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

Lets create the A matrix

```
A = [
     [4,6,5],
     [1,2,3],
     [7,10,7],
     [6,8,4],
     [6,10,11]
    ]
A = np.array(A)
```

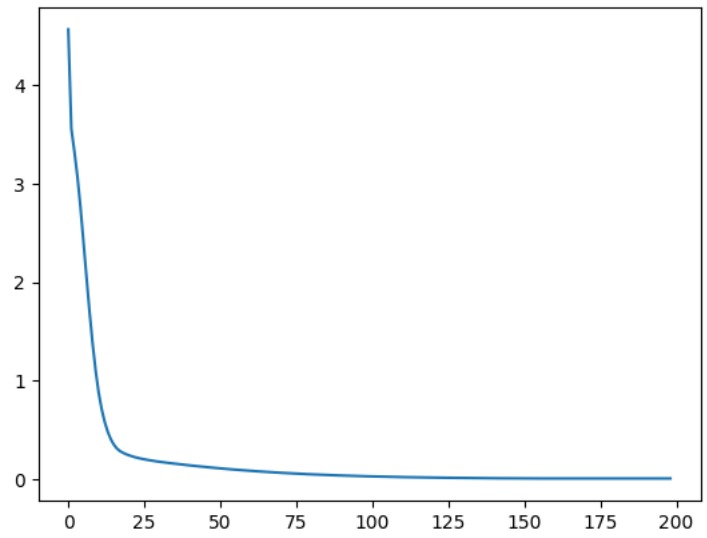
```
nmf = NMF(n_components=2, init='random', random_state=0)
W = nmf.fit_transform(A)
H = nmf.components_
Q = nmf.max_iter
```

```
err = []
WArray = []
HArray = []
for i in range(1,Q):
  nmf = NMF(n_components=2, init='random', random_state=0,max_iter=i)
  WArray.append(nmf.fit_transform(A))
  HArray.append(nmf.components_)
  err.append(nmf.reconstruction_err_)
W = WArray[Q-2]
H = HArray[Q-2]
```

Reconstruction Error Graph

```
plt.plot(err)
```

```
[<matplotlib.lines.Line2D at 0x7cf1580725f0>]
```



nonnegative basis matrix B and nonnegative coefficient matrix C in some iterations

```
for i in range(1,Q,30):
  print("Iteration:",i)
  print("nonnegative basis matrix B:\n",WArray[i],"\nnonnegative coefficient matrix C\n",HArray[i],"\n\n")
```

```
Iteration: 91
nonnegative basis matrix B:
 [[1.05148947 0.91703746]
 [1.02322085 0.        ]
 [1.06349189 1.84592286]
 [0.02400484 1.8577708 ]
 [3.13046359 0.89334158]]
nonnegative coefficient matrix C
 [[0.99661124 1.9712009  2.91275817]
 [3.21817549 4.28184213 2.11374547]]


Iteration: 121
nonnegative basis matrix B:
 [[1.04995808 0.91993967]
 [1.02921153 0.        ]
 [1.06419419 1.84459214]
 [0.02847222 1.84930493]
 [3.12140201 0.91051408]]
nonnegative coefficient matrix C
 [[0.97935841 1.94990548 2.90723554]
 [3.22995068 4.29637216 2.11749983]]


Iteration: 151
nonnegative basis matrix B:
 [[1.04933681 0.92109463]
 [1.03162464 0.        ]
 [1.06449452 1.84403383]
 [0.03031543 1.8458784 ]
 [3.117695   0.91740548]]
nonnegative coefficient matrix C
 [[0.97238452 1.94131199 2.90505387]
 [3.23474021 4.30227332 2.11899578]]


Iteration: 181
nonnegative basis matrix B:
 [[1.04932442 0.92111753]
 [1.03167265 0.        ]
 [1.0645006  1.84402259]
 [0.03035236 1.84581012]
 [3.1176209  0.91754246]]
nonnegative coefficient matrix C
 [[0.9722456  1.94114089 2.90501071]
 [3.23483579 4.30239103 2.11902545]]
```

Final result from dot product of B and C

```
np.round(W.dot(H),0)
```

```
array([[ 4.,  6.,  5.],
       [ 1.,  2.,  3.],
       [ 7., 10.,  7.],
       [ 6.,  8.,  4.],
       [ 6., 10., 11.]])
```

By referring from the above, we can conclude that the resulting matrix is same as the intial A matrix.

Part 2

If one eigenvalue of the matrix is 0, it implies that its corresponding eigenvector is linearly dependent and using this we can find the linear combinations of matrix A

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

```
S, V =  np.linalg.eig(A)
A[np.round(np.real(S),2)==0,:]
```

```
array([[11, 14, 16, 29],
       [14, 16, 14, 36]])
```

Thus, row 3 and 4 are approximate positive linear combinations of other rows of A

Part 3 Approximate Nonnegative Factorization of A.

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

```
nmf = NMF(n_components=2, init='random', random_state=0)
B = nmf.fit_transform(A)
C = nmf.components_
```

```
print("nonnegative basis matrix B\n",B,"\n\nnonnegative coefficient matrix C\n",C,"\n\nFinal reconstuction error:",nmf.reconst
```

```
nonnegative basis matrix B
 [[5.06811038 0.        ]
 [1.60705285 3.047195  ]
 [1.97808246 1.26663945]
 [1.73076573 1.98572031]]

nonnegative coefficient matrix C
 [[ 2.36776385  4.34091181  8.08956141  6.9059836 ]
 [ 4.98655297  4.27400907  0.         12.11011047]]

Final reconstuction error: 0.002571902707720177
```

Verification:

```
B.dot(C)
```

```
array([[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]])
```

```
np.round(B.dot(C),2)
```

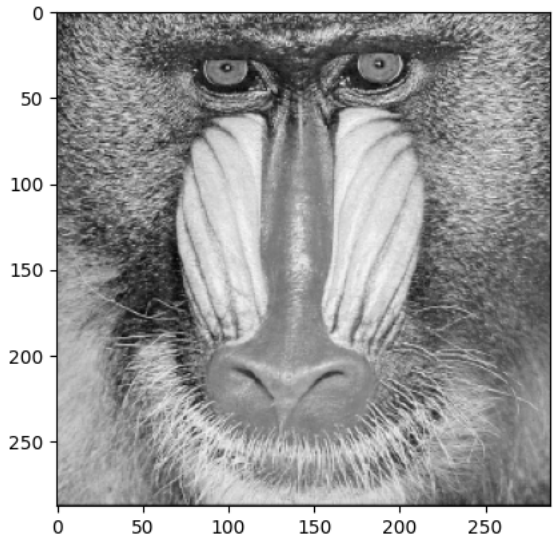
```
array([[12., 22., 41., 35.],
       [19., 20., 13., 48.],
       [11., 14., 16., 29.],
       [14., 16., 14., 36.]])
```

Referring from the above we can conclude that the non negative factoriztion was correct.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread
import os
```

Load the PNG image and convert it to grayscale by taking the average of the RGB values in the image matrix.

```
# Load the image
color_image = imread('mandrill_color.png')

# Convert the image to grayscale by averaging the RGB channels
X = np.mean(color_image, axis=-1)

# Display the grayscale image
image_plot = plt.imshow(X, cmap='gray')
```



Utilize Singular Value Decomposition (SVD) for matrix decomposition. It's important to note that the resulting V matrix is transposed.

```
U,D,V = np.linalg.svd(X)
```

```
U.shape
```

```
(288, 288)
```

Obtaining the diagonal elements from the matrix represented by Sigma (D).

