Question 1. Getting Started

Read through this page carefully. You may typeset your homework in latex or submit neatly handwritten/scanned solutions. Please start each question on a new page. Deliverables:

1. Submit a PDF of your writeup, **with an appendix for your code**, to assignment on Gradescope, "¡¡title¿¿ Write-Up". If there are graphs, include those graphs in the correct sections. Do not simply reference your appendix.
2. If there is code, submit all code needed to reproduce your results, "¡¡title¿¿ Code".
3. If there is a test set, submit your test set evaluation results, "¡¡title¿¿ Test Set".

After you've submitted your homework, watch out for the self-grade form.

(a) 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 with Weiran Liu. First, I double check if the latex version is the same as the PDF version because they were different last time. Second, I open a bottle of beer and turn on my music. Third, enjoy the homework.

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadvertently cheats.

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.

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. —Huanjie Sheng

Question 2. Total Least Squares

In most of the models we have looked at so far, we've accounted for noise in the observed y measurement and adjusted accordingly. However, in the real world it could easily be that our feature matrix X of data is also corrupted or noisy. Total least squares is a way to account for this. Whereas previously we were minimizing the y distance from the data point to our predicted line because we had assumed the features were definitively accurate, now we are minimizing the entire distance from the data point to our predicted line. In this problem we will explore the mathematical intuition for the TLS formula. We will then apply the formula to adjusting the lighting of an image which contains noise in its feature matrix due to inaccurate assumptions we make about the image, such as the image being a perfect sphere.

Let $\mathbb{X}$ and $\vec{y}$ be the true measurements. Recall that in the least squares problem, we want to solve for $\vec{w}$ in $\min_{\vec{w}} ||\mathbb{X}\vec{w} - \vec{y}||$. We measure the error as the difference between $\mathbb{X}\vec{w}$ and $\vec{y}$, which can be viewed as adding an error term $\vec{\epsilon_y}$ such that the equation $\mathbb{X}\vec{w} = \vec{y} + \vec{\epsilon_y}$ has a solution:

$$\min_{\vec{\epsilon_y}, \vec{w}} ||\vec{\epsilon_y}||_2, \text{ subject to } \mathbb{X}\vec{w} = \vec{y} + \vec{\epsilon_y} \tag{1}$$

Although this optimization formulation allows for errors in the measurements of $\vec{y}$, it does not allow for errors in the feature matrix $\mathbb{X}$ that is measured from the data. In this problem, we will explore a method called *total least squares* that allows for both error in the matrix $\mathbb{X}$ and the vector $\vec{y}$, represented by $\epsilon_{\mathbb{X}}$ and $\vec{\epsilon_y}$, respectively. For convenience, we absorb the negative sign into $\vec{\epsilon_y}$ and $\vec{\epsilon_X}$ and define true measurements $\vec{y}$ and $\vec{X}$ like so:

$$\vec{y}^{true} = \vec{y} + \vec{\epsilon_y} \tag{2}$$
$$\mathbb{X}^{true} = \mathbb{X} + \epsilon_{\mathbb{X}} \tag{3}$$

Specifically, the total least squares problem is to find the solution for $\vec{w}$ in the following minimization problem:

$$\min_{\vec{\epsilon_y}, \vec{\epsilon_X}, \vec{w}} ||[\epsilon_{\mathbb{X}}, \vec{\epsilon_y}]||_F^2, \text{ subject to } (\mathbb{X} + \epsilon_{\mathbb{X}})\vec{w} = \vec{y} + \vec{\epsilon_y} \tag{4}$$

where the matrix $[\epsilon_{\mathbb{X}}, \vec{\epsilon_y}]$ is the concatenation of the columns of $\epsilon_{\mathbb{X}}$ with the column vector $\vec{y}$. Notice that the minimization is over $\vec{w}$ because it's a free parameter, and it does not necessarily have to be in the objective function. Intuitively, this equation is finding the smallest perturbation to the matrix of data points $\mathbb{X}$ and the outputs $\vec{y}$ such that the linear model can be solved exactly. The constraint in the minimization problem can be rewritten as

$$[\mathbb{X} + \epsilon_{\mathbb{X}}, \vec{y} + \vec{\epsilon_y}] \begin{bmatrix} \vec{w} \\ -1 \end{bmatrix} = \vec{0} \tag{5}$$

(a) Let the matrix $\mathbb{X} \in \mathbb{R}^{n \times d}$ and $\vec{y} \in \mathbb{R}^n$ and note that $\epsilon_X \in \mathbb{R}^{n \times d}$ and $\vec{\epsilon_y} \in \mathbb{R}^n$. Assuming that $n > d$ and $\text{rank}(\mathbb{X} + \epsilon_{\mathbb{X}}) = d$, **explain why rank**$([\mathbb{X} + \epsilon_{\mathbb{X}}, \vec{y} + \vec{\epsilon_y}]) = d$**.**

Because $(\mathbb{X}+\epsilon_{\mathbb{X}})\vec{w} = \vec{y}+\vec{\epsilon_y}$, we know that $\vec{y}+\vec{\epsilon_y}$ is in the column space of $\mathbb{X}+\epsilon_{\mathbb{X}}$. We also assume that $n > d$ and $\text{rank}(\mathbb{X} + \epsilon_{\mathbb{X}}) = d$. Therefore the concatenated matrix is rank-deficient and we have $\text{rank}([\mathbb{X} + \epsilon_{\mathbb{X}}, \vec{y} + \vec{\epsilon_y}]) = d$.

(b) For the solution $\vec{w}$ to be unique, the matrix $[\mathbb{X}+\epsilon_{\mathbb{X}}, \vec{y}+\vec{\epsilon_y}]$ must have exactly d linearly independent columns. Since this matrix has d+1 columns in total, it must be rank-deficient by 1. Recall that the Eckart-Young-Mirsky Theorem tells us that the closest lower-rank matrix in the Frobenius norm is obtained by discarding the smallest singular values. Therefore, the matrix $[\mathbb{X}+\epsilon_{\mathbb{X}}, \vec{y}+\vec{\epsilon_y}]$ that minimizes

$$||[\epsilon_{\mathbb{X}}, \vec{\epsilon_y}]||_F^2 = ||[\mathbb{X}^{true}, \vec{y}^{true}] - [\mathbb{X}, \vec{y}]||_F^2$$

is given by

$$[\mathbb{X} + \epsilon_{\mathbb{X}}, \vec{y} + \vec{\epsilon_y}] = U \begin{bmatrix} \Sigma_d & \\ & 0 \end{bmatrix} V^\top$$

where $[\mathbb{X}, \vec{y}] = U\Sigma V^\top$ and Σ_d is the diagonal matrix of the d largest singular values of $[\mathbb{X}, \vec{y}]$.

Suppose we express the SVD of $[\mathbb{X}, \vec{y}]$ in terms of submatrices and vectors:

$$[\mathbb{X}, \vec{y}] = \begin{bmatrix} \mathbb{U}_{xx} & \vec{u}_{xy} \\ \vec{u}_{xy}^\top & u_{yy} \end{bmatrix} \begin{bmatrix} \Sigma_d & \\ & \sigma_{d+1} \end{bmatrix} \begin{bmatrix} \mathbb{V}_{xx} & \vec{v}_{xy} \\ \vec{v}_{xy}^\top & v_{yy} \end{bmatrix}^\top$$

where the vectors $\vec{u}_{xy} \in \mathbb{R}^d$ and $\vec{v}_{xy} \in \mathbb{R}^d$ are the first d elements of the (d+1)-th column of $\mathbb{U}$ and $\mathbb{V}$ respectively, u_{yy} and v_{yy} are the (d+1)-th element of the d+1 column of $\mathbb{U}$ and $\mathbb{V}$ respectively, $\mathbb{U}_{xx} \in \mathbb{R}^{d\times d}$ and $\mathbb{V}_{xx} \in \mathbb{R}^{d\times d}$ are the $d \times d$ top left submatrices of $\mathbb{U}$ and $\mathbb{V}$ respectively, and σ_{d+1} is the (d+1)-th eigenvalue of $[\mathbb{X}, \vec{y}]$.

Using this information show that

$$[\epsilon_{\mathbb{X}}, \vec{\epsilon_y}] = -\begin{bmatrix} \vec{u}_{xy} \\ u_{yy} \end{bmatrix} \sigma_{d+1} \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}^\top$$

$$\begin{aligned}
& [\epsilon_{\mathbb{X}}, \vec{\epsilon_y}] \\
= \; & [\mathbb{X} + \epsilon_{\mathbb{X}}, \vec{y} + \vec{\epsilon_y}] - [\mathbb{X}, \vec{y}] \\
= \; & \begin{bmatrix} \mathbb{U}_{xx} & \vec{u}_{xy} \\ \vec{u}_{yx}^\top & u_{yy} \end{bmatrix} \begin{bmatrix} \Sigma_d & \\ & 0 \end{bmatrix} \begin{bmatrix} \mathbb{V}_{xx} & \vec{v}_{xy} \\ \vec{v}_{yx}^\top & v_{yy} \end{bmatrix}^\top - \begin{bmatrix} \mathbb{U}_{xx} & \vec{u}_{xy} \\ \vec{u}_{yx}^\top & u_{yy} \end{bmatrix} \begin{bmatrix} \Sigma_d & \\ & \sigma_{d+1} \end{bmatrix} \begin{bmatrix} \mathbb{V}_{xx} & \vec{v}_{xy} \\ \vec{v}_{yx}^\top & v_{yy} \end{bmatrix}^\top \\
= \; & \begin{bmatrix} \mathbb{U}_{xx} & \vec{u}_{xy} \\ \vec{u}_{yx}^\top & u_{yy} \end{bmatrix} \left(\begin{bmatrix} \Sigma_d & \\ & 0 \end{bmatrix} - \begin{bmatrix} \Sigma_d & \\ & \sigma_{d+1} \end{bmatrix} \right) \begin{bmatrix} \mathbb{V}_{xx} & \vec{v}_{xy} \\ \vec{v}_{yx}^\top & v_{yy} \end{bmatrix}^\top \\
= \; & -\begin{bmatrix} \mathbb{U}_{xx} & \vec{u}_{xy} \\ \vec{u}_{yx}^\top & u_{yy} \end{bmatrix} \begin{bmatrix} 0 & \\ & \sigma_{d+1} \end{bmatrix} \begin{bmatrix} \mathbb{V}_{xx} & \vec{v}_{xy} \\ \vec{v}_{yx}^\top & v_{yy} \end{bmatrix}^\top \\
= \; & -\begin{bmatrix} \mathbb{U}_{xx} & \vec{u}_{xy} \\ \vec{u}_{yx}^\top & u_{yy} \end{bmatrix} \begin{bmatrix} 0 & 0 \\ \sigma_{d+1}\vec{v}_{yx}^\top & \sigma_{d+1}v_{yy} \end{bmatrix} \\
= \; & -\begin{bmatrix} \vec{u}_{xy}\sigma_{d+1}\vec{v}_{yx}^\top & \vec{u}_{xy}\sigma_{d+1}v_{yy} \\ \vec{u}_{yy}\sigma_{d+1}\vec{v}_{yx}^\top & \vec{u}_{xy}\sigma_{d+1}v_{yy} \end{bmatrix} \\
= \; & -\begin{bmatrix} \vec{u}_{xy} \\ u_{yy} \end{bmatrix} \sigma_{d+1} \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}^\top
\end{aligned}$$

I think those $\vec{u}_{xy}^\top$ and $\vec{v}_{xy}^\top$ should be $\vec{u}_{yx}^\top$ and $\vec{u}_{yx}^\top$ respectively. This is later fixed on Piazza.

(c) **Using the result from the previous part and that fact that v_{yy} is not 0 (see notes on Total Least Squares), find a nonzero solution to $[\mathbb{X} + \epsilon_{\mathbb{X}}, \vec{y} + \vec{\epsilon_y}] \begin{bmatrix} \vec{w} \\ -1 \end{bmatrix} = \vec{0}$ and thus solve for $\vec{w}$ in Equation (5).**

HINT: Looking at the last column of the product $[\mathbb{X}, \vec{y}]\mathbb{V}$ may or may not be useful for this problem, depending on how you solve it.

