Problem 1:

- 2

Problem 2:

- (i, j, k) →(0, 0, 1/sqrt(2))

Problem 3:

- PC2

Problem 4:

- 3

Problem 5:

- 2,3,4

M12-HW1

December 4, 2023

1 Problem 1

1.1 Problem Description

In this problem you will use PCA and TSNE to apply dimensionality reduction to 64x64 images of signed distance fields (SDFs) on parts belonging to 8 different classes. Each class is topologicaly similar, with some variation in void size and shape. These signed distance fields are helpful in the prediction of internal stress fields in the parts. You will also apply KNN to predict the class of the part with the reduced space.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

- 3x8 subplot visualization of the first 3 samples from each of the 8 classes
- Bar plot of the variance explained for the first 25 PCs and the number of PCs required to explain $> 90\%$ of the variance in the training data
- 4x8 subplot visualization of reconstructed samples using 3, 10, 50 and all PCs on the first sample from each of the 8 classes in the test set
- Test accuracy of KNN classifier trained on the 3D, 10D, and 50D PCA reduced feature spaces
- Plot of the 2D TSNE reduced feature space
- Test accuracy of the KNN classifier trained on the 2D TSNE reduced feature space
- Discussion questions 1 and 2

Imports and Utility Functions:

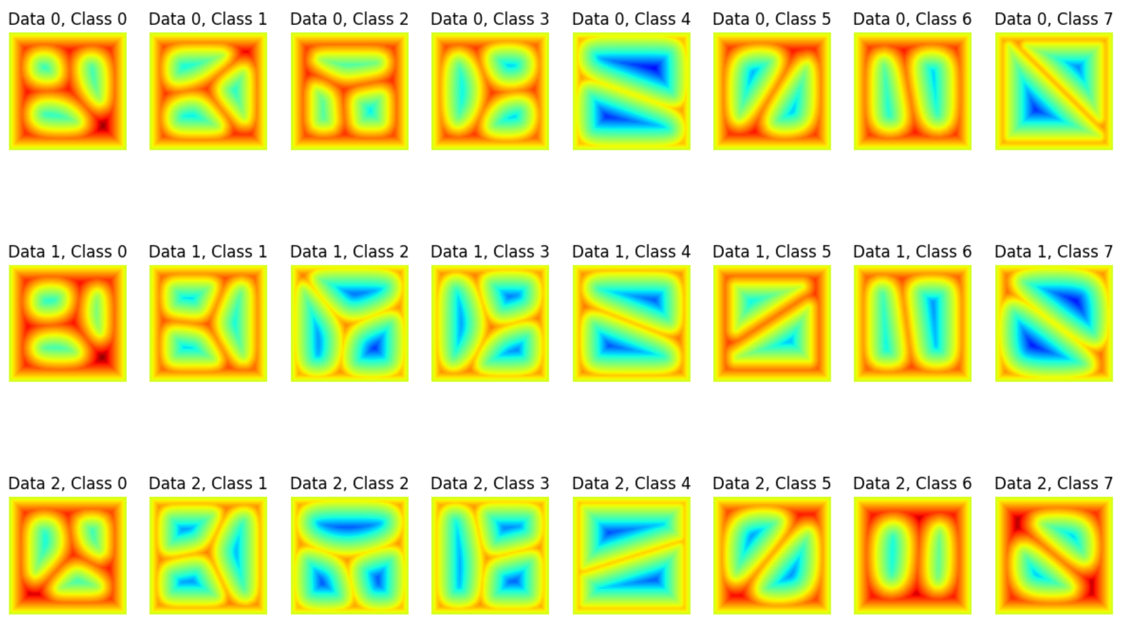
[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import io

from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

def dataLoader(filepath):
    # Load and flatten the SDF dataset
    mat = io.loadmat(filepath)
```

```python
    data = []
    for i in range(800):
        sdf = mat["sdf"][i][0].T
        data.append(sdf.flatten())
    data = np.vstack(data)
    # Assign labels
    labels = np.repeat(np.arange(8), 100)
    return data, labels

def plot_sdf(data, ax = None, title = None):
    # If no axes, make them
    if ax is None:
        ax = plt.gca()
    # Reshape image data into square
    sdf = data.reshape(64,64)
    # Plot image, with bounds of the SDF values for the entire dataset
    ax.imshow(sdf, vmin=-0.31857, vmax=0.206349, cmap="jet")
    ax.axis('off')
    # If there is a title, add it
    if title:
        ax.set_title(title)
```

1.2 Visualization

Using the provided `dataLoader()` function, load the data and labels from `sdf_images.mat`. The returned data will contain 800 samples, with 4096 features. Then, using the provided `plot_sdf()` function, generate a 3x8 subplot figure containing visualizations of the first 3 SDFs in each class.

[ ]:
```python
data, labels = dataLoader("data/sdf_images.mat")

