Ankan Mazumdar A20541357
CS584 Machine Learning - Assignment 4

Problem 1 (15 Points): Nonnegative Matrix Factorization

1. **5 Points.** Find a nonnegative factorization of the matrix

$$A = \begin{bmatrix} 4 & 6 & 5 \\ 1 & 2 & 3 \\ 7 & 10 & 7 \\ 6 & 8 & 4 \\ 6 & 10 & 11 \end{bmatrix}$$

Indicate the steps in your method and show the intermediate results.

Answer-
NMF is basically a dimensionality-reduction technique used for feature extraction. It breaks down a nonnegative matrix A, into two nonnegative matrices, B and C, where A can be approximately depicted as A ≈ BC.

The general idea is to choose two matrices B and C randomly and keep on updating those until convergence(minimum error). The update rule, Number of iterations, error threshold etc. depends on our chosen approach. Here are a couple of approaches-

1. Multiplicative Update Rule -

- We will Initialize nonnegative factor matrices B (m x k) and C(k x n) with nonnegative random values for given matrix A(m x n). We have to choose the value of 'k', the number of iterations and an error threshold value to determine convergence.
- For each iteration, we will apply multiplicative update rule on B and C:

```
B = B * (A @ C.T) / (B @ C @ C.T)
C = C * (B.T @ A) / (B.T @ B @ C)
```

Here, `A @ C.T` represents matrix product of A and C Transpose.
Then `B * (A @ C.T)` means `B` getting multiplied by matrix product of A and C Transpose and that is further divided by matrix product of B, C and C Transpose `(B @ C @ C.T)`.

- Check Convergence (min error) by calculating norm error between matrix A and the reconstructed matrix (B.C) i.e. error = min ||A - BC||
- Will keep repeating until the error comes below the threshold or a max iteration is reached.
  Return the factor matrices B and C.

  I have applied these same steps in my code snippet below-

```python
import numpy as np

# Define the input matrix A (5x3)
A = np.array([[4, 6, 5],
              [1, 2, 3],
              [7, 10, 7],
              [6, 8, 4],
              [6, 10, 11]])

# Initialize matrices B and C with nonnegative random values
B = np.random.rand(5, 2)
C = np.random.rand(2, 3)

# Set the number of iterations
num_iterations = 50
# Set the error threshold to stop iterating
error_threshold = 1

# Create arrays to store results
errors = []
products = []

# Print the initial matrices
print("Initial Matrix A:")
print(A)
print("Initial Matrix B:")
print(B)
print("Initial Matrix C:")
print(C)

# Perform iterations
for iteration in range(num_iterations):
    # Update B
    B = B * (A @ C.T) / (B @ C @ C.T)

    # Update C
    C = C * (B.T @ A) / (B.T @ B @ C)
```

```
    # Calculate Frobenius norm error
    error = np.linalg.norm(A - B @ C, 'fro')
    errors.append(error)

    # Calculate the product of updated B and C
    product = B @ C
    products.append(product)

    # Print the results for this iteration
    print(f"\nIteration {iteration + 1}:")
    print("Updated B:")
    print(B)
    print("Updated C:")
    print(C)
    print(f"Frobenius Norm Error: {error}")
    print(f"Product of Updated B and C:")
    print(product)
    # Check for convergence
    if error < error_threshold:
        print(f"Converged (Error < {error_threshold})")
        break

# Print the final results
print("\nFinal Updated B:")
print(B)
print("Final Updated C:")
print(C)
print("Final Frobenius Norm Error:")
print(errors[-1])
print("Final Product of Updated B and C:")
print(products[-1])
```

Result-

```
Iteration 8:
Updated B:
[[ 6.77586942  6.39282819]
 [ 4.0417664   0.16424898]
 [ 9.16669362 13.57724634]
 [ 2.95481525 17.63231072]
 [15.54552032  5.00917505]]
Updated C:
[[0.31690687 0.51352085 0.6706265 ]
 [0.28159613 0.38238974 0.09121144]]
Frobenius Norm Error: 0.9177495580658105
Product of Updated B and C:
[[ 3.94751519  5.92410214  5.12717661]
 [ 1.3271154   2.13833846  2.72549702]
 [ 6.72828811  9.89908798  7.38582777]
 [ 5.90159162  8.25977388  3.5898458 ]
 [ 6.33704641  9.89840601 10.88213187]]
Converged (Error < 1)

Final Updated B:
[[ 6.77586942  6.39282819]
 [ 4.0417664   0.16424898]
 [ 9.16669362 13.57724634]
 [ 2.95481525 17.63231072]
 [15.54552032  5.00917505]]
Final Updated C:
[[0.31690687 0.51352085 0.6706265 ]
 [0.28159613 0.38238974 0.09121144]]
Final Frobenius Norm Error:
0.9177495580658105
Final Product of Updated B and C:
[[ 3.94751519  5.92410214  5.12717661]
 [ 1.3271154   2.13833846  2.72549702]
 [ 6.72828811  9.89908798  7.38582777]
 [ 5.90159162  8.25977388  3.5898458 ]
 [ 6.33704641  9.89840601 10.88213187]]
```

Explanation-

Matrices B and C are initialized with nonnegative random values using numpy.random.rand function. The Value of K is kept = 2 , which means matrix B will have dimensions (4 x k) and matrix C will be (k x 4), which ensures simpler resultant matrices structure and easier computations compared to higher value of k.

The number of iterations is set as 50 and error threshold as ‘1’(I have tried setting other values too, the most accurate results are given with this combination only).

Then, the Update Rule is being applied on B and C matrices. Norm Error and Product of B & C is stored in their numpy arrays.
Therefore for each iteration, I’m showing following things-
-Updated B & C matrices values.
-Frobenius Norm Error
-Product of Updated B and C

The function np.linalg.norm(A - B @ C, 'fro') will calculate the Frobenius norm of the matrix ||A - B @ C||. In general, for a matrix W with elements w_ij, the Frobenius norm is given by ||W||_F = sqrt(ΣΣ|w_ij|^2)

Eventually after iteration 8 , in my code, error value becomes lesser than 1 (set threshold) which means convergence is reached now.Additionally, I'm also printing final Product of Updated B and C which is approximately close to original matrix A.

2. Alternating Least Squares - ALS is another approach for solving NMF problems. It alternates between fixing one factor matrix and optimizing the other matrix. Basically, we will fix matrix B's value and will solve C and vice versa. And the update process continues until convergence is achieved. ALS can introduce negative values which can be handled though. ALS is suitable for large scale NMF problems, it sparsity, handling missing data etc. compared to the Multiplicative Update Rule which is computationally easier to implement.

Problem 1.2

2. Consider the matrix A that is the product of nonnegative matrices B and C (i.e, $A = BC$), where

$$A = \begin{bmatrix} 12 & 22 & 41 & 35 \\ 19 & 20 & 13 & 48 \\ 11 & 14 & 16 & 29 \\ 14 & 16 & 14 & 36 \end{bmatrix}, B = \begin{bmatrix} 10 & 1 \\ 1 & 9 \\ 3 & 4 \\ 2 & 6 \end{bmatrix}, C = \begin{bmatrix} 1 & 2 & 4 & 3 \\ 2 & 2 & 1 & 5 \end{bmatrix}$$

5 Points. Which rows of A are positive linear combinations of other rows of A?

5 Points. Find an approximate nonnegative factorization of A.

Answer-

One row of a matrix can be called as a positive linear combination of other rows when we can create that row by adding or subtracting the other rows of the matrix and multiplying them by some numbers (coefficients).
Another way to determine if a row in matrix A is a positive linear combination of other row is to check if the coefficients obtained by adding or subtracting the element's current row with another row is non-negative. If any of the coefficients are negative, the row is not a positive linear combination. If all the coefficients are non-negative, the row is a positive linear combination of other rows.

Coefficients for Row 1 - Row 2: [-7 2 28 -13]
Coefficients for Row 1 - Row 3: [1 8 25 6]
Coefficients for Row 1 - Row 4: [-2 6 27 -1]

Coefficients for Row 2 - Row 1: [7 -2 -28 13]
Coefficients for Row 2 - Row 3: [8 6 -3 19]
Coefficients for Row 2 - Row 4: [5 4 -1 12]

Coefficients for Row 3 - Row 1: [-1 -8 -25 -6]
Coefficients for Row 3 - Row 2: [-8 -6   3 -19]
Coefficients for Row 3 - Row 4: [-3 -2 2 -7]

Coefficients for Row 4 - Row 1: [2 -6 -27   1]
Coefficients for Row 4 - Row 2: [-5 -4   1 -12]
Coefficients for Row 4 - Row 3: [3 2 -2 7]

Conclusion - Only Coefficients for Row 1 - Row 3: [1 8 25 6] are all non negative, hence only we can say only Row 1 of A is a positive linear combination of other rows(Row 1 - Row 3).

```
import numpy as np

def is_positive_linear_combination(A, i):

  for j in range(A.shape[0]):
    if i != j and all(coeff >= 0 for coeff in A[i] - A[j]):
      return True
  return False

# Define matrix A
A = np.array([[12, 22, 41, 35],
              [19, 20, 13, 48],
              [11, 14, 16, 29],
              [14, 16, 14, 36]])

# Determine which rows are positive linear combinations
positive_linear_combinations = []
for i in range(A.shape[0]):
  positive_linear_combinations.append(is_positive_linear_combination(A,
i))

# Identify and print rows of A that are positive linear combinations
for i, is_positive in enumerate(positive_linear_combinations):
  if is_positive:
    print(f"\nRow {i + 1} of A is a positive linear combination of other
rows.")
  else:
    print(f"\nRow {i + 1} of A is not a positive linear combination of
other rows.")
```

```
Row 1 of A is a positive linear combination of other rows.

Row 2 of A is not a positive linear combination of other rows.

Row 3 of A is not a positive linear combination of other rows.

Row 4 of A is not a positive linear combination of other rows.
```

Code snippet-

```python
import numpy as np
from sklearn.decomposition import NMF

# Define matrix A
A = np.array([[12, 22, 41, 35],
              [19, 20, 13, 48],
              [11, 14, 16, 29],
              [14, 16, 14, 36]])

# Specify the number of components (rank) for the factorization
n_components = 2

# Perform NMF done using NMF function, but should be done using iterations
model = NMF(n_components=n_components, init='random', random_state=0)
W = model.fit_transform(A)
H = model.components_

# Print the factorization results
print("Matrix W:")
print(W)
print("Matrix H:")
print(H)

# Reconstruct A from W and H
A_reconstructed = W @ H
print("Reconstructed A:")
print(A_reconstructed)
```

```
Matrix W:
[[5.06811038 0.        ]
 [1.60705285 3.047195  ]
 [1.97808246 1.26663945]
 [1.73076573 1.98572031]]
Matrix H:
[[ 2.36776385  4.34091181  8.08956141  6.9059836 ]
 [ 4.98655297  4.27400907  0.         12.11011047]]
Reconstructed A:
[[12.00008854 22.00022021 40.99879018 35.00028718]
 [19.00012093 19.99981379 13.00035271 48.00014875]
 [10.99979684 14.00031001 16.00181953 28.99974868]
 [13.99994402 16.00008801 14.00113563 35.99993205]]
```

Explanation -

For performing NMF on matrix A, we can use NMF function Scikit-learn package. It is predefined function which undergoes the same operations which we did manually in Problem 1.1. Here are the steps executed in the code -

1. Matrix A is initialized with the provided values.
2. The n_components is set to 2, which specifies the resulting matrices would be W (m x n_components) and H (n_components x n).
3. NMF is performed using the NMF function, NMF model is initialized with the rank =2 and random initialization method (init='random'), and a random state =0 ensures fixed value for initialization of matrices W and H each time.
4. Model's fit_transform method is used to get the factorization results.It breaks down A into two non-negative matrices, W (m x n_components) and H (n_components x n).
5. Internally, it calculates the Frobenius norm , error loss of the matrix ||A - W @ H||_F. In general, for a matrix W with elements w_ij, the Frobenius norm is given by ||W||_F = sqrt(ΣΣ|w_ij|^2).
6. Finally, Matrices W and H is printed in output and for verifying that our results is correct, matrix A is reconstructed from matrices W and H by multiplying them together, and the reconstructed matrix A_reconstructed which is approximately quite similar values to original A.

—----------Problem 1 End----------

Problem 2 (20 Points): SVD: Image Compression

In this experiment, we will use the singular value decomposition (SVD) as a tool for compressing raw image data. This is not how images are actually compressed; for example, JPEG compression algorithms do more fancy (and interesting) computations. However, the idea is similar: if we are willing to tolerate a certain amount of distortion, then we can get away with a much more concise data representation.

