Foundations of Data Science - Homework 1 Solution

Mohammad Afzal Shadab (ms82697)

mashadab@utexas.edu

Due Monday, February 8, 2021

Notes on the homework:

- This assignment is due by 11:59pm on February 8, 2021
- You are encouraged to discuss the homework problems with each other, but solutions and code must be written by yourself. Copying homework solutions from another student, webpage, or textbook is not allowed.
- You are required to typeset your solutions and submit them through Gradescope. You may find the .tex file for Homework 1 under Files in Canvas, and type your answers to the questions directly there. Also under Files in Canvas, you will find instructions on how to submit homework through Gradescope.

Exercises adapted from textbook (Ch. 2, *Best-Fit Subspaces and Singular Value Decomposition*)

1. **Let $\boldsymbol{A}$ be a square $n \times n$ matrix whose rows are orthonormal. Prove that the columns of $\boldsymbol{A}$ are orthonormal.**
   We know that column vector of a matrix $\mathbf{M}$ form an orthonormal set if and only if $\mathbf{M}^T = \mathbf{M}^{-1}$ or $\mathbf{M}^T\mathbf{M} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. So, let $\boldsymbol{A}^T = \mathbf{B}$, then the orthonormal rows of $\boldsymbol{A}$ become the orthonormal columns of $\mathbf{B}$. Now,

$$\begin{aligned} \mathbf{B}^T\mathbf{B} &= \mathbf{I} && (0.1)\\ (\boldsymbol{A}^T)^T\boldsymbol{A}^T &= \mathbf{I} && (0.2)\\ \boldsymbol{A}\boldsymbol{A}^T &= \mathbf{I} && (0.3)\\ \boldsymbol{A}^T &= \boldsymbol{A}^{-1}, \quad \boldsymbol{A} \text{ is full rank.} && (0.4) \end{aligned}$$

   Therefore, the columns of $\boldsymbol{A}$ are also orthonormal. □

2. **Suppose $\boldsymbol{A}$ is an $n \times n$ matrix with block diagonal structure with k equal size blocks where all entries of the ith block are a_i with $a_1 > a_2 > \cdots > a_k > 0$. Show that $\boldsymbol{A}$ has exactly k non-zero singular vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ where $\mathbf{v}_i$ has the value $(k/n)^{1/2}$ in the coordinates corresponding to the i^{th} block and 0 elsewhere. In other words, the singular vectors exactly identify the blocks of the diagonal.** Since $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is square and symmetric,

$$\boldsymbol{A} = \mathbf{V}\Sigma\mathbf{V}^T \tag{0.5}$$

   where $\mathbf{U} = \mathbf{V}$ and singular vectors become eigen vectors. Let $\boldsymbol{A}_i \in \mathbb{R}^{\frac{n}{k}\times\frac{n}{k}}$ be the i-th block, $\frac{n}{k}$ by $\frac{n}{k}$ in structure with same entries a_i. Then, the eigen values of $\mathbf{A}_i$ will be,

$$\lambda_i = \frac{n}{k}a_i \tag{0.6}$$

   and the eigen vectors are $\mathbf{v}_i = (1, 1, \ldots, 1)^T$. After normalizing,

$$\mathbf{v}_i = \left(\sqrt{\frac{n}{k}}, \sqrt{\frac{n}{k}}, \cdots, \sqrt{\frac{n}{k}}\right)^T \tag{0.7}$$

   So, the structure of $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 & & \\ & \ddots & \\ & & \boldsymbol{A}_k \end{bmatrix}$ and then the SVD takes the form,

$$\begin{aligned} \boldsymbol{A} &= \begin{bmatrix} \mathbf{U}_1\mathbf{D}_1\mathbf{V}_1^T & & \\ & \ddots & \\ & & \mathbf{U}_k\mathbf{D}_k\mathbf{V}_k^T \end{bmatrix} && (0.8)\\ &= \begin{bmatrix} \mathbf{U}_1 & & \\ & \ddots & \\ & & \mathbf{U}_k \end{bmatrix} \begin{bmatrix} \mathbf{D}_1 & & \\ & \ddots & \\ & & \mathbf{D}_k \end{bmatrix} \begin{bmatrix} \mathbf{V}_1^T & & \\ & \ddots & \\ & & \mathbf{V}_k^T \end{bmatrix} && (0.9)\\ &= \mathbf{U}_{n\times n}\Sigma_{n\times n}\mathbf{V}_{n\times n}^T \quad \square && (0.10) \end{aligned}$$

