CSE 382M: Found. Data Science & ML **Spring 2025**

Problem Set 2

Student Name: Noah Reef

Problem 1

Exercise 3.7

Let $A \in \mathbb{R}^{n \times n}$ be a block diagonal matrix

$$A = \begin{bmatrix} A_1 & 0 & \cdots & 0 \\ 0 & A_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_k \end{bmatrix}$$

where $A_n \in \mathbb{R}^{(n/k)\times(n/k)}$ matrix with $A_n^{(ij)} = a_n$ for all i, j and $a_1 > a_2 > \cdots > a_k > 0$. Consider the SVD of A,

$$A = U\Sigma V^T = \begin{bmatrix} U_1 & 0 & \cdots & 0 \\ 0 & U_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & U_k \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 & \cdots & 0 \\ 0 & \Sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \Sigma_k \end{bmatrix} \begin{bmatrix} V_1^T & 0 & \cdots & 0 \\ 0 & V_2^T & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & V_k^T \end{bmatrix}$$

where $A_n = U_n \Sigma_n V_n^T$. Note that since A is symmetric, we have that $V_n^T = V_n$ and hence we have that $v_n^{(1)}$ is constant and such that $\left|\left|v_n^{(1)}\right|\right| = 1$. This implies that $v_n^{(1)} = \frac{1}{\sqrt{n/k}} = \sqrt{\frac{k}{n}}$, since

$$\left|\left|v_n^{(1)}\right|\right|^2 = \sum_{i=1}^{n/k} \left(\sqrt{\frac{k}{n}}\right)^2 = \frac{k}{n}\frac{n}{k} = 1$$

Note that V_n is an orthogonal matrix up to the rank of A_n since A_n is a rank-1 matrix. This holds true for for all $v_n^{(j)}$ hence we have that V has entry $\sqrt{\frac{k}{n}}$ in the n-th block diagonal and zeros elsewhere. Note that if $a_1 = a_2 = \cdots = a_k$ then the singular values of A are equal hence has multiplicity k, and thus the singular vectors of A are any orthogonal basis of the n/k-dimensional subspace spanned by the columns of U_n and V_n.

Exercise 3.12

1. $||A_k||_F^2 = \sum_{i=1}^{k} \sigma_i^2$.
2. $||A_k||_2^2 = \sigma_1^2$
3. $||A - A_k||_F^2 = \sum_{i=k+1}^{r} \sigma_i^2$
4. $||A - A_k||_2^2 = \sigma_{k+1}^2$

Exercise 3.22

Let $A \in \mathbb{R}^{n \times n}$ be not necessarily invertible with SVD $A = \sum_i \sigma_i^2 u_i v_i^T$. Let $B = \sum_i \sigma_i^{-1} v_i u_i^T$ and let x be in the span of the right singular vectors of A, that is there exists y such that $Vy = x$. Then we have that

$$BAx = (V\Sigma^{-1}U^T)(U\Sigma V^T)x = VV^T x = x$$

note that $VV^T x = x$ since VV^T is the projection of x onto the columnspace of V but x is already in the columnspace of V hence the projection is x.

Exercise 3.28

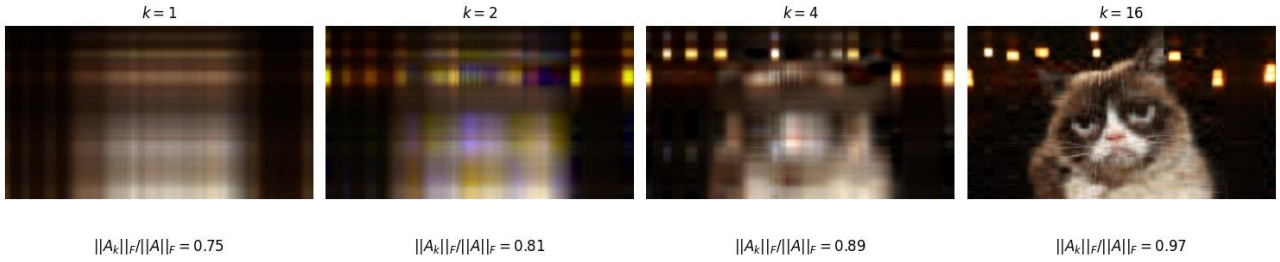

Figure 2.1. Exercise 3.28 Grumpy Cat Image

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

ks = [1,2,4,16]
cnt = 0
norms_F = [0,0,0,0]


for k in ks:
    # Open the image using Pillow
    img = Image.open('grumpy.jpg').convert('RGB')
    img_array = np.array(img)
    # Get dimensions: height, width, channels
    h, w, c = img_array.shape


