EEE361 HOMEWORK-1

Arda Can Aras

21704036

Bilkent EEE

The very first thing that need to be accomplished is to pre-process data to obtain desired **X**. With the given reference in the homework, the matrix composed as follows in the code segment below.

```
def load_images_from_folder(folder):
    images = []
    for filename in os.listdir(folder):
        img = mpimg.imread(os.path.join(folder,filename))
        img = img.flatten('F')
        images.append(img)
    return images

folder_train = "C:/Users/ARDA ARAS/Desktop/EEE361-HW1/Dataset/train"
folder_test = "C:/Users/ARDA ARAS/Desktop/EEE361-HW1/Dataset/test"
train_data = np.array(load_images_from_folder(folder_train))
test_data = np.array(load_images_from_folder(folder_test))
X = train_data.T
```

Observe that for the first question we are only dealing with the training data. Therefore, **X** is only composed of training data. It has 361 rows and 2429 columns. Since all images in training set has 19x19 pixels resolution, when we vectorize them they become 361x1. Therefore, every column of **X** corresponds to single sample image in vectorized form which obtained from the training data set.

Question 1

We have two parts for this question. First, we are asked to factorize matrix by using SVD and then by using NMF.

In this part, X matrix is formed by flattening the images in 'F' order which is concatenation in column order. Then, flattened images are added as columns of X vector. Therefore, last shape of X is (361,2429).

SVD factorization should be done as:

$$X = U\Sigma V^T$$

where Σ is (361,361), U is (361,361) and V is (2429,361). Σ has diagonal entries σ_i $1 <= i <= 361$.

Part 1

For the first part of the question the first task is to factorize **X** by using SVD. The following code block will accomplish this task.

Part 1-A

```
In [19]: u ,s, vt = np.linalg.svd(X,full_matrices = False)
         s_diag = np.diag(s)
         print(X.shape)
         print(u.shape)
         print(s_diag.shape)
         print(vt.shape)

(361, 2429)
(361, 361)
(361, 361)
(361, 2429)
```

From the figure above we can individually observe the size of matrixes. Where first entry in the output corresponds the row and the second one is column.

Then we are asked to plot the singular values of **X** and plot the accumulated energy. Following lines of codes used for that purpose.

Part 1-B

```
#Plotting the singular values of matrix X in logarithmic scale
plt.plot(s)
plt.title('Singular Values of X(train data) matrix')
plt.ylabel('Singular Value')
plt.xlabel('Singular Value Index')
plt.show(block = 'False')

#finding accumulated energy
s_square = [x**2 for x in s]
acc_energy = [ np.sum(s_square[:i]) for i in range(1,len(s)+1) ]
acc_energy_normalized = acc_energy / np.amax(acc_energy)
acc_energy_normalized = np.array(acc_energy_normalized)
plt.plot(acc_energy_normalized)
plt.title('Normalized Acummulated Energy')
plt.ylabel('Accumulated Energy')
plt.xlabel('Index')
plt.show(block=False)
```

We can obtain following plots when we execute the cell above.

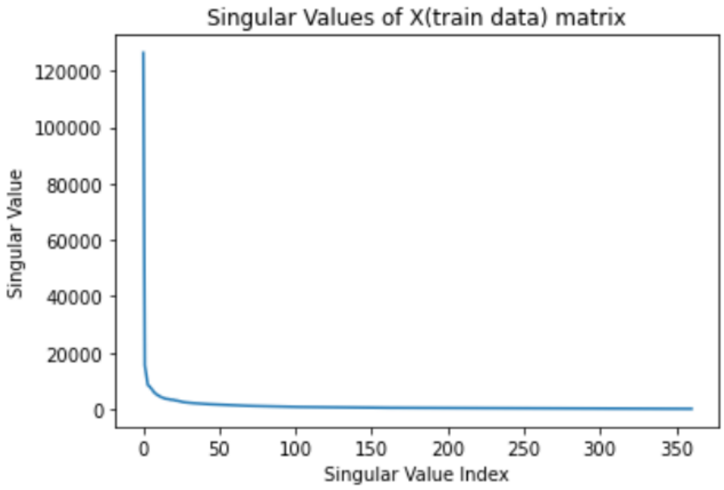


Figure 1: Singular Values of X versus Singular Value Index

From figure above we can observe that there is a significance decrease between the singular values. So, we expect to have more importance in the very first columns of **U**. This issue will be discussed later.

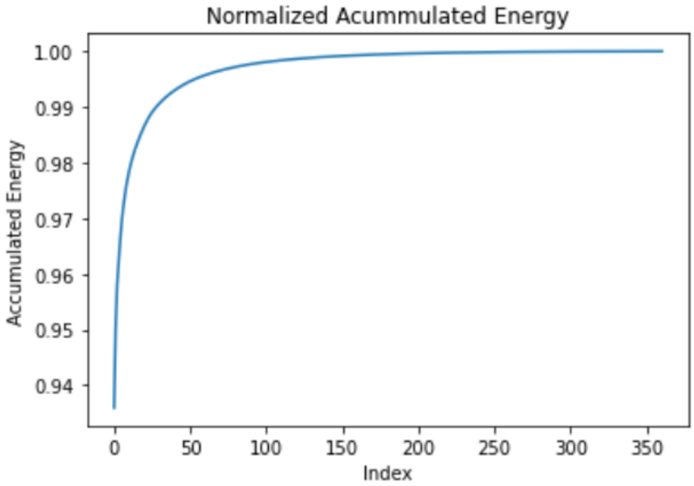


Figure 2: Normalized Accumulated Energy versus Index

We can also understand from the accumulated energy graph that, the energy reaches its 90 percentage by only considering the first index.

Then we are asked to identify some indices as defined in the homework. Following lines of code will be used to do that.

Part 1-C

```
In [7]: #PART1-C
I_90 = np.argmin(acc_energy_normalized > 0.90)
I_95 = np.argmax(acc_energy_normalized > 0.95)
I_99 = np.argmax(acc_energy_normalized > 0.99)
print("Normalized energy reaches its 0.9 at the singular value index ", I_90 + 1)
print("Normalized energy reaches its 0.95 at the singular value index ", I_95 + 1)
print("Normalized energy reaches its 0.99 at the singular value index ", I_99 + 1)

Normalized energy reaches its 0.9 at the singular value index  1
Normalized energy reaches its 0.95 at the singular value index  3
Normalized energy reaches its 0.99 at the singular value index  29
```

From the output we can understand the corresponding indexes *of* I_{90}, I_{95} and I_{99}.

Then we are asked to check the first I_{90} columns of **U**. The following lines of code can be used to accomplish this task.

```
plt.figure()
sing_face = u[:,0].reshape(19,19,order = 'F')
plt.imshow(sing_face,cmap='gray')
plt.title("The first I_90 singular faces")
plt.show(block=False)
```

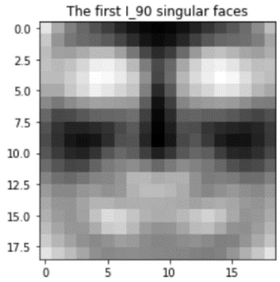


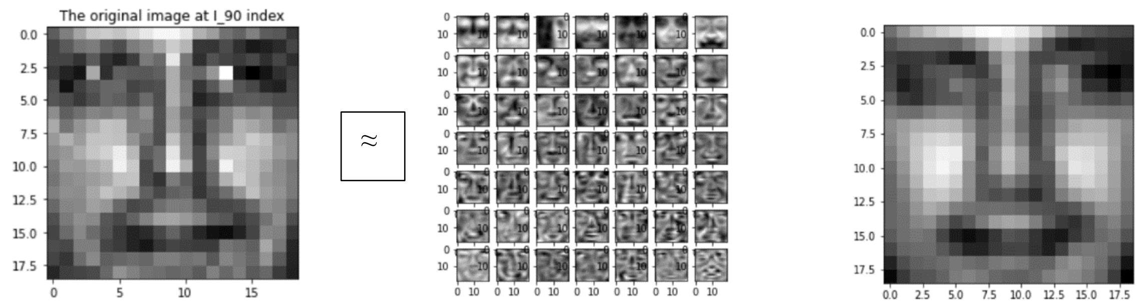
Figure 3: The first I_90 singular faces.

