Homework Set 2, CPSC 8420, Spring 2022

Song, Zhiyuan

March 16, 2022

A. to P1

In the case of maximize $\|\mathbf{X}\boldsymbol{\phi}\|_2^2, s.t.\ \|\boldsymbol{\phi}\|_2 = 1$, we can apply te equation

$$\|\mathbf{A}\|_2^2 = \text{Trace}(\mathbf{A}^T\mathbf{A}) = \text{Trace}(\mathbf{A}^T\mathbf{A}) \tag{1}$$

Consequently, maximize $\|\mathbf{X}\boldsymbol{\phi}\|_2^2, s.t.\ \|\boldsymbol{\phi}\|_2 = 1$ is equivalent to

$$\text{maximize}\,\text{Trace}(\boldsymbol{\phi}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\phi}) s.t.\ \|\boldsymbol{\phi}\|_2 = 1$$

If we consider an eigen value problem

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\phi} = \lambda\boldsymbol{\phi}$$

Where λ is the eigenvalue of $\mathbf{X}^T\mathbf{X}$. wW will get to

$$\boldsymbol{\phi}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\phi} = \lambda\boldsymbol{\phi} = \boldsymbol{\phi}^T\lambda\boldsymbol{\phi} \equiv \lambda\|\boldsymbol{\phi}\|_2 = \lambda$$

Therefore, if we want to maximize the $\boldsymbol{\phi}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\phi}$, it is equivalent to maximize the eigenvalue of $\mathbf{X}^T\mathbf{X}$. Accordingly, if we take SVD on $\mathbf{X}^T\mathbf{X}$, we will get singular values of $\mathbf{X}^T\mathbf{X}$ in order laying on $\mathbf{S}$ diagnoally as well as $\mathbf{U}$, where $[\mathbf{U}, \mathbf{S}] = svd(\mathbf{X}^T\mathbf{X})$. It will result in the largest λ at $\mathbf{S}_{11}$ and the corresponding eigenvector $\boldsymbol{\phi}_1$, which is the first column of $\mathbf{U}$. We can demostrate it using matrix notation. Assume $[\mathbf{U}, \boldsymbol{\Sigma}] = \text{SVD}\left(\mathbf{X}^T\mathbf{X}\right)$, we will have $\mathbf{X}^T\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{U}^T$. Consider an element at the ith row and the jth column of an matrix $\mathbf{U}$

$$(\mathbf{U})_{ij} = u_{ij}$$

Furthermore, we can denote matrix multiplication, transpose, unitarity and singular value decomposition as

$$(\mathbf{C})_{ij} = (\mathbf{AB})_{ij} = \sum_k a_{ik}b_{kj}$$

$$\left(\mathbf{A}^T\right)_{ij} = a_{ji}$$

$$\left(\mathbf{A}^T\mathbf{A}\right) = \mathbf{I} \equiv \sum_k \left(\mathbf{A}^T\right)_{ik} a_{kj} = \sum_k a_{ki}a_{kj} = \delta_{ij}$$

$$(\boldsymbol{\Sigma})_{ij} = \lambda_i\delta_{ij}$$

Therefore,

$$\left(\mathbf{X}^T\mathbf{X}\right)_{ij} = \left(\mathbf{U}\mathbf{\Sigma}\mathbf{U}^T\right)_{ij} = \sum_{m,l} u_{im}\lambda_m\delta_{ml}u_{jl}$$

If we consider to use qth column of $\mathbf{U}$, $(\boldsymbol{\phi}_q)_i = u_{iq}$, and get the corresponding eignvalue, we will have to evaluate

$$\boldsymbol{\phi}_q^T\left(\mathbf{U}\mathbf{\Sigma}\mathbf{U}^T\right)\boldsymbol{\phi}_q \equiv$$

$$\sum_i u_{iq}\left(\sum_{m,l} u_{im}\lambda_m\delta_{ml}u_{jl}\right)\sum_j u_{jq} = \sum_{m,l}\lambda_m\delta_{ml}\overbrace{\sum_i u_{im}u_{iq}}^{\delta_{mq}}\overbrace{\sum_j u_{jl}u_{jq}}^{\delta_{lq}}$$
$$= \lambda_q$$

