① A is n-by-n symmetric matrix, right singular vectors v_k, $k=1,\dots,n$

1% of v_1 if $b_0 = \frac{1}{\sqrt{n}}\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}$

right singular vectors $\Rightarrow$ $D = A^T A = V\Sigma^T\Sigma V^T = V\begin{bmatrix}\sigma_1^2 & & \\ & \ddots & \\ 0 & & \sigma_n^2\end{bmatrix}V^T \Rightarrow$

$\Rightarrow D b_{k-1} = D^k b_0 = V\Lambda^k g$

$$b_k = \frac{D b_{k-1}}{\|D b_{k-1}\|_2} = \frac{V\Lambda^k g}{\|V\Lambda^k g\|_2}$$

$$V\Lambda^k g = [v_1 \dots v_N]\begin{bmatrix}\lambda_1^k & & 0\\ & \ddots & \\ 0 & & \lambda_N^k\end{bmatrix}\begin{bmatrix}g_1\\ \vdots\\ g_N\end{bmatrix} = \lambda_1^k g_1 \underline{V}\begin{bmatrix}1 & & & \\ & \left(\frac{\lambda_2}{\lambda_1}\right)^k & & 0\\ & & \ddots & \\ 0 & & & \left(\frac{\lambda_N}{\lambda_1}\right)^k\end{bmatrix}\begin{bmatrix}1\\ g_2/g_1\\ \vdots\\ g_N/g_1\end{bmatrix}$$

but $\frac{\lambda_i}{\lambda_1} < 1$ so $\left(\frac{\lambda_i}{\lambda_i}\right)^k \to 0$

so,

$$V\Lambda^k g \to \lambda_1^k g_1 \underline{V}\begin{bmatrix}1 & & 0\\ & 0 & \\ 0 & \ddots & 0\end{bmatrix}\begin{bmatrix}1\\ \vdots\end{bmatrix} = \lambda_1^k g_1 \underline{v_1}$$

Then

$$b_k \to \frac{\lambda_1^k g_1 v_1}{\|\lambda_1^k g_1 v_1\|_2} = \frac{v_1}{\|v_1\|_2} = v_1$$

Now we want $b_k \to 0.99\, v_1$, we want k.

$$b_k = A b_{k-1} = A^k b_0 = A^k \frac{1}{\sqrt{N}}\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix} = 0.99\, v_1$$

②

a) It isn't a subspace because the (0,0,0) doesn't belong in the data.

b) We could move the data so that a point lie in the (0,0,0).

c) Yes, it seems that the (0,0,0) belongs to it.

d) The represented data clearly forms a line

e) a is the unit-norm vector $\Rightarrow a = U$

$x_{z,i} \approx a w_i \Rightarrow W = SV \Rightarrow w_i = \lambda_i v_i \Rightarrow$ For the rank 1 approximation we will use the first column

f) $x_i \approx a w_i + b \Rightarrow b$ is the mean of the data.

g) $E = X - \text{Rank-1}(X) = USV^T - S[0,0]\,U[:,0:1]\,V^T[0:1,:]$

$$E = \sum_{i=2}^{N} \sigma_i u_i v_i^T$$

$$\|A\|_F^2 = \sum_{i=1}^{N} \|a_i\|_2$$

Then,

$$\|E\|_F^2 = \sum_{i=2}^{N} \|\sigma_i u_i v_i^T\|_2$$

i) $x_i \approx a_1 w_{1i} + a_2 w_{2i} + b \quad , i = 1, \cdots, 1000$

w will be the same as e) but now for the rank 2 approximation we will use the second column.

j) $E = X - \text{Rank-1}(X) = USV^T - S[0,0]\,U[:,0:1]\,V^T[0:1,:] - S[1,1]\,U[:,0:2]\,V^T[0:1,:]$

$$E = \sum_{i=3}^{N} \sigma_i u_i v_i^T$$

Then,

$$\|E\|_F^2 = \sum_{i=3}^{N} \|\sigma_i u_i v_i^T\|_2$$

3

b)

$$w = (X^T X + \lambda I)^{-1} X^T \alpha \implies \text{using SVD}: \; X^T X = V\Sigma^2 V^T \;,\; \lambda I = V \lambda I V^T$$