To answer the question, we need to find a vector in the null space of $[\mathbb{X} + \epsilon_{\mathbb{X}}, \vec{y} + \vec{\epsilon_y}]$. We take a look at the last column of the product $[\mathbb{X} + \epsilon_{\mathbb{X}}, \vec{y} + \vec{\epsilon_y}]\mathbb{V}$.

$$
\begin{aligned}
& [\mathbb{X} + \epsilon_{\mathbb{X}}, \vec{y} + \vec{\epsilon_y}] \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} \\
= & \begin{bmatrix} \mathbb{U}_{xx} & \vec{u}_{xy} \\ \vec{u}_{yx}^\top & u_{yy} \end{bmatrix} \begin{bmatrix} \Sigma_d & \\ & 0 \end{bmatrix} \begin{bmatrix} \mathbb{V}_{xx} & \vec{v}_{xy} \\ \vec{v}_{yx}^\top & v_{yy} \end{bmatrix}^\top \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix} \\
= & \begin{bmatrix} \mathbb{U}_{xx} & \vec{u}_{xy} \\ \vec{u}_{yx}^\top & u_{yy} \end{bmatrix} \begin{bmatrix} \Sigma_d & \\ & 0 \end{bmatrix} \begin{bmatrix} \mathbb{V}_{xx}^\top \vec{v}_{xy} + \vec{v}_{xy} v_{yy} \\ \vec{v}_{yx}^\top \vec{v}_{xy} + v_{yy} v_{yy} \end{bmatrix} \\
= & \begin{bmatrix} \mathbb{U}_{xx} & \vec{u}_{xy} \\ \vec{u}_{yx}^\top & u_{yy} \end{bmatrix} \begin{bmatrix} \Sigma_d (\mathbb{V}_{xx}^\top \vec{v}_{xy} + \vec{v}_{xy} v_{yy}) \\ 0 \end{bmatrix} \\
= & \begin{bmatrix} \mathbb{U}_{xx} \Sigma_d (\mathbb{V}_{xx}^\top \vec{v}_{xy} + \vec{v}_{xy} v_{yy}) \\ 0 \end{bmatrix} \\
= & \, 0
\end{aligned}
$$

The last step is true because V is orthonormal so the last column of V is orthogonal to the rest of V. Therefore, we have:

$$\begin{bmatrix} \vec{w} \\ -1 \end{bmatrix} = -\frac{1}{v_{yy}} \begin{bmatrix} \vec{v}_{xy} \\ v_{yy} \end{bmatrix}$$

(d) From the previous part, you can see that $\begin{bmatrix} \vec{w} \\ -1 \end{bmatrix}$ is a right-singular vector of $[\mathbb{X}, \vec{y}]$. **Show that**

$$(\vec{X}^\top \vec{X} - \sigma_{d+1}^2 I)\vec{w} = \vec{X}^\top \vec{y} \tag{14}$$

$$\begin{bmatrix} \mathbb{X} & \vec{y} \end{bmatrix} \begin{bmatrix} \vec{w} \\ -1 \end{bmatrix} = \sigma_{d+1}\vec{u}$$

$$\begin{bmatrix} \mathbb{X}^\top \\ \vec{y}^\top \end{bmatrix} \begin{bmatrix} \mathbb{X} & \vec{y} \end{bmatrix} \begin{bmatrix} \vec{w} \\ -1 \end{bmatrix} = \sigma_{d+1} \begin{bmatrix} \mathbb{X}^\top \\ \vec{y}^\top \end{bmatrix} \vec{u}$$

$$\begin{bmatrix} \mathbb{X}^\top\mathbb{X} & \mathbb{X}^\top\vec{y} \\ \vec{y}^\top\mathbb{X} & \vec{y}^\top\vec{y} \end{bmatrix} \begin{bmatrix} \vec{w} \\ -1 \end{bmatrix} = \sigma_{d+1} \begin{bmatrix} \mathbb{X}^\top \\ \vec{y}^\top \end{bmatrix} \vec{u}$$

$$\begin{bmatrix} \mathbb{X}^\top\mathbb{X} & \mathbb{X}^\top\vec{y} \\ \vec{y}^\top\mathbb{X} & \vec{y}^\top\vec{y} \end{bmatrix} \begin{bmatrix} \vec{w} \\ -1 \end{bmatrix} = \sigma_{d+1}^2 \begin{bmatrix} \vec{w} \\ -1 \end{bmatrix}$$

$$\mathbb{X}^\top\mathbb{X}\vec{w} - \mathbb{X}^\top\vec{y} = \sigma_{d+1}^2\mathbb{I}\vec{w}$$

$$(\mathbb{X}^\top\mathbb{X} - \sigma_{d+1}^2\mathbb{I})\vec{w} = \mathbb{X}^\top\vec{y}$$

Figure 1: Tennis ball pasted on top of image of St. Peter's Basillica without lighting adjustment (left) and with lighting adjustment (right)

Figure 2: Image of a spherical mirror inside of St. Peter's Basillica

(e) In this problem, we will use total least squares to approximately learn the lighting in a photograph, which we can then use to paste new objects into the image while still maintaining the realism of the image. You will be estimating the lighting coefficients for the interior of St. Peter's Basillica, and you will then use these coefficients to change the lighting of an image of a tennis ball so that it can be pasted into the image. In Figure 1, we show the result of pasting the tennis ball in the image without adjusting the lighting on the ball. The ball looks too bright for the scene and does not look like it would fit in with other objects in the image.

To convincingly add a tennis ball to an image, we need to need to apply the appropriate lighting from the environment onto the added ball. To start, we will represent environment lighting as a spherical function $\vec{f}(\vec{n})$ where $\vec{n}$ is a 3 dimensional unit vector ($||\vec{n}||_2 = 1$), and $\vec{f}$ outputs a 3 dimensional color vector, one component for red, green, and blue light intensities. Because $\vec{f}(\vec{n})$ is a spherical function, the input $\vec{n}$ must correspond to a point on a sphere. The function $\vec{f}(\vec{n})$ represents the total incoming light from the direction $\vec{n}$ in the scene. The lighting function of a spherical object $\vec{f}(\vec{n})$ can be approximated by the first 9 spherical harmonic basis functions.

The first 9 unnormalized sperical harmonic basis functions are given by:

$$
\begin{aligned}
L_1 &= 1 \\
L_2 &= y \\
L_3 &= x \\
L_4 &= z \\
L_5 &= xy \\
L_6 &= yz \\
L_7 &= 3z^2 - 1 \\
L_8 &= xz \\
L_9 &= x^2 - y^2
\end{aligned}
$$

where $\vec{n} = [x, y, z]^\top$. The lighting function can then be approximated as

$$
\vec{f}(\vec{n}) \approx \sum_{i=1}^{9} \vec{\gamma}_i L_i(\vec{n})
$$

$$
\begin{bmatrix} — & \vec{f}(\vec{n_1}) & — \\ — & \vec{f}(\vec{n_2}) & — \\ & \vdots & \\ — & \vec{f}(\vec{n_n}) & — \end{bmatrix}_{n\times 3}
=
\begin{bmatrix} L_1(\vec{n_1}) & L_2(\vec{n_1}) & ... & L_9(\vec{n_1}) \\ L_1(\vec{n_2}) & L_2(\vec{n_2}) & ... & L_9(\vec{n_2}) \\ & \vdots & & \\ L_1(\vec{n_n}) & L_2(\vec{n_n}) & ... & L_9(\vec{n_n}) \end{bmatrix}_{n\times 9}
\begin{bmatrix} — & \vec{\gamma}_1 & — \\ — & \vec{\gamma}_2 & — \\ & \vdots & \\ — & \vec{\gamma}_9 & — \end{bmatrix}_{9\times 3}
$$

where $L_i(\vec{n})$ is the ith basis function from the list above.

The function of incoming light $\vec{f}(\vec{n})$ can be measured by photographing a spherical mirror placed in the scene of interest. In this case, we provide you with an image of the sphere as seen in Figure 2. In the

code provided, there is a function extractNormals(img) that will extract the training pairs $(\vec{n}_i, \vec{f}(\vec{n}_i))$ from the image. An example using this function is in the code.

Use the spherical harmonic basis functions to create a 9 dimensional feature vector for each sample. Use this to formulate an ordinary least squares problem and solve for the unknown coefficients $\vec{\gamma}_i$. Report the estimated values for $\vec{\gamma}_i$ and include a visualization of the approximation using the provided code. The code provided will load the images, extracts the training data, relights the tennis ball with incorrect coefficients, and saves the results. Your task is to compute the basis functions and solve the least squares problems to provide the code with the correct coefficients. To run the starter code, you will need to use Python with `numpy` and `scipy`. Because the resulting data set is large, we reduce it in the code by taking every 50^{th} entry in the data. This is done for you in the starter code, but you can try using the entire data set or reduce it by a different amount.

Coefficients:

$$\begin{bmatrix} 202.31845431 & 162.41956802 & 149.07075034 \\ -27.66555164 & -17.88905339 & -12.92356688 \\ -5.15203925 & -4.51375871 & -4.24262639 \\ -1.08629293 & 0.42947012 & 1.15475569 \\ -3.14053107 & -3.70269907 & -3.74382934 \\ 23.67671768 & 23.15698002 & 21.94638397 \\ -3.82167171 & 0.57606634 & 1.81637483 \\ 4.7346737 & 1.4677692 & -1.12253649 \\ -9.72739616 & -5.75691108 & -4.8395598 \end{bmatrix}$$

```
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):
```