Since I_{90} equals to 1, we only have a single image. So, the matrix U includes the eigen faces in each column that we can totally recover and single image by using all the columns (eigen faces) in U. However, what is more interesting is with less effort we can really approximate the image by using the first r columns of U for $1 \leq r \leq 361$ for our data set. As r reaches to 361, we can totally recover the image. Let us observe this affect by using r = 49.

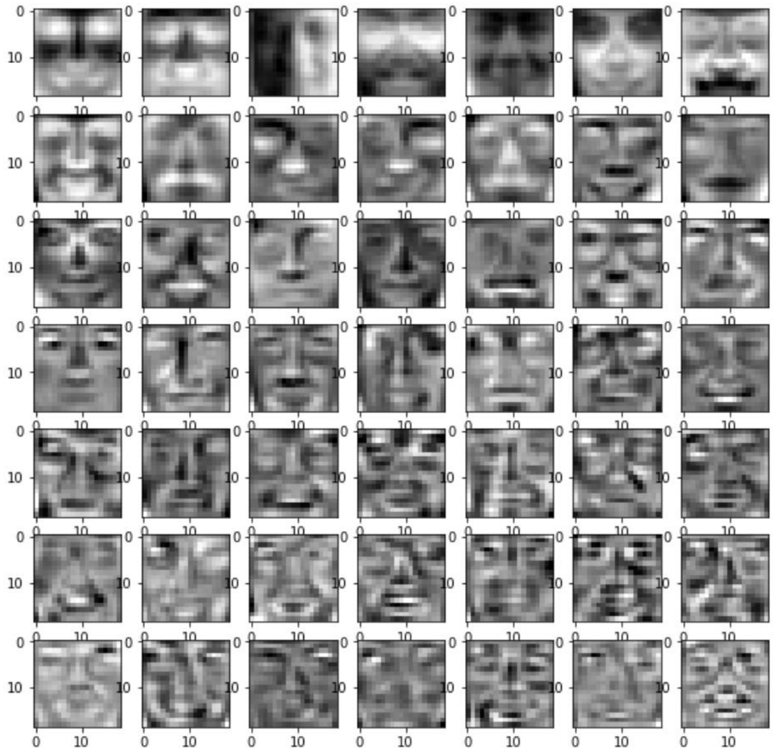
$$X_{approx} = U_{361xr}\, S_{rxr}\, V^{T}_{rx2429}$$

By using the formula above, we can approximate whole matrix and retrieve any image we want by simply selecting the specific column.

X (:,j) ≈ X_approx(:,j) (where X_approx given as above)



The figure in the middle corresponds the first r = 49 singular faces. Note that this approximation is different than the visualization in the reference [1] at the homework. In that scenario we use all the columns of W to approximate the single image to show that our X ≈ WH. Since SVD is exact factorization and we can fully recover the first image by using all the columns of U (d) section of Question 1 part1 need to be treated differently. Since the mentioned Figure 1 of [1] has different kind of approximation. The following figure is the bigger version of the singular faces above.



When I observe the eigen faces, I found that they have distributed features. We can observe that all the structures of face distributed evenly among the structures like eyes, nose, and eyebrows. When we investigate the features of W in NMF the meaning of distributed features will be clearer. Also entries of the matrix are non-negative.

Part 2

In that part we are asked to use Non-negative Matrix Factorization (NMF) to factorize matrix X. First, we are asked to use technique called HALS for factorization. Following line of codes can be used to implement this method and its updating policies. But first we need to initialize by using SVD based technique. I simply followed the instructions given in the reference [1] of homework to implement these functions. To improve the computation time, for loops in summations can be replaced by direct matrix multiplications. Therefore, I used some derivations to obtain better result.

For this part I initialize with r=49. Therefore, W matrix has 49 columns and H has 49 rows. Since r is user defined choice, I designed algorithm such that it can be computed for any r values. I obtained the following error versus iteration graph. Please check the Appendix A to see full derivations that implemented in code.

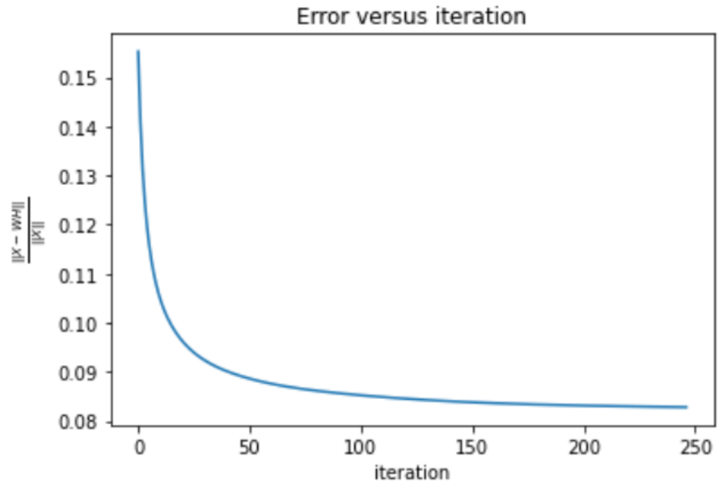


Figure 4: Error versus iteration graph for r=49.

From the figure above we can see that even the first iteration does not have large error. As we keep updating the W and H matrices, our error converges approximately to 0.10. Therefore, we can conclude that our approximation done a great job. Next, we can plot the corresponding eigenfaces which are columns of W.

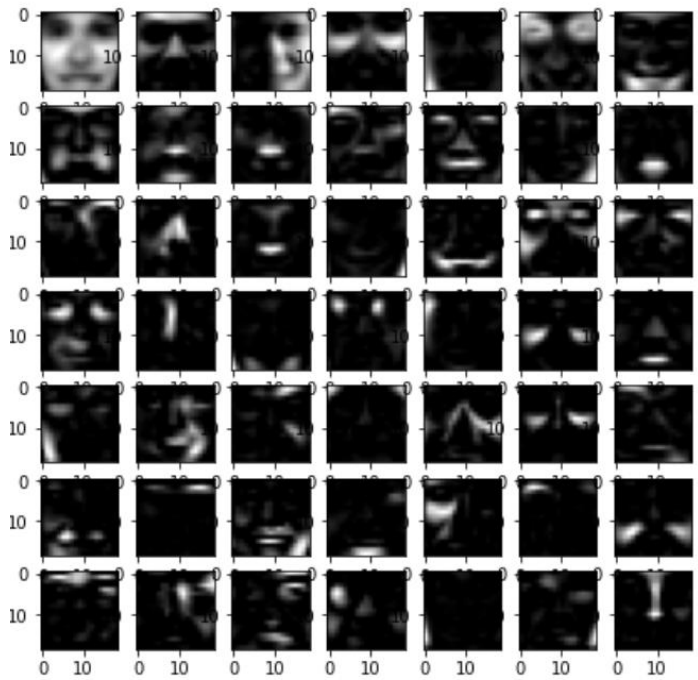
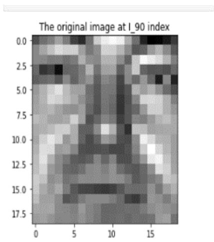
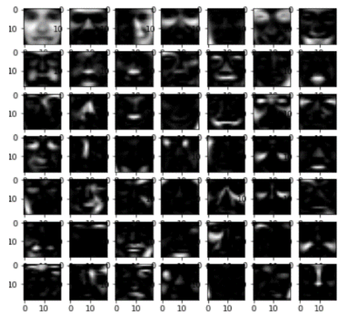


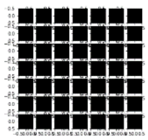
Figure 5: Eigenfaces (columns of W).

