EEE482 - Computational Neuroscience

Homework Assignment - 4

Ahmet Ali Nuhoğlu
21602149

Ahmet Ali NUHOĞLU
21602149

Homework 4

Question 1

In this question, we are provided a 1000x1024 matrix. Each row of this matrix represents a 32x32 face image. In total we have 1000 face images with resolution 32x32. We will be applying three different methods, which are **PCA, ICA** and **NNMF**, to reconstruct these and we will try to measure their accuracy by computing the mean and standard deviations of their errors.

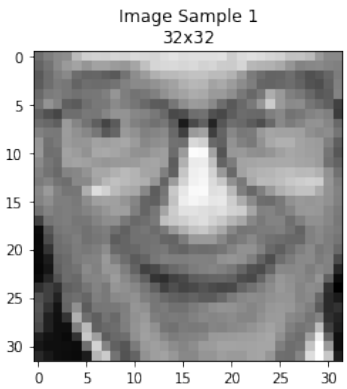
Part A

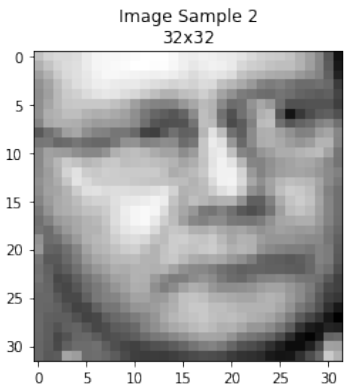
In part a we are expected to apply **PCA** and find the explained variance proportion of the first 100 principal components. Then, first 25 principal components will be displayed. Mathematical background of PCA reconstruction is as follows.

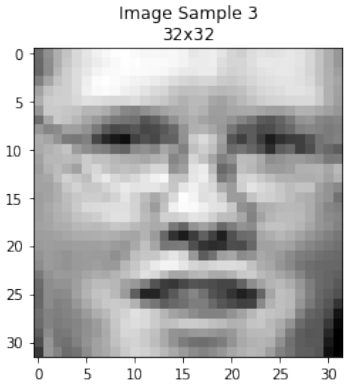
$$\textbf{PCA Reconstruction} = \textbf{PC scores} \cdot \textbf{Eigenvectors}^T + \textbf{Mean}$$

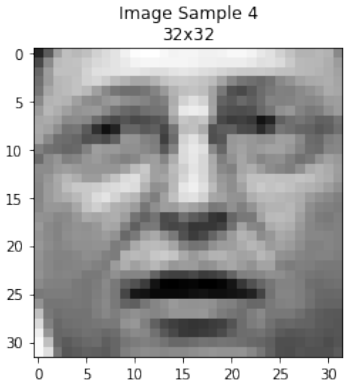
To do that PCA implementation of the **sklearn** library is used. Also, it already has a built in **explained_variance_ratio_** function which is definitely what we need for the first part. Implementation of these and the resulting code is given below.

```
faces = hw4_data1['faces'].T

# some sample images
for i in range(0, 5):
    plt.figure()
    plt.imshow(faces[i].reshape(32, 32).T, cmap=plt.cm.gray)
    plt.title('Image Sample %s\n32x32' %str(i+1))
    plt.show(block=False)
```

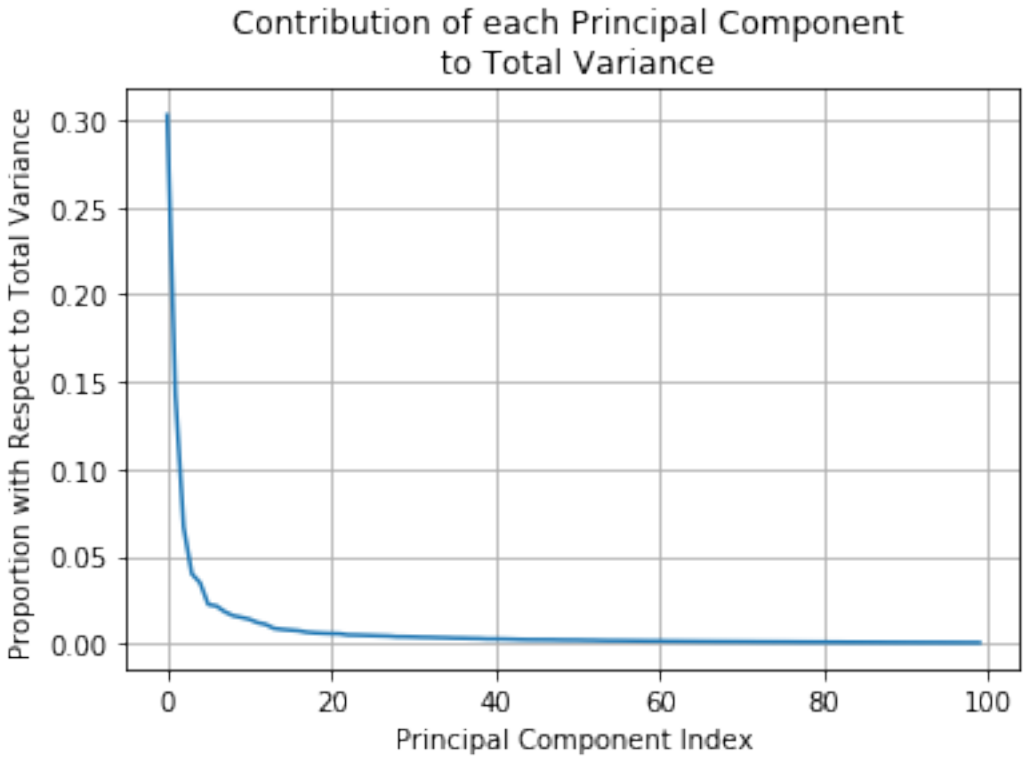








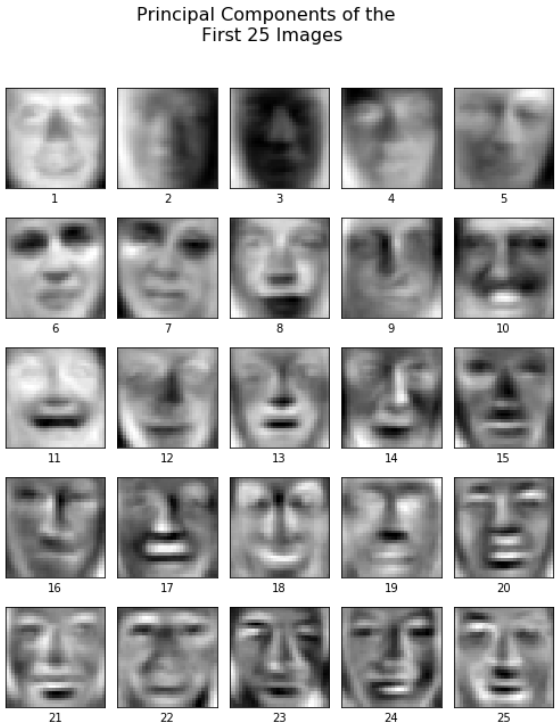

```
faces_pca = PCA(100)
faces_pca.fit(faces)

plt.figure()
plt.plot(faces_pca.explained_variance_ratio_)
plt.xlabel('Principal Component Index')
plt.ylabel('Proportion with Respect to Total Variance')
plt.title('Contribution of each Principal Component \n to Total␣
 ↪Variance')
plt.grid()
plt.show(block=False)
```



From the plot we can say that taking a look at approximately the first 25 principal components are nearly enough for us to understand the variance of the whole.

