AI61003: Program Assignment 1

Introduction :

In this assignment, we explored dimensionality reduction techniques using the Olivetti Faces dataset, focusing on Principal Component Analysis (PCA) and Singular Value Decomposition (SVD). We applied PCA at different levels of variance capture to predict and reconstruct facial images. Additionally, we experimented with reconstructing images where the lower half of the faces had been blacked out, using PCA models trained on varying thresholds of variance capture. SVD was also used for image reconstruction, where we examined the effect of using different numbers of singular values. To deepen our understanding of noise reduction, we introduced noise to the training data at varying intensities and applied SVD to mitigate the noise, analysing its effectiveness.

Overview of Techniques :

Principal Component Analysis (PCA)

PCA is a dimensionality reduction technique that transforms data into a new coordinate system, where the axes (principal components) represent directions of maximum variance (the eigenvectors of the covariance matrix). By retaining only the top components—those corresponding to the largest eigenvalues—PCA reduces the dimensionality of the data while preserving key features. In image processing, PCA simplifies image representations by focusing on the most significant patterns or features in the dataset. It is commonly used for feature extraction and visualisation.

Singular Value Decomposition (SVD)

SVD is a matrix factorization method that breaks a matrix down into three components: U (left singular vectors), Σ (singular values), and V^T (right singular vectors). By approximating the original matrix using a reduced number of singular values, SVD effectively reduces dimensionality. In image processing, SVD is often employed for image compression and noise reduction, as retaining only the largest singular values captures the essential structure of the image, while discarding less significant or noisy components.

Dataset Description :

The original dataset consists of 400 images, with 10 images each of 40 different individuals. To create the test set, one image per person (40 images total) is set aside, leaving the remaining 360 images for the training set.

Question 1: Calculate the performance on the Test dataset (in percentage)

Code:

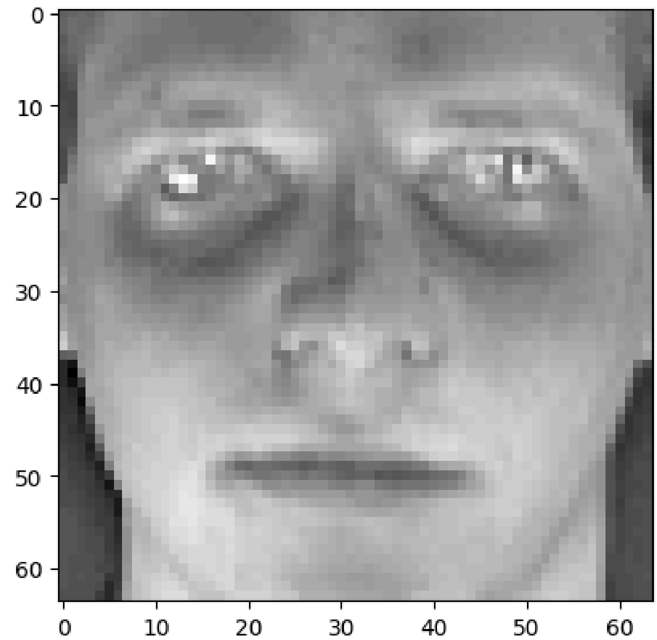
```python
# Q1 calculating performance on the test data of the model using optimize=True
performance=0
for i,image in enumerate(X_test_reduced):
    recognized_image, mu_rec = img_classifer.predict( image )
    performance += 100*(target_test[i] == recognized_image)/target_test.shape[0]

print("Accuracy =", performance, "%")
```

```
Accuracy = 92.5 %
```

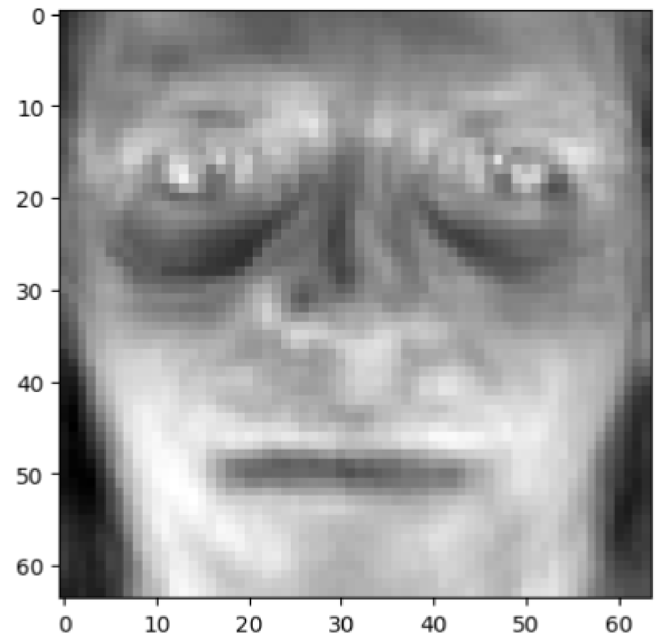
Result:

Following are the results obtained upon predicting classes, training and testing the model on images of reduced dimensions using PCA.
Number of dimensions taken: 62
Accuracy: 92.5%

Actual Image



Reconstructed Image



Question 2: Repeat the face recognition with extracting the eigenvectors that capture (a) 80% of the variance (b) 50% of the variance. Provide sample results and the performance accuracy.

(a) **80% of the variance**

Next, we reanalyze the data by applying PCA to capture 80% of the variance, and then train and test the model on this newly reduced dataset.

Code:

Below is the code used to calculate the model's accuracy after capturing 80% of the variance with PCA.