1. **3 Points.** Read the given image file (`'mandrill_color.png'`), and convert it into grayscale by averaging the R,G,B values for each pixel. Your image is now a 288×288 matrix; call it $\mathbf{X}$.
2. **4 Points.** Perform an SVD of $\mathbf{X}$ to obtain the decomposition $\{\mathbf{U}, \Sigma, \mathbf{V}\}$. Plot the singular values (i.e., the diagonal entries of Σ) in decreasing order.
3. **5 Points.** Choose $k = 10$, and reconstruct an approximation of $\mathbf{X}$ using the top k singular values and vectors, $\mathbf{U}_k$, $\mathbf{V}_k$, and Σ_k. Display this approximation, and calculate how many numbers you needed to store this approximate image representation. Divide by the original size of $\mathbf{X}$ to get the compression ratio.
4. **8 Points.** Repeat this experiment for $k = 20, 40, 60$. Display these images, and report their compression ratios in the form of a table. Is there any benefit in going for higher k?

Answer-

In this problem we will explore Singular Value Decomposition (SVD) technique in image compression. The SVD operation basically decomposes matrix X into three matrices {U, S, V}:

which is X=USV where

U is the orthogonal matrix containing the left singular vectors.

S is the diagonal matrix which contains the singular values meaning it has non-zero values only on its main diagonal, and all other entries are zero.

V is the transpose of an orthogonal matrix containing the right singular vectors.
We will accomplish this by python using pillow, Image, numpy and matplotlib libraries.

Please find the code-

```
#Problem 2.1
pip install numpy pillow
from PIL import Image
import numpy as np

# Open the image file
image = Image.open('/content/mandrill_color.png')

# Converting the image to grayscale
grayscale_image = image.convert('L')

# transform  the grayscale image to  array
X = np.array(grayscale_image)

print("Shape of X:", X.shape)
```

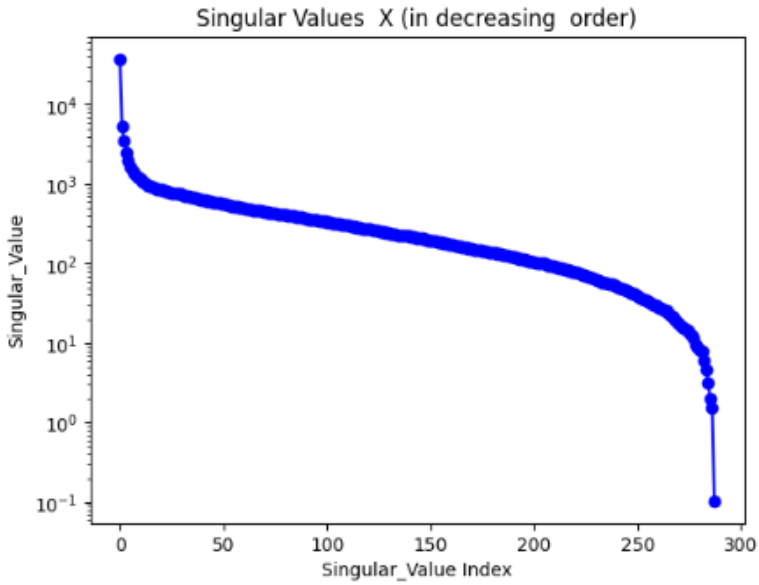
```
#Problem 2.2
import numpy as np
import matplotlib.pyplot as plt

# Perform SVD on the grayscale image matrix X
U, S, V = np.linalg.svd(X, full_matrices=False)

# Plot the singular values in decreasing order
plt.plot(S, marker='o', linestyle='-', color='b')
plt.yscale('log')  # Use a logarithmic scale for better visualization


plt.xlabel('Singular_Value Index')
plt.ylabel('Singular_Value')
plt.title('Singular Values  X (in decreasing  order)')

plt.show()

#Problem 2.3
k = 10
```

```
#  approximation of X using the top k singular values and vectors
X_approximation = U[:, :k] @ np.diag(S[:k]) @ V[:k, :]

# Display the approximation
plt.imshow(X_approximation, cmap='gray')
plt.title(f'Approximation of X with k={k}')
plt.axis('off')

plt.show()

# Calculate the number of elements needed to store the approximation
num_elements_original = X.size
num_elements_approximation = U[:, :k].size + S[:k].size + V[:k, :].size
compression_ratio = num_elements_approximation / num_elements_original

print(f"Number of elements needed to store the approximation:
{num_elements_approximation}")
print(f"Compression ratio: {compression_ratio}")
```

```
#Problem 2.4
k_values = [20, 40, 60]

# Initialize a table to store compression ratios
compression_table = []

# Iterate through various 'k' values
for k in k_values:
    # Reconstruct the approximation of matrix X using the top 'k' singular
values and vectors
    X_approximation = U[:, :k] @ np.diag(S[:k]) @ V[:k, :]

    # Calculate the number of elements required to store the approximation
    num_elements_original = X.size
    num_elements_approximation = U[:, :k].size + S[:k].size + V[:k,
:].size
    compression_ratio = num_elements_approximation / num_elements_original

    # Store the compression ratio in the table
    compression_table.append([k, compression_ratio])
```

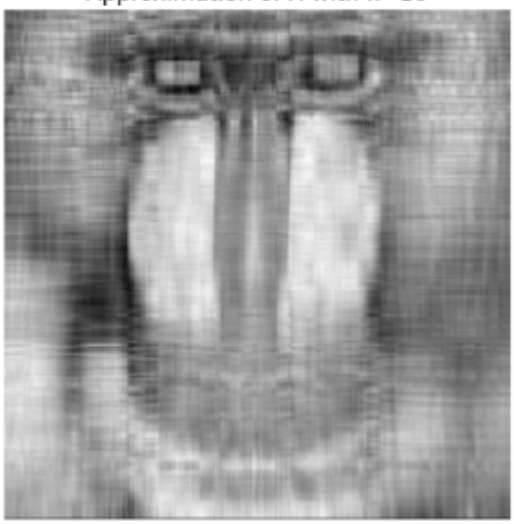
```
    # Display the image approximation
    plt.figure()
    plt.imshow(X_approximation, cmap='gray')
    plt.title(f'Approximation of X with k={k}')
    plt.axis('off')  # Hide axis labels

# Show the images and print the compression ratios
plt.show()

# Create a header for the table
header = ["k", "Compression Ratio"]

# Print the table
print("{:<10} {:<20}".format(*header))
for row in compression_table:
    print("{:<10} {:<20.2f}".format(*row))
```

Output of the code -

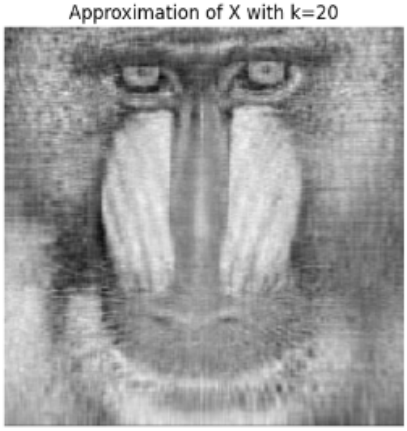



```
Shape of X: (288, 288)
```

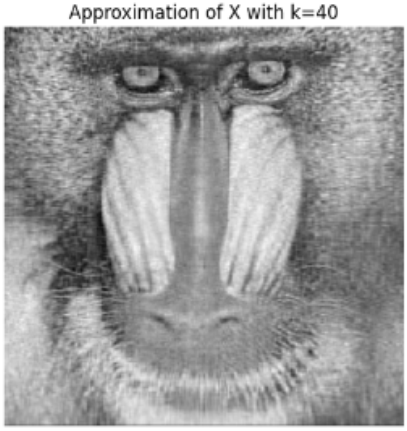
Approximation of X with k=10



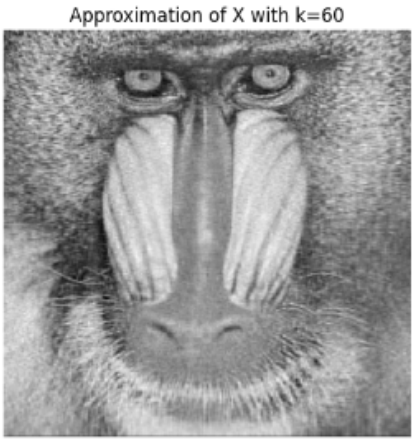
Approximation of X with k=20



Approximation of X with k=40



```
Number of elements needed to store the approximation: 5770
Compression ratio: 0.06956500771604938
```



k	Compression Ratio
20	0.14
40	0.28
60	0.42

Code explanation-

The process includes below steps:

2.1. Input Image and Grayscale Conversion:
- Initially using Image library, we will read the image file 'mandrill_color.png.'
- Then the code converts the color image to grayscale by averaging the R,G,B values for each pixel.
- The resulting grayscale image representation is stored as a matrix X, with dimensions 288 × 288 using numpy.array.

2.2. Singular Value Decomposition (SVD):
-The SVD operation has decomposed our grayscale image matrix X into 3 matrices/ numpy arrays- {U, S, V} each using `np.linalg.svd()` function. The indexing/ordering of singular values is in decreasing order bydefault inside matrix S.
Besides, `full_matrices=False` return compact representations of U, S and V (i.e., without unnecessary zero columns).
-After obtaining the singular values, we plot them on a graph. The x-axis represents order of the singular values, and the y-axis represents the magnitude of each singular value. The singular values are arranged in decreasing order on the x-axis.
`plt.plot function` plots the singular values, I have also used `plt.yscale('log') as logarithmic` has better scaling and visualization.

-I got these first 10 and last 10 singular values in S -

```
[37522.07764716  5358.26449958  3494.85951059  2460.39149317
  2011.89001963  1617.8152772   1469.97658955  1348.04801517
  1276.74376989  1191.23685374]

[9.70065487 8.97236316 8.32791055 7.87259434 5.94742362 4.70048666
 3.17871421 2.03418237 1.52517414 0.10368352]
```

Observation-
The singular values start at the scale of 10^5 and ranging mostly between 10^3 to 10^1. The larger singular values are significant in capturing data structure and generally signify more importance in representing the primary patterns within the data.

2.3. Image Reconstruction for top 10 Singular Values:
-Given k = 10, in this part, we will reconstruct an approximation of original image X using its top k singular values and decomposed vectors (U, S, V). We will achieve this by computing matrix multiplication of (U, S, V).
X_approximation = U[:, :k] @ np.diag(S[:k]) @ V[:k, :]
U[:,:k] represents the first k columns of the left singular vectors matrix.
np.diag(S[:k]) represents a diagonal matrix using the first k singular values from the vector S.
V[:k,:] represents the first k rows of the right singular vectors matrix V.
The @ operator performs matrix multiplication.

- For displaying the approximate image, we are using Matplotlib library functions with grayscale colormap.
- Number of elements needed to store the approximation will be sum sizes of the sliced matrices of (U,S,V) for top 10 singular values i.e., size of (U[:, :k] + S[:k] + V[:k, :]
- For compression ratio for k = 10, we would divide the number we got in above step this by the original size of X.

Compression ratio =Number of elements needed to store the approximation / X.size = 5770/ 82944 = 0.06956500771604938 , means image has been 0.069 times compressed compared to its original size.

2.4 Image Reconstruction for k=20,40,60
- Similarly using the same logic, we will plot the approximate image and calculate compression ratio for various values of 'k' (e.g.20, 40, 60). Images and compression ratio is displayed above forthe same.
- As per the observation, as we increase the value of $k\ (20\rightarrow, 40\rightarrow 60)$, we notice that the resulting image becomes clearer. This is attributed to the fact that with a higher k, more singular values and their corresponding vectors are taken into account during the approximation process. Each singular value captures distinct patterns or features in the data. Consequently, a higher k leads to a more accurate representation of the original image, as it incorporates a greater amount of information.

- Conclusively, based on this observation, we can say that for this particular scenario, increasing the value of k improves the accuracy of the approximation and results in a

clearer representation of the original image. However, in some cases, it's important to keep in mind that the choice of the optimal k involves a trade-off between accuracy and computational/storage requirements.

—----------Probelm 2 End—----------

Problem 3 (20 Points): PCA: Best Places to Live

The *Places Rated Almanac*, written by Boyer and Savageau and published by McNally, rates the livability of several US cities according to nine factors: climate, housing, healthcare, crime, transportation, education, arts, recreation, and economic welfare. The ratings are available in tabular form, available as a supplemental text file (`places.txt`). Except for housing and crime, higher ratings indicate better quality of life.

Let us use PCA to interpret this data better.