fig, ax = plt.subplots(3,8)
fig.set_size_inches(15,9)
for i in range(8):
    idxs = np.where(labels == i)[0][:3]
    plot_sdf(data[idxs[0],:], ax[0][i], f"Data 0, Class {i}")
    plot_sdf(data[idxs[1],:], ax[1][i], f"Data 1, Class {i}")
    plot_sdf(data[idxs[2],:], ax[2][i], f"Data 2, Class {i}")
```

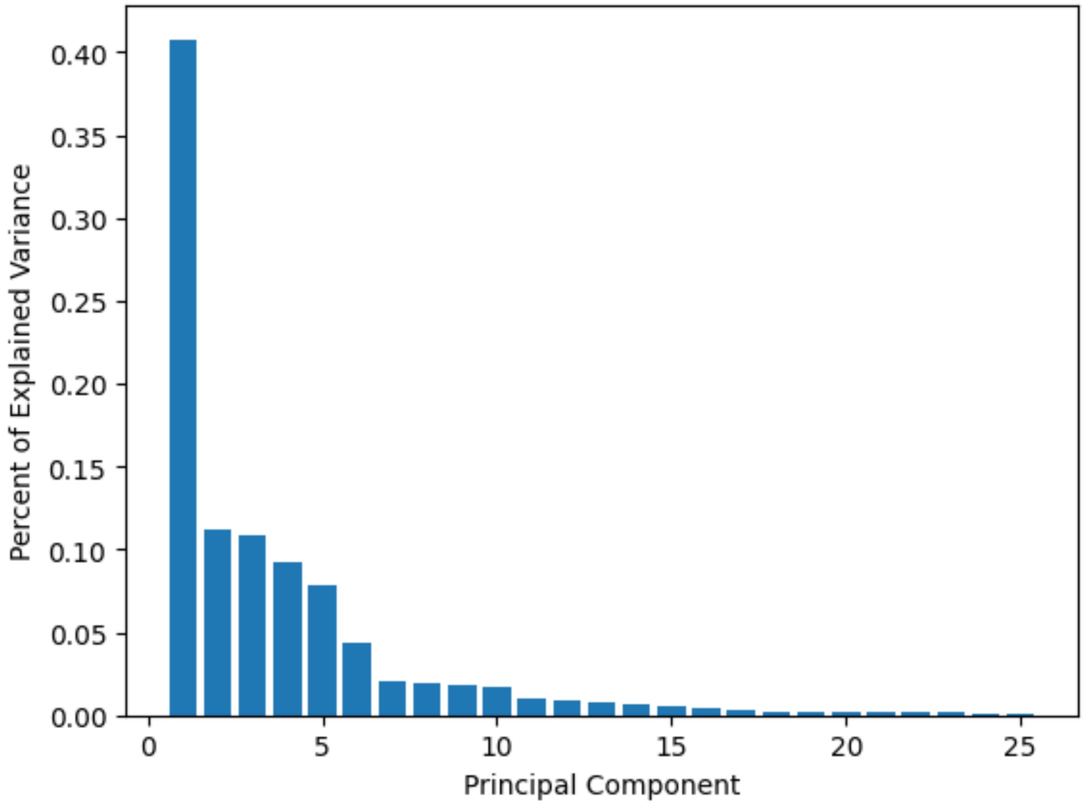


1.3 Explained Variance

Use `train_test_split()` to partition the data and labels into a training and test set with `test_size = 0.2` and `random_state = 0`. Then train a PCA model on the training data and generate a bar plot of the variance explained for the first 25 principal components. Determine the number of principal components required to explain $> 90\%$ of the variance in the training data.

[ ]:
```
X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.
  ↪2, random_state=0)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

pca = PCA(n_components=25)
pca.fit(X_train)
A = pca.transform(X_train)
X_recon = pca.inverse_transform(A)

plt.figure()
plt.bar(1+np.arange(25),pca.explained_variance_ratio_)
plt.xlabel("Principal Component")
plt.ylabel("Percent of Explained Variance")
plt.show()

s = 0
for i,percentage in enumerate(pca.explained_variance_ratio_):
    s += percentage
    if s > 0.9:
        print(f"{i} Components are needed to explain > 90% of the variance.")
        break
```