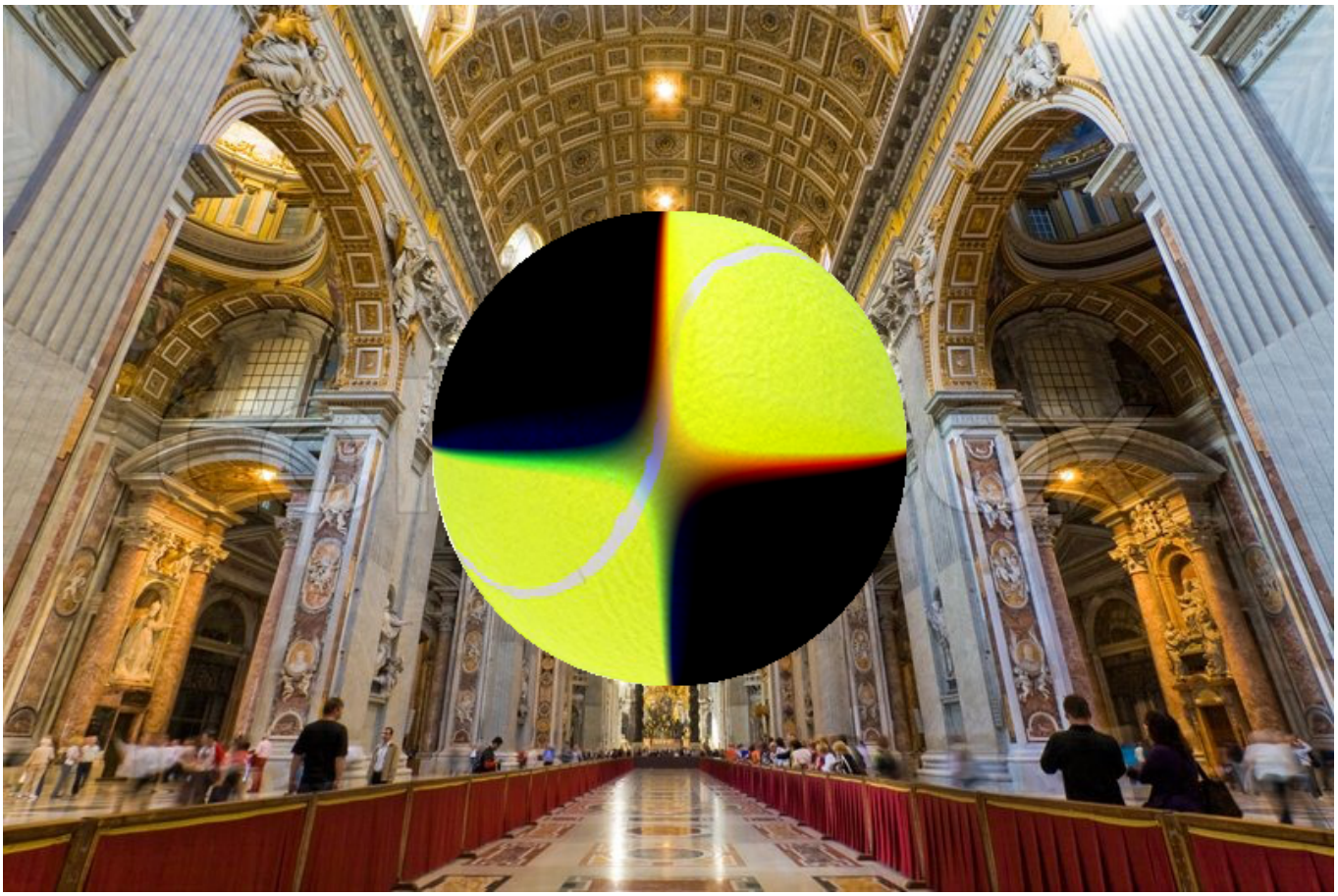
```
# Returns the first 9 spherical harmonic basis functions

################################################
# TODO: Compute the first 9 basis functions
################################################
B = np.ones((len(ns), 9)) # This line is here just to fill space
# L1 = 1
# B[:, 1] = 1
# L2 = y
B[:, 1] = ns[:, 1]
# L3 = x
B[:, 2] = ns[:, 0]
# L4 = z
B[:, 3] = ns[:, 2]
# L5 = xy
B[:, 4] = ns[:, 0] * ns[:, 1]
# L6 = yz
B[:, 5] = ns[:, 1] * ns[:, 2]
# L7 = 3z^2-1
B[:, 6] = 3 * (ns[:, 2] ** 2) - 1
# L6 = xz
B[:, 7] = ns[:, 0] * ns[:, 2]
# L7 = x^2-y^2
B[:, 8] = (ns[:, 0] ** 2) - (ns[:, 1] ** 2)
return B

def ordinary_least_squares(A, b):
# Make sure data are centralized
return np.linalg.solve(A.T.dot(A), A.T.dot(b))


def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
return '\n'.join(rv)


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
################################################
# coeff = np.zeros((9, 3))
# coeff[0, :] = 255
coeff = ordinary_least_squares(Bp, vsp)
```

```
img = relightSphere(tennis, coeff)

output = compositeImages(img, target)

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

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

imsave('Figure_2e.png', output)
```

(f) When we extract from the data the direction $\vec{n}$ to compute $(\vec{n}_i, \vec{f}(\vec{n}_i))$, we make some approximations about how the light is captured on the image. We also assume that the spherical mirror is a perfect sphere, but in reality, there will always be small imperfections. Thus, our measurement for $\vec{n}$ contains some error, which makes this an ideal problem to apply total least squares. **Solve this problem with total least squares by allowing perturbations in the matrix of basis functions. Report the estimated values for $\vec{\gamma}_i$ and include a visualization of the approximation.** The output image will be visibly wrong, and we'll explore how to fix this problem in the next part. Your implementation may only use the SVD and the matrix inverse functions from the linear algebra library in numpy.

Coefficients:

$$\begin{bmatrix} 2.13318421e+02 & 1.70780299e+02 & 1.57126297e+02 \\ -3.23046362e+01 & -2.02975310e+01 & -1.45516114e+01 \\ -4.31689131e+00 & -3.80778081e+00 & -4.83616306e+00 \\ -4.89811386e+00 & -3.37684058e+00 & -1.14207091e+00 \\ -7.05901066e+03 & -7.39934207e+03 & -4.26448732e+03 \\ -3.05378224e+02 & -1.56329401e+02 & 3.50285345e+02 \\ -9.76079364e+00 & -5.33182216e+00 & -1.55699782e+00 \\ 7.30792588e+02 & 3.52130316e+02 & -6.11683200e+02 \\ -9.08887079e+00 & -3.84309477e+00 & -4.16456437e+00 \end{bmatrix}$$



```
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

    ################################################
    # TODO: Compute the first 9 basis functions
    ################################################
    B = np.ones((len(ns), 9)) # This line is here just to fill space
    # L1 = 1
    # B[:, 1] = 1
    # L2 = y
    B[:, 1] = ns[:, 1]
    # L3 = x
    B[:, 2] = ns[:, 0]
    # L4 = z
    B[:, 3] = ns[:, 2]
    # L5 = xy
    B[:, 4] = ns[:, 0] * ns[:, 1]
    # L6 = yz
    B[:, 5] = ns[:, 1] * ns[:, 2]
    # L7 = 3z^2-1
    B[:, 6] = 3 * (ns[:, 2] ** 2) - 1
    # L6 = xz
    B[:, 7] = ns[:, 0] * ns[:, 2]
    # L7 = x^2-y^2
    B[:, 8] = (ns[:, 0] ** 2) - (ns[:, 1] ** 2)
    return B


    def total_least_squares(A, b):
    # Make sure data are centralized
    nfeature = int(A.shape[1])
    nout = int(b.shape[1])
    xy = np.hstack([A, b])
    _, _, vh = np.linalg.svd(xy)
    w_1 = vh[-nout:, :].T
    null_w = w_1[-nout:, :]
    return - w_1[:nfeature, :] .dot(np.linalg.inv(null_w))


    def bmatrix(a):
    """Returns a LaTeX bmatrix
    Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
    :a: numpy array
    :returns: LaTeX bmatrix as a string
    """
    if len(a.shape) > 2:
    raise ValueError('bmatrix can at most display two dimensions')
    lines = str(a).replace('[', '').replace(']', '').splitlines()
    rv = [r'\begin{bmatrix}']
    rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
    rv += [r'\end{bmatrix}']
    return '\n'.join(rv)


    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
##################################################
# coeff = np.zeros((9, 3))
# coeff[0, :] = 255
coeff = total_least_squares(Bp, vsp)

img = relightSphere(tennis, coeff)

output = compositeImages(img, target)

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

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

imsave('Figure_2f.png', output)
```

(g) In the previous part, you should have noticed that the visualization is drastically different than the one generated using least squares. Recall that in total least squares we are minimizing $|||[\epsilon_{\mathbb{X}}, \vec{\epsilon_y}]|||_F^2$. Intuitively, to minimize the Frobenius norm of components of both the inputs and outputs, the inputs and outputs should be on the same scale. However, this is not the case here. Color values in an image will typically be in $[0, 255]$, but the original image had a much larger range. We compressed the range to a smaller scale using tone mapping, but the effect of the compression is that relatively bright areas of the image become less bright. As a compromise, we scaled the image colors down to a maximum color value of 384 instead of 255. Thus, the inputs here are all unit vectors, and the outputs are 3 dimensional vectors where each value is in $[0, 384]$. **Propose a value by which to scale the outputs $\vec{f}(\vec{n}_i)$ such that the values of the inputs and outputs are roughly on the same scale. Solve this scaled total least squares problem, report the computed spherical harmonic coefficients and provide a rendering of the relit sphere.**

We rescale output to $[0, 1]$ by dividing 384. However, we also need to multiply 384 when we calculate the coefficients to make sure our predictions fall into the range of $[0, 384]$. (One thing should be noted that the inputs are not unit vectors but in the range of $[-1, 2]$)

Coefficients:

$$\begin{bmatrix} 209.38212459 & 169.03666402 & 155.36677288 \\ -30.26805402 & -20.30443706 & -15.20472049 \\ -5.753416 & -5.07881542 & -4.78144904 \\ -1.05630713 & 0.46377951 & 1.19195587 \\ -7.90569522 & -8.20316831 & -8.05137623 \\ 54.96251667 & 52.62398401 & 50.09265545 \\ -3.8491927 & 0.55663535 & 1.80236903 \\ 7.32655583 & 3.83064183 & 1.07500107 \\ -10.90665749 & -6.8522162 & -5.87526417 \end{bmatrix}$$

```
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

##################################################
# TODO: Compute the first 9 basis functions
##################################################
B = np.ones((len(ns), 9)) # This line is here just to fill space
# L1 = 1
# B[:, 1] = 1
# L2 = y
B[:, 1] = ns[:, 1]
# L3 = x
B[:, 2] = ns[:, 0]
# L4 = z
B[:, 3] = ns[:, 2]
# L5 = xy
B[:, 4] = ns[:, 0] * ns[:, 1]
# L6 = yz
B[:, 5] = ns[:, 1] * ns[:, 2]
# L7 = 3z^2-1
B[:, 6] = 3 * (ns[:, 2] ** 2) - 1
# L6 = xz
B[:, 7] = ns[:, 0] * ns[:, 2]
# L7 = x^2-y^2
B[:, 8] = (ns[:, 0] ** 2) - (ns[:, 1] ** 2)
return B


def total_least_squares(A, b):
# Make sure data are centralized
```

```
    nfeature = int(A.shape[1])
    nout = int(b.shape[1])
    xy = np.hstack([A, b])
    _, _, vh = np.linalg.svd(xy)
    w_1 = vh[-nout:, :].T
    null_w = w_1[-nout:, :]
    return - w_1[:nfeature, :] .dot(np.linalg.inv(null_w))


    def bmatrix(a):
    """Returns a LaTeX bmatrix
    Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
    :a: numpy array
    :returns: LaTeX bmatrix as a string
    """
    if len(a.shape) > 2:
    raise ValueError('bmatrix can at most display two dimensions')
    lines = str(a).replace('[', '').replace(']', '').splitlines()
    rv = [r'\begin{bmatrix}']
    rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
    rv += [r'\end{bmatrix}']
    return '\n'.join(rv)


    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
    ##################################################
    # coeff = np.zeros((9, 3))
    # coeff[0, :] = 255
    coeff = total_least_squares(Bp, vsp/384) * 384

    img = relightSphere(tennis, coeff)

    output = compositeImages(img, target)

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

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

    imsave('Figure_2g.png', output)
```

Question 3. PCA and Random Projections

In this question, we revisit the task of dimensionality reduction. Dimensionality reduction is useful for several purposes, including but not restricted to, visualization, storage, faster computation etc. While reducing dimension is useful, it is not undesirable to demand that such reductions preserve some properties of the original data. Often, certain geometric properties like distance and inner products are important to perform certain machine learning tasks. And as a result, we may want to perform dimensionality reduction but ensuring that we approximately maintain the pairwise distances and inner products.

While you have already seen many properties of PCA so far, in this question we investigate if random projections work are a good idea for dimensionality reduction. A few advantages of random projections over PCA can be listed as: (1) PCA is expensive when the underlying dimension is high and the number of principal components is also large (however note that there are several very fast algorithms dedicated to doing PCA), (2) PCA requires you to have access to the feature matrix for performing computations. The second requirement of PCA is a bottle neck when you want to take only a low dimensional measurement of a very high dimensional data, e.g., in FMRI and in compressed sensing. In such cases, one needs to design a projection scheme before seeing the data. We now turn to a concrete setting study a few properties of PCA and random projections.

Suppose you are given n points $\vec{x}_1, \ldots, \vec{x}_n$ in $\mathbb{R}^d$. Define the $n \times d$ matrix $\mathbb{X} = \begin{bmatrix} \vec{x}_1^\top \\ \vdots \\ \vec{x}_n^\top \end{bmatrix}$ where each row of the matrix represents one of the given points. In this problem, we will consider a few low-dimensional embedding $\psi : \mathbb{R}^d \mapsto \mathbb{R}^k$ that maps vectors from $\mathbb{R}^d$ to $\mathbb{R}^k$.