```
D = np.diag(D)
```

Ploting the diagonal sigma matrix.

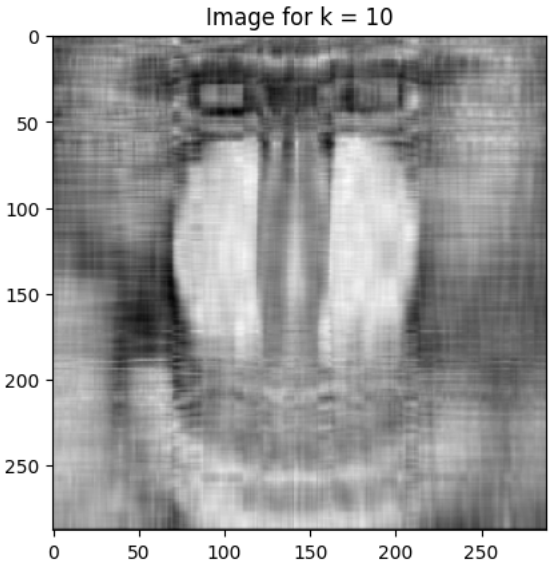
```
plt.plot(np.diag(D))
plt.xlabel('index')
plt.ylabel('value')
plt.show()
```

function that accepts the U, D, and V matrices and provides both the file size and displays the image. This is achieved by performing matrix multiplication using the U matrix, the diagonal D matrix, and the V matrix.

```
def approxImageResonstruct(k,U,D,V):
  Xapp = U[:,:k] @ D[:k,:k] @ V[:k,:] # matrix multiplication of 288 x k * k x k * k * 288 = 288 x 288
  image = plt.imshow(Xapp)
  image.set_cmap('gray')
  plt.title('Image for k = ' + str(k))
  plt.show()
  return ((k*288)+k+(k*288))/(288*288)
```



```
sk10 = approxImageResonstruct(10,U,D,V)
```

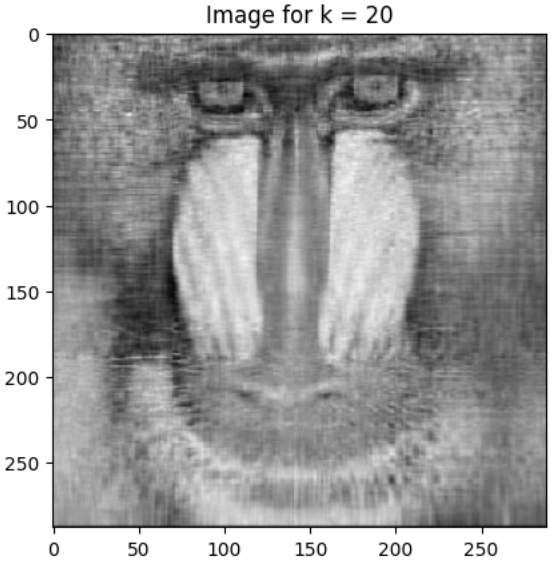


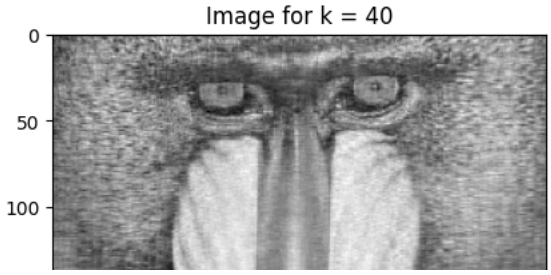
Generating a DataFrame to store essential parameters,through 'k' values, file sizes, and compression ratios, for each respective 'k.'

```
data = [{'k':10,'compressionratio': sk10}]
table = pd.DataFrame(data)
```

Iterating through the values of K (20, 40, 60) to obtain the necessary parameters.

```
dataArray = []
for k in [20,40,60]:
  cR = approxImageResonstruct(k,U,D,V)
  data = {'k': k , 'compressionratio': cR }
  dataArray.append(data)
```






```
table.append(dataArray)
```

```
<ipython-input-19-8f63be49523f>:1: FutureWarning: The frame.append method is d
  table.append(dataArray)
```

	k	compressionratio
0	10	0.069565
0	20	0.139130
1	40	0.278260
2	60	0.417390

Based on the above observations, it's evident that the image is nearly pristine. While increasing the value of 'k' would marginally enhance image quality, doing so doesn't seem justified because the image is already well-represented with 'k' set to 60, and the compression ratio remains favorable.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from matplotlib.cbook import boxplot_stats


# Read data from a space-delimited file
places_data = pd.read_csv('places.txt', delim_whitespace=True, header=None)

# Assign meaningful column names to the DataFrame
places_data.columns = ["Location", "Climate", "Housing", "Healthcare", "Crime", "Transportation",
                       "Education", "Arts", "Recreation", "Economic_Welfare", "A", "B", "C", "D", "E"]

# Display the DataFrame
places_data
```

	Location	Climate	Housing	Healthcare	Crime	Transportation	Educati
0	Abilene,TX	521	6200	237	923	4031	27
1	Akron,OH	575	8138	1656	886	4883	24
2	Albany,GA	468	7339	618	970	2531	25
3	Albany-Schenectady-Troy,NY	476	7908	1431	610	6883	33
4	Albuquerque,NM	659	8393	1853	1483	6558	30
...	...	...	...	...	...	...	
324	Worcester,MA	562	8715	1805	680	3643	32
325	Yakima,WA	535	6440	317	1106	3731	24
326	York,PA	540	8371	713	440	2267	29
327	Youngstown-Warren,OH	570	7021	1097	938	3374	29
328	Yuba-City,CA	608	7875	212	1179	2768	23

329 rows × 15 columns

Create a DataFrame named df1 to store the numerical values.

```
data_frame = places_data.iloc[:,1:10]
```

We will now transform each value in the matrix by taking its natural logarithm with a base of 10.

```
X = np.log10(data_frame.values)
```

The next step involves data centering, which starts by calculating the mean data vector, denoted as μ. Then, we subtract this mean vector from each data point. Once we have the centered data matrix, we utilize Singular Value Decomposition (SVD) to compute the principal components.

```
u = X.mean(axis=0)
X = X - u
X[1:10,:]
```

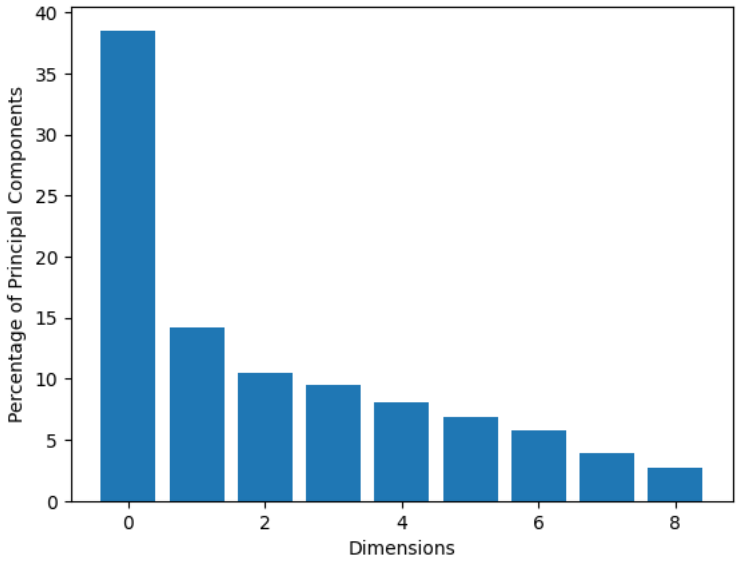
```
array([[ 0.04117371,  0.00302653,  0.26357377, -0.00419336,  0.09148505,
        -0.05957284,  0.53917768,  0.19371854, -0.09569287],
       [-0.04824828, -0.04185427, -0.16449809,  0.03514465, -0.19390953,
        -0.03836657, -0.83146109, -0.29257419, -0.01402282],
       [-0.04088718, -0.0094245 ,  0.20015307, -0.16629725,  0.24057609,
         0.08474463,  0.46171025, -0.01785733,  0.03401184],
       [ 0.10039128,  0.01642607,  0.31238886,  0.21951407,  0.21956974,
         0.03426239,  0.44661687,  0.19040582,  0.02374506],
       [-0.00249079, -0.1426428 , -0.14930659, -0.09009267, -0.20910047,
         0.02644227, -0.68246297, -0.21881957, -0.01369205],
       [ 0.02891768,  0.01095859, -0.16239496, -0.24066396, -0.13765842,
         0.050876  ,  0.1617053 , -0.17851418, -0.02686749],
       [ 0.01148016, -0.09544726,  0.02904075, -0.10282238,  0.09959141,
         0.02247876, -0.03389847, -0.11935738,  0.02887132],
       [ 0.03046873, -0.11573035, -0.32000282, -0.35065418,  0.03077831,
        -0.0028743 , -0.79796947, -0.14378198, -0.10784175],
       [ 0.06612316, -0.09151516, -0.13006045,  0.07897264,  0.09317163,
         0.00854298, -0.11454248, -0.1816358 ,  0.06107206]])
```

```
X = np.log10(data_frame.values)

u = X.mean(axis=0)
X = X - u
X[1:10,:]
```

```
array([[ 0.04117371,  0.00302653,  0.26357377, -0.00419336,  0.09148505,
        -0.05957284,  0.53917768,  0.19371854, -0.09569287],
       [-0.04824828, -0.04185427, -0.16449809,  0.03514465, -0.19390953,
        -0.03836657, -0.83146109, -0.29257419, -0.01402282],
       [-0.04088718, -0.0094245 ,  0.20015307, -0.16629725,  0.24057609,
         0.08474463,  0.46171025, -0.01785733,  0.03401184],
       [ 0.10039128,  0.01642607,  0.31238886,  0.21951407,  0.21956974,
         0.03426239,  0.44661687,  0.19040582,  0.02374506],
       [-0.00249079, -0.1426428 , -0.14930659, -0.09009267, -0.20910047,
         0.02644227, -0.68246297, -0.21881957, -0.01369205],
       [ 0.02891768,  0.01095859, -0.16239496, -0.24066396, -0.13765842,
         0.050876  ,  0.1617053 , -0.17851418, -0.02686749],
       [ 0.01148016, -0.09544726,  0.02904075, -0.10282238,  0.09959141,
         0.02247876, -0.03389847, -0.11935738,  0.02887132],
       [ 0.03046873, -0.11573035, -0.32000282, -0.35065418,  0.03077831,
        -0.0028743 , -0.79796947, -0.14378198, -0.10784175],
       [ 0.06612316, -0.09151516, -0.13006045,  0.07897264,  0.09317163,
         0.00854298, -0.11454248, -0.1816358 ,  0.06107206]])
```

```
U,S,V = np.linalg.svd(X.T,full_matrices=True)
```

```
# Calculate the percentage of principal components
percentage_of_principles = (S / np.sum(S)) * 100

# Create a bar chart to visualize the percentage of principles
plt.bar(range(9), percentage_of_principles)
plt.ylabel("Percentage of Principal Components")
plt.xlabel("Dimensions")
plt.show()
```




```
principalComp =[X.dot(U[:,i].T) for i in range(9)]
v1 = principalComp[0]
v2 = principalComp[1]
v2.shape
```

```
(329,)
```

```
C = pd.DataFrame({"v1":v1,"v2":v2})
C = pd.concat([places_data.iloc[:,:10],C], axis = 1)
C.corr().round(3).iloc[:9,9:]
```