Using the following code we have just plotted the first 25 principal components. They had already been generated by the **sklearn PCA**. Just by using **.components_** we have accessed and displayed them.

```
fig, axes = plt.subplots(5, 5, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.suptitle('Principal Components of the \n First 25 Images',␣
 ↪fontsize='16')
fig.subplots_adjust(left=None, bottom=None, right=None, top=None,␣
 ↪hspace=.3, wspace=-.4)
for i, ax in enumerate(axes.flat):
    ax.imshow(faces_pca.components_[i].reshape(32, 32).T, cmap=plt.cm.
 ↪gray)
    ax.set_xlabel(i+1)
```



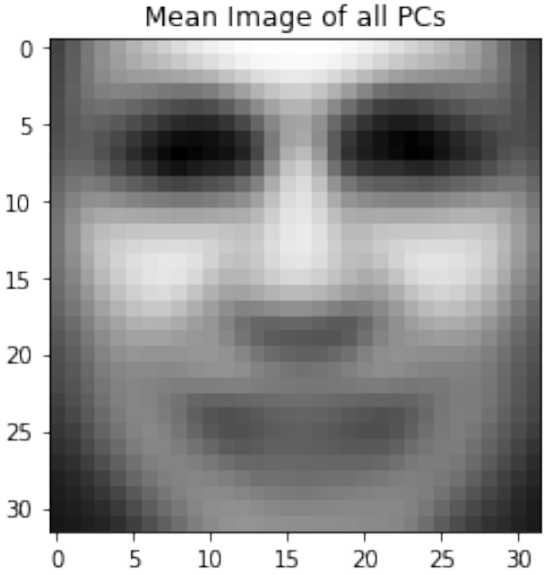
Part B

Now we will be reconstructing first 36 images based on 10, 25 and 50. To do that we will be using a previously stated which is

$$\textbf{PCA Reconstruction} = \textbf{PC scores} \cdot \textbf{Eigenvectors}^T + \textbf{Mean}$$

Code implementations of these operation is given below.

```
plt.figure()
faces_pca_mean = faces_pca.mean_
plt.imshow(faces_pca_mean.reshape(32, 32).T, cmap=plt.cm.gray)
plt.title("Mean Image of all PCs")
plt.show(block = False)
```



Just for curiosity, I have wondered and checked the what does the mean of 1000 face images look like. Also, we can say that all reconstructions will be based on this one as we use more PCs fro reconstruction, they will get closer to the original versions.

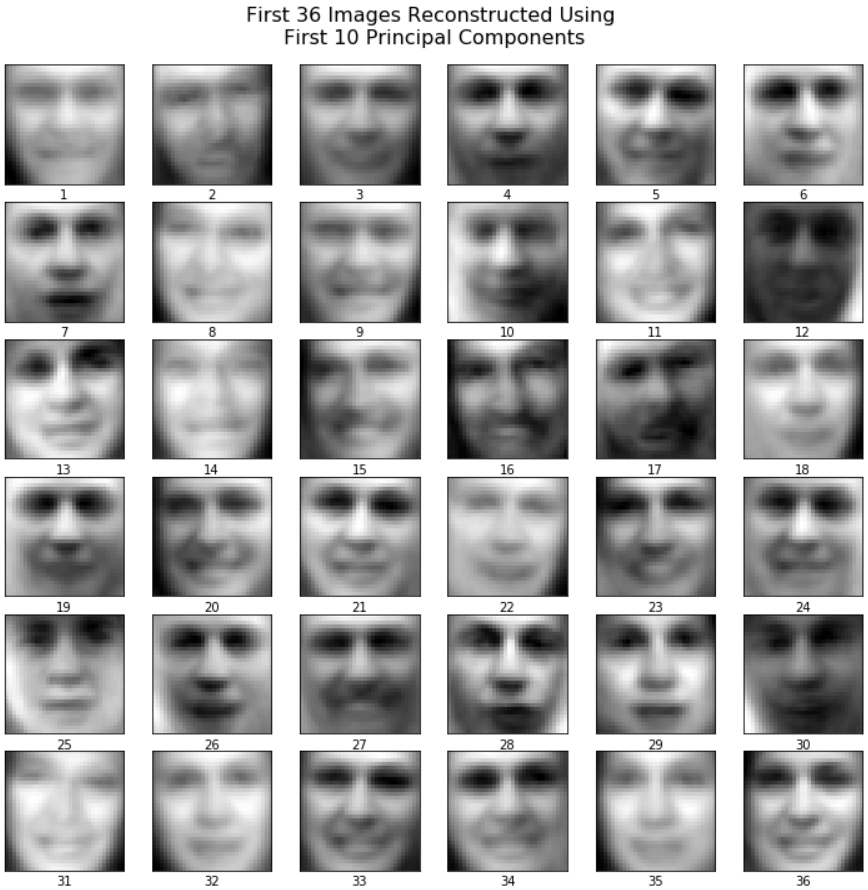
```
fig, axes = plt.subplots(6, 6, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.suptitle('Original Versions of the First 36 Images', fontsize='16')
fig.tight_layout(rect=[0, 0, 1, .95])
for i, ax in enumerate(axes.flat):
    ax.imshow(faces[i].reshape(32, 32).T, cmap=plt.cm.gray)
    ax.set_xlabel(i+1)
```

In the figure above we can see the original versions of the first 36 images. In the following pages we will be seeing the code implementations of reconstructions based on 10, 25 and 50 principal components respectively. Also, mean and standard deviations of their errors are provided right after the codes. They will be used to observe how increasing number of principal components effects the error of the reconstrucitons.

```
faces_pca_10 = (faces - faces_pca_mean).dot(faces_pca.components_[0:10].
 ↪T).dot(faces_pca.components_[0:10]) + faces_pca_mean #Change if␣
 ↪possible
fig, axes = plt.subplots(6, 6, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('First 36 Images Reconstructed Using\n First 10 Principal␣
 ↪Components', fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(faces_pca_10[i].reshape(32, 32).T, cmap=plt.cm.gray)
    ax.set_xlabel(i+1)

pca_10_mse = (faces_pca_10 - faces) ** 2
print("Mean of MSE: %f" %np.mean(pca_10_mse))
```

```
print("Standard Deviation of MSE: %f" %np.std(np.mean(pca_10_mse,␣
 ↪axis=1)))
```

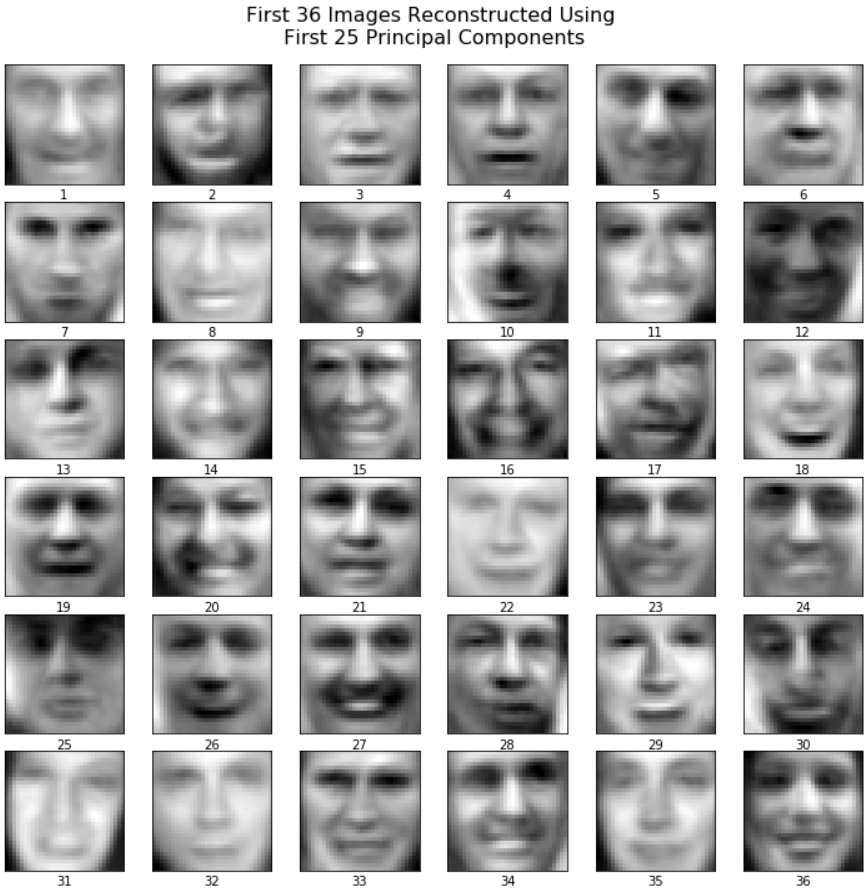
```
Mean of MSE: 523.241745
Standard Deviation of MSE: 257.641200
```




```
faces_pca_25 = (faces - faces_pca_mean).dot(faces_pca.components_[0:25].
 ↪T).dot(faces_pca.components_[0:25]) + faces_pca_mean
fig, axes = plt.subplots(6, 6, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('First 36 Images Reconstructed Using\n First 25 Principal␣
 ↪Components', fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(faces_pca_25[i].reshape(32, 32).T, cmap=plt.cm.gray)
    ax.set_xlabel(i+1)

pca_25_mse = (faces_pca_25 - faces) ** 2
print("Mean of MSE: %f" %np.mean(pca_25_mse))
```

```
print("Standard Deviation of MSE: %f" %np.std(np.mean(pca_25_mse,␣
 ↪axis=1)))
```

```
Mean of MSE: 332.256492
Standard Deviation of MSE: 153.110280
```




```
faces_pca_50 = (faces - faces_pca_mean).dot(faces_pca.components_[0:50].
 ↪T).dot(faces_pca.components_[0:50]) + faces_pca_mean #Change if␣
 ↪possible
fig, axes = plt.subplots(6, 6, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('First 36 Images Reconstructed Using\n First 50 Principal␣
 ↪Components', fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(faces_pca_50[i].reshape(32, 32).T, cmap=plt.cm.gray)
    ax.set_xlabel(i+1)

pca_50_mse = (faces_pca_50 - faces) ** 2
print("Mean of MSE: %f" %np.mean(pca_50_mse))
```

```
print("Standard Deviation of MSE: %f" %np.std(np.mean(pca_50_mse,␣
 ↪axis=1)))
```

```
Mean of MSE: 198.425194
Standard Deviation of MSE: 84.178155
```


```
Mean of MSE(10): 523.241745
Mean of MSE(25): 332.256492
Mean of MSE(50): 198.425194
Standard Deviation of MSE(10): 257.641200
Standard Deviation of MSE(25): 153.110280
Standard Deviation of MSE(50): 84.178155
```