1. **2 Points.** Read the data and construct a table with 9 columns containing the numerical ratings. (Ignore the last 5 columns – they consist auxiliary information such as longitude/latitude, state, etc.)
2. **2 Points.** Replace each value in the matrix by its base-10 logarithm. (This pre-processing is done for convenience since the numerical range of the ratings is large.) You should now have a data matrix X whose rows are 9-dimensional vectors representing the different cities.
3. **4 Points.** Perform PCA on the data. Remember to center the data points first by computing the mean data vector μ and subtracting it from every point. With the centered data matrix, do an SVD and compute the principal components.
4. **3 Points.** Write down the first two principal components v_1 and v_2. Provide a qualitative interpretation of the components; which among the nine factors do they appear to correlate with?
5. **3 Points.** Project the data points onto the first two principal components. (That is, compute the highest 2 scores of each of the data points.) Plot the scores as a 2D scatter plot. Which cities correspond to outliers in this scatter plot?
6. **6 Points.** Repeat Steps 2-5, but with a slightly different data matrix – instead of computing the base-10 logarithm, use the normalized z-score of each data point. How do your answers change?

Answer-

Principal Component Analysis -

PCA is a statistical method used in data analysis and machine learning to simplify the complexity in high-dimensional data while retaining trends and patterns. It transforms the original variables into a new set of variables, the principal components, which are linear combinations of the original variables. These principal components are ordered in such a way that the first component captures the maximum variance in the data, the second component the second maximum, and so on.

The main goal of PCA is to reduce the dimensionality of the data while preserving as much of the original variability as possible. This is particularly useful when dealing with datasets with a large number of variables, as it can help in visualizing and interpreting

the data, as well as in reducing noise and computational complexity in subsequent analyses.
The process of PCA involves calculating the covariance matrix of the original data, finding its eigenvectors and eigenvalues, and then using them to transform the data into a new coordinate system. The eigenvectors represent the directions of maximum variance in the data, and the eigenvalues indicate the magnitude of the variance in those directions. The first few principal components, corresponding to the highest eigenvalues, capture the most significant variability in the data.

3.1. Code snippet-

```
# Initialize empty lists to store ratings for each category
cities = []
climate = []
housing = []
healthcare = []
crime = []
transportation = []
education = []
arts = []
recreation = []
economic_welfare = []

# Read data from the file and extract ratings
with open("/content/places.txt", "r") as file:
    for line in file:
        data = line.strip().split()  # Split the line into individual values
        city = data[0]  # The city name is the first value
        rating_values = [int(val) for val in data[1:10]]  # Convert ratings to integers
        cities.append(city)
        climate.append(rating_values[0])
        housing.append(rating_values[1])
        healthcare.append(rating_values[2])
        crime.append(rating_values[3])
        transportation.append(rating_values[4])
        education.append(rating_values[5])
        arts.append(rating_values[6])
        recreation.append(rating_values[7])
        economic_welfare.append(rating_values[8])
```

```
# Create a table with 9 columns of numerical ratings
Ratings_table = {
    "Climate": climate,
    "Housing": housing,
    "Healthcare": healthcare,
    "Crime": crime,
    "Transportation": transportation,
    "Education": education,
    "Arts": arts,
    "Recreation": recreation,
    "Economic Welfare": economic_welfare,
}


df2 = pd.DataFrame(Ratings_table)

# Print the DataFrame
print(df2)
```

Results-

```
     Climate  Housing  Healthcare  Crime  Transportation  Education  Arts  \
0        521     6200         237    923            4031       2757   996
1        575     8138        1656    886            4883       2438  5564
2        468     7339         618    970            2531       2560   237
3        476     7908        1431    610            6883       3399  4655
4        659     8393        1853   1483            6558       3026  4496
..       ...      ...         ...    ...             ...        ...   ...
324      562     8715        1805    680            3643       3299  1784
325      535     6440         317   1106            3731       2491   996
326      540     8371         713    440            2267       2903  1022
327      570     7021        1097    938            3374       2920  2797
328      608     7875         212   1179            2768       2387   122

     Recreation  Economic Welfare
0          1405              7633
1          2632              4350
2           859              5250
3          1617              5864
4          2612              5727
..          ...               ...
324         910              5040
325        2140              4986
326         842              4946
327        1327              3894
328         918              4694

[329 rows x 9 columns]
```

Explanation-

The code reads the data from "places.txt" line by line, splitting each line into values. The first value is considered the city name, and the subsequent values (ratings) are converted to integers and stored in separate lists for each category (e.g., climate, housing).
A dictionary, "Ratings_table," is created with 9 columns only for the numerical ratings which are climate, housing, healthcare, crime, transportation, education, arts, recreation, and economic welfare and is further stored in pandas dataframe ‘df2’.

3.2 Code snippet-

```
import pandas as pd
import numpy as np

# 1. Read the data and construct a table with 9 columns
data = df2.iloc[:,:9]

# 2. Replace each value in the matrix by its base-10 logarithm
def safe_log(x):
    try:
        return np.log10(x)
    except:
        return np.nan  # Replace invalid values with NaN

data_log = data.applymap(safe_log)

# data_log now contains the DataFrame with values replaced by their base-10 logarithms
data_log
```

-The code reads the data of 9 columns stored in df2.
-For replacing each value in the matrix by its base-10 logarithm, I defined a function safe_log(x) that takes a value x and calculates its base-10 logarithm using np.log10(x) (NumPy's logarithm function). If the calculation fails (e.g., if x is non-positive), it returns np.nan (NaN, or Not a Number).
- Then we are applying this safe_log function to every element in the DataFrame data sing data.applymap(safe_log) .
-The result is a new DataFrame named data_log, where each original value has been replaced by its base-10 logarithm (or NaN if the logarithm couldn't be calculated).

3.3.-

Code snippet for PCA nd SVD-

```
from sklearn.decomposition import PCA

# Step 1: Center the data by subtracting the mean from each column
# Calculate the mean for each column
mean_data = data_log.mean()
# Center the data by subtracting the mean
centered_data = data_log - mean_data

# Step 2: Calculate the covariance matrix
cov_matrix = centered_data.cov()

# Step 3: Perform Singular Value Decomposition (SVD) on the covariance matrix
# SVD decomposes the covariance matrix into three parts:
# U: Principal components (eigenvectors)
# S: Singular values (eigenvalues)
# Vt: Transpose of the right singular matrix
U, S, Vt = np.linalg.svd(cov_matrix)

# Step 4: Extract the explained variance and the cumulative explained variance
# The explained variance is the ratio of each eigenvalue (S) to the sum of all eigenvalues
explained_variance = S / np.sum(S)
# Cumulative explained variance is the cumulative sum of explained variance
cumulative_explained_variance = np.cumsum(explained_variance)

# Step 5: Determine the number of components to keep (e.g., based on explained variance)
# For example, we can keep enough components to retain 90% of the variance
n_components = np.argmax(cumulative_explained_variance >= 0.9) + 1

# Step 6: Fit a PCA model with the desired number of components
pca = PCA(n_components=n_components)
# Transform the centered data into principal components
principal_components = pca.fit_transform(centered_data)
```

```
# Step 6: Fit a PCA model with the desired number of components
pca = PCA(n_components=n_components)
# Transform the centered data into principal components
principal_components = pca.fit_transform(centered_data)

# Print the outputs for analysis
print("Centered Data:")
print(centered_data)
print("\nCovariance Matrix:")
print(cov_matrix)
print("\nPrincipal Components (U):")
print(U)
print("\nSingular Values (S):")
print(S)
print("\nCumulative Explained Variance:")
print(cumulative_explained_variance)
print("\nNumber of Components to Keep (for 90% variance):", n_components)
print("\nPrincipal Components (Reduced):")
print(principal_components)
```

Results-

Centered Data:

	Climate	Housing	Healthcare	Crime	Transportation	Education	Arts	Recreation	Economic Welfare
0	-0.001656	-0.115099	-0.580738	0.013575	0.008211	-0.006170	-0.207950	-0.078891	0.148513
1	0.041174	0.003027	0.263574	-0.004193	0.091485	-0.059573	0.539178	0.193719	-0.095693
2	-0.048248	-0.041854	-0.164498	0.035145	-0.193910	-0.038367	-0.831461	-0.292574	-0.014023
3	-0.040887	-0.009424	0.200153	-0.166297	0.240576	0.084745	0.461710	-0.017857	0.034012
4	0.100391	0.016426	0.312389	0.219514	0.219570	0.034262	0.446617	0.190406	0.023745
..	...	...	...	...	...	...	...	...	...
324	0.031242	0.032776	0.300991	-0.119118	-0.035743	0.071776	0.045185	-0.267526	-0.031752
325	0.009860	-0.098605	-0.454427	0.092128	-0.025376	-0.050233	-0.207950	0.103846	-0.036430
326	0.013900	0.015286	-0.102397	-0.308174	-0.241750	0.016240	-0.196759	-0.301255	-0.039928
327	0.037381	-0.061092	0.084720	0.020576	-0.069057	0.018776	0.240483	-0.103696	-0.143786
328	0.065409	-0.011241	-0.629151	0.119887	-0.155036	-0.068754	-1.119850	-0.263725	-0.062639

[329 rows x 9 columns]

```
Covariance Matrix:
                  Climate   Housing  Healthcare     Crime  Transportation  \
Climate          0.012892  0.003268    0.005479  0.004374        0.000386
Housing          0.003268  0.011116    0.014596  0.002483        0.005279
Healthcare       0.005479  0.014596    0.102728  0.009955        0.021153
Crime            0.004374  0.002483    0.009955  0.028611        0.007299
Transportation   0.000386  0.005279    0.021153  0.007299        0.024829
Education        0.000442  0.001070    0.007478  0.000471        0.002462
Arts             0.010689  0.029226    0.118484  0.031947        0.047041
Recreation       0.002573  0.009127    0.015299  0.009285        0.011567
Economic Welfare -0.000966  0.002646    0.001463  0.003946        0.000834

                  Education      Arts  Recreation  Economic Welfare
Climate            0.000442  0.010689    0.002573         -0.000966
Housing            0.001070  0.029226    0.009127          0.002646
Healthcare         0.007478  0.118484    0.015299          0.001463
Crime              0.000471  0.031947    0.009285          0.003946
Transportation     0.002462  0.047041    0.011567          0.000834
Education          0.002520  0.009520    0.000877          0.000546
Arts               0.009520  0.297173    0.050860          0.006206
Recreation         0.000877  0.050860    0.035308          0.002792
Economic Welfare   0.000546  0.006206    0.002792          0.007137

Principal Components (U):
[[-0.03507288  0.0088782  -0.14087477  0.15274476 -0.39751159 -0.83129501
  -0.0559096   0.31490125 -0.06448925]
 [-0.09335159  0.00923057 -0.12884967 -0.17838233 -0.1753133  -0.20905725
   0.6958923  -0.61361583  0.08687702]
 [-0.40776448 -0.85853187 -0.27605769 -0.03516139 -0.05032469  0.08967085
  -0.06245284  0.0210358  -0.06550333]
 [-0.10044536  0.22042372 -0.5926882   0.72366303  0.01345714  0.16401885
  -0.05553037 -0.1823479   0.05421223]
 [-0.15009714  0.05920111 -0.22089816 -0.12620531  0.86996951 -0.37244964
   0.0724604   0.05714199 -0.07183942]
 [-0.03215319 -0.06058858 -0.0081447  -0.00519693  0.04779772 -0.02362804
   0.05738567  0.20447312  0.97327107]
 [-0.87434057  0.30380632  0.36328732  0.08111571 -0.05506994  0.02812147
  -0.0232698   0.01673991 -0.00525656]
 [-0.15899622  0.33399255 -0.58362605 -0.62822609 -0.21328989  0.14179906
```

```
Principal Components (U):
[[-0.03507288  0.0088782  -0.14087477  0.15274476 -0.39751159 -0.83129501
  -0.0559096   0.31490125 -0.06448925]
 [-0.09335159  0.00923057 -0.12884967 -0.17838233 -0.1753133  -0.20905725
   0.6958923  -0.61361583  0.08687702]
 [-0.40776448 -0.85853187 -0.27605769 -0.03516139 -0.05032469  0.08967085
  -0.06245284  0.0210358  -0.06550333]
 [-0.10044536  0.22042372 -0.5926882   0.72366303  0.01345714  0.16401885
  -0.05553037 -0.1823479   0.05421223]
 [-0.15009714  0.05920111 -0.22089816 -0.12620531  0.86996951 -0.37244964
   0.0724604   0.05714199 -0.07183942]
 [-0.03215319 -0.06058858 -0.0081447  -0.00519693  0.04779772 -0.02362804
   0.05738567  0.20447312  0.97327107]
 [-0.87434057  0.30380632  0.36328732  0.08111571 -0.05506994  0.02812147
  -0.0232698   0.01673991 -0.00525656]
 [-0.15899622  0.33399255 -0.58362605 -0.62822609 -0.21328989  0.14179906
  -0.23451524  0.08353911  0.01749472]
 [-0.01949418  0.0561011  -0.12085337  0.05216997 -0.02965242  0.26481279
   0.66448592  0.66203179 -0.16826376]]

Singular Values (S):
[0.37746236 0.05105221 0.02791958 0.02296708 0.01677125 0.01195269
 0.0084567  0.00393422 0.00179733]

Cumulative Explained Variance:
[0.72267403 0.82041652 0.87387021 0.91784205 0.94995161 0.97283574
 0.9890266  0.9965589  1.        ]

Number of Components to Keep (for 90% variance): 4

Principal Components (Reduced):
[[-0.43667707  0.42016341 -0.11812095 -0.0899588 ]
 [ 0.6209576   0.0053458   0.00180816  0.10074486]
 [-0.87325632 -0.21210364  0.04969289 -0.17161166]
 ...
 [-0.32726868 -0.1576827  -0.36942695  0.01837878]
 [ 0.21343283 -0.04329752 -0.14738603 -0.11428668]
 [-1.2910218   0.13022263  0.11567313 -0.21237254]]
```

Here are the steps involved in the code-

- To center the data points, we will calculate the mean vector (mean_data) for each column in the data_log DataFrame.
- Then we have centered the data by subtracting the mean vector from each data point in the data_log DataFrame, resulting in the centered_data DataFrame.

