Homework 4 Report

Ayhan Okuyan
ayhan.okuyan[at]ug.bilkent.edu.tr

May 10, 2020

Contents

Question 1 **2**
Part A 2
Part B 4
Part C 7
Part D 11

Question 2 **17**
Part A 17
Part B 18
Part C 20
Part D 23
Part E 25

Appendix A - Python Code **27**

Question 1

The question provides the $1024x1000$ data matrix called **faces**, which contains 1000 grayscale face images in a $32x32$ square grid. A sample stimuli (face image) is provided below.

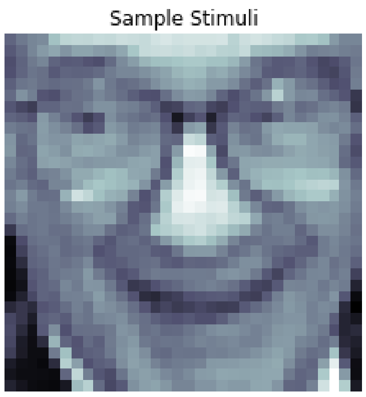

Figure 1: Sample Stimuli

Question revolves around applying known dimensionality reduction techniques given as PCA (Principal Component Analysis), ICA (Independent Component Analysis) and NNMF (Non-negative Matrix Factorization). The question provides some one MATLAB application in the name `FastICA_2.5` to run ICA algorithm and `dispImArray.m` to display image subplots. However, since this assignment is implemented in Python, for ICA and all other reduction techniques, `sklearn.decomposition` library is used. The subplots are plotted organically using `matplotlib.pyplot`. The dependencies for this question are provided below.

```python
import numpy as np
import matplotlib.pyplot as plt
import h5py
import sklearn.decomposition as decomp
```

Part A This part asks us to implement PCA, which is the eigenvalue decomposition of the covariance matrix of the data after a whitening procedure. Then, k biggest eigenvalues are selected with their corresponding eigenvectors, which are now called **principle components** (PCs), where the k value is generally chosen by how much of the variance of the original data is intended to be kept. The mathematical representation of PCA is provided below, where X is the original data matrix and μ_x is the feature-wise mean vector.

$$\tilde{X} = X - \mu_X \tag{1}$$

$$\Sigma = \tilde{X}^T \tilde{X} \tag{2}$$

Then define the eigen-decomposition of the covariance matrix as,

$$\Sigma = \Lambda D \Lambda^T \tag{3}$$

where, Λ is the matrix that contains eigenvectors at its columns and D is the diagonal matrix containing the eigenvalues in its diagonal. Assuming we would like to progress PCA with k PCs, we define Λ_k which are the eigenvectors of that correspond to k biggest eigenvalues. To reconstruct, the original data is simply projected onto the eigenspace created by the principle components to obtain a lower-dimensional representation of the data, which is given by the following formula.

$$\hat{X} = \tilde{X}\Lambda_k\Lambda_k^T + \mu_X \tag{4}$$

This part requires us to perform PCA on 1000 face images and then plot the proportion of variance explained for the first 100 PCs and then show the first 25 Principle Components on a 32x32 grid, which are then called as **eigenfaces** y convention. The outputs are provided below following the code.

```
pca_data = decomp.PCA(100, whiten=True)
pca_data.fit(faces.T)
plt.plot(pca_data.explained_variance_ratio_)
plt.xlabel('Principal Components')
plt.ylabel('Proportion of Variance Explained')
plt.title('Proportion of Explained Variance\n for Each Individual
                                    Principle Component')
plt.grid()
plt.show()
```

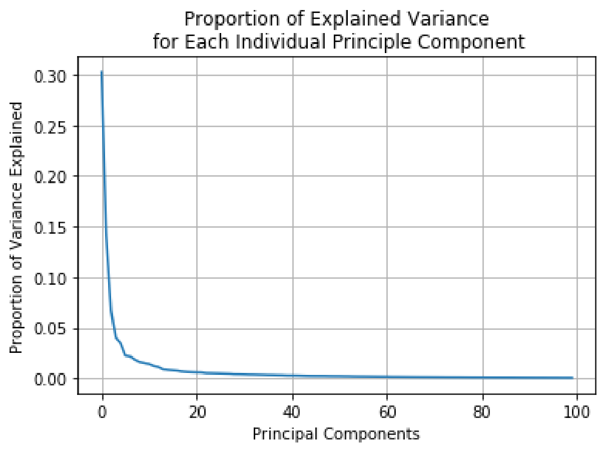

Figure 2: Proportion of Variance Explained vs Principle Components

It can be seen from the figure that the, the first 10 PCs explain most of the variability in the data, which is what makes PCA important as we can reduce dimension significantly by keeping most of the variance. One more thing to consider is that as all explained variances are added up, the sum would be equal to one, which is understandable since if we keep all of the PCs, we would reconstruct the original data. Then, we show the first 25 PCs as given below.

```
plt.figure(figsize=(9,9))
for i in range(1,26):
    plt.subplot(5,5,i)
    plt.imshow(pca_data.components_[i-1].reshape((32,32)).T, cmap=
                                                    'bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()
```

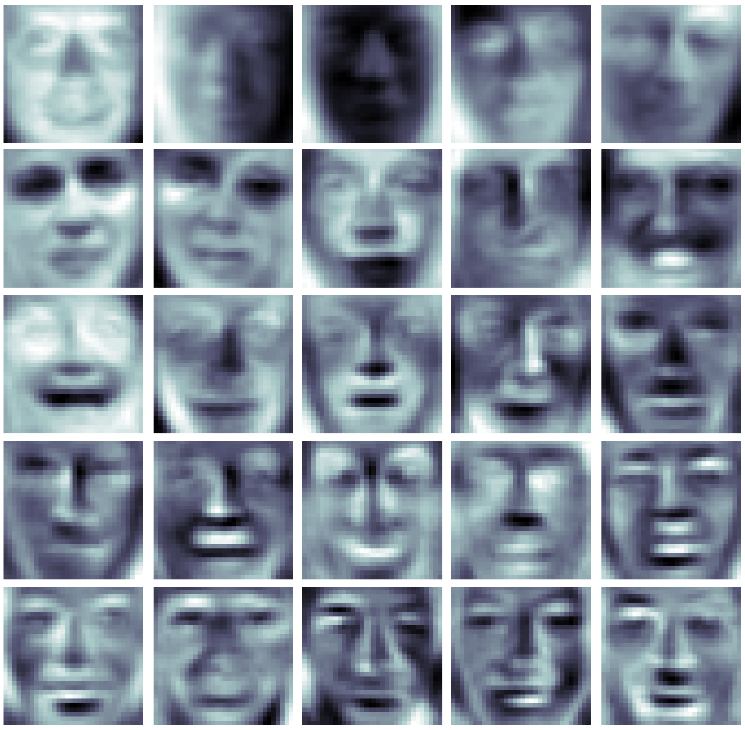

Figure 3: First 25 Principle Components

These are the principle components that explain human face the most. Here, we observe that while all components capture the overall structure of a face, some parts are highlighted while some are dimmed.

Part B This part asks us to reconstruct the `faces` matrix using 10, 25 and 50 PCs and then display the first 36 images with their reconstructions. The following code is presented to reconstruct the images as explained. The plots are provided below, for explicit code, refer to Appendix A.

```
faces_rec10 = (faces - pca_data.mean_.reshape(1024,1)).T.dot(
                                              pca_data.components_[0:10].T).
                                              dot(pca_data.components_[0:10]
                                              ) + pca_data.mean_
faces_rec25 = (faces - pca_data.mean_.reshape(1024,1)).T.dot(
                                              pca_data.components_[0:25].T).
                                              dot(pca_data.components_[0:25]
                                              ) + pca_data.mean_
faces_rec50 = (faces - pca_data.mean_.reshape(1024,1)).T.dot(
                                              pca_data.components_[0:50].T).
                                              dot(pca_data.components_[0:50]
                                              ) + pca_data.mean_
```

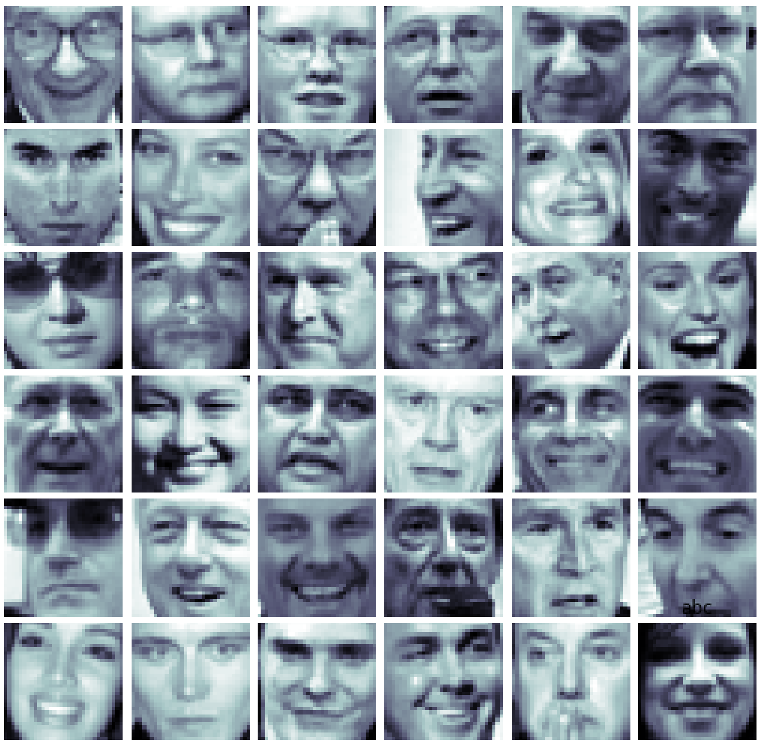

Figure 4: First 36 Images (Original Images)

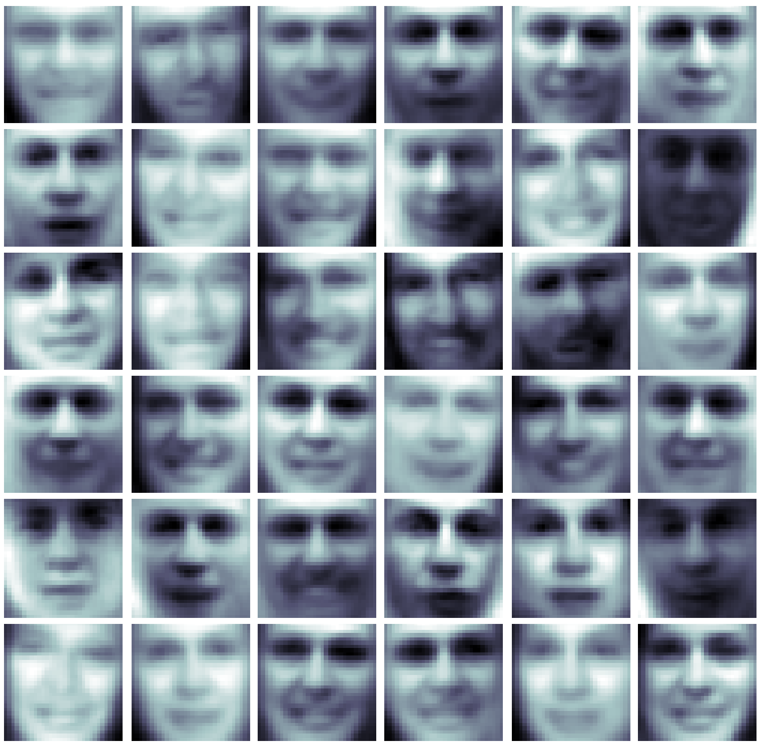

Figure 5: Reconstructed Images (10 PCs)

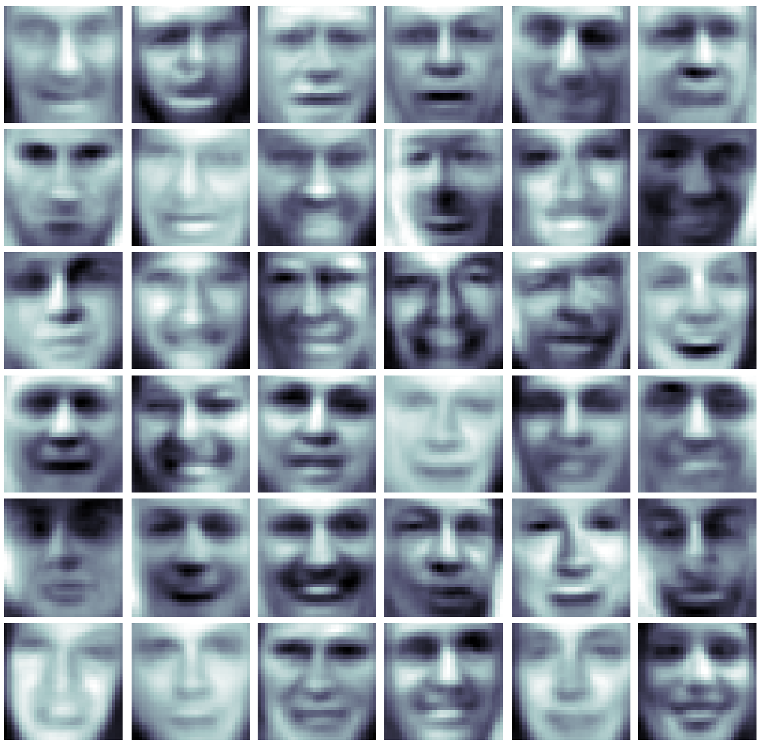

Figure 6: Reconstructed Images (25 PCs)

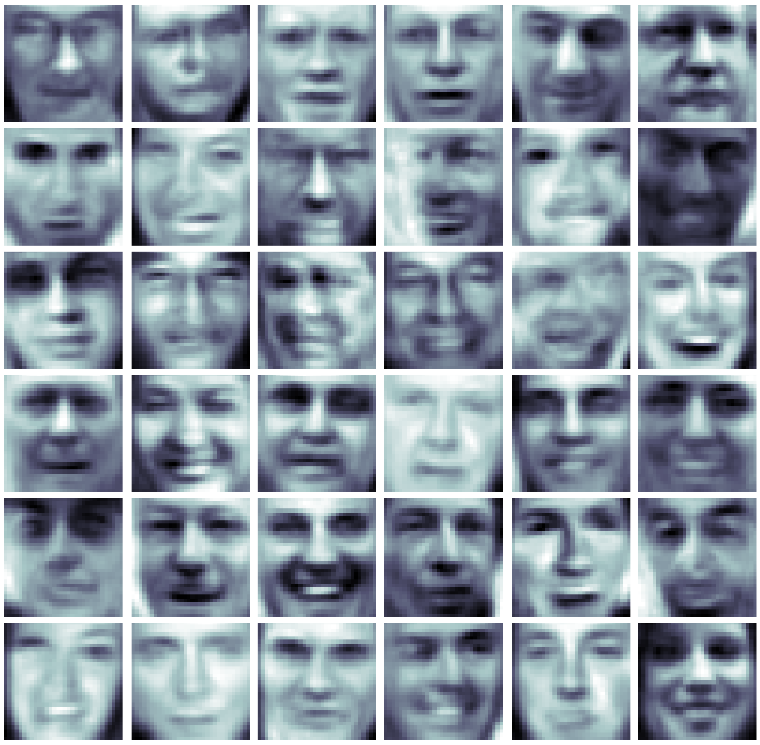

Figure 7: Reconstructed Images (50 PCs)

From the original and reconstructed images, we can say that as the number of PCs increase, the images get closer to their original versions, which is understandable. As PCA is a lossy compression method, we can calculate the mean-squared errors (MSEs) between the original and reconstructed values and observe the mean and standard deviation of the errors as number of ICs change. The code is provided below.

```
mse_10 = np.mean((faces.T- faces_rec10)**2)
std_10 = np.std(np.mean((faces.T - faces_rec10)**2, axis=1))
print('Reconstruction MSE Loss with %d PCs:\nMean: %f\nStd: %f ' %
                                (10, mse_10, std_10))
mse_25 = np.mean((faces.T- faces_rec25)**2)
std_25 = np.std(np.mean((faces.T - faces_rec25)**2, axis=1))
print('Reconstruction MSE Loss with %d PCs:\nMean: %f\nStd: %f ' %
                                (25, mse_25, std_25))
mse_50 = np.mean((faces.T- faces_rec50)**2)
std_50 = np.std(np.mean((faces.T - faces_rec50)**2, axis=1))
print('Reconstruction MSE Loss with %d PCs:\nMean: %f\nStd: %f ' %
                                (50, mse_50, std_50))
```

```
Reconstruction MSE Loss with 10 PCs:
Mean: 523.241745
Std: 257.641199
Reconstruction MSE Loss with 25 PCs:
Mean: 332.256492
Std: 153.110296
Reconstruction MSE Loss with 50 PCs:
Mean: 198.425229
Std: 84.179787
```

These results are in accordance with what is expected since the increase PCs should present a lower mean error with less variability.

Part C In this part, we are asked to apply ICA, which is a statistical method to separate linearly mixed non-Gaussian data. Fast ICA is a non-deterministic method than can result in amplitude and order ambiguities. The part requires us to show 10, 25 and 50 ICs (Independent Components), which is done via following code segment. The scripting involving the plots is presented in Appendix A.

```
rng = np.random.seed(10)
ica10 = decomp.FastICA(10, whiten=True, random_state=rng)
ica25 = decomp.FastICA(25, whiten=True, random_state=rng)
ica50 = decomp.FastICA(50, whiten=True, random_state=rng)
ica10.fit(faces.T)
ica25.fit(faces.T)
ica50.fit(faces.T)
```

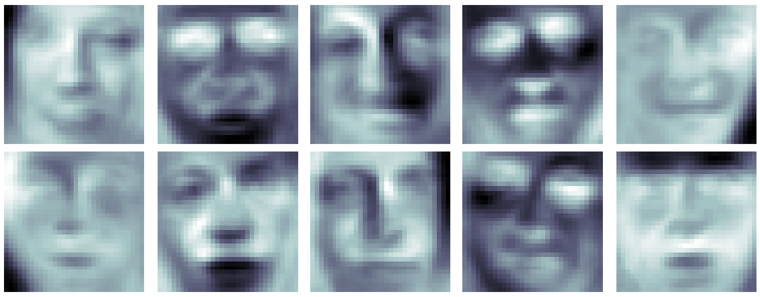
Figure 8: 10 ICs

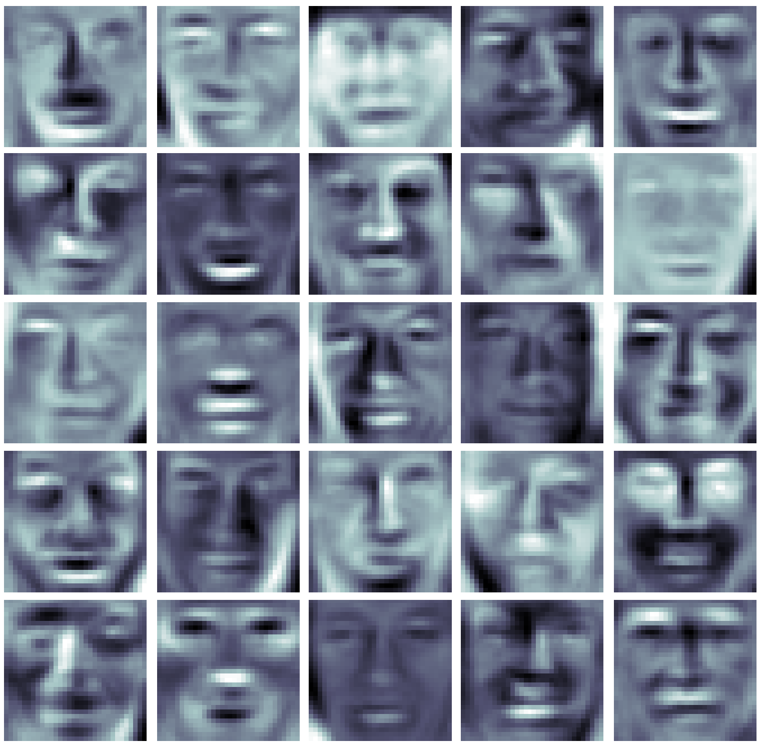
Figure 9: 25 ICs

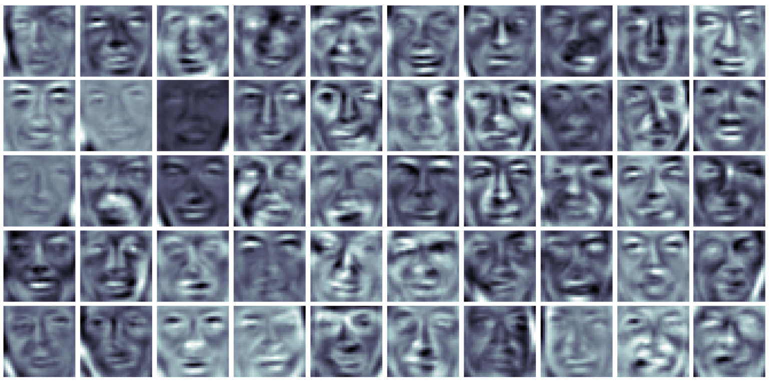
Figure 10: 50 ICs

The ICs, unlike PCA highlight mostly different parts of the face like nose, eyes and cheecks. Then, we reconstruct the stimuli given as below.

```
faces_ica10 = ica10.fit_transform(faces.T).dot(ica10.mixing_.T) +
                                   ica10.mean_
faces_ica25 = ica25.fit_transform(faces.T).dot(ica25.mixing_.T) +
                                   ica25.mean_
faces_ica50 = ica50.fit_transform(faces.T).dot(ica50.mixing_.T) +
                                   ica50.mean_
```

The reconstructed images are as follows and the code snippet showin the first plot is provided below.

```
print('Reconstructed Images with 10 ICs')
plt.figure(figsize=(9,9))
for i in range(1,37):
    plt.subplot(6,6,i)
    plt.imshow(faces_ica10[i-1].reshape((32,32)).T, cmap='bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()
```

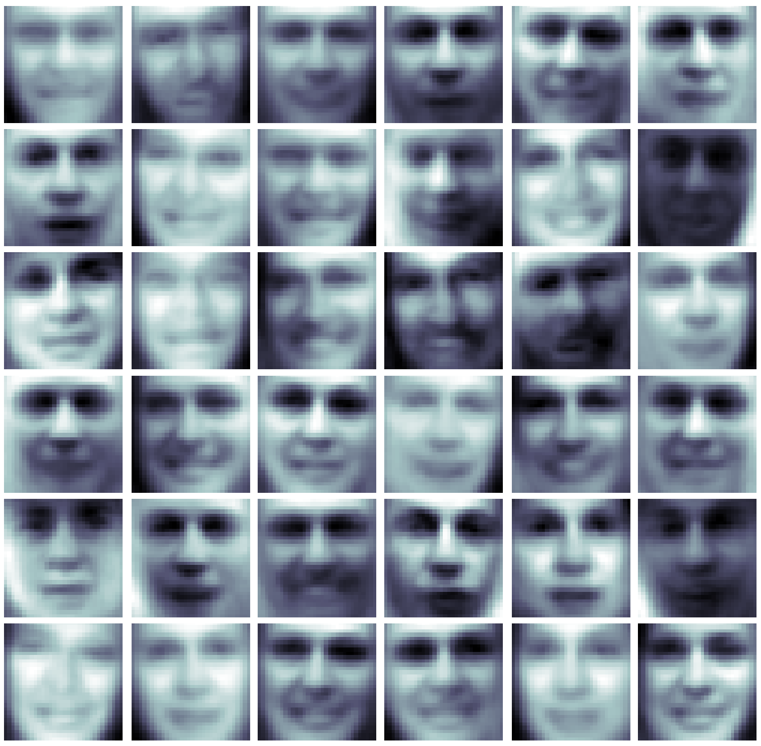

Figure 11: Reconstructed Images (10 ICs)

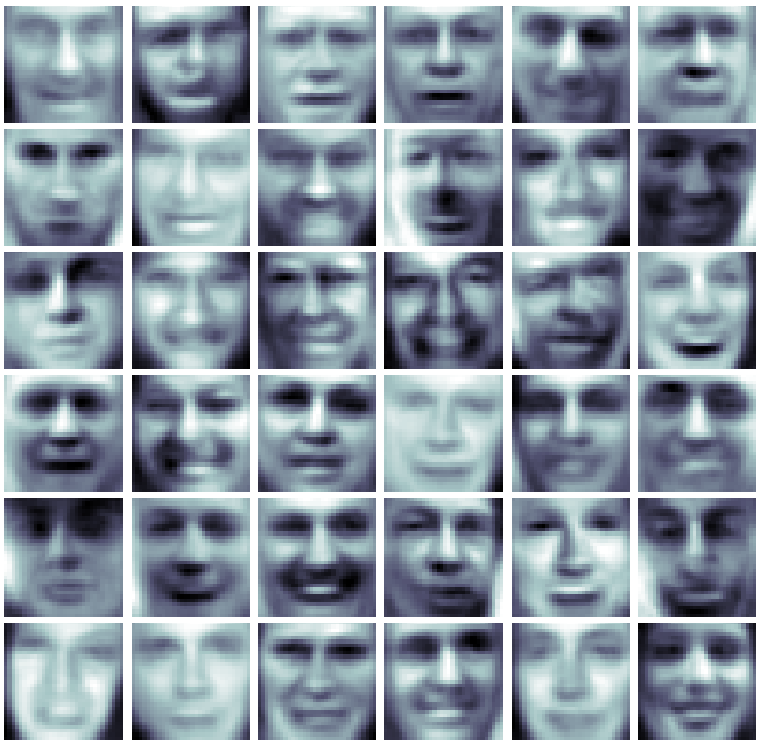
Figure 12: Reconstructed Images (25 ICs)

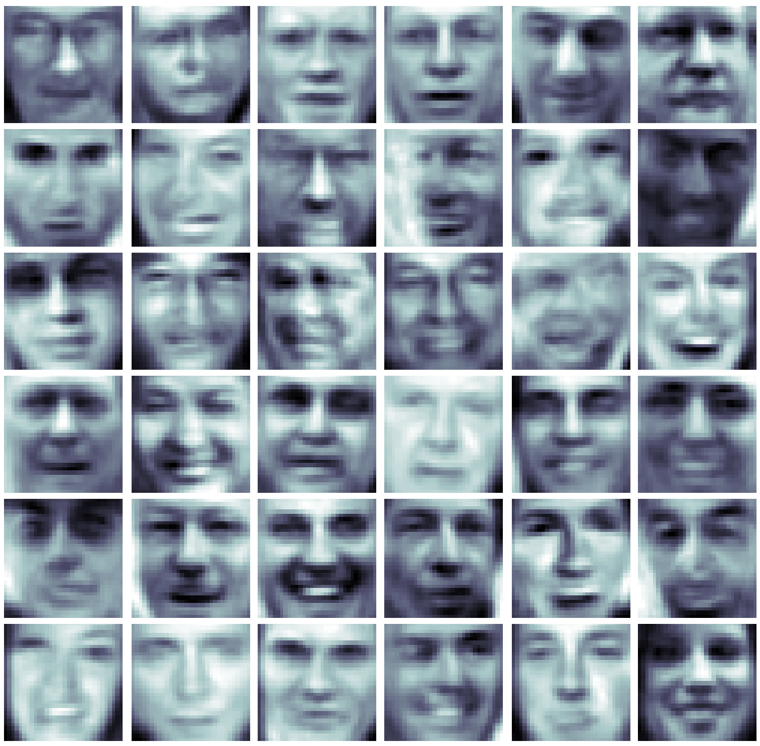
Figure 13: Reconstructed Images (50 ICs)

Then, again we find the mean and standard deviation of the Mean Squared Error.

```
mse_10 = np.mean((faces.T- faces_ica10)**2)
std_10 = np.std(np.mean((faces.T - faces_ica10)**2, axis=1))
print('Reconstruction MSE Loss with %d ICs:\nMean: %f\nStd: %f ' %
                                           (10, mse_10, std_10))
mse_25 = np.mean((faces.T- faces_ica25)**2)
std_25 = np.std(np.mean((faces.T - faces_ica25)**2, axis=1))
print('Reconstruction MSE Loss with %d ICs:\nMean: %f\nStd: %f ' %
                                           (25, mse_25, std_25))
mse_50 = np.mean((faces.T- faces_ica50)**2)
std_50 = np.std(np.mean((faces.T - faces_ica50)**2, axis=1))
print('Reconstruction MSE Loss with %d ICs:\nMean: %f\nStd: %f ' %
                                           (50, mse_50, std_50))
```

```
Reconstruction MSE Loss with 10 ICs:
Mean: 523.241745
Std: 257.641200
Reconstruction MSE Loss with 25 ICs:
Mean: 332.256492
Std: 153.110288
Reconstruction MSE Loss with 50 ICs:
Mean: 198.425067
Std: 84.179966
```

The results here are nearly as equal to the results observed in Part B. This is due to the inability of `sklearn` to provide an option to specify how many ICs to use to reconstruct the images, resulting with the algorithm using them all. In that sense the results come closer to PCA. However, if there were such option it is viable to hypothesize that ICA would perform worse than PCA since by default ICA involves randomization and due to the ICA dimensionality reduction conforms amplitude and sign ambiguity, the reconstruction error would be expected to be higher.

Part D This part, as mentioned in the introduction, asks us to implement NNMF, which is a posed problem to decompose the input matrix whose elements are non-negative as multiplication of two lower-rank matrices. If we address the input matrix as X and the lower-rank decomposition matrices as W and H, we can pose the optimization problem of NNMF as follows.

$$arg \min_{W,H}(||X-(W\cdot H)||)\ subject\ to(W\cdot .H)\geq 0 \quad (5)$$

This minimization process is similar to E-M Clustering in the sense that the problem is NP-Hard under normal circumstances, however, we again use an alternating minimization approach to a gradient descent step to one variable at a time. Here, we use `sklearn`'s `NMF` method to build the models.

```
faces_nn = faces + np.abs(np.min(faces))
nmf10 = decomp.NMF(n_components=10, solver='mu', max_iter=1000)
nmf25 = decomp.NMF(n_components=25, solver='mu', max_iter=1000)
```

```
nmf50 = decomp.NMF(n_components=50, solver='mu', max_iter=1000)
nmf10.fit(faces_nn.T)
nmf25.fit(faces_nn.T)
nmf50.fit(faces_nn.T)
```

Here, we add the minimum value of the original data matrix to scale the data up to a minimum value of zero since the data cannot include negative values in the NNMF algorithm. Then, we show the 10, 25 and 50 MFs provided as follows. The plot related scripts are present in Appendix A.

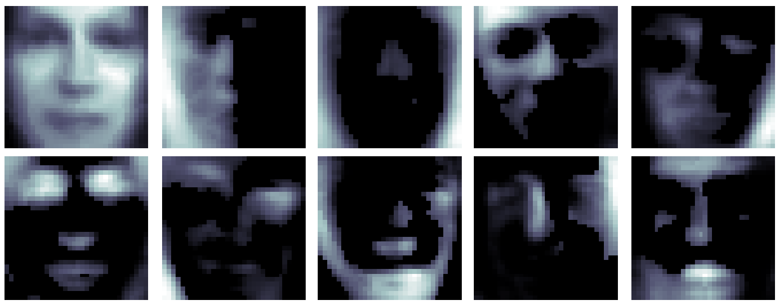
Figure 14: 10 MFs

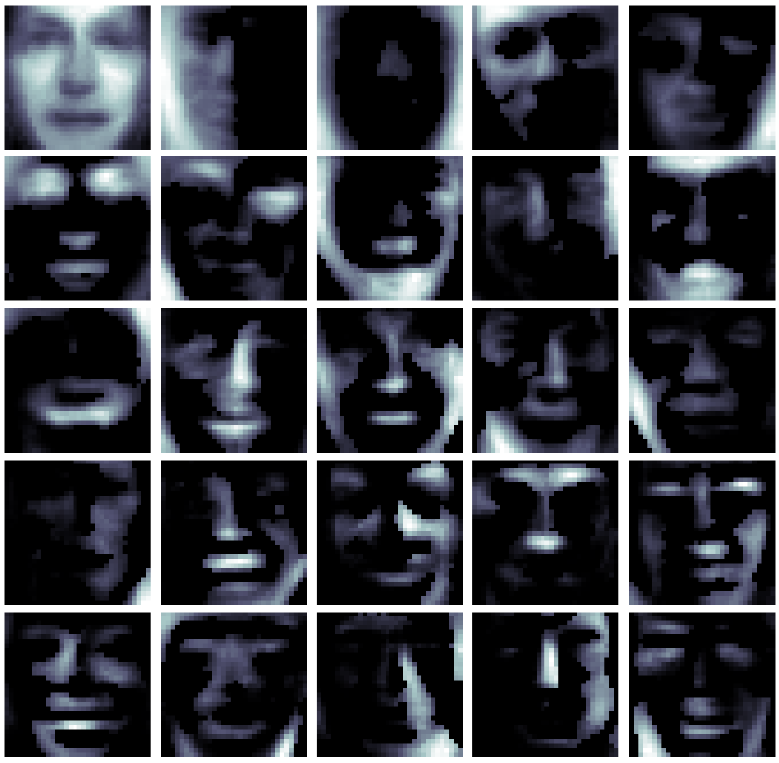
Figure 15: 25 MFs

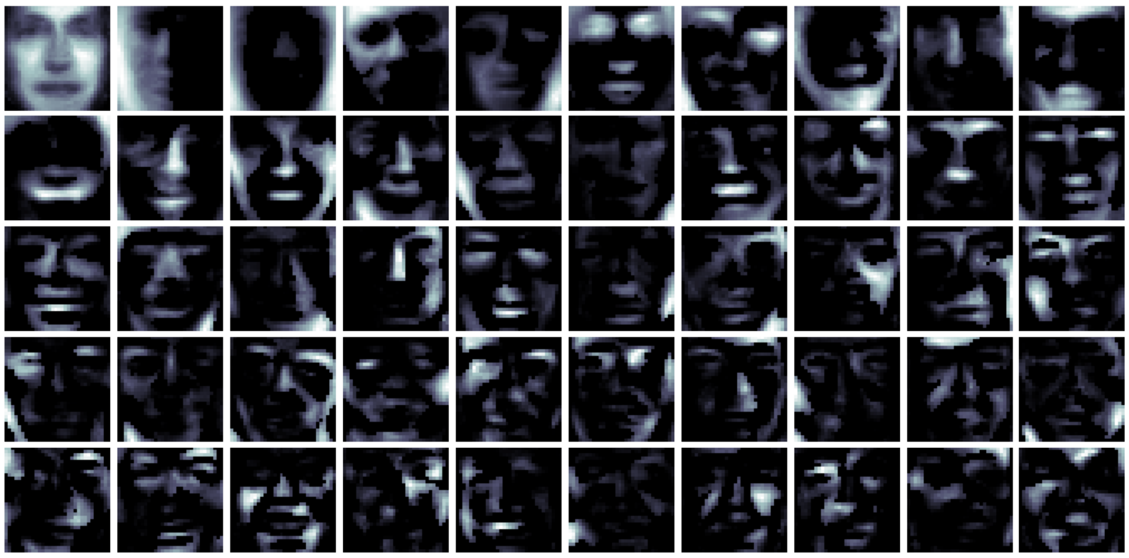

Figure 16: 50 MFs

Here, we see that the MFs are highly selective to separate face areas, much more than the ICA. After this observation, we continue to reconstruct the images and show them separately.

```
faces_nmf10 = nmf10.fit_transform(faces_nn.T).dot(nmf10.
                                                 components_) - np.abs(np.min(
                                                 faces))
faces_nmf25 = nmf25.fit_transform(faces_nn.T).dot(nmf25.
                                                 components_) - np.abs(np.min(
                                                 faces))
faces_nmf50 = nmf50.fit_transform(faces_nn.T).dot(nmf50.
                                                 components_) - np.abs(np.min(
                                                 faces))
plt.figure(figsize=(9,9))
for i in range(1,37):
    plt.subplot(6,6,i)
    plt.imshow(faces_nmf10[i-1].reshape((32,32)).T, cmap='bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()
plt.figure(figsize=(9,9))
for i in range(1,37):
    plt.subplot(6,6,i)
    plt.imshow(faces_nmf25[i-1].reshape((32,32)).T, cmap='bone')
    plt.axis('off')
plt.show()
plt.figure(figsize=(9,9))
for i in range(1,37):
    plt.subplot(6,6,i)
    plt.imshow(faces_nmf50[i-1].reshape((32,32)).T, cmap='bone')
    plt.axis('off')
plt.show()
```

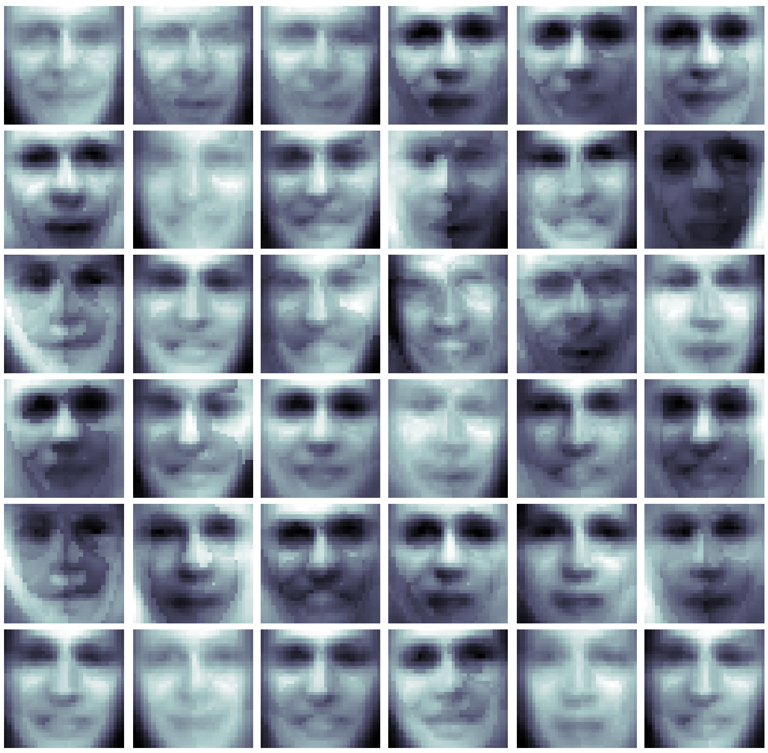
Figure 17: Reconstructed Images (10 MFs)

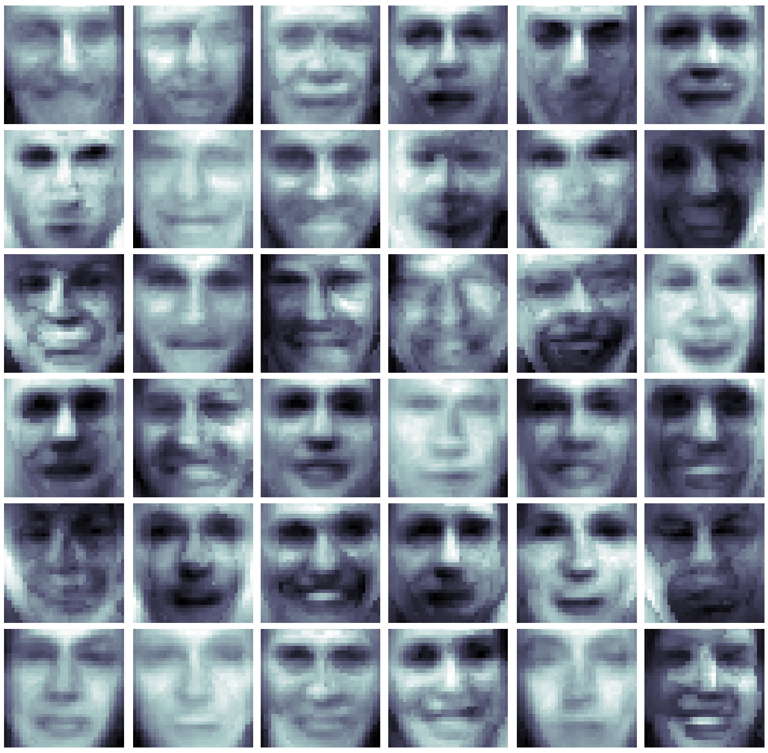
Figure 18: Reconstructed Images (25 MFs)

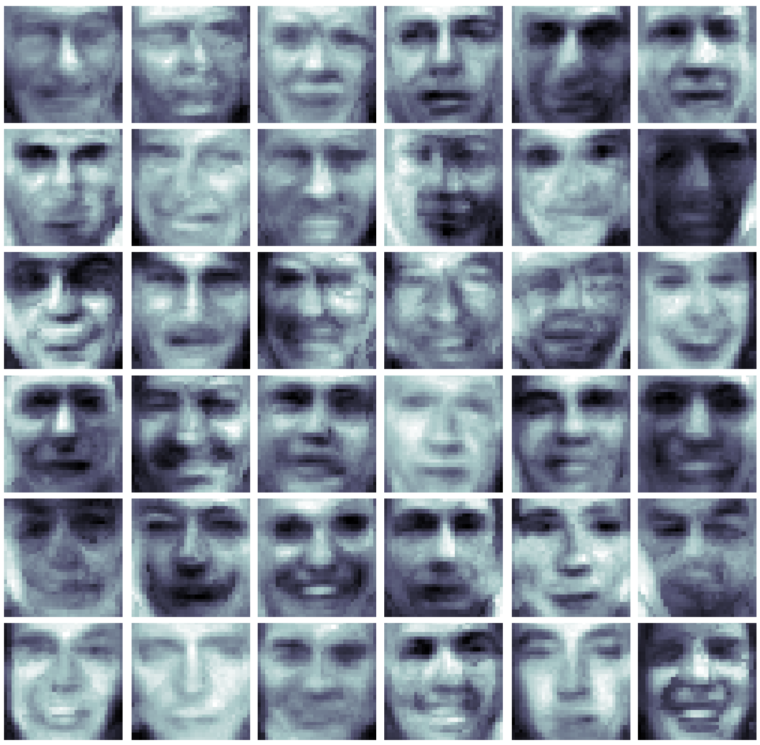

Figure 19: Reconstructed Images (50 MFs)

Then we compute the reconstruction losses.

```
mse_10 = np.mean((faces.T- faces_nmf10)**2)
std_10 = np.std(np.mean((faces.T - faces_nmf10)**2, axis=1))
print('Reconstruction MSE Loss with %d MFs:\nMean: %f\nStd: %f ' %
                                    (10, mse_10, std_10))
mse_25 = np.mean((faces.T- faces_nmf25)**2)
std_25 = np.std(np.mean((faces.T - faces_nmf25)**2, axis=1))
print('Reconstruction MSE Loss with %d MFs:\nMean: %f\nStd: %f ' %
                                    (25, mse_25, std_25))
mse_50 = np.mean((faces.T- faces_nmf50)**2)
std_50 = np.std(np.mean((faces.T - faces_nmf50)**2, axis=1))
print('Reconstruction MSE Loss with %d MFs:\nMean: %f\nStd: %f ' %
                                    (50, mse_50, std_50))
```

```
Reconstruction MSE Loss with 10 MFs:
Mean: 711.188410
Std: 373.313254
Reconstruction MSE Loss with 25 MFs:
Mean: 547.367230
Std: 275.441519
Reconstruction MSE Loss with 50 MFs:
Mean: 416.027792
Std: 200.738901
```

From the output, we observe that the errors are worse than PCA. However, we see that the outputs are highly interpretable since the face images look **divided** into face parts, which we believe the MFs correspond to. Overall, between these three methods PCA performed the best,which is easy to implement and compute a deterministic and statistically-sound method.

Question 2

In this question, we are given a neural population with the response patterns as Gaussian distributions. The popuulation consists of 21 neurons with tuning curves shaped by the following equation

$$f_i(x) = Ae^{\frac{-(x-\mu_i)^2}{2\sigma_i^2}} \tag{6}$$

It is given that the curves have an amplitude (A) and standard deviation (σ_i) of 1. The means of the Gaussians are spaced evenly between -10 and 10, meaning the each adjacent curve has an absolute mean difference of 1. Furthermore, the question requires us to implement both probabilistic and non-probabilistic population decoding approaches being WTA (Winner-Take-All), MLE (Maximum Likelihood Estimation) and MAP(Maximum A Posteriori Estimation). The packages used in this question are as follows.

```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm #to keep execution time since some take longer
                                    than expected
```

Part A This part asks us to plot all of the tuning curves on the same graph. Hence, we create a `tuning_curve` function which takes the Gaussian parameters and returns the specific value. Then we use `np.vectorize` to vectorize the function since, we would like to enter a range of stimuli or mean vector as input and parallelizing the computation is a logical approach.

```
def tuning_curve(x, A, mean, std):
    return A * np.exp(-(x-mean)**2/(2*std**2))
tuning_curve = np.vectorize(tuning_curve)
```

Then, we plot the tuning curves as follows.

```
x_range = np.arange(-15,15,0.01)
mean_vector = np.arange(-10,11,1)
STD = 1
A = 1
for mean in mean_vector:
    plt.plot(x_range,tuning_curve(x_range,A,mean,STD))
plt.title('Tuning Curves of A Population of Neurons')
plt.xlabel('Stimulus')
plt.ylabel('Neuron Activity')
plt.show()
```

Furthermore, this part asks us to plot the response received by the population to the stimulus $x = -1$. Hence, we again, use the defined function to find the result.

```
stim = -1
plt.plot(mean_vector, tuning_curve(stim,A,mean_vector,STD))
plt.title('Population Response to Stimulus $x=-1$\nvs. Preferred
                                        Stimuli')
plt.xlabel('Reffered Stimulus')
plt.ylabel('Response')
plt.show()
```

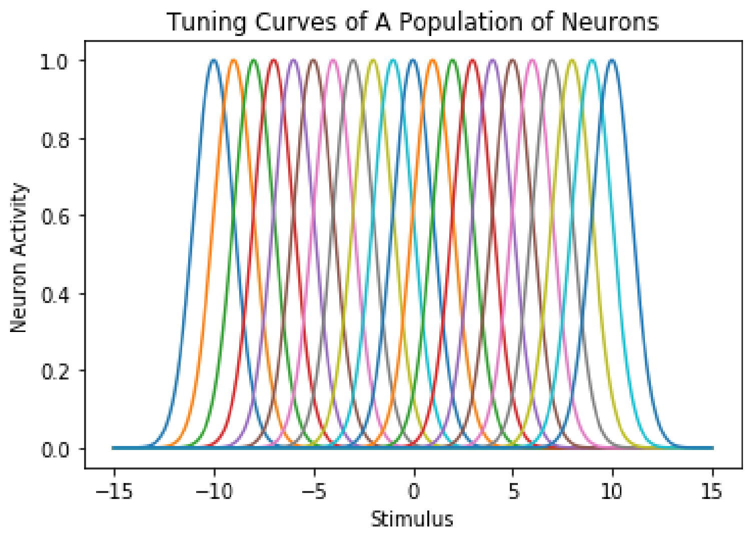

Figure 20: Tuning Curves of the Population Neurons

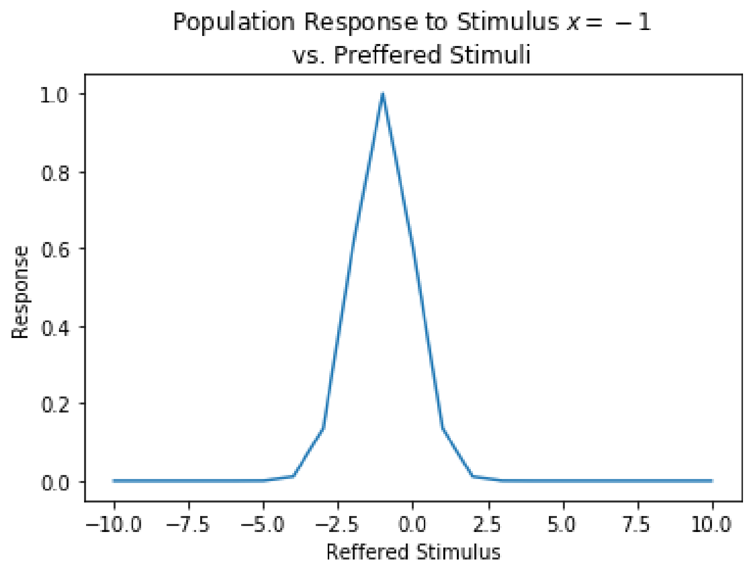

Figure 21: Population Response to Stimulus $x = -1$

Here, we observe that the neurons that have a preferred stimulus value closer to the input stimulus elicit higher response.

Part B This part asks us to create an experiment with 200 trials, in which, in each trial, we sample an input stimulus in the range $[-5, 5]$ and simulate the population response. For simplicity, we have chosen a resolution of 0.01 in the given interval. The response is modeled as the generated response combined with an additive Gaussian noise with zero mean and $\sigma/20$ standard deviation, which becomes $1/20$. Hence we can represent the noise model as,

$$\epsilon_i \sim \mathcal{N}(0, 1/20) \tag{1}$$

Then, we are asked to implement a Winner-Take-All (WTA) decoder, which estiamtes the input stimulus as the preferred stimulus of the neuron in the population the gave the highest response. In mathematical terms,

$$\hat{x} = \mu_i \, such \, that \, i = arg \max_i (r_i) \tag{2}$$

where r_i represents the response of the i^{th} neuron. This is implemented as follows.

```
def wta_decoder(pref_stim, resp):
    highest = np.argmax(resp)
    return pref_stim[highest]
```

Then, we construct the experiment, and calculate/save the absolute estimation error for each trial, record the actual and estimated stimuli, and plot them. Furthermore, we calculate the mean and standard deviation of the error vector.

```
NUM_TRIAL = 200
stim_interval = np.arange(-5,5.01,0.01)

stim_vector = []
est_stim_vector = []
resp_vector = []
error_vector = []

for trial in range(NUM_TRIAL):
    trial_stim = np.random.choice(stim_interval)
    trial_resp = tuning_curve(trial_stim, A, mean_vector, STD)
    noise = np.random.normal(0,STD/20, len(mean_vector))
    trial_resp += noise

    estimated_stim = wta_decoder(mean_vector, trial_resp)
    trial_error = np.abs(trial_stim-estimated_stim)

    stim_vector.append(trial_stim)
    est_stim_vector.append(estimated_stim)
    error_vector.append(trial_error)
    resp_vector.append(trial_resp)

plt.figure(figsize=(12,5))
plt.scatter(np.arange(NUM_TRIAL), stim_vector, c='r',s=10)
plt.scatter(np.arange(NUM_TRIAL), est_stim_vector, c='c', s=10)
plt.legend(['stimulus', 'estimate'])
plt.yticks(np.arange(-5,6,1))
plt.grid(axis='y')
plt.xlabel('Trial')
plt.ylabel('Stimulus')
plt.title('Actual and Winner-Take-All Estimated Stimuli Across 200
                                     Trials')
plt.show()
wta_error_mean = np.mean(error_vector)
wta_error_std = np.std(error_vector)
print('Mean Error in WTA Estimation: %f \nError standard deviation
                                     in WTA Estimation: %f' % (
                                    wta_error_mean,wta_error_std))
```

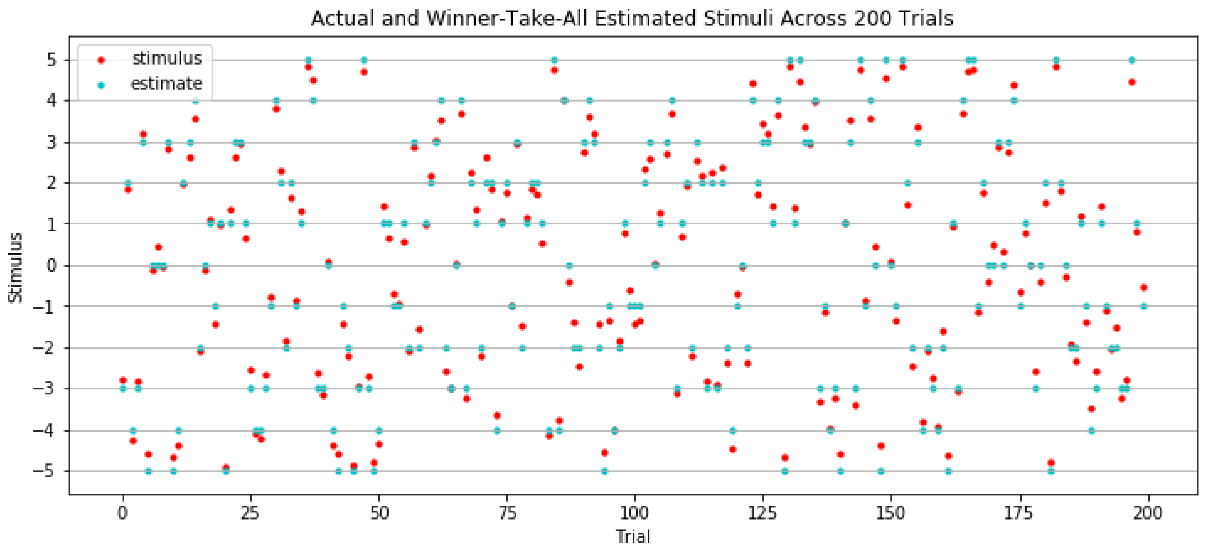

Figure 22: Real and WTA Estimated Stimuli for 200 Trials

```
Mean Error in WTA Estimation: 0.272000
Error standard deviation in WTA Estimation: 0.154081
```

The mean absolute error is relatively high at this decoder option, however this is an expected result since the WTA decoder is limited to the population means when estimating, which is a small set.

Part C In this part, we are required to implement a maximum likelihood estimate decoder, which considers the likelihood of each possible stimulus and selects the best. For implementation, we have simplified that likelihood estimation process. The prediction in this option is given by the following formula.

$$\hat{x} = arg\max_x(L(x)) = arg\max_x(P(\mathbf{r}|x)) \tag{3}$$

where $\mathbf{r}$ is the response vector for all 21 neurons for the given stimulus. We continue with the derivation of $L(x)$. Given $\mathbf{r} = [r_1, ..., r_{21}]^T$,

$$r_i(x) = f_i(x) + \mathcal{N}(0, \sigma/20)\ \forall i = 1, ..., 21 \tag{4}$$

Then,

$$r_i(x) \sim \mathcal{N}(f_i(x), \sigma/20) \tag{5}$$

$$f_{r_i(x)} = \frac{1}{\sqrt{2\pi}(\sigma/20)} e^{\frac{-(r_i(x)-f_i(x))^2}{2(\sigma/20)^2}} \tag{6}$$

Since, the neurons are assumed to be independent from each other, we can deifne the likelihood function as the multiplication of individual probabilities of neurons.

$$L(x) = P(\mathbf{r}|x) = \prod_{i=1}^{21} f_{r_i}(x) \tag{7}$$

Hence, we can reiterate the argmax function as follows.

$$\hat{x} = arg\max_x(L(x)) = arg\max_x log(L(x)) = arg\max_x l(x) \tag{8}$$

$$= arg\max_x log(\prod_{i=1}^{21} f_{r_i}(x)) = arg\max_x \sum_{i=1}^{21} log(f_{r_i}(x)) \tag{9}$$

$$= arg\max_x \sum_{i=1}^{21} log\left(\frac{1}{\sqrt{2\pi}(\sigma/20)} e^{\frac{-(r_i(x)-f_i(x))^2}{2(\sigma/20)^2}}\right) \tag{10}$$

$$= arg\max_x \sum_{i=1}^{21} \frac{-(r_i(x)-f_i(x))^2}{2(\sigma/20)^2} - log(\sqrt{2\pi}(\sigma/20)) \tag{11}$$

As the term on the right is a constant and the multiplication doesn't change the output of argmax,

$$= arg\max_x \sum_{i=1}^{21} -(r_i(x)-f_i(x))^2 = arg\min_x \sum_{i=1}^{21} (r_i(x)-f_i(x))^2 \tag{12}$$

which corresponds to the Least Squares Error (SSE), which is normal since the ML estimate of a Gaussian likelihood converts to least squares cost minimization problem. Hence, we implelent the following function to calculate this given as follows.

```python
def ols_error(x, responses, means, A=1, std=1):
    return np.sum((responses - tuning_curve(x,A,means,std))**2)

def MLE_decoder(response, means, A=1, std=1, stim_interval=np.
                                        arange(-5,5.01,0.01)):
    errors = []
    for stim in stim_interval:
        errors.append(ols_error(stim, response,means))
    idx = np.argmin(errors)
    return stim_interval[idx]
```

Then, we take the same stimulus set we have created in the previous part and apply MLE decoding, again plot the actual and estimated stimuli.

```python
est_stim_vector_MLE = []
error_vector_MLE = []

for resp,stim in zip(resp_vector,stim_vector):
    estimated_stim = MLE_decoder(resp, mean_vector)

    est_stim_vector_MLE.append(estimated_stim)
    error_vector_MLE.append(np.abs(stim-estimated_stim))

plt.figure(figsize=(12,5))
plt.scatter(np.arange(NUM_TRIAL), stim_vector, c='r',s=10)
plt.scatter(np.arange(NUM_TRIAL), est_stim_vector_MLE, c='c', s=10
                                        )
```

```
plt.legend(['stimulus', 'estimate'])
plt.yticks(np.arange(-5,6,1))
plt.grid(axis='y')
plt.xlabel('Trial')
plt.ylabel('Stimulus')
plt.title('Actual and MLE Estimated Stimuli Across 200 Trials')
plt.show()

mle_error_mean = np.mean(error_vector_MLE)
mle_error_std = np.std(error_vector_MLE)
print('Mean Error in MLE Estimation: %f \nError standard deviation
                                     in MLE Estimation: %f' % (
                                   mle_error_mean,mle_error_std))
```

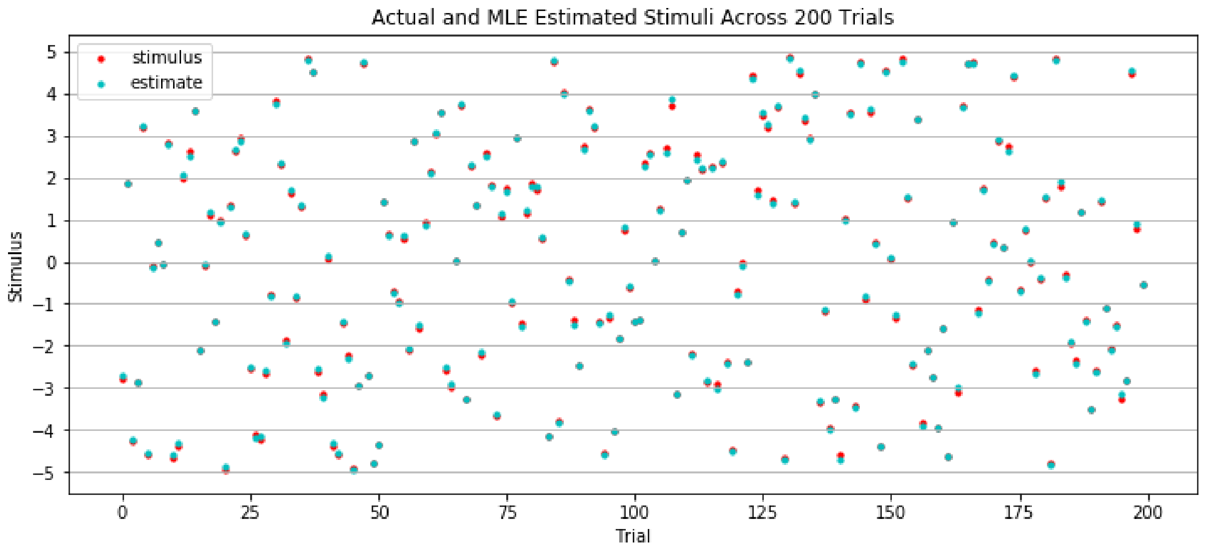

Figure 23: Real and ML Estimated Stimuli for 200 Trials

Then we observe the outputs of the mean and std of mean absolute errors. However, we should observe that the estimations are now closer to the actual stimuli than it was in the WTA estimation.

```
Mean Error in MLE Estimation: 0.040550
Error standard deviation in MLE Estimation: 0.029431
```

There is a significant difference in terms of error between ML and WTA estimates. ML estimates are far superior since the choice set is significantly better since the algorithm computer every possibility of stimulus between −5 and 5. The amount of error is very small which demonstrates the effectiveness of probabilistic decoders.

Part D In this part, similar to what we have done in the previous part, we implement a MAP decoder. This time, we are given a prior distribution of the data and we aim to estimate the posterior. Hence, we can define the objective function as follows, from the Bayes Rule.

$$\hat{x} = arg\max_x(P(x|\mathbf{r})) = arg\max_x(P(\mathbf{r}|x)P(x)) = arg\max_x(L(x)P(x)) \tag{13}$$

$$= arg\max_x log(L(x)) + log(P(x)) \tag{14}$$

We can see that the first part of the sum corresponds to the initial problem of the previous part, the ML estimate. Then we can focus on the second part and then combine the results. WE are given that the prior is also a Gaussian distribution with $\mu = 0$ and $\sigma = 2.5$.

$$log(P(x)) = \frac{1}{\sqrt{2\pi}2.5} e^{\frac{-x^2}{2(2.5)^2}} \tag{15}$$

$$= arg\max_x log(P(x)) = arg\max_x \left(-\frac{x^2}{2(2.5)^2} - log(\frac{1}{\sqrt{2\pi}2.5}) \right) \tag{16}$$

As the term on the right is a constant,

$$= arg\max_x - \frac{x^2}{2(2.5)^2} = arg\min_x \frac{x^2}{2(2.5)^2} \tag{17}$$

Hence, we can combine the previous part and define the MAP estimation as,

$$\hat{x} = arg\min_x \left(\sum_{i=1}^{21} \left(\frac{1}{2(\sigma/20)^2} (r_i(x) - f_i(x))^2 \right) + \frac{x^2}{2(2.5)^2} \right) \tag{18}$$

This results is the same with (12) with the addition of an L2 regularization term that comes from the existence of a prior. Hence, we follow the same procedure as we have in the previous part and implement this result.

```python
def map_error(x, responses, means, A=1, std=1):
    return np.sum((responses - tuning_curve(x,A,means,std))**2)/(2
                                          *(std/20)**2) + x**2/(2*2.
                                          5**2)

def MAP_decoder(response, means, A=1, std=1, stim_interval=np.
                                     arange(-5,5.01,0.01)):
    errors = []
    for stim in stim_interval:
        errors.append(map_error(stim, response, means))
    idx = np.argmin(errors)
    return stim_interval[idx]
```

We plot the estimated and actual stimuli, and run the error calculations.

```
est_stim_vector_MAP = []
error_vector_MAP = []

for resp,stim in zip(resp_vector,stim_vector):
    estimated_stim = MAP_decoder(resp, mean_vector)

    est_stim_vector_MAP.append(estimated_stim)
    error_vector_MAP.append(np.abs(stim-estimated_stim))

plt.figure(figsize=(12,5))
plt.scatter(np.arange(NUM_TRIAL), stim_vector, c='r',s=10)
plt.scatter(np.arange(NUM_TRIAL), est_stim_vector_MAP, c='c', s=10
                                    )
plt.legend(['stimulus', 'estimate'])
plt.yticks(np.arange(-5,6,1))
plt.grid(axis='y')
plt.xlabel('Trial')
plt.ylabel('Stimulus')
plt.title('Actual and MAP Estimated Stimuli Across 200 Trials')
plt.show()

map_error_mean = np.mean(error_vector_MAP)
map_error_std = np.std(error_vector_MAP)
print('Mean Error in MAP Estimation: %f \nError standard deviation
                                     in MAP Estimation: %f' % (
                                    map_error_mean,map_error_std))
```

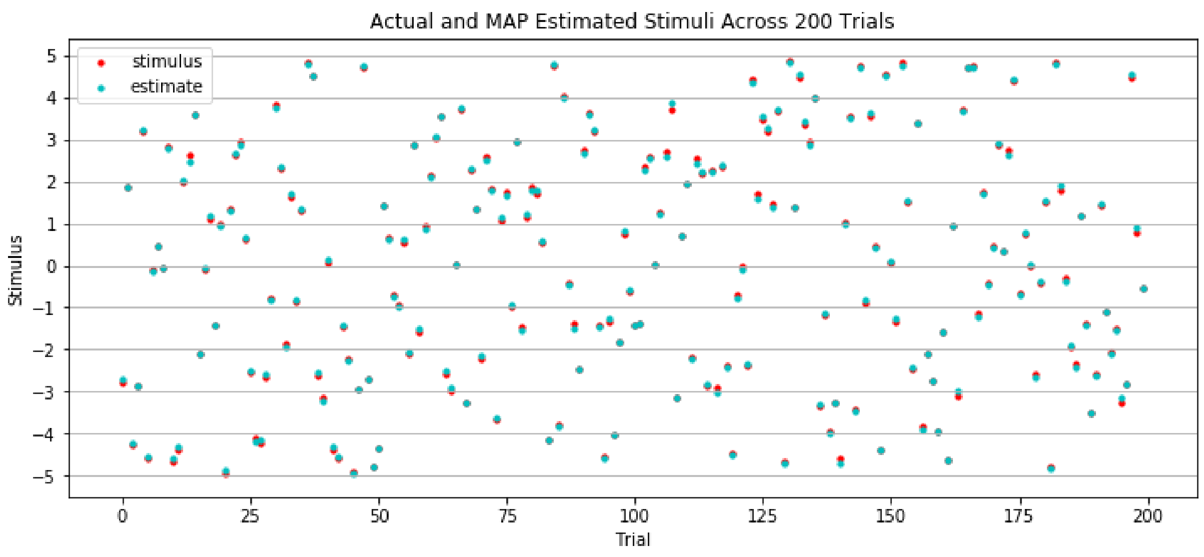

Figure 24: Real and MAP Estimated Stimuli for 200 Trials

```
Mean Error in MAP Estimation: 0.040650
Error standard deviation in MAP Estimation: 0.029582
```

Here, we observe that both the mean and standard deviation of error is smaller than the ML estimate, which shows that MAP estimation performed better than the ML estimate, but not by a large margin. This results was expected since MAP estimation contains additional knowledge about the data in the form of a prior distribution, which ultimately should lead to better performance. As a result, probabilistic estimations were better than the non-probabilistic estimations.

Part E The last part is different from the previous parts in the sense that here, we investigate the effect of the standard deviation value of the tuning curves on the performance of estimation. However, we still conduct the same experiment that we repeat for 200 times. For a trial, we have created a stimulus and a response, however this time, we have estimated the ML estimates from the response for six different values of standard deviation for the tuning curves. In this section, we use the **tqdm** package to make the progress visible in execution, since the execution takes around $\sim 2min$. The code is as follows.

```
std_vector = [0.1, 0.2, 0.5, 1, 2, 5]
error_all_trials = []
for trial in tqdm(range(NUM_TRIAL)):
    trial_stim = np.random.choice(stim_interval)
    error_single_trial = []
    for std in std_vector:
        trial_resp = tuning_curve(trial_stim, A, mean_vector, std)
        noise = np.random.normal(0,1/20, len(mean_vector))
        trial_resp += noise

        estimated_stim = MLE_decoder(trial_resp, mean_vector)
        error_single_trial.append(np.abs(estimated_stim -
                                          trial_stim))
    error_all_trials.append(error_single_trial)
error_all_trials = np.asarray(error_all_trials)
print(error_all_trials.shape)
```

Then, we plot all individual absolute error vs trial graphs, with the marking of the mean of error also the ±1 standard deviation range.

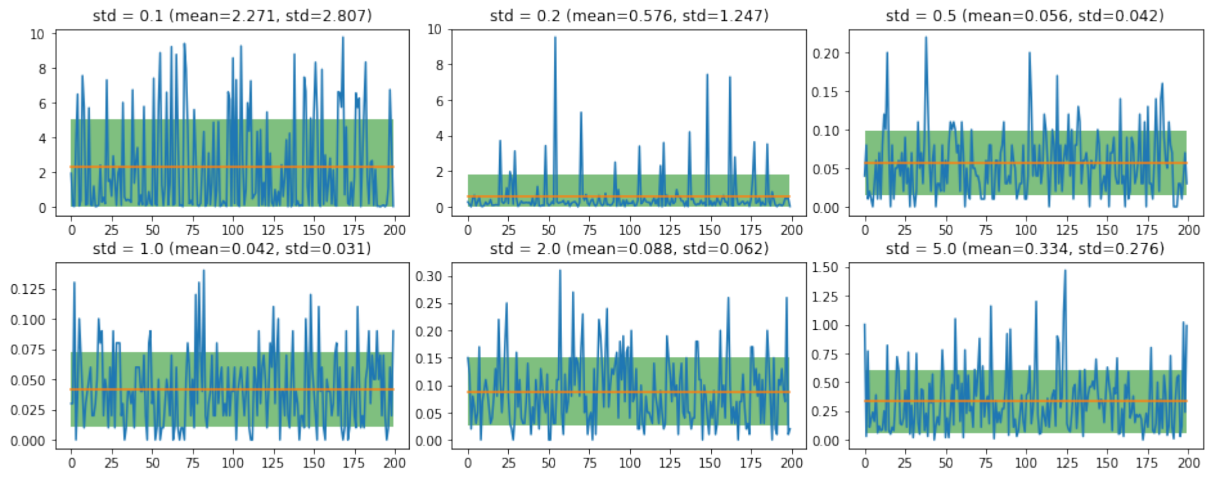

Figure 25: Absolute Errors vs. Trials (200 Trials, MLE)

By looking at Figure 25, we observe that both the error and standard deviation is higher than expected and the mean gets smaller until $\sigma = 1.0$ and then increases, which gives the idea that there is an optimal σ value that minimized the error mean. Also, we see that the, again until $\sigma = 1.0$ the standard deviation decreases, then increases. Hence from this preview of the error plots, we think that there is an optimal value of σ that would give the smallest mean and std of absolute error. To confirm this, we plot the Mean Absolute Error vs Standard Deviation curve.

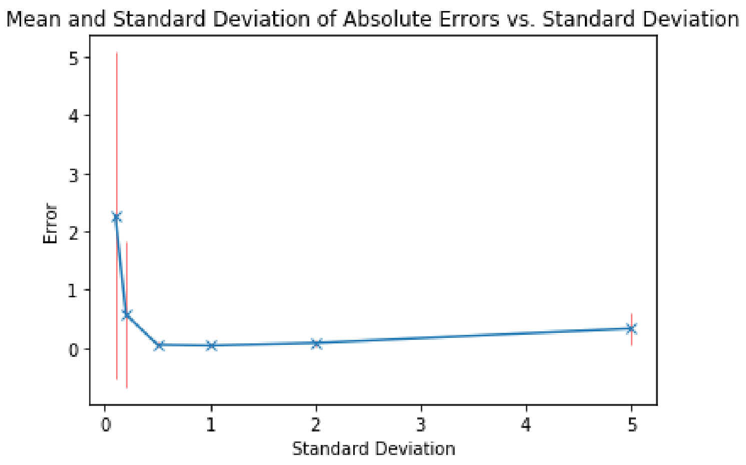

Figure 26: Mean Absolute Errors (and Standard Deviations) vs. Tuning Curve Standard Deviations

By looking at Figure 26, we can say that the premonition presented is valid, meaning there is an optimal point of σ which looks like a value between 0.5 and 1.0 in this case. Furthermore, as a more general approach, we can say that as σ increases, error standard deviation decreases with the MAE (Mean Absolute Error). Selecting σ too far from the optimal can reduce performance, but wider curves generally perform better then steep tuning curves.

Python Code

```
import sys
import numpy as np
import matplotlib.pyplot as plt
import h5py
import sklearn.decomposition as decomp
from tqdm import tqdm

question = sys.argv[1]

def ayhan_okuyan_21601531_hw4(question):
    if question == '1' :
        print('Question 1')
        with h5py.File('hw4_data1.mat', 'r') as file:
            keys = list(file.keys())
            print(keys)
            faces = np.asarray(file['faces'])
        print(faces.shape)

        print('Part A')

        plt.imshow(faces[:, 0].reshape((32, 32)).T, cmap='bone')
        plt.axis('off')
        plt.title('Sample Stimuli')
        plt.show()

        pca_data = decomp.PCA(100, whiten=True)
        pca_data.fit(faces.T)

        plt.plot(pca_data.explained_variance_ratio_)
        plt.xlabel('Principal Components')
        plt.ylabel('Proportion of Variance Explained')
        plt.title('Proportion of Explained Variance\n for Each
                                           Individual Principle
                                           Component')
        plt.grid()
        plt.show()

        print('First 25 Principal Components')
        plt.figure(figsize=(9, 9))
        for i in range(1, 26):
            plt.subplot(5, 5, i)
            plt.imshow(pca_data.components_[i - 1].reshape((32, 32)).T,
                                                cmap='bone')
            plt.axis('off')
        plt.subplots_adjust(wspace=0.05, hspace=0.05)
        plt.show()

        print('Part B')
        faces_rec10 = (faces - pca_data.mean_.reshape(1024, 1)).T.dot(
                                           pca_data.components_[0:10].T
                                           ).dot(
            pca_data.components_[0:10]) + pca_data.mean_
        faces_rec25 = (faces - pca_data.mean_.reshape(1024, 1)).T.dot(
                                           pca_data.components_[0:25].T
                                           ).dot(
            pca_data.components_[0:25]) + pca_data.mean_
        faces_rec50 = (faces - pca_data.mean_.reshape(1024, 1)).T.dot(
                                           pca_data.components_[0:50].T
                                           ).dot(
            pca_data.components_[0:50]) + pca_data.mean_
        print('Original Images')
        plt.figure(figsize=(9, 9))
        for i in range(1, 37):
            plt.subplot(6, 6, i)
            plt.imshow(faces.T[i - 1].reshape((32, 32)).T, cmap='bone')
            plt.axis('off')
        plt.subplots_adjust(wspace=0.05, hspace=0.05)
        plt.show()
        print('Reconstructed Images with 10 PCs')
        plt.figure(figsize=(9, 9))
        for i in range(1, 37):
            plt.subplot(6, 6, i)
            plt.imshow(faces_rec10[i - 1].reshape((32, 32)).T, cmap='
                                                bone')
            plt.axis('off')
        plt.subplots_adjust(wspace=0.05, hspace=0.05)
        plt.show()
        print('Reconstructed Images with 25 PCs')
        plt.figure(figsize=(9, 9))
        for i in range(1, 37):
            plt.subplot(6, 6, i)
            plt.imshow(faces_rec25[i - 1].reshape((32, 32)).T, cmap='
                                                bone')
            plt.axis('off')
        plt.subplots_adjust(wspace=0.05, hspace=0.05)
        plt.show()
        print('Reconstructed Images with 50 PCs')
        plt.figure(figsize=(9, 9))
        for i in range(1, 37):
            plt.subplot(6, 6, i)
            plt.imshow(faces_rec50[i - 1].reshape((32, 32)).T, cmap='
                                                bone')
            plt.axis('off')
        plt.subplots_adjust(wspace=0.05, hspace=0.05)
        plt.show()

        mse_10 = np.mean((faces.T - faces_rec10) ** 2)
        std_10 = np.std(np.mean((faces.T - faces_rec10) ** 2, axis=1))
        print('Reconstruction MSE Loss with %d PCs:\nMean: %f\nStd: %f '
                                           % (10, mse_10, std_10))
        mse_25 = np.mean((faces.T - faces_rec25) ** 2)
        std_25 = np.std(np.mean((faces.T - faces_rec25) ** 2, axis=1))
        print('Reconstruction MSE Loss with %d PCs:\nMean: %f\nStd: %f '
                                           % (25, mse_25, std_25))
        mse_50 = np.mean((faces.T - faces_rec50) ** 2)
        std_50 = np.std(np.mean((faces.T - faces_rec50) ** 2, axis=1))
```

```
        print('Reconstruction MSE Loss with %d PCs:\nMean: %f\nStd: %f '
                                                      % (50, mse_50, std_50))

        print('Part C')

        rng = np.random.seed(10)
        ica10 = decomp.FastICA(10, whiten=True, random_state=rng)
        ica25 = decomp.FastICA(25, whiten=True, random_state=rng)
        ica50 = decomp.FastICA(50, whiten=True, random_state=rng)
        ica10.fit(faces.T)
        ica25.fit(faces.T)
        ica50.fit(faces.T)

        print('10 ICs')
        plt.figure(figsize=(9, 3.5))
        for i in range(1, 11):
            plt.subplot(2, 5, i)
            plt.imshow(ica10.components_[i - 1].reshape((32, 32)).T,
                                                   cmap='bone')
            plt.axis('off')
        plt.subplots_adjust(wspace=0.05, hspace=0.05)
        plt.show()
        print('25 ICs')
        plt.figure(figsize=(9, 9))
        for i in range(1, 26):
            plt.subplot(5, 5, i)
            plt.imshow(ica25.components_[i - 1].reshape((32, 32)).T,
                                                   cmap='bone')
            plt.axis('off')
        plt.subplots_adjust(wspace=0.05, hspace=0.05)
        plt.show()
        print('50 ICs')
        plt.figure(figsize=(18, 9))
        for i in range(1, 51):
            plt.subplot(5, 10, i)
            plt.imshow(ica50.components_[i - 1].reshape((32, 32)).T,
                                                   cmap='bone')
            plt.axis('off')
        plt.subplots_adjust(wspace=0.05, hspace=0.05)
        plt.show()

        faces_ica10 = ica10.fit_transform(faces.T).dot(ica10.mixing_.T)
                                                    + ica10.mean_
        faces_ica25 = ica25.fit_transform(faces.T).dot(ica25.mixing_.T)
                                                    + ica25.mean_
        faces_ica50 = ica50.fit_transform(faces.T).dot(ica50.mixing_.T)
                                                    + ica50.mean_

        print('Reconstructed Images with 10 ICs')
        plt.figure(figsize=(9, 9))
        for i in range(1, 37):
            plt.subplot(6, 6, i)
            plt.imshow(faces_ica10[i - 1].reshape((32, 32)).T, cmap='
                                                   bone')
            plt.axis('off')
```

```
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()
print('Reconstructed Images with 25 ICs')
plt.figure(figsize=(9, 9))
for i in range(1, 37):
    plt.subplot(6, 6, i)
    plt.imshow(faces_ica25[i - 1].reshape((32, 32)).T, cmap='
                                    bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()
print('Reconstructed Images with 50 ICs')
plt.figure(figsize=(9, 9))
for i in range(1, 37):
    plt.subplot(6, 6, i)
    plt.imshow(faces_ica50[i - 1].reshape((32, 32)).T, cmap='
                                    bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()

mse_10 = np.mean((faces.T - faces_ica10) ** 2)
std_10 = np.std(np.mean((faces.T - faces_ica10) ** 2, axis=1))
print('Reconstruction MSE Loss with %d ICs:\nMean: %f\nStd: %f '
                                 % (10, mse_10, std_10))
mse_25 = np.mean((faces.T - faces_ica25) ** 2)
std_25 = np.std(np.mean((faces.T - faces_ica25) ** 2, axis=1))
print('Reconstruction MSE Loss with %d ICs:\nMean: %f\nStd: %f '
                                 % (25, mse_25, std_25))
mse_50 = np.mean((faces.T - faces_ica50) ** 2)
std_50 = np.std(np.mean((faces.T - faces_ica50) ** 2, axis=1))
print('Reconstruction MSE Loss with %d ICs:\nMean: %f\nStd: %f '
                                 % (50, mse_50, std_50))

print('Part D')
faces_nn = faces + np.abs(np.min(faces))
nmf10 = decomp.NMF(n_components=10, solver='mu', max_iter=1000)
nmf25 = decomp.NMF(n_components=25, solver='mu', max_iter=1000)
nmf50 = decomp.NMF(n_components=50, solver='mu', max_iter=1000)
nmf10.fit(faces_nn.T)
nmf25.fit(faces_nn.T)
nmf50.fit(faces_nn.T)

print('10 MFs')
plt.figure(figsize=(9, 3.5))
for i in range(1, 11):
    plt.subplot(2, 5, i)
    plt.imshow(nmf10.components_[i - 1].reshape((32, 32)).T,
                                    cmap='bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()
print('25 MFs')
plt.figure(figsize=(9, 9))
for i in range(1, 26):
    plt.subplot(5, 5, i)
    plt.imshow(nmf25.components_[i - 1].reshape((32, 32)).T,
                                                cmap='bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()
print('50 MFs')
plt.figure(figsize=(18, 9))
for i in range(1, 51):
    plt.subplot(5, 10, i)
    plt.imshow(nmf50.components_[i - 1].reshape((32, 32)).T,
                                                cmap='bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()

faces_nmf10 = nmf10.fit_transform(faces_nn.T).dot(nmf10.
                                        components_) - np.abs(np.min
                                        (faces))
faces_nmf25 = nmf25.fit_transform(faces_nn.T).dot(nmf25.
                                        components_) - np.abs(np.min
                                        (faces))
faces_nmf50 = nmf50.fit_transform(faces_nn.T).dot(nmf50.
                                        components_) - np.abs(np.min
                                        (faces))

print('Reconstructed Images with 10 MFs')
plt.figure(figsize=(9, 9))
for i in range(1, 37):
    plt.subplot(6, 6, i)
    plt.imshow(faces_nmf10[i - 1].reshape((32, 32)).T, cmap='
                                                bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()
print('Reconstructed Images with 25 MFs')
plt.figure(figsize=(9, 9))
for i in range(1, 37):
    plt.subplot(6, 6, i)
    plt.imshow(faces_nmf25[i - 1].reshape((32, 32)).T, cmap='
                                                bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()
print('Reconstructed Images with 50 MFs')
plt.figure(figsize=(9, 9))
for i in range(1, 37):
    plt.subplot(6, 6, i)
    plt.imshow(faces_nmf50[i - 1].reshape((32, 32)).T, cmap='
                                                bone')
    plt.axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()

mse_10 = np.mean((faces.T - faces_nmf10) ** 2)
```

```
        std_10 = np.std(np.mean((faces.T - faces_nmf10) ** 2, axis=1))
        print('Reconstruction MSE Loss with %d MFs:\nMean: %f\nStd: %f '
                                                % (10, mse_10, std_10))
        mse_25 = np.mean((faces.T - faces_nmf25) ** 2)
        std_25 = np.std(np.mean((faces.T - faces_nmf25) ** 2, axis=1))
        print('Reconstruction MSE Loss with %d MFs:\nMean: %f\nStd: %f '
                                                % (25, mse_25, std_25))
        mse_50 = np.mean((faces.T - faces_nmf50) ** 2)
        std_50 = np.std(np.mean((faces.T - faces_nmf50) ** 2, axis=1))
        print('Reconstruction MSE Loss with %d MFs:\nMean: %f\nStd: %f '
                                                % (50, mse_50, std_50))

    elif question == '2' :
        print('Question 2')



        print('Part A')
        x_range = np.arange(-15, 15, 0.01)
        mean_vector = np.arange(-10, 11, 1)
        STD = 1
        A = 1
        for mean in mean_vector:
            plt.plot(x_range, tuning_curve(x_range, A, mean, STD))
        plt.title('Tuning Curves of A Population of Neurons')
        plt.xlabel('Stimulus')
        plt.ylabel('Neuron Activity')
        plt.show()

        stim = -1
        plt.plot(mean_vector, tuning_curve(stim, A, mean_vector, STD))
        plt.title('Population Response to Stimulus $x=-1$\nvs. Preffered
                                                Stimuli')
        plt.xlabel('Reffered Stimulus')
        plt.ylabel('Response')
        plt.show()

        print('Part B')
        NUM_TRIAL = 200
        stim_interval = np.arange(-5, 5.01, 0.01)

        stim_vector = []
        est_stim_vector = []
        resp_vector = []
        error_vector = []

        for trial in range(NUM_TRIAL):
            trial_stim = np.random.choice(stim_interval)
            trial_resp = tuning_curve(trial_stim, A, mean_vector, STD)
            noise = np.random.normal(0, STD / 20, len(mean_vector))
            trial_resp += noise

            estimated_stim = wta_decoder(mean_vector, trial_resp)
            trial_error = np.abs(trial_stim - estimated_stim)
            stim_vector.append(trial_stim)
            est_stim_vector.append(estimated_stim)
            error_vector.append(trial_error)
            resp_vector.append(trial_resp)

        plt.figure(figsize=(12, 5))
        plt.scatter(np.arange(NUM_TRIAL), stim_vector, c='r', s=10)
        plt.scatter(np.arange(NUM_TRIAL), est_stim_vector, c='c', s=10)
        plt.legend(['stimulus', 'estimate'])
        plt.yticks(np.arange(-5, 6, 1))
        plt.grid(axis='y')
        plt.xlabel('Trial')
        plt.ylabel('Stimulus')
        plt.title('Actual and Winner-Take-All Estimated Stimuli Across
                                             200 Trials')
        plt.show()

        wta_error_mean = np.mean(error_vector)
        wta_error_std = np.std(error_vector)
        print('Mean Error in WTA Estimation: %f \nError standard
                                             deviation in WTA Estimation:
                                              %f' % (
        wta_error_mean, wta_error_std))

        print('Part C')
        est_stim_vector_MLE = []
        error_vector_MLE = []

        for resp, stim in zip(resp_vector, stim_vector):
            estimated_stim = MLE_decoder(resp, mean_vector)

            est_stim_vector_MLE.append(estimated_stim)
            error_vector_MLE.append(np.abs(stim - estimated_stim))

        plt.figure(figsize=(12, 5))
        plt.scatter(np.arange(NUM_TRIAL), stim_vector, c='r', s=10)
        plt.scatter(np.arange(NUM_TRIAL), est_stim_vector_MLE, c='c', s=
                                             10)
        plt.legend(['stimulus', 'estimate'])
        plt.yticks(np.arange(-5, 6, 1))
        plt.grid(axis='y')
        plt.xlabel('Trial')
        plt.ylabel('Stimulus')
        plt.title('Actual and MLE Estimated Stimuli Across 200 Trials')
        plt.show()

        mle_error_mean = np.mean(error_vector_MLE)
        mle_error_std = np.std(error_vector_MLE)
        print('Mean Error in MLE Estimation: %f \nError standard
                                             deviation in MLE Estimation:
                                              %f' % (
        mle_error_mean, mle_error_std))

        print('Part D')
        est_stim_vector_MAP = []
```

```
error_vector_MAP = []

for resp, stim in zip(resp_vector, stim_vector):
    estimated_stim = MAP_decoder(resp, mean_vector)

    est_stim_vector_MAP.append(estimated_stim)
    error_vector_MAP.append(np.abs(stim - estimated_stim))

plt.figure(figsize=(12, 5))
plt.scatter(np.arange(NUM_TRIAL), stim_vector, c='r', s=10)
plt.scatter(np.arange(NUM_TRIAL), est_stim_vector_MAP, c='c', s=
                                    10)
plt.legend(['stimulus', 'estimate'])
plt.yticks(np.arange(-5, 6, 1))
plt.grid(axis='y')
plt.xlabel('Trial')
plt.ylabel('Stimulus')
plt.title('Actual and MAP Estimated Stimuli Across 200 Trials')
plt.show()

map_error_mean = np.mean(error_vector_MAP)
map_error_std = np.std(error_vector_MAP)
print('Mean Error in MAP Estimation: %f \nError standard
                                    deviation in MAP Estimation:
                                     %f' % (
map_error_mean, map_error_std))

print('Part E')
std_vector = [0.1, 0.2, 0.5, 1, 2, 5]
error_all_trials = []
for trial in tqdm(range(NUM_TRIAL)):
    trial_stim = np.random.choice(stim_interval)
    error_single_trial = []
    for std in std_vector:
        trial_resp = tuning_curve(trial_stim, A, mean_vector,
                                            std)
        noise = np.random.normal(0, 1 / 20, len(mean_vector))
        trial_resp += noise

        estimated_stim = MLE_decoder(trial_resp, mean_vector)
        error_single_trial.append(np.abs(estimated_stim -
                                            trial_stim))
    error_all_trials.append(error_single_trial)
error_all_trials = np.asarray(error_all_trials)
print(error_all_trials.shape)

plt.figure(figsize=(16, 6))
for i in range(1, 7):
    plt.subplot(2, 3, i)
    error_vals = error_all_trials[:, i - 1]
    plt.plot(np.arange(NUM_TRIAL), error_vals)
    mean = np.mean(error_vals)
    std = np.std(error_vals)
    plt.plot([mean] * NUM_TRIAL)
```

```
            plt.fill_between(np.arange(NUM_TRIAL), np.max([mean - std, 0
                                                    ]), mean + std, alpha=0.
                                                    5, facecolor='green')
            plt.title('std = %0.1f (mean=%0.3f, std=%0.3f)' % (
                                                    std_vector[i - 1], mean,
                                                     std))
        plt.subplots_adjust(wspace=0.12, hspace=0.25)
        plt.show()

        means = np.mean(error_all_trials, axis=0)
        stds = np.std(error_all_trials, axis=0)
        plt.errorbar(std_vector, means, yerr=stds, marker='x',
                                                markerfacecolor='r', ecolor=
                                                'r', elinewidth=0.5)
        plt.title('Mean and Standard Deviation of Absolute Errors vs.
                                                Standard Deviation')
        plt.xlabel('Standard Deviation')
        plt.ylabel('Error')
        plt.show()


#Question 2 Functions
def tuning_curve(x, A, mean, std):
      return A * np.exp(-(x - mean) ** 2 / (2 * std ** 2))
tuning_curve = np.vectorize(tuning_curve)

def wta_decoder(pref_stim, resp):
    highest = np.argmax(resp)
    return pref_stim[highest]

def ols_error(x, responses, means, A=1, std=1):
    return np.sum((responses - tuning_curve(x,A,means,std))**2)

def MLE_decoder(response, means, A=1, std=1, stim_interval=np.arange(-5,
                                    5.01,0.01)):
    errors = []
    for stim in stim_interval:
        errors.append(ols_error(stim, response,means))
    idx = np.argmin(errors)
    return stim_interval[idx]

def map_error(x, responses, means, A=1, std=1):
    return np.sum((responses - tuning_curve(x,A,means,std))**2)/(2*(std/
                                        20)**2) + x**2/(2*2.5**2)

def MAP_decoder(response, means, A=1, std=1, stim_interval=np.arange(-5,
                                    5.01,0.01)):
    errors = []
    for stim in stim_interval:
        errors.append(map_error(stim, response, means))
    idx = np.argmin(errors)
    return stim_interval[idx]

ayhan_okuyan_21601531_hw4(question)
```