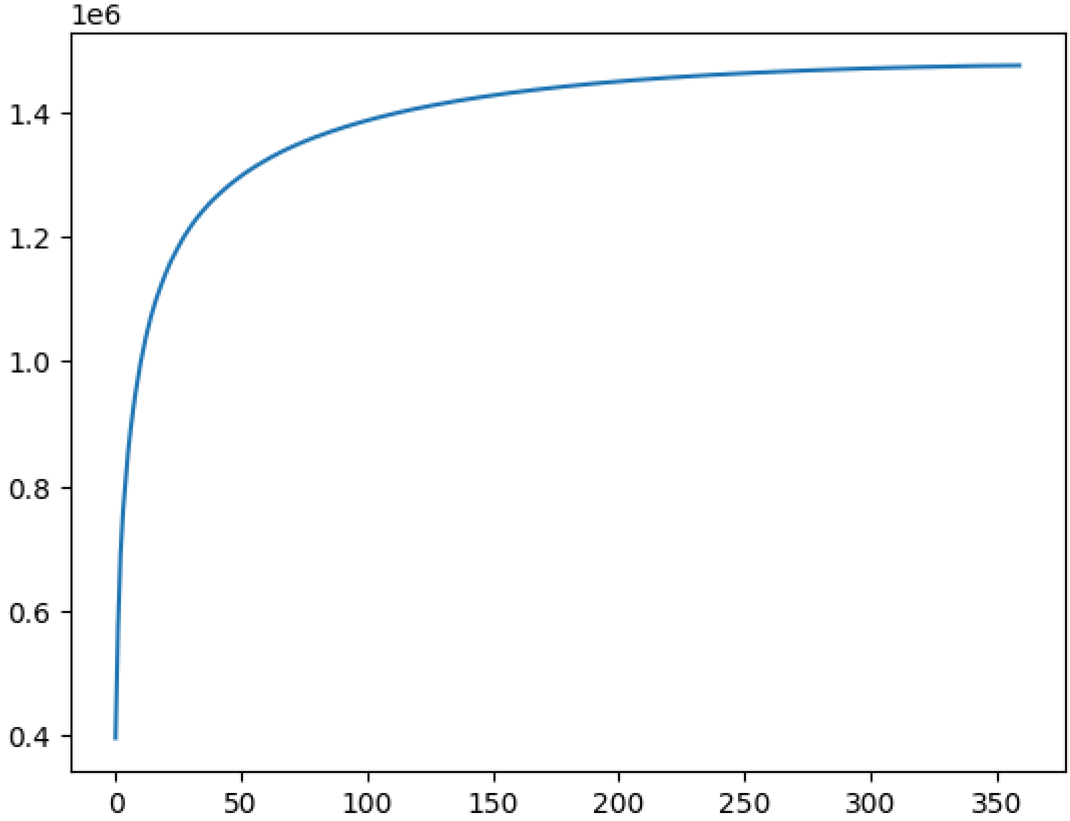
```python
#Q2  PCA with 80% variance
pca_80 = PCA( optimize = True )
B_80 = pca_80.fit(X_train)
pca_80.plot_eigenvals()
num_dim_80 = pca_80.get_num_components(0.8)
print( num_dim_80 )
B_80 = B_80[:,:num_dim_80]
# show top 4 eigenfaces
show_images( B_80, 4 )

# Projection class for capturing variance 80%
proj_80 = Projection( B_80 )
X_train_reduced_80 = proj_80.reduce_dim(X_train)
print("X_train_reduced.shape="+str(X_train_reduced_80.shape))
show_images(X_train.T, 1)
r_img_80 = proj_80.reconstruct( X_train_reduced_80[0,:])
r_img_80 = np.reshape(r_img_80,(4096,1))
show_images(r_img_80, 1)

# reducing dimensionality
X_test_reduced_80 = proj_80.reduce_dim(X_test)
print("X_test_reduced.shape="+str(X_test_reduced_80.shape) )

# fitting model
print("X_train_reduced.shape=" + str(X_train_reduced_80.shape))
print("X_test_reduced.shape=" + str(X_test_reduced_80.shape))
img_classifer_80 = ImageClassifier(40)
img_classifer_80.fit(X_train_reduced_80,target_train )
recognized_class_80, mu_rec_80 = img_classifer_80.predict( X_test_reduced_80[5,: ] )
print("recognized_class="+ str(recognized_class_80))

##### testing model
accuracy_80=0
for i,image in enumerate(X_test_reduced_80):
    recognized_class_80, mu_rec_80 = img_classifer_80.predict( image )
    accuracy_80 += 100*(target_test[i] == recognized_class_80)/target_test.shape[0]
print("Accuracy_80_var =", accuracy_80, "%")
```

Results:

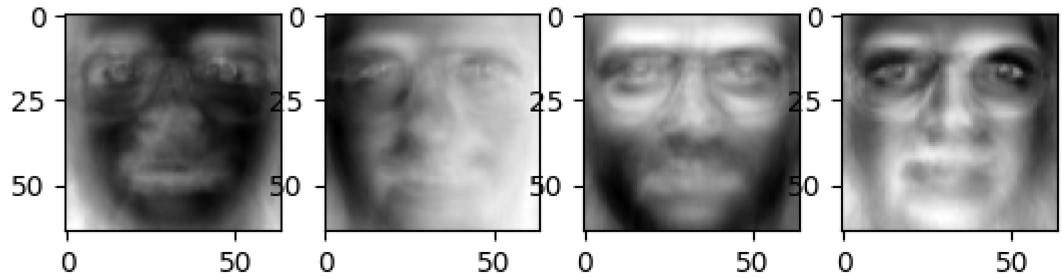
Number of dimensions taken = 25
Accuracy: 92.5%

The output of the code above is as follows. We get the number of dimensions required to be taken as 25 and the reduced shape and predicted classes as follows

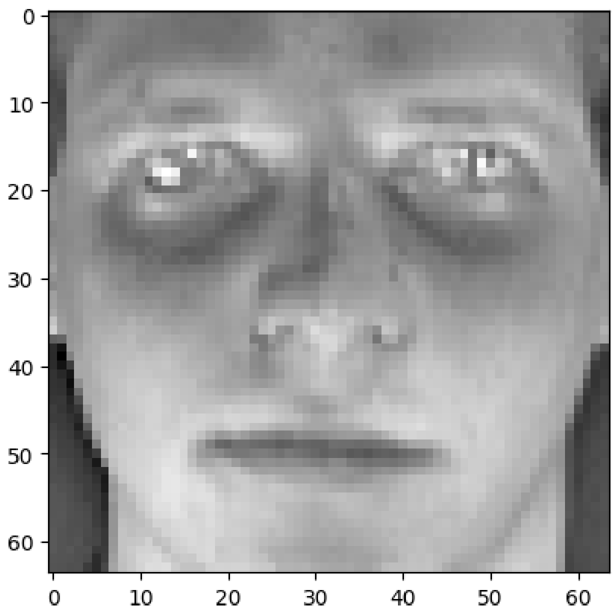
25
X_train_reduced.shape=(360, 25)
X_test_reduced.shape=(40, 25)
X_train_reduced.shape=(360, 25)
X_test_reduced.shape=(40, 25)
recognized_class=5



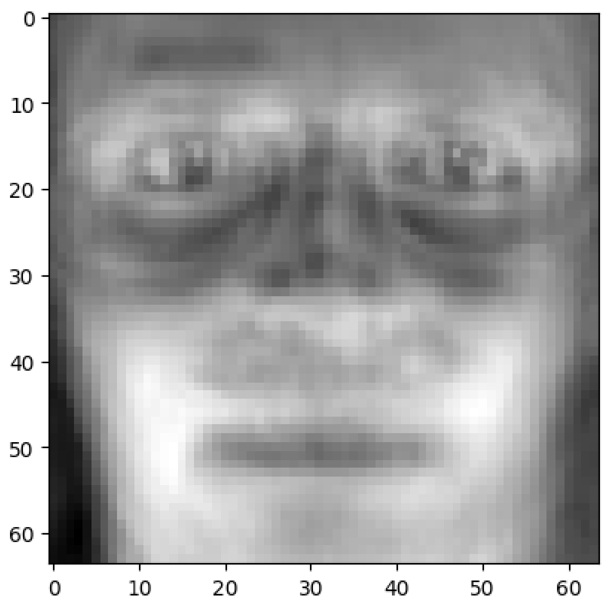
Eigenfaces



True Image



Reconstructed Image



(b) 50% of the variance

Next, we reanalyze the data by applying PCA to capture 50% of the variance, and then train and test the model on this newly reduced dataset.

Code:

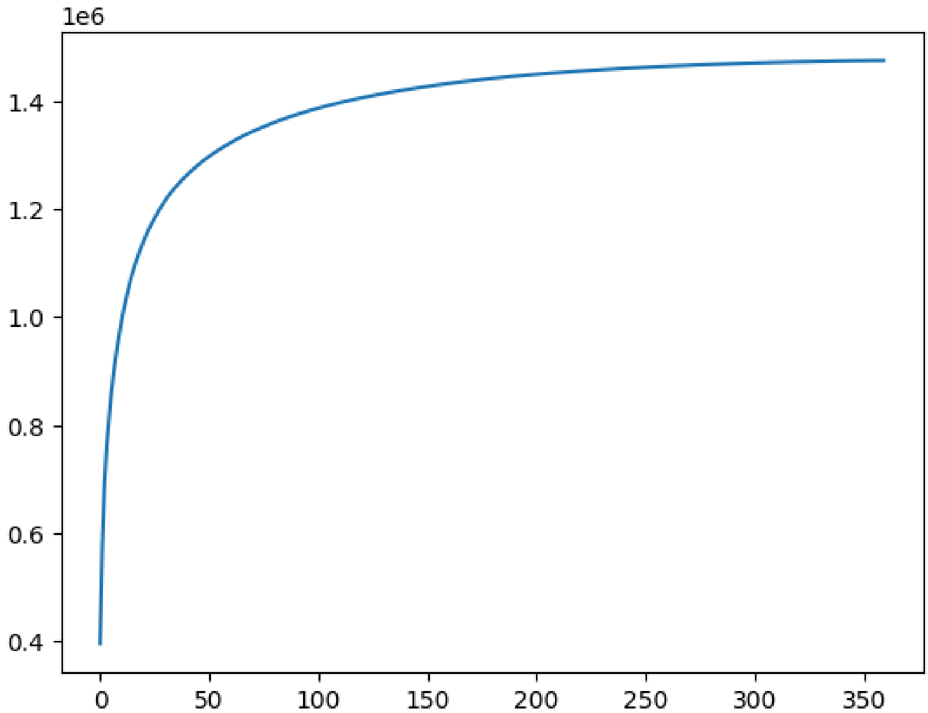
Below is the code used to calculate the model's accuracy after capturing 80% of the variance with PCA.