    # Prepare an array to store the reconstructed channels
    img_reconstructed = np.zeros_like(img_array, dtype=np.float64)

    # Process each channel individually
```

```
    for channel in range(c):
        # Get the 2D array for this channel
        X = img_array[:, :, channel]

        # Perform SVD on the channel matrix
        U, S, Vt = np.linalg.svd(X, full_matrices=False)

        # Truncate to rank k
        U_k = U[:, :k]
        S_k = np.diag(S[:k])
        Vt_k = Vt[:k, :]

        norms_F[cnt] += np.linalg.norm(S_k, 'fro')**2

        # Reconstruct the channel
        X_reconstructed = U_k @ S_k @ Vt_k

        # Store the result
        img_reconstructed[:, :, channel] = X_reconstructed

    # Clip and convert back to uint8
    img_reconstructed = np.clip(img_reconstructed, 0, 255).astype('uint8')
    cnt += 1
    # Convert the array back to an image
    img_final = Image.fromarray(img_reconstructed, 'RGB')
    img_final.save('grumpy_k{}.jpg'.format(k))

# Compute the Frobenius norm of the original image
true_norm_F = 0
for channel in range(c):
    X = img_array[:, :, channel]
    U, S, Vt = np.linalg.svd(X, full_matrices=False)
    S = np.diag(S)
    true_norm_F += np.linalg.norm(S, 'fro')**2

print('Percentage of Frobenius norm preserved:')
for i in range(4):
    print('k = {}: {:.2f}%'.format(ks[i], norms_F[i] / true_norm_F * 100))

# Display the original image
# Load some example images (replace with your own images or arrays)
img1 = np.array(Image.open('grumpy_k1.jpg'))
img2 = np.array(Image.open('grumpy_k2.jpg'))
img3 = np.array(Image.open('grumpy_k4.jpg'))
img4 = np.array(Image.open('grumpy_k16.jpg'))

# Create a figure with 1 row and 3 columns
fig, axes = plt.subplots(1, 4, figsize=(15, 5))

# Display each image and set a title
axes[0].imshow(img1)
axes[0].set_title('$k=1$')
axes[0].axis('off')  # Optionally turn off axis ticks
```

```
axes[0].text(0.5, -0.3, '$||A_k||_F/||A||_F = {:.2f}$'.format(norms_F[0]/true_norm_F), transform=axes[0].transAxes,
             ha='center', fontsize=12)

axes[1].imshow(img2)
axes[1].set_title('$k=2$')
axes[1].axis('off')
axes[1].text(0.5, -0.3, '$||A_k||_F/||A||_F = {:.2f}$'.format(norms_F[1]/true_norm_F), transform=axes[1].transAxes,ha='center', fontsize=12)

axes[2].imshow(img3)
axes[2].set_title('$k=4$')
axes[2].axis('off')
axes[2].text(0.5, -0.3, '$||A_k||_F/||A||_F = {:.2f}$'.format(norms_F[2]/true_norm_F), transform=axes[2].transAxes,ha='center', fontsize=12)

axes[3].imshow(img4)
axes[3].set_title('$k=16$')
axes[3].axis('off')
axes[3].text(0.5, -0.3, '$||A_k||_F/||A||_F = {:.2f}$'.format(norms_F[3]/true_norm_F), transform=axes[3].transAxes,ha='center', fontsize=12)

# Adjust layout and show the figure
plt.tight_layout()

plt.savefig('grumpy_final.jpg')
```

Problem 2

Below is the code,

```
from numpy import *
from numpy.linalg import *
from numpy.random import randn, rand
from sklearn.cluster import KMeans
import pandas as pd