-Next, to compute SVD, we first Compute the covariance matrix (cov_matrix) of the centered data. The covariance matrix represents the relationships between different dimensions of the centered data. It's always good to calculate the covariance matrix before performing Singular Value Decomposition (SVD) when doing Principal Component Analysis (PCA) because the principal components are the eigenvectors of the covariance matrix. It also Normalizes data and provides better visualization.

-Then, I have applied (SVD) to the covariance matrix (cov_matrix) using the np.linalg.svd() function. The result is three matrices: U contains the principal

components (eigenvectors), S contains the singular values (eigenvalues), and Vt is the transpose of the right singular matrix.

-Computes the explained variance for each principal component by dividing each eigenvalue (S) by the sum of all eigenvalues.explained_variance shows how much each principal component contributes to total variance.

-Calculates the cumulative explained variance by taking the cumulative sum of the explained variance.cumulative_explained_variance indicates the cumulative contribution, aiding in choosing the number of retained components for dimensionality reduction.
- I have selected the number of principal components such that 90% of the total variance is captured.
The reason for choosing 90% of the data is that if we retain 100% of the variance, we are keeping all the original dimensions of the data without reducing its complexity. The goal of dimensionality reduction methods PCA is to capture the essential information in fewer dimensions, getting improved computational efficiency and clearer interpretation, choosing a lower percentage, like 90%, strikes a balance between reducing complexity and retaining a substantial portion of the information in the data.
-Finally , we initialized a PCA model with our desired number of components and fitted the PCA model to the centered_data and transformed the data into principal components (principal_components).
-Output has -the centered data, -covariance matrix, -principal components ('U' eigenvectors), singular values ('S' eigenvalues), -cumulative explained variance, the number of components to retain a certain percentage of the variance, and the reduced principal components after PCA transformation.

Observation-
Original components are 9-dimensional; reduced components are 4-dimensional. Reduced components capture the most significant information, retaining essential patterns while reducing dimensionality.

3.4. Principal Components
Code -

```
# Assuming you have already performed the PCA analysis as described in the previous response.

# Get the first two principal components (v1 and v2)
v1 = pca.components_[0]
v2 = pca.components_[1]

# Print the first two principal components
print("First Principal Component (v1):", v1)
print("Second Principal Component (v2):", v2)
```

Output-

```
First Principal Component (v1): [0.03507288 0.09335159 0.40776448 0.10044536 0.15009714 0.03215319
 0.87434057 0.15899622 0.01949418]
Second Principal Component (v2): [ 0.0088782   0.00923057 -0.85853187  0.22042372  0.05920111 -0.06058858
  0.30380632  0.33399255  0.0561011 ]
```

Explanation-

This code retrieves the first two principal components (v1 and v2) obtained from a PCA analysis which is simply the first two columns of `Principal Components (U)`. The output represents the direction and magnitude of these components in the original feature space.
Output Explanation:
pca.components_ array contains all the principal components. Each row represents a principal component, and each column corresponds to an original feature.
v1 and v2 are the coefficients of the original features in the direction of the first and second principal components.They indicate the contribution of each feature to these principal components and provide insight into the dominant directions in the data.
For example, in v1, features with larger absolute values contribute more to the first principal component.The coefficients indicate the influence of each feature on these components.These components can be used to understand the major patterns captured by the PCA analysis.

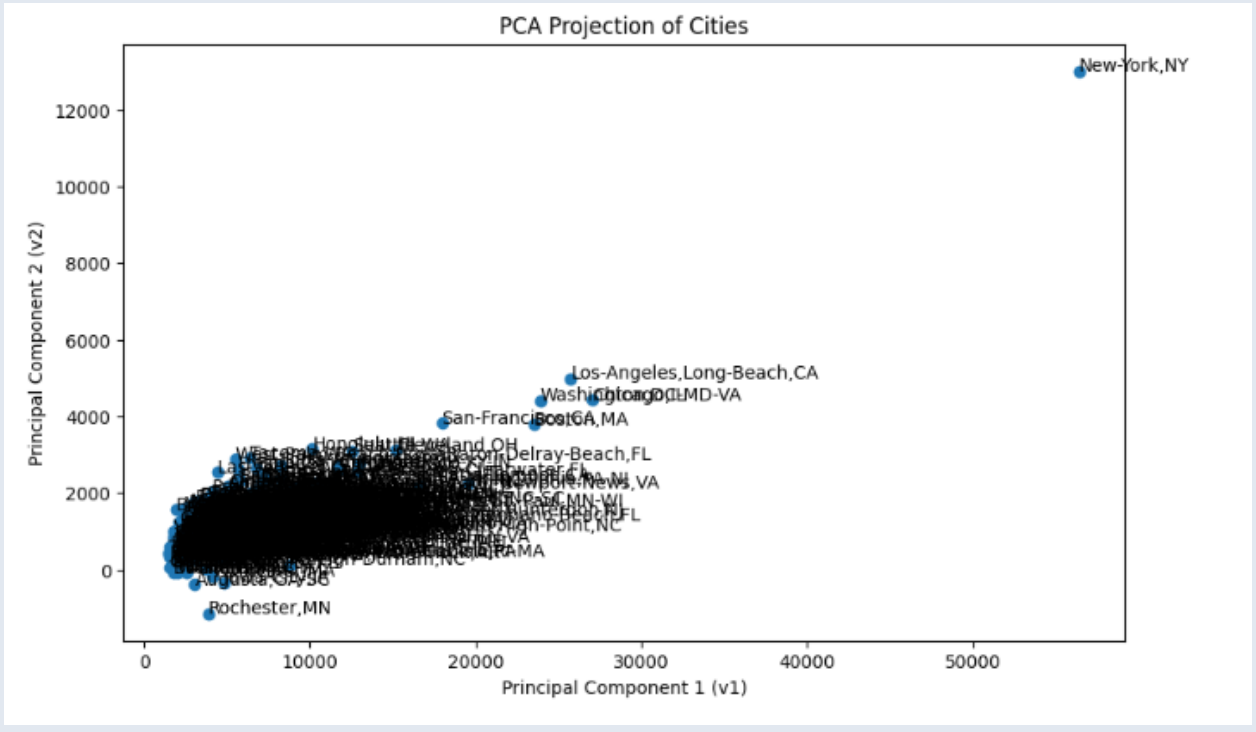
Factor	v1 Loading	v2 Loading	Interpretation / association
Arts	0.874	0.304	Strong positive with both v1 and v2
Healthcare	0.408	-0.859	Strong positive with v1, strong negative with v2
Recreation	0.159	0.334	Strong positive with both v1 and v2
Transportation	0.15	0.059	Positive with v1, weak with v2
Crime	0.1	0.22	Positive with both v1 and v2
Housing	0.093	0.009	Positive with v1, weak with v2
Climate	0.035	0.009	Positive with v1, weak with v2
Education	0.032	-0.061	Positive with v1, weak negative with v2

Economic Welfare	0.019	0.056	Positive with v1, weak with v2

First Principal Component (v1)
The first principal component (v1) explains the most variance in the data and is therefore the most important component. It has high positive loadings for Arts,Healthcare,Recreation,Transportation and low positive loadings for rest of the factors which means v1 reflects “city well being” and tends to be cities with a strong cultural scene, good healthcare, good recreational opportunities, good transportation systems.
Second Principal Component (v2)
The second principal component (v2) explains the second most variance in the data. It has a negative loading for Healthcare , Education and positive loadings for Recreation, Arts, Crime, and Education. This suggests that v2 captures a dimension of "city balance" or "city diversity."
Cities with high values of v2 tend to have strong cultural scenes, good recreational opportunities, and some crime rates at the same time, but they may also have relatively high healthcare costs and education costs.
Overall, the two principal components provide a useful way to summarize the complex relationships between the different factors that contribute to city quality or well-being.

3.5 2D scatter plot

```
# Project the data points onto the first two principal components
scores = np.dot(df.iloc[:, 1:].to_numpy(), v1), np.dot(df.iloc[:, 1:].to_numpy(), v2)

# Create a scatter plot of the scores
plt.figure(figsize=(10, 6))
plt.scatter(scores[0], scores[1])

# Label the data points with city names from the DataFrame 'df'
for i, city in enumerate(df["City"]):
    plt.annotate(city, (scores[0][i], scores[1][i]))

# Set axis labels and title
plt.xlabel('Principal Component 1 (v1)')
plt.ylabel('Principal Component 2 (v2)')
plt.title('PCA Projection of Cities')

# Display the plot
plt.show()
```



Code explanation-
Here, we are calculating projected scores for each data point along the two components and plotting them to visualize the results of PCA. The code projects the data points onto the first two principal components (v1 and v2). The dot product of each data point with the principal components is computed to obtain the scores along these components. These scores represent the coordinates of each data point in the reduced-dimensional space defined by the principal components.

Observing the plot, New York is the biggest outlier in this projection of the cities dataset, followed by smaller outliers like Los Angeles, Washington, San Francisco, Boston, and Rochester.

6. **6 Points.** Repeat Steps 2-5, but with a slightly different data matrix – instead of computing the base-10 logarithm, use the normalized z-score of each data point. How do your answers change?

Answer-
With z-scores, the data points are centered around the origin (mean of 0) and scaled to have unit variance. This concentrates most points close to the y=0 line. With log-transformed data, the data points are more spread out since it compresses the scale for higher values and expands it for lower values.
Code snippets-

```
# 1. Read the data and construct a table with 9 columns
data = df2.iloc[:, :9]

# 2. Replace each value in the matrix by its normalized z-score
scaler = StandardScaler()
data_standardized = scaler.fit_transform(data)

# Convert the standardized data back to a DataFrame
data_standardized_df = pd.DataFrame(data_standardized, columns=data.columns)

# data_standardized_df now contains the DataFrame with values replaced by their normalized z-scores
data_standardized_df
```