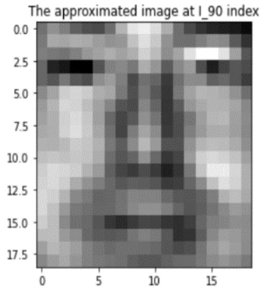
We can observe that only a single portion of face (mouth, lips, eyebrows) is lighted. When I observe the entries of the columns of W they are composed of non-negative values and positive values clumps together. So, we can conclude that it has localized feature with tries to enlighten a specific area in the face. Therefore, we can conclude that we reached our task which was to minimize the error with the constraint both W and H having non-negative entries. As it was shown in the reference [1] of the homework we can obtain the image and its approximate by using W and H.

$$\underbrace{X(:,j)}_{} \approx \sum_{k=1}^{r} \underbrace{W(:,k)}_{} \underbrace{H(k,j)}_{} = \underbrace{WH(:,j)}_{} .$$









jth facial image

facial features

importance of features in jth image

approximation of jth image

Question 2

For that question we are asked to reconstruct the noisy version of the images at the test data set and observe the effect of noise. At the previous part we had factorize our training data matrix by SVD and NMF. Since we compose singular faces and eigenfaces matrices from them, we expect to see that for a given image in test data set, by using some columns of U and W we can obtain the approximate image. For the first part we will deal with SVD based reconstruction. When I decompose the matrix X by using SVD and reconstruct the noisy version images from the columns of U, I obtained the following graphs for different noise levels.

Figure 6: Error of SVD for different values of r when n = 1,10 and 25

From the figure above, we can conclude that as we include more columns from the matrix U, our approximation gets better and better. There is an exponential decay trend as it can be seen from the graphs in the figure above. To fasten the computations, we can find simple matrix multiplication to replace summation and dot products. Check the appendix B to see full derivations.

Then we are asked to reconstruct noisy images from the columns of W. After I followed the instructions of the reconstruction, I obtained the following graphs.

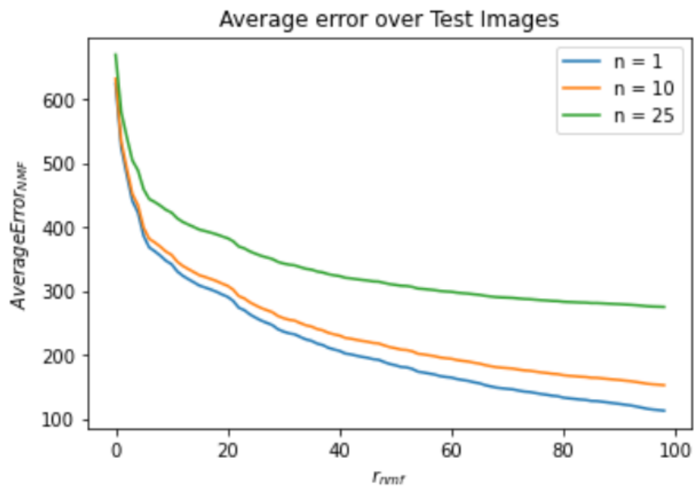


Figure 7: Error of NMF for different values of r when n = 1,10 and 25

We can also observe the same trend above like. However, initial error for NMF is larger than SVD.

Question 3

For this question we are asked to plot two graphs showing the error versus r values for SVD and NMF factorization. Since we already factorize the matrix X, we need to compose masked array and follow the instructions in homework. As a result, I obtained following graphs.

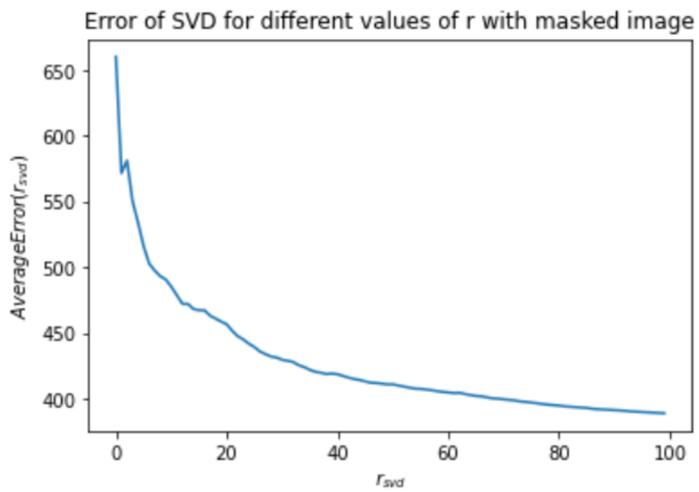


Figure 8: Error of SVD for different values of r with masked image.

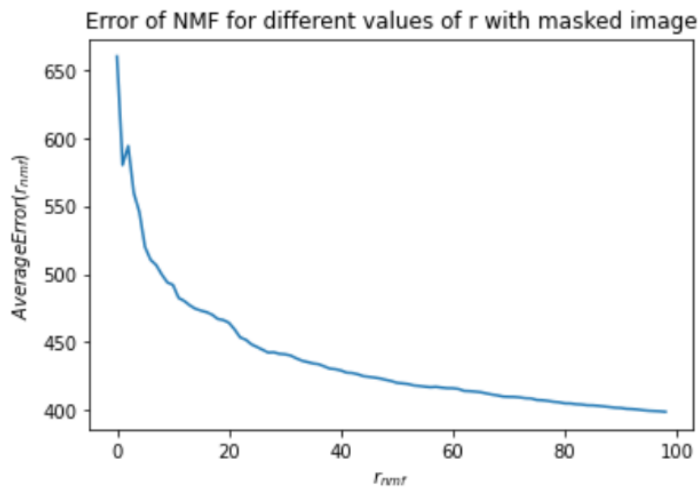


Figure 9: Error of NMF for different values of r with masked image.

When we compare the figures above, they have similar trends. As we increase the r, our approximation gets better and error decreases. Even we are asked to determine for first 100 r values, derivations of SVD speed up to process so I can compute for all the r ranging from 1 to 361. The following graph is obtained.

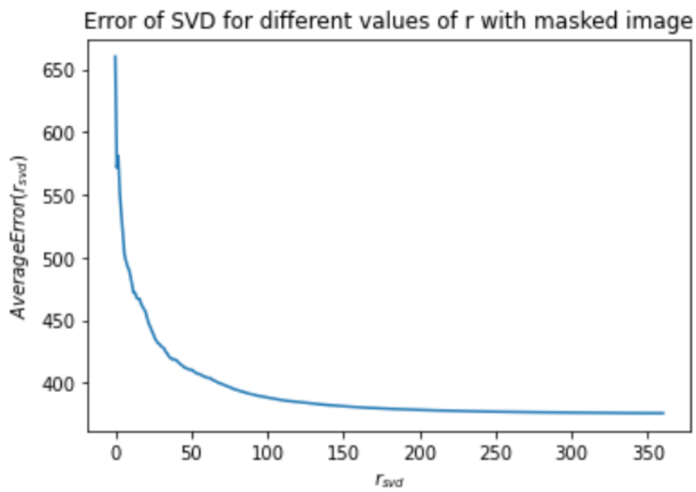


Figure 10: Error of SVD for all values of r with masked image.

At the end error converges to 376.09 when we use all of the columns and this is the best approximate we can obtain by using SVD.

Appendices

Appendix A.