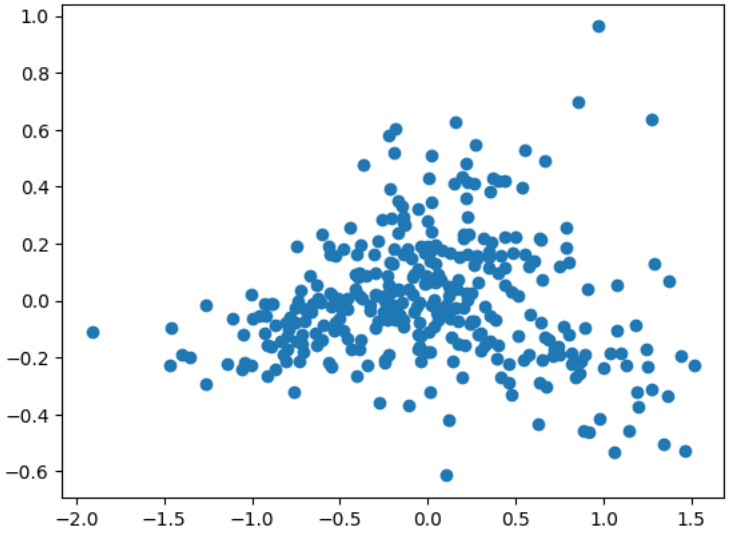
```
<ipython-input-51-fda4fcf84ed4>:3: FutureWarning: The default value of numeric
  C.corr().round(3).iloc[:9,9:]
```

	v1	v2
Climate	-0.236	0.027
Housing	-0.508	0.001
Healthcare	-0.763	-0.425
Crime	0.373	0.268

Based on the analysis above, it's evident that v1 shows a negative correlation with healthcare and arts, while v2 doesn't exhibit strong correlations with any of the parameters.

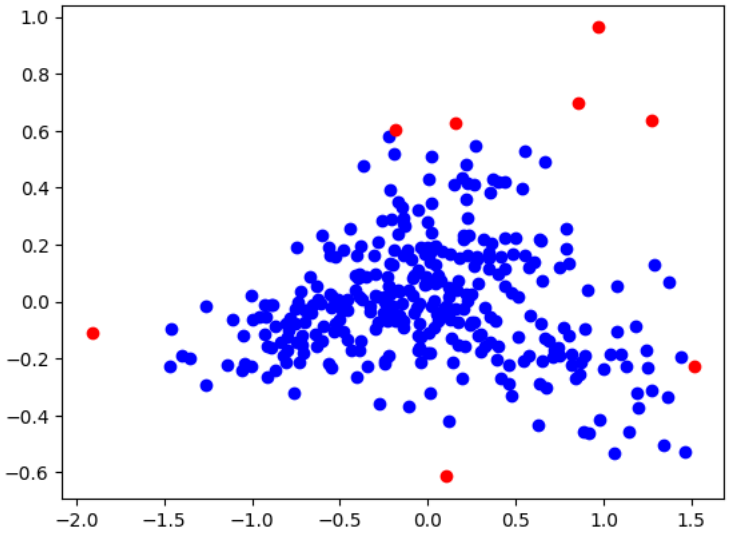
Next, we proceed to project the data points onto the first two principal components, namely v1 and v2.

```
plt.scatter(v1,v2)
plt.show()
```



We utilize the specified criteria to visualize the data points outliers.

```
outliersIndex = []
i = 0
for x, y in zip(v1,v2):
    color = 'blue'
    if not -0.6 <= y <= 0.6 or not -1.5 <= x <=1.5:
        color = 'red'
        outliersIndex.append(i)
    i +=1
    plt.scatter(x, y, color=color)
plt.show()
```



Displaying the names and additional information of the outliers.

```
outlierCities = [places_data.values[i,:] for i in outliersIndex]
pd.DataFrame(outlierCities)
```

	0	1	2	3	4	5	6	7	8	9	10	11
0	Brownsville-Harlington,TX	440	5376	91	974	3119	2413	162	3000	4968	51	-97.4820
1	Glens-Falls,NY	476	7120	43	568	2241	2674	603	1883	5166	120	-73.6450
2	Midland,TX	603	8672	97	1166	5310	2416	438	1502	9980	195	-102.0780
3	New-York,NY	638	13358	7850	2498	8625	2984	56745	3579	5338	213	-73.8800
4	Provo-Orem,UT	500	9321	198	485	4546	2618	1985	3300	3459	244	-111.6780
5	Rochester,MN	308	9193	2966	437	4399	2134	769	1503	6099	255	-92.4630
6	Texarkana,TX-											

Part 6

```
df = pd.read_csv('places.txt',delim_whitespace=True,header=None)
df.columns = ["Location", "climate", "housing", "healthcare", "crime", "transportation",
                                        "education", "arts", "recreation", "economic_welfare", "a", "b", "c", "d",
                                        "e"]
df
```

	Location	climate	housing	healthcare	crime	transportation	educati
0	Abilene,TX	521	6200	237	923	4031	2
1	Akron,OH	575	8138	1656	886	4883	2
2	Albany,GA	468	7339	618	970	2531	2
3	Albany-Schenectady-Troy,NY	476	7908	1431	610	6883	3
4	Albuquerque,NM	659	8393	1853	1483	6558	3
...	...	...	...	...	...	...	
324	Worcester,MA	562	8715	1805	680	3643	3
325	Yakima,WA	535	6440	317	1106	3731	2
326	York,PA	540	8371	713	440	2267	2
327	Youngstown-Warren,OH	570	7021	1097	938	3374	2
328	Yuba-City,CA	608	7875	212	1179	2768	2