Let $\mathbb{X} = \mathbb{U}\mathbf{\Sigma}\mathbb{V}^\top$ denote the singular value decomposition of the matrix $\mathbb{X}$. Assume that $N \geq d$ and let $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_d$ denote the singular values of $\mathbb{X}$.

Let $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_d$ denote the columns of the matrix $\mathbb{V}$. We now consider the following k-dimensional PCA embedding: $\psi_{\text{PCA}}(\vec{x}) = (\vec{v}_1^\top \vec{x}, \ldots, \vec{v}_k^\top \vec{x})^\top$. Note that this embedding projects a d-dimensional vector on the linear span of the set $\{\vec{v}_1, \ldots, \vec{v}_k\}$ and that $\vec{v}_i^\top \vec{x}$ denotes the i-th coordinate of the projected vector in the new space.

We begin with a few matrix algebra relationships, and use it to investigate certain mathematical properties of PCA and random projections in the first few parts, and then see them in action on a synthetic dataset in the later parts.

Notation: The symbol $[n]$ stands for the set $\{1, \ldots, n\}$.

(a) **What is the ij-th entry of the matrices $\mathbb{X}\mathbb{X}^\top$ and $\mathbb{X}^\top\mathbb{X}$? Express the matrix $\mathbb{X}\mathbb{X}^\top$ in terms of $\mathbb{U}$ and $\mathbf{\Sigma}$, and, express the matrix $\mathbb{X}^\top\mathbb{X}$ in terms of $\mathbf{\Sigma}$ and $\mathbb{V}$.**

Let's denote $\mathbb{X}$ as:

$$\mathbb{X} = \begin{bmatrix} \vec{x}_1^\top \\ \vdots \\ \vec{x}_n^\top \end{bmatrix} = \begin{bmatrix} \vec{f}_1 & \ldots & \vec{f}_n \end{bmatrix}$$

$$(\mathbb{X}\mathbb{X}^\top)_{ij} = \vec{x}_i^\top \vec{x}_j$$

$$(\mathbb{X}^\top\mathbb{X})_{ij} = \vec{f}_i^\top \vec{f}_j$$

$$\mathbb{X}\mathbb{X}^\top = \mathbb{U}\Sigma^\top\mathbb{V}^\top\mathbb{V}\Sigma\mathbb{U}^\top = \mathbb{U}\Sigma^\top\Sigma\mathbb{U}^\top$$

$$\mathbb{X}^\top\mathbb{X} = \mathbb{V}\Sigma^\top\mathbb{U}^\top\mathbb{U}\Sigma\mathbb{V}^\top = \mathbb{V}\Sigma^\top\Sigma\mathbb{V}^\top$$

(b) Show that

$$\psi_{\mathbf{PCA}}(\vec{x}_i)^\top \psi_{\mathbf{PCA}}(\vec{x}_j) = \vec{x}_i^\top \mathbb{V}_k \mathbb{V}_k^\top \vec{x}_j \quad \textbf{where} \quad \mathbb{V}_k = \begin{bmatrix} \vec{v}_1 & \vec{v}_2 & \dots & \vec{v}_k \end{bmatrix}.$$

Also show that $\mathbb{V}_k \mathbb{V}_k^\top = \mathbb{V} \mathbb{I}^k \mathbb{V}^\top$, where the matrix $\mathbb{I}^k$ denotes a $d \times d$ diagonal matrix with first k diagonal entries as 1 and all other entries as zero.

$$\begin{aligned}
& \psi_{\mathbf{PCA}}(\vec{x}_i)^\top \psi_{\mathbf{PCA}}(\vec{x}_j) \\
= \; & \begin{bmatrix} \vec{x}_i^\top \vec{v}_1 & \vec{x}_i^\top \vec{v}_2 & \dots & \vec{x}_i^\top \vec{v}_k \end{bmatrix} \begin{bmatrix} \vec{v}_1^\top \vec{x}_j \\ \vec{v}_2^\top \vec{x}_j \\ \vdots \\ \vec{v}_k^\top \vec{x}_j \end{bmatrix} \\
= \; & \vec{x}_i^\top \mathbb{V}_k \mathbb{V}_k^\top \vec{x}_j
\end{aligned}$$

$$\begin{aligned}
& \mathbb{V} \mathbb{I}^k \mathbb{V}^\top \\
= \; & \begin{bmatrix} \vec{v}_1 & \vec{v}_2 & \dots & \vec{v}_d \end{bmatrix} \begin{bmatrix} \mathbb{I}^k & \\ & \vec{0} \end{bmatrix} \begin{bmatrix} \vec{v}_1^\top \\ \vec{v}_2^\top \\ \vdots \\ \vec{v}_d^\top \end{bmatrix} \\
= \; & \begin{bmatrix} \mathbb{V}_k & \mathbb{V}_{d-k} \end{bmatrix} \begin{bmatrix} \mathbb{I}^k & \\ & \vec{0} \end{bmatrix} \begin{bmatrix} \mathbb{V}_k^\top \\ \mathbb{V}_{d-k}^\top \end{bmatrix} \\
= \; & \begin{bmatrix} \mathbb{V}_k & \mathbb{V}_{d-k} \end{bmatrix} \begin{bmatrix} \mathbb{V}_k^\top \\ \vec{0} \end{bmatrix} \\
= \; & \mathbb{V}_k \mathbb{V}_k^\top
\end{aligned}$$

(c) Suppose that we know the first k singular values are the dominant singular values. In particular, we are given that

$$\frac{\sum_{i=1}^{k}\sigma_j^2}{\sum_{i=1}^{d}\sigma_i^2} \geq 1-\epsilon,$$

for some $\epsilon \in (0,1)$. Then show that the PCA projection to the first k-right singular vectors preserves the *inner products* on average:

$$\frac{1}{\sum_{i=1}^{n}\sum_{j=1}^{n}(\vec{x}_i^\top \vec{x}_j)^2}\sum_{i=1}^{n}\sum_{j=1}^{n}|(\vec{x}_i^\top \vec{x}_j) - (\psi_{\mathbf{PCA}}(\vec{x}_i)^\top \psi_{\mathbf{PCA}}(\vec{x}_j))|^2 \leq \epsilon. \tag{6}$$

Thus, we find that if there are dominant singular values, PCA projection can preserve the inner products on average.

Hint: Using previous two parts and the definition of Frobenius norm might be useful.

First, let's take a look at the definition of Frobenius norm:

$$\|X\|_F^2 = \sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}^2$$

Therefore, we have

$$\sum_{i=1}^{n}\sum_{j=1}^{n}(\vec{x}_i^\top \vec{x}_j)^2 = \|\mathbb{X}\mathbb{X}^\top\|_F^2 = tr(\mathbb{X}\mathbb{X}^\top\mathbb{X}\mathbb{X}^\top)$$

Because we know that the trace of a matrix is the sum of its eigenvalues, we have

$$tr(\mathbb{X}\mathbb{X}^\top\mathbb{X}\mathbb{X}^\top) = tr(\Sigma^\top\Sigma\Sigma^\top\Sigma)$$

For simplicity, we denote $\hat{\Sigma} = \Sigma^\top\Sigma$, $\hat{\Sigma}_k = \mathbb{I}^k\Sigma$

$$\begin{aligned}
\tfrac{1}{\sum_{i=1}^{n}\sum_{j=1}^{n}(\vec{x}_i^\top \vec{x}_j)^2}\textstyle\sum_{i=1}^{n}\sum_{j=1}^{n}|(\vec{x}_i^\top \vec{x}_j) - (\psi_{\mathbf{PCA}}(\vec{x}_i)^\top \psi_{\mathbf{PCA}}(\vec{x}_j))|^2 &\leq \epsilon \\
\tfrac{\|\mathbb{X}\mathbb{X}^\top - \mathbb{X}\mathbb{I}^k\mathbb{V}^\top\mathbb{X}^\top\|_F^2}{\|\mathbb{X}\mathbb{X}^\top\|_F^2} &\leq \epsilon \\
tr\left((\mathbb{X}\mathbb{X}^\top - \mathbb{X}\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{X}^\top)(\mathbb{X}\mathbb{X}^\top - \mathbb{X}\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{X}^\top)\right) &\leq \epsilon tr(\hat{\Sigma}^2) \\
tr(\mathbb{X}\mathbb{X}^\top\mathbb{X}\mathbb{X}^\top) - 2tr(\mathbb{X}\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{X}^\top\mathbb{X}\mathbb{X}^\top) + tr(\mathbb{X}\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{X}^\top\mathbb{X}\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{X}^\top) &\leq \epsilon tr(\hat{\Sigma}^2) \\
tr(\hat{\Sigma}^2) - 2tr(\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{X}^\top\mathbb{X}\mathbb{X}^\top\mathbb{X}) + tr(\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{X}^\top\mathbb{X}\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{X}^\top\mathbb{X}) &\leq \epsilon tr(\hat{\Sigma}^2) \\
tr(\hat{\Sigma}^2) - 2tr(\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{V}\hat{\Sigma}\mathbb{V}^\top\mathbb{V}\hat{\Sigma}\mathbb{V}^\top) + tr(\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{V}\hat{\Sigma}\mathbb{V}^\top\mathbb{V}\mathbb{I}^k\mathbb{V}^\top\mathbb{V}\hat{\Sigma}\mathbb{V}^\top) &\leq \epsilon tr(\hat{\Sigma}^2) \\
tr(\hat{\Sigma}^2) - 2tr(\mathbb{V}\mathbb{I}^k\hat{\Sigma}^2\mathbb{V}^\top) + tr(\mathbb{V}\mathbb{I}^k\Sigma\mathbb{I}^k\Sigma\mathbb{V}^\top) &\leq \epsilon tr(\hat{\Sigma}^2) \\
tr(\hat{\Sigma}^2) - 2tr(\mathbb{V}\hat{\Sigma}_k^2\mathbb{V}^\top) + tr(\mathbb{V}\hat{\Sigma}_k^2\mathbb{V}^\top) &\leq \epsilon tr(\hat{\Sigma}^2) \\
-tr(\mathbb{V}\hat{\Sigma}_k^2\mathbb{V}^\top) &\leq (\epsilon-1)tr(\hat{\Sigma}^2) \\
-tr(\hat{\Sigma}_k^2\mathbb{V}^\top\mathbb{V}) &\leq (\epsilon-1)tr(\hat{\Sigma}^2) \\
-tr(\hat{\Sigma}_k^2) &\leq (\epsilon-1)tr(\hat{\Sigma}^2) \\
tr(\hat{\Sigma}_k^2) &\geq (1-\epsilon)tr(\hat{\Sigma}^2) \\
\textstyle\sum_{i=1}^{k}\sigma_j^4 &\geq (1-\epsilon)\textstyle\sum_{i=1}^{d}\sigma_i^4 \\
\tfrac{\sum_{i=1}^{k}\sigma_j^4}{\sum_{i=1}^{d}\sigma_i^4} &\geq 1-\epsilon
\end{aligned}$$

The reverse of this derivation is also true. (As is pointed out on Piazza, it should be σ_i^4 instead of σ_i^2)

(d) Now consider a different embedding $\psi : \mathbb{R}^d \mapsto \mathbb{R}^k$ which preserves all pairwise distances and norms up-to a multiplicative factor, that is,

$$(1-\epsilon)\|\vec{x}_i\|^2 \leq \|\psi(\vec{x}_i)\|^2 \leq (1+\epsilon)\|\vec{x}_i\|^2 \qquad \textbf{for all } i \in [n], \quad \textbf{and} \tag{7}$$

$$(1-\epsilon)\|\vec{x}_i - \vec{x}_j\|^2 \leq \|\psi(\vec{x}_i) - \psi(\vec{x}_j)\|^2 \leq (1+\epsilon)\|\vec{x}_i - \vec{x}_j\|^2 \qquad \textbf{for all } i,j \in [n], \tag{8}$$

where $0 < \epsilon \ll 1$ is a small scalar. Further assume that $\|\vec{x}_i\| \leq 1$ for all $i \in [n]$. Show that the embedding ψ satisfying equations (8) and (7) preserves *each pairwise inner product*:

$$|\psi(\vec{x}_i)^\top \psi(\vec{x}_j) - (\vec{x}_i^\top \vec{x}_j)| \leq C\epsilon, \qquad \textbf{for all } i,j \in [n], \tag{9}$$

for some constant C. Thus, we find that if an embedding approximately preserves distances and norms upto a small multiplicative factor, and the points have bounded norms, then inner products are also approximately preserved upto an additive factor.