Then,

$$w = (V(\Sigma^2 + \lambda I)V^T)^{-1} V \Sigma U^T \alpha = V \overbrace{\boxed{(\Sigma^2 + \lambda I)^{-1} \Sigma}}^{D} U^T \alpha$$

$$D = \begin{bmatrix} \frac{1}{\sigma_1^2+\lambda} & & \underline{0} \\ & \ddots & \\ \underline{0} & & \frac{1}{\sigma_p^2+\lambda} \end{bmatrix} \begin{bmatrix} \sigma_1 & & \underline{0} \\ & \ddots & \\ \underline{0} & & \sigma_p \end{bmatrix} = \begin{bmatrix} \frac{\sigma_1}{\sigma_1^2+\lambda} & & \underline{0} \\ & \ddots & \\ \underline{0} & & \frac{\sigma_p}{\sigma_p^2+\lambda} \end{bmatrix} \implies w = \sum_{i=1}^{p} \frac{\sigma_i}{\sigma_i^2+\lambda} v_i (u_i^T \alpha)$$

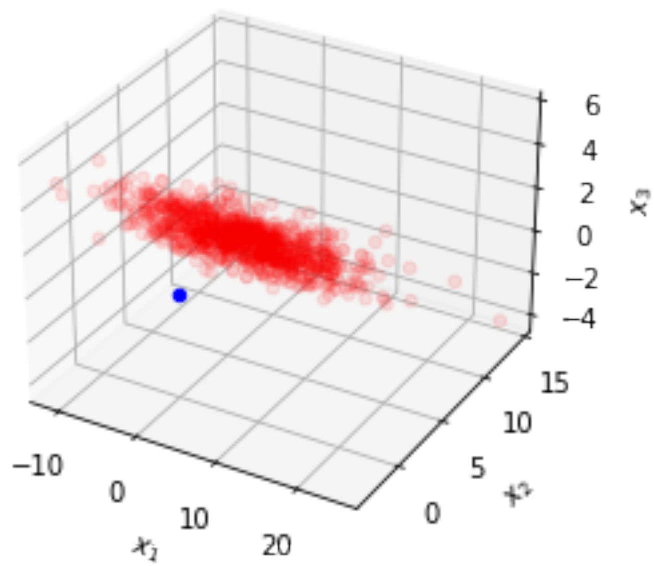
Assign6Starter

March 28, 2022

[ ]:
```
# Enable interactive rotation of graph
# %matplotlib notebook
%matplotlib inline

import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.decomposition import PCA

# Load data for activity
X = np.loadtxt('sdata.csv',delimiter=',')
center_point = np.array([0,0,0])
center_point in X
```

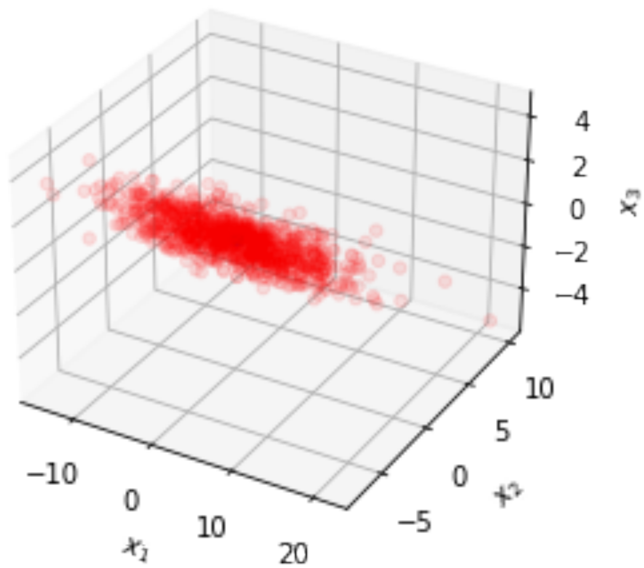
[ ]:
```
False
```

[ ]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X[:,0], X[:,1], X[:,2], c='r', marker='o', alpha=0.1)
ax.scatter(0,0,0,c='b', marker='o')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$x_3$')

plt.show()
```



[ ]:
```
# Subtract mean
X_m = X - np.mean(X, 0)
```

[ ]:
```
# display zero mean scatter plot
fig = plt.figure()

ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_m[:,0], X_m[:,1], X_m[:,2], c='r', marker='o', alpha=0.1)

ax.scatter(0,0,0,c='b', marker='o')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$x_3$')

plt.show()
```

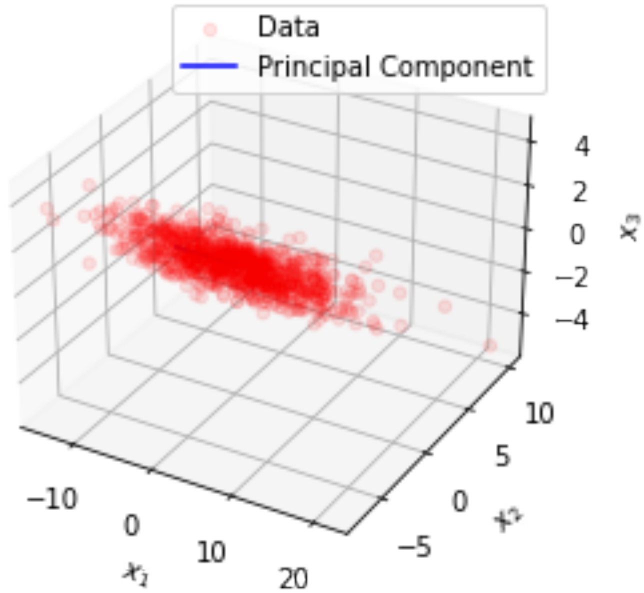


[ ]:
```
# Use SVD to find first principal component

U,s,VT = np.linalg.svd(X_m,full_matrices=False)

# complete the next line of code to assign the first principal component to a
a = VT[0]
a
```

[ ]: array([-0.87325954, -0.43370914, 0.2220679])

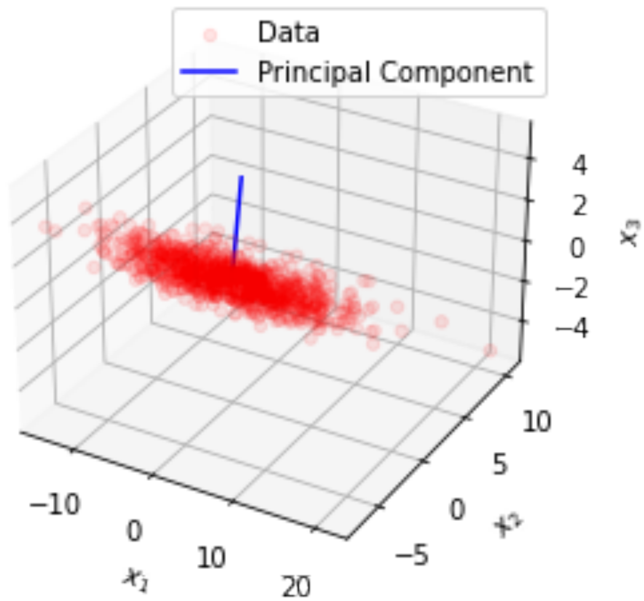
[ ]:
```
# display zero mean scatter plot and first principal component

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

#scale length of line by root mean square of data for display
ss = s[0]/np.sqrt(np.shape(X_m)[0])

ax.scatter(X_m[:,0], X_m[:,1], X_m[:,2], c='r', marker='o', label='Data',␣
 ↪alpha=0.1)

ax.plot([0,ss*a[0]],[0,ss*a[1]],[0,ss*a[2]], c='b',label='Principal Component')

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$x_3$')
```

```
ax.legend()
plt.show()
```




```
[ ]: # h)
a_2 = VT[1]

S_matrix = np.zeros_like(X_m)
np.fill_diagonal(S_matrix, s)