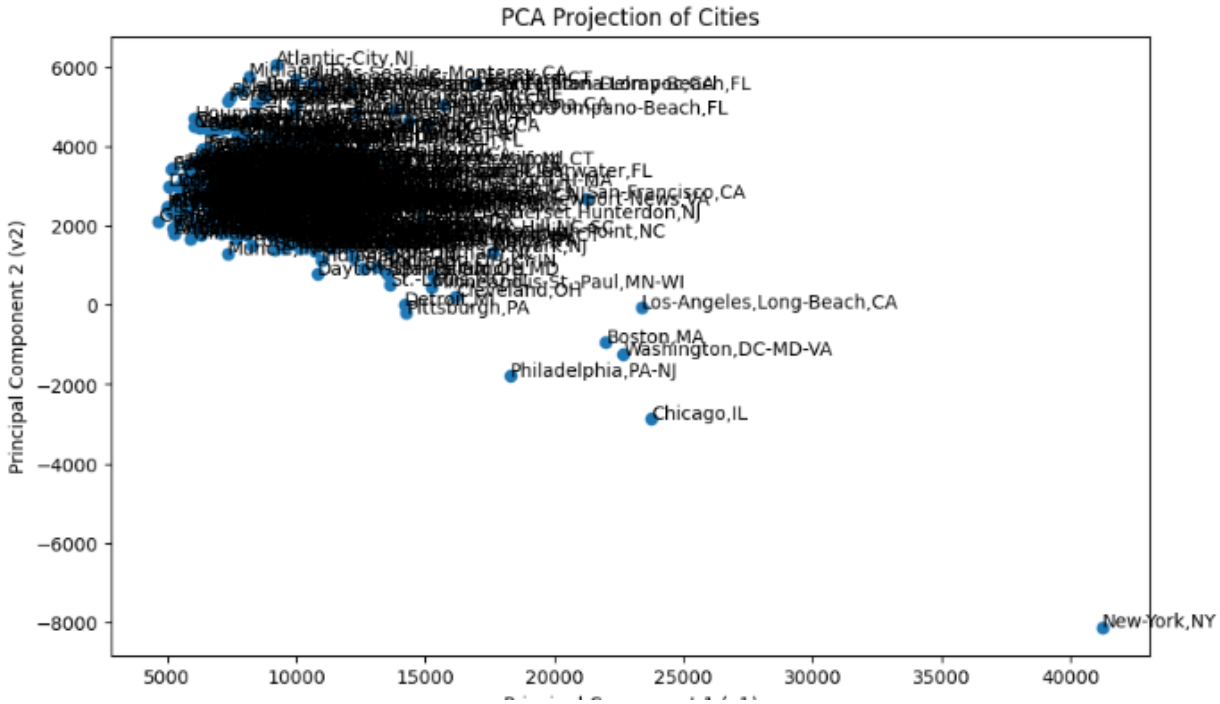
```
# Step 1: Center the data by subtracting the mean from each column
# Calculate the mean for each column
mean_data = data_standardized_df.mean()
# Center the data by subtracting the mean
centered_data = data_standardized_df - mean_data

# Step 2: Calculate the covariance matrix
cov_matrix = centered_data.cov()

# Step 3: Perform Singular Value Decomposition (SVD) on the covariance matrix
U, S, Vt = np.linalg.svd(cov_matrix)

# Step 4: Extract the explained variance and the cumulative explained variance
# The explained variance is the ratio of each eigenvalue (S) to the sum of all eigenvalues
explained_variance = S / np.sum(S)
# Cumulative explained variance is the cumulative sum of explained variance
cumulative_explained_variance = np.cumsum(explained_variance)

# Step 5: Determine the number of components to keep (e.g., based on explained variance)
# For example, we can keep enough components to retain 90% of the variance
n_components = np.argmax(cumulative_explained_variance >= 0.9) + 1

# Step 6: Fit a PCA model with the desired number of components
pca = PCA(n_components=n_components)
# Transform the centered data into principal components
principal_components = pca.fit_transform(centered_data)

# Print the outputs for analysis
print("\nPrincipal Components (U):")
print(U)
print("\nNumber of Components to Keep (for 90% variance):", n_components)
print("\nPrincipal Components (Reduced):")
print(principal_components)
```

```python
# Assuming you have already performed the PCA analysis as described in the previous response.

# Get the first two principal components (v1 and v2)
v1 = pca.components_[0]
v2 = pca.components_[1]

# Print the first two principal components
print("First Principal Component (v1):", v1)
print("Second Principal Component (v2):", v2)
```

```
First Principal Component (v1): [0.20641395 0.35652161 0.46021465 0.28129838 0.35115078 0.27529264
 0.46305449 0.32788791 0.13541225]
Second Principal Component (v2): [ 0.21783531  0.250624   -0.29946528  0.35534227 -0.17960448 -0.48338209
 -0.19478992  0.38447464  0.47128328]
```

```python
# Project the data points onto the first two principal components
scores = np.dot(df.iloc[:, 1:].to_numpy(), v1), np.dot(df.iloc[:, 1:].to_numpy(), v2)

# Create a scatter plot of the scores
plt.figure(figsize=(10, 6))
plt.scatter(scores[0], scores[1])

# Label the data points with city names from the DataFrame 'df'
for i, city in enumerate(df["City"]):
    plt.annotate(city, (scores[0][i], scores[1][i]))

# Set axis labels and title
plt.xlabel('Principal Component 1 (v1)')
plt.ylabel('Principal Component 2 (v2)')
plt.title('PCA Projection of Cities')

# Display the plot
plt.show()
```

The scatter plot depicts the direction has changed , but the relative scattering and the overall outliers remain the same.



Comparison table-

Factor	z scalar v1	Log 10 v1	z scalar v2	Log 10 v2
Arts	0.463	0.874	-0.195	0.304
Healthcare	0.46	0.408	-0.299	-0.859
Housing	0.357	0.159	0.251	0.334
Transportation	0.351	0.15	-0.18	0.059
Recreation	0.328	0.1	0.384	0.22
Crime	0.281	0.093	0.355	0.009
Education	0.275	0.035	-0.483	0.009
Climate	0.206	0.032	0.218	-0.061
Economic Welfare	0.135	0.019	0.471	0.056

Changes observed-

The values of Principal Component $v1$ for the nine factors show that, with z-scale transformation, the magnitudes change, but the relative ordering of the factors remains the same. However, for $v2$, there are contrasting loadings, particularly for "Arts," "Transportation," "Education," and "Climate."

—----------Problem 3 End—----------

Problem 4 (20 Points): Manifold Learning: Order the Faces

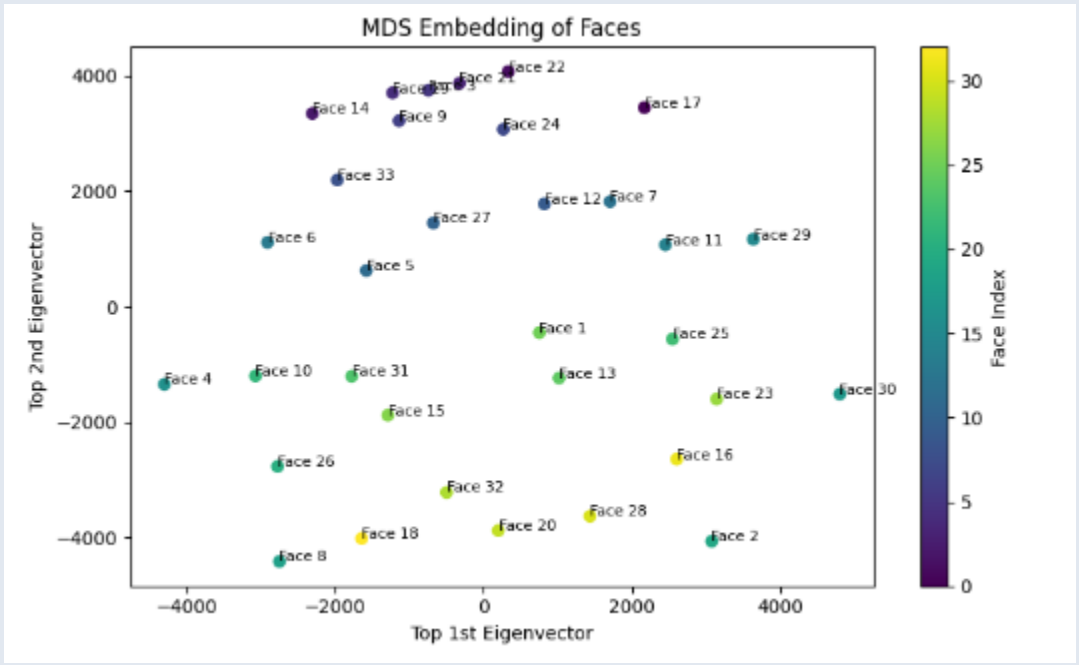
The dataset (`face.mat`) contains 33 faces of the same person ($Y \in \mathbb{R}^{112\times92\times33}$) in different angles. You may create a data matrix $X \in \mathbb{R}^{n\times p}$, where $n = 33, p = 112 \times 92 = 10304$ (e.g., `X=reshape(Y,[10304,33])'`; in MATLAB).

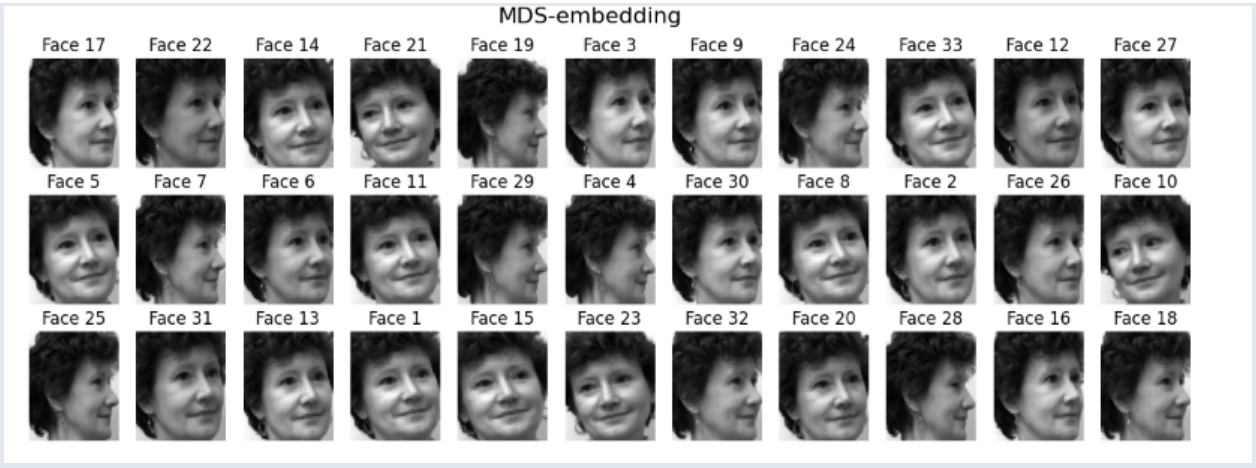
1. **5 Points.** Explore the MDS-embedding of the 33 faces on top two eigenvectors: order the faces according to the top 1st eigenvector and visualize your results with figures.
2. **5 Points.** Explore the ISOMAP-embedding of the 33 faces on the $k = 5$ nearest neighbor graph and compare it against the MDS results. Note: you may try Tenenbaum's Matlab code (`isomapII.m`).
3. **5 Points.** Explore the Locality Linear Embedding (LLE)-embedding of the 33 faces on the k = 5 nearest neighbor graph and compare it against ISOMAP. Note: you may try the following Matlab code (`lle.m`).
4. **5 Points.** Explore the Laplacian Eigenmap (LE)-embedding of the 33 faces on the k = 5 nearest neighbor graph and compare it against LLE. Note: you may try the following Matlab code (`le.m`).

4.1. Code -

```
from sklearn.manifold import MDS

# Set a random seed for reproducibility
np.random.seed(42)
# Load the face dataset from the 'face.mat' file
data = scipy.io.loadmat('/content/face.mat')
Y = data['Y']

# Reshape the data matrix
X = np.reshape(Y, (33, -1))

# Compute MDS embedding
mds = MDS(n_components=2)
mds_result = mds.fit_transform(X)

# Order faces based on the top 1st eigenvector of MDS
sorted_mds_indices = np.argsort(mds_result[:, 0])

# Plot MDS and ISOMAP embeddings side by side
plt.figure(figsize=(15, 5))

# Plot MDS embedding
plt.subplot(1, 2, 1)
plt.scatter(mds_result[:, 0], mds_result[:, 1], c=np.arange(33), cmap='viridis', marker='o')
plt.title('MDS Embedding of Faces')
plt.xlabel('Top 1st Eigenvector')
plt.ylabel('Top 2nd Eigenvector')
plt.colorbar(label='Face Index')
for i in range(33):
    plt.text(mds_result[i, 0], mds_result[i, 1], f'Face {sorted_mds_indices[i] + 1}', fontsize=8)

plt.tight_layout()
plt.show()

# Plot the faces for MDS embedding
plt.figure(figsize=(15, 5))
for i in range(33):
    plt.subplot(3, 11, i + 1)
    plt.imshow(Y[:, :, sorted_mds_indices[i]], cmap='gray')
    plt.axis('off')
    plt.title(f'Face {sorted_mds_indices[i] + 1}')
plt.suptitle('MDS-embedding', fontsize=16)
plt.show()
```

Output scatter plot -





Explanation-

MDS embedding is a technique for reducing the dimensionality of data while preserving the pairwise distances between data points as much as possible. In simpler terms, it's like taking a high-dimensional map and projecting it onto a lower-dimensional map, such as from 3D to 2D, while trying to maintain the relative distances between the points on the map.

The 33 faces represented by the dataset are being analyzed to understand their relationships in a lower-dimensional space. The MDS-embedding will map the 33 faces onto a two-dimensional space, allowing for visualization of their relative positions based on their similarities or differences.

Eigenvectors are special directions in a dataset that represent the directions of maximum variance. I used the sklearn.manifold.MDS module which uses the classical MDS algorithm. It basically finds the pairwise distances of data points using Euclidean distance. Then it constructs a dissimilarity matrix which is a symmetric matrix where each element is the distance between two data points.
Then it computes the double-centered dissimilarity matrix by subtracting the row means and column means of the centered dissimilarity matrix from each element and then find the eigenvectors and eigenvalues of the matrix.
Then The top eigenvectors which are the principal components of the data, and they represent the directions of greatest variance in the data. This results in a low-dimensional representation of the data where the pairwise distances are preserved as much as possible.

