HW05 - CS 189

1.
Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

I worked on this homework with Ehimare Okoyomon, Prashanth Ganeth, and Daniel Mockaitis. We worked by getting together throughout the week and communicating on facebook.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up}

Nicholas Lorio, 26089160

Question 2 E

In [30]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc import imread,imsave

imFile = 'stpeters_probe_small.png'
compositeFile = 'tennis.png'
targetFile = 'interior.jpg'

# This loads and returns all of the images needed for the problem
# data - the image of the spherical mirror
# tennis - the image of the tennis ball that we will relight
# target - the image that we will paste the tennis ball onto
def loadImages():
    imFile = 'stpeters_probe_small.png'
    compositeFile = 'tennis.png'
    targetFile = 'interior.jpg'

    data = imread(imFile).astype('float')*1.5
    tennis = imread(compositeFile).astype('float')
    target = imread(targetFile).astype('float')/255

    return data, tennis, target


# This function takes as input a square image of size m x m x c
# where c is the number of color channels in the image.  We
# assume that the image contains a scphere and that the edges
# of the sphere touch the edge of the image.
# The output is a tuple (ns, vs) where ns is an n x 3 matrix
# where each row is a unit vector of the direction of incoming light
# vs is an n x c vector where the ith row corresponds with the
# image intensity of incoming light from the corresponding row in ns
def extractNormals(img):

    # Assumes the image is square
    d = img.shape[0]
    r = d / 2
    ns = []
    vs = []
    for i in range(d):
        for j in range(d):

            # Determine if the pixel is on the sphere
            x = j - r
            y = i - r
            if x*x + y*y > r*r-100:
                continue

            # Figure out the normal vector at the point
            # We assume that the image is an orthographic projection
```

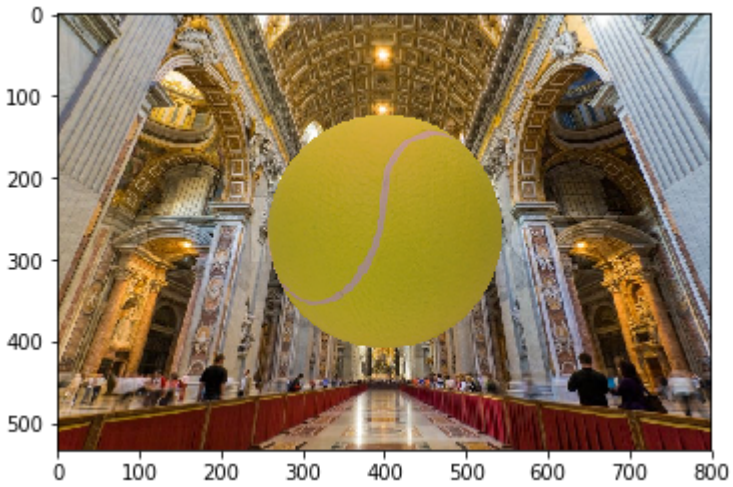
```
            z = np.sqrt(r*r-x*x-y*y)
            n = np.asarray([x,y,z])
            n = n / np.sqrt(np.sum(np.square(n)))
            view = np.asarray([0,0,-1])
            n = 2*n*(np.sum(n*view))-view
            ns.append(n)
            vs.append(img[i,j])

    return np.asarray(ns), np.asarray(vs)

# This function renders a diffuse sphere of radius r
# using the spherical harmonic coefficients given in
# the input coeff where coeff is a 9 x c matrix
# with c being the number of color channels
# The output is an 2r x 2r x c image of a diffuse sphere
# and the value of -1 on the image where there is no sphere
def renderSphere(r,coeff):

    d = 2*r
    img = -np.ones((d,d,3))
    ns = []
    ps = []

    for i in range(d):
        for j in range(d):

            # Determine if the pixel is on the sphere
            x = j - r
            y = i - r
            if x*x + y*y > r*r:
                continue

            # Figure out the normal vector at the point
            # We assume that the image is an orthographic projection
            z = np.sqrt(r*r-x*x-y*y)
            n = np.asarray([x,y,z])
            n = n / np.sqrt(np.sum(np.square(n)))
            ns.append(n)
            ps.append((i,j))

    ns = np.asarray(ns)
    B = computeBasis(ns)
    vs = B.dot(coeff)

    for p,v in zip(ps,vs):
        img[p[0],p[1]] = np.clip(v,0,255)

    return img

# relights the sphere in img, which is assumed to be a square image
# coeff is the matrix of spherical harmonic coefficients
def relightSphere(img, coeff):
    img = renderSphere(int(img.shape[0]/2),coeff)/255*img/255
    return img

# Copies the image of source onto target
# pixels with values of -1 in source will not be copied
```

```
def compositeImages(source, target):

    # Assumes that all pixels not equal to 0 should be copied
    out = target.copy()
    cx = int(target.shape[1]/2)
    cy = int(target.shape[0]/2)
    sx = cx - int(source.shape[1]/2)
    sy = cy - int(source.shape[0]/2)

    for i in range(source.shape[0]):
        for j in range(source.shape[1]):
            if np.sum(source[i,j]) >= 0:
                out[sy+i,sx+j] = source[i,j]

    return out

# Fill in this function to compute the basis functions
# This function is used in renderSphere()
def computeBasis(ns):
    # Returns the first 9 spherical harmonic basis functions

    ##################################################
    B = np.ones((len(ns), 9))
    for i in range(len(ns)):
        x,y,z = ns[i] / np.linalg.norm(ns[i])
        B[i] = np.array([1,
                         y,
                         x,
                         z,
                         x*y,
                         y*z,
                         3*z**2 - 1,
                         x*z,
                         x**2 - y**2])
    ##################################################
    return B

#if __name__ == '__main__':

data,tennis,target = loadImages()
ns, vs = extractNormals(data)
B = computeBasis(ns)


# reduce the number of samples because computing the SVD on
# the entire data set takes too long
Bp = B[::50]
vsp = vs[::50]

##################################################
# TODO: Solve for the coefficients using least squares
# or total least squares here
print("Starting Least Squares")
w = np.linalg.lstsq(Bp, vsp)

#Total Least Sqaures
# w = np.solve(np.transpose(Bp) @ Bp - sigma2 @ np.ones(len(Bp[0])) = np.tr
```

```
anspose(Bp) @ svp )


####################################################
coeff = w[0]

#coeff[0,:] = 255

img = relightSphere(tennis,coeff)

output = compositeImages(img,target)

print('Coefficients:\n'+str(coeff))

plt.figure(1)
plt.imshow(output)
plt.show()

imsave('output.png',output)
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:18: DeprecationW
arning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:19: DeprecationW
arning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:20: DeprecationW
arning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```

```
Starting Least Squares
Coefficients:
[[ 202.31845431  162.41956802  149.07075034]
 [ -27.66555164  -17.88905339  -12.92356688]
 [  -5.15203925   -4.51375871   -4.24262639]
 [  -1.08629293    0.42947012    1.15475569]
 [  -3.14053107   -3.70269907   -3.74382934]
 [  23.67671768   23.15698002   21.94638397]
 [  -3.82167171    0.57606634    1.81637483]
 [   4.7346737     1.4677692    -1.12253649]
 [  -9.72739616   -5.75691108   -4.8395598 ]]
```