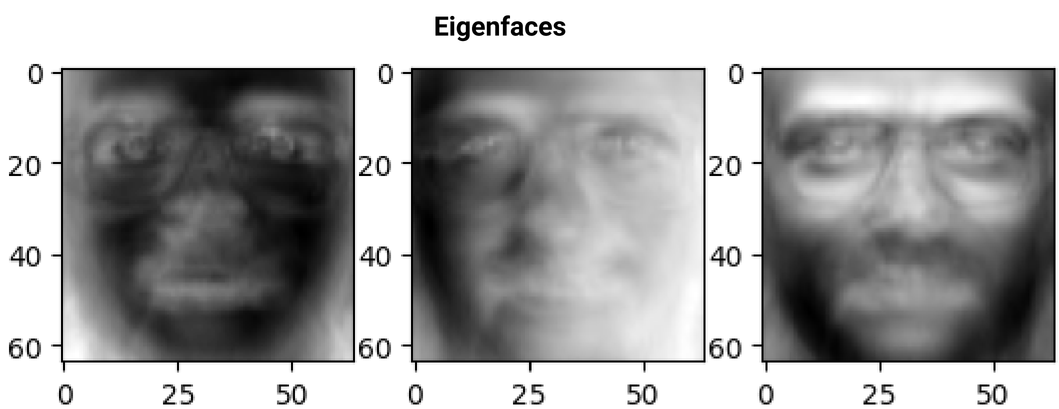
```
#Q2  PCA with 50% variance
pca_50 = PCA( optimize = True )
B_50 = pca_50.fit(X_train)
pca_50.plot_eigenvals()
num_dim_50 = pca_50.get_num_components(0.5)
print( num_dim_50 )
B_50 = B_50[:,:num_dim_50]
# show top 4 eigenfaces
show_images( B_50, 3 )

# Projection class for capturing variance 50%
proj_50 = Projection( B_50 )
X_train_reduced_50 = proj_50.reduce_dim(X_train)
print("X_train_reduced.shape="+str(X_train_reduced_50.shape))
show_images(X_train.T, 1)
r_img_50 = proj_50.reconstruct( X_train_reduced_50[0,:])
r_img_50 = np.reshape(r_img_50,(4096,1))
show_images(r_img_50, 1)

# reducing dimensionality
X_test_reduced_50 = proj_50.reduce_dim(X_test)
print("X_test_reduced.shape="+str(X_test_reduced_50.shape) )

# fitting model
print("X_train_reduced.shape=" + str(X_train_reduced_50.shape))
print("X_test_reduced.shape=" + str(X_test_reduced_50.shape))
img_classifer_50 = ImageClassifier(40)
img_classifer_50.fit(X_train_reduced_50,target_train )
recognized_class_50, mu_rec_50 = img_classifer_50.predict( X_test_reduced_50[5,: ] )
print("recognized_class="+ str(recognized_class_50))

##### testing model
accuracy_50=0
for i,image in enumerate(X_test_reduced_50):
    recognized_class_50, mu_rec_50 = img_classifer_50.predict( image )
    accuracy_50 += 100*(target_test[i] == recognized_class_50)/target_test.shape[0]
print("Accuracy_50 var =", accuracy_50, "%")
```

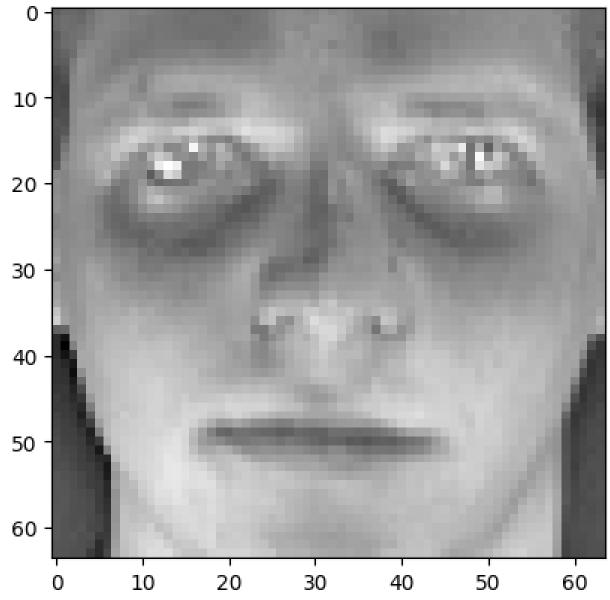
Results:

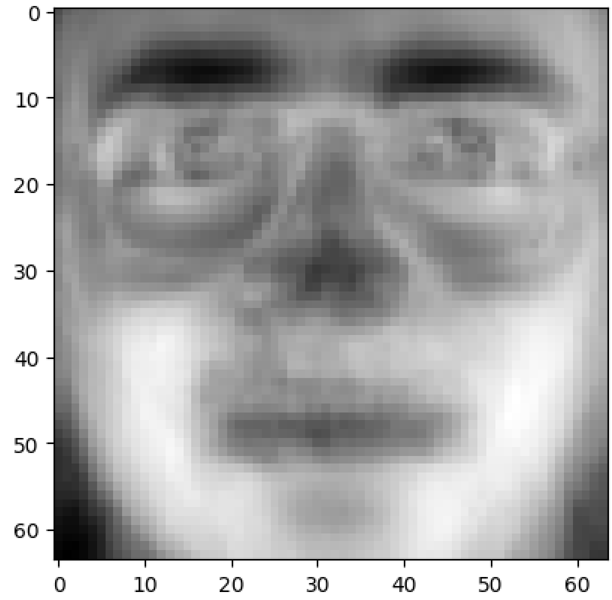
Number of dimensions taken = 3
Accuracy: 37.5%
The output of the code above is as follows. We get the number of dimensions required to be taken as 25 and the reduced shape and predicted classes as follows

3
X_train_reduced.shape=(360, 3)
X_test_reduced.shape=(40, 3)
X_train_reduced.shape=(360, 3)
X_test_reduced.shape=(40, 3)
recognized_class=5









Qn 3: Repeat the face reconstruction with extracting the eigenvectors that capture (a) 80% of the variance (b) 50% of the variance. Provide sample results.