HALS UPDATE DERIVATIONS

$\underline{\underline{H}}_{r\times n}$, $\underline{\underline{W}}_{p\times r}$: $\underline{\underline{H}} = \begin{bmatrix} h_1 \\ \vdots \\ h_r \end{bmatrix}$ $\quad \underline{\underline{W}} = [\underline{w}_1 \; \cdots \; \underline{w}_r]$

$\underline{\underline{X}}_{p\times n}$

$\underline{\underline{W}}(:,\ell) = \underline{w}_\ell$, $\quad \underline{\underline{H}}(\ell,:) = \underline{h}_\ell$

$$\max\left(0, \frac{\underline{\underline{X}}\,\underline{h}_\ell^T - \sum_{k\neq \ell} \underline{w}_k (\underline{h}_k \underline{h}_\ell^T)}{\|\underline{h}_\ell\|_2^2}\right) \longrightarrow \underline{w}_\ell$$

observe that

$$\sum_{k=1}^{r} \underline{w}_k (\underbrace{\underline{h}_k \underline{h}_\ell^T}_{\alpha_k}) = \alpha_1 \underline{w}_1 + \cdots + \alpha_r \underline{w}_r = \underline{\underline{W}}\,\underline{\underline{B}}_{(p\times r)(r\times 1)}$$

$\ell \rightarrow$ included

$$\alpha_k = \underline{h}_k \underline{h}_\ell^T \qquad \underline{\underline{B}} = \begin{bmatrix} h_1 \cdot h_\ell^T \\ \vdots \\ h_\ell \cdot h_\ell^T \\ \vdots \\ h_r \cdot h_\ell^T \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_r \end{bmatrix}_{r\times 1} = \underline{b}$$

$$\max\left(0, \frac{\underline{\underline{X}}\,\underline{h}_\ell^T - \underline{\underline{W}}\,\underline{\underline{B}} + \underline{w}_\ell (\underline{h}_\ell \underline{h}_\ell^T)}{\|\underline{h}_\ell\|_2^2}\right) \rightarrow \underline{w}_\ell$$

$$\max\left(0, \frac{\underline{\underline{X}}\,\underline{h}_\ell^T - \underline{\underline{W}}\,\underline{b} + \underline{w}_\ell \overbrace{(\underline{h}_\ell \underline{h}_\ell^T)}^{\underline{b}(\ell)}}{\underbrace{\|\underline{h}_\ell\|_2^2}_{\underline{b}(\ell)}}\right)$$

Appendix B.

SVD Reconstruction Derivations

$$\underline{\underline{Y}} = [\underline{y}_1 \cdots \underline{y}_{472}]$$

$$\underline{y}_k \xrightarrow{\text{noise}} \underline{\tilde{y}}_k \xrightarrow{\text{SVD}} \bar{y}_k^{\text{SVD}}$$

$$\bar{y}_k^{\text{SVD}} = \sum_{p=1}^{r^{\text{SVD}}} \alpha_p \underline{u}_p \qquad \alpha_p = \tilde{y}_k^T u_p$$

Observe that

$$\bar{y}_k^{\text{SVD}} = \underline{\underline{U}}_{(361 \times r)} \underline{\underline{U}}^T_{(r \times 361)} \underline{\underline{\tilde{y}}}_k$$

$$\downarrow$$

$$\underline{\underline{Y}}^{\text{SVD}} = [\underline{y}_1^{\text{SVD}} \cdots \underline{y}_{472}^{\text{SVD}}]$$

$$= \underline{\underline{U}}_{(361 \times r)} \underline{\underline{U}}^T_{(r \times 361)} \underline{\underline{Y}}_{(361 \times 472)}$$

→ we can simply reconstruct by closed form solution

```
In [1]:
import os
import time
import numpy as np
import numpy.ma as ma
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

Data Pre-Processing

```
In [2]:
def load_images_from_folder(folder):
    images = []
    for filename in os.listdir(folder):
        img = mpimg.imread(os.path.join(folder,filename))
        img = img.flatten('F')
        images.append(img)
    return images

folder_train = "C:/Users/ARDA ARAS/Desktop/EEE361-HW1/Dataset/train"
folder_test = "C:/Users/ARDA ARAS/Desktop/EEE361-HW1/Dataset/test"
train_data = np.array(load_images_from_folder(folder_train))
test_data = np.array(load_images_from_folder(folder_test))
X = train_data.T
X_test = test_data
```

Question 1

Part 1

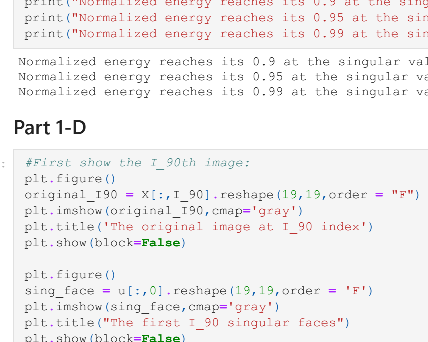
Part 1-A

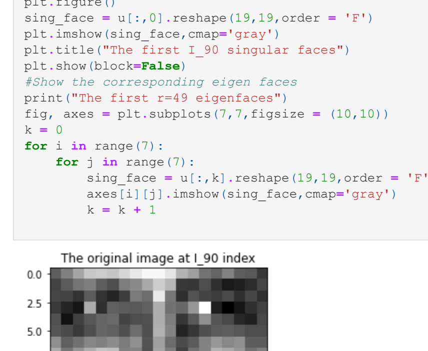
```
In [3]:
u ,s, vt = np.linalg.svd(X,full_matrices = False)
s_diag = np.diag(s)
print(X.shape)
print(u.shape)
print(s_diag.shape)
print(vt.shape)
```

```
(361, 2429)
(361, 361)
(361, 361)
(361, 2429)
```

Part 1-B

```
In [4]:
#Plotting the singular values of matrix X in logarithmic scale
plt.plot(s)
plt.title('Singular Values of X(train data) matrix')
plt.ylabel('Singular Value')
plt.xlabel('Singular Value Index')
plt.show(block = 'False')

#finding accumulated energy
s_square = [x**2 for x in s]
acc_energy = [ np.sum(s_square[:i]) for i in range(1,len(s)+1) ]
acc_energy_normalized = acc_energy / np.amax(acc_energy)
acc_energy_normalized = np.array(acc_energy_normalized)
plt.plot(acc_energy_normalized)
plt.title('Normalized Acummulated Energy')
plt.ylabel('Accumulated Energy')
plt.xlabel('Index')
plt.show(block=False)
```





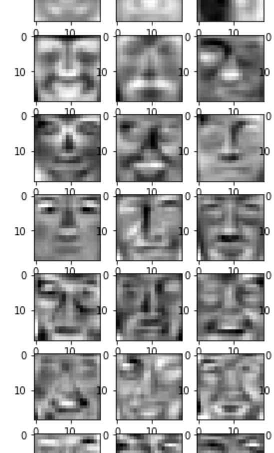
Part 1-C

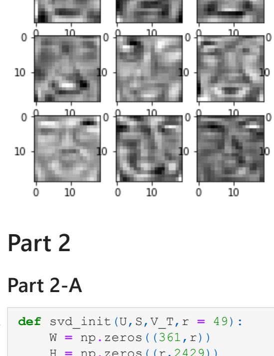
```
In [5]:
#PART1-C
I_90 = np.argmin(acc_energy_normalized > 0.90)
I_95 = np.argmax(acc_energy_normalized > 0.95)
I_99 = np.argmax(acc_energy_normalized > 0.99)
print("Normalized energy reaches its 0.9 at the singular value index ", I_90 + 1)
print("Normalized energy reaches its 0.95 at the singular value index ", I_95 + 1)
print("Normalized energy reaches its 0.99 at the singular value index ", I_99 + 1)
```

```
Normalized energy reaches its 0.9 at the singular value index  1
Normalized energy reaches its 0.95 at the singular value index  3
Normalized energy reaches its 0.99 at the singular value index  29
```

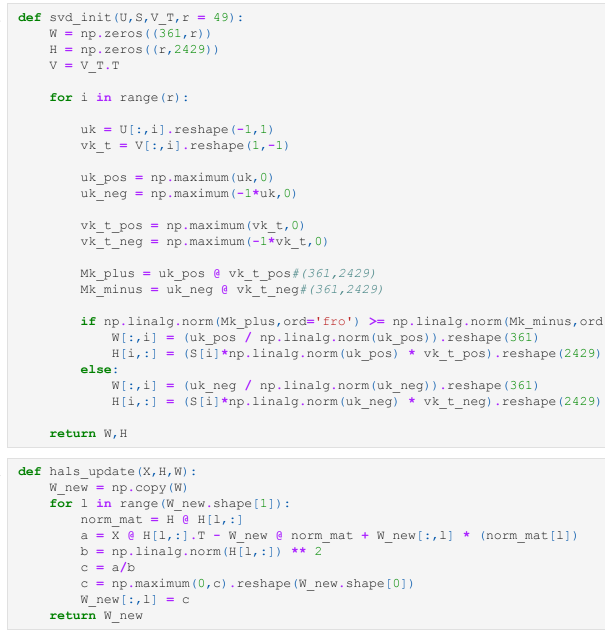
Part 1-D