3. **Suppose $\boldsymbol{A}$ is a square invertible matrix with SVD $\boldsymbol{A} = \sum_i \sigma_i \mathbf{u}_i \mathbf{v}_i^T$. Prove that the inverse of $\boldsymbol{A}$ is $\sum_i \frac{1}{\sigma_i} \mathbf{v}_i \mathbf{u}_i^T$.**

Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ be invertible and the singular vectors be $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n]$ and $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n]$, and the diagonal matrix be, $\Sigma = diag(\sigma_1, \sigma_2, \ldots, \sigma_n)$ Then

$$\boldsymbol{A} = \sum_{i=1}^{n} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \tag{0.11}$$

$$\Rightarrow \boldsymbol{A} = \mathbf{U}\Sigma\mathbf{V}^T \tag{0.12}$$

$$\Rightarrow \boldsymbol{A}^{-1} = [\mathbf{U}\Sigma\mathbf{V}^T]^{-1} = \mathbf{V}^{-T}\Sigma^{-1}\mathbf{U}^{-1} \tag{0.13}$$

$$\Rightarrow \boldsymbol{A}^{-1} = \mathbf{V}\Sigma^{-1}\mathbf{U}^T, \quad \text{Since } \mathbf{U} \text{ and } \mathbf{V} \text{ are unitary} \tag{0.14}$$

$$\Rightarrow \boldsymbol{A}^{-1} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n] diag(\sigma_1^{-1}, \sigma_2^{-1}, \ldots, \sigma_n^{-1}) \begin{bmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_n^T \end{bmatrix} \tag{0.15}$$

$$\Rightarrow \boldsymbol{A}^{-1} = [\sigma_1^{-1}\mathbf{v}_1, \sigma_2^{-1}\mathbf{v}_2, \ldots, \sigma_n^{-1}\mathbf{v}_n] \begin{bmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_n^T \end{bmatrix} \tag{0.16}$$

$$\Rightarrow \boldsymbol{A}^{-1} = \sigma_1^{-1}\mathbf{v}_1\mathbf{u}_1^T + \sigma_2^{-1}\mathbf{v}_2\mathbf{u}_2^T + \ldots + \sigma_n^{-1}\mathbf{v}_n\mathbf{u}_n^T \tag{0.17}$$

$$\Rightarrow \boldsymbol{A}^{-1} = \sum_{i=1}^{n} \frac{1}{\sigma_i} \mathbf{v}_i \mathbf{u}_i^T \quad \square \tag{0.18}$$

4. (a) **Write a program to implement the power method for computing the first singular vector of a matrix. Apply your program to the matrix**

$$\boldsymbol{A} = \begin{pmatrix} 1 & 2 & 3 & \ldots & 9 & 10 \\ 2 & 3 & 4 & \ldots & 10 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 9 & 10 & 0 & \ldots & 0 & 0 \\ 10 & 0 & 0 & \ldots & 0 & 0 \end{pmatrix}$$

Code:

```
#class    : Foundations of ML and Data Science
#homework: 1
#question: 4(a)
#author   : Mohammad Afzal Shadab
#email    : mashadab@utexas.edu

#Power method
import numpy as np

tol     = 1e-14     #tolerance for power method
max_iter= 5000      #maximum number of iterations
n       = 10        #size
A       = np.zeros((n,n)) #placeholder for A matrix

for i in range(0,n):
```

```
    for j in range(0,n):
        if i + j < n :
            A[i,j] = i + j + 1

print(A)

v       = np.random.rand(n,1) #randomly choosing a vector v
v       = v/np.linalg.norm(v) #normalize v
v_old   = np.ones_like(v)     #placeholder for old vector v

i       = 0

#power method
while (np.linalg.norm(v-v_old)/np.linalg.norm(v))>tol and
                                    i < max_iter: #relative
                                    error
    v_old = np.copy(v)
    v     = A @ v_old               #finding new vector
    v     = v/np.linalg.norm(v)     #normalize
    lmbda = (np.matrix.transpose(v) @ (A @ v))[0,0]
    i = i + 1
print(i,lmbda,v)
```

Output:
iterations= 53
$\sigma_1 = 43.43043275068864$
$\mathbf{v}_1 = (0.3198, 0.3696, 0.3981, 0.4039, 0.3873, 0.3500, 0.2951, 0.2272, 0.1514, 0.0736)^T$
The signs of the singular vectors can be the exact opposite because $\boldsymbol{A}$ is real and symmetric.