As a result, if we want the maximum ($q = 1$) singular value of $\mathbf{X}^T\mathbf{X}$, we can just calculate $\boldsymbol{\phi}_1^T\mathbf{X}^T\mathbf{X}\boldsymbol{\phi}_1$ in order to maximize $\|\mathbf{X}\boldsymbol{\phi}\|_2^2$, *s.t.* $\|\boldsymbol{\phi}\|_2 = 1$. Moreover, we can consider to use a linear combination of eigenvectors $\mathbf{X}^T\mathbf{X}$, $\boldsymbol{\psi}$,to act on $\mathbf{X}^T\mathbf{X}$

$$\boldsymbol{\psi} = \sum_i p_i\boldsymbol{\phi}_i \;\; s.t. \sum_i p_i^2 = 1$$

$$\|\boldsymbol{\psi}\|_2 \equiv \sum_{k,a} p_a p_k \overbrace{\sum_i u_{ia}u_{ik}}^{\delta_{ak}}$$
$$= \sum_a p_a p_a = 1$$

$$\therefore \boldsymbol{\psi}_q^T\left(\mathbf{U}\mathbf{\Sigma}\mathbf{U}^T\right)\boldsymbol{\psi}_q \equiv$$

$$\sum_{i,k} p_k u_{ik}\left(\sum_{m,l} u_{im}\lambda_m\delta_{ml}u_{jl}\right)\sum_{a,j} p_a u_{ja} = \sum_{m,l,k,a}\lambda_m\delta_{ml}p_k\overbrace{\sum_i u_{ik}u_{im}}^{\delta_{km}}p_a\overbrace{\sum_j u_{jl}u_{ja}}^{\delta_{la}}$$
$$= \sum_m p_m p_m\lambda_m$$
$$\leq \sum_m p_m p_m\lambda_1 = \lambda_1$$

Therefore, the first column of $\mathbf{U}$ will be the most optimal eigenvector to get the largest λ.

A. to P2

If the dataset includes outlayers, the sum of square residuals may amply such differences, leaading to biased centroids. Different from K-means, K-medians update a centroid by calculating the median value of data points inside the cluster. Besides, k-medians will be evaluated by summation of absolute residuals. Here is a specific experiment to present their different as shown in Figure 1. Centroids in K-median clusters showed less impact from added outlyers than K-means. Besides, they performed differently on iterations and clustering results as well as different objective scores.