```
In [6]:
#First show the I_90th image:
plt.figure()
original_I90 = X[:,I_90].reshape(19,19,order = "F")
plt.imshow(original_I90,cmap='gray')
plt.title('The original image at I_90 index')
plt.show(block=False)

plt.figure()
sing_face = u[:,0].reshape(19,19,order = 'F')
plt.imshow(sing_face,cmap='gray')
plt.title("The first I_90 singular faces")
plt.show(block=False)
#Show the corresponding eigen faces
print("The first r=49 eigenfaces")
fig, axes = plt.subplots(7,7,figsize = (10,10))
k = 0
for i in range(7):
    for j in range(7):
        sing_face = u[:,k].reshape(19,19,order = 'F')
        axes[i][j].imshow(sing_face,cmap='gray')
        k = k + 1
```






```
The first r=49 eigenfaces
```



Part 2

Part 2-A

```
In [343...
def svd_init(U,S,V_T,r = 49):
    W = np.zeros((361,r))
    H = np.zeros((r,2429))
    V = V_T.T

    for i in range(r):

        uk = U[:,i].reshape(-1,1)
        vk_t = V[:,i].reshape(1,-1)

        uk_pos = np.maximum(uk,0)
        uk_neg = np.maximum(-1*uk,0)

        vk_t_pos = np.maximum(vk_t,0)
        vk_t_neg = np.maximum(-1*vk_t,0)

        Mk_plus = uk_pos @ vk_t_pos#(361,2429)
        Mk_minus = uk_neg @ vk_t_neg#(361,2429)

        if np.linalg.norm(Mk_plus,ord='fro') >= np.linalg.norm(Mk_minus,ord = 'fro'):
            W[:,i] = (uk_pos / np.linalg.norm(uk_pos)).reshape(361)
            H[i,:] = (S[i]*np.linalg.norm(uk_pos) * vk_t_pos).reshape(2429)
        else:
            W[:,i] = (uk_neg / np.linalg.norm(uk_neg)).reshape(361)
            H[i,:] = (S[i]*np.linalg.norm(uk_neg) * vk_t_neg).reshape(2429)

    return W,H
```

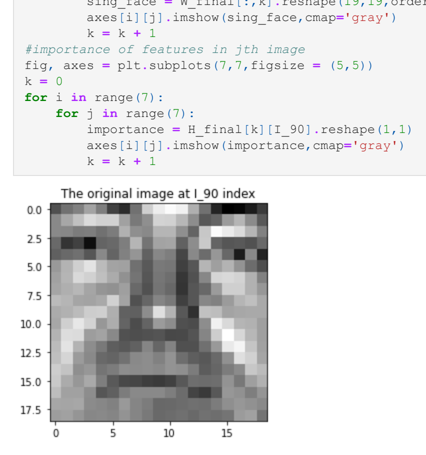
```
In [381...
def hals_update(X,H,W):
    W_new = np.copy(W)
    for l in range(W_new.shape[1]):
        norm_mat = H @ H[l,:]
        a = X @ H[l,:].T - W_new @ norm_mat + W_new[:,l] * (norm_mat[l])
        b = np.linalg.norm(H[l,:]) ** 2
        c = a/b
        c = np.maximum(0,c).reshape(W_new.shape[0])
        W_new[:,l] = c
    return W_new
```

```
In [422...
def coordinate_descent(X,W_0,H_0, delta = 0.01):

    W_prev = hals_update(X, H_0, W_0)
    H_prev = hals_update(X.T,W_prev.T,H_0.T).T

    stop = delta*np.linalg.norm(W_prev - W_0)
    stop2 = delta*np.linalg.norm(H_prev - H_0)

    norm = np.linalg.norm(X)
    W_new = np.zeros_like(W_prev)
    H_new = np.zeros_like(H_prev)
    err_acc = []

    curr_diff1 = np.linalg.norm(W_new - W_prev)
    curr_diff2 = np.linalg.norm(H_new - H_prev)
    i = 0
    print("Stops are " , stop,stop2)
    while (curr_diff1 > stop) and (curr_diff2 > stop2):

        W_new = hals_update(X,H_prev,W_prev)
        H_new = hals_update(X.T,W_new.T,H_prev.T).T

        curr_diff1 = np.linalg.norm(W_new - W_prev)
        curr_diff2 = np.linalg.norm(H_new - H_prev)

        err = np.linalg.norm( X - W_new @ H_new ) / norm
        err_acc.append(err)

        W_prev, H_prev = W_new, H_new

    return W_new,H_new,np.array(err_acc)
```

```
In [405...
W_0,H_0 = svd_init(u,s,vt)
W,H,err_accum = coordinate_descent(X_train, W_0, H_0)
```

```
Stops are  0.027064456027554293 171.75475517164548
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
```

```
In [406...
plt.figure()
plt.title("Error versus iteration")
plt.ylabel(r'$\frac{||X-WH||}{||X||}$')
plt.xlabel("iteration")
plt.plot(err_accum)
plt.show()
```

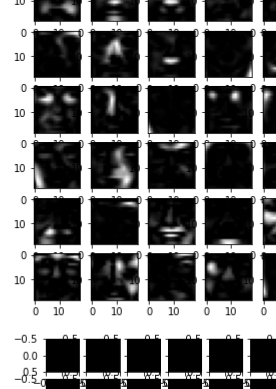


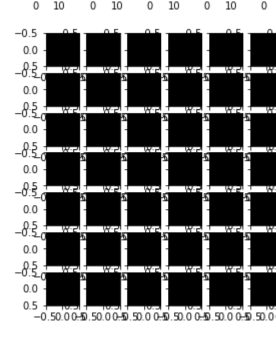

```
In [293...
print(np.all(W_prev == W_init))
```

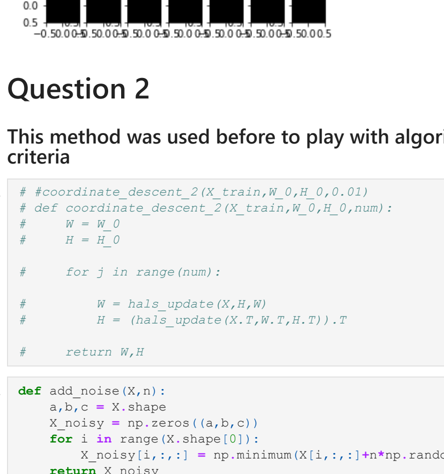
```
False
```

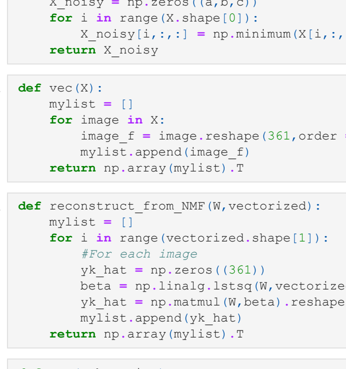
Part 2-B

```
In [14]:
plt.figure()
original_I90 = X_train[:,100].reshape(19,19,order = "F")
plt.imshow(original_I90,cmap='gray')
plt.title('The original image at I_90 index')
plt.show(block=False)

approx_image = W_final @ H_final[:,100]
approx_image = approx_image.reshape(19,19,order = 'F')
plt.imshow(approx_image,cmap='gray')
plt.title('The approximated image at I_90 index')
plt.show(block=False)

#facial features
fig, axes = plt.subplots(7,7,figsize = (7,7))
k = 0
for i in range(7):
    for j in range(7):
        sing_face = W_final[:,k].reshape(19,19,order = 'F')
        axes[i][j].imshow(sing_face,cmap='gray')
        k = k + 1
#importance of features in jth image
fig, axes = plt.subplots(7,7,figsize = (5,5))
k = 0
for i in range(7):
    for j in range(7):
        importance = H_final[k][I_90].reshape(1,1)
        axes[i][j].imshow(importance,cmap='gray')
        k = k + 1
```









Question 2

This method was used before to play with algorithm without stopping criteria

