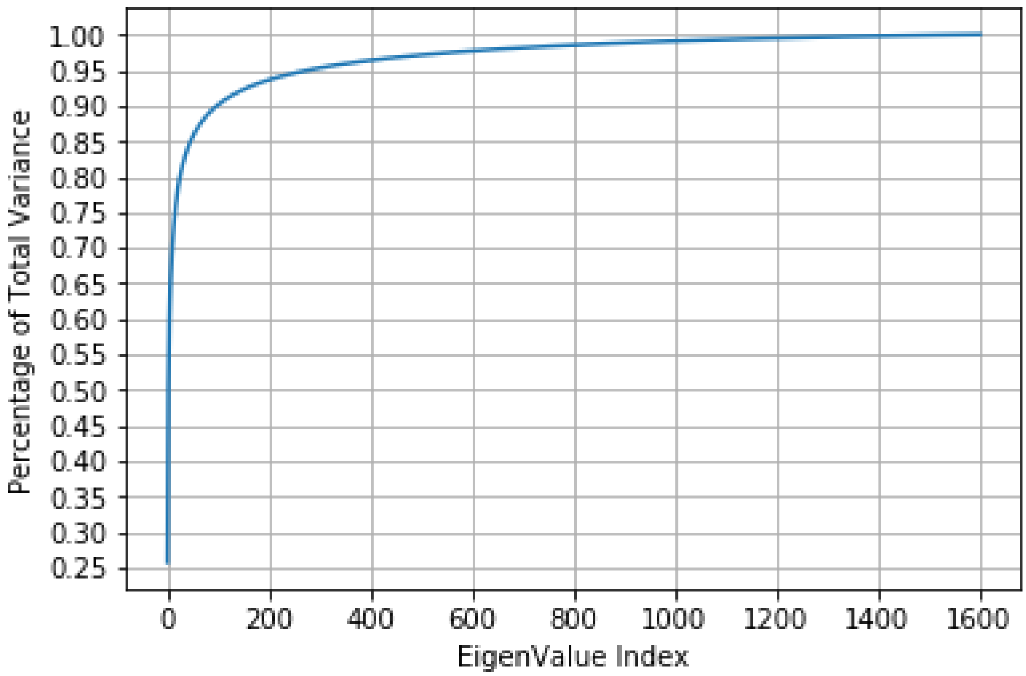
In [12]:
```
count = 1
for i in range(1,6):
    for j in range(1,3):
        plt.subplot(2,5,count)
        plt.imshow(eigen_vectors[:,count].reshape(W_d,H_d), cmap='gray')
        plt.axis('off')
        plt.title('{}'.format(count))
        count += 1


plt.suptitle('First 10 EigenFaces')
plt.show()
```



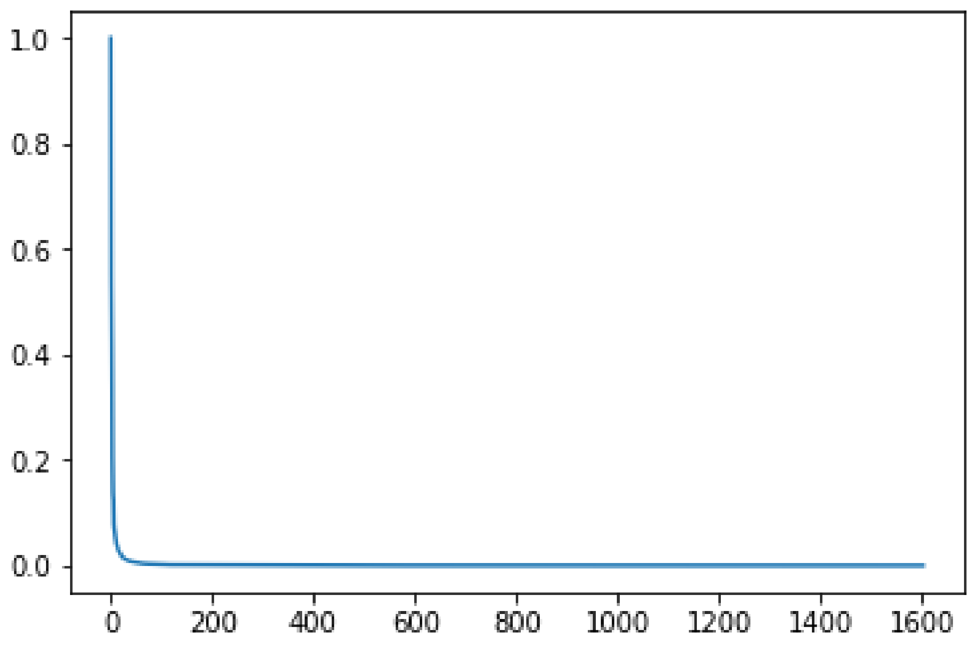
1.1.2 2.1.2 Number of Principal components required

Decide on the number of principal components N_p required to represent the data. You can use either of the two methods for dimensionality estimation discussed in the class. Justify your choice and support it with an appropriate graph.