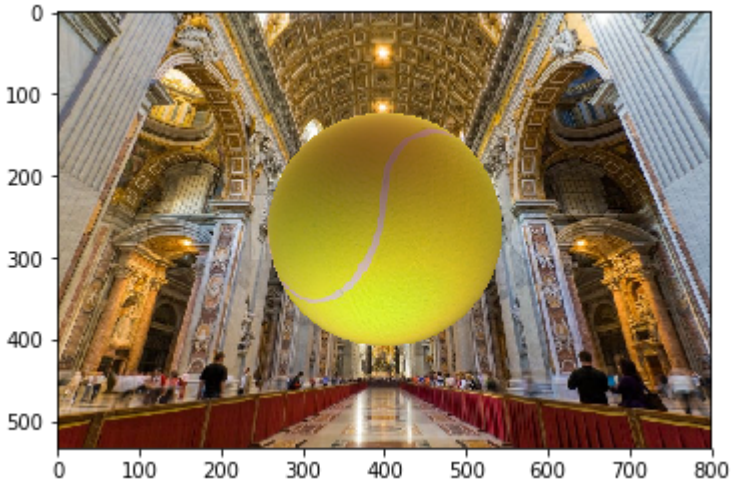



```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:182: Deprecation
Warning: `imsave` is deprecated!
`imsave` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imwrite`` instead.
```

Question 2 F

In [3]:
```
# TODO: Solve for the coefficients using least squares
# or total least squares here
print("Starting Total Least Squares")

#Total Least Squares
S = np.linalg.svd(np.hstack((Bp, vsp)), compute_uv = False)[-1]**2
w = np.linalg.inv(np.transpose(Bp).dot(Bp) -  S*np.eye(9)).dot(np.transpose(Bp
).dot(vsp))

##################################################
coeff = w

#coeff[0,:] = 255

img = relightSphere(tennis,coeff)

output = compositeImages(img,target)

print('Coefficients:\n'+str(coeff))

plt.figure(1)
plt.imshow(output)
plt.show()

imsave('output.png',output)
```

```
Starting Total Least Squares
Coefficients:
[[ 214.8315375   172.45812442  158.28249759]
 [ -33.32866277  -21.49356942  -15.49471908]
 [  -6.10484778   -5.35738272   -5.05359386]
 [  -1.36017113    0.46289619    1.34025806]
 [ -25.34212773  -29.68688704  -29.89218111]
 [ 195.27624821  190.46787726  180.03847995]
 [  -4.14989152    0.58493533    1.92140102]
 [  43.5067045    16.78214693   -4.59898962]
 [ -12.45808493   -7.37757515   -6.20446144]]
```

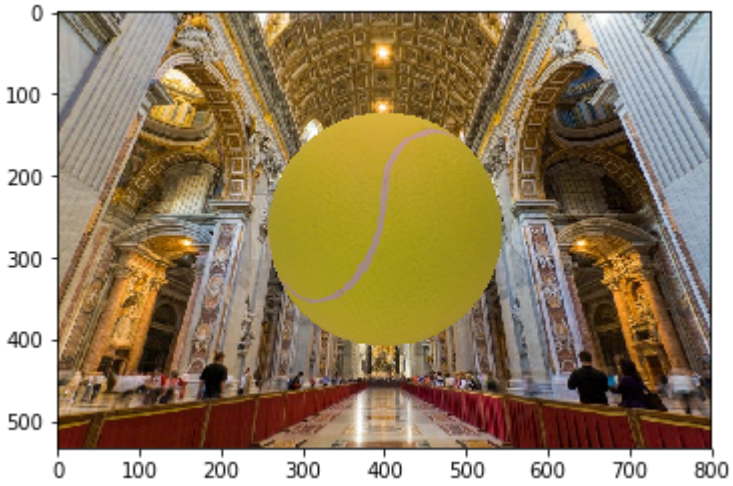



```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:24: DeprecationW
arning: `imsave` is deprecated!
`imsave` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imwrite`` instead.
```

Question 2 G

In [9]:
```
# TODO: Solve for the coefficients using least squares
# or total least squares here
print("Starting Total Least Squares")

#Total Least Squares
vsp_adj = vsp/np.max(vsp)
S = np.linalg.svd(np.hstack((Bp, vsp_adj)), compute_uv = False)[-1]**2
w = np.linalg.inv(np.transpose(Bp).dot(Bp) -  S*np.eye(9)).dot(np.transpose(Bp
).dot(vsp_adj))

##################################################
coeff = w

#coeff[0,:] = 255

img = relightSphere(tennis,coeff) * np.max(vsp)

output = compositeImages(img,target)

print('Coefficients:\n'+str(coeff))

plt.figure(1)
plt.imshow(output)
plt.show()

imsave('output.png',output)
```

```
Starting Total Least Squares
Coefficients:
[[ 0.52901341  0.42468755  0.38978365]
 [-0.07235944 -0.04678891 -0.03380164]
 [-0.01347512 -0.01180571 -0.01109657]
 [-0.00284099  0.00112345  0.00302042]
 [-0.00822827 -0.00970116 -0.00980891]
 [ 0.06203423  0.06067251  0.05750066]
 [-0.00999302  0.00150638  0.00474959]
 [ 0.01240554  0.00384608 -0.00294068]
 [-0.02544484 -0.01505888 -0.01265929]]
```




```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:25: DeprecationW
arning: `imsave` is deprecated!
`imsave` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imwrite`` instead.
```

Question 3 H

In [73]:
```
import numpy as np
import matplotlib.pyplot as plt
# %matplotlib inline
import sklearn.linear_model
from sklearn.model_selection import train_test_split


######## PROJECTION FUNCTIONS ##########

## Random Projections ##
def random_matrix(d, k):
    '''
    d = original dimension
    k = projected dimension
    '''
    return 1./np.sqrt(k)*np.random.normal(0, 1, (d, k))

def random_proj(X, k):
    _, d= X.shape
    return X.dot(random_matrix(d, k))
```

```
## PCA and projections ##
def my_pca(X, k):
    '''
    compute PCA components
    X = data matrix (each row as a sample)
    k = #principal components
    '''
    n, d = X.shape
    assert(d>=k)
    _, _, Vh = np.linalg.svd(X)
    V = Vh.T
    return V[:, :k]

def pca_proj(X, k):

    '''
    compute projection of matrix X
    along its first k principal components
    '''
    P = my_pca(X, k)
     # P = P.dot(P.T)
    return X.dot(P)




######### LINEAR MODEL FITTING #############

def rand_proj_accuracy_split(X, y, k):
    '''
    Fitting a k dimensional feature set obtained
    from random projection of X, versus y
    for binary classification for y in {-1, 1}
    '''

    # test train split
    _, d = X.shape
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
random_state=42)

    # random projection
    J = np.random.normal(0., 1., (d, k))
    rand_proj_X = X_train.dot(J)

    # fit a linear model
    line = sklearn.linear_model.LinearRegression(fit_intercept=False)
    line.fit(rand_proj_X, y_train)

    # predict y
    y_pred=line.predict(X_test.dot(J))

    # return the test error
    return 1-np.mean(np.sign(y_pred)!= y_test)

def pca_proj_accuracy(X, y, k):
```