(b) **Modify the power method to find the first *four* singular vectors of a matrix $\boldsymbol{A}$ as follows. Randomly select four vectors and find an orthonormal basis for the space spanned by the four vectors. Then multiply each of the basis vectors times $\boldsymbol{A}$ and find a new orthonormal basis for the space spanned by the resulting four vectors. Apply your method to find the first four singular vectors of a matrix $\boldsymbol{A}$ from part 1. (In Matlab, the command "orth" finds an orthonormal basis for the space spanned by a set of vectors). Code**:

```
#class    : Foundations of Data Science
#homework: 1
#question: 4(b)
#author   : Mohammad Afzal Shadab
#email    : mashadab@utexas.edu

#Block Power method
import numpy as np
from scipy.linalg import orth

tol     = 1e-14     #tolerance for power method
max_iter= 5000      #maximum number of iterations
n       = 10        #size
A       = np.zeros((n,n)) #placeholder for A matrix
s       = 4         #number of singular vectors

for i in range(0,n):
    for j in range(0,n):
```

```
        if i + j < n :
            A[i,j] = i + j + 1

print('A: \n',A,'\n')

v       = np.random.rand(n,s) #randomly choosing a vector v
err     = 1.0
i       = 0

#block power method
while err > tol and i < max_iter:  #error comparison
    v_old = orth(v)
    v     = A @ v_old       #finding new matrix B =Av
    err   = np.linalg.norm(v - v_old)
    i = i + 1

v     = v/np.linalg.norm(v, ord=2, axis=0, keepdims=True)
                                  #normalize each column

print('Iteration: ',i,'\n','V: \n',v)
```

Output:

$$\mathbf{V} = \begin{pmatrix} -0.3197506 & -0.45784552 & -0.42415456 & -0.39363387 \\ -0.36962502 & -0.3936509 & -0.24288284 & -0.02849182 \\ -0.39811309 & -0.25497036 & 0.07043602 & 0.36152104 \\ -0.4039189 & -0.06980555 & 0.33936334 & 0.38322644 \\ -0.38728043 & 0.12450888 & 0.41233765 & 0.014406 \\ -0.3499587 & 0.2887672 & 0.24775341 & -0.37382129 \\ -0.29512626 & 0.38972804 & -0.06268814 & -0.39083196 \\ -0.22716239 & 0.40675711 & -0.34546459 & -0.01997829 \\ -0.15136864 & 0.33594883 & -0.44265159 & 0.36463802 \\ -0.07362363 & 0.19090282 & -0.30058613 & 0.3749948 \end{pmatrix}$$

(c) **(Bonus question) Modify the power method from part (b) to compute the leading r singular values along with the leading r singular vectors. Verify numerically that the algorithm works on the matrix A from part (a).**

Code:

```
#class    : Foundations of Data Science
#homework: 1
#question: 4(c)
#author   : Mohammad Afzal Shadab
#email    : mashadab@utexas.edu

#Block Power method
import numpy as np
from scipy.linalg import orth

tol     = 1e-15     #tolerance for power method
max_iter= 5000      #maximum number of iterations
n       = 10        #size
A       = np.zeros((n,n)) #placeholder for A matrix
s       = 8         #number of singular vectors

for i in range(0,n):
    for j in range(0,n):
        if i + j < n :
```

```
            A[i,j] = i + j + 1

print('A: \n',A,'\n')

v       = np.random.rand(n,s) #randomly choosing a vector v
err     = 1.0
i       = 0

#block power method
while err > tol and i < max_iter:  #error comparison
    v_old = orth(v)
    v     = A @ v_old       #finding new matrix B =Av
    err   = np.linalg.norm(v - v_old)
    i = i + 1

v     = v/np.linalg.norm(v, ord=2, axis=0, keepdims=True) #
                                normalize each column

lmbda = np.matrix.transpose(v) @ (A @ v)                  #
                                finding Rayleigh-quotients

for i in range(0,s):
    if lmbda[i,i] < 0.0: #for negative singular values, change
                                    the sign
        lmbda[i,i] = np.abs(lmbda[i,i])
        v[:,i]     = -v[:,i]

print('Iteration: ',i,'\n','V: \n',v)
print('Singular values', np.diag(lmbda))

#Verify
U,D,VT =  np.linalg.svd(A)
V_direct = np.transpose(VT[0:s,:])

print('Frobenius norm of error in the V matrices', np.linalg.
                                norm(np.absolute(v) - np.
                                absolute(V_direct)))
```