(a) 80% of the variance

```
    std_dev_mod[std_dev == 0 ] = 1
##### replacing standard deviation of 0 by 1 to avoid division by 0


class PCA:

    def __init__( self, optimise = False ):

        self.__optimize = optimise

        #####

 '''

sets self.optimize. optimise=True should be used if number of
dimensions>>number of datapoints

 eigenvalue eigenvector decomposition of X.T @ X is similar to eigenvalue
eigenvector decomposition of X @ X.T as follows

            (X.T@X) @ Q = Q @ L                ------(1)
```

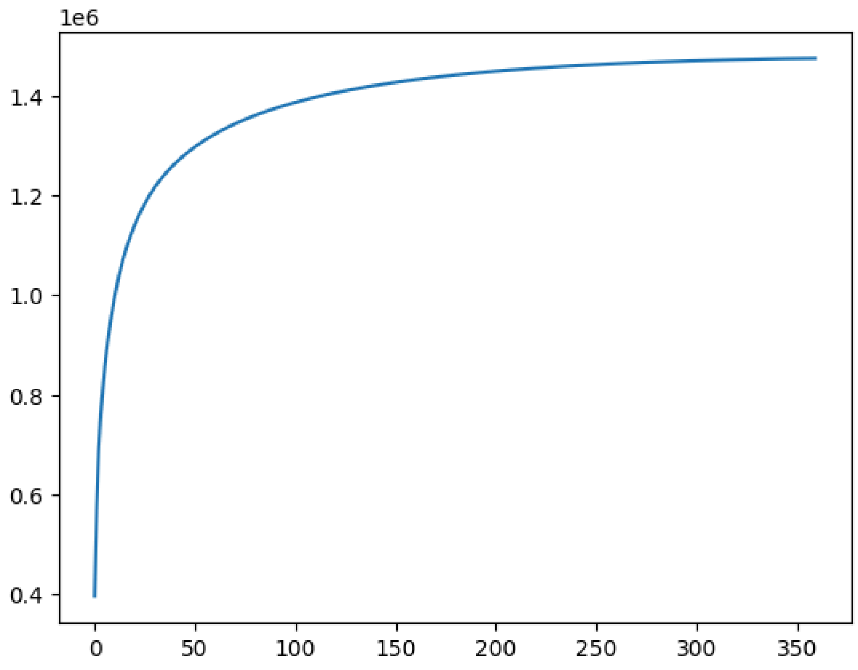
```
also

  (X@X.T) @ Q' = Q' @ L

=>  (X.T@X) @ (X.T@Q') = (X.T@Q') @ L   (multiplying X.T on both sides)
------(2)

=> (X.T@X) @ Q = Q @ L (which is the same as equation (1) if X.T@Q' != 0)

        '''

 #####
```

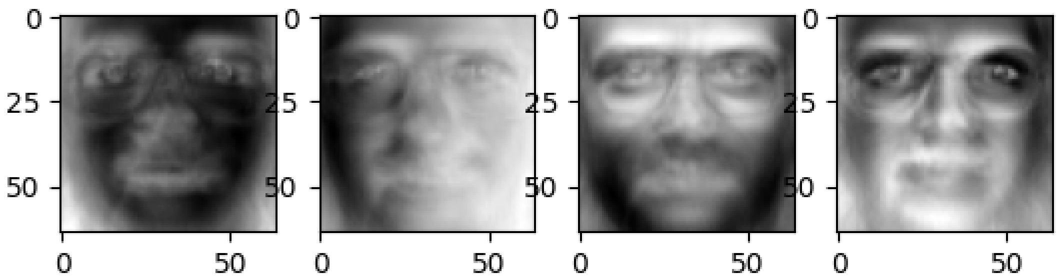
let’s address the task of determining the optimal value of K.

1. We plot the cumulative sum of the eigenvalues to visualise how many eigenvectors capture the maximum variance in the training data. From the plot below, we observe that most of the variance is captured by the first 50 features.
2. We extract the eigenvectors that capture around 90% of the variance (this threshold can be adjusted as needed).

Given that the number of dimensions (D = 4096) is much greater than the number of training samples (N = 360), we compute the covariance matrix with dimensions N x N instead of D x D.

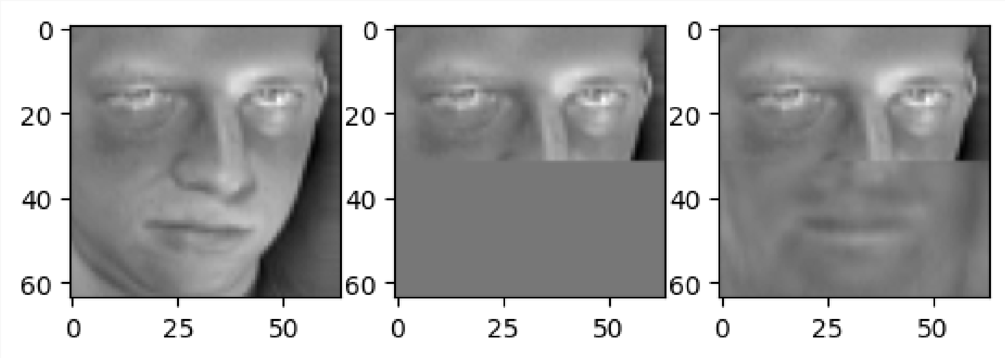
```
pca = PCA( optimize = True )
B = pca.fit(X_train)
pca.plot_eigenvals()
num_dim = pca.get_num_components(0.9)
print( num_dim )
B = B[:,:num_dim]
# show top 4 eigenfaces
show_images( B, 4 )
62
```

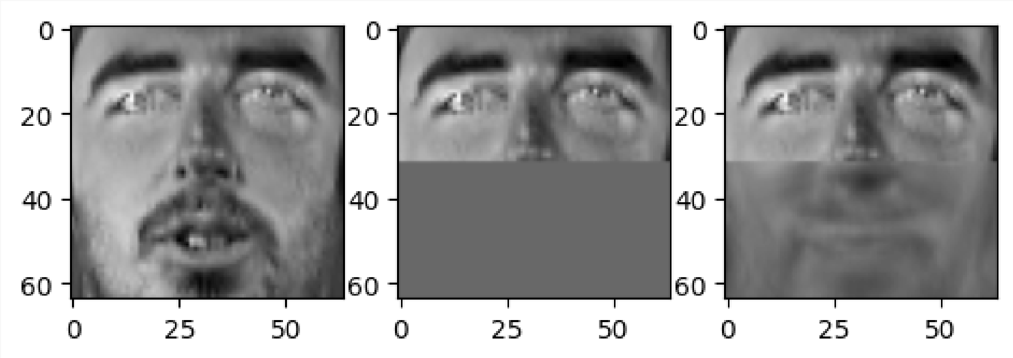


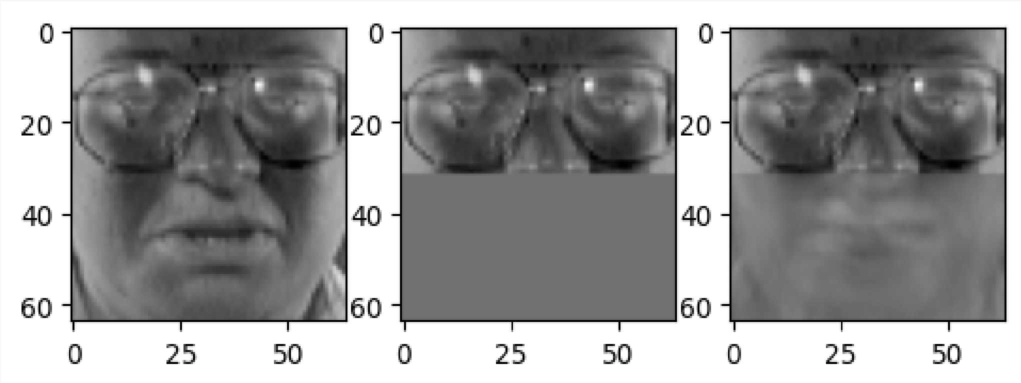


Qn 3: Repeat the face reconstruction with extracting the eigenvectors that capture (a) 80% of the variance (b) 50% of the variance. Provide sample results.