```
(640, 4096)
(160, 4096)
(640,)
(160,)
```



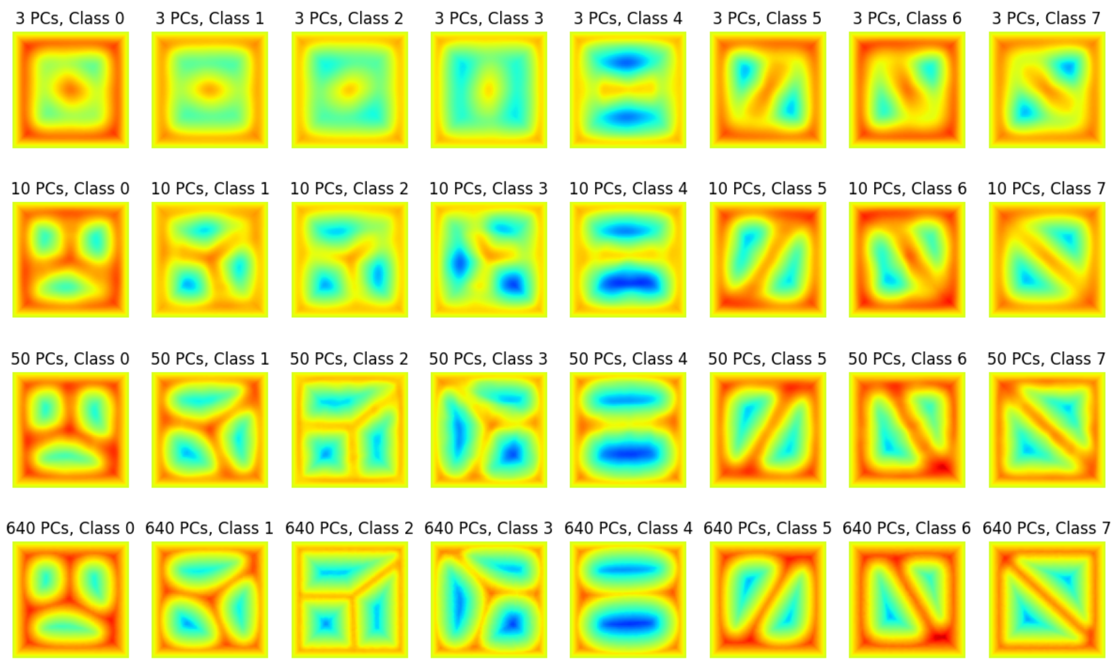
```
8 Components are needed to explain > 90% of the variance.
```

1.4 PCA Reconstruction

Using the training data, generate 4 PCA models using 3, 10, 50, and all of the principal components. Use these models to transform the test data into the reduced space, and then reconstruct the data from the reduced space. Plot the reconstruction for each model, on the first occurence of each class in the test set. Your generated plot should be a 4x8 subplot figure, with each subplot title containing the class and the number of PCs used.

```
[ ]: pcas = []
     fig, ax = plt.subplots(4,8)
```

```
fig.set_size_inches(15,9)
for i,n_components in enumerate([3, 10, 50, len(X_train[:,0])]):
    pca = PCA(n_components=n_components)
    pca.fit(X_train)
    A = pca.transform(X_test)
    X_recon = pca.inverse_transform(A)
    for j in range(8):
        idxs = np.where(y_test == j)[0]
        plot_sdf(X_recon[idxs[0],:], ax[i][j], f"{n_components} PCs, Class {j}")