        # number of clusters
nk = 2000     # number of points per cluster
ds = [128,64,16,8]      # dimensions. Try these vaulues (128, 64,16 2)
seps = [0.8,2.8,4.8,8.8]   # separation factor. Try these values: 0.8,2.8,4.8,8.8
df = pd.DataFrame(columns=['d','sep','err','err_k'])

for d in ds:
    for sep in seps:
        k = d//2
        # ---- GENERATE POINTS
        def getpoints(d, k, nk, sep):
            '''
            d: dimensions (or number of features)
            k: number of Gaussians
            nk: average number of points per Gaussian
```

```
            sep: separation between means:  sep*  d**(1/4)

            Returns:
                X : 2d array  d by n,  n : total number of points
                mu: Gaussian means -- ground truth
                R:  distance of Gaussian means from origin
            '''
            a = 2*pi/k
            R = d**(1/4)/sqrt(2*(1-cos(a)))*sep #prediction of separatoin
            angles = arange(k)*a
            mu = zeros((d,k))
            mu[0,:] = cos(angles)*R
            mu[1,:] = sin(angles)*R
            nk = int64( nk*(rand(k)+0.5))
            X = zeros((d,sum(nk)))
            cnt = 0
            for j in range(k):
                m = mu[:,j]
                X[:,cnt:cnt+nk[j]] = randn(d,nk[j]) + m[:,newaxis]
                cnt += nk[j]
            return X,mu,R



        X,mu,R = getpoints(d,k,nk,sep)

        # remove NaN values from X and mu
        X = X[:,~isnan(X).any(axis=0)]
        mu = mu[:,~isnan(mu).any(axis=0)]

        # ---- CLUSTERING
        kmeans = KMeans(n_clusters=k).fit(X.T)
        print(kmeans.cluster_centers_.shape)
        print(mu.shape)

        def l2_err(X,Y):
            return sum(norm(X[:,i]-Y[:,i]) for i in range(X.shape[1]))

        # Error without best projection
        err = l2_err(mu,kmeans.cluster_centers_.T)

        if d != 2:
            # ---- FIND BEST PROJECTION

            # Use PCA to find best projection

            # SVD of X
            U,s,Vt = svd(X,full_matrices=False)

            # Truncate
            U = U[:,:k]
            s = s[:k]
            Vt = Vt[:k,:]

```

```
            # Project
            X_k = U @ diag(s) @ Vt

            # Error with best projection
            kmeans_k = KMeans(n_clusters=k).fit(X_k.T)
            err_k = l2_err(mu,kmeans_k.cluster_centers_.T)


        # store result in dataframe
        df = pd.concat([df,pd.DataFrame([[d,sep,err,err_k]],columns=['d','sep','err','err_k'])],ignore_index=True)

print(df.to_latex(index=False))
```

Where the metric used is the l_2 norm of the means. The results are as follows.

d	sep	err	err_k
128	2.800000	7833.715623	7530.961809
128	4.800000	12910.938768	12727.029125
128	8.800000	23276.776722	23842.659448
64	0.800000	407.226630	519.826968
64	2.800000	1515.458966	1661.103212
64	4.800000	2866.817932	2833.834689
64	8.800000	5078.749828	5125.914381
16	0.800000	24.355424	19.554686
16	2.800000	73.194710	62.157570
16	4.800000	123.091010	150.093613
16	8.800000	223.719214	217.659231
8	0.800000	3.620507	7.620087
8	2.800000	9.540235	16.193273
8	4.800000	16.244131	16.254271
8	8.800000	50.563351	71.443154

l for $k = d/2$ and where `err` is the standard l_2 error between the μ_i using the normal dataset, while `err_k` is the l_2 error after using the truncated SVD for the best projection of X onto k-dimensional subspace.

Problem 3

Consider the matricies $B \in \mathbb{R}^{m\times n}$ and $C \in \mathbb{R}^{k\times l}$ that are fixed. Let $X \in \mathbb{R}^{n\times k}$ be such that the entries are i.i.d. with expectation 0 and variance 1. Then we see that

$$||BXC||_F^2 = (C^T X^T B^T BXC)$$

and hence the expectation is given by

$$\mathbb{E}\left[||BXC||_F^2\right] = \text{Tr}(C^T \mathbb{E}\left[X^T B^T BX\right] C) = \text{Tr}(C^T \mathbb{E}\left[X^T X\right] B^T BC) = \text{Tr}(C^T B^T BC) = ||BC||_F^2 = ||B||_F^2 ||$$

Problem 4