Just as expected ratio of the error decreases due to the increase in the number of the principal components. Also, this can be observed from the provided reconstructed images above. After, 50 principal components as expected it is not possible to observe a decrease like these three in the error since we already have a great intuition about the total variance.

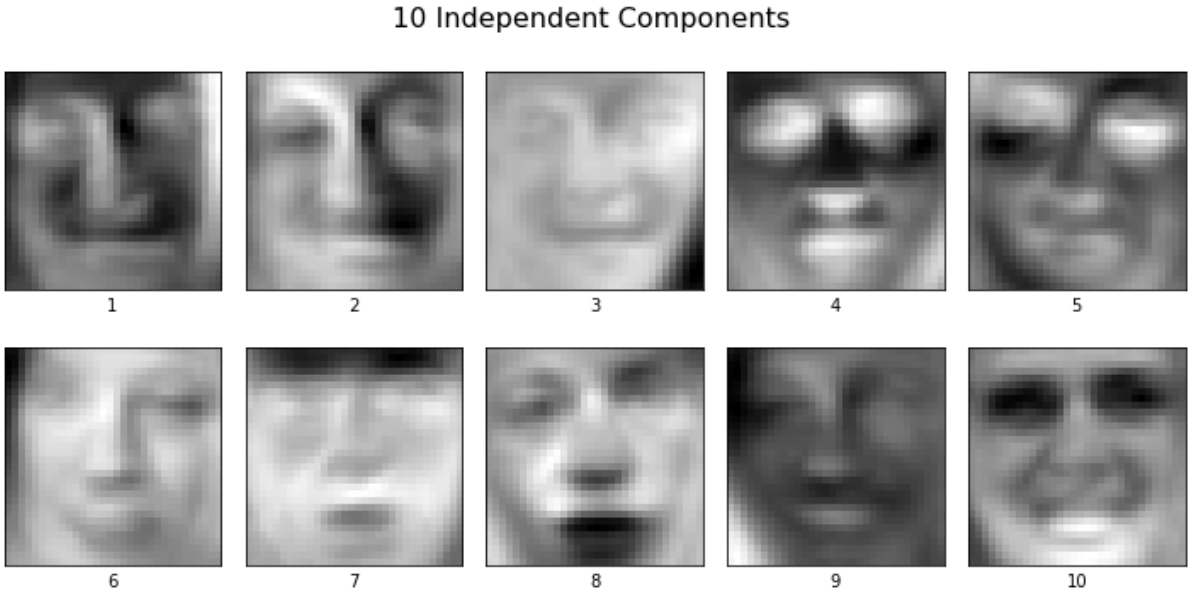
Reconstructed images are good but not perfect. I think that if we had images with higher resolutions our reconstructed images would be more satisfying.

Part C

In this part we will be using **Independent Component Analysis** technique to reconstruct our images. Also, we will again compute the mean and the standard deviation of the error made in these reconstructed images and we will compare it with **Principal Component Analysis**.

While conducting the ICA, **FastICA** function definition of the **sklearn** library is used. In the following code blocks, how 10,25 and 50 independent components are generated can be found and after each block images of those components are displayed.

```
ica_10 = FastICA(10, random_state=np.random.seed(2)).fit(faces)

fig, axes = plt.subplots(2, 5, figsize=(10,5), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('10 Independent Components', fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(ica_10.components_[i].reshape(32, 32).T, cmap=plt.cm.gray)
    ax.set_xlabel(i+1)
```




```
ica_25 = FastICA(25, random_state=np.random.seed(2)).fit(faces)

fig, axes = plt.subplots(5, 5, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('25 Independent Components', fontsize='16')
```

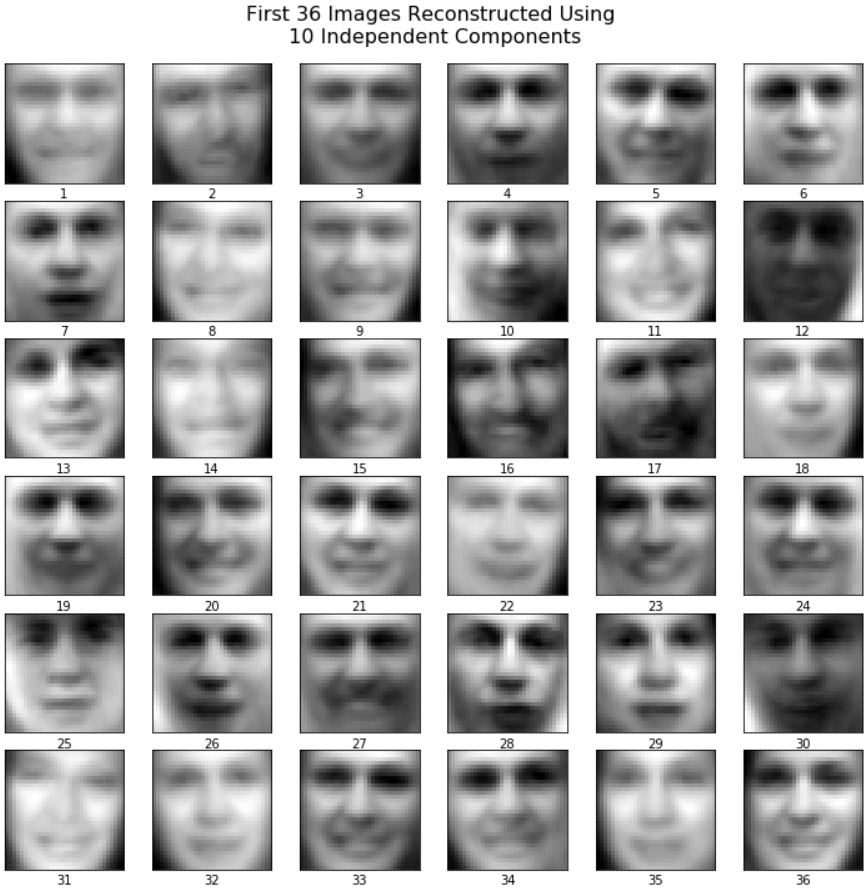
```
for i, ax in enumerate(axes.flat):
    ax.imshow(ica_25.components_[i].reshape(32, 32).T, cmap=plt.cm.gray)
    ax.set_xlabel(i+1)
```


```
ica_50 = FastICA(50, random_state=np.random.seed(2)).fit(faces)
fig, axes = plt.subplots(10, 5, figsize=(10,20), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .96])
fig.suptitle('50 Independent Components', fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(ica_50.components_[i].reshape(32, 32).T, cmap=plt.cm.gray)
    ax.set_xlabel(i+1)
```