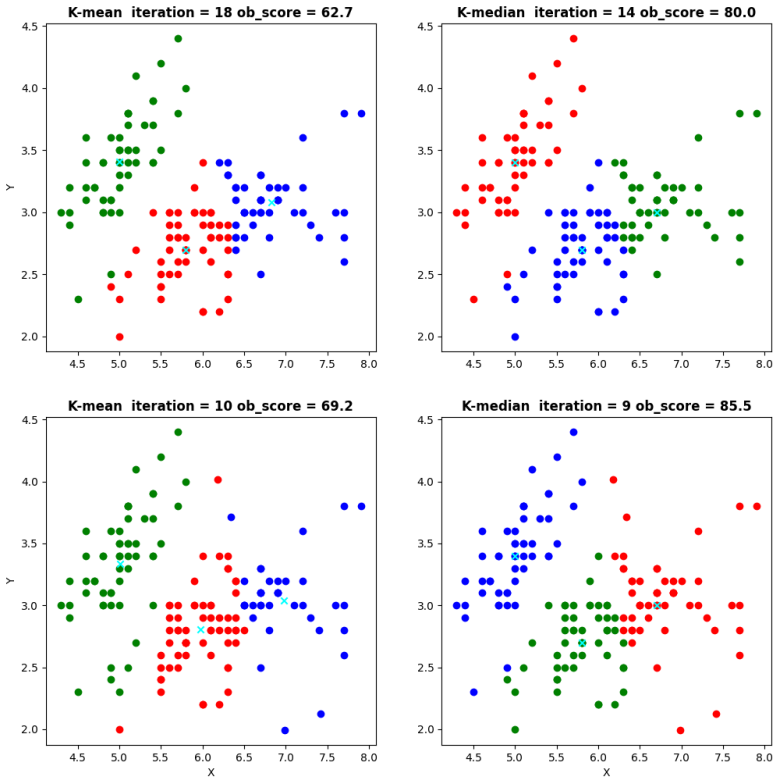

Figure 1: **Differences of K-means and K-medians on clustering the same datapoints** Upper two pannels shows the clustering results without outlayers. While lower two include outlayers. Green, blue red denote different clusters; Cyan points denote centroids of the corresponding clusters.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Mar  7 16:53:14 2022

@author: zhiyuas
"""
import math
import numpy as np
from sklearn import datasets
import random
import matplotlib.pyplot as plt

def dist(point1,point2):
    return math.sqrt(np.sum([(p1-p2)**2 for p1, p2 in zip(point1,point2)]))
def dist_ab(point1,point2):
    return np.sum([abs(p1-p2) for p1, p2 in zip(point1,point2)])
class K_M:
     class kmean():
         def __init__(self,n_k,tol,data):
             ob_scores = [0,9999] #objective fuction
             xy_lims = [[np.min(data[:,i]),np.max(data[:,i])] for i in range(len(data[0]))]
             #initialize centroids at the beginning
             centroids = [[random.uniform(j[0],j[1]) for j in xy_lims] for i in range(n_k)]
             iter_step = 0
             while abs(ob_scores[-1] -ob_scores[-2]) >= tol:
                 #initialize clusters every step
                 cluster_assigned = dict([(i, []) for i in range(n_k)])
                 for row in data:
                     cluster_assignment = np.argmin([dist(row,centroids[i]) for i in range(n_k)])
                     cluster_assigned[cluster_assignment].append(row.tolist())
                 dis_sum = 0
                 for i in range(n_k):
                     dis_sum = dis_sum + np.sum([dist(centroids[i],cluster_assigned[i][j]) \
                                     for j in range(len(cluster_assigned[i]))])
                 ob_scores.append(dis_sum)
                 centroids = [ [np.mean(np.array(cluster_assigned[j])[:,i])\
                               for i in range(len(cluster_assigned[j][0]))] for j in range(n_k)]
                 iter_step = iter_step + 1
                 self.ob_scores = ob_scores;self.centroids = centroids
                 self.step = iter_step;self.clusters = cluster_assigned
     class kmedian():
         def __init__(self,n_k,tol,data):
             ob_scores = [0,9999] #objective fuction
             xy_lims = [[np.min(data[:,i]),np.max(data[:,i])] for i in range(len(data[0]))]
             #initialize centroids at the beginning
             centroids = [[random.uniform(j[0],j[1]) for j in xy_lims] for i in range(n_k)]
             iter_step = 0
             while abs(ob_scores[-1] -ob_scores[-2]) >= tol:
                 cluster_assigned = dict([(i, []) for i in range(n_k)])
                 for row in data:
```

```
                    cluster_assignment = np.argmin([dist(row,centroids[i]) for i in range(n_k)])
                    cluster_assigned[cluster_assignment].append(row.tolist())
                dis_sum = 0
                for i in range(n_k):
                    dis_sum = dis_sum + np.sum([dist_ab(centroids[i],cluster_assigned[i][j])\
                          for j in range(len(cluster_assigned[i]))])
                ob_scores.append(dis_sum)
                centroids = [ [np.median(np.array(cluster_assigned[j])[:,i])\
                        for i in range(len(cluster_assigned[j][0]))] for j in range(n_k)]
                iter_step = iter_step + 1
                self.ob_scores = ob_scores;self.centroids = centroids
                self.step = iter_step;self.clusters = cluster_assigned
X,y = datasets.load_iris(return_X_y=True);X = X[:,:2]
points_added = np.array([[6.181,4.016],[6.34,3.714],[6.987,1.994],[7.415,2.126]])
X_add = []
X_add.append(X)
X_add.append(np.concatenate((X,points_added),axis=0))
random.seed(2022)
colors = ['r','green','blue']
fig,axes = plt.subplots(figsize=(12,12),nrows=2, ncols=2)