Hint: You may use the Cauchy-Schwarz inequality.

$$\begin{aligned}
& |\psi(\vec{x}_i)^\top \psi(\vec{x}_j) - (\vec{x}_i^\top \vec{x}_j)| \\
= \; & \tfrac{1}{2}|2\psi(\vec{x}_i)^\top \psi(\vec{x}_j) - 2(\vec{x}_i^\top \vec{x}_j)| \\
= \; & \tfrac{1}{2}|\psi(\vec{x}_i)^\top \psi(\vec{x}_i) + \psi(\vec{x}_j)^\top \psi(\vec{x}_j) - (\psi(\vec{x}_i) - \psi(\vec{x}_j))^\top (\psi(\vec{x}_i) - \psi(\vec{x}_j)) \\
& \quad -\vec{x}_i^\top \vec{x}_i - \vec{x}_j^\top \vec{x}_j + (\vec{x}_i - \vec{x}_j)^\top (\vec{x}_i - \vec{x}_j)| \\
= \; & \tfrac{1}{2}|\|\psi(\vec{x}_i)\|_2^2 + \|\psi(\vec{x}_j)\|_2^2 - \|\psi(\vec{x}_i) - \psi(\vec{x}_j)\|_2^2 - \|\vec{x}_i\|_2^2 - \|\vec{x}_j\|_2^2 + \|\vec{x}_i - \vec{x}_j\|_2^2| \\
\leq \; & \tfrac{1}{2}|(1+\epsilon)(\|\vec{x}_i\|_2^2 + \|\vec{x}_j\|_2^2) - (1-\epsilon)(\|\vec{x}_i - \vec{x}_j\|_2^2) - \|\vec{x}_i\|_2^2 - \|\vec{x}_j\|_2^2 + \|\vec{x}_i - \vec{x}_j\|_2^2| \\
= \; & \tfrac{1}{2}|(1+\epsilon)(\|\vec{x}_i\|_2^2 + \|\vec{x}_j\|_2^2) - (1-\epsilon)(\|\vec{x}_i - \vec{x}_j\|_2^2) - \|\vec{x}_i\|_2^2 - \|\vec{x}_j\|_2^2 + \|\vec{x}_i - \vec{x}_j\|_2^2| \\
= \; & \tfrac{1}{2}|\epsilon(\|\vec{x}_i\|_2^2 + \|\vec{x}_j\|_2^2 + \|\vec{x}_i - \vec{x}_j\|_2^2)| \\
\leq \; & \tfrac{\epsilon}{2} \times (1^2 + 1^2 + 2^2) \\
= \; & 3\epsilon
\end{aligned}$$

(e) Now we consider the *random projection* using a Gaussian matrix as introduced in the section. In next few parts, we work towards proving that if the dimension of projection is moderately big, then with high probability, the random projection preserves norms and pairwise distances approximately as described in equations (8) and (7).

Consider the random matrix $\mathbb{J} \in \mathbb{R}^{k\times d}$ with each of its entry being i.i.d. $\mathcal{N}(0,1)$ and consider the map $\psi_{\mathbb{J}} : \mathbb{R}^d \mapsto \mathbb{R}^k$ such that $\psi_{\mathbb{J}}(\vec{x}) = \frac{1}{\sqrt{k}}\mathbb{J}\vec{x}$. Show that for any *fixed non-zero vector* $\vec{u}$, the random variable $\dfrac{\|\psi_{\mathbb{J}}(\vec{u})\|^2}{\|\vec{u}\|^2}$ can be written as

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

where Z_i's are i.i.d. $\mathcal{N}(0,1)$ random variables.

$$\begin{aligned} & \frac{\|\psi_{\mathbb{J}}(\vec{u})\|^2}{\|\vec{u}\|^2} \\ = & \frac{\frac{1}{\sqrt{k}}\vec{u}^\top\mathbb{J}^\top\frac{1}{\sqrt{k}}\mathbb{J}\vec{u}}{\vec{u}^\top\vec{u}} \\ = & \frac{1}{k}\frac{\vec{u}^\top\mathbb{J}^\top\mathbb{J}\vec{u}}{\vec{u}^\top\vec{u}} \end{aligned}$$

Because all entries of $\mathbb{J}$ are i.i.d. $\mathcal{N}(0,1)$, its linear combination is also Gaussian. Therefore, we can define a new vector of random variables $\vec{w} = \mathbb{J}\vec{u}$ and each entry of w is $w_i \sim \mathcal{N}(0, \vec{u}^\top\vec{u})$

$$\begin{aligned} & \frac{\|\psi_{\mathbb{J}}(\vec{u})\|^2}{\|\vec{u}\|^2} \\ = & \frac{1}{k}\frac{\vec{w}^\top\vec{w}}{\vec{u}^\top\vec{u}} \end{aligned}$$

We notice that the denominator $\vec{u}^\top\vec{u}$ is a scalar and the numerator is a inner product of random variable. We learned in the discussion that inner product is bilinear. That is we can push the scalar into each term of the inner product. Now we need a new random variable where $\vec{z} = \frac{1}{\sqrt{\vec{u}^\top\vec{u}}}\vec{w}$ and each entry of z is $z_i \sim \mathcal{N}(0,1)$ then we have:

$$\begin{aligned} & \frac{\|\psi_{\mathbb{J}}(\vec{u})\|^2}{\|\vec{u}\|^2} \\ = & \frac{1}{k}\vec{z}^\top\vec{z} \\ = & \frac{1}{k}\sum_{i=1}^{k} Z_i^2 \\ & \textbf{where} \quad \vec{z} = \begin{bmatrix} Z_1 & Z_2 & \dots & Z_k \end{bmatrix}^\top \end{aligned}$$

(f)

For i.i.d. $Z_i \sim \mathcal{N}(0,1)$**, we have the following probability bound**

$$\mathbb{P}\left[\left|\frac{1}{k}\sum_{i=1}^{k} Z_i^2\right| \notin (1-t, 1+t)\right] \leq 2e^{-kt^2/8}, \qquad \textbf{for all } t \in (0,1).$$

Note that this bound suggests that $\sum_{i=1}^{k} Z_i^2 \approx \sum_{i=1} E[Z_i^2] = k$ **with high probability. In other words, sum of square of Gaussian random variables concentrates around its mean with high probability. Using this bound and the previous part, now show that if** $k \geq \frac{16}{\epsilon^2}\log\left(\frac{N}{\delta}\right)$**, then**

$$\mathbb{P}\left[\textbf{ for all } i,j \in [n], i \neq j, \frac{\left\|\psi_{\mathbb{J}}(\vec{x}_i) - \psi_{\mathbb{J}}(\vec{x}_j)\right\|^2}{\left\|\vec{x}_i - \vec{x}_j\right\|^2} \in (1-\epsilon, 1+\epsilon)\right] \geq 1-\delta.$$

That is show that for k **large enough, with high probability the random projection** $\psi_{\mathbb{J}}$ **approximately preserves the pairwise distances. Using this result, we can conclude that random projection serves as a good tool for dimensionality reduction if we project to enough number of dimensions. This result is popularly known as the *Johnson-Lindenstrauss Lemma*.**

Hint 1: The following (powerful technique cum) bound might be useful: For a set of events A_{ij}**, we have**

$$\mathbb{P}\left[\cap_{i,j} A_{ij}\right] = 1 - \mathbb{P}\left[(\cap_{i,j} A_{ij})^c\right] = 1 - \mathbb{P}\left[\cup_{i,j} A_{ij}^c\right] \geq 1 - \sum_{ij} \mathbb{P}\left[A_{ij}^c\right].$$

You may define the event $A_{ij} = \left\{\frac{\left\|\psi_{\mathbb{J}}(\vec{x}_i) - \psi_{\mathbb{J}}(\vec{x}_j)\right\|^2}{\left\|\vec{x}_i - \vec{x}_j\right\|^2} \in (1-\epsilon, 1+\epsilon)\right\}$ **and use the union bound above.**

I'm too lazy not to follow the hint, so let's define the event

$$A_{ij} = \left\{\frac{\left\|\psi_{\mathbb{J}}(\vec{x}_i) - \psi_{\mathbb{J}}(\vec{x}_j)\right\|^2}{\left\|\vec{x}_i - \vec{x}_j\right\|^2} \in (1-\epsilon, 1+\epsilon)\right\}$$

$$\mathbb{P}\left[\textbf{ for all } i,j \in [n], i \neq j, \frac{\left\|\psi_{\mathbb{J}}(\vec{x}_i) - \psi_{\mathbb{J}}(\vec{x}_j)\right\|^2}{\left\|\vec{x}_i - \vec{x}_j\right\|^2} \in (1-\epsilon, 1+\epsilon)\right] = \mathbb{P}\left[\cap_{i,j} A_{ij}\right] \geq 1 - \sum_{ij} \mathbb{P}\left[A_{ij}^c\right]$$

Now, we need to calculate $\sum_{ij} \mathbb{P}\left[A_{ij}^c\right]$**. Let's define a new vector where** $\vec{z} = \vec{x}_i - \vec{x}_j$

$$\begin{aligned}
&\sum_{ij, i\neq j} \mathbb{P}\left[A_{ij}^c\right] \\
&= \sum_{ij, i\neq j} \mathbb{P}\left[\left|\frac{\left\|\psi_{\mathbb{J}}(\vec{x}_i) - \psi_{\mathbb{J}}(\vec{x}_j)\right\|^2}{\left\|\vec{x}_i - \vec{x}_j\right\|^2}\right| \notin (1-\epsilon, 1+\epsilon)\right] \\
&= \sum_{ij, i\neq j} \mathbb{P}\left[\left|\frac{\left\|\psi_{\mathbb{J}}(\vec{z})\right\|^2}{\left\|\vec{z}\right\|^2}\right| \notin (1-\epsilon, 1+\epsilon)\right] \\
&= \sum_{ij, i\neq j} \mathbb{P}\left[\left|\frac{1}{k}\sum_{i=1}^{k} Z_i^2\right| \notin (1-\epsilon, 1+\epsilon)\right]
\end{aligned}$$

$$
\begin{aligned}
&\leq \sum_{ij, i \neq j} 2e^{-k\epsilon^2/8} \\
&= \frac{N(N-1)}{2} 2e^{-k\epsilon^2/8} \\
&\leq N^2 e^{-\frac{16}{\epsilon^2}\log\left(\frac{N}{\delta}\right)\frac{\epsilon^2}{8}} \\
&= N^2 e^{\log\left((\frac{N}{\delta})^{-2}\right)} \\
&= N^2 (\frac{\delta}{N})^2 \\
&= \delta^2
\end{aligned}
$$

I don't quite get the same result but we know δ is small so $1 - \delta^2 \geq 1 - \delta$. Therefore, we have:

$$
\mathbb{P}\left[\textbf{for all } i, j \in [n], i \neq j, \frac{\left\|\psi_{\mathbb{J}}(\vec{x}_i) - \psi_{\mathbb{J}}(\vec{x}_j)\right\|^2}{\left\|\vec{x}_i - \vec{x}_j\right\|^2} \in (1-\epsilon, 1+\epsilon) \right] \geq 1 - \delta.
$$

(g) Suppose there are two clusters of points $S_1 = \{\vec{u}_1, \ldots, \vec{u}_n\}$ and $S_2 = \{\vec{v}_1, \ldots, \vec{v}_m\}$ which are far apart, i.e., we have

$$d^2(S_1, S_2) = \min_{u \in S_1, v \in S_2} \|u - v\|^2 \geq \gamma.$$

Then using the previous part, show that the random projection $\psi_{\mathbb{J}}$ also approximately maintains the distance between the two clusters if k is large enough, that is, with high probability

$$d^2(\psi_{\mathbb{J}}(S_1), \psi_{\mathbb{J}}(S_2)) = \min_{u \in S_1, v \in S_2} \|\psi_{\mathbb{J}}(\vec{u}) - \psi_{\mathbb{J}}(\vec{v})\|^2 \geq (1-\epsilon)\gamma \quad \textbf{if } k \geq \frac{C}{\epsilon^2} \log(m+n)$$

for some constant C. Note that such a property can help in several machine learning tasks. For example, if the clusters of features for different labels were far in the original dimension, then this problem shows that even after randomly projecting the clusters they will remain far enough and a machine learning model may perform well even with the projected data. We now turn to visualizing some of these conclusions on a synthetic dataset.