Since we have obtained our independent components, we canmove to the reconstruciton phase.

```
ica_10_reconstructed = ica_10.fit(faces).transform(faces).dot(ica_10.
 ↪mixing_.T) + ica_10.mean_

fig, axes = plt.subplots(6, 6, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('First 36 Images Reconstructed Using\n 10 Independent␣
 ↪Components', fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(ica_10_reconstructed[i].reshape(32, 32).T, cmap=plt.cm.
 ↪gray)
    ax.set_xlabel(i+1)

ica_10_mse = (ica_10_reconstructed - faces) ** 2
print("Mean of MSE: %f" %np.mean(ica_10_mse))
print("Standard Deviation of MSE: %f" %np.std(np.mean(ica_10_mse,␣
 ↪axis=1)))
```

```
Mean of MSE: 523.241745
Standard Deviation of MSE: 257.641200
```



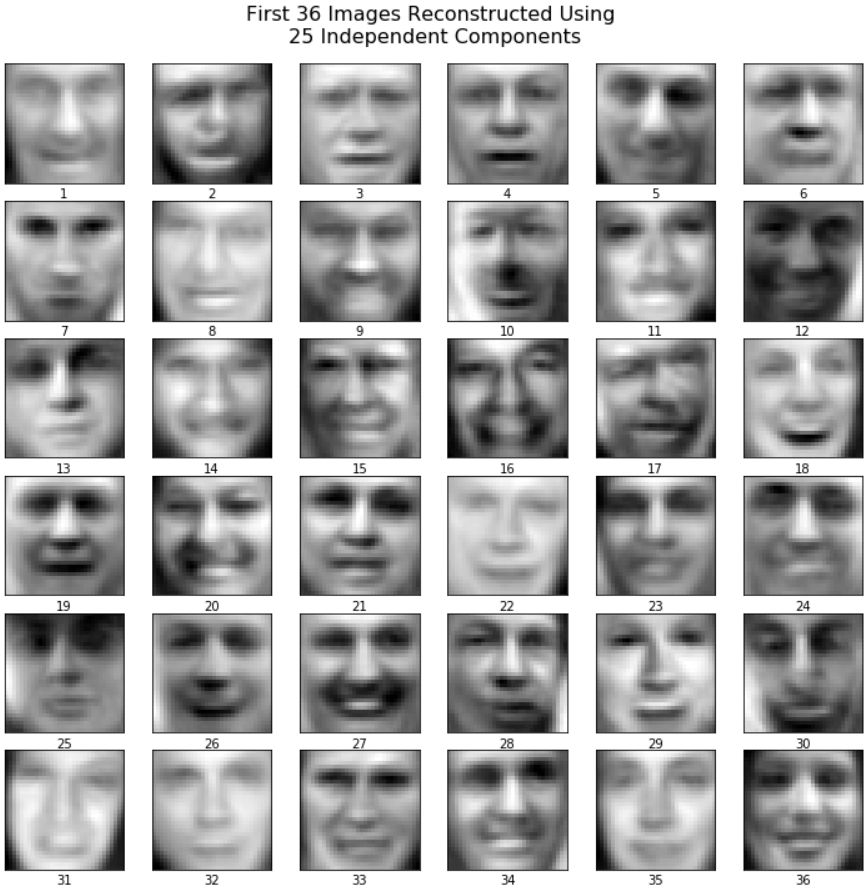
By taking a look at the generated independent components we can say that, curvy parts of our face like, nose, eyes, mouth can be observed at the independent components.

Code for reconstructing the original versions of the images, mean and standard deviation of the errors and reconstructed images are given below in the increasing number of independent components order.

```
ica_25_reconstructed = ica_25.fit(faces).transform(faces).dot(ica_25.
 ↪mixing_.T) + ica_25.mean_

fig, axes = plt.subplots(6, 6, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('First 36 Images Reconstructed Using\n 25 Independent␣
 ↪Components', fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(ica_25_reconstructed[i].reshape(32, 32).T, cmap=plt.cm.
 ↪gray)
```

```
    ax.set_xlabel(i+1)

ica_25_mse = (ica_25_reconstructed - faces) ** 2
print("Mean of MSE: %f" %np.mean(ica_25_mse))
print("Standard Deviation of MSE: %f" %np.std(np.mean(ica_25_mse,␣
 ↪axis=1)))
```

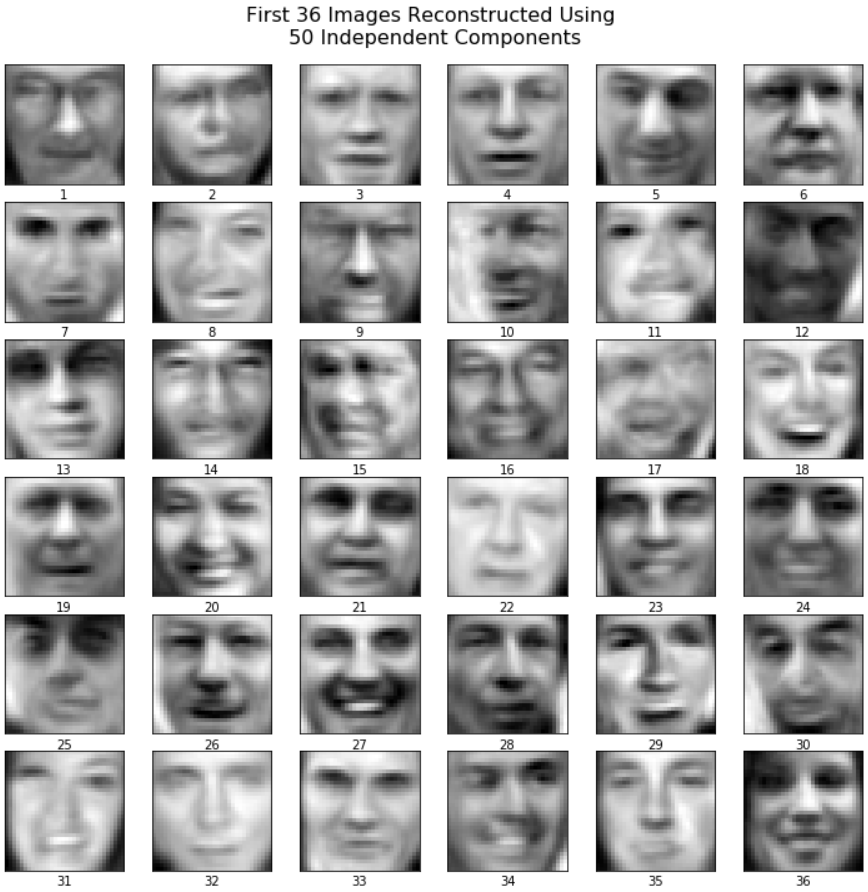
```
Mean of MSE: 332.256492
Standard Deviation of MSE: 153.110288
```




```
ica_50_reconstructed = ica_50.fit(faces).transform(faces).dot(ica_50.
 ↪mixing_.T) + ica_50.mean_

fig, axes = plt.subplots(6, 6, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('First 36 Images Reconstructed Using\n 50 Independent␣
 ↪Components', fontsize='16')
for i, ax in enumerate(axes.flat):
```

```
    ax.imshow(ica_50_reconstructed[i].reshape(32, 32).T, cmap=plt.cm.
 ↪gray)
    ax.set_xlabel(i+1)

ica_50_mse = (ica_50_reconstructed - faces) ** 2
print("Mean of MSE: %f" %np.mean(ica_50_mse))
print("Standard Deviation of MSE: %f" %np.std(np.mean(ica_50_mse,␣
 ↪axis=1)))
```

```
Mean of MSE: 198.425067
Standard Deviation of MSE: 84.179966
```




```
Mean of MSE(10): 523.241745
Mean of MSE(25): 332.256492
Mean of MSE(50): 198.425067

Standard Deviation of MSE(10): 257.641200
Standard Deviation of MSE(25): 153.110288
Standard Deviation of MSE(50): 84.179966
```