    pcas.append(pca)
```



1.5 KNN on PCA Reduced Data

Now train a KNN classifier to predict the class of the 3D, 10D, and 50D PCA reduced data. You should train the KNN on the reduced training data, and report the prediction accuracy on the test set. You will also need to determine the `n_neighbors` parameter for your KNN classifier that gives good results.

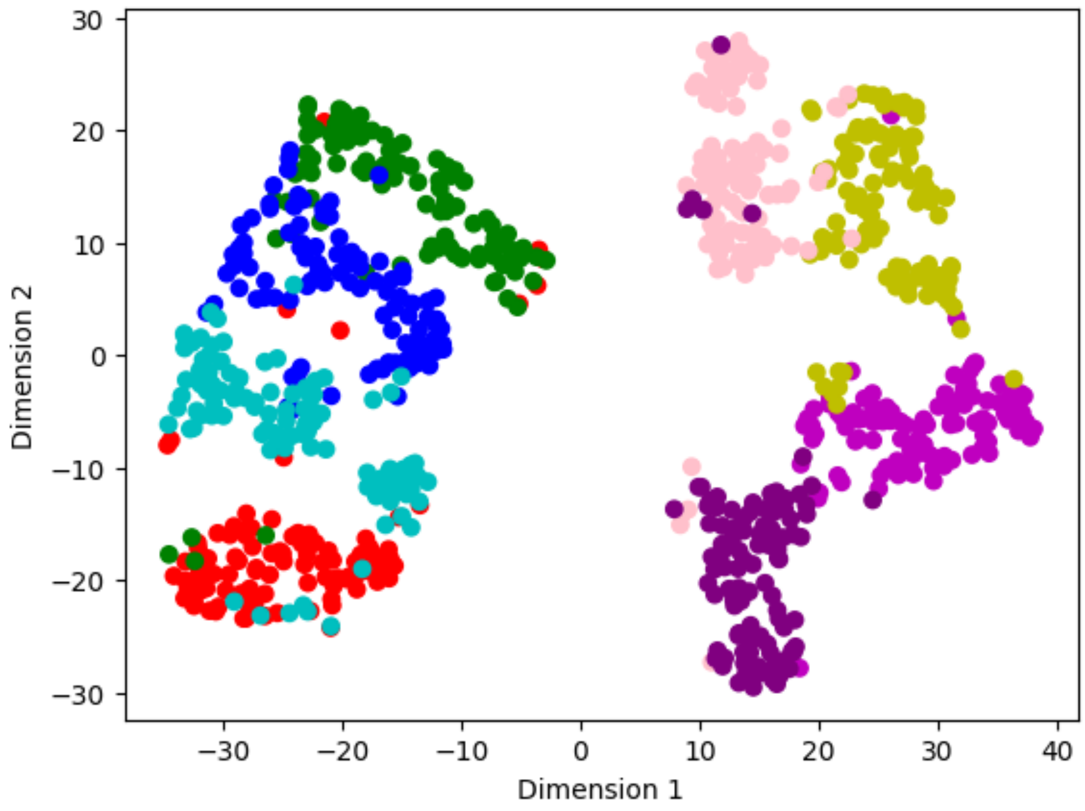
```
[ ]: for pca,name in zip(pcas[:-1],["3D", "10D", "50D"]):
         A_train = pca.transform(X_train)
         knn = KNeighborsClassifier(n_neighbors=15).fit(A_train, y_train)
         A_test = pca.transform(X_test)
         y_pred = knn.predict(A_test)
         acc = np.sum(y_pred == y_test)/len(y_test) * 100
         print(f"{name} PCA Classifier KNN Test Accuracy: {acc}%")
```

```
3D PCA Classifier KNN Test Accuracy: 70.625%
10D PCA Classifier KNN Test Accuracy: 90.0%
50D PCA Classifier KNN Test Accuracy: 93.125%
```

1.6 TSNE Visualization

First reduced the full dataset to 50D using PCA, and then further reduced the data to 2D using TSNE. Plot the 2D reduced feature space with a scatter plot, coloring each point according to its class.

[ ]:
```
A_50D = pcas[2].transform(data)
A_2D = TSNE(2,perplexity=30,random_state=123).fit_transform(A_50D)

c = ['r', 'g', 'b', 'c', 'm', 'y', 'pink', 'purple']
colors = [c[i] for i in labels]
plt.figure()
plt.scatter(A_2D[:,0], A_2D[:,1],color=colors)
plt.xlabel("Dimension 1")
plt.ylabel("Dimension 2")
plt.show()
```



1.7 KNN on PCA/TSNE Reduced Data

Using the same 2D PCA/TSNE data, split the data into train and test data and labels using `train_test_split` with a `random_state = 0` parameter so you have the same train/test partition as before. Then, train a KNN on this 2D feature space with the training set, and report the KNN classifier accuracy on the test set. Again, you will need to determine the `n_neighbors` parameter in the KNN classifier that gives good results.

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(A_2D, labels, test_size=0.
      ↪2, random_state=0)

     knn = KNeighborsClassifier(n_neighbors=15).fit(X_train, y_train)
     y_pred = knn.predict(X_test)
     acc = np.sum(y_pred == y_test)/len(y_test) * 100
     print(f"Reduced PCA/TSNE KNN Classifier Test Accuracy: {acc}%")
```

```
Reduced PCA/TSNE KNN Classifier Test Accuracy: 91.25%
```

1.8 Discussion

1. Discuss how the number of principal components relates to the quality of reconstruction of the data. Using all of the principal components, should there be any error in the reconstruction of a sample from the training data? What about in the reconstruction of an unseen sample from the testing data?
2. Discuss how you determined `k`, the number of neighbors in your KNN models. Why do we perform dimensionality reduction to our data before feeding it to our KNN classifier?