If we can prove

$$\frac{\|\psi_{\mathbb{J}}(\vec{u}) - \psi_{\mathbb{J}}(\vec{v})\|^2}{\|u - v\|^2} \geq (1-\epsilon)$$

$$\|\psi_{\mathbb{J}}(\vec{u}) - \psi_{\mathbb{J}}(\vec{v})\|^2 \geq (1-\epsilon)\|u - v\|^2$$

Because

$$\min_{u \in S_1, v \in S_2} \|u - v\|^2 \geq \gamma.$$

We will get:

$$\min_{u \in S_1, v \in S_2} \|\psi_{\mathbb{J}}(\vec{u}) - \psi_{\mathbb{J}}(\vec{v})\|^2 \geq (1-\epsilon)\gamma$$

The conclusion to the previous part can be rewritten as:

$$\mathbb{P}\left[\textbf{for all } i \in [n], j \in [m], \frac{\|\psi_{\mathbb{J}}(\vec{u}_i) - \psi_{\mathbb{J}}(\vec{v}_j)\|^2}{\|\vec{u}_i - \vec{v}_j\|^2} \in (1-\epsilon, 1+\epsilon) \right] \geq 1 - \delta.$$

$$\textbf{where} \quad k \geq \frac{C}{\epsilon^2} \log\left(\frac{m+n}{\delta}\right)$$

Here is the work to get the inequality above. We take a new vector $\vec{s} = \vec{u} - \vec{v}$

$$\begin{aligned}
& \textstyle\sum_{ij} \mathbb{P}\left[\left| \frac{\|\psi_{\mathbb{J}}(\vec{u}_i) - \psi_{\mathbb{J}}(\vec{v}_j)\|^2}{\|\vec{u}_i - \vec{v}_j\|^2} \right| \notin (1-\epsilon, 1+\epsilon) \right] \\
= \; & \textstyle\sum_{ij} \mathbb{P}\left[\left| \frac{1}{k} \sum_{i=1}^k S_i^2 \right| \notin (1-\epsilon, 1+\epsilon) \right] \\
\leq \; & 2mne^{-k\epsilon^2/8} \\
\leq \; & (m+n)^2 e^{-k\epsilon^2/8} \leq \sigma^2 \leq \sigma
\end{aligned}$$

For the formula above we get:

$$\begin{aligned}
\tfrac{k\epsilon^2}{8} &\geq \log\left(\tfrac{(m+n)^2}{\sigma^2}\right) \\
\tfrac{k\epsilon^2}{8} &\geq \tfrac{16}{\epsilon^2} \log\left(\tfrac{m+n}{\sigma}\right)
\end{aligned}$$

Let's take a relatively small $\delta = 0.05$, then we have:

$$\mathbb{P}\left[\text{ for all } i \in [n], j \in [m], \frac{\|\psi_{\mathbb{J}}(\vec{u}_i) - \psi_{\mathbb{J}}(\vec{v}_j)\|^2}{\|\vec{u}_i - \vec{v}_j\|^2} \in (1-\epsilon, 1+\epsilon)\right] \geq 0.95.$$

$$\text{where} \quad k \geq \frac{C'}{\epsilon^2}\log(m+n)$$

$$C' = C\left(1 - \frac{\log(0.05)}{\log(m+n)}\right)$$

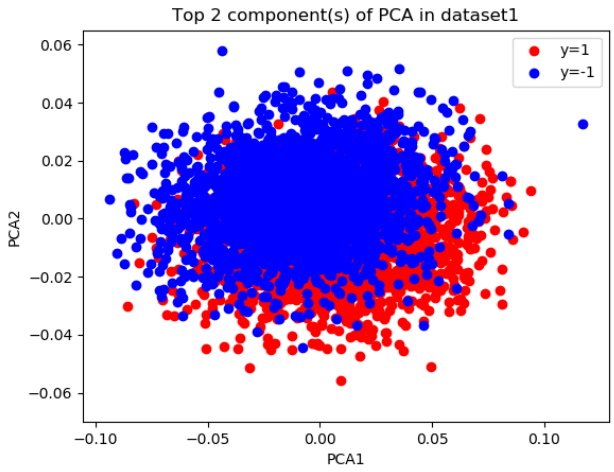
As long as S_1 and S_2 are finite but not empty sets C' is bounded in $[C, C\left(1 - \frac{\log(0.05)}{\log(2)}\right)]$, so we have:

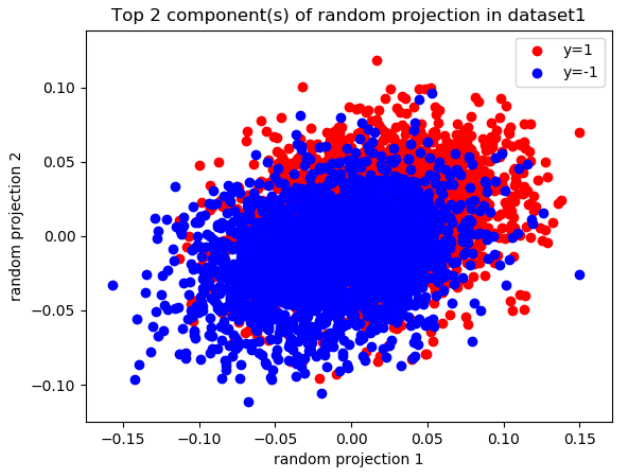
$$d^2(\psi_{\mathbb{J}}(S_1), \psi_{\mathbb{J}}(S_2)) = \min_{u \in S_1, v \in S_2} \|\psi_{\mathbb{J}}(\vec{u}) - \psi_{\mathbb{J}}(\vec{v})\|^2 \geq (1-\epsilon)\gamma \quad \text{if } k \geq \frac{C'}{\epsilon^2}\log(m+n)$$

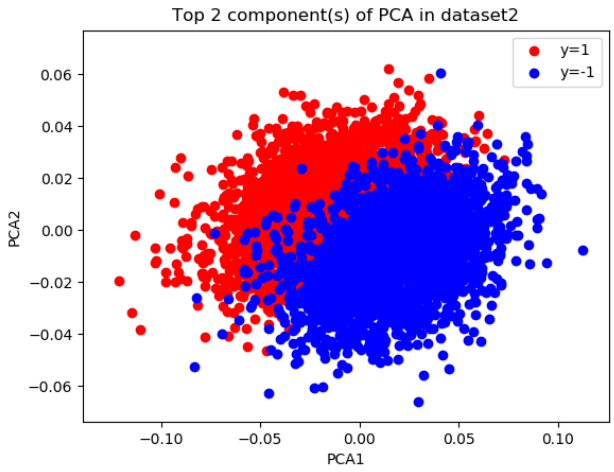
(h) In the next few parts, we visualize the effect of PCA projections and random projections on a classification task. You are given 3 datasets in the data folder and a starter code.

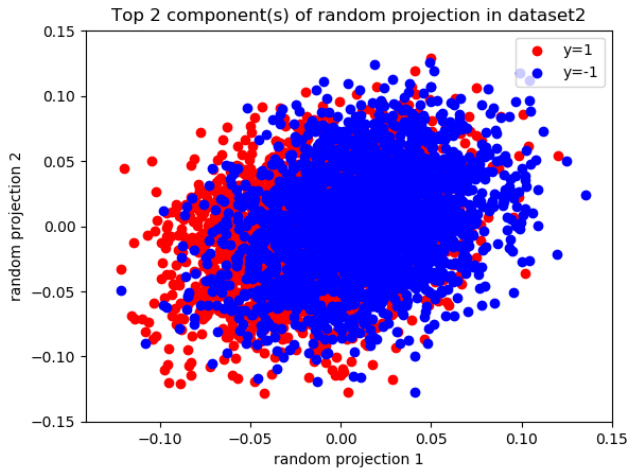
Use the starter code to load the three datasets one by one. Note that there are two unique values in $\vec{y}$. Visualize the features of $\mathbb{X}$ these datasets using (1) Top-2 PCA components, and (2) 2-dimensional random projections. Use the code to project the features to 2 dimensions and then scatter plot the 2 features with a different color for each class. Note that you will obtain 2 plots for each dataset (total 6 plots for this part). Do you observe a difference in PCA vs random projections? Do you see a trend in the three datasets?

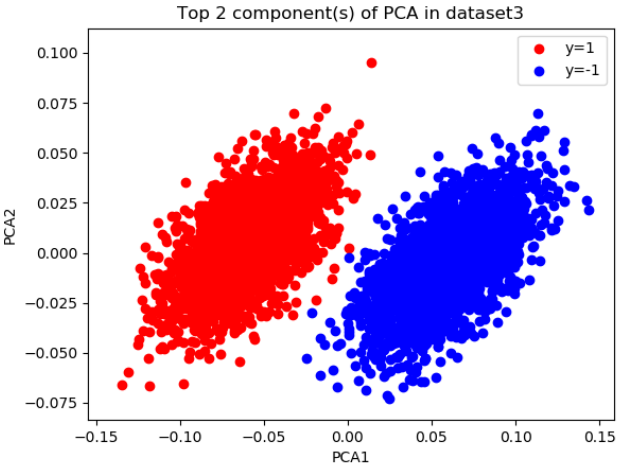
I can see difference between PCA and random projection. Random projection seems to give a result with more overlaps between two classes. The orientations of the projected data are also different for random projection sometimes (i.e. in data set 3). The two classes in data set 3 seem to be almost linear separable but not so much in the first two data sets.

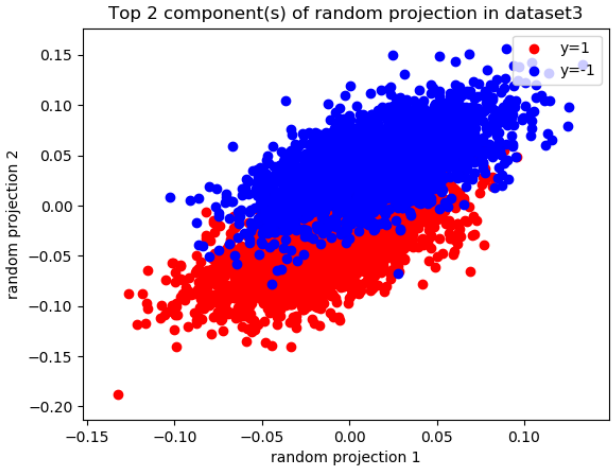









Top 2 component(s) of PCA in dataset3
y=1
y=-1
PCA2
PCA1
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15

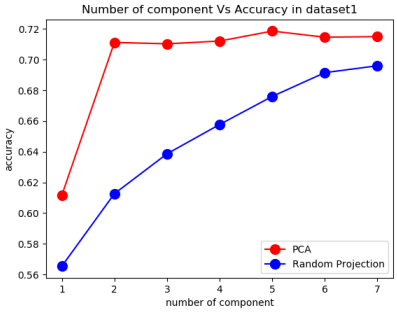

Top 2 component(s) of random projection in dataset3
y=1
y=-1
random projection 2
random projection 1
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10

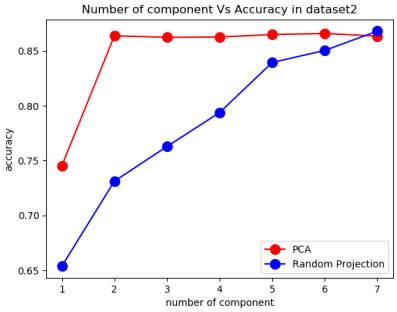
(i) For each dataset, we will now fit a linear model on different projections of features to perform classification. The code for fitting a linear model with projected features and predicting a label for a given feature, is given to you. Use these functions and write a code that does prediction in the following way: (1) Use top k-PCA features to obtain one set of results, and (2) Use k-dimensional random projection to obtain the second set of results (take average accuracy over 10 random projections for smooth curves). Use the projection functions given in the starter code to select these features. You should vary k from 1 to d where d is the dimension of each feature $\vec{x}_i$. Plot the accuracy for PCA and Random projection as a function of k. Comment on the observations on these accuracies as a function of k and across different datasets.