Output:
For rank - 8 approximation ($s = 8$)

$$(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4) = \begin{pmatrix} -0.3197506 & -0.45784552 & -0.42415456 & -0.39363387 \\ -0.36962502 & -0.3936509 & -0.24288284 & -0.02849182 \\ -0.39811309 & -0.25497036 & 0.07043602 & 0.36152104 \\ -0.4039189 & -0.06980555 & 0.33936334 & 0.38322644 \\ -0.38728043 & 0.12450888 & 0.41233765 & 0.014406 \\ -0.3499587 & 0.2887672 & 0.24775341 & -0.37382129 \\ -0.29512626 & 0.38972804 & -0.06268814 & -0.39083196 \\ -0.22716239 & 0.40675711 & -0.34546459 & -0.01997829 \\ -0.15136864 & 0.33594883 & -0.44265159 & 0.36463802 \\ -0.07362363 & 0.19090282 & -0.30058613 & 0.3749948 \end{pmatrix}$$

$$(\mathbf{v}_5, \mathbf{v}_6, \mathbf{v}_7, \mathbf{v}_8) = \begin{pmatrix} -0.35928017 & -0.30894468 & -0.25879154 & -0.19715914 \\ 0.16223358 & 0.33039178 & 0.42025921 & 0.42611629 \\ 0.42525922 & 0.25466734 & -0.05410832 & -0.33720346 \\ 0.01618867 & -0.37319199 & -0.38321484 & -0.0069124 \\ -0.4228252 & -0.20038893 & 0.33382114 & 0.34352558 \\ -0.20263225 & 0.40034053 & 0.13317656 & -0.42484112 \\ 0.33032268 & 0.13400429 & -0.43609885 & 0.18453022 \\ 0.33822599 & -0.42582275 & 0.18443889 & 0.19124493 \\ -0.19198658 & -0.07310871 & 0.29510228 & -0.42680718 \\ -0.42811309 & 0.43152222 & -0.40763954 & 0.33839183 \end{pmatrix}$$

Singular values $= (43.43043, 23.98317, 14.11092, 10.49705, 8.39218, 7.15942, 6.34854, 5.82636)$

$||\mathbf{V} - \mathbf{V}_{SVD}||_F = 1.0990222843105398e - 14$

The small difference in the Frobenius norm of singular vectors from the present algorithm w.r.t. the direct SVD numerically verifies the algorithm.

5. **Read in a grayscale image of your choice (of resolution at least 256 by 256). Perform a singular value decomposition of the matrix. Reconstruct the image using only 1, 4, 16, and 32 singular values.**

   (a) **Plot the original image along with the reconstructed images. How good is the quality of the reconstructed photo?**

Figure 1: For part 4(a), Top: Original image used for SVD based compression. Reconstructed image with 1 mode (middle) and 4 modes (bottom).