```
In [408...
# #coordinate_descent_2(X_train,W_0,H_0,0.01)
# def coordinate_descent_2(X_train,W_0,H_0,num):
#     W = W_0
#     H = H_0

#     for j in range(num):

#         W = hals_update(X,H,W)
#         H = (hals_update(X.T,W.T,H.T)).T

#     return W,H
```

```
In [409...
def add_noise(X,n):
    a,b,c = X.shape
    X_noisy = np.zeros((a,b,c))
    for i in range(X.shape[0]):
        X_noisy[i,:,:] = np.minimum(X[i,:,:]+n*np.random.rand(19,19),255)
    return X_noisy
```

```
In [410...
def vec(X):
    mylist = []
    for image in X:
        image_f = image.reshape(361,order = 'F')
        mylist.append(image_f)
    return np.array(mylist).T
```

```
In [411...
def reconstruct_from_NMF(W,vectorized):
    mylist = []
    for i in range(vectorized.shape[1]):
        #For each image
        yk_hat = np.zeros((361))
        beta = np.linalg.lstsq(W,vectorized[:,i],rcond = None)[0].reshape(-1,1)
        yk_hat = np.matmul(W,beta).reshape(361)
        mylist.append(yk_hat)
    return np.array(mylist).T
```

```
In [412...
def mat(y_hat,size):
    mylist = []
    for i in range(y_hat.shape[1]):
        mylist.append(y_hat[:,i].reshape(size,size,order = 'F'))
    return np.array(mylist)
```

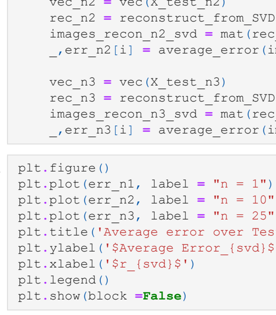
```
In [413...
def average_error(images_recon,X):
    #They should be same size 472,19,19
    err = np.zeros((472))
    for i in range(images_recon.shape[0]):
        err[i] = np.linalg.norm(X[i,:,:] - images_recon[i,:,:],ord = 'fro')
    return err, np.average(err)
```

```
In [414...
print(X_test.shape)
X_test = X_test.reshape(472,19,19,order = 'f')
print(X_test.shape)
```

```
(472, 19, 19)
(472, 19, 19)
```

```
In [415...
plt.figure()
plt.imshow(X_test[0],cmap='gray')
```

```
Out[415... <matplotlib.image.AxesImage at 0x1e3b21b00d0>
```

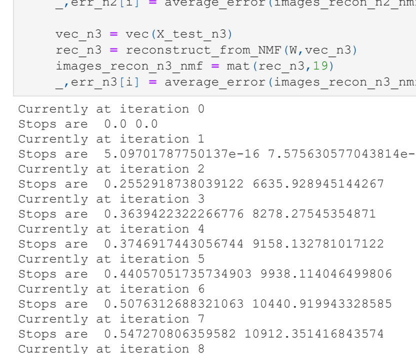


```
In [416...
#Constructing noisy version of X with given n parameter
X_test_n1 = add_noise(X_test,1)
X_test_n2 = add_noise(X_test,10)
X_test_n3 = add_noise(X_test,25)
U_train = u
```

```
In [417...
def reconstruct_from_SVD(U_train,vec,r_svd):
    mylist = []
    a = U_train[:,:r_svd]
    yk_svd = a @ a.T @ vec
    return yk_svd
```

```
In [418...
#Now we need to reconstruct to noisy version of the images for 1 <= r <= 100.
#Meaning that we want to observe the affect of adding new columns from the U matrix t
err_n1 = np.zeros((100))
err_n2 = np.zeros((100))
err_n3 = np.zeros((100))

for i in range(0,100):
    vec_n1 = vec(X_test_n1)
    rec_n1 = reconstruct_from_SVD(U_train,vec_n1,i+1)
    images_recon_n1_svd = mat(rec_n1,19)
    _,err_n1[i] = average_error(images_recon_n1_svd, X_test)

    vec_n2 = vec(X_test_n2)
    rec_n2 = reconstruct_from_SVD(U_train,vec_n2,i+1)
    images_recon_n2_svd = mat(rec_n2,19)
    _,err_n2[i] = average_error(images_recon_n2_svd, X_test)

    vec_n3 = vec(X_test_n3)
    rec_n3 = reconstruct_from_SVD(U_train,vec_n3,i+1)
    images_recon_n3_svd = mat(rec_n3,19)
    _,err_n3[i] = average_error(images_recon_n3_svd, X_test)
```

```
In [419...
plt.figure()
plt.plot(err_n1, label = "n = 1")
plt.plot(err_n2, label = "n = 10")
plt.plot(err_n3, label = "n = 25")
plt.title('Average error over Test Images by SVD')
plt.ylabel('$Average Error_{svd}$')
plt.xlabel('$r_{svd}$')
plt.legend()
plt.show(block =False)
```



Using NMF

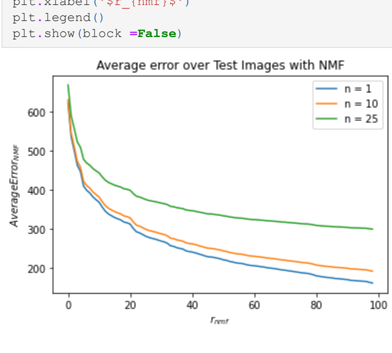
```
In [424...
#Now we can reconstruct by using NMF
err_n1 = np.zeros((100))
err_n2 = np.zeros((100))
err_n3 = np.zeros((100))

#coordinate_descent(X,W_0,H_0, delta = 0.01):
for i in range(0,100):
    print("Currently at iteration", i)

    W_0,H_0 = svd_init(u,s,vt,i)
    W,H,_ = coordinate_descent(X_train,W_0,H_0,1)

    vec_n1 = vec(X_test_n1)
    rec_n1 = reconstruct_from_NMF(W,vec_n1)
    images_recon_n1_nmf = mat(rec_n1,19)
    _,err_n1[i] = average_error(images_recon_n1_nmf, X_test)

    vec_n2 = vec(X_test_n2)
    rec_n2 = reconstruct_from_NMF(W,vec_n2)
    images_recon_n2_nmf = mat(rec_n2,19)
    _,err_n2[i] = average_error(images_recon_n2_nmf, X_test)

    vec_n3 = vec(X_test_n3)
    rec_n3 = reconstruct_from_NMF(W,vec_n3)
    images_recon_n3_nmf = mat(rec_n3,19)
    _,err_n3[i] = average_error(images_recon_n3_nmf, X_test)
```

```
Currently at iteration 0
Stops are  0.0 0.0
Currently at iteration 1
Stops are  5.09701787750137e-16 7.575630577043814e-11
Currently at iteration 2
Stops are  0.2552918738039122 6635.928945144267
Currently at iteration 3
Stops are  0.3639422322266776 8278.27545354871
Currently at iteration 4
Stops are  0.3746917443056744 9158.132781017122
Currently at iteration 5
Stops are  0.44057051735734903 9938.114046499806
Currently at iteration 6
Stops are  0.5076312688321063 10440.919943328585
Currently at iteration 7
Stops are  0.547270806359582 10912.351416843574
Currently at iteration 8
Stops are  0.5745150869442465 11283.483023967865
Currently at iteration 9
Stops are  0.6126969940776105 11567.166493542747
Currently at iteration 10
Stops are  0.6990111540977931 11835.624087131371
Currently at iteration 11
Stops are  0.770307281095543 12001.87091256069
Currently at iteration 12
Stops are  0.8605103209766303 12264.138935135741
Currently at iteration 13
Stops are  0.9157752219004726 12422.594199090108
Currently at iteration 14
Stops are  0.9633455278630187 12613.408374842149
Currently at iteration 15
Stops are  1.0234386473001613 12735.536947891847
Currently at iteration 16
Stops are  1.0741185695674724 12960.193064322233
Currently at iteration 17
Stops are  1.1260110651649455 13133.130671601431
Currently at iteration 18
Stops are  1.1759675077476937 13298.577730099027
Currently at iteration 19
Stops are  1.2115663728588253 13464.322401868874
Currently at iteration 20
Stops are  1.2420030187026483 13624.544024096143
Currently at iteration 21
Stops are  1.3095846786672858 13744.062037647536
Currently at iteration 22
Stops are  1.410164687229071 13947.10289372057
Currently at iteration 23
Stops are  1.446717990408205 14098.297377719913
Currently at iteration 24
Stops are  1.5267700042942116 14269.315974224752
Currently at iteration 25
Stops are  1.5721287608952952 14415.910883253347
Currently at iteration 26
```