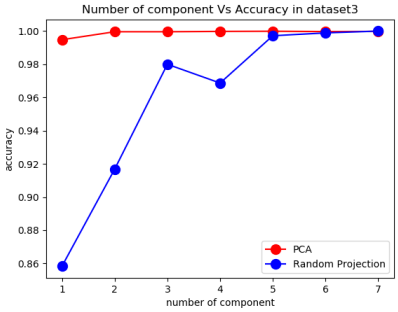
For PCA, the accuracy curves are sigmoid-like, which means it increases very when k reaches a certain threshold.

For random projection, the accuracy increases as more components are added.

The first two data sets look very similar. This tells us even the data sets can be very different, we could only use a couple of principle components to describe most of their variances. The last data set begin with high accuracy even with only one top principle component. This means two classes in data set 3 are almost distinct with only one top principle component, which indicates that further addition of principle components do not improve the accuracy very much. On the other hand, the trend for random projection is always increasing. That is, adding more components in random projection usually help improve the accuracy.



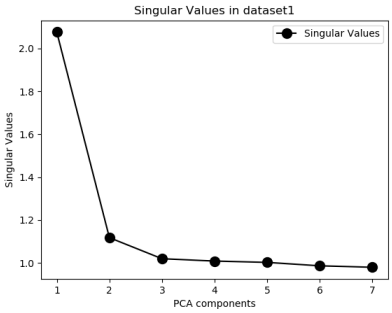


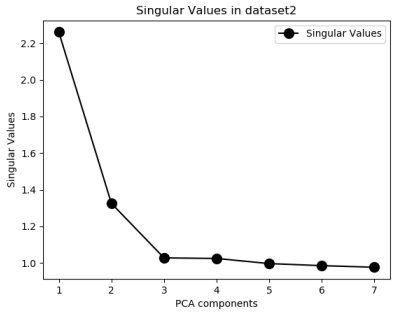


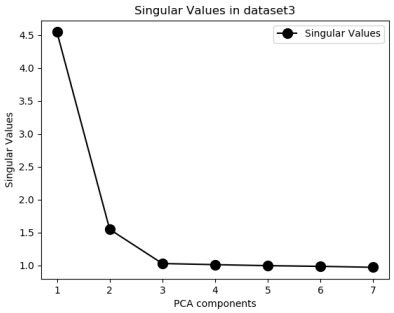
(j) Now plot the singular values for the feature matrices of the three datasets. Do you observe a pattern across the three datasets? Does it help to explain the performance of PCA observed in the previous parts?

Yes. The singular value curves are also sigmoid-like. In all three data sets the singular values drop a lot after the first two principle components. That is, the variances in the data sets can be mostly explained by the first two principle components.

This is helpful to explain the result we got in the previous part. It is at the same k where the accuracy increases the most and the singular value decreases the most. This is true for the first two data sets but not dataset 3. This might be because dataset 3 is linearly separable even only using the first principle component.







```
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 #########
np.random.seed(0) # seed the random number generator
n_dataset = 3 # the number of data set
n_trials = 10 # to average for accuracies over random projections
default_k = 2 # the default number of principle component to keep
for iData in range(1, n_dataset + 1):
# to load the data:
data = np.load('data' + str(iData) + '.npz')
X = data['X']
y = data['y']
n, d = X.shape

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

# Using PCA and Random Projection for:
# Visualizing the datasets
pcaXk = pca_proj(X, default_k)
randomXk = random_proj(X, default_k)
one_ids = y == 1
neg_one_ids = y == -1

plt.figure()
plt.scatter(pcaXk[one_ids, 0], pcaXk[one_ids, 1], c='r', marker='o', label='y=1')
plt.scatter(pcaXk[neg_one_ids, 0], pcaXk[neg_one_ids, 1], c='b', marker='o', label='y=-1')
plt.legend(loc='upper right')
plt.title("Top " + str(default_k) + " component(s) of PCA in dataset" + str(iData))
```

```
plt.xlabel("PCA1")
plt.ylabel("PCA2")
plt.savefig("Figure_3h_visual_pca_data" + str(iData) + ".png")
plt.close()

plt.figure()
plt.scatter(randomXk[one_ids, 0], randomXk[one_ids, 1], c='r', marker='o', label='y=1')
plt.scatter(randomXk[neg_one_ids, 0], randomXk[neg_one_ids, 1], c='b', marker='o', label='y=-1')
plt.title("Top " + str(default_k) + " component(s) of random projection in dataset" + str(iData))
plt.legend(loc='upper right')
plt.xlabel("random projection 1")
plt.ylabel("random projection 2")
plt.savefig("Figure_3h_visual_random_data" + str(iData) + ".png")
plt.close()

# Computing the accuracies over different datasets.
rand_accuracies = np.zeros(d)
pca_accuracies = np.zeros(d)
for k in range(d):
for iTry in range(n_trials):
rand_accuracies[k] += rand_proj_accuracy_split(X, y, k + 1)
pca_accuracies[k] += pca_proj_accuracy(X, y, k + 1)

rand_accuracies /= n_trials
pca_accuracies /= n_trials

plt.figure()
plt.plot(np.linspace(1, d, d), pca_accuracies, c='r', marker='.', markersize=20, label='PCA')
plt.plot(np.linspace(1, d, d), rand_accuracies, c='b', marker='.', markersize=20, label='Random
    Projection')
plt.title("Number of component Vs Accuracy in dataset" + str(iData))
plt.legend(loc='lower right')
plt.xlabel("number of component")
plt.ylabel("accuracy")
plt.savefig("Figure_3i_numVsAccuracy_data" + str(iData) + ".png")
plt.close()
# Don't forget to average the accuracy for multiple
# random projections to get a smooth curve.

# And computing the SVD of the feature matrix
Sigma = np.linalg.svd(X, compute_uv=False)
plt.figure()
plt.plot(np.linspace(1, len(Sigma), len(Sigma)), Sigma, c='k', marker='.', markersize=20, label='Singular
    Values')
plt.title("Singular Values in dataset" + str(iData))
plt.legend(loc='upper right')
plt.xlabel("PCA components")
plt.ylabel("Singular Values")
plt.savefig("Figure_3j_SingularValues_data" + str(iData) + ".png")
plt.close()

######## YOU CAN PLOT THE RESULTS HERE ########

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

Question 4. Fruits and Veggies! The goal of the problem is help the class build a dataset for image classification, which will be used later in the course to classify fruits and vegetables. Please pick ten of the following fruits:

1. Apple
2. Banana
3. Oranges
4. Grapes
5. Strawberry
6. Peach
7. Cherry
8. Nectarine
9. Mango
10. Pear
11. Plum
12. Watermelon
13. Pineapple

and ten of the following veggies:

1. Spinach
2. Celery
3. Potato (not sweet potato)
4. Bell Peppers
5. Tomato
6. Cabbage
7. Radish
8. Broccoli
9. Cauliflower
10. Carrot
11. Eggplant
12. Garlic
13. Ginger

Take two pictures of each specific fruit and veggie, for a total of 20 fruit pictures, 20 veggie pictures against any background such that the fruit is centered in the image and the object takes up approximately a third of the image in area; see below for examples. Save these pictures as .png files. **Do not** save the pictures as .jpg files, since these are lower quality and will interfere with the results of your future coding project. Place all the images in a folder titled *data*. Each image should be titled *[fruit name]_[number].png* or *[veggie name]_[number].png* where [number]$\in \{0, 1\}$. Ex: apple_0.png, apple_1.png, banana_0.png, etc ... (the ordering is irrelevant). Please also include a file titled *rich_labels.txt* which contain entries on new lines prefixed by the file name *[fruit name]_[number]* or *[veggie name]_[number].png*, followed by a description of the image (maximum of eight words) with only a space delimiter in between. Ex: apple_0 one fuji red apple on wood table. To turn in the folder compress the file to a .zip and upload it to Gradescope.

Figure 3: Example of properly centered images of four bananas against a table (left) and one orange on a tree (right).

Please keep in mind that data is an integral part of Machine Learning. A large part of research in this field relies heavily on the integrity of data in order to run algorithms. It is, therefore, vital that your data is in the proper format and is accompanied by the correct labeling not only for your grade on this section, but for the integrity of your data.

Note that if your compressed file is over 100 MB, you will need to downsample the images. You can use the following functions from skimage to help.

```
from skimage.io import imread, imsave
from skimage.transform import resize
```

The images are packed and uploaded on Gradescope. The code I used to compress my images is listed below:

```
from skimage.io import imsave
from skimage.transform import resize
from os import listdir
from os.path import isfile, join
from rawpy import imread

mypath = 'D:\Chrome\Camera\dng'
savepath = 'D:\Chrome\Camera\data'
onlyfiles = [f for f in listdir(mypath) if isfile(join(mypath, f))]