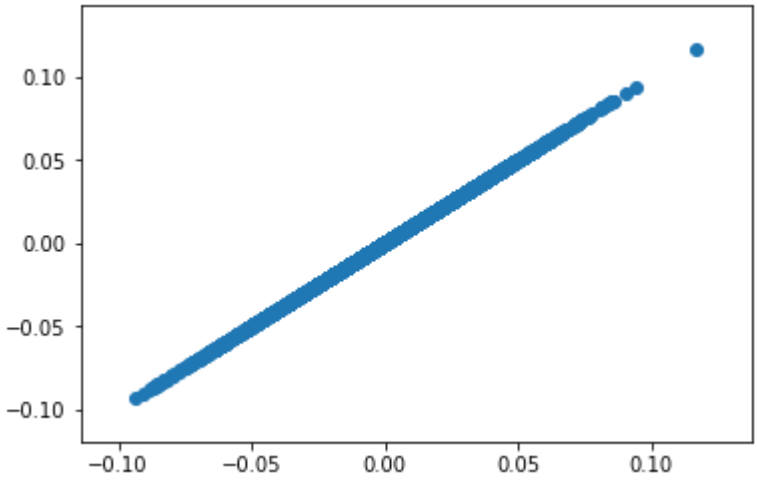
```
    '''
    Fitting a k dimensional feature set obtained
    from PCA projection of X, versus y
    for binary classification for y in {-1, 1}
    '''

    # test-train split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)

    # pca projection
    P = my_pca(X_train, k)
    P = P.dot(P.T)
    pca_proj_X = X_train.dot(P)

    # fit a linear model
    line = sklearn.linear_model.LinearRegression(fit_intercept=False)
    line.fit(pca_proj_X, y_train)

     # predict y
    y_pred=line.predict(X_test.dot(P))


    # return the test error
    return 1-np.mean(np.sign(y_pred)!= y_test)


######## LOADING THE DATASETS #########

# to load the data:
# data = np.load('data/data1.npz')
# X = data['X']
# y = data['y']
# n, d = X.shape

data1 = np.load('data1.npz')
data2 = np.load('data2.npz')
data3 = np.load('data3.npz')

X1 = data1['X']
y1 = data1['y']
n1, d1 = X1.shape

X2 = data2['X']
y2 = data2['y']
n2, d2 = X2.shape

X3 = data3['X']
y3 = data3['y']
n3, d3 = X3.shape

# n_trials = 10  # to average for accuracies over random projections

######### YOUR CODE GOES HERE ##########

# Using PCA and Random Projection for:
# Visualizing the datasets
```

```
#k = 2
def visualize(X, k):
    pca_projection =pca_proj(X, k)
    x = np.zeros(len(pca_projection))
    y = np.zeros(len(pca_projection))
    for i in range(len(pca_projection)):
        x[i] = pca_projection[i][0]
        y[i] = pca_projection[i][0]
    return x, y

def visualize_random(X, k):
    pca_rand_proj = random_proj(X, k)
    x = np.zeros(len(pca_rand_proj))
    y = np.zeros(len(pca_rand_proj))
    for i in range(len(pca_rand_proj)):
        x[i] = pca_rand_proj[i][0]
        y[i] = pca_rand_proj[i][0]
    return x, y


v1_x, v1_y = visualize(X1, 2)
v1r_x, v1r_y = visualize_random(X1, 2)

v2_x, v2_y = visualize(X2, 2)
v2r_x, v2r_y = visualize_random(X2, 2)

v3_x, v3_y = visualize(X3, 2)
v3r_x, v3r_y = visualize_random(X3, 2)


# Computing the accuracies over different datasets.


def compute_accuracy(X, y, k):
    accuracy_pca = pca_proj_accuracy(X, y, k)
    r = np.zeros(10)
    for i in range(10):
        r[i] = rand_proj_accuracy_split(X, y, k)
    accuracy_rand_pca = np.average(r)

    return accuracy_pca, accuracy_rand_pca

def plot_accuracy(X, y):
    d = X.shape[1]
    A = np.zeros(d)
    A_rand = np.zeros(d)
    for i in range(d):
        accuracies = compute_accuracy(X, y, i+1)
        A[i] = accuracies[0]
        A_rand[i] = accuracies[1]
    #print(A)
    plt.plot(range(1, d+1), A)
    plt.plot(range(1, d+1), A_rand, color='red')
    plt.show()
    return
```

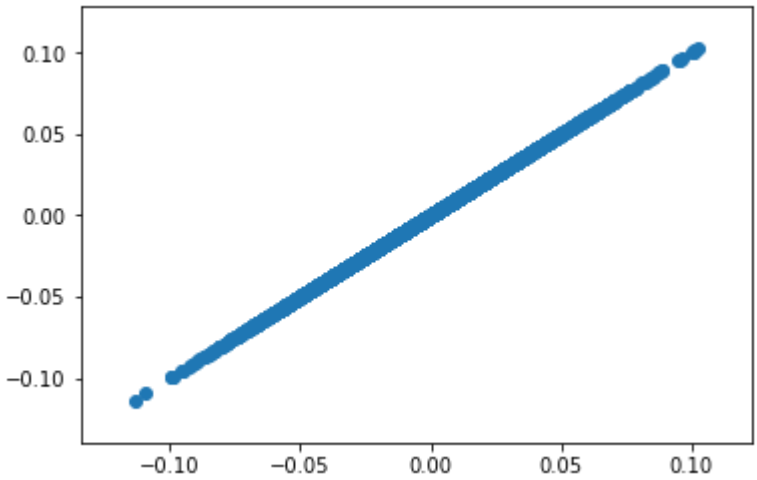
```
# Don't forget to average the accuracy for multiple
# random projections to get a smooth curve.




# And computing the SVD of the feature matrix

def compute_plot_svd(X):
    svd = np.linalg.svd(X, compute_uv = False)
    print("Singular Value Plot")
    plt.bar(range(1, len(svd)+1), svd)
    plt.show()
######## YOU CAN PLOT THE RESULTS HERE ########

# plt.plot, plt.scatter would be useful for plotting


#PLOT PART H
def plot_partH():
    plt.scatter(v1_x, v1_y)
    print("Data1 Projection")
    plt.show()
    plt.scatter(v1r_x, v1r_y)
    print("Data1 Random Projection")
    plt.show()

    plt.scatter(v2_x, v2_y, color='red')
    print("Data2 Projection")
    plt.show()
    plt.scatter(v2r_x, v2r_y, color = 'red')
    print("Data2 Random Projection")
    plt.show()

    plt.scatter(v3_x, v3_y, color = 'orange')
    print("Data3 Projection")
    plt.show()
    plt.scatter(v3r_x, v3r_y, color = 'orange')
    print("Data3 Random Projection")
    plt.show()
    return

#PLOT PART I
def plot_partI():

    print("Red is for Random_Projection Accuracy Errors")
    print("Blue is for PCA Accuracy Errors ")
    plot_accuracy(X1, y1)
    plot_accuracy(X2, y2)
    plot_accuracy(X3, y3)
    return

def plot_partJ():
```

```
    print('X1')
    compute_plot_svd(X1)
    print('X2')
    compute_plot_svd(X2)
    print('X3')
    compute_plot_svd(X3)

plot_partH()
plot_partI()
plot_partJ()
```