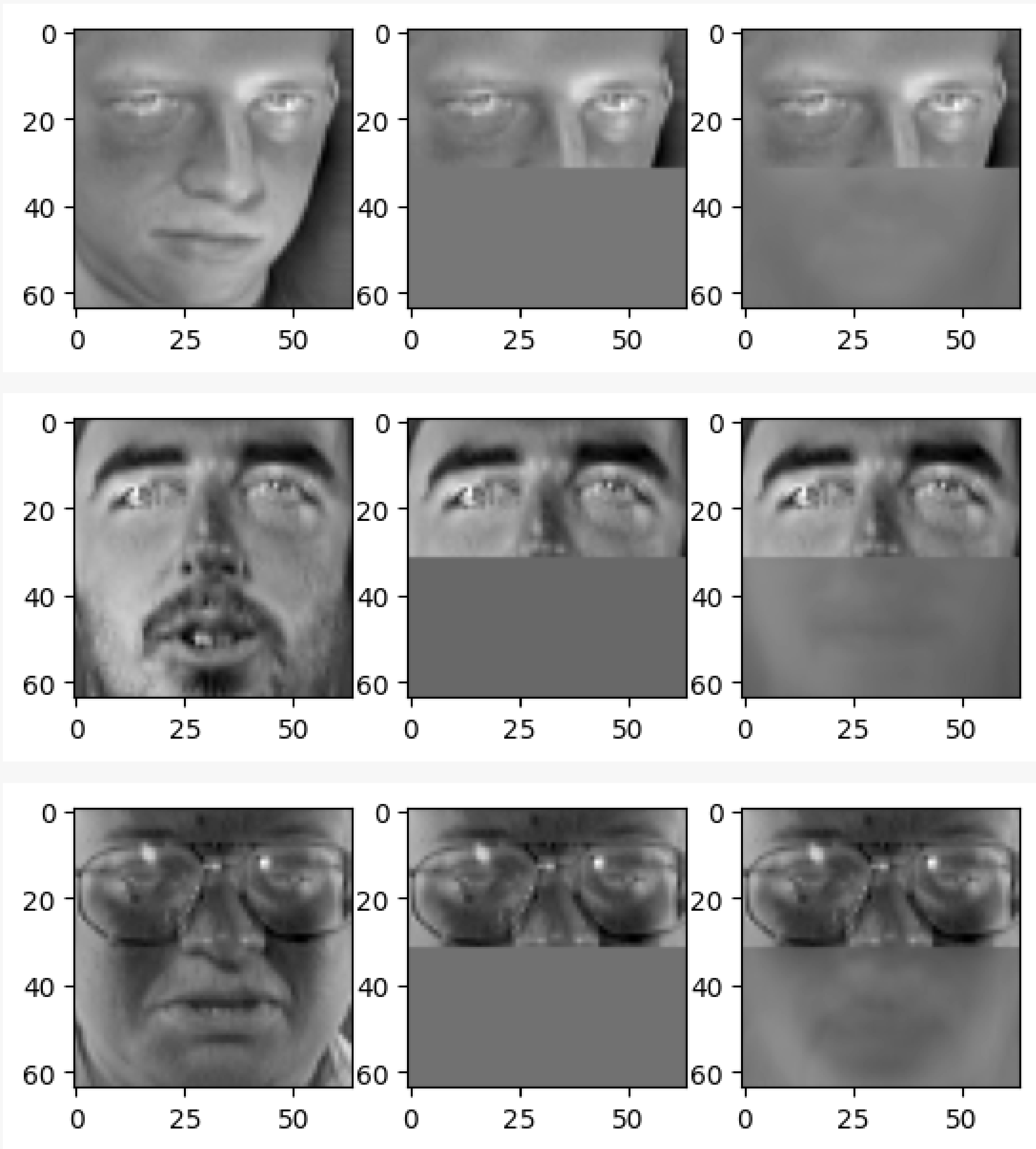
```
# Q3
# reconstruct image with PCA capturing 80% variance
def reconstruct_half_images_80( test_indexes ):
   for i in test_indexes:
       half_image, orig_image = get_half_image(i)
       N,D = half_image.shape
       new_image = proj_80.project( half_image  )
       new_image[0,0:2048,] = orig_image[0, 0:2048]
       images_for_display = np.concatenate((orig_image.T, half_image.T,
new_image.T), axis=1 )
       show_images(images_for_display, 3)

reconstruct_half_images_80([0,10,30])
```







```
#Q3
#reconstruct image with PCA capturing 50% variance
def reconstruct_half_images_50( test_indexes ):
    for i in test_indexes:
        half_image, orig_image = get_half_image(i)
        N,D = half_image.shape
        new_image = proj_50.project( half_image  )
        new_image[0,0:2048,] = orig_image[0, 0:2048]
        images_for_display = np.concatenate((orig_image.T, half_image.T, 
new_image.T), axis=1 )
        show_images(images_for_display, 3)

reconstruct_half_images_50([0,10,30])
```