#Rank-2 aprox
X_2_approx = S_matrix[0,0]*U[:,0:1]@VT[0:1,:]+S_matrix[1,1]*U[:,1:2]@VT[1:2,:]
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

#scale length of line by root mean square of data for display
ss = s[0]/np.sqrt(np.shape(X_2_approx)[0])

ax.scatter(X_2_approx[:,0], X_2_approx[:,1], X_2_approx[:,2], c='r',␣
 ↪marker='o', label='Data', alpha=0.1)

ax.plot([0,ss*a_2[0]],[0,ss*a_2[1]],[0,ss*a_2[2]], c='b',label='Principal␣
 ↪Component')

ax.set_xlabel('$x_1$')
```

```
ax.set_ylabel('$x_2$')
ax.set_zlabel('$x_3$')


ax.legend()
plt.show()
```



[ ]:
```
S_matrix = np.zeros_like(X_m)
np.fill_diagonal(S_matrix, s)

#Rank-1 aprox
X_1_approx = S_matrix[0,0]*U[:,0:1]@VT[0:1,:]
```

[ ]:
```
E_2 = X_m - X_2_approx
E_1 = X_m - X_1_approx

print("Frobenius Norm of E in Rank-1 approximation: " ,np.linalg.norm(E_1,␣
↪ord='fro'))
print("Frobenius Norm of E in Rank-2 approximation: " ,np.linalg.norm(E_2,␣
↪ord='fro'))
```

```
Frobenius Norm of E in Rank-1 approximation:  25.03377559191337
Frobenius Norm of E in Rank-2 approximation:  12.367116712429967
```

[ ]:
```
def select_randoms():
    to_return = [None,None,None,None,None,None]
```

```
    for i in range(len(to_return)):
        random_num = np.random.randint(0,9)
        while random_num in to_return:
            random_num = np.random.randint(0,9)
        to_return[i] = random_num
    return to_return
```

[ ]:
```
def get_u_and_y(randoms,U,y):
    randoms.sort()
    j = randoms[0]
    new_u = U[j+(j*16):(j+(j*16))+16]
    new_y = y[(j+(j*16)):(j+(j*16))+16]
    for i in range(len(randoms)):
        j = randoms[i]
        new_y = np.concatenate((new_y,y[(j+(j*16)):(j+(j*16))+16]))
        new_u = np.concatenate((new_u,U[j+(j*16):(j+(j*16))+16]))
    return new_u,new_y
```

[ ]:
```
def get_w(randoms, V, S, U, y):
    new_u,new_y = get_u_and_y(randoms,U,y)
    return V@S@new_u.transpose()@new_y
```

[ ]:
```
# 3)
# a)

data = loadmat('face_emotion_data.mat')
X = data['X']
y = data['y']

U,s,VT = np.linalg.svd(X, full_matrices= False)
S = np.arange(81).reshape(9,9)
S_matrix = np.zeros_like(S)
np.fill_diagonal(S_matrix, s)

S_matrix_inverse = np.zeros_like(S_matrix)
S_matrix_inverse = np.float_(S_matrix_inverse)

for i in range (0,9):
    S_matrix_inverse[i][i] = 1/S_matrix[i][i]

error_rates = np.float_(np.arange(56))
min_error_rate = None
min_random_group = np.array([0,0,0,0,0,0])

for k in range(56):
    misclassiffications = 0
    randoms = np.array([0,0,0,0,0,0])
```

```
    randoms = select_randoms()
    w = get_w(randoms,VT.transpose(),S_matrix_inverse,U,y)
    y_hat = np.sign(X@w)
    aux = y_hat - y
    for value in aux:
        if value != 0:
            misclassiffications += 1
    error_rates[k] = misclassiffications/96
    if min_error_rate == None or error_rates[k] < min_error_rate:
        min_error_rate = error_rates[k]
        min_random_group = randoms

print(error_rates)
print("Mean error rate: " ,error_rates.mean())
print("Group: " ,min_random_group, "Error rate: ", min_error_rate)
```