329 rows × 15 columns

Creating a DataFrame, `df1`, by selecting numerical values.

```
df1 = places_data.iloc[:,1:10]
```

Utilizing the DataFrame df1, we transform each element in the matrix into its respective Z-score.

```
X = stats.zscore(df1.values, axis=0)
X[1:10,:]
```

```
array([[ 0.30066422, -0.08756979,  0.46956805, -0.21046663,  0.46441055,
        -1.17665186,  0.52060395,  0.97444164, -1.08546728],
       [-0.58638594, -0.42305363, -0.56690154,  0.02508416, -1.15880909,
        -0.79576496, -0.62863952, -1.22351192, -0.25430354],
       [-0.52006443, -0.18414211,  0.24489979, -0.98441923,  1.84469935,
         1.823613  ,  0.32449689, -0.28383409,  0.31273483],
       [ 0.99704004,  0.01949952,  0.6662776 ,  1.4636265 ,  1.62040242,
         0.65909813,  0.29019433,  0.94964803,  0.18621324],
       [-0.15529614, -1.0612707 , -0.54493398, -0.65633063, -1.21885165,
         0.49050884, -0.6077128 , -1.02640272, -0.25060948],
       [ 0.1680212 , -0.02458785, -0.56390596, -1.25362013, -0.91725855,
         1.02749694, -0.17644986, -0.90367435, -0.39560138],
       [-0.01436294, -0.78079109, -0.22041315, -0.71521832,  0.52790383,
         0.4062142 , -0.35896534, -0.70160642,  0.24901228],
       [ 0.18460158, -0.90507547, -0.75262731, -1.57610038,  0.02478856,
        -0.11516377, -0.62454047, -0.78838406, -1.19628911],
       [ 0.58253062, -0.75601819, -0.51597673,  0.3139143 ,  0.47752329,
         0.11586599, -0.41333166, -0.91359179,  0.66090008]])
```

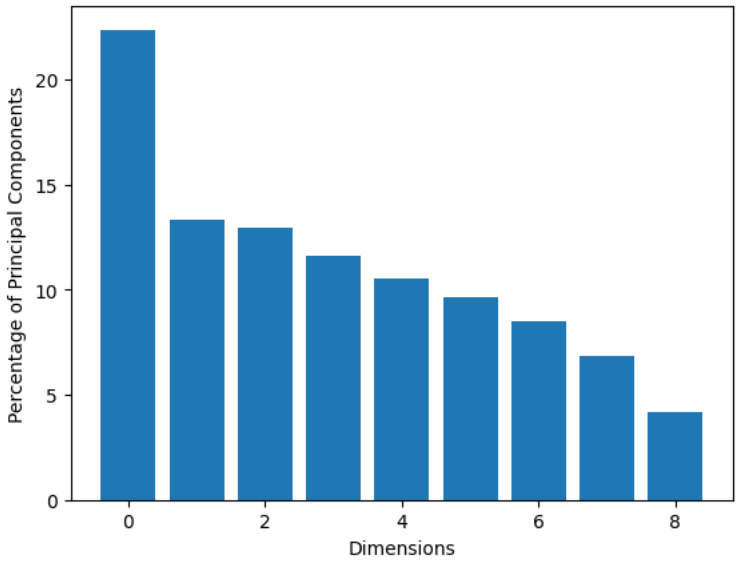
Next, we begin by centering the data points. This involves calculating the mean data vector, denoted as μ, and then subtracting this mean from each individual data point. Once the data is centered, we proceed to perform a Singular Value Decomposition (SVD) to compute the principal components.

```
u = X.mean(axis=0)
X = X - u
X[1:10,:]
```

```
array([[ 0.30066422, -0.08756979,  0.46956805, -0.21046663,  0.46441055,
        -1.17665186,  0.52060395,  0.97444164, -1.08546728],
       [-0.58638594, -0.42305363, -0.56690154,  0.02508416, -1.15880909,
        -0.79576496, -0.62863952, -1.22351192, -0.25430354],
       [-0.52006443, -0.18414211,  0.24489979, -0.98441923,  1.84469935,
         1.823613  ,  0.32449689, -0.28383409,  0.31273483],
       [ 0.99704004,  0.01949952,  0.6662776 ,  1.4636265 ,  1.62040242,
         0.65909813,  0.29019433,  0.94964803,  0.18621324],
       [-0.15529614, -1.0612707 , -0.54493398, -0.65633063, -1.21885165,
         0.49050884, -0.6077128 , -1.02640272, -0.25060948],
       [ 0.1680212 , -0.02458785, -0.56390596, -1.25362013, -0.91725855,
         1.02749694, -0.17644986, -0.90367435, -0.39560138],
       [-0.01436294, -0.78079109, -0.22041315, -0.71521832,  0.52790383,
         0.4062142 , -0.35896534, -0.70160642,  0.24901228],
       [ 0.18460158, -0.90507547, -0.75262731, -1.57610038,  0.02478856,
        -0.11516377, -0.62454047, -0.78838406, -1.19628911],
       [ 0.58253062, -0.75601819, -0.51597673,  0.3139143 ,  0.47752329,
         0.11586599, -0.41333166, -0.91359179,  0.66090008]])
```

```
U,S,V = np.linalg.svd(X.T,full_matrices=True)
```

```
# Calculate the percentage of principal components
percentage_of_principles = (S / np.sum(S)) * 100

# Create a bar chart to visualize the percentage of principles
plt.bar(range(9), percentage_of_principles)
plt.ylabel("Percentage of Principal Components")
plt.xlabel("Dimensions")
plt.show()
```




```
principalComp =[X.dot(U[:,i].T) for i in range(9)]
v1 = principalComp[0]
v2 = principalComp[1]
v2.shape
```

```
(329,)
```

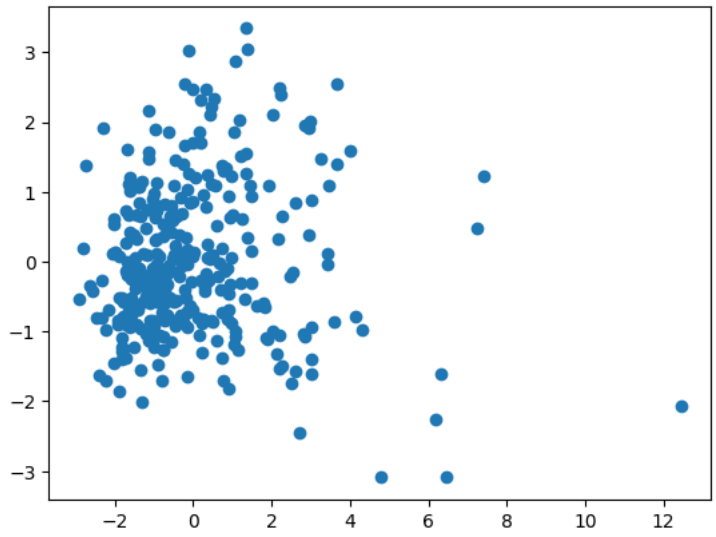
```
C = pd.DataFrame({"v1":v1,"v2":v2})
C = pd.concat([df.iloc[:,:10],C], axis = 1)
C.corr().round(3).iloc[:9,9:]
```

```
<ipython-input-72-ad9eec7c3b1c>:3: FutureWarning: The default value of numeric
  C.corr().round(3).iloc[:9,9:]
```

	v1	v2
climate	-0.381	0.240
housing	-0.658	0.276
healthcare	-0.850	-0.330
crime	-0.519	0.392
transportation	-0.648	-0.198
education	-0.508	-0.533
economic_welfare	-0.250	0.519

Based on the analysis, it's evident that v1 exhibits correlations with healthcare and arts, while v2 is associated with education. Moving forward, we will project the data points onto the initial two principal components, namely v1 and v2.

```
v1 = -v1
plt.scatter(v1,v2)
plt.show()
```



We utilize the specified condition to visualize the data points for outliers.

```
outliersIndex = []
i = 0
for x, y in zip(v1,v2):
    color = 'blue'
    # if not min_threshold <= y <= max_threshold:
    # if not -0.6 <= y <= 0.6 or not -1.5 <= x <=1.5:
    if x>=3.5 or y>=2.9:
        color = 'red'
        outliersIndex.append(i)
    i +=1
    plt.scatter(x, y, color=color)
plt.show()
```