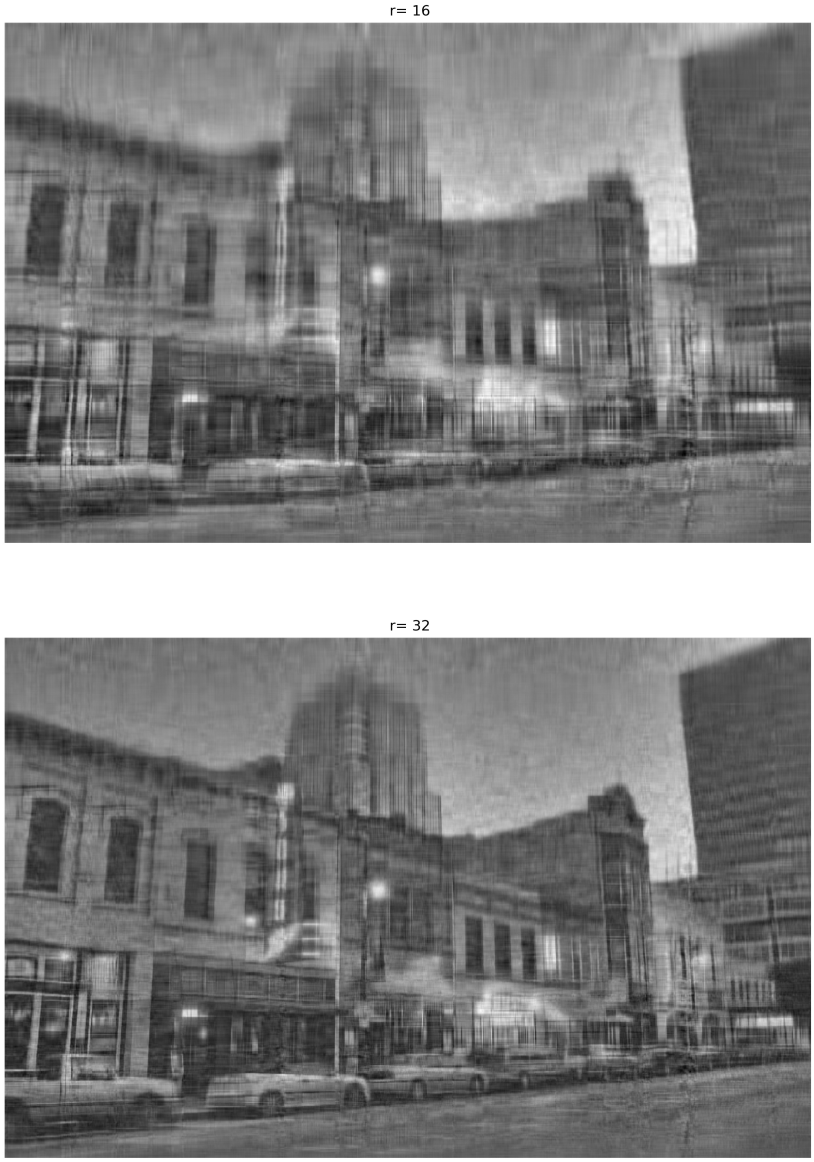

Figure 2: For part 4(a), reconstructed image with 16 (top) & 32 modes (bottom).

The quality of the reconstructed image is best for 32 modes but that is still insufficient to get a clear picture. From the following plot 3 of cumulative sum ratio $\sum_{i=1}^{s} \sigma_i / \sum_{j=1}^{n} \sigma_j$ of singular values with respect to modes, it can be observed that atleast 200 modes are required to obtain around 60 % of the energy.

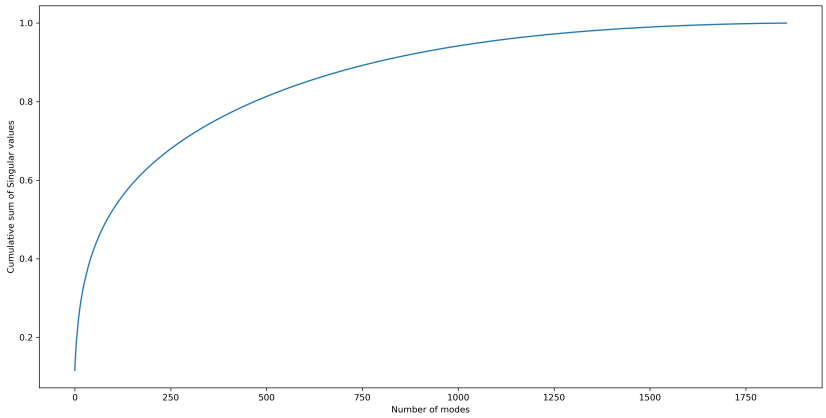

Figure 3: The ratio of cumulative sum $\sum_{i=1}^{s} \sigma_i / \sum_{j=1}^{n} \sigma_j$ of singular values with respect to the sum of all singular values of the image.

(b) **What percent of the Frobenius norm is captured in each case? (If you use Matlab, the command to read a photo is "imread". The command "imread" will read the file in unit8 and you need to convert to double for the SVD code. The types of files that can be read are given by "imformats". To plot the file, use "imshow")**

Mode(s)	**Frobenius norm (%)**
1	89.48
4	93.22
16	96.75
32	97.94

Table 1: Percentage ratio of Frobenius norms of the reconstructed images with respect to the original image.

Since the image is not centered as done in principal component analysis, the first mode has very high norm percentage.