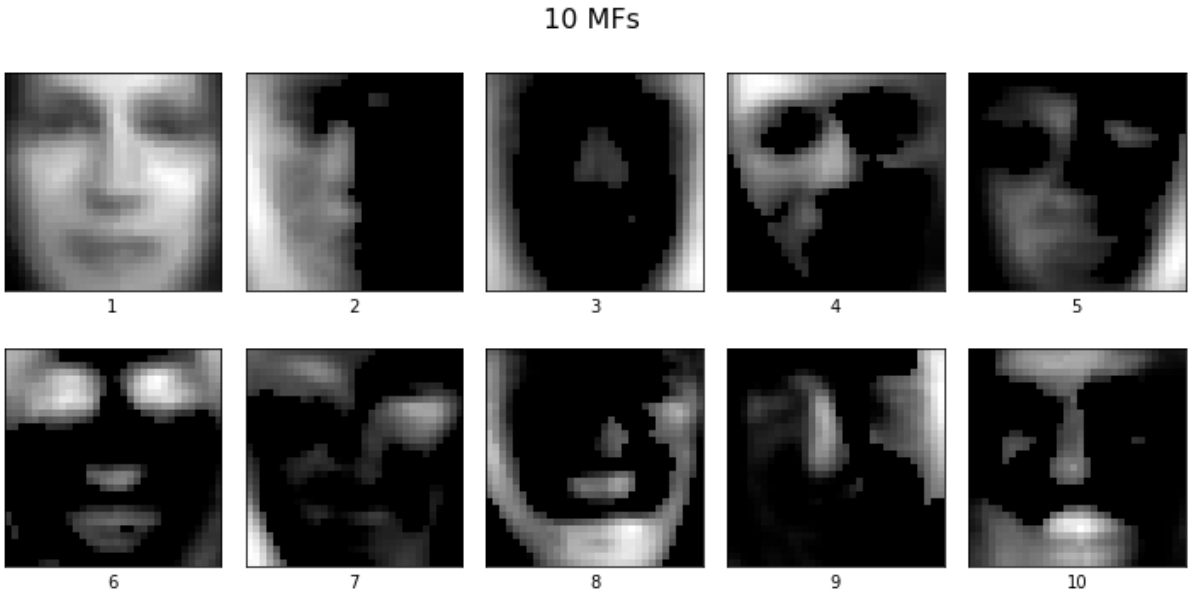
From the results we can say that, we get very similar results to the PCA reconstructions

of the images.

Part D

In this part we will be applying **Non-Negative Matrix Factorization** to reconstruct tho original versions of the images. This technique is conducted by initially making all the pixels non-negative by adding the minimum pixel to all pixels and decomposing the input matrix into two lower rank matrices. To obtain the original versions of the images, those two lower rank matrices will be multiplied and added element will be subtracted. Following three code segments provide us the matrix faces at different amounts expected in the assignment.

```
nnmf_faces = faces + np.abs(np.min(faces))
nnmf_10 = NMF(n_components=10, solver="mu", max_iter=500)
nnmf_10_w = nnmf_10.fit(nnmf_faces).transform(nnmf_faces)

fig, axes = plt.subplots(2, 5, figsize=(10,5), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('10 MFs', fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(nnmf_10.components_[i].reshape(32, 32).T, cmap=plt.cm.
 ↪gray)
    ax.set_xlabel(i+1)
```




```
nnmf_25 = NMF(n_components=25, solver="mu", max_iter=1000)
nnmf_25_w = nnmf_25.fit(nnmf_faces).transform(nnmf_faces)
```

```
fig, axes = plt.subplots(5, 5, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('25 MFs', fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(nnmf_25.components_[i].reshape(32, 32).T, cmap=plt.cm.
 ↪gray)
    ax.set_xlabel(i+1)
```

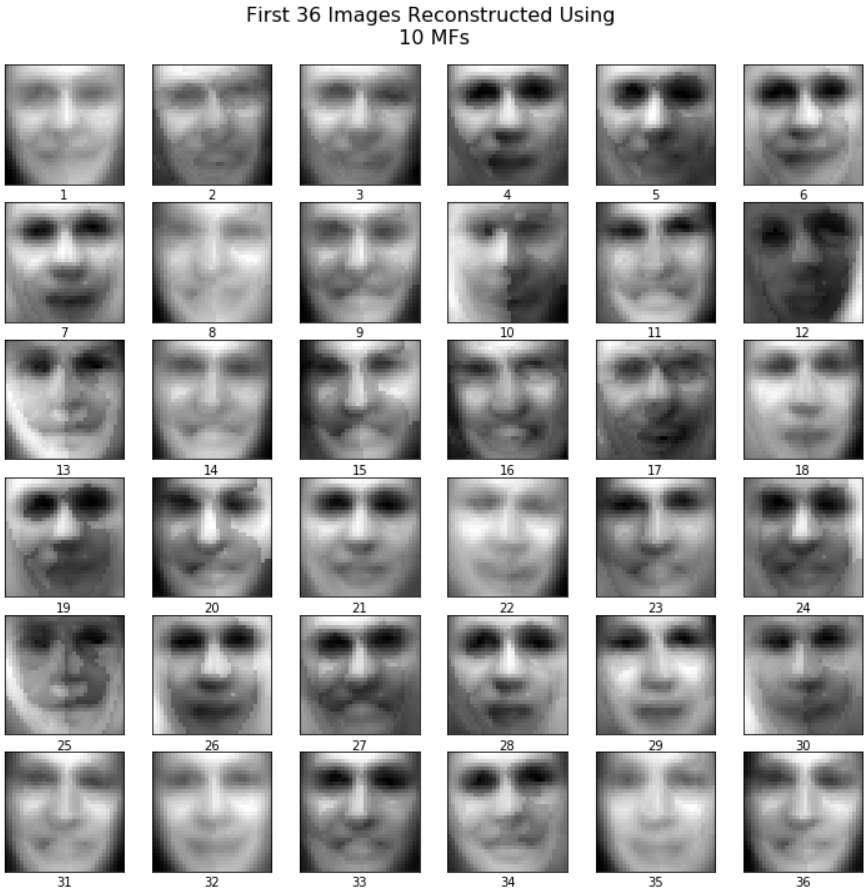

```
nnmf_faces = faces + np.abs(np.min(faces))
nnmf_50 = NMF(n_components=50, solver="mu", max_iter=500)
nnmf_50_w = nnmf_50.fit(nnmf_faces).transform(nnmf_faces)

fig, axes = plt.subplots(10, 5, figsize=(10,20), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .96])
fig.suptitle('50 MFs', fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(nnmf_50.components_[i].reshape(32, 32).T, cmap=plt.cm.
 ↪gray)
```

```
    ax.set_xlabel(i+1)
```

Since we have obtained the matrix faces, we can move to the reconstruction phase. Using those obtained matrix faces, we will be applying the stated procedure to obtain the original versions of the images and also mean and standard deviations of the reconstructed images will be computed in order to make it easier for us to compare it with two other methods. Following three segments are reconstructing the images using non-negative factorization matrix method and displaying the results. Also, they provide mean and standard deviations of the errors of those reconstructed images.