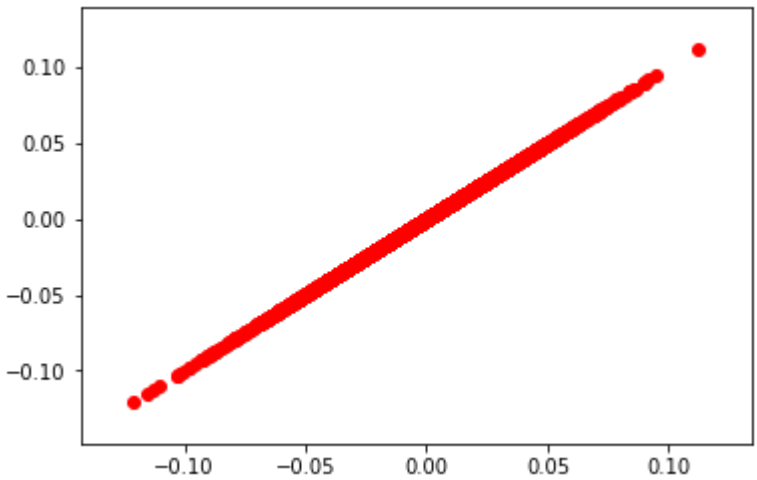
Data1 Projection



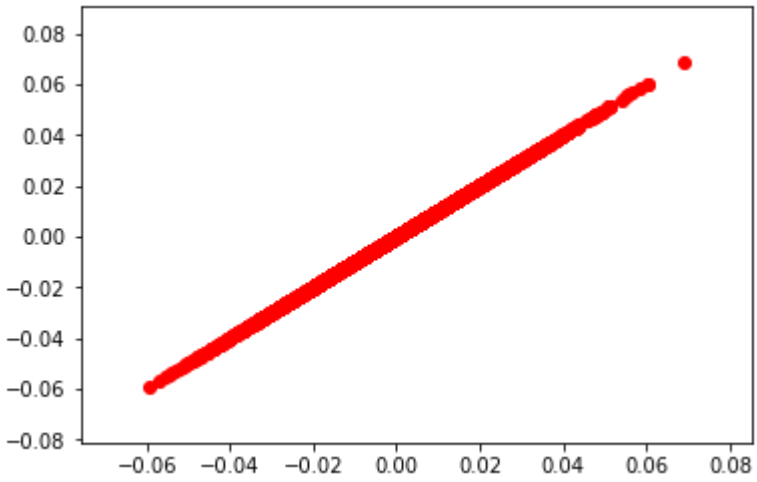
Data1 Random Projection



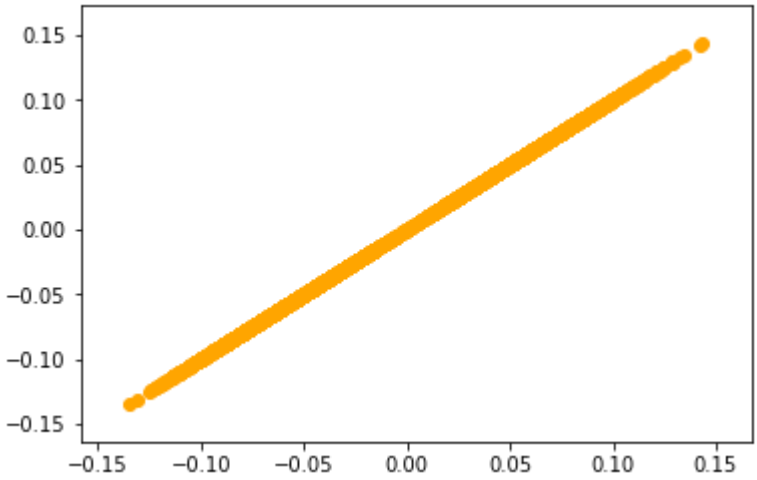
Data2 Projection



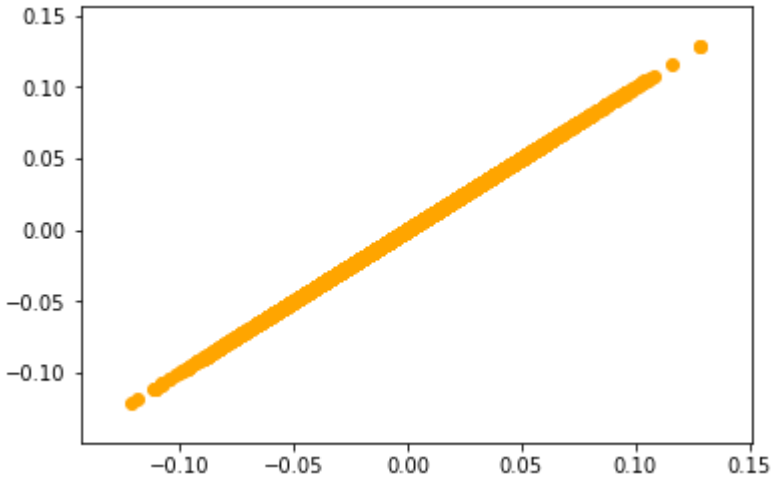
Data2 Random Projection



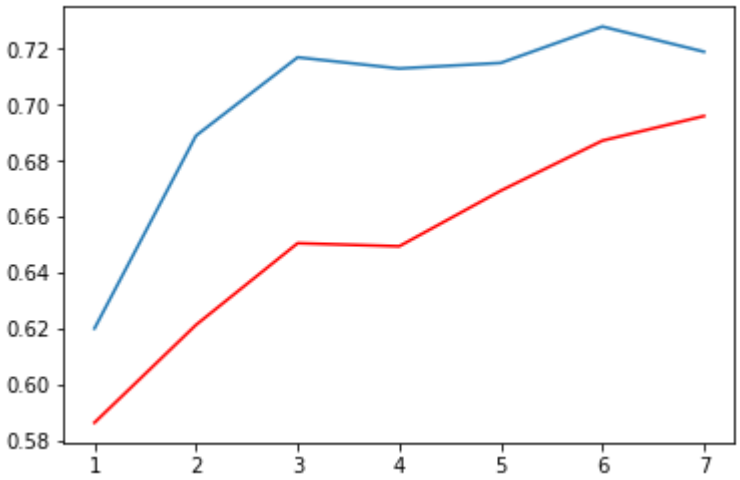
Data3 Projection

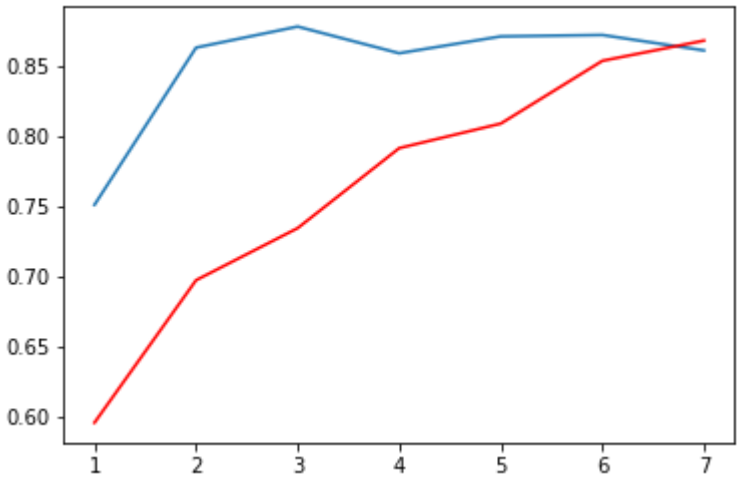


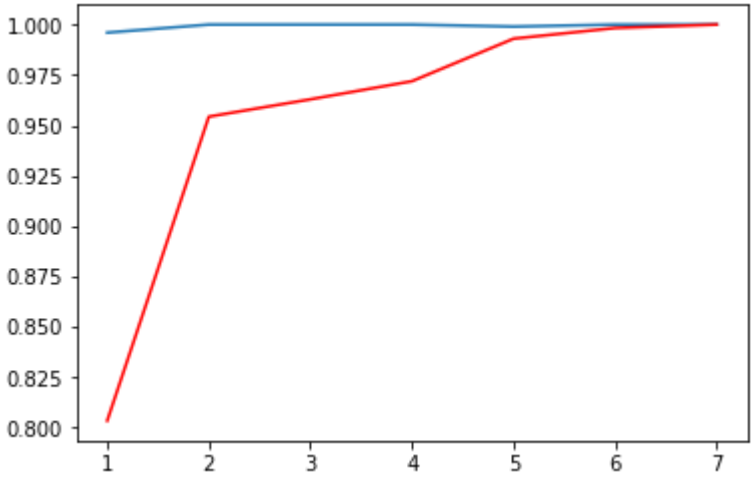
Data3 Random Projection



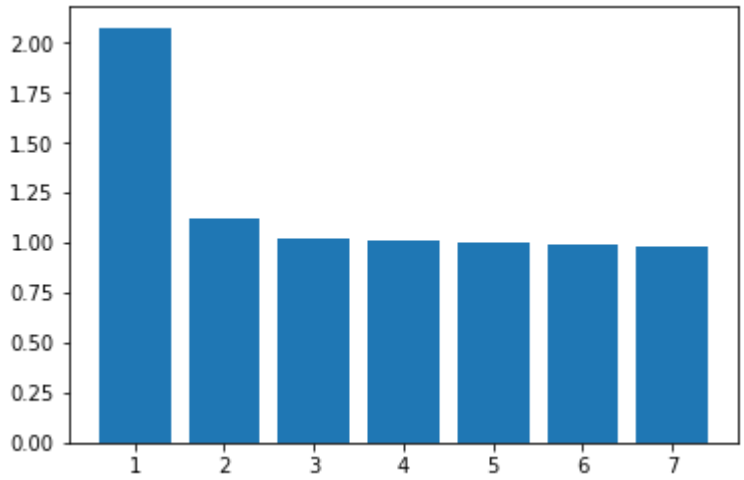
Red is for Random_Projection Accuracy Errors
Blue is for PCA Accuracy Errors



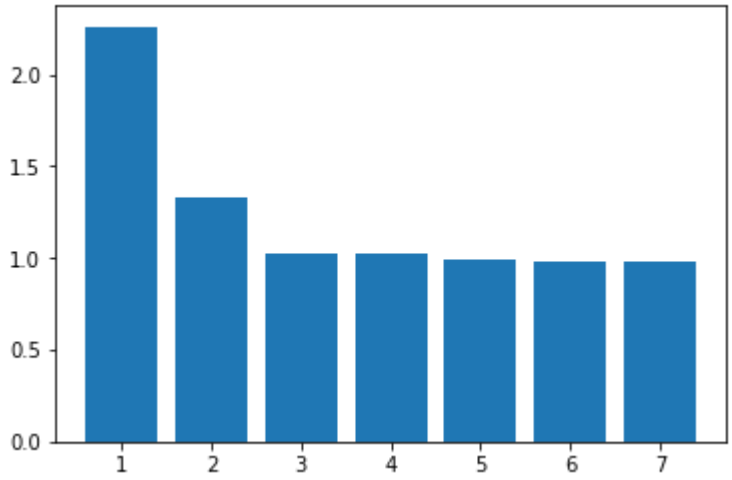




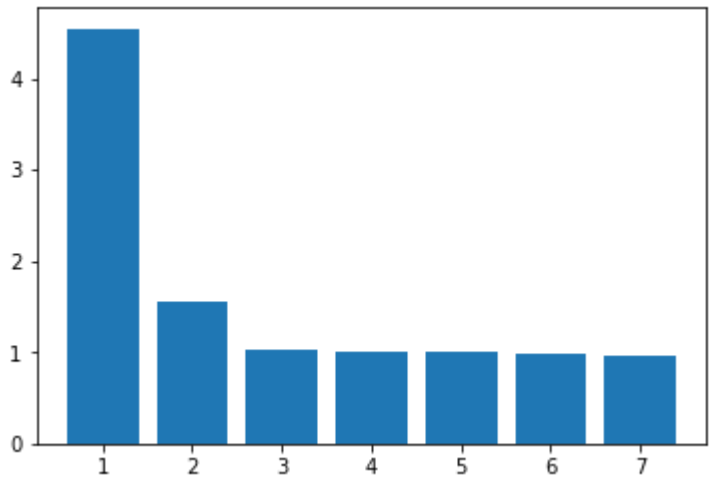
X1
Singular Value Plot



X2
Singular Value Plot



X3
Singular Value Plot



Hwes 189

Lorio

2. $X \in \mathbb{R}^{n \times d}$ $y \in \mathbb{R}^n$ $E_X \in \mathbb{R}^{n \times d}$ $E_y \in \mathbb{R}^n$

case $n > d$, $\text{rank}(X + E_X) = d$ explain $\text{rank}([X + E_X, y + E_y]) = d$

$X_{true} = X + E_X$

$y_{true} = y + E_y$

$X \begin{bmatrix} n \times d \\ \end{bmatrix}$ $y \begin{bmatrix} n \times 1 \\ \end{bmatrix}$

$E_X \begin{bmatrix} n \times d \\ \end{bmatrix}$ $E_y \begin{bmatrix} n \times 1 \\ \end{bmatrix}$

$X + E_X \cdot \begin{bmatrix} n \times d \\ \end{bmatrix}$ • has d linearly independent columns
• it is a d dimensional space

$\left[\begin{bmatrix} n \times d \end{bmatrix}, \begin{bmatrix} n \times 1 \end{bmatrix} \right]$

this column is a linear combination of the preceeding d columns since the resultant matrix $[X + E_X, y + E_y]$ has rank d.