tol = 1e-5; n_k = 3
for ax,data in zip(axes,X_add):
    A = K_M.kmean(n_k, tol, data);B = K_M.kmedian(n_k, tol, data)
    for i in range(n_k):
        x_coord = [x[0] for x in A.clusters[i]];y_coord = [y[1] for y in A.clusters[i]]
        ax[0].scatter(x_coord,y_coord,color=colors[i])
        ax[0].scatter(A.centroids[i][0],A.centroids[i][1],color='cyan',marker='x')
        x_coord = [x[0] for x in B.clusters[i]];y_coord = [y[1] for y in B.clusters[i]]
        ax[1].scatter(x_coord,y_coord,color=colors[i])
        ax[1].scatter(B.centroids[i][0],B.centroids[i][1],color='cyan',marker='x')
    ax[0].set_title('K-mean  iteration = '+str(A.step)+' ob_score = '\
                     +str('%.1f'%A.ob_scores[-1]),fontweight='bold')
    ax[1].set_title('K-median  iteration = '+str(B.step)+' ob_score = '\
                    +str('%.1f'%B.ob_scores[-1]),fontweight='bold')

for i in range(2):
        axes[1][i].set_xlabel('X')
        axes[i][0].set_ylabel('Y')
plt.savefig('prob2.png')
```

A. to P3

1. If we take SVD on the matrix $\mathbf{A} \in \mathbb{R}^{n\times p}$, we can get $\mathbf{A} = \left[\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\right]$, where $\mathbf{U} \in \mathbb{R}^{n\times n}$, $\mathbf{\Sigma} \in \mathbb{R}^{n\times n}$, and $\mathbf{V} \in \mathbb{R}^{p\times n}$, instead of $\mathbf{U} \in \mathbb{R}^{n\times p}$, $\mathbf{\Sigma} \in \mathbb{R}^{p\times p}$, and $\mathbf{V} \in \mathbb{R}^{p\times p}$. Then, we can

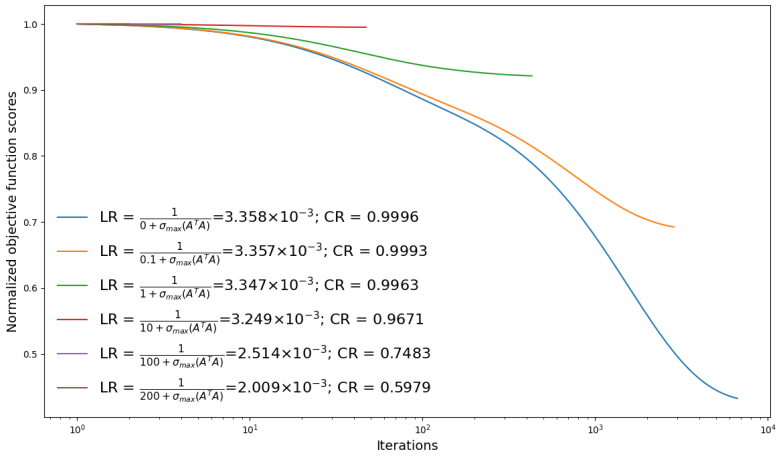

Figure 2: **Normalized objective function scores vs iteration steps in logspace** The learning rate for each λ was denoted as LR and convergence rates were denoted as CR

calculate $\mathbf{A}^T = \mathbf{V}\mathbf{\Sigma}\mathbf{U}^T$. Therefore,

$$\begin{aligned}\mathbf{A}^T\mathbf{A} &= \mathbf{V}\mathbf{\Sigma}\mathbf{U}^T\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \\ &= \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T\end{aligned}$$

Consider that

$$(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}$$

We should have obtained

$$\begin{aligned}\left(\mathbf{A}^T\mathbf{A}\right)^{-1} &= \left(\mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T\right) \\ &= \left(\mathbf{V}^T\right)^{-1}\left(\mathbf{\Sigma}\right)^{-2}\left(\mathbf{V}\right)^{-1}\end{aligned}$$

However, neither $\mathbf{V} \in \mathbb{R}^{p\times n}$ nor $\mathbf{V}^T \in \mathbb{R}^{n\times p}$, is invertible. Therefore, in the case of $\mathbf{A} \in \mathbb{R}^{n\times p}$ and $p > n$, $\mathbf{A}^T\mathbf{A}$ is not invertible, leading to the inapplicable $(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y}$.