for _, filename in enumerate(onlyfiles):
pure_name = filename[:-4]
img = imread(mypath + '\\' + filename).postprocess()
height, width, _ = img.shape
start = int(width/2-height/2)
# img = img[:, start:start+height]
img = resize(img, (1000, 1000), mode='reflect')
imsave(savepath + '\\' + pure_name + '.png', img)
```

Question 5. Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

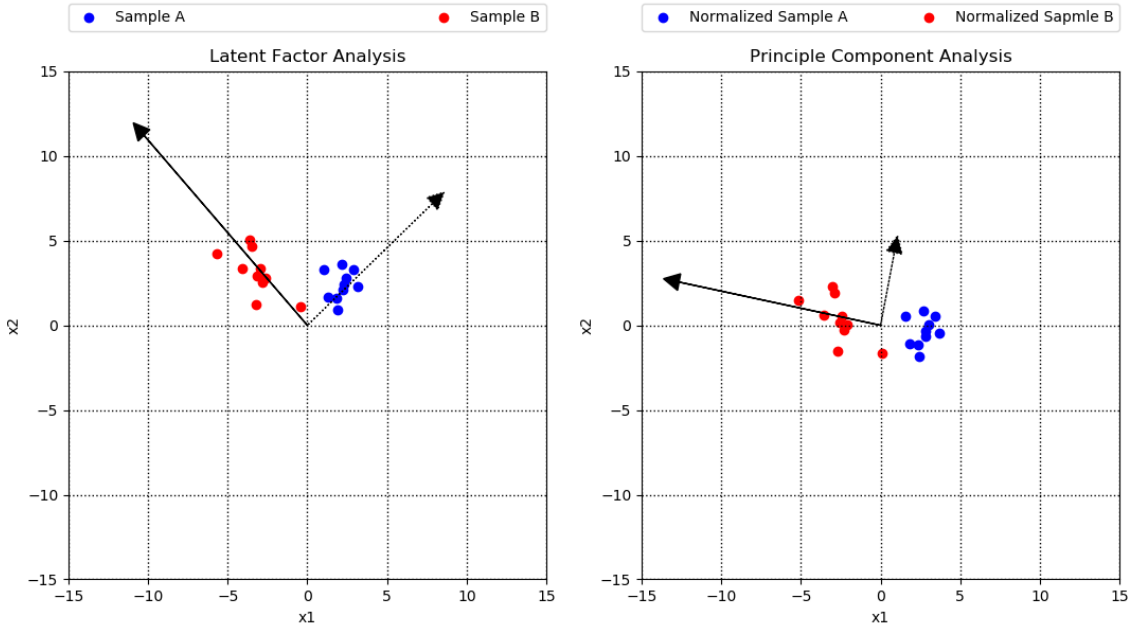
As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.

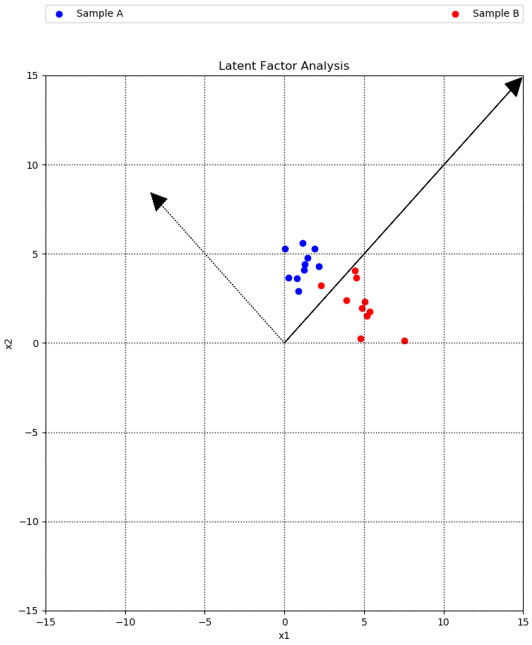
What if we don't center the data before doing SVD? What does the result tell us?

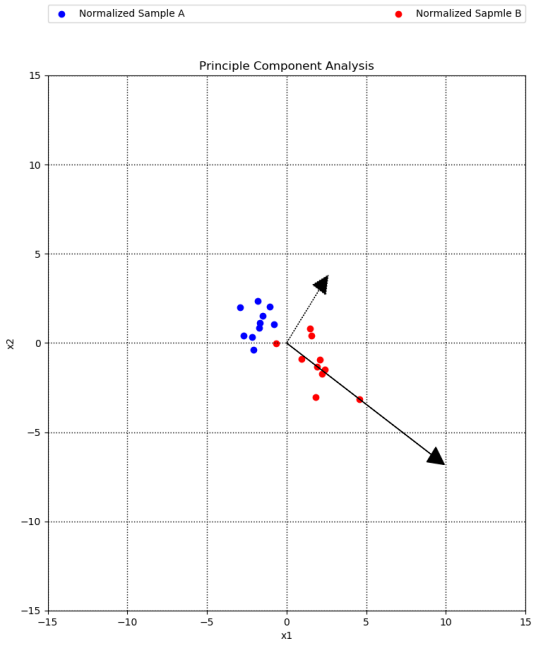
It turns out this is the so-called "Latent Factor Analysis", which can be found here: `https://en.wikipedia.org/wiki/Factor_analysis` and in Jonathan Shewchuk's lecture (`https://people.eecs.berkeley.edu/~jrs/189/lec/24.pdf`). Instead of working through the proofs, I would like to do an example here.

I randomly generated two populations of normally distributed random sample points. Then I compared the directions of eigenvectors before(left) and after normalization (right). As you can see, without normalization, the eigenvectors are pointing to the two clusters. This can serve as a way to cluster and find similarities in our data.

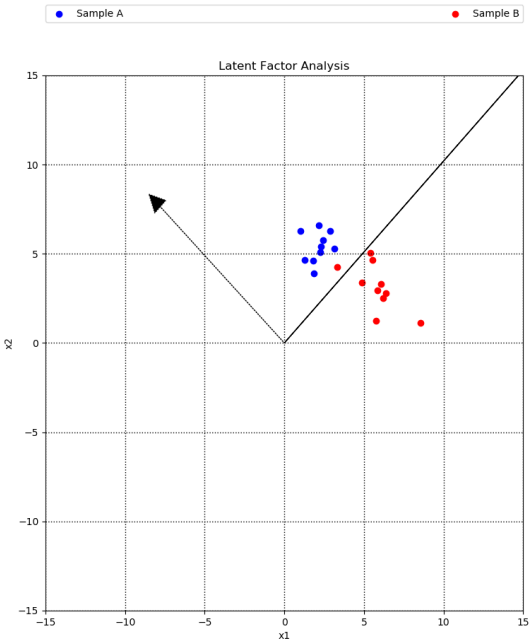


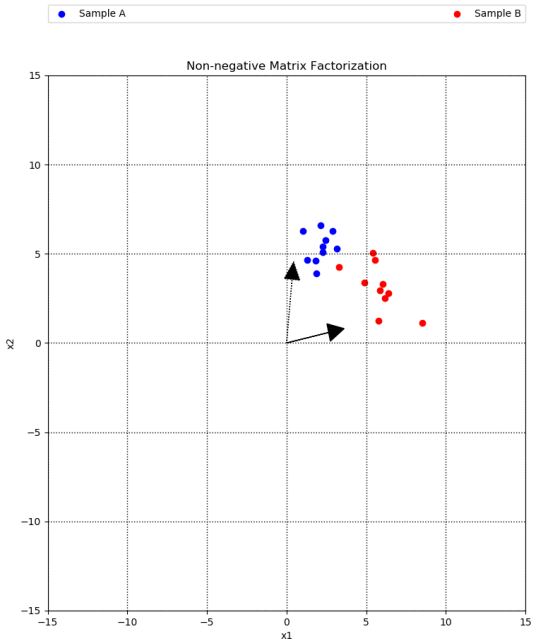
However, sometimes we don't get that clear result. This leads to a different analysis called non-negative matrix factorization (NMF). (`https://stats.stackexchange.com/questions/152879/latent-semantic-indexing-and-data-centering`)





As we can see in the figure below, NMF does a much better job because it doesn't require the derived latent bases to be orthogonal (see the figure below `https://people.eecs.berkeley.edu/~jfc/hcc/courseSP05/lecs/lec14/NMF03.pdf`).
On the other hand, SVD will give negative values which doesn't represent the data.





LSI code:

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(0)
n = 10
# create two clusters of data
# mu =[3, 3]
```

```
mu = [2, 5]
sigma = [[1, 0.5], [0.5, 1]]
samplesA = np.random.multivariate_normal(mu, sigma, size=n)

# mu =[-3, 3]
mu = [5, 2]
sigma = [[1, -0.5], [-0.5, 1]]
samplesB = np.random.multivariate_normal(mu, sigma, size=n)

norm_samples = np.vstack((samplesA, samplesB))
norm_samplesA = samplesA - np.ones((samplesA.shape[0], 1)).dot(
np.reshape(np.mean(norm_samples, axis=0), (1, norm_samples.shape[1])))

norm_samplesB = samplesB - np.ones((samplesB.shape[0], 1)).dot(
np.reshape(np.mean(norm_samples, axis=0), (1, norm_samples.shape[1])))

norm_samples = np.vstack((norm_samplesA, norm_samplesB))

_, norm_Lambda, norm_Vd = np.linalg.svd(norm_samples)

samples = np.vstack((samplesA, samplesB))
_, Lambda, Vd = np.linalg.svd(samples)

plt.figure()
plt.subplot(1, 2, 1)
plt.scatter(samplesA[:, 0], samplesA[:, 1], c='b', label='Sample A')
plt.scatter(samplesB[:, 0], samplesB[:, 1], c='r', label='Sample B')
plt.arrow(0, 0, -Lambda[0] * Vd[0, 0], -Lambda[0] * Vd[0, 1], head_width=1,
head_length=1, fc='k', ec='k')
plt.arrow(0, 0, -Lambda[1] * Vd[1, 0], -Lambda[1] * Vd[1, 1], head_width=1,
head_length=1, fc='k', ec='k', linestyle=':')
lgd = plt.legend(bbox_to_anchor=(0., 1.03, 1., .102), loc=2, ncol=2, mode="expand", borderaxespad=0.)
plt.xlabel('x1')
plt.ylabel('x2')
plt.grid(color='k', linestyle=':', linewidth=1)
plt.title('Latent Factor Analysis')
plt.xlim([-15, 15])
plt.ylim([-15, 15])

plt.subplot(1, 2, 2)
plt.scatter(norm_samplesA[:, 0], norm_samplesA[:, 1], c='b', label='Normalized Sample A')
plt.scatter(norm_samplesB[:, 0], norm_samplesB[:, 1], c='r', label='Normalized Sapmle B')
plt.arrow(0, 0, norm_Lambda[0] * norm_Vd[0, 0], norm_Lambda[0] * norm_Vd[0, 1], head_width=1,
head_length=1, fc='k', ec='k')
plt.arrow(0, 0, -norm_Lambda[1] * norm_Vd[1, 0], -norm_Lambda[1] * norm_Vd[1, 1], head_width=1,
head_length=1, fc='k', ec='k', linestyle=':')
lgd = plt.legend(bbox_to_anchor=(0., 1.03, 1., .102), loc=2, ncol=2, mode="expand", borderaxespad=0.)
plt.xlabel('x1')
plt.ylabel('x2')
plt.grid(color='k', linestyle=':', linewidth=1)
plt.title('Principle Component Analysis')
plt.xlim([-15, 15])
plt.ylim([-15, 15])
mng = plt.get_current_fig_manager()
mng.window.state('zoomed')
plt.show()
```

NMF code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import NMF
```

```
np.random.seed(0)
n = 10
# create two clusters of data
# mu =[3, 3]
mu = [3, 6]
sigma = [[1, 0.5], [0.5, 1]]
samplesA = np.random.multivariate_normal(mu, sigma, size=n)


# mu =[-3, 3]
mu = [6, 3]
sigma = [[1, -0.5], [-0.5, 1]]
samplesB = np.random.multivariate_normal(mu, sigma, size=n)
samplesB = np.abs(samplesB)

samples = np.vstack((samplesA, samplesB))
_, Lambda, Vd = np.linalg.svd(samples)

model = NMF(n_components=2, init='random', random_state=0)
W = model.fit_transform(samples)
H = model.components_

plt.figure()
plt.subplot(1, 2, 1)
plt.scatter(samplesA[:, 0], samplesA[:, 1], c='b', label='Sample A')
plt.scatter(samplesB[:, 0], samplesB[:, 1], c='r', label='Sample B')
plt.arrow(0, 0, -Lambda[0] * Vd[0, 0], -Lambda[0] * Vd[0, 1], head_width=1,
head_length=1, fc='k', ec='k')
plt.arrow(0, 0, -Lambda[1] * Vd[1, 0], -Lambda[1] * Vd[1, 1], head_width=1,
head_length=1, fc='k', ec='k', linestyle=':')
lgd = plt.legend(bbox_to_anchor=(0., 1.03, 1., .102), loc=2, ncol=2, mode="expand", borderaxespad=0.)
plt.xlabel('x1')
plt.ylabel('x2')
plt.grid(color='k', linestyle=':', linewidth=1)
plt.title('Latent Factor Analysis')
plt.xlim([-15, 15])
plt.ylim([-15, 15])

plt.subplot(1, 2, 2)
plt.scatter(samplesA[:, 0], samplesA[:, 1], c='b', label='Sample A')
plt.scatter(samplesB[:, 0], samplesB[:, 1], c='r', label='Sample B')
plt.arrow(0, 0, H[0, 0], H[1, 0], head_width=1,
head_length=1, fc='k', ec='k')
plt.arrow(0, 0, H[0, 1], H[1, 1], head_width=1,
head_length=1, fc='k', ec='k', linestyle=':')
lgd = plt.legend(bbox_to_anchor=(0., 1.03, 1., .102), loc=2, ncol=2, mode="expand", borderaxespad=0.)
plt.xlabel('x1')
plt.ylabel('x2')
plt.grid(color='k', linestyle=':', linewidth=1)
plt.title('Non-negative Matrix Factorization')
plt.xlim([-15, 15])
plt.ylim([-15, 15])
mng = plt.get_current_fig_manager()
mng.window.state('zoomed')
plt.show()
```