In [13]:
```
percentage_of_total_variance = eigen_values.cumsum()/eigen_values.sum()
plt.plot(percentage_of_total_variance)
plt.grid(True)
plt.yticks(np.arange(0.25, 1.01, step=0.05))
plt.xlabel('EigenValue Index')
plt.ylabel('Percentage of Total Variance')
plt.show()
```




```
In [14]: normalised_eigen_values = eigen_values / np.max(eigen_values)
         plt.plot(normalised_eigen_values)
         plt.show()
```



We can choose N_p according to threshold on either total explained variacne or normalised eigen values. Threshold on total explained variance can be useful when we are trying to do face detection and want to use reconstruction of the faces. In this case as we are interested in face classification threshold on total explanied variance might not be useful. Similarly we can also see that eigen values are skewed, i.e. initial eigen values are comparatively really high then later eigen values. Due to this we will choose number of eigen vectors according to threshold on normalized eigen values. Specifically, we will choose threshold where normalized eigen values are >0.05.

```
In [15]: N_p = np.max(np.where(percentage_of_total_variance < 0.95)) + 2
         print('Number of Prinicipal Components N_p required to represent data according to 0.95 total variance: {}'.format(N_p))

         N_p = np.max(np.where(normalised_eigen_values > 0.05)) + 1
         print('Number of Prinicipal Components N_p required to represent data according to 0.05 normalized eigen values: {}'.format(N_p))
```

```
Number of Prinicipal Components N_p required to represent data according to 0.95 total variance: 279
Number of Prinicipal Components N_p required to represent data according to 0.05 normalized eigen values: 10
```

1.2 2.2 Probabilistic Face Recognition

1.2.1 2.2.1 Using Bayes Rule, derive an expression for the posterior density of a subject label y for a given test image x*

$$P(y^*/\phi(x^*), \phi(X), Y) = \frac{P(\phi(x^*)/y^*, \phi(X), Y) \quad P(y^*/\phi(X), Y)}{P(\phi(x^*)/\phi(X), Y)} \tag{1}$$

$$\therefore P(y^*/\phi(x^*)) = \frac{P(\phi(x^*)/y^*, \phi(X), Y) \quad P(y^*/\phi(X), Y)}{\sum_{y^*} P(\phi(x^*)/y^*, \phi(X), Y) \quad P(y^*/\phi(X), Y)} \tag{2}$$

1.2.2 2.2.2 Find eigher representation phi(x) for each training image x i.e. project the image x on the first N_p Eigenfaces and find corresponding N_p coefficients.

In [16]:
```
Top_eigenvectors = eigen_vectors[:,0:N_p]
```

In [17]:
```
# Projection of Training data onto Eigenspace
Proj_X_train = np.dot(X_train.T, Top_eigenvectors)
```

1.2.3 2.2.3 Build total 52 Gaussian density functions for the likelihood, one for each subject in the training set.

When we project the data into EigenSpace; overall data becomes uncorrelated, therefore

In [18]:
```
Mean = np.zeros((num_subjects, N_p))
Covar = np.zeros((num_subjects, N_p, N_p))

# we will use scipy.stats.multivariate_normal to calcualte multivariate gaussian distribution.
# keep in mind that this function uses Pseudo Inverse and Pseudo Determinant of Cov matrix.
# Ref: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.multivariate_normal.html

GDF = []
for i in range(num_subjects):
    Mean[i] = np.mean(Proj_X_train[np.equal(Y_train, i),:], axis=0)
    Covar[i] = np.cov(Proj_X_train[np.equal(Y_train, i),:].T)
    GDF.append(multivariate_normal(mean=Mean[i], cov=Covar[i]))
```

1.2.4 2.3.4 Projection of test data into Eigenspace and its recognition

In [19]:
```
X_test = (X_test.T - mean_face).T # subtract mean face from testing dataset
#X_test = (X_test.T / std_face).T # divide by std face from testing dataset
```

(i) Eigen Representation of X_test

In [20]:
```
Proj_X_test = np.dot(X_test.T, Top_eigenvectors)
```

(ii) Recognition using bayesian inference

In [21]:
```
#def mvgpdf(x, mean, covar):
#    dim = x.shape[1]
#
#    eig_values, _ = np.linalg.eig(covar)
#    pdet_cov = np.product(eig_values[eig_values > 1e-12])
#
#    mult = (1/np.sqrt(((2*np.pi)**dim)*pdet_cov))
#
#    PInv_covar = np.linalg.pinv(covar)
#    x = x-mean
#
#    return mult * np.exp(-np.sum(((x @ PInv_covar) * x),1))
```