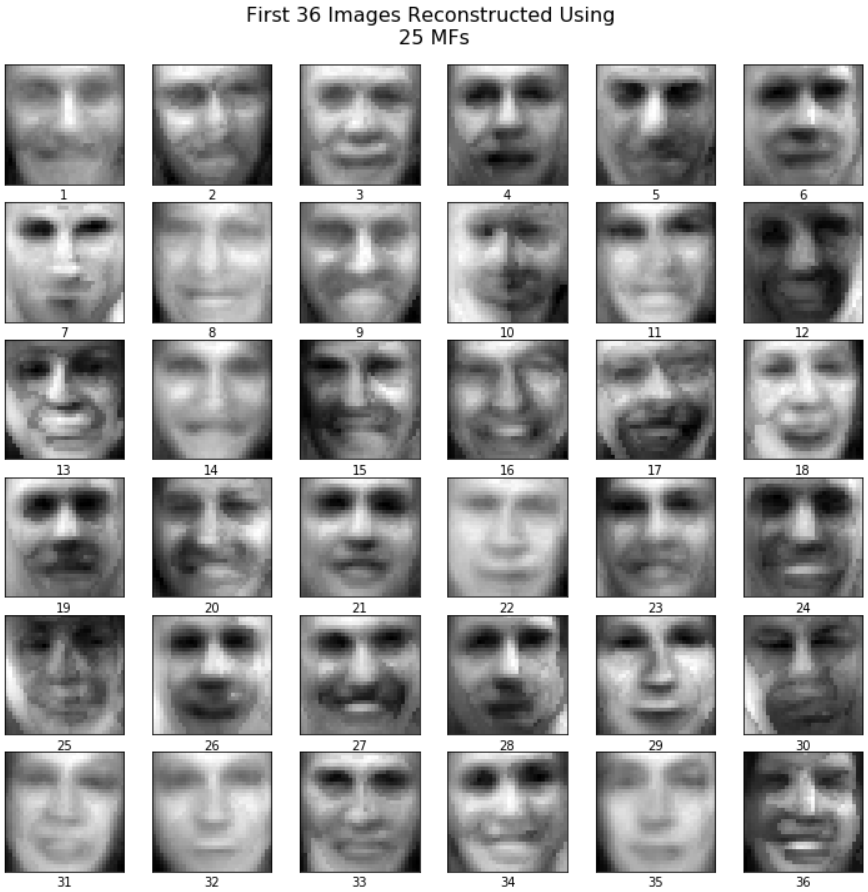
```
nnmf_10_reconstructed = nnmf_10_w.dot(nnmf_10.components_) - np.abs(np.
 ↪min(faces))

fig, axes = plt.subplots(6, 6, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('First 36 Images Reconstructed Using\n 10 MFs',␣
 ↪fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(nnmf_10_reconstructed[i].reshape(32, 32).T, cmap=plt.cm.
 ↪gray)
    ax.set_xlabel(i+1)

nnmf_10_mse = (nnmf_10_reconstructed - faces) ** 2
print("Mean of MSE: %f" %np.mean(nnmf_10_mse))
print("Standard Deviation of MSE: %f" %np.std(np.mean(nnmf_10_mse,␣
 ↪axis=1)))
```

```
Mean of MSE: 666.090782
Standard Deviation of MSE: 375.586055
```




```
nnmf_25_reconstructed = nnmf_25_w.dot(nnmf_25.components_) - np.abs(np.
 ↪min(faces))

fig, axes = plt.subplots(6, 6, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('First 36 Images Reconstructed Using\n 25 MFs',␣
 ↪fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(nnmf_25_reconstructed[i].reshape(32, 32).T, cmap=plt.cm.
 ↪gray)
    ax.set_xlabel(i+1)

nnmf_25_mse = (nnmf_25_reconstructed - faces) ** 2
print("Mean of MSE: %f" %np.mean(nnmf_25_mse))
print("Standard Deviation of MSE: %f" %np.std(np.mean(nnmf_25_mse,␣
 ↪axis=1)))
```

```
Mean of MSE: 488.767425
Standard Deviation of MSE: 265.883010
```




```
nnmf_50_reconstructed = nnmf_50_w.dot(nnmf_50.components_) - np.abs(np.
 ↪min(faces))

fig, axes = plt.subplots(6, 6, figsize=(10,10), facecolor='white',␣
 ↪subplot_kw={'xticks':[], 'yticks':[]})
fig.tight_layout(rect=[0, 0, 1, .93])
fig.suptitle('First 36 Images Reconstructed Using\n 50 MFs',␣
 ↪fontsize='16')
for i, ax in enumerate(axes.flat):
    ax.imshow(nnmf_50_reconstructed[i].reshape(32, 32).T, cmap=plt.cm.
 ↪gray)
    ax.set_xlabel(i+1)

nnmf_50_mse = (nnmf_50_reconstructed - faces) ** 2
print("Mean of MSE: %f" %np.mean(nnmf_50_mse))
print("Standard Deviation of MSE: %f" %np.std(np.mean(nnmf_50_mse,␣
 ↪axis=1)))
```

```
Mean of MSE: 355.523034
Standard Deviation of MSE: 192.786914
```


```
Mean of MSE(10): 666.090782
Mean of MSE(25): 488.767425
Mean of MSE(50): 355.523034
Standard Deviation of MSE(10): 375.586055
Standard Deviation of MSE(25): 265.883010
Standard Deviation of MSE(50): 192.786914
```

From the results of the errors and the reconstructed images, we can clearly state that this method is not as good as the others or for some reasons it did not applied to this set of data as well as the others.

Question 2

In this question, we will be considering 21 independent neurons with gaussian-shaped tuning curves and the function to obtain those curves are provided in the assignment to us as follows:

$$f_i(x) = A \cdot e^{\frac{-(x-\mu_i)^2}{2\sigma_i^2}}$$

Also, we are provided that $A = 1$, $\sigma_i = 1$ and μ_i's are evenly spaced starting from -10 to 10 along the x axis.

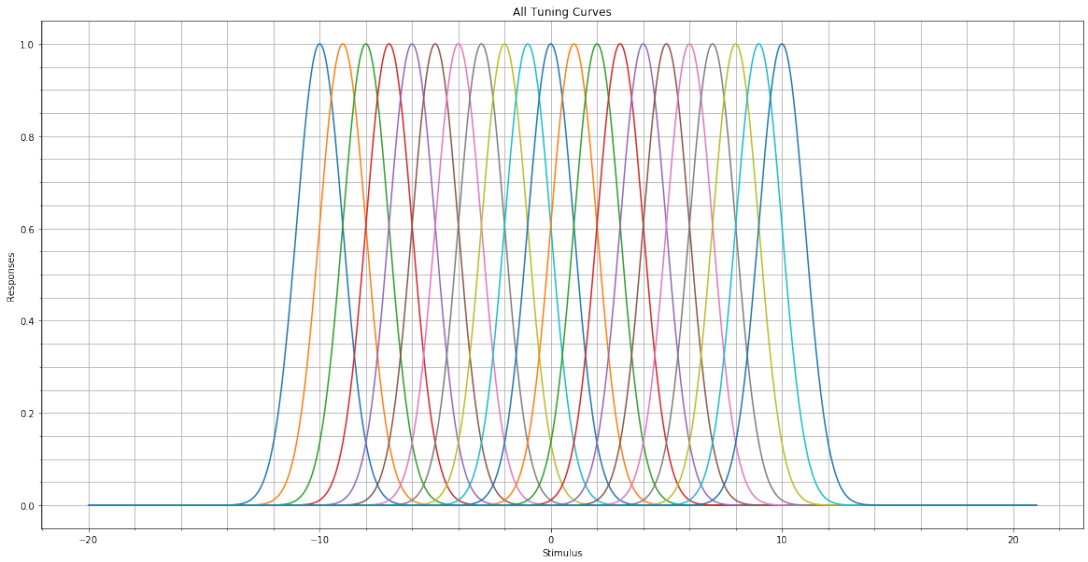
Part A

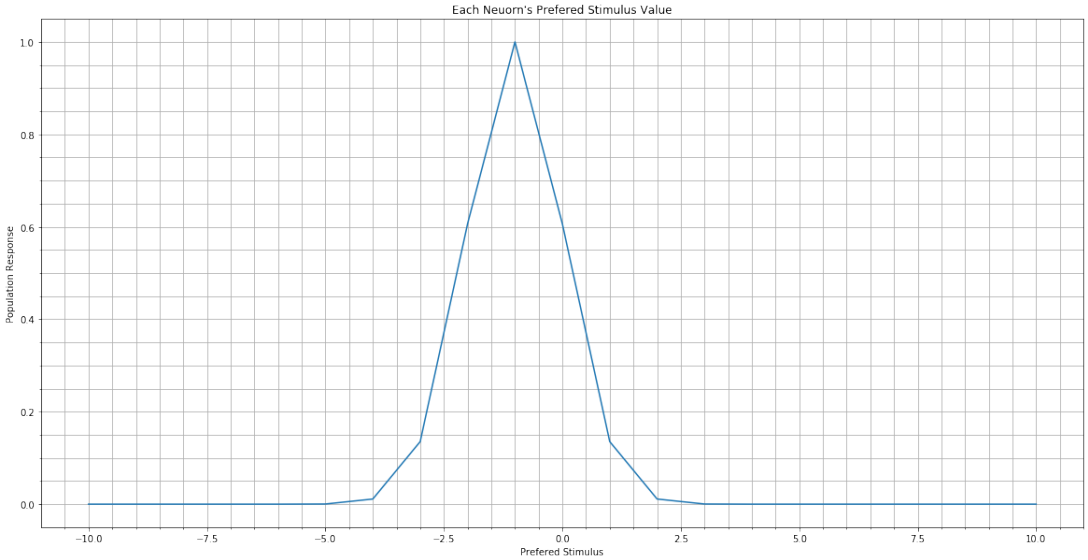
Initially, I have defined a function to tune the curve for a neuron and using it, I have plotted all of the curves of the neurons in our population.