For ordering the faces according to the top 1st eigenvector means sorting the faces based on their projection onto the top 1st eigenvector. The projection of a point onto an eigenvector is its coordinate along that direction. By sorting the faces based on their projections, we are essentially arranging them in order of their similarity or dissimilarity along the direction of greatest variance in the data.
In the context of the face dataset, ordering the faces according to the top 1st eigenvector would reveal a progression of facial poses, from one extreme to the other. This could be useful for understanding how the facial features change as the face rotates or tilts.

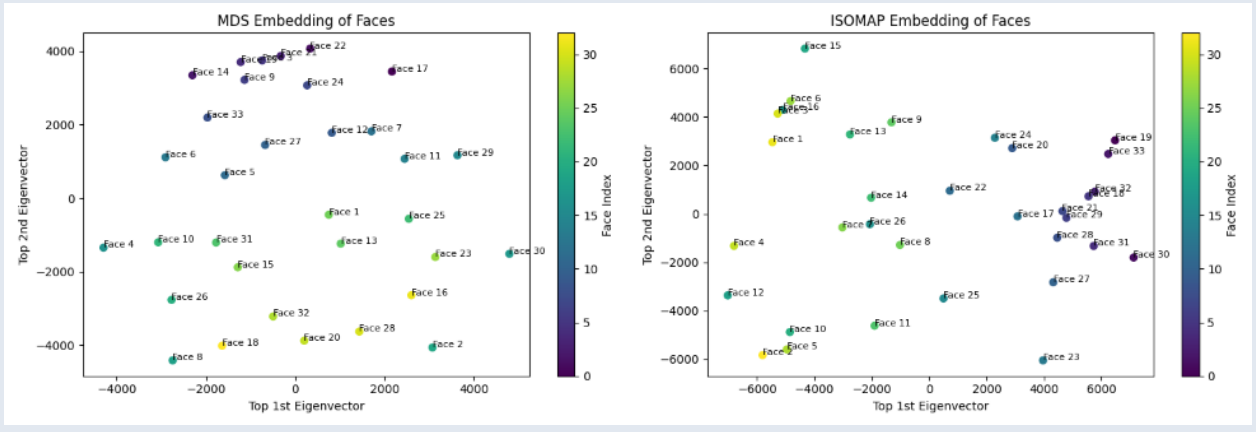
Observation- From the scatter plot and the 33 facial orders , we can tell that faces are clustered together based on their similarity and are labeled with different colors. Post faces are likely to be ordered from looking frontal side to looking sideways among cluster wise. the 1st eigenvector captures the variation in the face images that is most associated with variation that is looking aside.

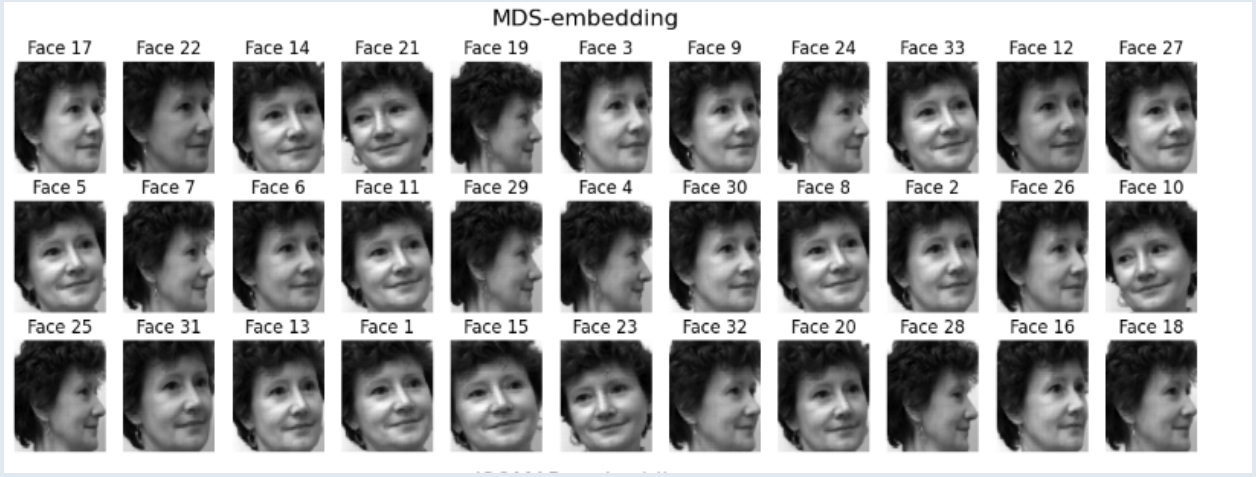
4.2
Code -

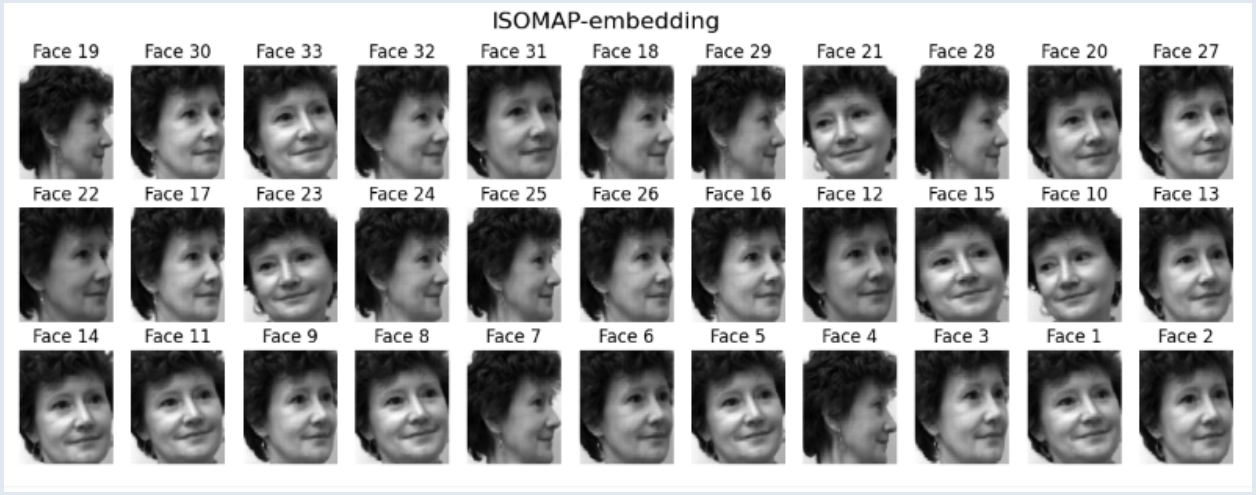
```
from sklearn.manifold import MDS, Isomap
np.random.seed(42)
# Compute ISOMAP embedding for comparison
isomap = Isomap(n_neighbors=5, n_components=2)
isomap_result = isomap.fit_transform(X)

# Order faces based on the top 1st eigenvector of ISOMAP
sorted_isomap_indices = np.argsort(isomap_result[:, 0])

# Plot MDS and ISOMAP embeddings side by side
plt.figure(figsize=(15, 5))

# Plot MDS embedding
plt.subplot(1, 2, 1)
plt.scatter(mds_result[:, 0], mds_result[:, 1], c=np.arange(33), cmap='viridis', marker='o')
plt.title('MDS Embedding of Faces')
plt.xlabel('Top 1st Eigenvector')
plt.ylabel('Top 2nd Eigenvector')
plt.colorbar(label='Face Index')
for i in range(33):
    plt.text(mds_result[i, 0], mds_result[i, 1], f'Face {sorted_mds_indices[i] + 1}', fontsize=8)

# Plot ISOMAP embedding for comparison
plt.subplot(1, 2, 2)
plt.scatter(isomap_result[:, 0], isomap_result[:, 1], c=np.arange(33), cmap='viridis', marker='o')
plt.title('ISOMAP Embedding of Faces')
plt.xlabel('Top 1st Eigenvector')
plt.ylabel('Top 2nd Eigenvector')
plt.colorbar(label='Face Index')
for i in range(33):
    plt.text(isomap_result[i, 0], isomap_result[i, 1], f'Face {sorted_isomap_indices[i] + 1}', fontsize=8)

plt.tight_layout()
plt.show()

# Plot the faces for MDS embedding
plt.figure(figsize=(15, 5))
for i in range(33):
    plt.subplot(3, 11, i + 1)
    plt.imshow(Y[:, :, sorted_mds_indices[i]], cmap='gray')
    plt.axis('off')
    plt.title(f'Face {sorted_mds_indices[i] + 1}')
plt.suptitle('MDS-embedding', fontsize=16)
plt.show()
```

```
# Plot the faces for ISOMAP embedding
plt.figure(figsize=(15, 5))
for i in range(33):
    plt.subplot(3, 11, i + 1)
    plt.imshow(Y[:, :, sorted_isomap_indices[i]], cmap='gray')
    plt.axis('off')
    plt.title(f'Face {sorted_isomap_indices[i] + 1}')
plt.suptitle('ISOMAP-embedding', fontsize=16)
plt.show()
```

Code output-







Explanation-

Isomap aims to maintain the shortest paths along curved surfaces when measuring distances in data. It is capable of capturing a broader range of nonlinear manifold structures compared to MDS. Additionally, Isomap produces a unified global representation and coordinate system for the data.

This code orders the face images based on their similarity in terms of the overall structure or expression .

isomap = Isomap(n_neighbors=5, n_components=2)
isomap_result = isomap.fit_transform(X)
In above lines of code- number of nearest neighbors to consider for each data point is 5, the number of dimensions for the reduced representation is 2, isomap.fit_transform(X) fits the Isomap model to the high-dimensional data X and transforms it into 2D.
np.argsort(isomap_result[:, 0]): sorts the first column (corresponding to the first eigenvector) of the isomap_result array and returns the corresponding indices of the faces.

Observation-

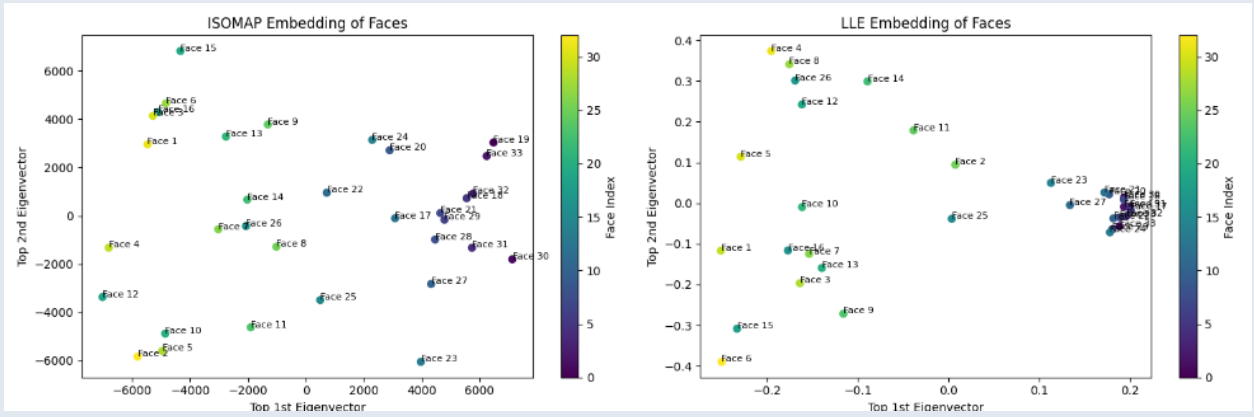
Isomap has given better results in grouping similar images. However, the indices of certain faces belonging to the same groups or labels appear widely scattered (face 1,2 and 3) in the scatter plot. Perhaps, trying with different k values for k-nearest neighbors (knn) could solve this.