In [22]:
```
Likelihood = np.zeros((num_subjects, Y_test.size))
```

In [23]:
```
for i in range(num_subjects):
    for j in range(Y_test.size):
        Likelihood[i, j] = GDF[i].pdf(Proj_X_test[j,:])
```

In [24]:
```
#for i in range(num_subjects):
#        Likelihood[i, :] = mvgpdf(Proj_X_test, Mean[i], Covar[i])
```

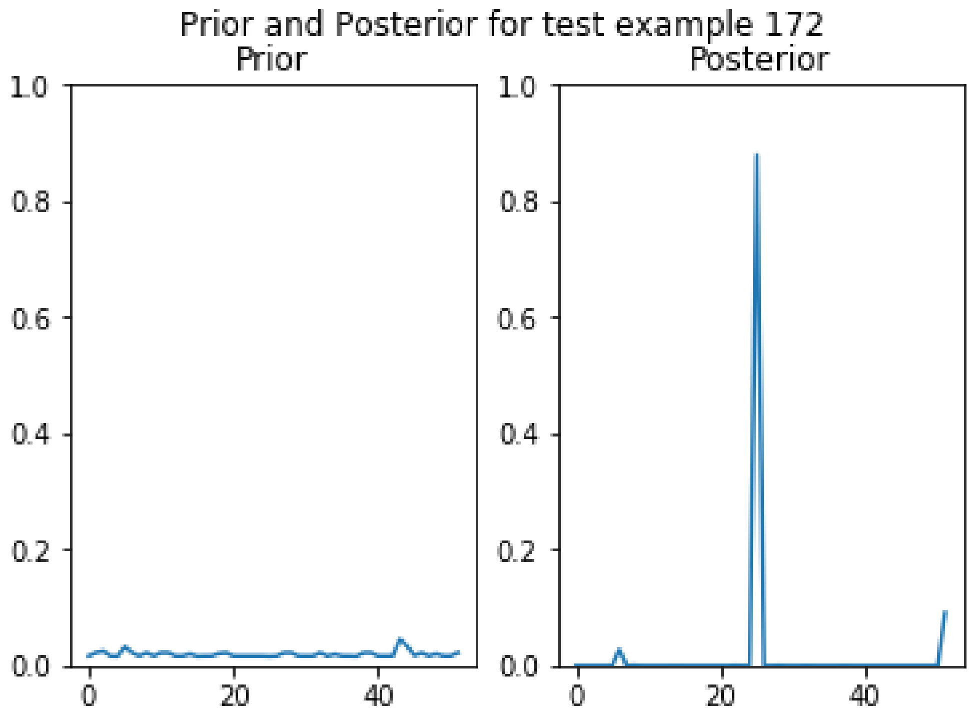
In [25]:
```
#Prior = np.ones((num_subjects, Y_test.size)) * 1/52 # equal prior
Prior = np.repeat(subject_frequency[:, np.newaxis], Y_test.size, axis=-1) # prior according to training dataset frequency
```

In [26]:
```
Marginal = Likelihood * Prior
Marginal = np.sum(Marginal, axis=0) # summation over classes for each subject
```

```
In [27]: Posterior = (Likelihood * Prior) / Marginal
```

```
In [28]: Pred_Y_test_MAP = np.argmax(Posterior,axis=0)
```

```
In [29]: example_number = np.random.randint(0,Y_test.size)
         plt.subplot(121)
         plt.plot(Prior[:,example_number])
         plt.ylim(0,1.0)
         plt.title('Prior')
         plt.subplot(122)
         plt.plot(Posterior[:,example_number])
         plt.ylim(0,1.0)
         plt.title('Posterior')
         plt.suptitle('Prior and Posterior for test example {}'.format(example_number))
         plt.show()
```



(iii) Recognition using Nearest Neighbours

```
In [30]: pairwise_diff = np.sum((Proj_X_train.reshape(-1,1,N_p) - Proj_X_test.reshape(1,-1,N_p))**2,2)
         closest_train_point = np.argmin(pairwise_diff,axis=0)
         Pred_Y_test_NN = Y_train[closest_train_point]
```

1.2.5 2.3.5 Accuracy of classification

```
In [31]: MAP_accuracy = np.sum(Pred_Y_test_MAP == Y_test) / total_test
```

```
In [32]: NN_accuracy = np.sum(Pred_Y_test_NN == Y_test) / total_test
```

```
In [33]: print("Accuracy of MAP based method: {}".format(MAP_accuracy))
```

```
Accuracy of MAP based method: 0.5751173708920188
```

```
In [34]: print("Accuracy of NN based method: {}".format(NN_accuracy))
```

```
Accuracy of NN based method: 0.676056338028169
```

From above results, we can see that NN based method is able to give better performance in comparison to MAP based method.