• we have that $\min_{E_y E_X} \|E_X, E_y\|_F^2$ s.t. $(X + E_X)w = y + E_y$, $w_d = -1$

or

save

s.t. $[X_{true}, y_{true}] \begin{bmatrix} w \\ -1 \end{bmatrix} = 0$

• the minimized TLS constraint shows the preceeding comment regarding linear dependence to be true.

2.b. Show that $[\varepsilon_x, \varepsilon_y] = -\begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} \cdot \sigma_{d+2} \cdot \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}^T$

$$[\chi + \varepsilon_x, \vec{\gamma} + \varepsilon_y] = U \begin{bmatrix} \sigma_d & 0 \\ 0 & 0 \end{bmatrix} V^T = \begin{bmatrix} \vec{v}_{xx} & \vec{v}_{xy} \\ v_{yx} & v_{yy} \end{bmatrix} \begin{bmatrix} \sigma_d & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \vec{v}_{xx} & \vec{v}_{xy} \\ v_{yx} & v_{yy} \end{bmatrix}^T$$

$$[\chi, \vec{\gamma}] = \begin{bmatrix} \vec{v}_{xx} & \vec{v}_{xy} \\ v_{yx} & v_{yy} \end{bmatrix} \begin{bmatrix} \sigma_d & \\ & \sigma_{d+2} \end{bmatrix} \begin{bmatrix} \vec{v}_{xx} & \vec{v}_{xy} \\ v_{yx} & v_{yy} \end{bmatrix}^T$$

$(d+1)^{th}$ eigenvalue of $[\chi, \gamma]$ $\in \mathbb{R}^{d+2}$ first d elements of the $(d+1)^{th}$ column respectively

i) $[\chi, \vec{\gamma}] + [\varepsilon_x, \varepsilon_y] = [\chi + \varepsilon_x, \gamma + \varepsilon_y]$

i) $[\chi, \gamma]$

(ii) $-$(i) gives us $[\varepsilon_x, \varepsilon_y]$

$$= -[\varepsilon_x, \varepsilon_y] = \begin{bmatrix} \vec{v}_{xx} & \vec{v}_{xy} \\ v_{yx} & v_{yy} \end{bmatrix} \left(\begin{bmatrix} \sigma_d & \\ & \sigma_{d+2} \end{bmatrix} - \begin{bmatrix} \sigma_d & \\ & 0 \end{bmatrix} \right) \begin{bmatrix} \vec{v}_{xx} & \vec{v}_{xy} \\ v_{yx} & v_{yy} \end{bmatrix}^T$$

$$= -\begin{bmatrix} a & b \\ c & \end{bmatrix} \begin{bmatrix} 0 & \\ & \sigma_{d+1} \end{bmatrix} \begin{bmatrix} \\ \\ \end{bmatrix}^T$$

$$[\varepsilon_x, \varepsilon_y] = -\begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} \sigma_{d+2} \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}^T$$ "as other terms will be zero"

$$\begin{bmatrix} x+\varepsilon_x, & y+\varepsilon_y \end{bmatrix} \begin{bmatrix} w \\ -1 \end{bmatrix} = 0$$

2.c. (Hint $v_{yy} \neq 0$) & V is a right singular vector

of $[x \; y]$ ∴ it is an eigenvector of the Matrix.

"From Notes" $\begin{bmatrix} w \\ -1 \end{bmatrix} = \alpha V_i$, $V = \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}$

$$[x \; y] V \Rightarrow [x \; y] \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} = \sigma_{last} \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}$$

last, bcuz ⊕ last column of product.

Subst. eqn from 2.b ~~$A x = \lambda x$~~

insert $[x \; y]$ expression works out to this ↓

$$[\varepsilon_x \; \varepsilon_y] = -[x \; y] \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}^T$$

$$[x \; y] + [\varepsilon_x \; \varepsilon_y] = [x \; y] - [x \; y] \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}^T$$

$$[x+\varepsilon_x, \; y+\varepsilon_y] \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} = [x \; y] \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} - [x \; y] \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} = 0$$

∴ $\begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}$ is a solution & $\vec{w} =$

$(x+\varepsilon_x) w - (y+\varepsilon_y) = 0 \qquad w = \dfrac{y+\varepsilon_y}{x+\varepsilon_x}$

$(x+\varepsilon_x) \vec{v}_{xy} - (y+\varepsilon_y) v_{yy} = 0$

$\vec{v}_{xy} = \dfrac{(y+\varepsilon_y)}{(x+\varepsilon_x)} v_{yy} \qquad \vec{v}_{xy} = w v_{yy}$ $\boxed{w = \vec{v}_{xy} v_{yy}^{-1}}$

$\begin{bmatrix} w \\ -1 \end{bmatrix} = \alpha V$ right singular vector of $[x \; y]$

2.d. $v_{yy} \neq 0$, $[\varepsilon_x, \varepsilon_y] = -\begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} \sigma_{d+1} \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}$

Find Nontrivial sol'n for $[x+\varepsilon_x, y+\varepsilon_y]\begin{bmatrix} w \\ -1 \end{bmatrix} = 0$ & thus solve for $\underline{w}$.

$$(x+\varepsilon_x)w - (y+\varepsilon_y) = 0$$

$$\min_{\varepsilon_x \varepsilon_y} \left\| [\varepsilon_x, \varepsilon_y] \right\|_F^2 \quad s.t \quad [x+\varepsilon_x, y+\varepsilon_y]\begin{bmatrix} w \\ -1 \end{bmatrix} = 0$$

Show that $(x^T x - \sigma_{d+1}^2 I)w = x^T y$

First d components give us w, $[w \ -1]$ is a right singular vector of $[x\,y]$ & corresponds to v_{d+1}

$$[x\,y]^T [x\,y] = \begin{bmatrix} x^T x & x^T y \\ y^T x & y^T y \end{bmatrix}$$

$$\begin{bmatrix} x^T x & x^T y \\ y^T x & y^T y \end{bmatrix}\begin{bmatrix} w \\ -1 \end{bmatrix} = \sigma_{d+1}^2 \begin{bmatrix} w \\ -1 \end{bmatrix}$$

$$\rightsquigarrow x^T x w - x^T y = \sigma_{d+1}^2 w$$

$$\rightsquigarrow (x^T x - \sigma_{d+1}^2 I) w = x^T y$$

$$\begin{bmatrix} d\times n \\ v_1 \cdots v_n \end{bmatrix} \begin{bmatrix} \sigma_1^2 & & \\ & \ddots & \\ & & \sigma_n^2 \end{bmatrix} \begin{bmatrix} n\times d \\ v_1 \cdots v_d \end{bmatrix}$$