2. & 3. The form of ridge regression will reduce to general least square form as well as the learning rate form if λ is set to be zero. Hence, sub-problem 2 and 3 can be taken together into account such that λ varies from $0, 0.1, 1, 10, 100, 200$. Normalized objective scores were computed to emphasize the converge of the gradient descent method. Otherwise, the scores would change too slightly to see the decreases. Besides, the precison was set to be 10^{-5} on percent difference instead. As shown in Figure 2, the learning and convergence rates decreased when λ increased, suggesting faster converges and shorter iterations.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
"""
Created on Sat Mar 12 15:54:26 2022

@author: zhiyuas
"""

import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg as sl
def func(A,y,beta,lam):
    return ((A.dot(beta)-y).T.dot(A.dot(beta)-y)+lam*beta.T.dot(beta))[0][0]/2
def grad_func(A,y,beta,lam):
    return A.T.dot(A.dot(beta)-y)+lam*beta
def GD(f,gradient_f,feature, target, ini_beta, lam, learn_rate, tol):
    beta = ini_beta;ite = 0
    diff = 1e10
    #recording objective function
    ob_s = [f(feature,target,ini_beta,lam)]
    while diff > tol:
        beta = beta - learn_rate*gradient_f(feature,target,beta,lam)
        ite = ite + 1
        ob_s.append(f(feature,target,beta,lam))
        diff = abs((ob_s[-1]-ob_s[-2])/ob_s[-1])
    return ite, beta, ob_s[1:]
A = np.array([[1,2,4],[1,3,5],[1,7,7],[1,8,9]]);y=np.array([[1],[2],[3],[4]])
#f(x) = (A*beta-y)^T*(A*beta-y) innitial inputs
tol = 1e-5; learn_rate = 1; beta = np.array([0 for i in range(3)])[...,None];ite = 0;lam=0
lambdas = [0,0.1,1,10,100,200]
ites = [];ks = [];ob_score = [];cr = []
for lam in lambdas:
    LR = 1/(np.max(sl.svd(A.T.dot(A))[1])+lam)
    ite,_,ob_s = GD(func,grad_func,A,y,beta,lam,LR,tol)
    ites.append(ite);ks.append(LR);ob_score.append(ob_s)
    _,s,_ = sl.svd(A.T.dot(A)+lam*np.identity(3))
    cr.append(1-np.min(s)/np.max(s))
ob_score = [ob_s/ob_s[0] for ob_s in ob_score] #normalization
#plots
plt.figure(figsize=(14,8))
for lam,ite,ob_s,k,c in zip(lambdas,ites,ob_score,ks,cr):
    plt.plot(np.arange(1,ite+1),ob_s,label=r'LR = $\frac{{1}}{{}+\sigma_{{max}}(A^TA)}}$='\
             .format(lam)+'%.5s'%(k*1000)+r'$\times 10^{-3}$; CR = '+'%.6s'%c)
plt.xscale('log')
plt.legend(frameon=False,fontsize=16)
plt.xlabel('Iterations',fontsize=14)
plt.ylabel('Normalized objective function scores',fontsize=14)
plt.savefig('prob3.png')
```

Figure 3: **Image reconstructions with the increase of the size of the eigenvector set**