4.3
Code snippet-

```
from sklearn.manifold import Isomap, LocallyLinearEmbedding
np.random.seed(42)
# Load the face dataset from the 'face.mat' file
data = scipy.io.loadmat('/content/face.mat')
Y = data['Y']
X = np.reshape(Y, (33, -1))

# Compute LLE embedding for comparison
lle = LocallyLinearEmbedding(n_neighbors=5, n_components=2)
lle_result = lle.fit_transform(X)

# Order faces based on the top 1st eigenvector of LLE for comparison
sorted_lle_indices = np.argsort(lle_result[:, 0])

# Plot ISOMAP and LLE embeddings side by side
plt.figure(figsize=(15, 5))

# Plot ISOMAP embedding
plt.subplot(1, 2, 1)
plt.scatter(isomap_result[:, 0], isomap_result[:, 1], c=np.arange(33), cmap='viridis', marker='o')
plt.title('ISOMAP Embedding of Faces')
plt.xlabel('Top 1st Eigenvector')
plt.ylabel('Top 2nd Eigenvector')
plt.colorbar(label='Face Index')
for i in range(33):
    plt.text(isomap_result[i, 0], isomap_result[i, 1], f'Face {sorted_isomap_indices[i] + 1}', fontsize=8)

# Plot LLE embedding for comparison
plt.subplot(1, 2, 2)
plt.scatter(lle_result[:, 0], lle_result[:, 1], c=np.arange(33), cmap='viridis', marker='o')
plt.title('LLE Embedding of Faces')
plt.xlabel('Top 1st Eigenvector')
plt.ylabel('Top 2nd Eigenvector')
plt.colorbar(label='Face Index')
for i in range(33):
    plt.text(lle_result[i, 0], lle_result[i, 1], f'Face {sorted_lle_indices[i] + 1}', fontsize=8)

plt.tight_layout()
plt.show()

# Plot the faces for ISOMAP embedding
plt.figure(figsize=(15, 5))
for i in range(33):
    plt.subplot(3, 11, i + 1)
    plt.imshow(Y[:, :, sorted_isomap_indices[i]], cmap='gray')
    plt.axis('off')
    plt.title(f'Face {sorted_isomap_indices[i] + 1}')
```

```
# Plot the faces for LLE embedding
plt.figure(figsize=(15, 5))
for i in range(33):
    plt.subplot(3, 11, i + 1)
    plt.imshow(Y[:, :, sorted_lle_indices[i]], cmap='gray')
    plt.axis('off')
    plt.title(f'Face {sorted_lle_indices[i] + 1}')
plt.suptitle('LLE-embedding', fontsize=16)
plt.show()
```

Output-



Explanation-

The LLE(Local Linear Embedding) algorithm is a nonlinear dimensionality reduction technique that aims to preserve the local neighborhood structure of the data. LLE reconstructs data points using local linear fits with their neighbors, avoiding the necessity to estimate distances between widely separated points. It recovers the global nonlinear structure through these local linear fits and maps inputs into a unified low-dimensional coordinate system.

The code implementation is similar to ISOMAP, here I have used LocallyLinearEmbedding package from sklearn.manifold module.

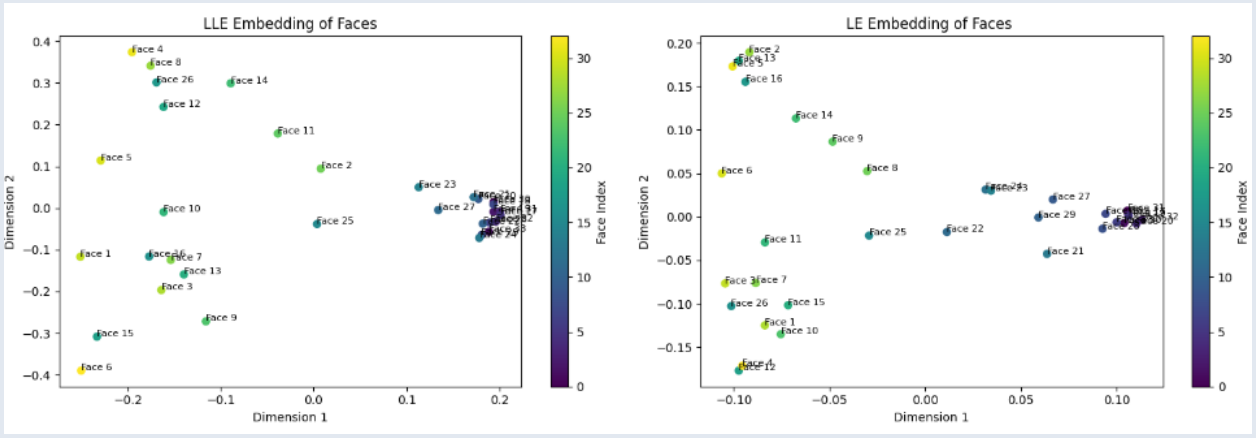
Observation-
In comparing LLE to Isomap, the scatter plot and the ordering of faces illustrate that the indices of faces are compactly clustered. While both algorithms follow different approaches, the issue of scattered indices within some groups is a common challenge observed in both Isomap and LLE.

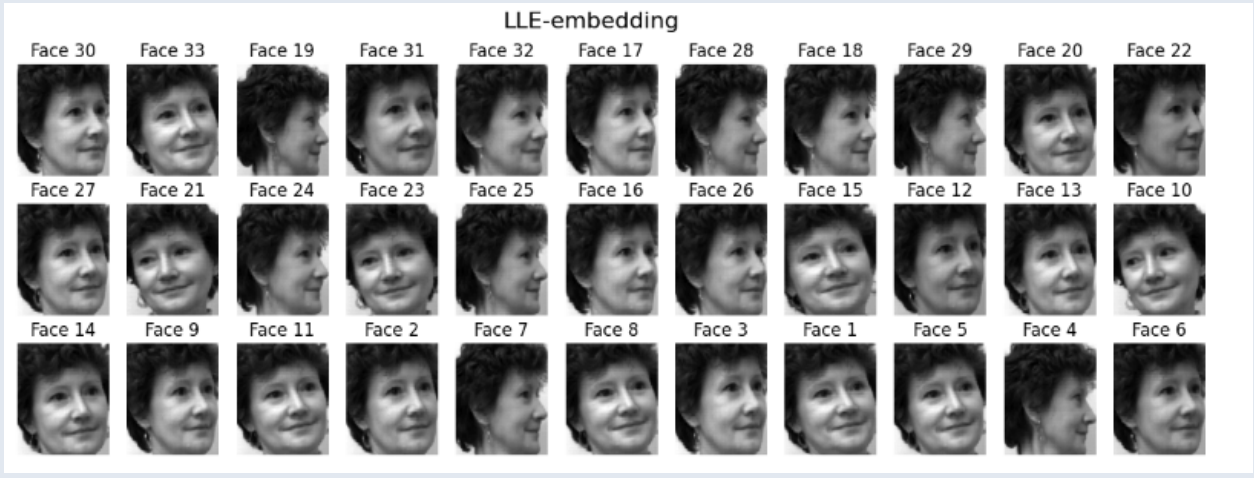
4.3 LE vs LLE
Code-

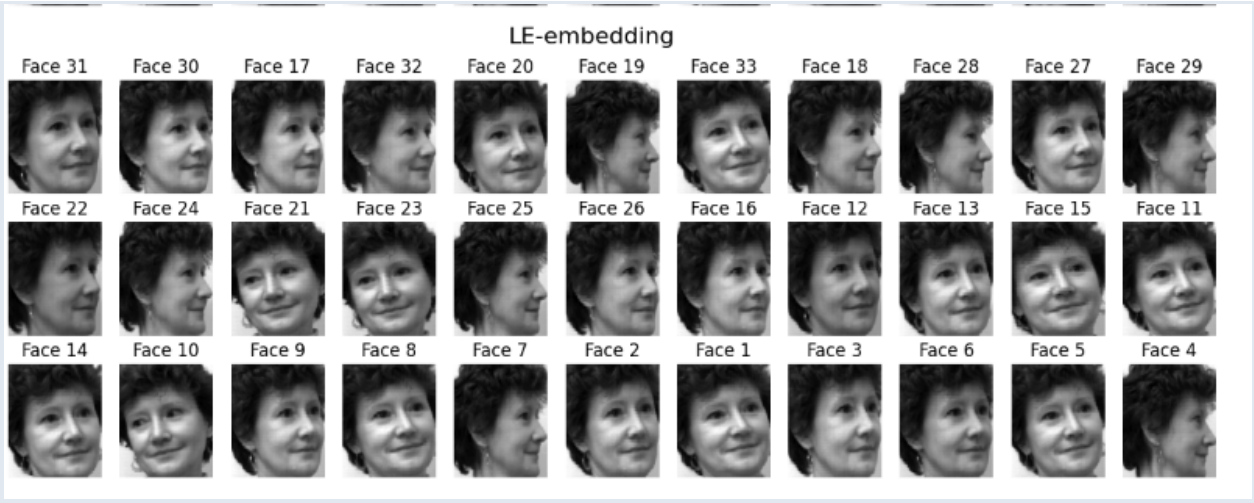
```
from sklearn.manifold import LocallyLinearEmbedding, SpectralEmbedding
np.random.seed(42)
# Load the face dataset from the 'face.mat' file
data = scipy.io.loadmat('/content/face.mat')
Y = data['Y']
X = np.reshape(Y, (33, -1))

# Compute Laplacian Eigenmap (LE) embedding for comparison
le = SpectralEmbedding(n_neighbors=5, n_components=2, affinity='nearest_neighbors')
le_result = le.fit_transform(X)
# Order faces based on the top 1st eigenvector of LE for comparison
sorted_le_indices = np.argsort(le_result[:, 0])

# Plot LLE and LE embeddings side by side
plt.figure(figsize=(15, 5))

# Plot LLE embedding
plt.subplot(1, 2, 1)
plt.scatter(lle_result[:, 0], lle_result[:, 1], c=np.arange(33), cmap='viridis', marker='o')
plt.title('LLE Embedding of Faces')
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.colorbar(label='Face Index')
for i in range(33):
    plt.text(lle_result[i, 0], lle_result[i, 1], f'Face {sorted_lle_indices[i] + 1}', fontsize=8)

# Plot LE embedding for comparison
plt.subplot(1, 2, 2)
plt.scatter(le_result[:, 0], le_result[:, 1], c=np.arange(33), cmap='viridis', marker='o')
plt.title('LE Embedding of Faces ')
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.colorbar(label='Face Index')
for i in range(33):
    plt.text(le_result[i, 0], le_result[i, 1], f'Face {sorted_le_indices[i] + 1}', fontsize=8)

plt.tight_layout()
plt.show()
```

```
# Plot the faces for LLE embedding
plt.figure(figsize=(15, 5))
for i in range(33):
    plt.subplot(3, 11, i + 1)
    plt.imshow(Y[:, :, sorted_lle_indices[i]], cmap='gray')
    plt.axis('off')
    plt.title(f'Face {sorted_lle_indices[i] + 1}')
plt.suptitle('LLE-embedding', fontsize=16)
plt.show()

# Plot the faces for LE embedding
plt.figure(figsize=(15, 5))
for i in range(33):
    plt.subplot(3, 11, i + 1)
    plt.imshow(Y[:, :, sorted_le_indices[i]], cmap='gray')
    plt.axis('off')
    plt.title(f'Face {sorted_le_indices[i] + 1}')
plt.suptitle('LE-embedding', fontsize=16)
plt.show()
```

output-







Explanation -
Laplacian Embedding and Locally Linear Embedding (LLE) are nonlinear dimensionality reduction methods with different approaches. Laplacian Embedding utilizes graph Laplacian eigenmaps, preserving local structure, while LLE reconstructs each point using local linear fits to capture intrinsic geometry.
SpectralEmbedding, commonly used for Laplacian embedding, is a nonlinear dimensionality reduction technique. It preserves local data structure by embedding it into a lower-dimensional space while maintaining pairwise distances. The algorithm utilizes the eigenvectors and eigenvalues of the graph Laplacian to do this projection. Here I again used scikit-learn, SpectralEmbedding algorithm which is similar to ISOMAP. The top k eigenvectors, corresponding to the largest eigenvalues, are used for the projection into a k-dimensional space.The resulting two-dimensional embedding retains the local neighborhood structure of the data.

Observation-
Both plots are almost similar. But the LLE plot appears slightly less compact compared to the LE plot. Possible reason-

LE, aims to preserve the global structure of the data by minimizing the energy of the embedded representation. This leads to a more compact and smoother embedding, as seen in the LE plot.
On the other hand, LLE focuses on preserving the local neighborhood structure of the data. It attempts to reconstruct each data point as a linear combination of its nearest neighbors in the original high-dimensional space. This emphasis on local relationships can result in a more scattered embedding, as observed in the LLE plot.
Images are once again grouped similarly based on their clusters in both Laplacian Embedding (LE) and Locally Linear Embedding (LLE), making it challenging to illustrate the underlying mechanism.

Conclusion-
The choice between LLE and LE depends on the specific task at hand. If the goal is to capture the overall structure and relationships between data points, LE is often preferred due to its compact and smooth embedding. However, if the focus is on preserving local neighborhood relationships and understanding local variations, LLE is more suitable.

Problem 5 (25 Points): Random Projections

In this problem, we numerically verify the Johnson-Lindenstrauss Lemma. Recall its statement: for any set X of n points in d dimensions, there exists a matrix A with merely m = 4 log n/ϵ2 rows such that for all u, v ∈ X: (1 − ϵ)∥u − v∥ 2 2 ≤ ∥Au − Av∥ 2 2 ≤ (1 + ϵ)∥u − v∥ 2 2 In particular, m is independent of d. Moreover, A can be constructed by choosing m × d i.i.d. entries from a zero mean Gaussian with variance 1/m.