3. $[n] \sim \{1, \ldots, n\}$

a. $XX^T = U\Sigma V^T (U\Sigma V^T)^T = U\Sigma V^T V \Sigma^T U^T$
$= U\Sigma\Sigma^T U^T$

$(U\Sigma\Sigma^T U^T)_{ij} = \sigma_{ij}^2 \cdot U_i U_j^T$

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

$(V\Sigma^T\Sigma V^T)_{ij} = \sigma_{ij}^2 V_i V_j^T$

$$XX^T = \begin{bmatrix} - & x_1^T & - \\ & \vdots & \\ - & x_n^T & - \end{bmatrix} \begin{bmatrix} - & x_1 & - \\ & \vdots & \\ - & x_n^T & - \end{bmatrix}^T$$

$$= \begin{bmatrix} - & x_1^T & - \\ & \vdots & \\ - & x_n^T & - \end{bmatrix} \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix} = \begin{bmatrix} x_1^T x_1 & \cdots & x_n^T x_n \\ & & \end{bmatrix}$$

$n\times d$ $\quad d\times n$ $\quad n\times n$

$(XX^T)_{ij} = x_i^T x_j$

$$X^T X = \begin{bmatrix} d\times n \\ x_1 \cdots x_n \end{bmatrix} \begin{bmatrix} n\times d \\ - x_1^T - \\ \vdots \\ - x_n^T - \end{bmatrix} = \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$$

$x_{11}\ x_{21}\ x_{31}$

$(X^T X)_{ij} = \sum_{b=1}^{n} (x_{b,i} \cdot x_{b,j}^T)$ $\quad \left(\begin{array}{l} x_{b,i} \text{ } b\text{th element of } i\text{th} \\ b\text{th column} \end{array} \right.$

3.b. $\psi_{PCA}(x_i)^T \psi_{PCA}(x_j) = x_i^T V_k V_k^T x_j$

$\psi_{PCA}(x_i) = (v_1^T x, \ldots, v_k^T x)^T = \begin{bmatrix} - v_1^T x - \\ \vdots \\ - v_k^T x - \end{bmatrix}$

$\begin{bmatrix} v_1^T x_i & \ldots & v_k^T x_i \end{bmatrix}$

Similar to $X^T X$: $\left(\psi_{PCA}(x)^T \psi_{PCA}(x_i)\right)_{ij} = \sum_{b=1}^{k} (x_i^T v_{i,b}) \cdot (v_i^T x_j)$

$= \sum_{b=1}^{k} \left(x_i^T v_{i,b} v_{i,b}^T x_j\right)$

$= x_i^T \sum_{b=1}^{k} \left(v_{i,b} v_{i,b}^T\right) x_j$

$= x_i^T V_k V_k^T x_j$

$V_k = \begin{bmatrix} v_1 & v_2 & \ldots & v_k \end{bmatrix}$

also show $V_k V_k^T = \underline{V I^k V^T}$

$I^k = \begin{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}_{k \times k} & 0_{k \times d} \\ 0_{d \times k} & 0_{d \times d} \end{bmatrix}$ [b?]

$\begin{bmatrix} | & & | \\ v_1 & \ldots & v_k \\ | & & | \end{bmatrix} \begin{bmatrix} - v_1^T - \\ \vdots \\ - v_k^T - \end{bmatrix} = \begin{bmatrix} | & & | \\ v_1 & \ldots & v_d \\ | & & | \end{bmatrix} \begin{bmatrix} 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \\ 0 & \ldots & 0 & 0_{d \times d} \end{bmatrix} \begin{bmatrix} - v_1^T - \\ \vdots \\ - v_d^T - \end{bmatrix}$

It just is, trivially derived.

$V I^k V^T = V I^k I^k V^T$

3.c. $\frac{\sum_{i=1}^{k} \sigma_i^4}{\sum_{i=1}^{d} \sigma_i^4} \geq 1 - \varepsilon \qquad \varepsilon \in (0,1)$

def^ Frobenius norm: $\|A\|_F = \sqrt{\mathrm{Tr}(AA^T)}$

$= \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij}|^2}$

from parts.

$$\frac{1}{\sum_{i=1}^{\hat{n}}\sum_{j=1}^{\hat{n}} (x_i^T x_j)^2} \left\| (XX^T) - X V_k V_k^T X^T \right\|_F^2 \leq \varepsilon$$

(i) $\sum_{i=}^{\hat{}}\sum_{j=1}^{\hat{}} (x_i^T x_j)^2 = \sum(\text{entries of } XX^T)^2 = \sum \mathrm{Trace}((XX^T)^T(XX^T))$

$= \sum$ eigenvalues of $(XX^T)^T(XX^T) = \sum$

$= \sum$ eigenval $(U\Sigma\Sigma^T U^T)^T(U\Sigma\Sigma^T U^T) = (U\Sigma^T\Sigma U^T)(U\Sigma\Sigma^T U^T)$

$= \sum$ eigenval $U\Sigma\Sigma^T U^T U\Sigma\Sigma^T U^T = \sum$ eigenval $U\Sigma\Sigma^T\Sigma\Sigma^T U^T$

Σ Symmetric & Diagonal $= \sum_{i=1}^{d} \sigma_i^4$

(ii) $\|XX^T - XV_kV_k^T X^T\|_F^2 = \mathrm{trace}\left[(XX^T - XV_kV_k^T X^T)^T(XX^T - XV_kV_k^T X^T)\right]$

$\mathrm{trace}\left[(XX^T)^T(XX^T) - (XX^T)(XV_kV_k^T X^T) - (XV_kV_k^T X^T)^T(XX^T) + (XV_kV_k^T X^T)^T(XV_kV_k^T X^T)\right]$

3.c. (cont) use (i), (ii), $XV_k V_k^T X^T = XVI_k V^T X^T$

$= U\Sigma I^k \Sigma^T U^T$

$XV = U\Sigma$

$\left[= \Sigma_k^2 \quad \text{each } \sigma_{k,i}^2 = \sigma_i^2 \text{ if } i > k \Rightarrow = 0 \right]$

$= U\Sigma_k^2 U^T$

$\therefore$ From (i)

(ii) $= \sum_{i=1}^{d} \sigma_i^4 + tr\left[-2\cdot(U\Sigma\Sigma^T U^T)(U\Sigma_k^2 U^T) + (U\Sigma_k^2 U^T)^T(U\Sigma_k^2 U^T)\right]$

$-2U\Sigma^4 U^T + U\Sigma_k^4 U^T = -U\Sigma^4 U^T$

$= \sum_{i=1}^{d} \sigma_i^4 - tr(U\Sigma^4 U^T) = \sum_{i=1}^{d} \sigma_i^4 - \sum_{i=1}^{k} \sigma_i^4$

Sum eigenvalues & squared

Sum squared eigenvalues

(all) $= \frac{1}{\sum_{i=1}^{d} \sigma_i^4}\left[\sum_{i=1}^{d} \sigma_i^4 - \sum_{i=1}^{k} \sigma_i^4\right] \leq \varepsilon$