(c) **Repeat steps (a),(b) above, but now for a "white noise" image of the same dimensions, generated as follows: each pixel is generated as an independent uniform random variable on** $[0, 1]$**. In Matlab, an** $d \times n$ **matrix of i.i.d. uniform random variables can be created using the command rand**(d, n)**. Compare the behavior of the singular values for the noise image and the real image.**

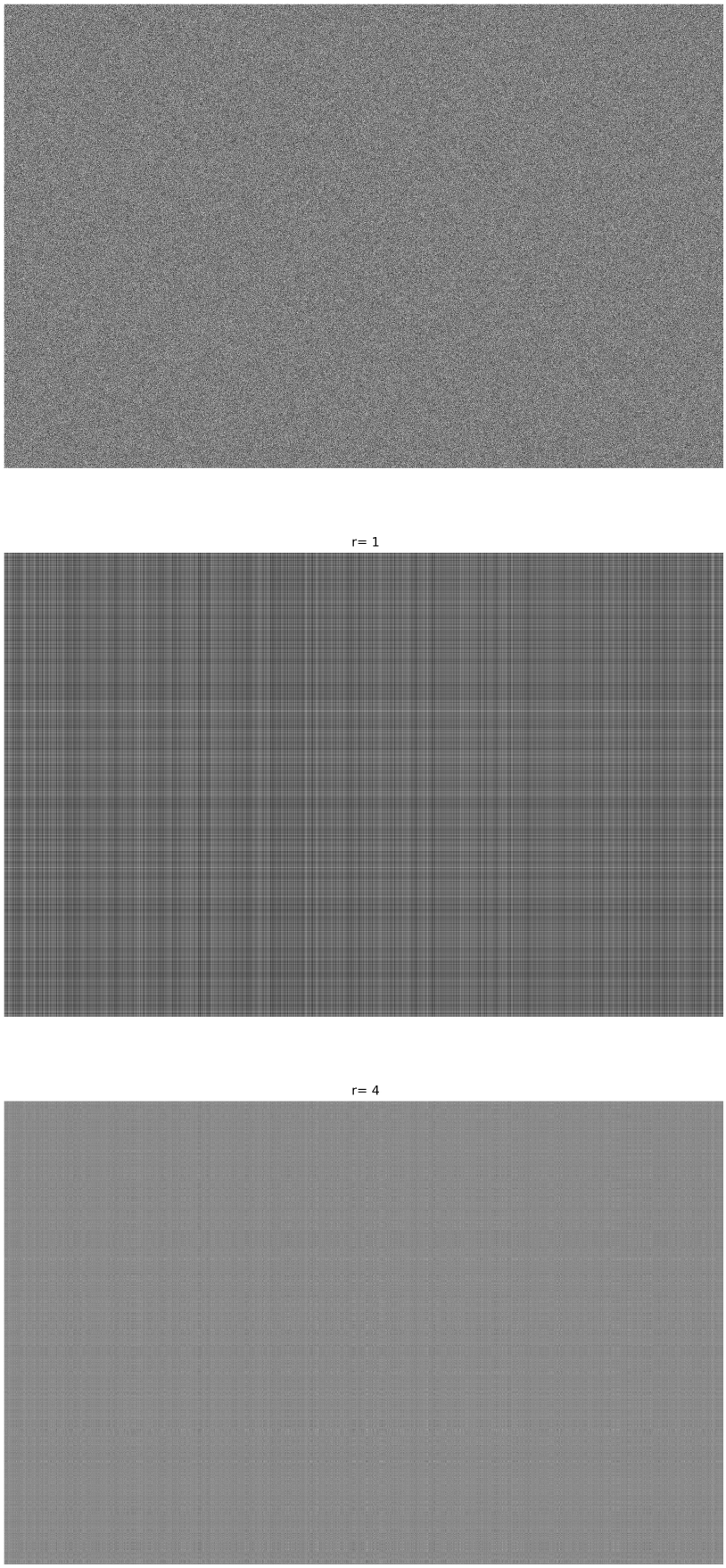

Figure 4: For part 4(c), Top: Original image with white noise used for SVD based compression. Reconstructed image with 1 mode (middle) and 4 modes (bottom).