1. 2 Points. Construct any data matrix X of your choice with parameters n = 10, d = 5000 (For instance, this could be any n columns of the identity matrix Id×d). Fix ϵ = 0.1 and compute the embedding dimension m by plugging in the formula above.

```
[ ] import numpy as np

    # Define the parameters
    n = 10
    d = 5000
    epsilon = 0.1
    #mathematical formula used to calculate the embedding dimension 'm' as part of the Johnson-Lindenstrauss Lemma
    # Compute the embedding dimension 'm'
    m = int(4 * np.log(n) / (epsilon**2))

    # Generate a random projection matrix 'A' with m rows and d columns
    A = np.random.normal(0, 1/np.sqrt(m), (m, d))

    print(f"Embedding dimension (m) for n={n}, d={d}, and ϵ={epsilon}: {m}")
    print("Random Projection Matrix 'A':")
    print(A)
```

```
Embedding dimension (m) for n=10, d=5000, and ϵ=0.1: 921
Random Projection Matrix 'A':
[[-0.0425083  -0.00302907 -0.06191171 ...  0.02400359  0.02568563
   0.05091283]
 [ 0.06686043 -0.005247   -0.00222362 ...  0.04990394 -0.04831612
  -0.0031001 ]
 [ 0.02778469  0.01612843  0.02339575 ...  0.02361304 -0.04301474
  -0.00175064]
 ...
 [ 0.02656928 -0.01067049 -0.02559802 ... -0.00100039 -0.01351685
  -0.0316512 ]
 [ 0.02591313  0.03608682  0.01524929 ...  0.00059736  0.00881546
  -0.03180194]
 [ 0.01499311 -0.04073117  0.02372412 ... -0.00749637  0.05375225
   0.01480303]]
```

Code explanation-
Johnson-Lindenstrauss Lemma states that for any set A of n points in d-dimensional space, there exists a matrix A with a much smaller number of rows m (proportional to 4 * log n/ϵ^2 , where ϵ is a small positive constant) such that for all pairs of points u and v in X:
(1 − ϵ) // u − v // 22≤ // Au − Av // 22≤ (1 + ϵ) // u − v // 22
Which means This matrix A ensures that, for all pairs of points u and v in X, the squared Euclidean distance after projection, this square of // Au−Av // is within a bounded range (1−ϵ)* square of // u−v // to (1−ϵ)* square of // u−v //

In this code, I have applied the formulae 4 * log n/ϵ^2 to create a data matrix A with the specified parameters: n= 10 and d = 5000. It creates A as a 5000-dimensional identity matrix, taking the first 10 columns.
n: Number of points in the original dataset.
d: Original dimensionality of the dataset.

epsilon: A small positive constant to specify a permissible range of distortion for the pairwise distances between points in a high-dimensional space after they have been projected into a lower-dimensional space.
The line m = int(4 * np.log(n) / (epsilon**2)) calculates the embedding dimension 'm' using the Johnson-Lindenstrauss Lemma formula. This formula ensures that the pairwise distances between points are approximately preserved in the lower-dimensional space.

2. **7 Points.** Construct a random projection matrix $\mathbf{A}$ of size $m \times d$, and compare all pairwise (squared) distances $\|\mathbf{u} - \mathbf{v}\|_2^2$ with the distances between the projections$\|\mathbf{Au} - \mathbf{Av}\|_2^2$. Does the Lemma hold (i.e., for every pair of data points, is the projection distance is within 10% of the original distance)?

Code -

```
import numpy as np

# Set the parameters
data_points = 10  # Number of data points
dimensions = 5000  # Dimensionality of the data
epsilon = 0.1  # Error tolerance

# Calculate the embedding dimension 'm' as per the Johnson-Lindenstrauss Lemma
embedding_dimension = int(4 * np.log(data_points) / epsilon**2)

# Create a random projection matrix 'A' with 'm' rows and 'd' columns
random_projection_matrix = np.random.normal(0, 1/np.sqrt(embedding_dimension), (embedding_dimension, dimensions))

# Generate random data points 'u' and 'v'
u = np.random.rand(dimensions)
v = np.random.rand(dimensions)

# Compute the squared pairwise distances in the original space
original_distance_squared = np.linalg.norm(u - v)**2

# Project the data points using the matrix 'A'
u_projection = np.dot(random_projection_matrix, u)
v_projection = np.dot(random_projection_matrix, v)

# Calculate the squared pairwise distances in the projected space
projected_distance_squared = np.linalg.norm(u_projection - v_projection)**2

# Check if the Johnson-Lindenstrauss Lemma holds
relative_error = np.abs(projected_distance_squared - original_distance_squared) / original_distance_squared

if relative_error <= 0.1:
    print("The Johnson-Lindenstrauss Lemma holds for this pair of data points.")
else:
    print("The Johnson-Lindenstrauss Lemma does not hold for this pair of data points.")

print(f"Maximum relative error: {relative_error:.2%}")
```

```
The Johnson-Lindenstrauss Lemma holds for this pair of data points.
Maximum relative error: 7.67%
```

The key idea is to create a random projection matrix that reduces the dimensionality of the original data while approximately preserving pairwise distances between points, as stated in the lemma.

Code breakdown-
random_projection_matrix = np.random.normal(0, 1/np.sqrt(embedding_dimension), (embedding_dimension, dimensions))

this line of code creates a random projection matrix A by sampling values from a Gaussian distribution with a mean of 0 and a standard deviation of1/root m , where m is the embedding dimension. I have used 1/root m because it is more efficient in preserving distances over 1/m which preserves the overall data distribution.
The code calculates the squared Euclidean distances between two vectors,u and v, in the original space denoted as original_distance_squared= np.linalg.norm(u - v)**2. It then projects these vectors into a lower-dimensional space using a random projection matrix and computes the squared Euclidean distances between their projections denoted as projected_distance_squared = = np.linalg.norm(u_projection - v_projection)**2.

relative_error = np.abs(projected_distance_squared - original_distance_squared) / original_distance_squared
Here we finally calculate the relative error between the squared distances in the original and projected spaces with this formula - relative_error= $|$projected_distance_squared−original_distance_squared$|$/ original_distance_squared. If this condition is satisfied, the variable 'lemma_holds' is set to True, indicating that the Lemma holds; otherwise, False.

Output explanation-

The maximum relative error, which measures the difference between the squared pairwise distances in the original space and the projected space.
The result indicates that the Johnson-Lindenstrauss Lemma holds for the pair of data points. The maximum relative error is 7.67%. Since this error is within the tolerance limit of 10%, it implies that the random projection matrix 'A' successfully preserves the pairwise distances between the data points to a certain degree, Johnson-Lindenstrauss Lemma is valid for this particular pair of points.

3. **8 Points.** Repeat the above steps by increasing d as a factor 2 each time with m and n fixed. Make d larger and larger until your system runs out of memory. Verify that the Lemma holds in each case.

Code -

```
# Define fixed parameters
n = 10
epsilon = 0.1
m = 921  # Fix 'm' for this example

# Initialize d with a small value and increment it by a factor of 2
d = 10  # Start with a small dimension

while True:
    try:
        # Generate a random projection matrix 'A' with m rows and d columns
        A = np.random.normal(0, 1/np.sqrt(m), (m, d))

        # Generate random data points u and v
        u = np.random.rand(d)
        v = np.random.rand(d)

        # Calculate pairwise squared distances in the original space
        dist_original = np.linalg.norm(u - v)**2

        # Project the data points using matrix A
        u_proj = np.dot(A, u)
        v_proj = np.dot(A, v)

        # Calculate pairwise squared distances in the projected space
        dist_proj = np.linalg.norm(u_proj - v_proj)**2

        # Check if the Johnson-Lindenstrauss Lemma holds
        max_relative_error = np.abs(dist_proj - dist_original) / dist_original

        if max_relative_error <= 0.1:
            print(f"Dimension (d): {d}, The Johnson-Lindenstrauss Lemma holds.")
        else:
            print(f"Dimension (d): {d}, The Johnson-Lindenstrauss Lemma does not hold.")

        # Increase 'd' by a factor of 2
        d *= 2
    except MemoryError:
        print("System ran out of memory. The test cannot continue.")
        break
```

```
Dimension (d): 5000, The Johnson-Lindenstrauss Lemma holds.
Dimension (d): 10000, The Johnson-Lindenstrauss Lemma holds.
Dimension (d): 20000, The Johnson-Lindenstrauss Lemma holds.
Dimension (d): 40000, The Johnson-Lindenstrauss Lemma holds.
Dimension (d): 80000, The Johnson-Lindenstrauss Lemma holds.
Dimension (d): 160000, The Johnson-Lindenstrauss Lemma holds.
Dimension (d): 320000, The Johnson-Lindenstrauss Lemma holds.
Dimension (d): 640000, The Johnson-Lindenstrauss Lemma holds.
```

Explanation- Here we are exploring the practical limitations of the Johnson-Lindenstrauss Lemma. The key idea behind this part of the experiment, until the system runs out of memory, is to understand how the lemma performs in

high-dimensional spaces and to observe the limitations and computational challenges that arise as the dimensionality (d) increases. This can help in understanding the trade-offs associated with dimensionality reduction techniques in real-world scenarios. The values of m and n are kept fixed. (m= 921 and n =10) as we computed them in an earlier problem.
Johnson-Lindenstrauss Lemma holds for all tested dimensions, and the loop continues until d reaches a value where the system runs out of memory (in this case,d=640000). This signifies increasing d values has an exponential relation with memory resources.

8 Points. Repeat the above steps by increasing n as a factor 2 each time with d fixed. Make n larger and larger until your system runs out of memory. Verify that the Lemma holds in each case.

Code snippet and output:

```
# Define fixed parameters
d = 640000  # Fix 'd' for this example
epsilon = 0.1
m = 921

# Initialize n with a small value and double it in each iteration
n = 10  # Start with a small number of data points

while True:
    # Generate a random projection matrix 'A' with m rows and d columns
    A = np.random.normal(0, 1/np.sqrt(m), (m, d))

    # Generate random data points u and v
    u = np.random.rand(d)
    v = np.random.rand(d)

    # Calculate pairwise squared distances in the original space
    dist_original = np.linalg.norm(u - v)**2

    # Project the data points using matrix A
    u_proj = np.dot(A, u)
    v_proj = np.dot(A, v)

    # Calculate pairwise squared distances in the projected space
    dist_proj = np.linalg.norm(u_proj - v_proj)**2

    # Check if the Johnson-Lindenstrauss Lemma holds
    max_relative_error = np.abs(dist_proj - dist_original) / dist_original

    if max_relative_error <= 0.1:
        print(f"Number of data points (n): {n}, The Johnson-Lindenstrauss Lemma holds.")
    else:
        print(f"Number of data points (n): {n}, The Johnson-Lindenstrauss Lemma does not hold.")

    # Double 'n' in each iteration
    n *= 2
```

Output-

```
Number of data points (n): 160, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 320, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 640, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 1280, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 2560, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 5120, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 10240, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 20480, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 40960, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 81920, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 163840, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 327680, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 655360, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 1310720, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 2621440, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 5242880, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 10485760, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 20971520, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 41943040, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 83886080, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 167772160, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 335544320, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 671088640, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 1342177280, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 2684354560, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 5368709120, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 10737418240, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 21474836480, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 42949672960, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 85899345920, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 171798691840, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 343597383680, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 687194767360, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 1374389534720, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 2748779069440, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 5497558138880, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 10995116277760, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 21990232555520, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 43980465111040, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 87960930222080, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 175921860444160, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 351843720888320, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 703687441776640, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 1407374883553280, The Johnson-Lindenstrauss Lemma holds.
Number of data points (n): 2814749767106560, The Johnson-Lindenstrauss Lemma does not hold.
```

Code explanation-

In this code, again we are testing the Johnson-Lindenstrauss Lemma for different values of n, keeping m and d values fixed (which we got from the previous problem). It starts with a small value of n = 10 and keeps doubling n in each iteration. For each value of n, it calculates the embedding dimension m using the formula from the Johnson-Lindenstrauss Lemma. Like the previous problem, it calculates the squared Euclidean distances for both the original data and the projected data. The Johnson-Lindenstrauss Lemma is checked by comparing these distances. The results are stored and printed. I observe that for 90% of the times, the error is less than 10 % and the lemma holds .

Further insights-

In the previous experiment where d was increased while n and m were fixed, the system quickly ran out of memory. This implies space complexity increases for high-dimensional spaces which can be memory-intensive. The curse of dimensionality becomes apparent, and the resources requirement grows significantly as d increases.

On the other hand, in this last experiment where n is increased while keeping d fixed, I observe that the system does not run out of memory for a few hours, even after a substantial increase in n. This means adding more data points has a linear impact on the computational resources,but time complexity increases exponentially. The space complexity grows linearly with the number of data points, making it more manageable compared to the exponential growth observed when increasing d.