```
Stops are  1.5985229249955453 14552.811103591697
Currently at iteration 27
Stops are  1.640905880608863 14686.435823150401
Currently at iteration 28
Stops are  1.7046294898846455 14798.361539953672
Currently at iteration 29
Stops are  1.732538700872032 14918.736069491362
Currently at iteration 30
Stops are  1.805427903623056 15112.369190770209
Currently at iteration 31
Stops are  1.8466712084963242 15236.71782173049
Currently at iteration 32
Stops are  1.8879634095952669 15368.214631231072
Currently at iteration 33
Stops are  1.9253699544561143 15520.305981673295
Currently at iteration 34
Stops are  1.9817671943158937 15657.293854536329
Currently at iteration 35
Stops are  2.0243119080994436 15760.94527616649
Currently at iteration 36
Stops are  2.062744454926657 15878.994059413328
Currently at iteration 37
Stops are  2.1026370656804896 15995.628082883331
Currently at iteration 38
Stops are  2.150723894313959 16110.18146159508
Currently at iteration 39
Stops are  2.184058836007961 16222.219216488713
Currently at iteration 40
Stops are  2.2266626645120673 16337.210804611701
Currently at iteration 41
Stops are  2.3231161318756897 16423.262047345353
Currently at iteration 42
Stops are  2.3662831344206987 16545.804900773026
Currently at iteration 43
Stops are  2.416644675147206 16644.41328326959
Currently at iteration 44
Stops are  2.467829045763378 16749.108977833683
Currently at iteration 45
Stops are  2.54520454189648 16822.538871502962
Currently at iteration 46
Stops are  2.571864176461001 16914.31803083819
Currently at iteration 47
Stops are  2.609052436561159 17009.050353293558
Currently at iteration 48
Stops are  2.6531299261566734 17099.188431752158
Currently at iteration 49
Stops are  2.706445602755429 17175.47551716455
Currently at iteration 50
Stops are  2.755085747213815 17268.915749556403
Currently at iteration 51
Stops are  2.8078682961367547 17363.93669385115
Currently at iteration 52
Stops are  2.837813920944978 17455.39181225748
Currently at iteration 53
Stops are  2.9099896291470992 17552.464373663668
Currently at iteration 54
Stops are  2.973854837267441 17631.57740298663
Currently at iteration 55
Stops are  3.0019648081203574 17716.266102211484
Currently at iteration 56
Stops are  3.0324268152975544 17795.43370770398
Currently at iteration 57
Stops are  3.067203035284717 17884.651746745112
Currently at iteration 58
Stops are  3.1022107227847853 17965.5511124008
Currently at iteration 59
Stops are  3.1417436602590114 18050.740170722383
Currently at iteration 60
Stops are  3.1837171114394556 18148.228745649685
Currently at iteration 61
Stops are  3.2329310523620425 18234.173586558547
Currently at iteration 62
Stops are  3.2939381854891394 18336.10364865252
Currently at iteration 63
Stops are  3.3748565343675874 18400.505562623042
Currently at iteration 64
Stops are  3.4107244656833884 18470.967991199894
Currently at iteration 65
Stops are  3.473113020006286 18567.40188526997
Currently at iteration 66
Stops are  3.506848297025616 18643.378355417106
Currently at iteration 67
Stops are  3.54609589012316 18720.609049740222
Currently at iteration 68
Stops are  3.5718613351047352 18790.437761717763
Currently at iteration 69
Stops are  3.602502985263081 18868.955954188084
Currently at iteration 70
Stops are  3.632164869889736 18951.741133801155
Currently at iteration 71
Stops are  3.6722560240788322 19027.84795222617
Currently at iteration 72
Stops are  3.7719410993819436 19085.93410106769
Currently at iteration 73
Stops are  3.854458182171882 19139.713268653515
Currently at iteration 74
Stops are  3.9149469055281263 19246.234615092068
Currently at iteration 75
Stops are  3.9561091105399204 19319.51667986214
Currently at iteration 76
Stops are  4.005957520824947 19427.8603560365
Currently at iteration 77
Stops are  4.038598470032724 19496.459393228844
Currently at iteration 78
Stops are  4.069161318999478 19558.893657731303
Currently at iteration 79
Stops are  4.1976900068881084 19606.668014577233
Currently at iteration 80
Stops are  4.230785288167186 19683.293778174004
Currently at iteration 81
Stops are  4.277112006095669 19752.499041333365
Currently at iteration 82
Stops are  4.299670438694955 19816.56146300458
Currently at iteration 83
Stops are  4.339414467066677 19888.27684116449
Currently at iteration 84
Stops are  4.367736433966448 19950.026536279453
Currently at iteration 85
Stops are  4.49878119042716 19986.227197081484
Currently at iteration 86
Stops are  4.556821545759329 20079.762405562153
Currently at iteration 87
Stops are  4.5921596481611 20145.32491077644
Currently at iteration 88
Stops are  4.649640388985925 20259.01129973603
Currently at iteration 89
Stops are  4.690889849879015 20323.191204136594
Currently at iteration 90
Stops are  4.720844324786472 20372.38261152567
Currently at iteration 91
Stops are  4.779410314476011 20433.635691943902
Currently at iteration 92
Stops are  4.811999958015311 20507.534449673516
Currently at iteration 93
Stops are  4.8520961277507775 20568.93311088568
Currently at iteration 94
Stops are  4.900875890399595 20662.14044597346
Currently at iteration 95
Stops are  4.9317280696862635 20719.831485823277
Currently at iteration 96
Stops are  5.113499419461251 20766.356228140132
Currently at iteration 97
Stops are  5.170878598951927 20821.282730630686
Currently at iteration 98
Stops are  5.210467969030274 20879.929663486004
Currently at iteration 99
Stops are  5.239628093057101 20939.46041718373
```

In [425…

```python
plt.figure()
plt.plot(err_n1[1:], label = "n = 1")
plt.plot(err_n2[1:], label = "n = 10")
plt.plot(err_n3[1:], label = "n = 25")
plt.title('Average error over Test Images with NMF')
plt.ylabel('$Average Error_{NMF}$')
plt.xlabel('$r_{nmf}$')
plt.legend()
plt.show(block =False)
```



Question 3

In [426…

```python
X_test = test_data
X_test = X_test.reshape(472,19,19,order = 'f')
```

Reconstruction using SVD

In [427…

```python
test_images = X_test
S = np.ones(test_images[0].shape)
for i in range(S.shape[0]):
    for j in range(11, S.shape[1]):
        S[i, j] -= 0.05 * (j - 10)

masked_test_images = [np.multiply(image, S) for image in test_images]

vectorized_masked_test_images = np.array(
        [image.flatten(order = 'f') for image in masked_test_images]).T

print(vectorized_masked_test_images.shape)
```

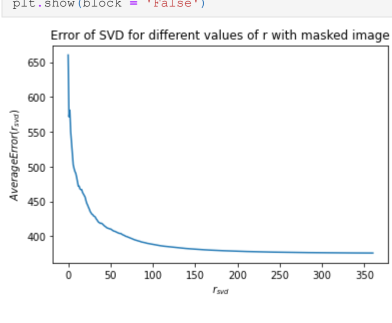
```
(361, 472)
```

In [428…

```python
#Reconstruction using SVD
error_mask = np.zeros((361))
for i in range(0,361):
    y_hat = reconstruct_from_SVD(U_train,vectorized_masked_test_images,i+1)
    mask_recon = mat(y_hat,19)
    _,error_mask[i] = average_error(mask_recon, X_test)
```

In [429…