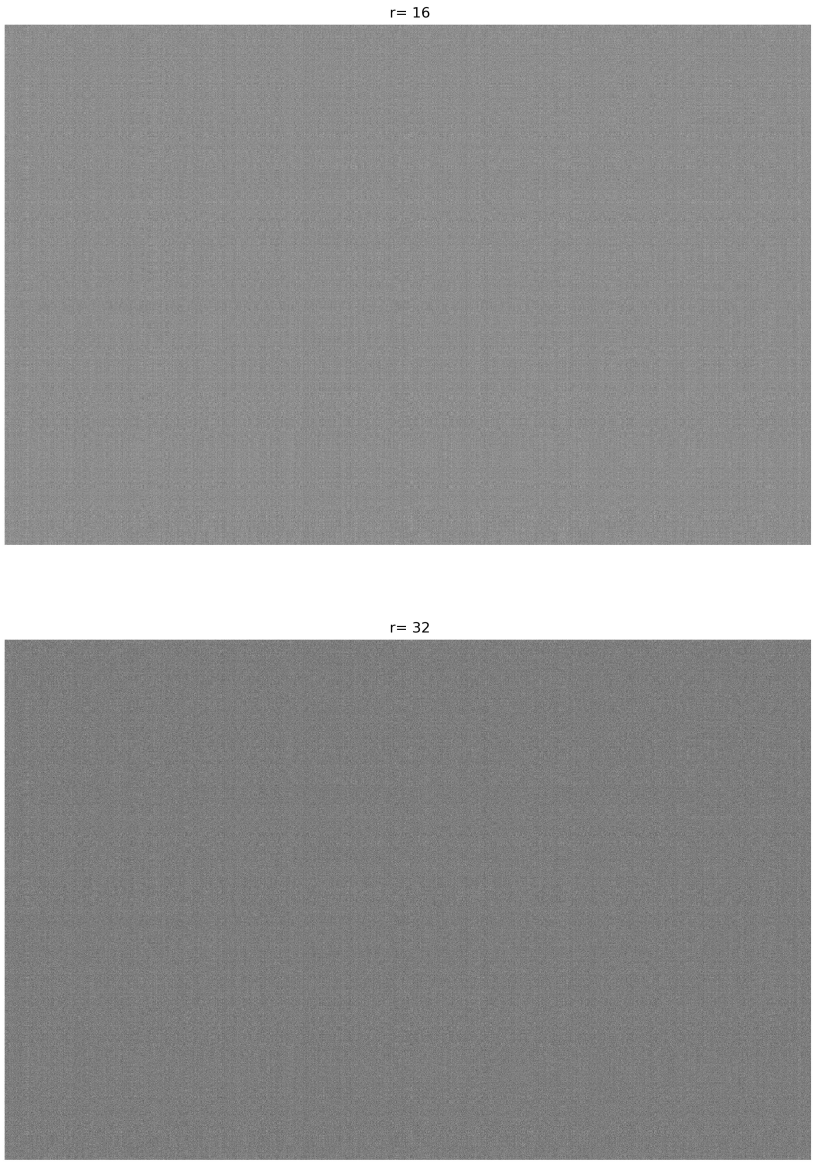

Figure 5: For part 4(c), reconstructed image with 16 (top) & 32 modes (bottom).

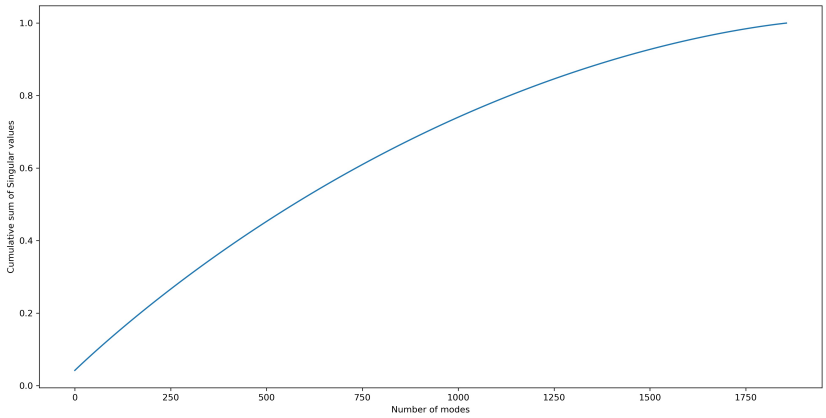

Figure 6: The ratio of cumulative sum $\sum_{i=1}^{s} \sigma_i / \sum_{j=1}^{n} \sigma_j$ of singular values with respect to the sum of all singular values for white noise image.