Afterwards, as expected in the assignment I have simulated the response of each neuron to stimulus x=1 and found their preferred stimulus value.

Code of this implementation and respective graphs are provided below.

```
mu_values = np.arange(-10, 11, 1)
def tuning_curves(x, mu, sigma=1):
    return np.exp(-((x - mu) ** 2) / (2 * sigma**2))

stimulus = np.linspace(-20, 21, 1000)

plt.figure(figsize=(20,10))
for mu in mu_values:
    plt.plot(stimulus, tuning_curves(stimulus, mu))
plt.minorticks_on()
plt.grid(which='both')
plt.xlabel('Stimulus')
plt.ylabel('Responses')
plt.title('All Tuning Curves')
plt.show(block=False)

plt.figure(figsize=(20,10))
plt.plot(mu_values, tuning_curves(-1, mu_values))
plt.minorticks_on()
plt.grid(which='both')
plt.xlabel('Prefered Stimulus')
plt.ylabel('Population Response')
plt.title('Each Neuorn\'s Prefered Stimulus Value' )
plt.show(block=False)
```





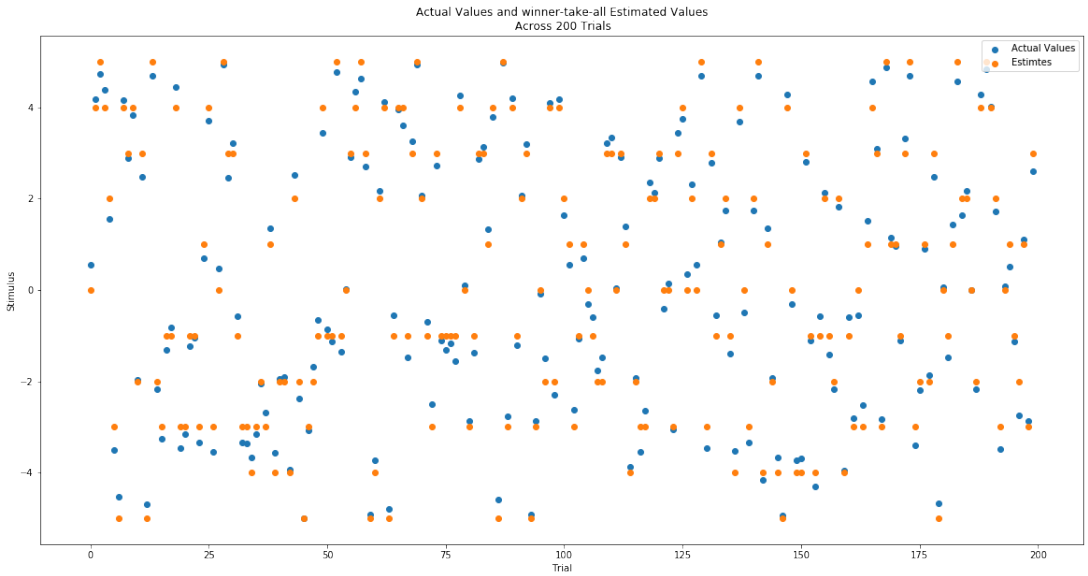
Part B

In this part and the following parts we will be simulating 200 trials and with respect to these trials, using different types of decoders, we will try to estimate the input stimulus.

Also, we are expected to take some additive gaussian noise with zero mean and $\frac{1}{20}$ standard deviation into account while making those estimates.

Specific to this part, we will be implementing a winner-take-all decoder. This type of decoder can be explained as it will estimate the input stimuli via the neuron with the highest response. Code implementation of this decoder and estimations of this procedure is given in the following code segment. Also, graph of these estimations and computed mean and standard deviation of error in stimulus is given for further discussion and comparison between the decoders.

```
stimulus_interval = np.linspace(-5, 5, 1000)

def wta_decoder(pref_stim, response):
    return pref_stim[np.argmax(response)]

stimulus = []
responses = []
wta_est = []
wta_error = []

np.random.seed(6)
for i in range(200):
    gaussian_noise = np.random.normal(0, 1/20, 21)
    stimulus.append(np.random.choice(stimulus_interval))
    response = tuning_curves(stimulus[i], mu_values)
    response_with_noise = response + gaussian_noise
    responses.append(response_with_noise)
    wta_est.append(wta_decoder(mu_values, response_with_noise))
    wta_error.append(np.abs(stimulus[i] - wta_est[i]))

plt.figure(figsize=(20,10))
plt.scatter(np.arange(200), stimulus)
plt.scatter(np.arange(200), wta_est)
plt.xlabel('Trial')
plt.ylabel('Stimulus')
plt.title('Actual Values and winner-take-all Estimated Values\n Across␣
 ↪200 Trials')
plt.legend(['Actual Values', 'Estimtes'], loc='upper right')
plt.show(block=False)
print("Winner-take-all decoder estimate error mean: %f" %np.
 ↪mean(wta_error))
print("Winner-take-all decoder estimate error standard deviation: %f"␣
 ↪%np.std(wta_error))
```




```
Winner-take-all decoder estimate error mean: 0.268754
Winner-take-all decoder estimate error standard deviation: 0.156388
```

Part C

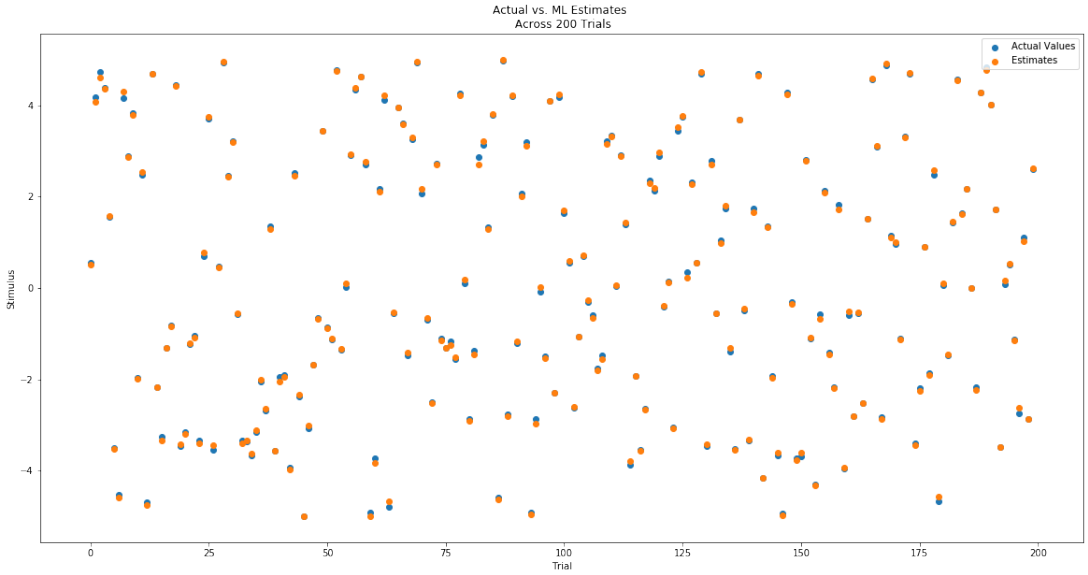
In this part same procedure is applied just with a different decoder. This time **Maximum Likelihood Decoder** is used for estimation purposes. As provided as a hint in the assignment we will be using log-likelihood for our stimulus interval, in our ML Decoder.

Codes for estimation, graph and computing errors are given in the following code segment.

```
def ml_decoder(stim_interval, response, sigma=1):
    log_likelihood = []
    for s in stim_interval:
        log_l_val = 0
        for r, mu in zip(response, mu_values):
            log_l_val += (r - tuning_curves(s, mu)) ** 2
        log_likelihood.append(log_l_val)
    return stim_interval[np.argmin(log_likelihood)]
```

```
stimulus_mle, mle_error = [], []
for resp, stim in zip(responses, stimulus):
    stimulus_mle.append(ml_decoder(stimulus_interval, resp))
    mle_error.append(float(np.abs(stim -␣
 ↪stimulus_mle[len(stimulus_mle)-1])))
```

```
plt.figure(figsize=(20,10))
plt.scatter(np.arange(200), stimulus)
plt.scatter(np.arange(200), stimulus_mle)
plt.xlabel('Trial')
plt.ylabel('Stimulus')
plt.title('Actual vs. ML Estimates \n Across 200 Trials')
plt.legend(['Actual Values', 'Estimates'], loc='upper right')
plt.show(block=False)
print("ML decoder estimate error mean: %f" %np.mean(mle_error))
print("ML decoder estimate error standard deviation: %f" %np.
 ↪std(mle_error))
```




```
ML decoder estimate error mean: 0.041792
ML decoder estimate error standard deviation: 0.032628
```

Part D

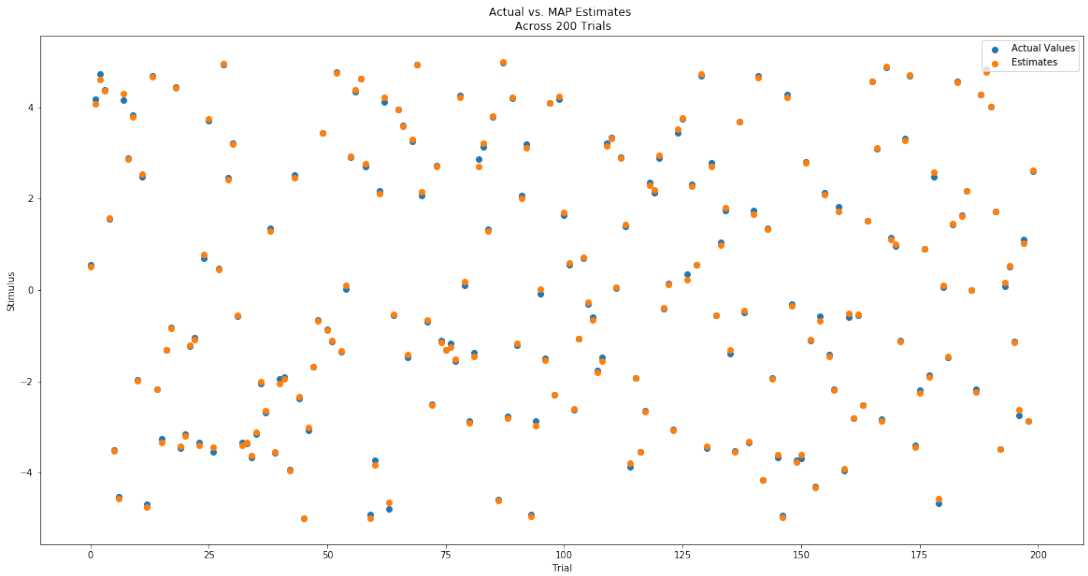
In this part we will be again applying the same procedure with a **Maximum-a-Posteriori Decoder**. Main principle behinde MAP decoders is to find the input with the maximized posterior probability.

Resulting mean and standard deviations of error is given with the graph of the estimates and actual values after the code segment repreeenting the procedure.

```
def map_decoder(stim_interval, response):
    log_posterior = []
```

```
    for stim in stim_interval:
        log_post_val = 0
        for r, mu in zip(response, mu_values):
            log_post_val += (r - tuning_curves(stim, mu)) ** 2
        log_post_val = log_post_val * 200 + (stim ** 2) / 10
        log_posterior.append(log_post_val)
    return stim_interval[np.argmin(log_posterior)]
```

```
stimulus_map, map_error = [], []
for resp, stim in zip(responses, stimulus):
    stimulus_map.append(map_decoder(stimulus_interval, resp))
    map_error.append(float(np.abs(stim -␣
 ↪stimulus_map[len(stimulus_map)-1])))

plt.figure(figsize=(20,10))
plt.scatter(np.arange(200), stimulus)
plt.scatter(np.arange(200), stimulus_map)
plt.xlabel('Trial')
plt.ylabel('Stimulus')
plt.title('Actual vs. MAP Estimates \n Across 200 Trials')
plt.legend(['Actual Values', 'Estimates'], loc='upper right')
plt.show(block=False)
print("MAP decoder estimate error mean: %f" %np.mean(map_error))
print("MAP decoder estimate error standard deviation: %f" %np.
 ↪std(map_error))
```




```
MAP decoder estimate error mean: 0.041542
```

```
MAP decoder estimate error standard deviation: 0.032556
```

Part E

In this part we are asked to apply the same procedure using ML Decoder. But this time, σ values will be changing according to the values given in the assignment. We will be comparing their mean and standard deviations of error in order to make a decision between narrow or wide tuning curves.

```
sigma_i = [.1, .2, .5, 1, 2, 5]
xml_mle_err = []

np.random.seed(3)
for i in range(200):
    stim = np.random.choice(stimulus_interval)
    xi_err = []
    for sig in sigma_i:
        resp = tuning_curves(stim, mu_values, sig) + np.random.
 ↪normal(0, 1/20, 21)
        xi_err.append(np.abs(stim - float(ml_decoder(stimulus_interval,␣
 ↪resp, sig))))
    xml_mle_err.append(np.array(xi_err))

xml_mle_err = np.array(xml_mle_err)

print("Mean of errors in MLE (for Sigma = 0.1): %f" %np.
 ↪mean(xml_mle_err[:, 0]))
print("STD of errors in MLE (for Sigma = 0.1): %f" %np.std(xml_mle_err[:
 ↪, 0]))

print("Mean of errors in MLE (for Sigma = 0.2): %f" %np.
 ↪mean(xml_mle_err[:, 1]))
print("STD of errors in MLE (for Sigma = 0.2): %f" %np.std(xml_mle_err[:
 ↪, 1]))

print("Mean of errors in MLE (for Sigma = 0.5): %f" %np.
 ↪mean(xml_mle_err[:, 2]))
print("STD of errors in MLE (for Sigma = 0.5): %f" %np.std(xml_mle_err[:
 ↪, 2]))

print("Mean of errors in MLE (for Sigma = 1): %f" %np.mean(xml_mle_err[:
 ↪, 3]))
print("STD of errors in MLE (for Sigma = 1): %f" %np.std(xml_mle_err[:,␣
 ↪3]))
```

```
print("Mean of errors in MLE (for Sigma = 2): %f" %np.mean(xml_mle_err[:
 ↪, 4]))
print("STD of errors in MLE (for Sigma = 2): %f" %np.std(xml_mle_err[:,␣
 ↪4]))

print("Mean of errors in MLE (for Sigma = 5): %f" %np.mean(xml_mle_err[:
 ↪, 5]))
print("STD of errors in MLE (for Sigma = 5): %f" %np.std(xml_mle_err[:,␣
 ↪5]))
```

```
Mean of errors in MLE (for Sigma = 0.1): 1.971171
STD of errors in MLE (for Sigma = 0.1): 2.290644
Mean of errors in MLE (for Sigma = 0.2): 0.418118
STD of errors in MLE (for Sigma = 0.2): 0.925003
Mean of errors in MLE (for Sigma = 0.5): 0.057157
STD of errors in MLE (for Sigma = 0.5): 0.036964
Mean of errors in MLE (for Sigma = 1): 0.041041
STD of errors in MLE (for Sigma = 1): 0.030869
Mean of errors in MLE (for Sigma = 2): 0.079930
STD of errors in MLE (for Sigma = 2): 0.060876
Mean of errors in MLE (for Sigma = 5): 0.390090
STD of errors in MLE (for Sigma = 5): 0.298560
```