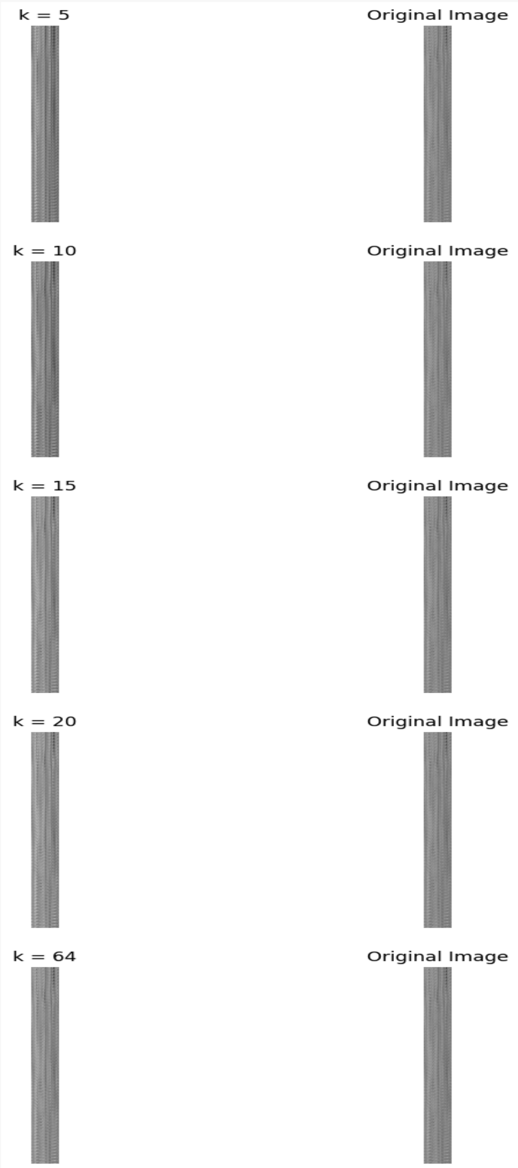
Singular Value Decomposition

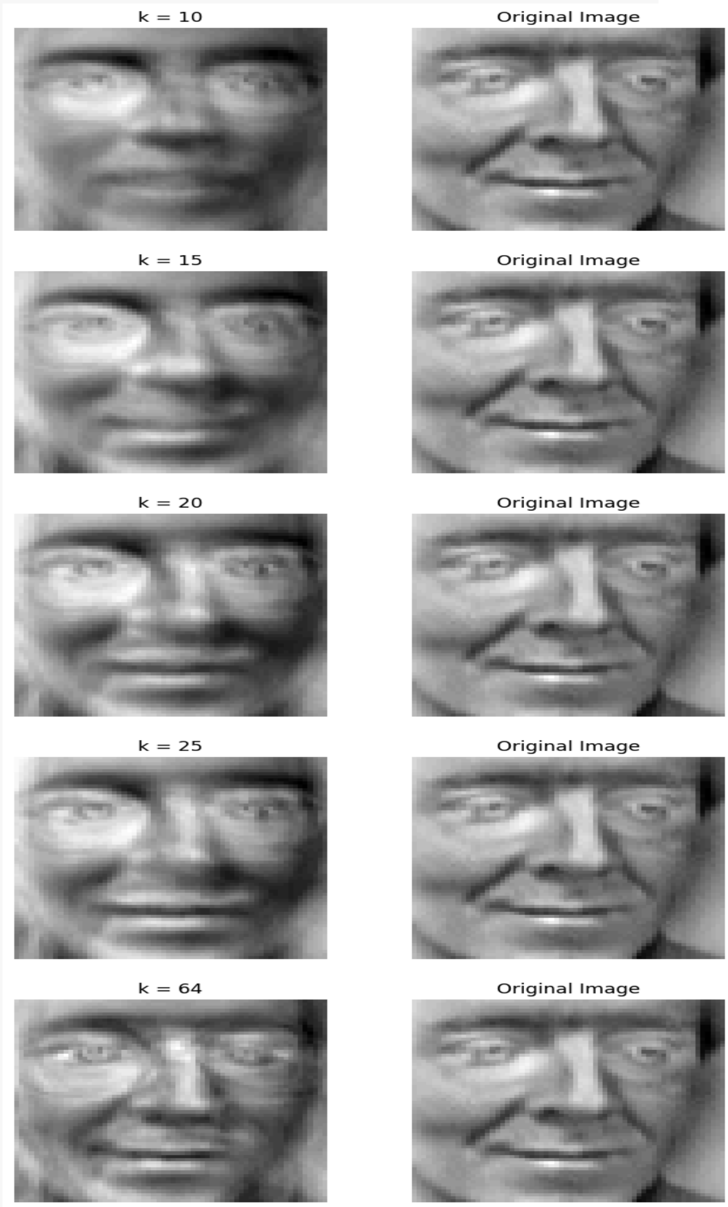
To reduce the noise in data.

To address the task of adding random noise to the data and comparing the results with the original no-noise case (as shown in the provided image), you can follow these steps:

1. **Add Random Noise:** In your dataset, add random noise to the images before applying PCA or SVD. You may generate noise by adding a small random value to each pixel.
2. **Train with Noise:** Re-apply PCA or SVD after adding noise, following the same process as with the clean data.

3. **Reconstruct Images:** Reconstruct the images using different values of k (number of components or singular values). For instance, apply PCA or SVD with k values of 5, 10, 15, 20, and 64, just like in the original comparison.
4. **Visual Comparison:** Plot and compare the reconstructed images for both the noise-added and no-noise cases, side by side, for each k value. This will allow you to observe how well the models perform in denoising and reconstruction.