Mode(s)	**Frobenius norm (%)**
1	86.61
4	86.69
16	86.98
32	87.35

Table 2: Percentage ratio of Frobenius norms of the reconstructed images with respect to the original white noise image. Higher value for the first mode is due to non-zero expected value (=0.5) or non-centered values.

From tables 1 and 2, it is clear that the increment in the number of modes is not significantly enhancing the percentage of Frobenius norm for the white noise image compared to our image. The same behavior can be seen from Figures 3 and 6, where the cumulative percentage of singular values (or the energy) for white noise image is rising slowly with number of modes in comparison with our image, underscoring the i.i.d. nature of the pixels.

Code:

```
#class    : Foundations of Data Science
#homework: 1
#question: 5
#author   : Mohammad Afzal Shadab
#email    : mashadab@utexas.edu

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread
import os
plt.rcParams['figure.figsize'] = [16.0, 8.0]
plt.rcParams['figure.dpi'] = 300

#Part (a) Image analysis using SVD
A = imread('austin_image.jpg')  #reading the image
X = np.mean(A, -1)              #converting RGB to grayscale
```

```
img = plt.imshow(X)
plt.set_cmap('gray')
plt.axis('off')
plt.savefig(f'original.jpg')

U, S, VT = np.linalg.svd(X, full_matrices =False) #Performing
                                    reduced SVD
S        = np.diag(S)                              #converting into
                                    diagonal matrix

j = 0

for r in (1,4,16,32):
    #Constructing a low rank approximation of the image
    Xapprox = U[:,:r] @ S[:r,:r] @ VT[:r,:]
    plt.figure(j+1)
    img = plt.imshow(Xapprox)
    plt.set_cmap('gray')
    plt.axis('off')
    plt.title(f' r= {r}')
    plt.savefig(f'svd_{r}modes.jpg')
    j += 1

plt.figure(j+1)
plt.semilogy(np.diag(S))
plt.ylabel('Singular values')
plt.xlabel('Number of modes')
plt.savefig('singular_values.jpg')

plt.figure(j+2)
plt.plot(np.cumsum(np.diag(S))/np.sum(np.diag(S)))
plt.ylabel('Cumulative sum of Singular values')
plt.xlabel('Number of modes')
plt.savefig('cum_sum_singular_values.jpg')


#Part (b) Percentage of Frobenius norm
S = np.diag(S)

norm = np.zeros((4,2))
j = 0
for r in (1,4,16,32):
    norm[j,0] = int(r)
    norm[j,1] = np.linalg.norm(S[:r])/np.linalg.norm(S)*100
    j += 1

print('Modes          Frobenius norm \n')
print(norm)


#Part (c) White noise image analysis using SVD
X    = np.random.rand(*X.shape)

img = plt.imshow(X)
plt.set_cmap('gray')
plt.axis('off')
plt.savefig(f'noise_original.jpg')

U, S, VT = np.linalg.svd(X, full_matrices =False) #Performing
                                    reduced SVD
S        = np.diag(S)                              #converting into
                                    diagonal matrix
```

```
j = 0

for r in (1,4,16,32):
    #Constructing a low rank approximation of the image
    Xapprox = U[:,:r] @ S[:r,:r] @ VT[:r,:]
    plt.figure(j+1)
    img = plt.imshow(Xapprox)
    plt.set_cmap('gray')
    plt.axis('off')
    plt.title(f' r= {r}')
    plt.savefig(f'noise_svd_{r}modes.jpg')
    j += 1

plt.figure(j+1)
plt.semilogy(np.diag(S))
plt.ylabel('Singular values')
plt.xlabel('Number of modes')
plt.savefig('noise_singular_values.jpg')

plt.figure(j+2)
plt.plot(np.cumsum(np.diag(S))/np.sum(np.diag(S)))
plt.ylabel('Cumulative sum of Singular values')
plt.xlabel('Number of modes')
plt.savefig('noise_cum_sum_singular_values.jpg')

S = np.diag(S)

norm = np.zeros((4,2))
j = 0
for r in (1,4,16,32):
    norm[j,0] = int(r)
    norm[j,1] = np.linalg.norm(S[:r])/np.linalg.norm(S)*100
    j += 1
print('White noise \n')
print('Modes          Frobenius norm \n')
print(norm)
```