```python
from numpy import *
from numpy.linalg import *
from numpy.random import randn
import builtins

def getAb(m,n):
    a = -1.0
    S = (arange(m)+1)**a
    G = randn(m,n); Q,_ = qr(G)
    Q = identity(m)[:m,:n]
    A = diag(S*S)@Q
    x = Q.T@diag(S)@randn(m); x /= norm(x)
    return A,A@x

def truncated_SVD(A,b,k):
    U,S,V = svd(A,full_matrices=False)
    U = U[:,:k]
    S = diag(S[:k])
    V = V[:k,:]
    A_m = U @ S @ V
    x_m = V.T @ inv(S) @ U.T @ b
    return A_m,x_m


def truncated_svd_solution(A, b, k):
    # Exact SVD
    U, s, Vt = svd(A, full_matrices=False)
    # Truncate to rank k
    U_k, s_k, Vt_k = U[:, :k], s[:k], Vt[:k, :]
    Sigma_k_inv = np.diag(1/s_k)
    x_k = Vt_k.T @ Sigma_k_inv @ U_k.T @ b
    A_k = U_k @ np.diag(s_k) @ Vt_k
    return A_k,x_k

def rand_truncated_SVD(A, b, k, p=10):
    m, n = A.shape
    Omega = randn(n, k+p)
    Y = A @ Omega
    Q,_ = qr(Y)
    B = Q.T @ A
    U_B, s_B, Vt = svd(B, full_matrices=False)
    U = Q @ U_B
    A_k = U @ diag(s_B) @ Vt
    x_k = Vt.T @ inv(diag(s_B)) @ U.T @ b
    return A_k, x_k

def rand_sketch_JLT(A, b, k,):
    m,n = A.shape
    # Create a Gaussian sketching matrix (with appropriate scaling)
    S = random.randn(n,m) / sqrt(k)
    A_k = S @ A
```

```
    b_sketch = S @ b
    x_k = lstsq(A_k, b_sketch)[0]
    A_k = pinv(S) @ A_k
    return A_k,x_k

def main():
    m=1024
    n=512
    k=[32*j for j in range(1,5)]
    for rank in k:# Try k=32, 64, 128
        A,b = getAb(m,n)
        x = lstsq(A,b)[0]

        # Approximate A by A_m and get LS solution x_m by truncated SVD
        A_m,x_m = truncated_SVD(A,b,rank)

        # Compute residual error of truncated SVD
        trunc_svd_res_err = norm(b - A_m@x_m)/norm(b)

        # Compute relative solution error of truncated SVD
        trunc_svd_sol_err = norm(x_m - x)/norm(x)

        # Compute relative matrix error of truncated SVD
        trunc_svd_mat_err = norm(A - A_m)/norm(A)

        # true truncated SVD solution
        _,S,_ = svd(A,full_matrices=False)
        print(S[rank]/S[0])
        # Approximate A by A_m and get LS solution x_m by random truncated SVD
        A_m,x_m = rand_truncated_SVD(A,b,rank)

        # Compute residual error of random truncated SVD
        rand_trunc_svd_res_err = norm(b - A_m@x_m)/norm(b)

        # Compute relative solution error of random truncated SVD
        rand_trunc_svd_sol_err = norm(x_m - x)/norm(x)

        # Compute relative matrix error of random truncated SVD
        rand_trunc_svd_mat_err = norm(A - A_m)/norm(A)

        # Approximate A by A_m and get LS solution x_m by random sketching JLT
        A_m,x_m = rand_sketch_JLT(A,b,rank)
        # Compute residual error of random sketching JLT
        rand_sketching_JLT_res_err = norm(b - A.dot(x_m))/norm(b)