```
[0.08333333 0.0625     0.08333333 0.11458333 0.10416667 0.04166667
 0.10416667 0.08333333 0.08333333 0.04166667 0.0625     0.08333333
 0.0625     0.04166667 0.10416667 0.07291667 0.0625     0.03125
 0.05208333 0.08333333 0.07291667 0.05208333 0.0625     0.07291667
 0.08333333 0.08333333 0.11458333 0.04166667 0.09375    0.08333333
 0.09375    0.08333333 0.09375    0.0625     0.0625     0.07291667
 0.04166667 0.02083333 0.03125    0.0625     0.08333333 0.07291667
 0.0625     0.10416667 0.08333333 0.08333333 0.07291667 0.10416667
 0.08333333 0.07291667 0.0625     0.02083333 0.08333333 0.10416667
 0.10416667 0.10416667]
Mean error rate:  0.07403273809523811
Group:  [1, 3, 4, 5, 6, 7] Error rate:  0.020833333333333332
```

[ ]:
```
def get_w_ridge(randoms, V, S, lambda_matrix, U, y):
    new_u,new_y = get_u_and_y(randoms,U,y)
    return V@np.linalg.inv((S@S) + lambda_matrix)@S@new_u.transpose()@new_y
```

[ ]:
```
# b)
lambdas = np.array([0, 2**(-1), 1, 2, 2**2, 2**3, 2**4])

U,s,VT = np.linalg.svd(X,full_matrices=False)
S = np.arange(81).reshape(9,9)
S_matrix = np.zeros_like(S)
np.fill_diagonal(S_matrix, s)

aux = np.arange(81).reshape(9,9)
lambda_complete = np.float_(np.zeros_like(aux))

min_error_rate_ridge = np.array([None,None,None,None,None,None,None])
avg_error_rate_ridge = np.float_(np.array([0,0,0,0,0,0,0]))
```

```
min_random_group_ridge = np.array([[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],

for i in range(len(lambdas)):
    error_rates_ridge = np.float_(np.arange(56))
    for k in range(56):
        misclassifications = 0
        randoms = select_randoms()
        j=0
        np.fill_diagonal(lambda_complete, [lambdas[i]]*9)
        w = get_w_ridge(randoms,VT.transpose(),S_matrix_inverse,lambda_complete,U,y)
        y_hat = np.sign(X@w)
        aux = y_hat - y
        for value in aux:
            if value != 0:
                misclassifications += 1
        error_rates_ridge[k] = misclassifications/96
        if min_error_rate_ridge[i] == None or error_rates_ridge[k] < min_error_rate_ridge[i]:
            min_error_rate_ridge[i] = error_rates[k]
            min_random_group_ridge[i] = randoms
    avg_error_rate_ridge[i] = error_rates_ridge.mean()

for i in range(len(min_error_rate_ridge)):
    print("Lambda = ", i , ": \t\t Group: ", min_random_group_ridge[i], " \nError rate: ", min_error_rate_ridge[i], " \t Mean error rate: ", avg_error_rate_ridge[i])
    print()
```

```
Lambda =  0 :            Group:  [0 2 3 4 7 8]
Error rate:  0.08333333333333333         Mean error rate:  0.2533482142857143

Lambda =  1 :            Group:  [0 1 2 3 4 5]
Error rate:  0.020833333333333332        Mean error rate:  0.056919642857142856

Lambda =  2 :            Group:  [0 1 2 4 5 6]
Error rate:  0.020833333333333332        Mean error rate:  0.056547619047619055

Lambda =  3 :            Group:  [1 2 5 6 7 8]
Error rate:  0.020833333333333332        Mean error rate:  0.06008184523809524

Lambda =  4 :            Group:  [0 2 4 5 6 8]
Error rate:  0.10416666666666667         Mean error rate:  0.060267857142857144

Lambda =  5 :            Group:  [1 3 4 5 6 7]
Error rate:  0.020833333333333332        Mean error rate:  0.05970982142857143
```

```
Lambda =  6 :            Group:  [1 2 3 6 7 8]
Error rate:  0.020833333333333332        Mean error rate:  0.07031249999999999
```