A. to P4

Figure 3 shows the reconstructed images with the corresponding principal components. As the principal number increases, the resolution of the image becomes higher. The way I defined to determine the number of principal numbers is to consider both the cumulative variance of the reconstructed image matrix to the original one and the number of principal components. The equivalent objective funtion

$$\sum_{i,j}^{m,n} (a_{ij} - z_{ij})^2 + k$$

Where a_{ij} denote the reconstructed image matrix, z_{ij} refers to the original matrix, and k is the number of selected principal components. However, the variance-like term and component term were normalized to their maximum calculation in practice, which is to keep the same order of magnitude for two terms, as shown in 4. Therefore, the best trade-off in my case will be around 42 principal components. The outcome image will be better than the one involving 20 components but vaguer than the one involving 50 components in Figure 3.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 16 13:53:35 2022

@author: zhiyuas
"""

import numpy as np
```

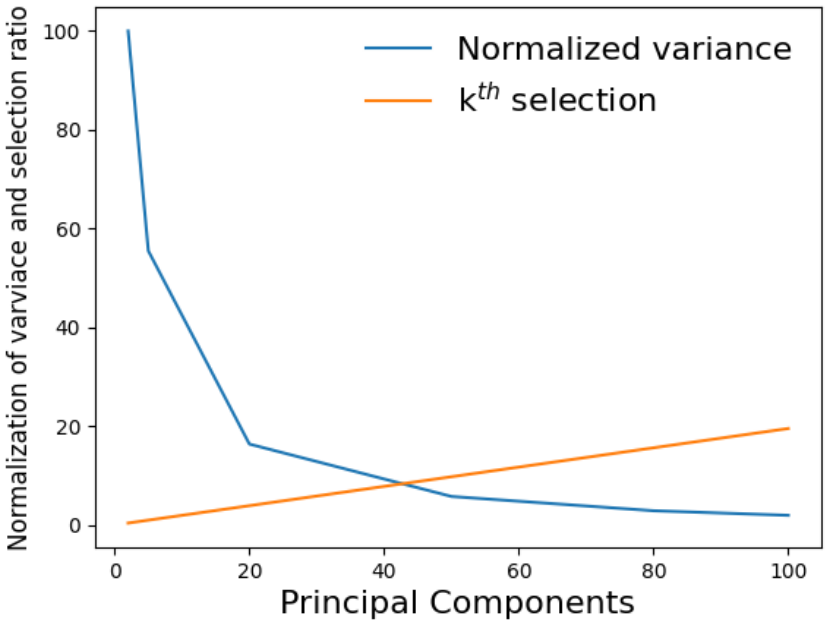

Figure 4: **Normalized cumulative variace and the number of principal component**

```
from PIL import Image
from scipy.linalg import svd
import matplotlib.pyplot as plt
def PAC_Decom_var(k_ev,image):
    m,n = image.size
    data = list(image.getdata())
    #read colors from the image
    color_sets = [np.array([x[j] for x in data]).reshape(m,n) for j in range(3)]
    #comput mean value for each color channel and centerize each channel
    mean_set = [np.mean(x) for x in color_sets]
    color_sets = [x-y for x,y in zip(color_sets,mean_set)]
    k=k_ev
    redu_sets = []
    for color_channel in color_sets:
        U,s,VT = svd(color_channel)
        redu_sets.append((U[:,:k].dot(np.diag(s[:k]).dot(VT[:k,:])))).reshape(m*n))
    redu_sets = np.rint(np.array([x+y for x,y in zip(redu_sets,mean_set)]))\
        .astype(int).transpose()
    return (np.sum((redu_sets-np.array(data))**2)/(len(data)*3))
fontsize_label = 16
#import image
im = Image.open('Lenna_(test_image).png')
#produce an empty image
best_com_num = [2,5,20,50,80,100]
cv = []
```

```
for k in best_com_num:
    cv.append(PAC_Decom_var(k,im))
plt.plot(best_com_num,np.array(cv)/cv[0]*100,label='Normalized variance')
plt.plot(best_com_num,np.array(best_com_num)*100/512,label=r'k$^{th}$ selection')
plt.xlabel('Principal Components',fontsize=fontsize_label)
plt.ylabel('Normalization of varviace and selection ratio',fontsize=12)
plt.legend(frameon=False,fontsize=fontsize_label)
plt.savefig('prob4-2.png')
```