        # Compute relative solution error of random sketching JLT
        rand_sketching_JLT_sol_err = norm(x_m - x)/norm(x)

        # Compute relative matrix error of random sketching JLT
        rand_sketching_JLT_mat_err = norm(A - A_m)/norm(A)

        # Output results
```

```
104         print(f'k : {rank}')
105         print(f'Truncated SVD residual error : {trunc_svd_res_err}')
106         print(f'Truncated SVD relative solution error : {trunc_svd_sol_err
    }')
107         print(f'Truncated SVD relative matrix error : {trunc_svd_mat_err}'
    )
108         print(f'Random Truncated SVD residual error : {
    rand_trunc_svd_res_err}')
109         print(f'Random Truncated SVD relative solution error : {
    rand_trunc_svd_sol_err}')
110         print(f'Random Truncated SVD relative matrix error : {
    rand_trunc_svd_mat_err}')
111         print(f'Random Sketching JLT residual error : {
    rand_sketching_JLT_res_err}')
112         print(f'Random Sketching JLT relative solution error : {
    rand_sketching_JLT_sol_err}')
113         print(f'Random Sketching JLT relative matrix error : {
    rand_sketching_JLT_mat_err}')
114
115
116 main()
```

```
k : 32
Truncated SVD residual error : 0.00021539630399243222
Truncated SVD relative solution error : 0.1823317035797015
Truncated SVD relative matrix error : 0.0029941785915632807
Random Truncated SVD residual error : 0.0020759015385376704
Random Truncated SVD relative solution error : 0.16563236678082463
Random Truncated SVD relative matrix error : 0.004581212823524299
Random Sketching JLT residual error : 2.5292680760579108e-14
Random Sketching JLT relative solution error : 1.0792896639048368e-09
Random Sketching JLT relative matrix error : 0.7100545287451024
k : 64
Truncated SVD residual error : 2.670396776574217e-05
Truncated SVD relative solution error : 0.1333622364102042
Truncated SVD relative matrix error : 0.0010701924157428576
Random Truncated SVD residual error : 0.0004208383334356015
Random Truncated SVD relative solution error : 0.1254128321483101
Random Truncated SVD relative matrix error : 0.0018422519684725594
Random Sketching JLT residual error : 1.2532847848155484e-13
Random Sketching JLT relative solution error : 7.911462547771843e-09
Random Sketching JLT relative matrix error : 0.6914105787508413
k : 96
Truncated SVD residual error : 1.8772571718525934e-05
Truncated SVD relative solution error : 0.12061373309693171
Truncated SVD relative matrix error : 0.0005834500703576864
Random Truncated SVD residual error : 0.00037101495082050325
Random Truncated SVD relative solution error : 0.11523066159483344
Random Truncated SVD relative matrix error : 0.0010837674698358627
Random Sketching JLT residual error : 2.047158543358448e-14
Random Sketching JLT relative solution error : 7.901659673763206e-10
Random Sketching JLT relative matrix error : 0.6855212353876912
k : 128
Truncated SVD residual error : 2.0904106004641193e-06
Truncated SVD relative solution error : 0.0567793062724264
Truncated SVD relative matrix error : 0.0003779631412231671
Random Truncated SVD residual error : 6.213404041244468e-05
Random Truncated SVD relative solution error : 0.05665729472030038
Random Truncated SVD relative matrix error : 0.0007253831903335213
Random Sketching JLT residual error : 1.898351155072386e-14
Random Sketching JLT relative solution error : 1.8497787423157028e-09
Random Sketching JLT relative matrix error : 0.7195184860097096
```

Figure 2.2. Code Results

For the matrix error of each method, we have that for Truncated SVD that the expected error is given by

$$\frac{||A - A_k||_2}{||A||_2} - \frac{\sigma_{k+1}}{\sigma_1}$$

which in the case for $k = 32$ we get that the theoreitcal error is ≈ 0.0009 for randomized SVD we have that for a $2k$ factorization that the expected error is given by

$$\mathbb{E}\left[||A - A_k||_2\right] = O(1 + 9\sqrt{kn})\sigma_{k+1}$$