$= 1 - \frac{\sum_{i=1}^{k} \sigma_i^4}{\sum_{i=1}^{d} \sigma_i^4} \leq \varepsilon$, $\left[1 - \varepsilon \leq \frac{\sum_{i=1}^{k} \sigma_i^4}{\sum_{i=1}^{d} \sigma_i^4}\right]$

//

s.t. $\varphi : \mathbb{R}^d \to \mathbb{R}^N$ $\quad 0 < \varepsilon < 1$ $\quad \|x_i\| \le 1$ $\quad i \in [n]$

$(1-\varepsilon)\|x_i\|_2^2 \le \|\varphi(x_i)\|_2^2 \le (1+\varepsilon)\|x_i\|_2^2$ $\quad i \in [n]$

$(1-\varepsilon)\|x_i - x_j\|_2^2 \le \|\varphi(x_i) - \varphi(x_j)\|_2^2 \le (1+\varepsilon)\|x_i - x_j\|_2^2$ $\quad \forall\, i,j \in [n]$

show $\quad |\varphi(x_i)^T \varphi(x_j) - (x_i^T x_j)| \le C \cdot \varepsilon$ $\quad i,j \in [n]$

from 8,

Substract inequality w/ x_i & x_j

$(1-\varepsilon)(\|x_i - x_j\|_2^2 - \|x_i\|^2 - \|x_j\|^2) \le -2\varphi(x_i)^T\varphi(x_j) \le (1+\varepsilon)(\|x_i - x_j\|_2^2 - \|x_i\|^2 - \|x_j\|^2)$

$(1-\varepsilon)(-2x_i^T x_j - 2) \le -2\varphi(x_i)^T\varphi(x_j) - 2 \le (1+\varepsilon)(-2x_i^T x_j - 2)$

lower

$-\varphi(x_i)^T\varphi(x_j) \le (1+\varepsilon)(-x_i^T x_j)$

$-\varphi(x_i)^T\varphi(x_j) \le -x_i^T x_j - \varepsilon x_i^T x_j$

$\varphi(x_i)^T\varphi(x_j) - x_i^T x_j \le \varepsilon x_i^T x_j$

upper

$(1-\varepsilon)(-x_i^T x_j) \le -\varphi(x_i)^T\varphi(x_j)$

$-x_i^T x_j + \varepsilon x_i^T x_j \le -\varphi(x_i)^T\varphi(x_j)$

$\varphi(x_i)^T\varphi(x_j) - x_i^T x_j \ge -\varepsilon x_i^T x_j$

$\Downarrow$ $\qquad$ $\Uparrow$

$$-\varepsilon x_i^T x_j \le \varphi(x_i)^T\varphi(x_j) - (x_i^T x_j) \le \varepsilon x_i^T x_j$$

thus Bound both sides by $C \cdot \varepsilon$ where C is some const which is equal to $C = x_i^T x_j$ (or $\|x_i\|\|x_j\|$ w/ cauchy schwartz)

3.e. $u \in \mathbb{R}^d$

$J \in \mathbb{R}^{k \times d}$ random Matrix each entry iid $N(0,1)$

$\psi_J : \mathbb{R}^d \to \mathbb{R}^k$ s.t $\psi_J(x) = \frac{1}{\sqrt{k}} Jx$

Show $\frac{\|\psi_J(u)\|^2}{\|u\|^2} = \frac{1}{k} \sum_{i=1}^{k} z_i^2$ z_i are iid $N(0,1)$ r.v.

$$\frac{\|\psi_J(u)\|^2}{\|u\|^2} = \frac{\|(k)^{(-1/2)} J\vec{u}\|^2}{\|\vec{u}\|^2} = \frac{(k^{-1/2} Ju)^T (k^{-1/2} Ju)}{(u^T u)}$$

$$k^{-1/2} k^{-1/2} \frac{(u^T J^T J u)}{u^T u} = \frac{1}{k} \frac{u^T J^T J u}{u^T u}$$

$u^T J^T = \underset{1\times d}{[u_1 \dots u_d]} \begin{bmatrix} N(0,1) & & \\ & N(0,1) & \\ & & \ddots \end{bmatrix}_{d \times k}$, $u^T u$ = scalar $u_1^2 + \dots + u_d^2$

$= \underset{1\times k}{\left[\left(u_1 z_1 \right) + u_2 N(0,1) + \dots u_d N(0,1) \right]} \left(\quad \right) \dots \left(\quad \right)$

z: var: 1

$u_i z_i$ var: u_i^2

- Mean: 0 Variance: $u_1^2 + u_2^2 + \dots + u_d^2$ for $u^T J$ $1 \times d$

$Ju = \begin{bmatrix} N(0,1) & \cdots \\ \vdots & \ddots \end{bmatrix}_{k \times d} \begin{bmatrix} u_1 \\ \vdots \\ u_d \end{bmatrix}_{d \times 1}$ • Mean = 0 Variance: $u_1^2 + \dots + u_d^2$ for Ju $k \times 1$

$= \frac{1}{k} \cdot \frac{1}{u_1^2 + \dots + u_d^2} u^T J^T J u = \frac{1}{k(u_1^2 + \dots + u_d^2)} \underbrace{\underset{1\times k}{\left[u_1^2 z_1 \cdots u_d^2 z_k \right]} \underset{k\times 1}{\begin{bmatrix} u_1^2 z_1 \\ \vdots \\ u_d^2 z_k \end{bmatrix}}}_{\begin{bmatrix} \text{Mean } 0 \\ \text{variance } (u_1^2 + \dots + u_d^2)^2 \end{bmatrix}}$

$= \frac{1}{k} \sum_{i=1}^{k} z_i^2$

3.e (cont)

Varies as z

$$\frac{1}{K}\left[\frac{1}{(v_1^2+\dots+v_k^2)^2}(v_{k+1}^2+\dots+v_k^2)\sum z_i\right]$$

Varies: 2

$$\frac{1}{K}\frac{1}{(v_1^2+\dots+v_k^2)^2}(v_1^2+\dots+v_k^2)^2\sum_{i=1}^{k} z_i$$

5. Own Question

Q: • What other Machine learning models could we have used for part 3 (computing adjustments for inserting an image into another image)?
- Would other models be more performant?
- How can we account for glare?

A: • we could use Ridgeregression least squares. This could help account for glare or Directed lighting on our object being inserted & would not suffer from the same downsides as total least squares. RR is good for colinearity

3 H Description

The PCA and random projections have very similar results.Dont observe a difference. There is a strong positive linear trend in all three data sets.

3 I Comment

As we increase the number of principal components the accuracy of our two methods of projections converge.

3 J Observations & Explanation

The size of the first singular values, and thus the first principal component, dominates in each of the data sets. The other singular values are smaller and similar in magnitude.

For regular the greatest reduction in error increases after the selectin of the first principal component and does not improve much thereafter. Knowing this it makes sense that the random takes longer to reduce errors.
IS