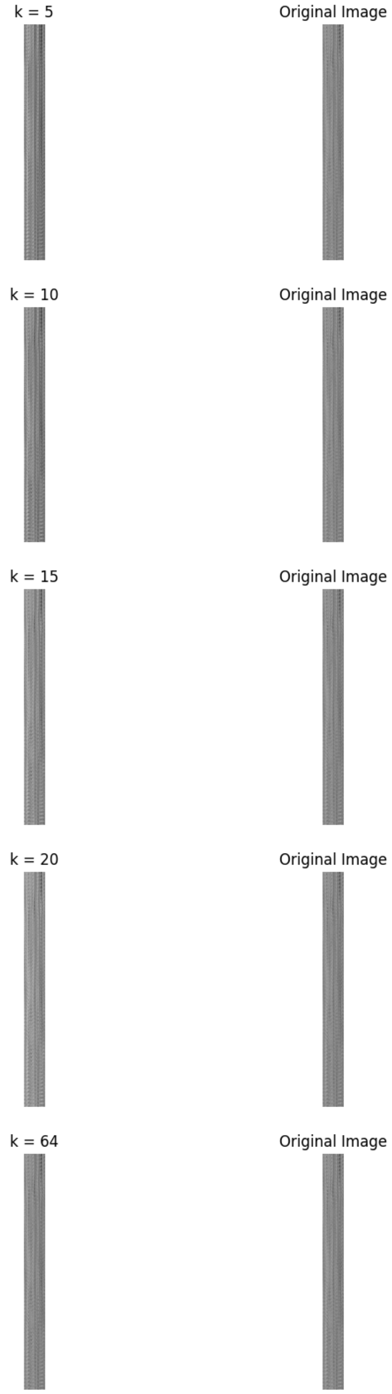


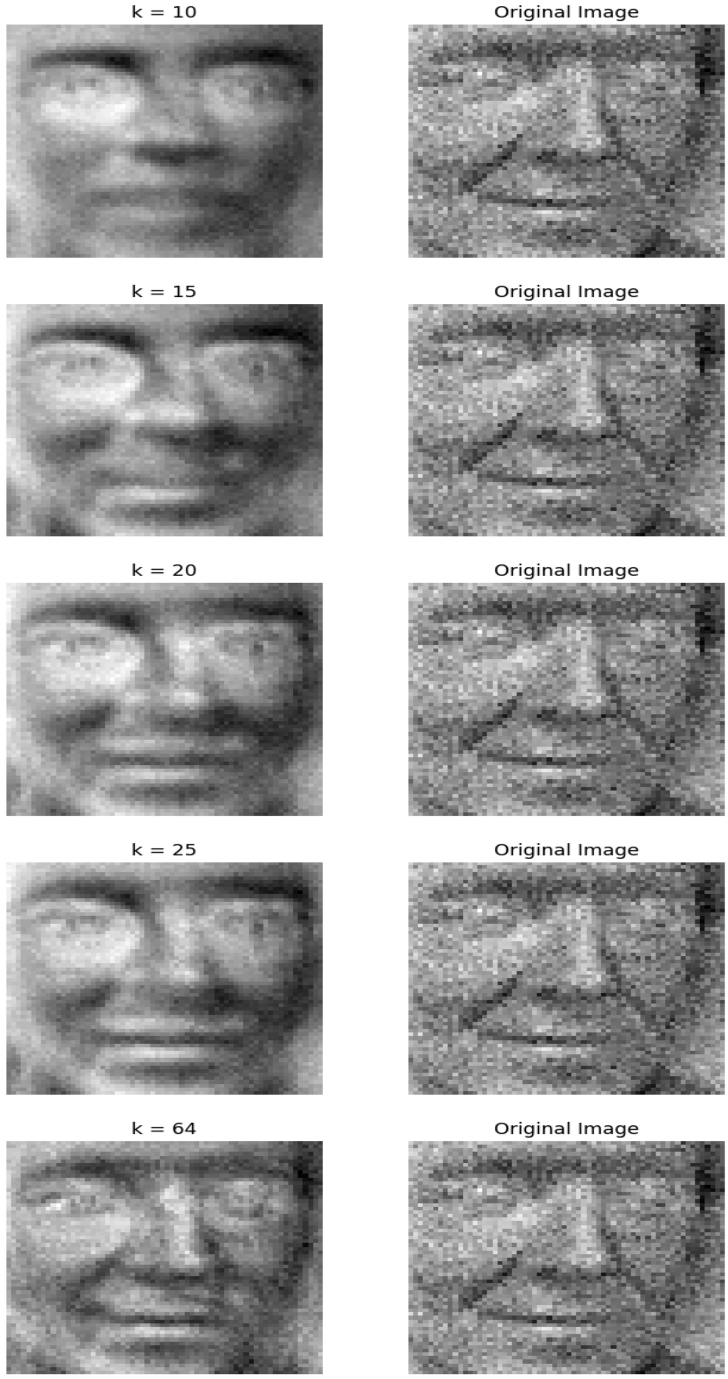


Q4. Add Random noise to the data and compare these results with the no noise case above. (you may uncomment the noise addition part in the code).

```
# experimenting on noise with st_deviation of 0.5
data_std = X_train.T
# Add noise
noise = np.random.normal(0,0.5,data.shape)
noisy_data = data_std + noise
data_std = noisy_data

plt.imshow(data_std, cmap='gray')
print("data_std.shape=" + str(data_std.shape))

# calculate the SVD and plot the image
U_std, S_std, V_T_std = svd(data_std, full_matrices=False)
S_std = np.diag(S_std)
fig_std, ax_std = plt.subplots(5, 2, figsize=(8, 20))


curr_fig = 0
for r in [5, 10, 15, 20, 64]:
    data_approx_std_1 = U_std[:, :r] @ S_std[0:r, :r] @ V_T_std[:r, :]
    ax_std[curr_fig][0].imshow(data_approx_std_1, cmap='gray')
    ax_std[curr_fig][0].set_title("k = "+str(r))
    ax_std[curr_fig, 0].axis('off')
    ax_std[curr_fig][1].set_title("Original Image")
    ax_std[curr_fig][1].imshow(data, cmap='gray')
    ax_std[curr_fig, 1].axis('off')
    curr_fig += 1
plt.show()
```

Result:





Q5. Use standard deviation of noise as 0.05, 0.1, and 0.2.

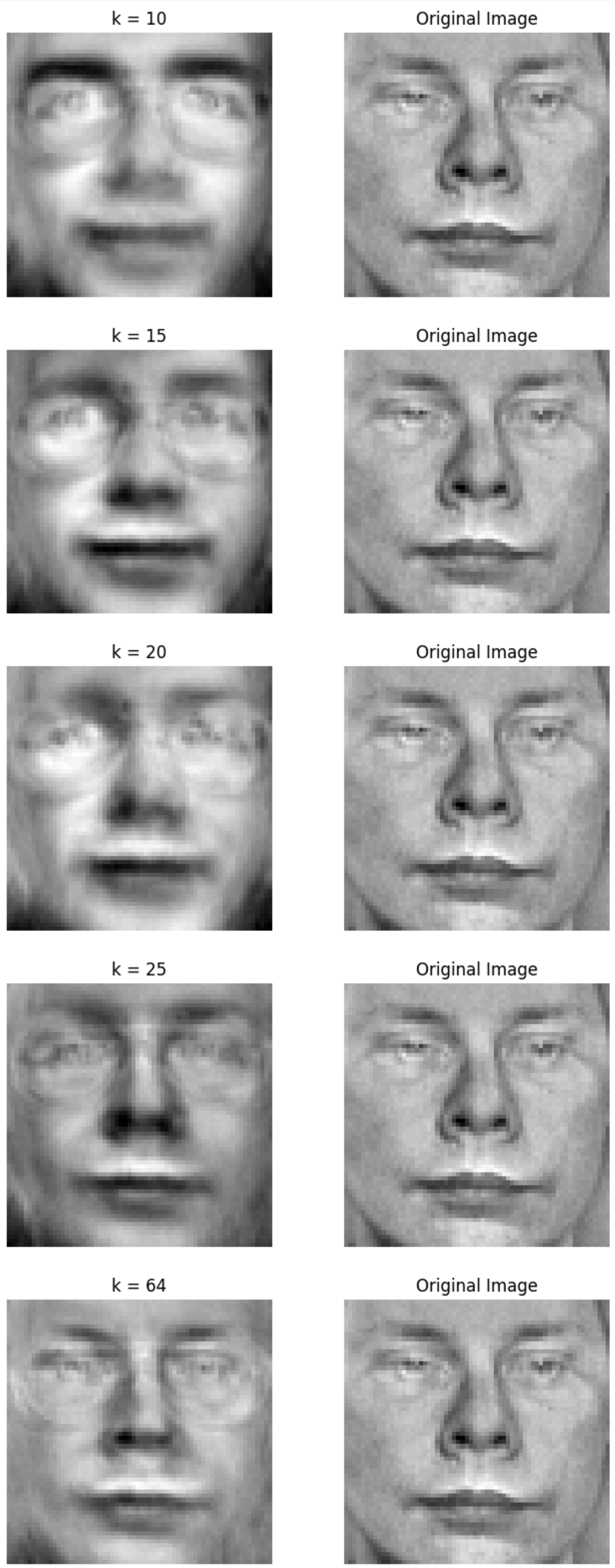
(a) Use standard deviation 0.05

Code:

```
# Q5
# adding noise with st_deviation of 0.05
data_std_05 = X_train.T
# Adding noise
```

```
noise = np.random.normal(0,0.05,data.shape)
noisy_data = data_std_05 + noise
data_std_05 = noisy_data

plt.imshow(data_std_05, cmap='gray')
print("data_std_05.shape=" + str(data_std_05.shape))

# calculate the SVD and plot the image
U_std_05, S_std_05, V_T_std_05 = svd(data_std_05, full_matrices=False)
S_std_05 = np.diag(S_std_05)
fig_std_05, ax_std_05 = plt.subplots(5, 2, figsize=(8, 20))


curr_fig = 0
for r in [5, 10, 15, 20, 64]:
   data_approx_std_05 = U_std_05[:, :r] @ S_std_05[0:r, :r] @
V_T_std_05[:r, :]
   ax_std_05[curr_fig][0].imshow(data_approx_std_0_05, cmap='gray')
   ax_std_05[curr_fig][0].set_title("k = "+str(r))
   ax_std_05[curr_fig, 0].axis('off')
   ax_std_05[curr_fig][1].set_title("Original Image")
   ax_std_05[curr_fig][1].imshow(data, cmap='gray')
   ax_std_05[curr_fig, 1].axis('off')
   curr_fig += 1
plt.show()
```

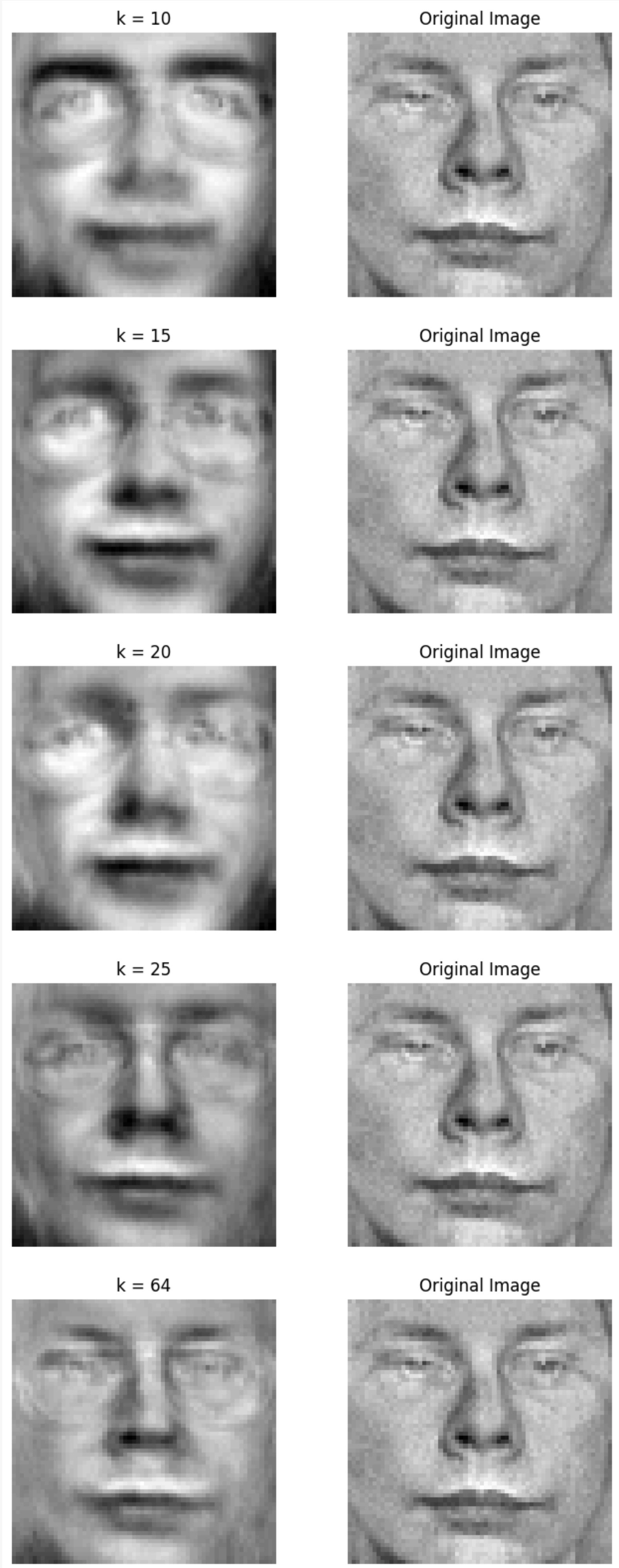
Result:


k = 10
Original Image
k = 15
Original Image
k = 20
Original Image
k = 25
Original Image
k = 64
Original Image

(b) Use standard deviation 0.1

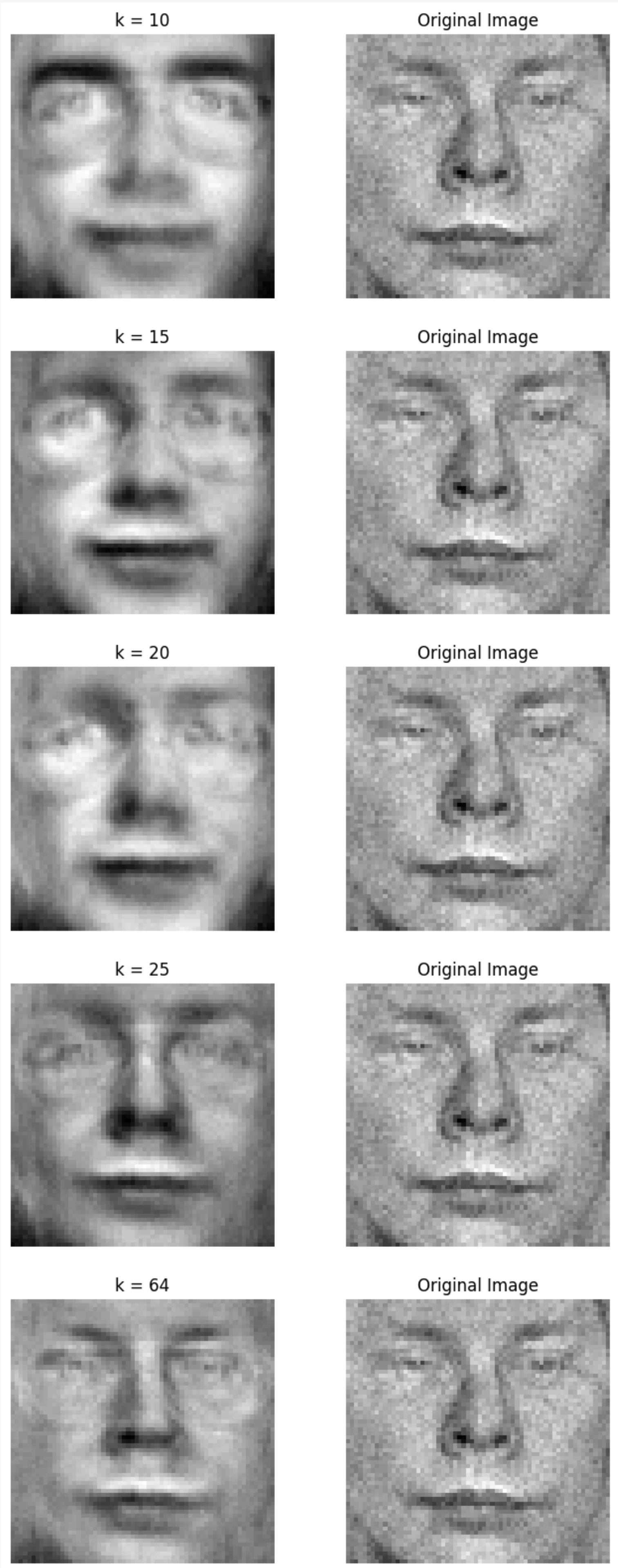
```
# Q5
# adding noise with st_deviation of 0.1
data_std_1 = X_train.T
# Adding noise
noise = np.random.normal(0,0.1,data.shape)
noisy_data = data_std_1 + noise
data_std_1 = noisy_data

plt.imshow(data_std_1, cmap='gray')
print("data_std_1.shape=" + str(data_std_1.shape))

# calculate the SVD and plot the image
U_std_1, S_std_1, V_T_std_1 = svd(data_std_1,
full_matrices=False)
S_std_1 = np.diag(S_std_1)
fig_std_1, ax_std_1 = plt.subplots(5, 2, figsize=(8, 20))


curr_fig = 0
for r in [5, 10, 15, 20, 64]:
   data_approx_std_1 = U_std_1[:, :r] @ S_std_1[0:r, :r] @
V_T_std_1[:r, :]
   ax_std_1[curr_fig][0].imshow(data_approx_std_0_05,
cmap='gray')
   ax_std_1[curr_fig][0].set_title("k = "+str(r))
   ax_std_1[curr_fig, 0].axis('off')
   ax_std_1[curr_fig][1].set_title("Original Image")
   ax_std_1[curr_fig][1].imshow(data, cmap='gray')
   ax_std_1[curr_fig, 1].axis('off')
   curr_fig += 1
plt.show()
```

Result:


k = 10
Original Image
k = 15
Original Image
k = 20
Original Image
k = 25
Original Image
k = 64
Original Image

```
   © use standard deviation of 0.2

##### Q5
##### adding noise with standard deviation of 0.2

data_std_0_2 = X_train.T
# Adding noise
noise = np.random.normal(0,0.2,data.shape)
noisy_data = data_std_0_2 + noise
data_std_0_2 = noisy_data

plt.imshow(data_std_0_2, cmap='gray')
print("data_std_0_2.shape=" + str(data_std_0_2.shape))

# calculate the SVD and plot the image
U_std_0_2, S_std_0_2, V_T_std_0_2 = svd(data_std_0_2,
full_matrices=False)
S_std_0_2 = np.diag(S_std_0_2)
fig_std_0_2, ax_std_0_2 = plt.subplots(5, 2, figsize=(8, 20))


curr_fig = 0
for r in [5, 10, 15, 20, 64]:
   data_approx_std_0_2 = U_std_0_2[:, :r] @ S_std_0_2[0:r, :r] @
V_T_std_0_2[:r, :]
   ax_std_0_2[curr_fig][0].imshow(data_approx_std_0_2, cmap='gray')
   ax_std_0_2[curr_fig][0].set_title("k = "+str(r))
   ax_std_0_2[curr_fig, 0].axis('off')
   ax_std_0_2[curr_fig][1].set_title("Original Image")
   ax_std_0_2[curr_fig][1].imshow(data, cmap='gray')
   ax_std_0_2[curr_fig, 1].axis('off')
   curr_fig += 1
plt.show()
```

Result:


k = 10
Original Image
k = 15
Original Image
k = 20
Original Image
k = 25
Original Image
k = 64
Original Image