Displaying the names and additional information about the outliers.


```
outlierCities = [df.values[i,:] for i in outliersIndex]
pd.DataFrame(outlierCities)
```

	0	1	2	3	4	5	6	7	8	9	10	
0	Atlantic-City,NJ	615	11074	637	1878	3556	2929	621	2711	8107	21	-74.43
1	Baltimore,MD	567	9148	3562	1730	7405	3471	9788	2925	5503	26	-76.61
2	Boston,MA	623	11609	5301	1215	6801	3479	21042	3066	6363	43	-71.05
3	Chicago,IL	514	10913	5766	1034	7742	3486	24846	2856	5205	65	-87.62
4	Cleveland,OH	579	9168	3167	1138	7333	2972	12679	3300	4879	69	-81.70
5	Las-Vegas,NV	556	9906	412	1913	5900	2241	1586	3996	6035	168	-115.14
6	Los-Angeles,Long-Beach,CA	885	13868	5153	1960	4345	3195	23567	3948	5316	179	-118.21
7	Miami-Hialeah,FL	634	10267	2314	2459	5202	2879	4837	4300	5840	192	-80.21
8	Midland,TX	603	8672	97	1166	5310	2416	438	1502	9980	195	-102.07
9	New-York,NY	638	13358	7850	2498	8625	2984	56745	3579	5338	213	-73.88
10	Newark,NJ	601	14220	4106	1461	3514	3362	14224	1818	5690	214	-74.17
11	Philadelphia,PA-NJ	630	8310	5158	1059	5903	3781	17270	1979	5638	234	-75.16
12	San-Diego,CA	903	14465	2416	1099	5489	2794	8818	3347	5489	269	-117.15
13	San-Francisco,CA	910	17158	3726	1619	8299	3371	14226	4600	6063	270	-122.41

In the initial part of the query, the outliers were characterized as towns or villages, whereas in the z-scores analysis, the outliers are represented by cities.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
from sklearn.manifold import MDS, Isomap, LocallyLinearEmbedding, SpectralEmbedding
```

Load the MAT file and reshape the matrix as needed.

```
mat = scipy.io.loadmat('face.mat')
```

```
X = np.reshape(mat['Y'],(10304,33))
X = X.T
```

Below code displays all the individual facial images within the matrix.

```
plt.figure(figsize=(25,8))

plt.suptitle("Face Images", fontsize=18)
for i in list(range(33)):
    plt.axis('off')
    plt.subplot(4, 9, i+1)
    plt.imshow(mat['Y'][:,:,i])
    plt.title('%s' % (i+1))
plt.show()
```

```
<ipython-input-27-dbe573ab2b29>:6: MatplotlibDeprecationWarning: Auto-removal
  plt.subplot(4, 9, i+1)
```

The function below is defined for plotting face images, and we will utilize it for creating multiple plots with different embedding techniques.

```
def plot_faces(R,title,subx=4,suby=9,figsize=(25,8)):
  plt.figure(figsize=figsize)
  plt.suptitle(title, fontsize=18)
  i = 0
  for r in R:
      plt.axis('off')
      plt.subplot(subx, suby, i+1)
      i+=1
      plt.imshow(mat['Y'][:,:,r])
      plt.title('%s' % (r+1))
  plt.show()
```

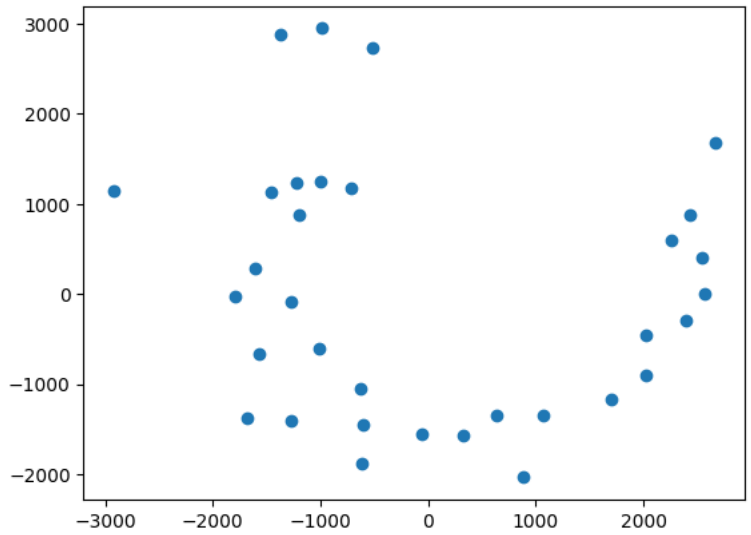
Now, we will examine the MDS embedding based on the top two eigen vectors.

```
mds = MDS(n_components=2,max_iter=500, n_init=1)
X_transform_mds = mds.fit_transform(X)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/manifold/_mds.py:299: FutureWarning: The default value of `normalized_st
  warnings.warn(
```

We proceed to create a scatter plot for the matrix transformed through MDS embedding.

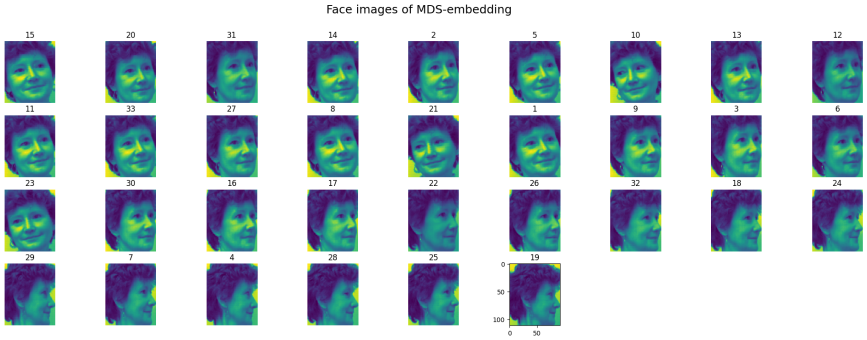
```
plt.scatter(X_transform_mds[:,0],X_transform_mds[:,1])
plt.show()
```



Below code generates a plot of facial ordering based on the first eigenvector, resulting in the following facial order.

```
R = X_transform_mds[:,0].argsort()
plot_faces(R,"Face images of MDS-embedding")
```

```
<ipython-input-5-49145cffdca9>:7: MatplotlibDeprecationWarning: Auto-removal o
  plt.subplot(subx, suby, i+1)
```

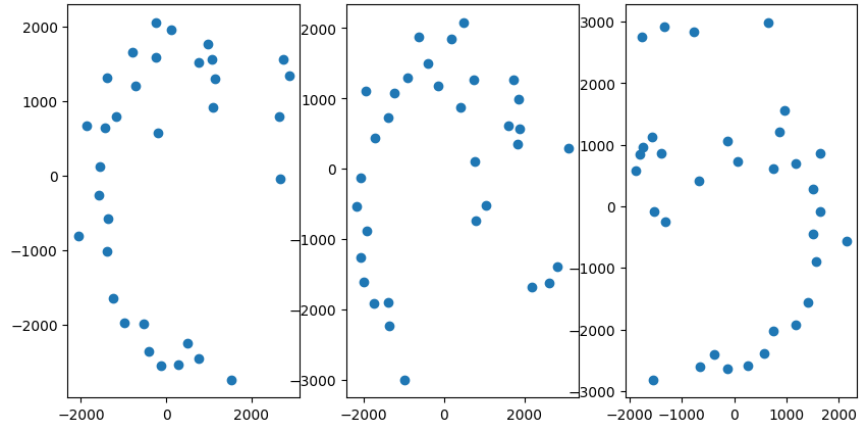


Multiple faces appear to be incorrectly positioned by MDS. Given the high instability of MDS, we generate three additional maps and present the faces in their correct order in each of these maps

```
mdsTransformArray = [MDS(n_components=2,max_iter=500, n_init=1).fit_transform(X) for i in range(3)]
Rarray = [mdsTransformArray[i][:,0].argsort() for i in range(3)]
plt.figure(figsize=(10,5))
for i in range(3):
```

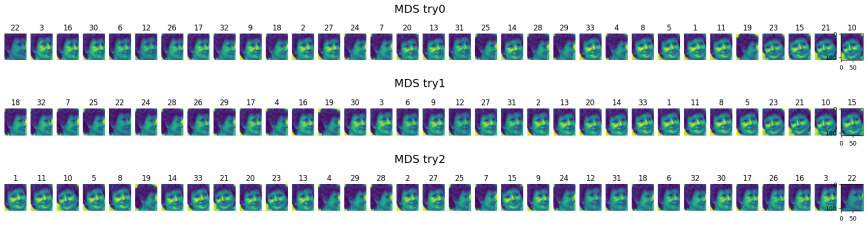
```
  plt.subplot(1,3,i+1)
  plt.scatter(mdsTransformArray[i][:,0],mdsTransformArray[i][:,1])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/manifold/_mds.py:299: FutureWa
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/manifold/_mds.py:299: FutureWa
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/manifold/_mds.py:299: FutureWa
  warnings.warn(
```



```
for i in range(3):
  plot_faces(Rarray[i],f"MDS try"+str(i),1,33,(25,2))
```

```
<ipython-input-5-49145cffdca9>:7: MatplotlibDeprecationWarning: Auto-removal o
  plt.subplot(subx, suby, i+1)
```

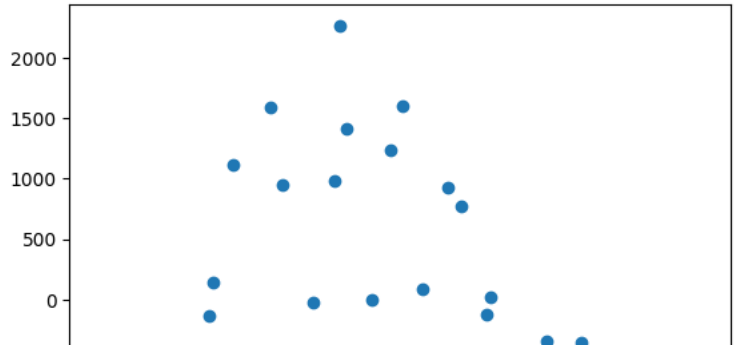


Based on the preceding information, it's evident that MDS-embedding doesn't yield satisfactory results since it relies on Euclidean distances for similarity calculations.

Now, we'll investigate the ISOMAP-embedding of the 33 faces using a graph with the k = 5 nearest neighbors.

```
isomapTransformer = Isomap(n_neighbors=5, n_components=2,max_iter=5000)
X_Isomap_transform = isomapTransformer.fit_transform(X)
R_isomap_order = X_Isomap_transform[:,0].argsort()
```

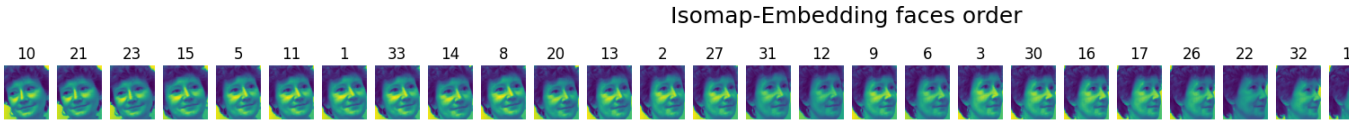
```
plt.scatter(X_Isomap_transform[:,0],X_Isomap_transform[:,1])
plt.show()
```



Faces in the order:

```
plot_faces(R_isomap_order,"Isomap-Embedding faces order",1,33,(25,2))
```

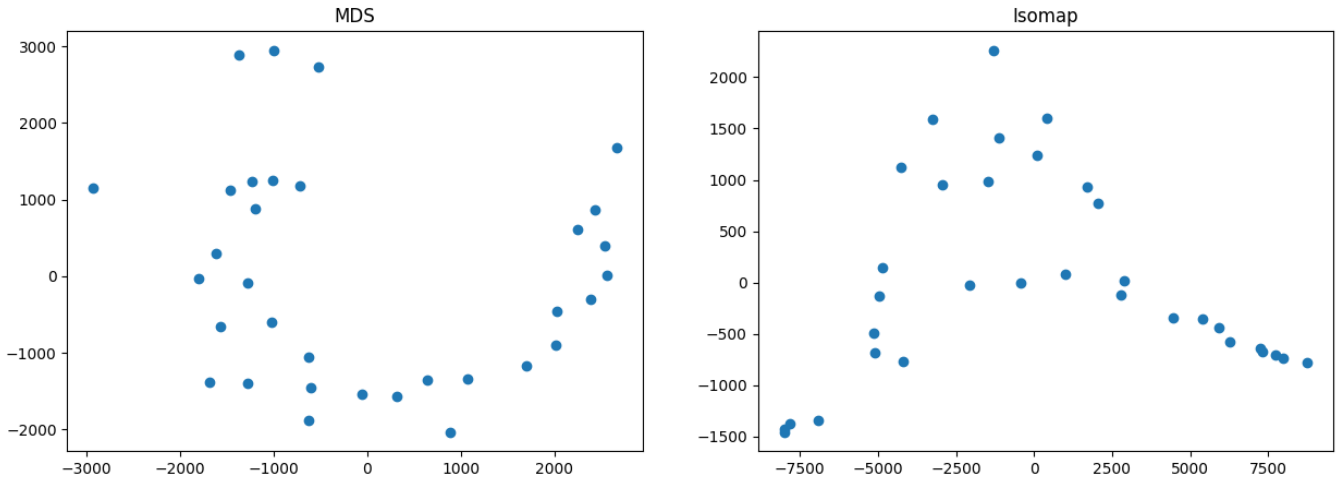
```
<ipython-input-5-49145cffdca9>:7: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6
  plt.subplot(subx, suby, i+1)
```



The facial ordering is evident from the information provided above.

Comparison of Isomap vs. Multidimensional Scaling (MDS)

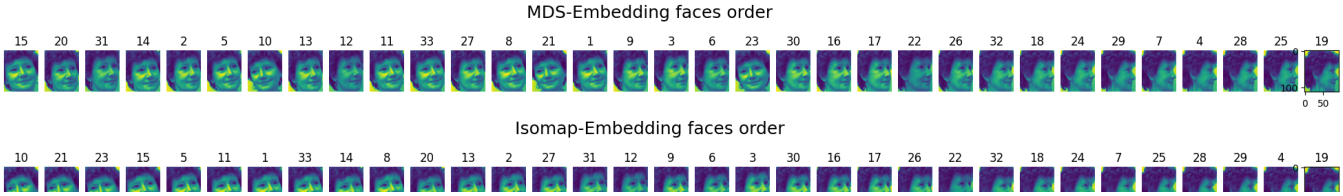
```
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.scatter(X_transform_mds[:,0],X_transform_mds[:,1])
plt.title("MDS")
plt.subplot(1,2,2)
plt.scatter(X_Isomap_transform[:,0],X_Isomap_transform[:,1])
plt.title("Isomap")
plt.show()
```




```
plot_faces(R,"MDS-Embedding faces order",1,33,(25,2))
```

```
plot_faces(R_isomap_order,"Isomap-Embedding faces order",1,33,(25,2))
```

```
<ipython-input-5-49145cffdca9>:7: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6
  plt.subplot(subx, suby, i+1)
```

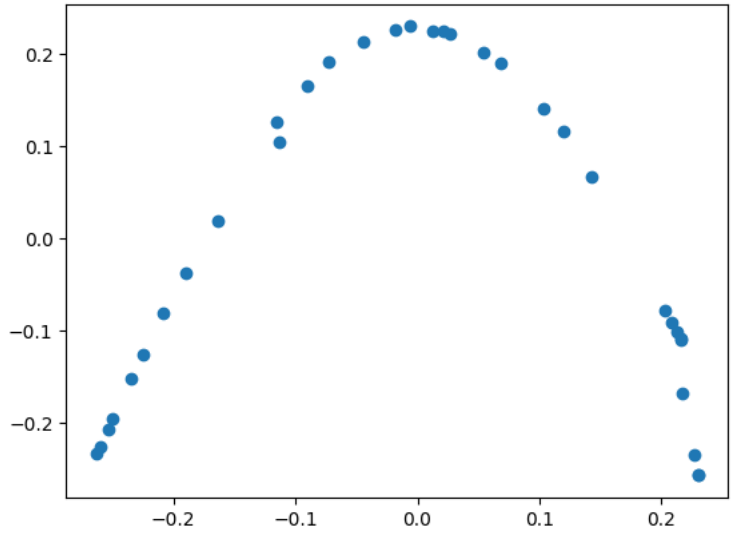


Based on the comparison, we observe that the MDS embedding produces inaccurate face orderings, whereas Isomap offers a superior ordering. This improvement can be attributed to Isomap's use of graph distances as opposed to the Euclidean metric for dissimilarity.

Now, let's investigate the Locality Linear Embedding (LLE) of the 33 faces within a k = 5 nearest neighbor graph.

```
LLETransformer = LocallyLinearEmbedding(n_neighbors=5)
X_LLE_transform = LLETransformer.fit_transform(X)
R_LLE_order = X_LLE_transform[:,0].argsort()
```

```
plt.scatter(X_LLE_transform[:,0],X_LLE_transform[:,1])
plt.show()
```



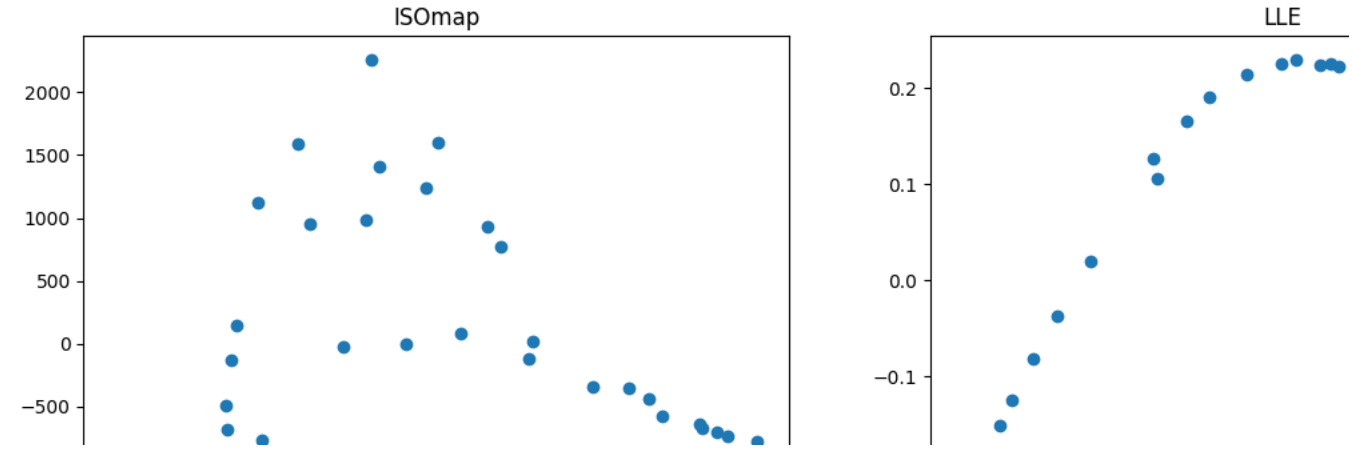
Faces in the order:

```
plot_faces(R_LLE_order,"LLE-Embedding faces order",1,33,(25,2))
```

```
<ipython-input-5-49145cffdca9>:7: MatplotlibDeprecationWarning: Auto-removal o
  plt.subplot(subx, suby, i+1)
```

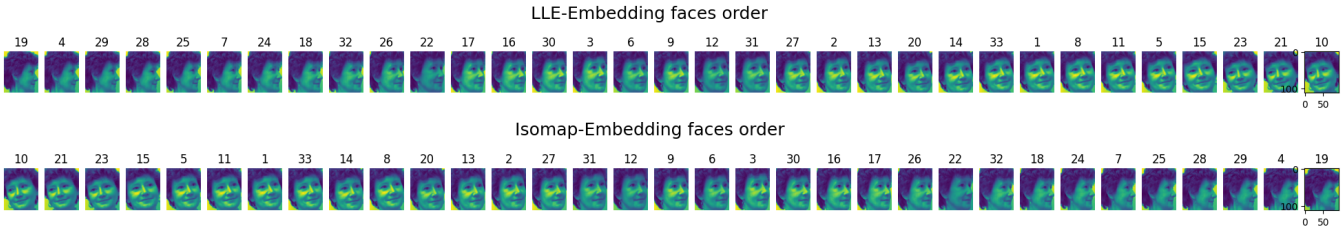
Comparison: LLE vs Isomap

```
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.scatter(X_Isomap_transform[:,0],X_Isomap_transform[:,1])
plt.title("ISOmap")
plt.subplot(1,2,2)
plt.scatter(X_LLE_transform[:,0],X_LLE_transform[:,1])
plt.title("LLE")
plt.show()
```




```
plot_faces(R_LLE_order,"LLE-Embedding faces order",1,33,(25,2))
plot_faces(R_isomap_order,"Isomap-Embedding faces order",1,33,(25,2))
```

```
<ipython-input-5-49145cffdca9>:7: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6
  plt.subplot(subx, suby, i+1)
```

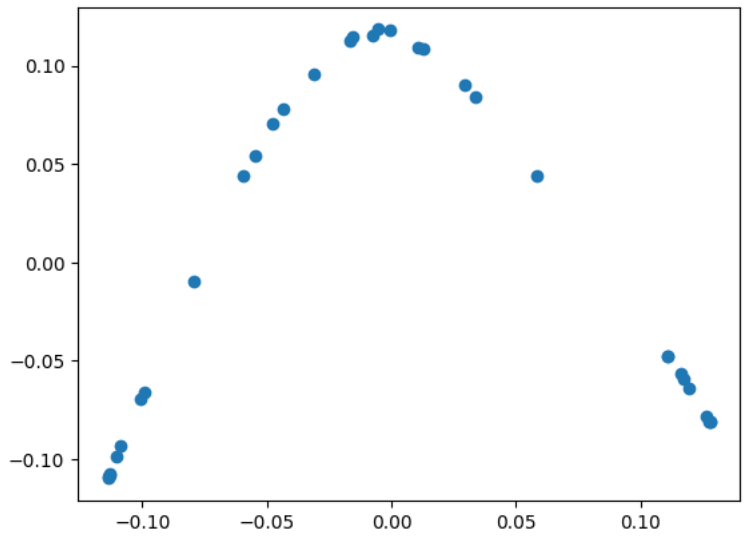


Based on the observations above, it is evident that LLE exhibits a more gradual curve compared to Isomap. Additionally, both methods correctly arrange the faces in the desired order.

Let's delve into the Laplacian Eigenmap (LE) embedding of the 33 faces, focusing on the graph with the 5 nearest neighbors.

```
LETransformer = SpectralEmbedding(n_neighbors=5)
X_LE_transform = LETransformer.fit_transform(X)
R_LE_order = X_LE_transform[:,0].argsort()
```

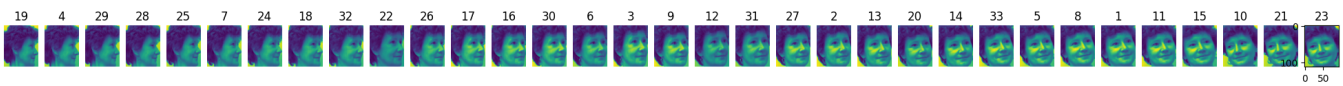
```
plt.scatter(X_LE_transform[:,0],X_LE_transform[:,1])
plt.show()
```



Faces in the order:

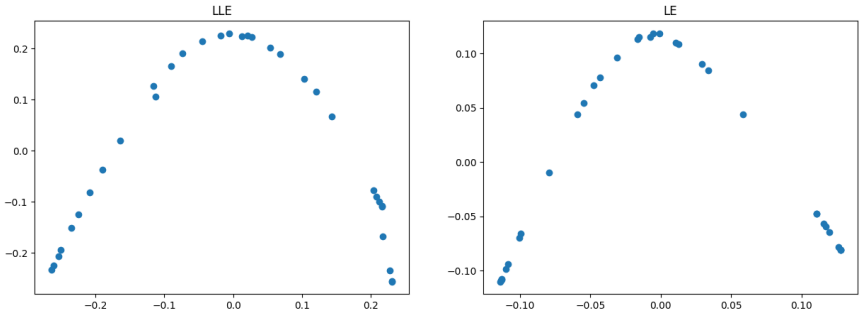
```
plot_faces(R_LE_order,"LE-Embedding faces order",1,33,(25,2))
```

```
<ipython-input-5-49145cffdca9>:7: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6
  plt.subplot(subx, suby, i+1)
```



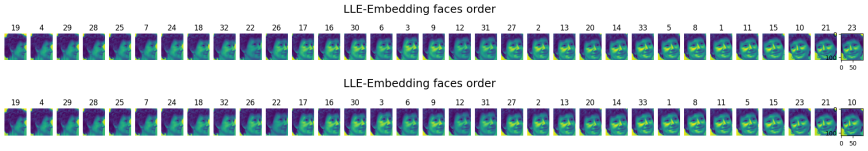
Comparing LLE and Isomap

```
plt.figure(figsize=(15,5))

plt.subplot(1,2,1)
plt.scatter(X_LLE_transform[:,0],X_LLE_transform[:,1])
plt.title("LLE")

plt.subplot(1,2,2)
plt.scatter(X_LE_transform[:,0],X_LE_transform[:,1])
plt.title("LE")

plt.show()
```




```
plot_faces(R_LE_order,"LLE-Embedding faces order",1,33,(25,2))
plot_faces(R_LLE_order,"LLE-Embedding faces order",1,33,(25,2))
```

```
<ipython-input-5-49145cffdca9>:7: MatplotlibDeprecationWarning: Auto-removal o
  plt.subplot(subx, suby, i+1)
```



Based on the information above, it is evident that Local Linear Embedding (LE) exhibits a notably smoother curve compared to Locally Linear Embedding (LLE). Furthermore, both methods successfully arrange the faces in the correct order.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
from sklearn.random_projection import GaussianRandomProjection, johnson_lindenstrauss_min_dim
from sklearn.metrics.pairwise import euclidean_distances
```

We create a data matrix X with parameters n = 10 and d = 5000, using the identity matrix of size d x d as a basis.

```
matrix = np.identity(5000)
index = np.random.choice(matrix.shape[1], 10, replace=False)
X = matrix[index,:]
```

Set a constant value of ε to 0.1, and determine the embedding dimension, denoted as 'm,' by applying the relevant formula.

```
n = 10
e = 0.1
denomi = (e**2 / 2) - (e**3 / 3)
m = 4*(np.log(n))/denomi
m.astype(np.int64)
```

```
1973
```

Verifying the outcomes using the "johnson_lindenstrauss_min_dim" package.

```
johnson_lindenstrauss_min_dim(n_samples=n, eps=0.1)
```

```
1973
```

We will create a random projection matrix A with dimensions m × d.

```
gaussianProjector = GaussianRandomProjection(n_components='auto',eps=0.1)
A = gaussianProjector.fit_transform(X)
```

Below verifies the shape of the resulting projection matrix

```
A.shape
```

```
(10, 1973)
```

We proceed to calculate the squared pairwise distances between pairs of vectors u and v, denoted as $\|u - v\|^2$, as well as the squared pairwise distances between their projections, denoted as $\|Au - Av\|^2$. Subsequently, we determine the absolute difference between these two sets of distances.

```
X_dist = euclidean_distances(X)
A_dist = euclidean_distances(A)
A_X_diff = abs(X_dist - A_dist)
```

Below function returns the validtity of the Johnson-Lindenstrauss Lemma by monitoring the inequality

---

$(1-\varepsilon)\ \|u-v\|2 \leq \|Au-Av\|2 \leq (1+\varepsilon)\|u-v\|2$

```
def JohnsonLindenstraussLemmaValidity(initial_matrix_dist,projected_matrix_dist,e):
  flag = True
  for i in range(0,len(initial_matrix_dist)):
    for j in range(0,len(initial_matrix_dist[0])):
      if ((1-e)*initial_matrix_dist[i][j] <= projected_matrix_dist[i][j] <= (1+e)*initial_matrix_dist[i][j]):
        continue
      else:
        flag = False
        break
  return "Johnson Lindenstrauss Lemma is valid" if flag else "Johnson Lindenstrauss Lemma is invalid"
```

We now check if the before calculated distance is valid with Johnson-Lindenstrauss Lemma

```
print(JohnsonLindenstraussLemmaValidity(X_dist,A_dist,e))
```

```
Johnson Lindenstrauss Lemma is valid
```

We iterate through the previously mentioned steps, progressively increasing the value of 'd' by a factor of 2 in each iteration while keeping 'm' and 'n' constant. This is done to assess the applicability of the Johnson-Lindenstrauss Lemma.

```
e = 0.1
denomi = (e**2 / 2) - (e**3 / 3)
m = 4*(np.log(n))/denomi
n = 10
d = 5000

while True:
  d *= 2
  matrix = np.identity(d)
  index = np.random.choice(matrix.shape[1], 10, replace=False)
  X = matrix[index,:]
  gaussianProjector = GaussianRandomProjection(n_components='auto',eps=0.1)
  A = gaussianProjector.fit_transform(X)
  X_dist = euclidean_distances(X)
  A_dist = euclidean_distances(A)
  A_X_diff = abs(X_dist - A_dist)
  print(JohnsonLindenstraussLemmaValidity(X_dist,A_dist,e),"for d=",d)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-306f4d23ad22> in <cell line: 7>()
     15   A_dist = euclidean_distances(A)
     16   A_X_diff = abs(X_dist - A_dist)
---> 17   print(JohnsonLindenstraussLemmaValidity(X_dist,A_dist,e),"for d=",d)

NameError: name 'JohnsonLindenstraussLemmaValidity' is not defined
```

SEARCH STACK OVERFLOW

The provided code encountered an error and crashed during its 5th iteration. The error message indicated that the session ran out of available RAM. If you require access to high-RAM runtimes, you might consider exploring Colab Pro as an option.

We iterate through the initial steps, incrementing 'n' by a factor of 2 while keeping 'd' constant. In each iteration, we calculate 'm' and verify its compliance with the Johnson-Lindenstrauss Lemma.

```
e = 0.1
denomi = (e**2 / 2) - (e**3 / 3)
n = 10
d = 5000

while True:
  n *= 2
  m = 4*(np.log(n))/denomi
  matrix = np.identity(d)
  index = np.random.choice(matrix.shape[1], n, replace=False)
  X = matrix[index,:]
  gaussianProjector = GaussianRandomProjection(eps=0.1)
  A = gaussianProjector.fit_transform(X)
  X_dist = euclidean_distances(X)
  A_dist = euclidean_distances(A)
  A_X_diff = abs(X_dist - A_dist)
  print(JohnsonLindenstraussLemmaValidity(X_dist,A_dist,e),"for constant d=",d,"and n = ",n,"m calculated = ",m)
```

```
Johnson Lindenstrauss Lemma is valid for constant d= 5000 and n =  20 m calcu
Johnson Lindenstrauss Lemma is valid for constant d= 5000 and n =  40 m calcu
Johnson Lindenstrauss Lemma is valid for constant d= 5000 and n =  80 m calcu
Johnson Lindenstrauss Lemma is valid for constant d= 5000 and n =  160 m calcu
Johnson Lindenstrauss Lemma is valid for constant d= 5000 and n =  320 m calcu
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-b0f6f7947fc0> in <cell line: 6>()
```

Based on the information provided, it's evident that the Johnson-Lindenstrauss Lemma holds true in all instances, subject to the constraint of error.

```
     15   A_dist = euclidean_distances(A)

 ---------------------------------------- 2 frames ----------------------------------------
/usr/local/lib/python3.10/dist-packages/sklearn/random_projection.py in
fit(self, X, y)
    388
    389             elif self.n_components_ > n_features:
--> 390                 raise ValueError(
    391                     "eps=%f and n_samples=%d lead to a target
dimension of "
    392                     "%d which is larger than the original space with
"

ValueError: eps=0.100000 and n_samples=640 lead to a target dimension of 5538
which is larger than the original space with n_features=5000
```