Your response goes here

1. As the number of principle components increases, the final reconstructed image approaches the original image. This continues until the number of principle components reaches the maximum at which the reconstructed image should match exactly to the original image. For data in the test set, the reconstructed image should match fairly closely to the original image with the maximum principal components but shouldn't match perfectly.
2. K was adjusted until the test accuracy of the models was maximized across all trained models. So for the 3 trained together, the number of neighbors was tuned to have good performance across all 3 models.

M12-L1-P1

December 4, 2023

1 M12-L1 Problem 1

This problem is intended to demonstrate PCA on a small 2D dataset. This will emphasize how PCs are computed and what they mean.

```
[ ]: import numpy as np
     import matplotlib.pyplot as plt

     X = np.array([[2.5, 2.4],[0.5, 0.7],[2.2, 2.9],[1.9, 2.2],[3.1, 3. ],
                   [2.3, 2.7],[2., 1.6],[1., 1.1],[1.5, 1.6],[1.1, 0.9]])
```

1.1 Computing the Principal Components

First, compute the principal components of the dataset by following these steps: 1. Compute `M` (1×2), the mean of each dimension in `X` 2. Compute `S` (2×2), the covariance matrix of `X` (see `np.cov`) 3. Report `w`, the 2 eigenvalues of `S` (see `np.linalg.eig`) 4. Get `e1` and `e2`, the eigenvectors corresponding to the elements of `w`

The principal components in this problem are then `e1` and `e2`.

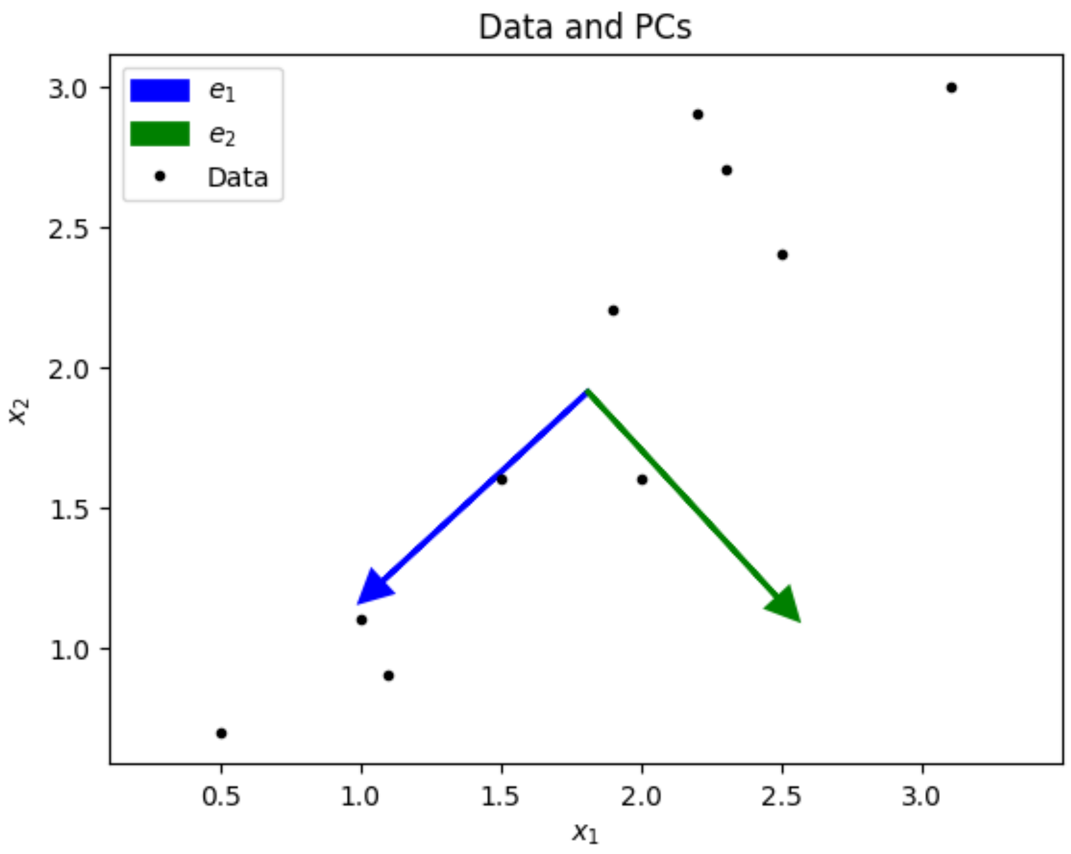
```
[ ]: print('X:\n', X)

     M = np.mean(X, axis=0)
     print('\nMean of each dimension:\n', M)

     S = np.cov(X.T)
     print('\nCovariance Matrix:\n', S)


     w = np.linalg.eig(S)[0]
     print('\nEigenvalues of covariance matrix:\n',w)

     e1,e2 = np.linalg.eig(S)[1]
     print('\nPrincipal Components:')
     print('e1:',e1)
     print('e2:',e2)
```

```
X:
 [[2.5 2.4]
 [0.5 0.7]
 [2.2 2.9]
 [1.9 2.2]
 [3.1 3. ]
 [2.3 2.7]
 [2.  1.6]
 [1.  1.1]
 [1.5 1.6]
 [1.1 0.9]]

Mean of each dimension:
 [1.81 1.91]

Covariance Matrix:
 [[0.61655556 0.61544444]
 [0.61544444 0.71655556]]

Eigenvalues of covariance matrix:
 [0.0490834  1.28402771]

Principal Components:
e1: [-0.73517866 -0.6778734 ]
e2: [ 0.6778734  -0.73517866]
```

1.2 Plotting data with principal components

Complete the code below to plot the original data with principal components represented as unit vector arrows.

[ ]:
```
plt.figure()
plt.title("Data and PCs")

e1, e2 = e1.flatten(), e2.flatten()
plt.arrow(M[0],M[1],e1[0],e1[1], color="blue", linewidth=2, head_width=0.1,␣
 ↪head_length=0.1, label="$e_1$")
plt.arrow(M[0],M[1],e2[0],e2[1], color="green", linewidth=2, head_width=0.1,␣
 ↪head_length=0.1, label="$e_2$")
plt.plot(X[:,0],X[:,1],'.',color="black", label="Data")

plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.legend()
plt.axis("equal")
plt.show()
```



1.3 Plotting transformed data

Now, transform the data with the formula $a_i = (x - \mu) \bullet e_i$.

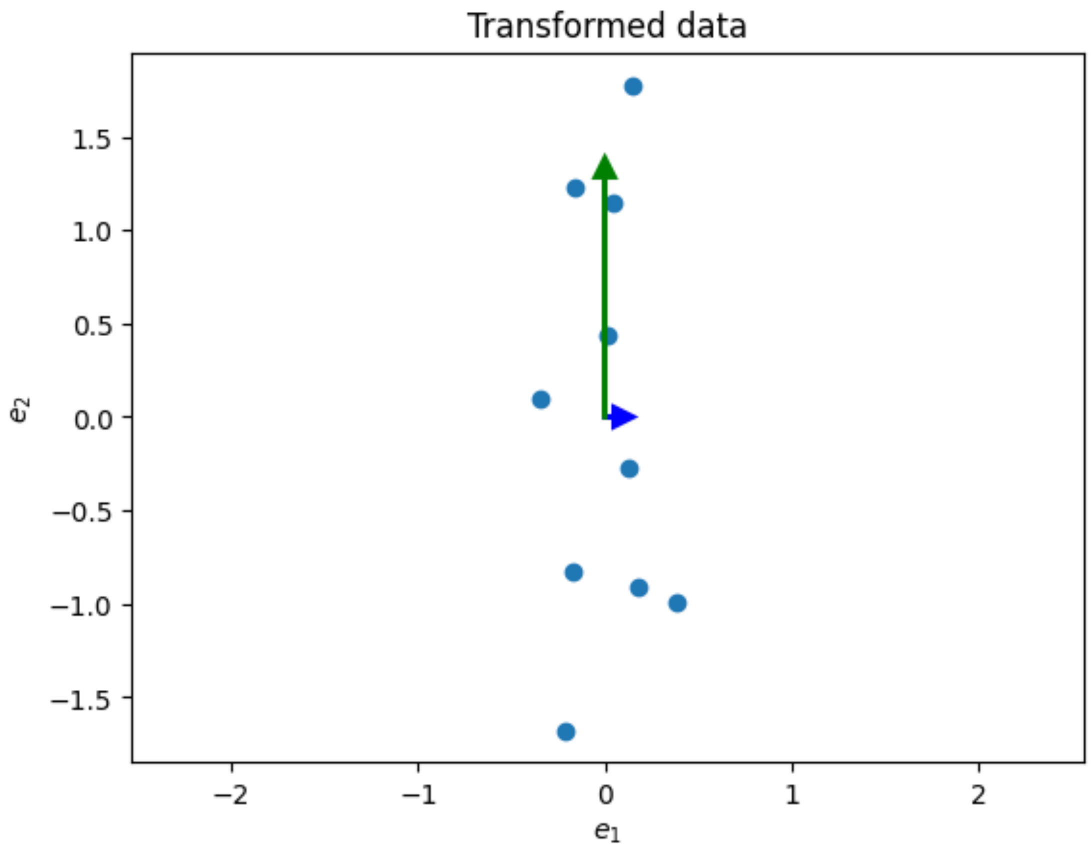
Print the transformed data matrix columns `a1` and `a2`.

Then plot the transformed data on $e_1 - e_2$ axes.

```
[ ]: a1, a2 = ((X - M) @ np.array([e1.T, e2.T])).T

     print("a_1 = ",a1)
     print("a_2 = ",a2)

     plt.figure()
     plt.title("Transformed data")

     e1, e2 = e1.flatten(), e2.flatten()
     plt.scatter(a1, a2)
     plt.arrow(0,0,w[0],0, color="blue", linewidth=2, head_width=0.1, head_length=0.
      ↪1, label="$e_1$")
```

```
plt.arrow(0,0,0,w[1], color="green", linewidth=2, head_width=0.1, head_length=0.
↪1, label="$e_2$")

plt.xlabel("$e_1$")
plt.ylabel("$e_2$")
plt.axis("equal")
plt.show()
```

```
a_1 =  [-0.17511531  0.14285723  0.38437499  0.13041721 -0.20949846  0.17528244
 -0.3498247   0.04641726  0.01776463 -0.16267529]
a_2 =  [-0.82797019  1.77758033 -0.99219749 -0.27421042 -1.67580142 -0.9129491
  0.09910944  1.14457216  0.43804614  1.22382056]
```



M12-L2-P1

December 4, 2023

1 M12-L1 Problem 2

Sometimes the dimensionality is greater than the number of samples. For example,in this problem X has 19 features, but there are only 4 data points. You will need to use the alternate PCA formulation in this case. Follow the steps in the cells below to implement this method.

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error

X = np.array([  [-2,  1,  2, -3,  4,  1,  0,  3,  0,  2,  1,  1,  2,  3, -2,␣
 ↪-3, 2,  1,  0],
                [ 1,  2, -4,  2, -4,  2,  5,  2,  2,  1, -3,  0,  0,  1, -2, ␣
 ↪1, 1, -3, -2],
                [ 1, -3,  2,  1,  0, -3, -5, -1,  3,  3, -2, -3, -2, -1,  1, ␣
 ↪0, 5,  4,  2],
                [ 3, -1,  0,  2,  2, -5, -4, -1,  2, -1,  3,  4,  4,  2,  1, ␣
 ↪2, -2,  1, -1]])
```

1.1 Computing Principal Components

1.1.1 The A matrix

First, you should compute the `A` matrix, where A is $(X - \mu)'$. (Note the transpose)

Print this matrix below. It should have size 19×4.

[ ]:
```python
A = (X - np.mean(X,axis=0)).T
print("A = \n", A)
```

```
A =
 [[-2.75  0.25  0.25  2.25]
 [ 1.25  2.25 -2.75 -0.75]
 [ 2.   -4.    2.    0.  ]
 [-3.5   1.5   0.5   1.5 ]
 [ 3.5  -4.5  -0.5   1.5 ]
 [ 2.25  3.25 -1.75 -3.75]
 [ 1.    6.   -4.   -3.  ]
 [ 2.25  1.25 -1.75 -1.75]
 [-1.75  0.25  1.25  0.25]
 [ 0.75 -0.25  1.75 -2.25]
 [ 1.25 -2.75 -1.75  3.25]
 [ 0.5  -0.5  -3.5   3.5 ]
 [ 1.   -1.   -3.    3.  ]
 [ 1.75 -0.25 -2.25  0.75]
 [-1.5  -1.5   1.5   1.5 ]
 [-3.    1.    0.    2.  ]
 [ 0.5  -0.5   3.5  -3.5 ]
 [ 0.25 -3.75  3.25  0.25]
 [ 0.25 -1.75  2.25 -0.75]]
```

1.1.2 "Small" covariance matrix

By transposing $X - \mu$ to get A, now we can compute a smaller covariance matrix with $A'A$. Compute this matrix, `C`, below and print the result.

[ ]:
```
C = A.T@A
print("C = \n", C)
```

```
C =
 [[ 69.875 -18.875 -26.375 -24.625]
 [-18.875 121.375 -53.125 -49.375]
 [-26.375 -53.125  98.375 -18.875]
 [-24.625 -49.375 -18.875  92.875]]
```

1.1.3 Finding nonzero eigenvectors

Next, find the useful (nonzero) eigenvectors of `C`.

For validation purposes, there should be 3 useful eigenvectors, and the first one is `[-0.06628148 -0.79038331 0.47285044 0.38381435]`.

Keep these eigenvectors in a 4×3 array `e`.

[ ]:
```
w,v = np.linalg.eig(C)
w,v = np.real(w), np.real(v)
idx = np.argsort(-w)
w,v = w[idx], v[:,idx]
e = v[:,0:3]
print(e)
```

```
[[-0.06628148  0.04124587 -0.86249959]
 [-0.79038331 -0.06822502  0.34733208]
 [ 0.47285044 -0.69123739  0.22046165]
 [ 0.38381435  0.71821654  0.29470586]]
```

1.1.4 Calculating "eigenfaces"

Now, we have all we need to compute `U`, the matrix of eigenfaces.

$\mathbf{U_i} = \mathbf{Ae_i}$

$(19 \times 3) = (19 \times 4)(4 \times 3)$

Compute and print U. Be sure to normalize your eigenvectors `e` before using the above equation.

```
[ ]: U = A @ e
     U /= np.linalg.norm(U, axis=0)
     print("Eigenfaces, U:\n",U)
```

```
Eigenfaces, U:
 [[ 0.07294372  0.12277459  0.33008441]
 [-0.26034151  0.11787331 -0.11677714]
 [ 0.29998485 -0.09606164 -0.27776956]
 [-0.01067529  0.04536213  0.42516696]
 [ 0.27653993  0.17530224 -0.44157072]
 [-0.37621372 -0.15082188 -0.23925816]
 [-0.59257956  0.02265222 -0.05657115]
 [-0.19897063 -0.0037123  -0.250194  ]
 [ 0.04569305 -0.07236581  0.20213547]
 [ 0.0084373  -0.25979087 -0.10504274]
 [ 0.18948616  0.35382298 -0.1518308 ]
 [ 0.00380575  0.46650428 -0.03585222]
 [ 0.03449119  0.40571147 -0.10256065]
 [-0.05241297  0.20419008 -0.19442141]
 [ 0.19396809  0.00756997  0.16057937]
 [ 0.01329023  0.11639359  0.36617258]
 [ 0.0508452  -0.45626561 -0.08985059]
 [ 0.3456779  -0.16842745 -0.07563409]
 [ 0.16171488 -0.18371276 -0.0569842 ]]
```

1.2 Projecting data into 3D

Now project your data into 3 dimensions with the formula:

$ = U^T A $

$(3 \times 4) = (3 \times 19)(19 \times 4)$

Call the projected data Ω "W". Print `W.T`

```
[ ]: W = U.T@A
     print('Projected data in 3 dimensions:\n',W.T)
```

```
Projected data in 3 dimensions:
 [[ -0.8782013    0.44099733  -8.3011616 ]
 [-10.47224127  -0.72945617   3.34291139]
 [  6.26506632  -7.39065157   2.12184196]
 [  5.08537624   7.67911041   2.83640825]]
```

1.3 Reconstructing data in 19-D

We can project the transformed data `W` back into the original 19-D space using:

$\Gamma_f = U\Omega + \Psi$
where:
$ _f = $ reconstructed data
$U = $ eigenfaces
$ = $ Reduced data
$ = $ Means

Do this, and compute the MSE between each reconstructed sample and corresponding original points. Report all 4 MSE values.

[ ]:
```
MSE = []
reconstructed = (U@W) + np.mean(X,axis=1)
mse = np.mean((X - reconstructed.T) **2, axis=1)
for i in range(4):
    print(f"MSE for sample {i}: {mse[i]}")
```

```
MSE for sample 0: 0.7468836565096959
MSE for sample 1: 0.716412742382272
MSE for sample 2: 0.7164127423822705
MSE for sample 3: 0.691481994459834
```

1.4 2-D Reconstruction

What if we had only used the first 2 eigenvectors to compute the eigenfaces? Below, redo the earlier calculations, but use only two eigenfaces. Compute the 4 MSE values that you would get in this case.

(You should get an MSE of 3.626 for the first sample.)

[ ]:
```
MSE2 = []
U2 = A @ e[:,:2]
U2 /= np.linalg.norm(U2, axis=0)
W2 = U2.T@A
reconstructed = (U2@W2) + np.mean(X,axis=1)
print("Using only 2 eigenvectors:")
mse = np.mean((X - reconstructed.T) ** 2, axis=1)
for i in range(4):
    print(f"MSE for sample {i}: {mse[i]}")
```

```
Using only 2 eigenvectors:
MSE for sample 0: 3.882784888090766
MSE for sample 1: 1.3575824909878655
MSE for sample 2: 0.9870175957576055
MSE for sample 3: 1.260330232133847
```