```python
plt.figure()
plt.plot(error_mask)
plt.title('Error of SVD for different values of r with masked image')
plt.ylabel('$AverageError(r_{svd})$')
plt.xlabel('$r_{svd}$')
plt.show(block = 'False')
```



Reconstruction using NMF

In [430…

```python
np.amin(error_mask)
```

Out[430…

```
376.0963351122951
```

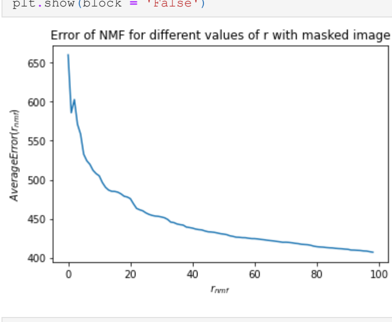
In [431…

```python
#Reconstruction using NMF
error_mask = np.zeros((100))

for i in range(0,100):
    print(i)
    W_0,H_0 = svd_init(u,s,vt,i)
    W,H,_ = coordinate_descent(X_train,W_0,H_0,1)

    rec_masked = reconstruct_from_NMF(W,vectorized_masked_test_images)
    images_recon_masked_nmf = mat(rec_masked,19)
    _,error_mask[i] = average_error(images_recon_masked_nmf, X_test)
```

```
0
Stops are  0.0 0.0
1
Stops are  5.09701787750137e-16 7.575630577043814e-11
2
Stops are  0.2552918738039122 6635.928945144267
3
Stops are  0.3639422322266776 8278.27545354871
4
Stops are  0.3746917443056744 9158.132781017122
5
Stops are  0.44057051735734903 9938.114046499806
6
Stops are  0.5076312688321063 10440.919943328585
7
Stops are  0.547270806359582 10912.351416843574
8
Stops are  0.5745150869442465 11283.483023967865
9
Stops are  0.6126969940776105 11567.166493542747
10
Stops are  0.6990111540977931 11835.624087131371
11
Stops are  0.770307281095543 12001.87091256069
12
Stops are  0.8605103209766303 12264.138935135741
13
Stops are  0.9157752219004726 12422.594199090108
14
Stops are  0.9633455278630187 12613.408374842149
15
Stops are  1.0234386473001613 12735.536947891847
16
Stops are  1.0741185695674724 12960.193064322233
17
Stops are  1.1260110651649455 13133.130671601431
18
Stops are  1.1759675077476937 13298.577730099027
19
Stops are  1.2115663728588253 13464.322401868874
20
Stops are  1.2420030187026483 13624.544024096143
21
Stops are  1.3095846786672858 13744.062037647536
22
Stops are  1.410164687229071 13947.10289372057
23
Stops are  1.446717990408205 14098.297377719913
24
Stops are  1.5267700042942116 14269.315974224752
25
Stops are  1.5721287608952952 14415.910883253347
26
Stops are  1.5985229249955453 14552.811103591697
27
Stops are  1.640905880608863 14686.435823150401
28
Stops are  1.7046294898846455 14798.361539953672
29
Stops are  1.732538700872032 14918.736069491362
30
Stops are  1.805427903623056 15112.369190770209
31
Stops are  1.8466712084963242 15236.71782173049
32
Stops are  1.8879634095952669 15368.214631231072
33
Stops are  1.9253699544561143 15520.305981673295
34
Stops are  1.9817671943158937 15657.293854536329
35
Stops are  2.0243119080994436 15760.94527616649
36
Stops are  2.062744454926657 15878.994059413328
37
Stops are  2.1026370656804896 15995.628082883331
38
Stops are  2.150723894313959 16110.18146159508
39
Stops are  2.184058836007961 16222.219216488713
40
Stops are  2.2266626645120673 16337.210804611701
41
Stops are  2.3231161318756897 16423.262047345353
42
Stops are  2.3662831344206987 16545.804900773026
43
Stops are  2.416644675147206 16644.41328326959
44
Stops are  2.467829045763378 16749.108977833683
45
Stops are  2.54520454189648 16822.538871502962
46
Stops are  2.571864176461001 16914.31803083819
47
Stops are  2.609052436561159 17009.050353293558
48
Stops are  2.6531299261566734 17099.188431752158
49
Stops are  2.706445602755429 17175.47551716455
50
Stops are  2.755085747213815 17268.915749556403
51
Stops are  2.8078682961367547 17363.93669385115
52
Stops are  2.837813920944978 17455.39181225748
53
Stops are  2.9099896291470992 17552.464373663668
54
Stops are  2.973854837267441 17631.57740298663
55
Stops are  3.0019648081203574 17716.266102211484
56
Stops are  3.0324268152975544 17795.43370770398
57
Stops are  3.067203035284717 17884.651746745112
58
Stops are  3.1022107227847853 17965.5511124008
59
Stops are  3.1417436602590114 18050.740170722383
60
Stops are  3.1837171114394556 18148.228745649685
61
Stops are  3.2329310523620425 18234.173586558547
62
Stops are  3.2939381854891394 18336.10364865252
63
Stops are  3.3748565343675874 18400.505562623042
64
Stops are  3.4107244656833884 18470.967991199894
65
Stops are  3.473113020006286 18567.40188526997
66
Stops are  3.506848297025616 18643.378355417106
67
Stops are  3.54609589012316 18720.609049740222
68
Stops are  3.5718613351047352 18790.437761717763
69
Stops are  3.602502985263081 18868.955954188084
70
Stops are  3.632164869889736 18951.741133801155
71
Stops are  3.6722560240788322 19027.84795222617
72
Stops are  3.7719410993819436 19085.93410106769
73
Stops are  3.854458182171882 19139.713268653515
74
Stops are  3.9149469055281263 19246.234615092068
75
Stops are  3.9561091105399204 19319.51667986214
76
Stops are  4.005957520824947 19427.8603560365
77
Stops are  4.038598470032724 19496.459393228844
78
Stops are  4.069161318999478 19558.893657731303
79
Stops are  4.1976900068881084 19606.668014577233
80
Stops are  4.230785288167186 19683.293778174004
81
Stops are  4.277112006095669 19752.499041333365
82
Stops are  4.299670438694955 19816.56146300458
83
Stops are  4.339414467066677 19888.27684116449
84
Stops are  4.367736433966448 19950.026536279453
85
Stops are  4.49878119042716 19986.227197081484
86
Stops are  4.556821545759329 20079.762405562153
87
Stops are  4.5921596481611 20145.32491077644
88
Stops are  4.649640388985925 20259.01129973603
89
Stops are  4.690889849879015 20323.191204136594
90
Stops are  4.720844324786472 20372.38261152567
91
Stops are  4.779410314476011 20433.635691943902
92
Stops are  4.811999958015311 20507.534449673516
93
Stops are  4.8520961277507775 20568.93311088568
94
Stops are  4.900875890399595 20662.14044597346
95
Stops are  4.9317280696862635 20719.831485823277
96
Stops are  5.113499419461251 20766.356228140132
97
Stops are  5.170878598951927 20821.282730630686
98
Stops are  5.210467969030274 20879.929663486004
99
Stops are  5.239628093057101 20939.46041718373
```

In [434…

```python
plt.figure()
plt.plot(error_mask[1:])
plt.title('Error of NMF for different values of r with masked image')
plt.ylabel('$AverageError(r_{nmf})$')
plt.xlabel('$r_{nmf}$')
plt.show(block = 'False')
```

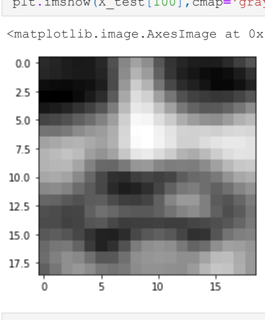


In [435…

```python
plt.figure()
plt.imshow(X_test[100],cmap='gray')
```

Out[435…

```
<matplotlib.image.AxesImage at 0x1e3b21e7e80>
```



In [ ]: