EEE482 - Computational Neuroscience

Homework Assignment - 4

Berkan Ozdamar

21602353

Question 1

Part a

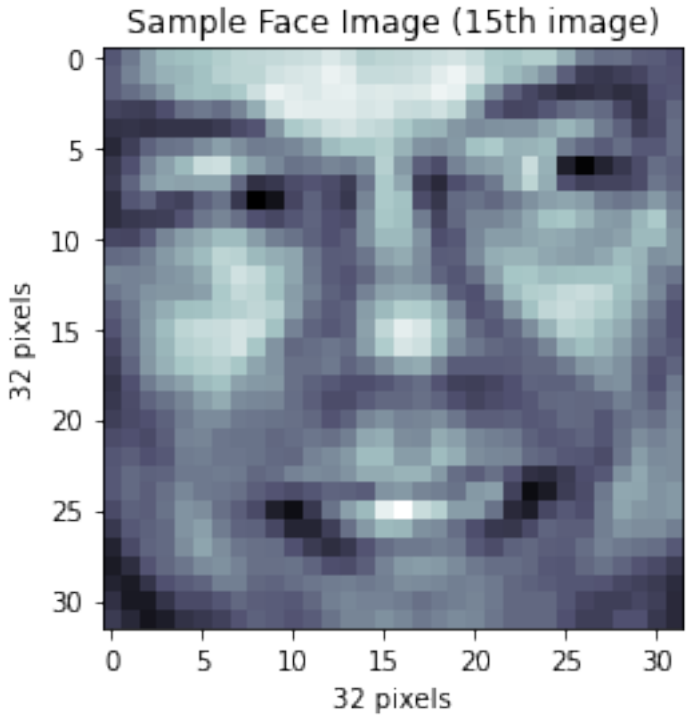
We are given 1000 images with size downsampled to 32x32 square grid. The aim of the question is to use different dimension reduction techniques to express the same image with less dimensions. Used libraries and a sample image is given below:

[1]:
```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
import hdf5storage
import h5py
# For Part C and D
from sklearn.decomposition import FastICA
from sklearn.decomposition import NMF
```

[2]:
```
with h5py.File('hw4_data1.mat', 'r') as file:
    faces = list(file['faces'])


faces = np.array(faces)
#faces = faces.T
print(np.shape(faces))
```

```
(1024, 1000)
```

[3]:
```
# Part A

# A sample stimuli
figure_num = 0
plt.figure(figure_num)
plt.title('Sample Face Image (15th image)')
plt.xlabel('32 pixels')
plt.ylabel('32 pixels')
plt.imshow(faces[:,15].reshape(32, 32).T, cmap=plt.cm.bone)

plt.show(block=False)
```



First dimensionality reduction technique we will implement is the Principal Component Analysis (PCA). To be able to implement PCA, firstly eigenvalue decompositon will be applied to covariance matrix of given data. Then eigenvalues will be sorted from highest to lowest(descending order) and eigenvectors will be sorted correspondingly. The reason behind this sorting is that, highest eigenvalue and corresponding eigenvector means the largest variance in the data. Then k of those eigenpairs will be chosen to respresent the same image with less dimensions but still keep the variance structure. Implementation of PCA in python is given below:

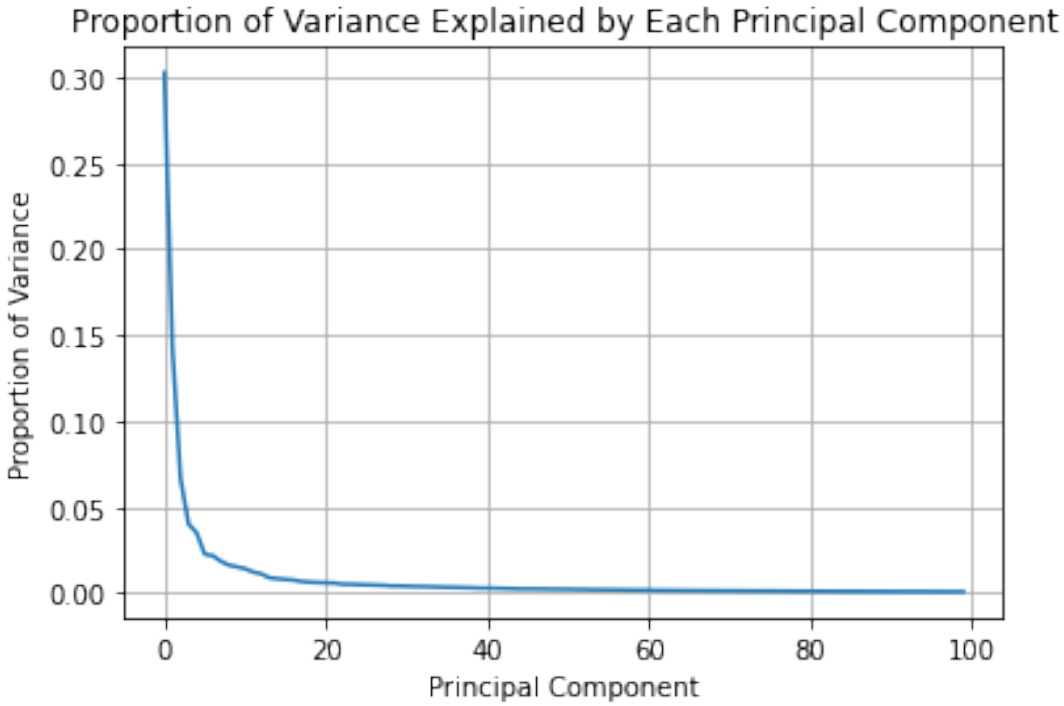
[4]:
```
def PCA(data, numberOfPC):
    data = data.T
    data = data - np.mean(data, axis=0)
    covarianceMatrix = np.dot(data.T, data)
    eigenvalues, eigenvectors = np.linalg.eig(covarianceMatrix)
    eigenvectors = eigenvectors.T
    indexs = np.argsort(eigenvalues)[::-1]
    eigenvectors_sorted = eigenvectors[indexs]
    eigenvalues_sorted = eigenvalues[indexs]

    # store first n eigenvectors
    eigenvectors_f = eigenvectors_sorted[0:numberOfPC]

    variance = []
    normalizer = np.sum(eigenvalues)
    for i in range(numberOfPC):
        variance.append(eigenvalues[i] / normalizer)
    result = np.dot(data, eigenvectors_f.T).dot(eigenvectors_f) + np.mean(data,␣
↪axis=0)
    result = np.array(result)
    result = np.real(result)
    eigenvectors_f = np.real(eigenvectors_f)
    return result, variance, eigenvectors_f
```

[5]:
```
pca_face100, var100, PC100 = PCA(faces, 100)
pca_face25, var25, PC25 = PCA(faces, 25)
```

[6]:
```
print(np.shape(pca_face25))
```

```
(1000, 1024)
```

[7]:
```
figure_num += 1
plt.figure(figure_num)
plt.plot(var100)
plt.title('Proportion of Variance Explained by Each Principal Component')
plt.xlabel('Principal Component')
plt.ylabel('Proportion of Variance')
plt.grid()
plt.show(block=False)
```



The graph illustrates the first 100 principal components and their respective proportion of variance. It can be seen from the graph that first few principal components can explain the data fairly well since their variance proportion is high. It is important to keep in mind that, proportion of variance adds up to 1 when more components are used. When principal component's variance proportion is summed to 1, it means that original data is explained. Then, we will display the first 25 principal components.

[8]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(25):
    ax1 = plt.subplot(5,5,-i+25)
    ax1.imshow(PC25[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

The figure above represents the first 25 principal components. Those principal components are the directions that explain the original data best. It can be said that those first 25 principal components are best to explain eyes, nose and mouth areas.

Part b

In this part, we are asked to recounstruct the principal components to recreate the image. This principal will be done for first 10 PC, 25 PC and 50 PCs. First the original 36 images will be shown and then recreated versions for those 36 images with 10, 25 and 50 PCs will be shown respectively. The procedure to do that is:
Represent the projection of our image data onto the matrix defined by the chosen k principal components.

$$P = (X - \mu_X).V$$

where X is the image data, V is the matrix that contains the chosen k principal components and μ_X is the mean of X. Then, we should map the projection into the original space. So from P, we have to map $\hat{X}$. To do that,

$$\hat{X} = P.V^T + \mu_X$$

should be calculated.

This procedure is done in the function defined in part A already. The original 36 images and recreation of those images with 10, 25 and 50 PCs are given below:

[9]:
```
# Part B

pca_face10, var10, PC10 = PCA(faces, 10)
pca_face25, var25, PC25 = PCA(faces, 25)
pca_face50, var50, PC50 = PCA(faces, 50)
```

[10]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(36):
    ax1 = plt.subplot(6,6,i+1)
    ax1.imshow(faces[:,i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: The First 36 Images

[11]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
```

```
for i in range(36):
    ax1 = plt.subplot(6,6,i+1)
    ax1.imshow(pca_face10[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

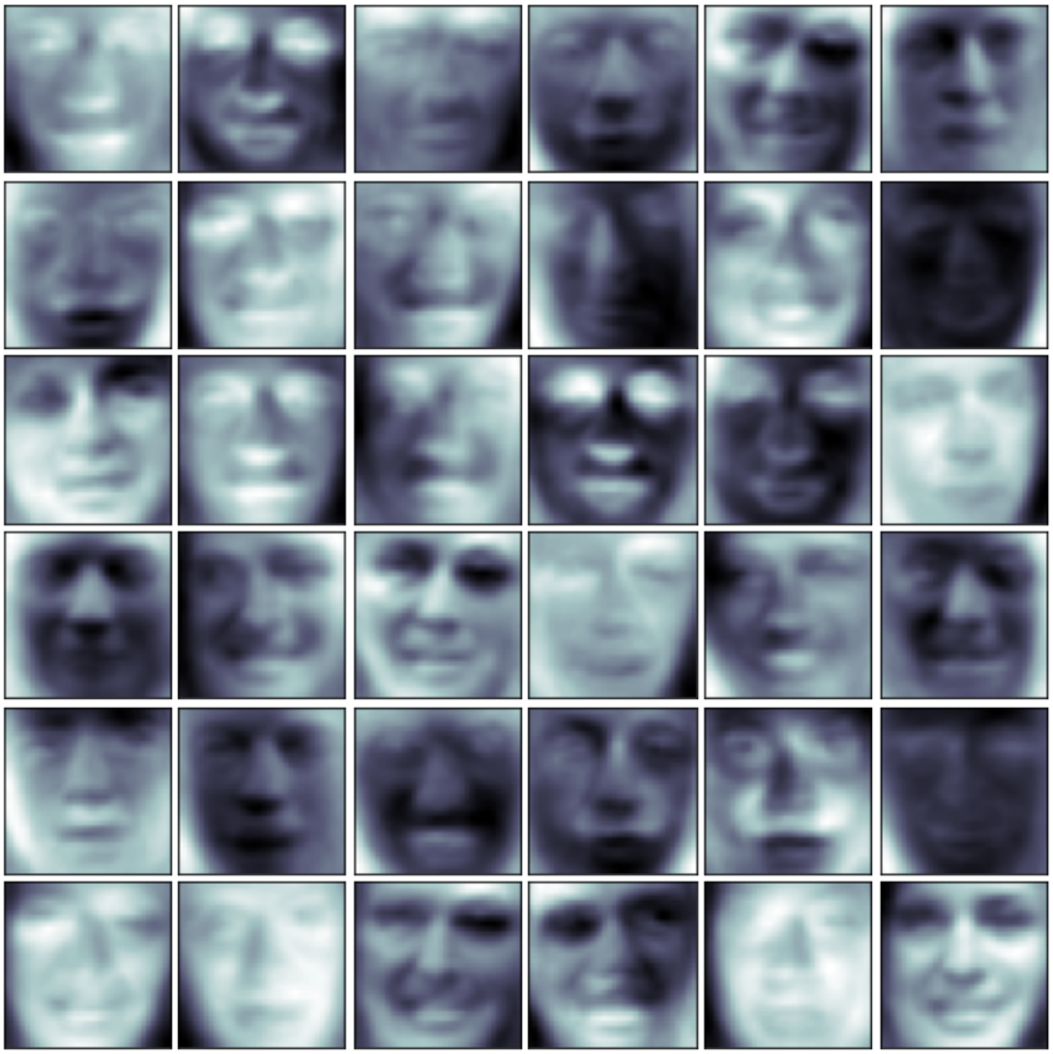


Figure: The Reconstructions with First 10 PCs

[12]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(36):
    ax1 = plt.subplot(6,6,i+1)
    ax1.imshow(pca_face25[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: The Reconstructions with First 25 PCs

[13]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(36):
    ax1 = plt.subplot(6,6,i+1)
    ax1.imshow(pca_face50[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: The Reconstructions with First 50 PCs

As can be seen from the figures, when more principal components used for reconstruction, images become more clear. But those reconstructions are not lossless. We will calculate the Mean Squared Errors(MSE) between the original image and reconstructed ones for each reconstruction. Mean and standart deviation for those errors are given below:

[14]:
```
# the mean and standard deviation
MSE_PCA10 = np.mean((pca_face10.T - faces) ** 2)
std_PCA10 = np.std(np.mean((faces.T - pca_face10) ** 2, axis=1))
MSE_PCA25 = np.mean((pca_face25.T - faces) ** 2)
std_PCA25 = np.std(np.mean((faces.T - pca_face25) ** 2, axis=1))
MSE_PCA50 = np.mean((pca_face50.T - faces) ** 2)
std_PCA50 = np.std(np.mean((faces.T - pca_face50) ** 2, axis=1))
```

[15]:
```
print('10 PCs:')
print('mean of MSEs = %f' % MSE_PCA10)
print('std of MSEs = % f' % std_PCA10)
print('\n')
print('25 PCs:')
print('mean of MSEs = %f' % MSE_PCA25)
```

```
print('std of MSEs = % f' % std_PCA25)
print('\n')
print('50 PCs:')
print('mean of MSEs = %f' % MSE_PCA50)
print('std of MSEs = % f' % std_PCA50)
```

```
10 PCs:
mean of MSEs = 828.361644
std of MSEs =  264.238297


25 PCs:
mean of MSEs = 637.376391
std of MSEs =  156.361695


50 PCs:
mean of MSEs = 503.544966
std of MSEs =  84.950508
```

When more principal components are used for reconstruction, MSE values are lower. This result is expected since with more PCs, more variance of the original data is explained, therefore better results are obtained.

Part c

In this part, another dimensionality reduction technique called Independent Component Analysis (ICA), will be used. ICA seperates the independent sources that are linearly mixed. For implementation of ICA, FastICA fuction from sklearn library is used. First 10, 25 and 50 ICs are found and plotted below:

[16]:
```
# Part C
```

[17]:
```
ica_component10 = FastICA(10)
ica_component10.fit(faces.T)
ica_component25 = FastICA(25)
ica_component25.fit(faces.T)
ica_component50 = FastICA(50)
ica_component50.fit(faces.T)
```

[18]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,3))
for i in range(10):
    ax1 = plt.subplot(2,5,i+1)
    ax1.imshow(ica_component10.components_[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

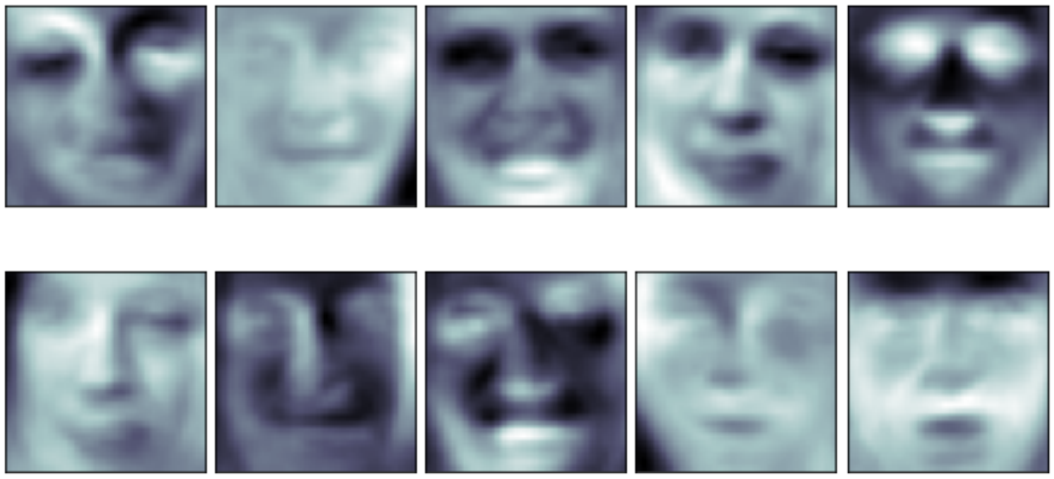


Figure: First 10 ICs

[19]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(25):
    ax1 = plt.subplot(5,5,i+1)
    ax1.imshow(ica_component25.components_[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: First 25 ICs

[20]:
```
figure_num += 1
plt.figure(figure_num,figsize=(10,5))
for i in range(50):
    ax1 = plt.subplot(5,10,i+1)
    ax1.imshow(ica_component50.components_[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: First 50 ICs

ICs explain eyes, nose and mouth as PCs do. However, PCs were a little bit more clearer than ICs. This may be due to randomness included in the ICA, it is not deterministic like PCA. Then just like we did for PCA, we will reconstruct the images for 10, 25 and 50 ICs.

[21]:
```
ica_face10 = ica_component10.fit_transform(faces) .dot(ica_component10.mixing_.T) +␣
 ↪ica_component10.mean_
ica_face25 = ica_component25.fit_transform(faces) .dot(ica_component25.mixing_.T) +␣
 ↪ica_component25.mean_
ica_face50 = ica_component50.fit_transform(faces) .dot(ica_component50.mixing_.T) +␣
 ↪ica_component50.mean_
ica_face10 = ica_face10.T
ica_face25 = ica_face25.T
ica_face50 = ica_face50.T
```

[22]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(30):
    ax1 = plt.subplot(6,5,i+1)
    ax1.imshow(ica_face10[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

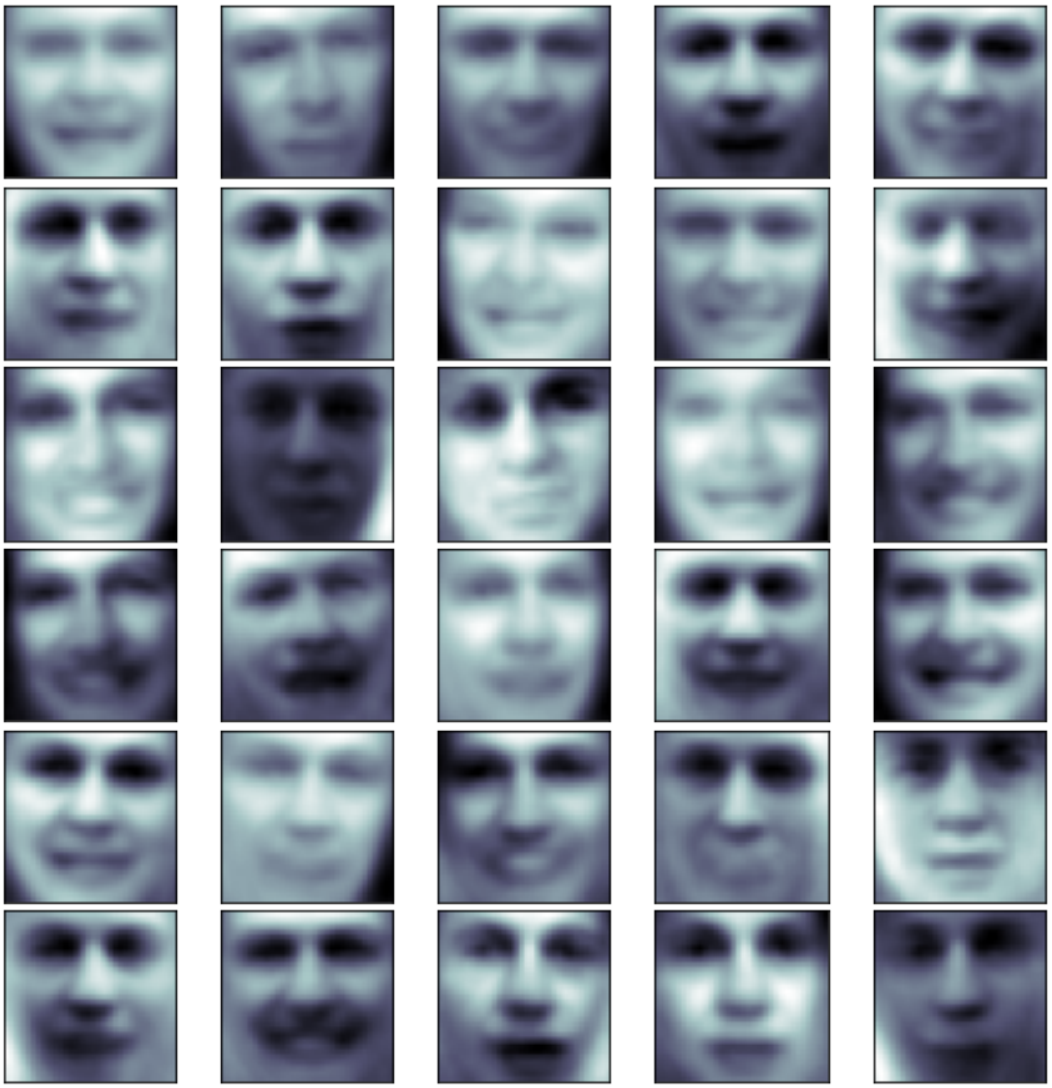


Figure: The Reconstructions with First 10 ICs

[23]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(30):
    ax1 = plt.subplot(6,5,i+1)
    ax1.imshow(ica_face25[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: The Reconstructions with First 25 ICs

[24]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(30):
    ax1 = plt.subplot(6,5,i+1)
    ax1.imshow(ica_face50[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: The Reconstructions with First 50 ICs

As expected, the more ICs used, the more clear the images become. Reconstruction errors for ICA models are given below:

[25]:
```
#the mean and standard deviation
MSE_ICA10 = np.mean((ica_face10.T - faces) ** 2)
std_ICA10 = np.std(np.mean((faces.T - ica_face10) ** 2, axis=1))
MSE_ICA25 = np.mean((ica_face25.T - faces) ** 2)
std_ICA25 = np.std(np.mean((faces.T - ica_face25) ** 2, axis=1))
MSE_ICA50 = np.mean((ica_face50.T - faces) ** 2)
std_ICA50 = np.std(np.mean((faces.T - ica_face50) ** 2, axis=1))
```

[26]:
```
print('10 ICs:')
print('mean of MSEs = %f' % MSE_ICA10)
print('std of MSEs = % f' % std_ICA10)
print('\n')
print('25 ICs:')
print('mean of MSEs = %f' % MSE_ICA25)
print('std of MSEs = % f' % std_ICA25)
```

```
print('\n')
print('50 ICs:')
print('mean of MSEs = %f' % MSE_ICA50)
print('std of MSEs = % f' % std_ICA50)
```

```
10 ICs:
mean of MSEs = 505.884160
std of MSEs =  246.670797


25 ICs:
mean of MSEs = 326.186755
std of MSEs =  150.232516


50 ICs:
mean of MSEs = 195.499545
std of MSEs =  82.737542
```

Just like in PCA; when more ICs are used, MSE errors are lower. This result was as expected.

Part d

Last technique that will be used for dimensionality reduction is Nonnegative Matrix Factorization (NNMF). NNMF aims to decompose data(X) into two lower rank matrices H and W.

$$argmin(||X - (W.H)||) \quad s.t \quad (W.H) \geqslant 0$$

What that equation means is that, NNMF is decomposing X into H and W such that the least square errors are smallest and matrix multiplication of W.H is non-zero. For NNMF implementation, NMF function of sklearn library is used.

[27]:
```
# Part D

nmf_10 = NMF(10, solver="mu")
nmf_component10 = nmf_10.fit_transform(faces.T + np.abs(np.min(faces.T)))

nmf_25 = NMF(25, solver="mu")
nmf_component25 = nmf_25.fit_transform(faces.T + np.abs(np.min(faces.T)))

nmf_50 = NMF(50, solver="mu")
nmf_component50 = nmf_50.fit_transform(faces.T + np.abs(np.min(faces.T)))
```

[28]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,3))
for i in range(10):
    ax1 = plt.subplot(2,5,i+1)
    ax1.imshow(nmf_10.components_[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

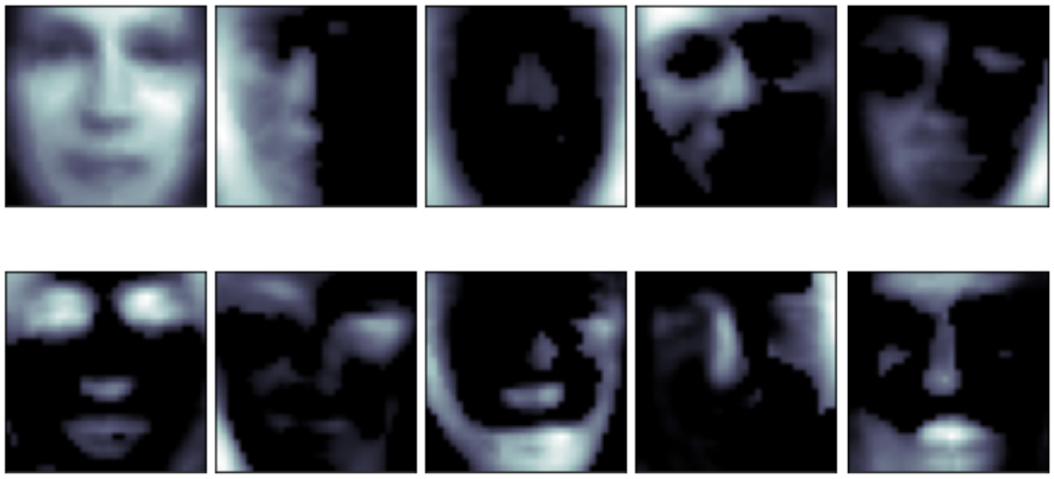


Figure: First 10 MFs

[29]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(25):
    ax1 = plt.subplot(5,5,i+1)
    ax1.imshow(nmf_25.components_[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: First 25 MFs

[30]:
```
figure_num += 1
plt.figure(figure_num,figsize=(10,5))
for i in range(50):
    ax1 = plt.subplot(5,10,i+1)
    ax1.imshow(nmf_50.components_[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: First 50 MFs

Then, we will reconstruct the images according to 10, 25 and 50 MFs. The reconstruction and images are given below:

[31]:
```
nmf_face10 = nmf_component10.dot(nmf_10.components_) - np.abs(np.min(faces.T))
nmf_face25 = nmf_component25.dot(nmf_25.components_) - np.abs(np.min(faces.T))
nmf_face50 = nmf_component50.dot(nmf_50.components_) - np.abs(np.min(faces.T))
# nmf_face10 = nmf_face10.T
# nmf_face25 = nmf_face25.T
# nmf_face50 = nmf_face50.T
```

[32]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(30):
    ax1 = plt.subplot(6,5,i+1)
    ax1.imshow(nmf_face10[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: The Reconstructions with First 10 MFs

[33]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(30):
    ax1 = plt.subplot(6,5,i+1)
    ax1.imshow(nmf_face25[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: The Reconstructions with First 25 MFs

[34]:
```
figure_num += 1
plt.figure(figure_num,figsize=(6,6))
for i in range(30):
    ax1 = plt.subplot(6,5,i+1)
    ax1.imshow(nmf_face50[i].reshape(32,32).T, cmap=plt.cm.bone)
    ax1.set_yticks([])
    ax1.set_xticks([])
plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=1)
plt.show()
```

Figure: The Reconstructions with First 50 MFs

MSE losses are given below:

[35]:
```
# the mean and standard deviation
MSE_NMF10 = np.mean((nmf_face10.T - faces) ** 2)
std_NMF10 = np.std(np.mean((faces.T - nmf_face10) ** 2, axis=1))
MSE_NMF25 = np.mean((nmf_face25.T - faces) ** 2)
std_NMF25 = np.std(np.mean((faces.T - nmf_face25) ** 2, axis=1))
MSE_NMF50 = np.mean((nmf_face50.T - faces) ** 2)
std_NMF50 = np.std(np.mean((faces.T - nmf_face50) ** 2, axis=1))
```

[36]:
```
print('10 MFs:')
print('mean of MSEs = %f' % MSE_NMF10)
print('std of MSEs = % f' % std_NMF10)
print('\n')
print('25 MFs:')
print('mean of MSEs = %f' % MSE_NMF25)
print('std of MSEs = % f' % std_NMF25)
print('\n')
```

```
print('50 MFs:')
print('mean of MSEs = %f' % MSE_NMF50)
print('std of MSEs = % f' % std_NMF50)
```

```
10 MFs:
mean of MSEs = 711.185217
std of MSEs =  373.310572


25 MFs:
mean of MSEs = 549.875278
std of MSEs =  277.780251


50 MFs:
mean of MSEs = 424.952589
std of MSEs =  208.793500
```

Again, losses are as expected since when more components are used, error is lower.

When comparing all the results, ICA seems to be best in terms of MSE losses. But when image clarity is compared between three techniques, PCA seems to have clear images but it has more MSE error than ICA. However, i implemented PCA manually and used prebuild libraries for ICA and NNMF. So, the high MSE error of PCA can be a consequence of my implementation of the technique.

Question 2

In this question, 21 neuron having Gaussian shaped tuning curves with amplitude 1, standart daviation of 1 and centers μ_i that are evenly spaced between -10 and 10 are given.

Part a

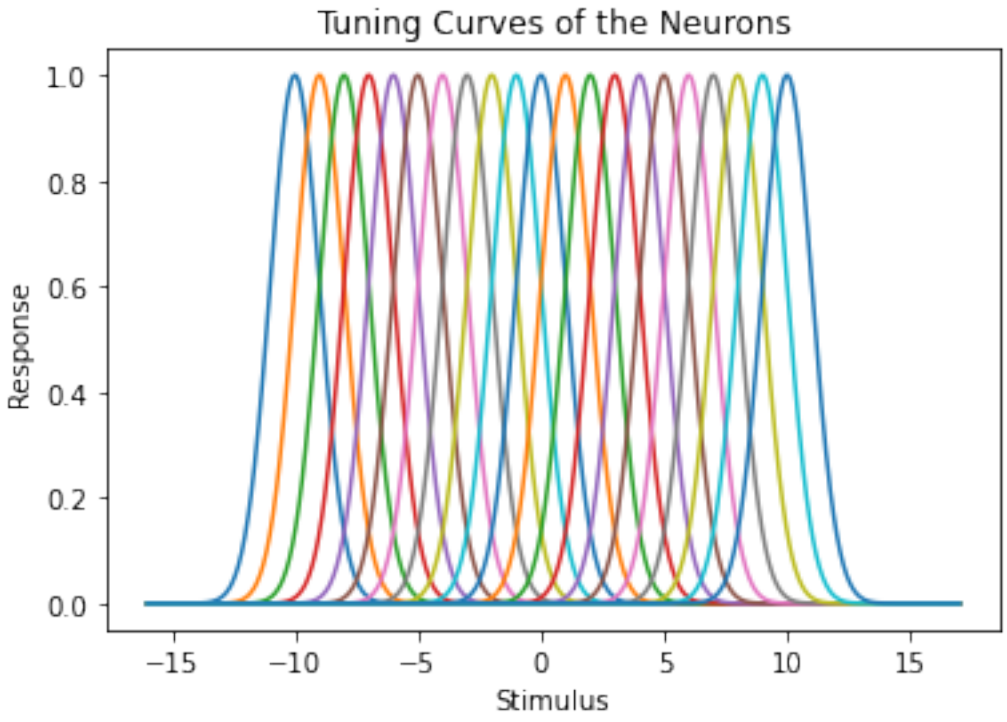
Visualization of tuning curves for all 21 neurons is given below.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
# Part A

def tuningCurves(A, x, mu, sigma):
    return A * np.exp(-((x - mu) ** 2) / (2 * (sigma ** 2)))
```

[3]:
```
mu = np.arange(-10,11)
responses = []
for i in range(len(mu)):
    responses.append(tuningCurves(1,np.linspace(-16, 17, 750), mu[i],1))
```

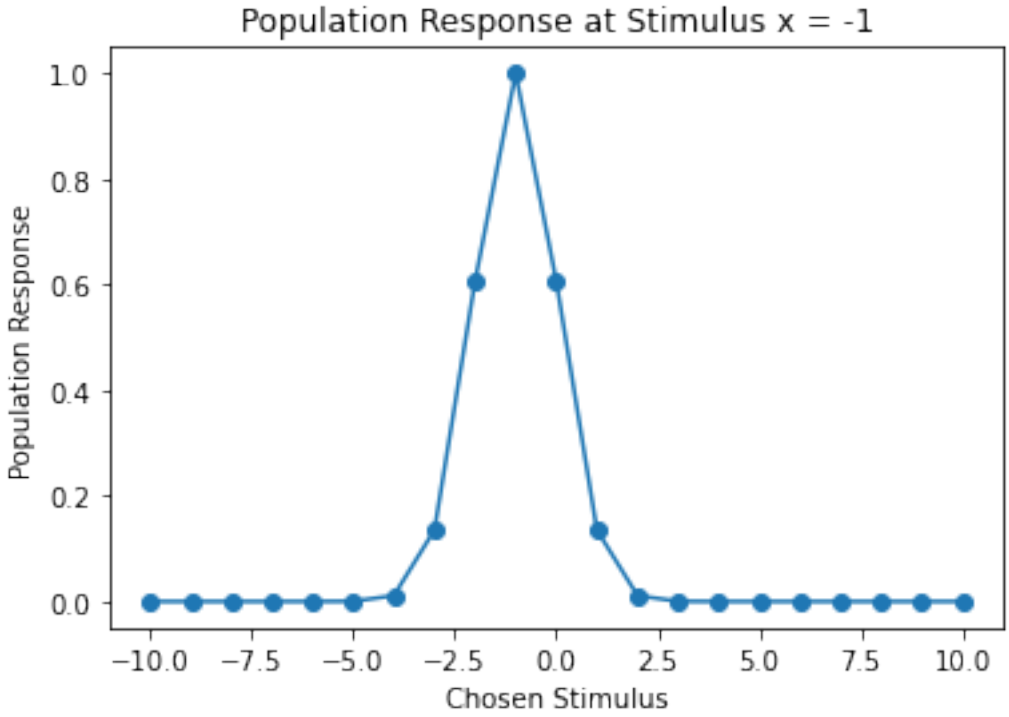
[4]:
```
fig_num = 0
plt.figure(fig_num)
plt.title('Tuning Curves of the Neurons')
plt.xlabel('Stimulus')
plt.ylabel('Response')
for i in range(len(responses)):
    plt.plot(np.linspace(-16, 17, 750), responses[i])
plt.show(block=False)
```



It can be seen from the figure that center of each neuron is the mean of its tuning curve. Then, we will visualize the response of the population to the stimulus x = -1.

[5]:
```
response_x = []
for i in range(len(mu)):
    response_x.append(tuningCurves(1,-1, mu[i],1))
```

[6]:
```
fig_num += 1
plt.figure(fig_num)
plt.title('Population Response at Stimulus x = -1')
plt.xlabel('Chosen Stimulus')
plt.ylabel('Population Response')
plt.plot(mu, response_x, marker='o')
plt.show(block=False)
```



It can be seen from the figure that, neuron with chosen stimuli elicits higher response than other neurons.

Part b

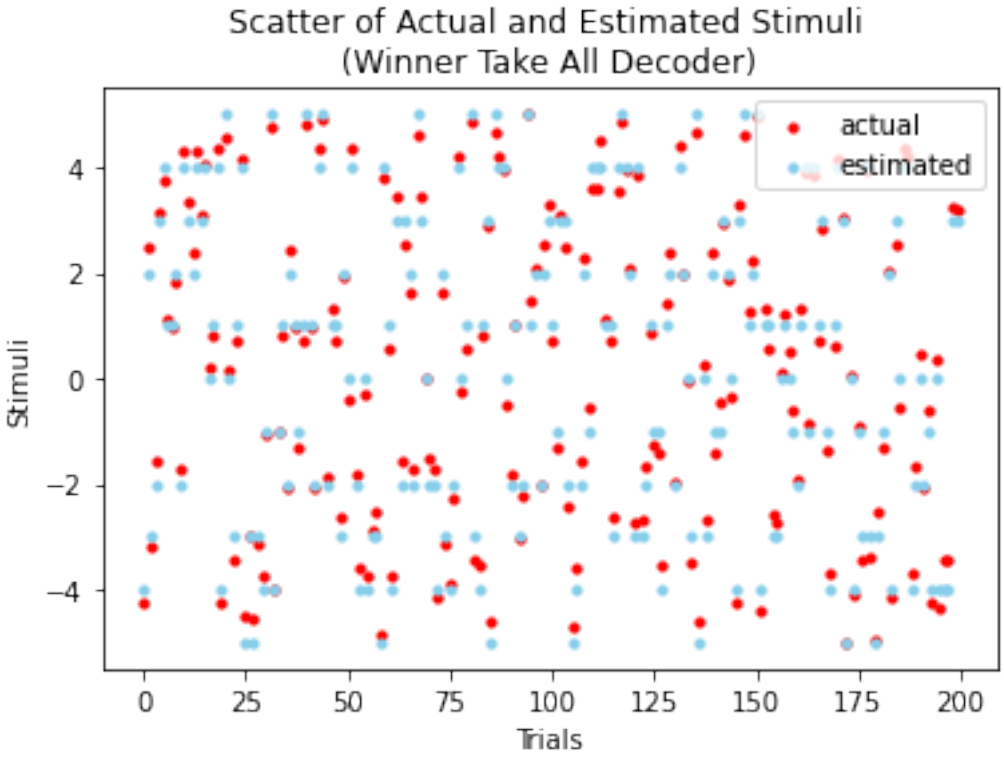
In this part, Winner Take All (WTA) decoder will be implemented. WTA decoder allows the neuron with the highest response to stay active and other neurons diminishes. So the neuron with the highest response acts as the chosen neuron. We can express it as:

$$\hat{x} = \mu_i \quad s.t \quad i = \underset{i}{\text{argmax}}(response_i)$$

where $\hat{x}$ is estimated stimuli and $response_i$ is the i^{th} neuron's response. This behavior is similar to second figure in 'part a' with input stimuli x = -1.
Experiment with 200 trials will be made. Input stimuli is between [-5, 5]. Each neuron's response has a Gaussian noise with 0 mean and 1/20 standart deviation. Implementation of the experiment is given below:

[7]:
```
# Part B

numberOfTrials = 200
responses_B = []
stimuli = []
est_WTA = []
error_WTA = []
np.random.seed(7)
for i in range(numberOfTrials):
    response_B = []
    random = 10 * np.random.random_sample() - 5
    stimuli.append(random)
    for k in range(len(mu)):
        response_B.append(tuningCurves(1,stimuli[i], mu[k],1))
    response_B = response_B + np.random.normal(0, 0.05, 21)
    chosen_index = np.argmax(response_B)
    est_WTA.append(mu[chosen_index])
    error_WTA.append(np.abs(stimuli[i] - est_WTA[i]))
    responses_B.append(response_B)
error_WTA_mean = np.mean(error_WTA)
error_WTA_std = np.std(error_WTA)
```

[22]:
```
fig_num += 1
plt.figure(fig_num)
plt.xlabel('Trials')
plt.ylabel('Stimuli')
plt.title('Scatter of Actual and Estimated Stimuli \n(Winner Take All Decoder)')
x_index = np.arange(0,numberOfTrials)
plt.scatter(x_index, stimuli, color='r', s=10)
plt.scatter(x_index, est_WTA, color='skyblue', s=10)
plt.legend(['actual', 'estimated'], loc='upper right')
plt.show(block=False)
```



In the scatter plot, actual and estimated stimuli for each trial using WTA is visualized. Then, error will be calculated. Mean and standard deviation of the error will be shown.

[9]:
```
print('Mean of error:', error_WTA_mean)
print('Standard deviation of error:', error_WTA_std)
```

```
Mean of error: 0.2663288643871281
Standard deviation of error: 0.15168019788290255
```

The mean of error for WTA is 0.26 which is not low. The reason behind that much of error may be since in this experiment, input stimuli was limited and a small set which was between [-5 ,5].

Part c

In this part, the same experiment will be repeated but the only change will be the decoder type. This time, Maximum Likelihood Decoder(MLD) will be used instead of WTA. MLD estimates the likelihood function of the given input. On other words, MLD tries to maximize the probability that the chosen stimuli elicited the simulated response. It can be expressed as:

$$\hat{x} = \underset{x}{\text{argmax}}(L(x)) = \underset{x}{\text{argmax}}(P(response|x))$$

where the response (r) is the vector that contains all 21 neurons' responses. Then response vector can be expressed as:

$$r = [r_1, r_2, ..., r_{21}]$$
$$r_i(x) = f_i(x) + Noise$$
$$r_i(x) = f_i(x) + N(0, (\sigma/20)^2)$$
$$r_i(x) \sim N(f_i(x), (\sigma/20)^2)$$
$$f(r_i(x)) = \frac{1}{\sqrt{2\pi}(\sigma/20)} e^{\frac{-(r_i(x)-f_i(x))^2}{2(\sigma/20)^2}}$$

Then we can write the L(X) = P(r | x) as:

$$L(X) = P(r|x) = \prod_{i=1}^{21} f(r_i(x))$$

To make the computation easier, we will compute log(L(X)) instead of L(X). Then the equation becomes:

$$log(L(X)) = l(x) = \sum_{i=1}^{21} f(r_i(x))$$

$$log(L(X)) = l(x) = \sum_{i=1}^{21} \frac{1}{\sqrt{2\pi}(\sigma/20)} e^{\frac{-(r_i(x)-f_i(x))^2}{2(\sigma/20)^2}}$$

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

$$l(x) \propto -\sum_{i=1}^{21} (r_i(x) - f_i(x))^2$$

Since l(x) is minus termed, we are required to maximize it. Instead, we can minimize the -l(x).

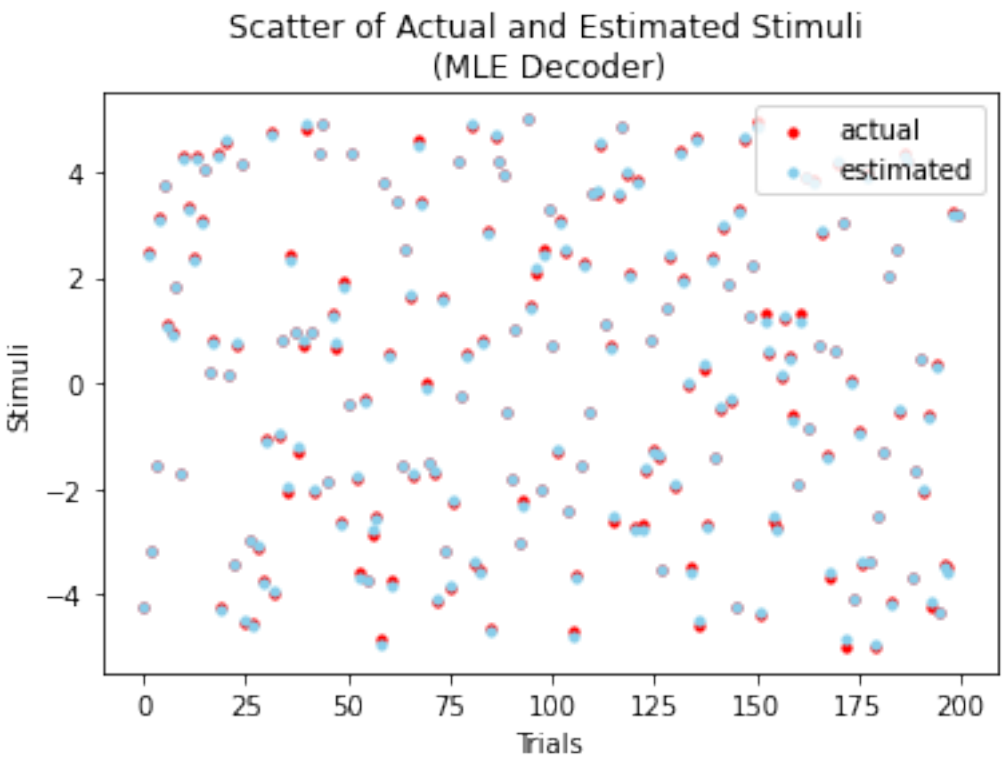
$$\hat{x} = \underset{x}{\text{argmin}}(-l(x)) = \underset{x}{\text{argmin}}(\sum_{i=1}^{21} (r_i(x) - f_i(x))^2)$$

When ML decoder applied to Normal distributions,it relates to least squares error. Python code for MLD is given below:

[10]:
```
# Part C

def MLE_decoder(A, x, mu, sigma, response):
    loglikelihood = 0
    loglikelihoods = []
    for i in range(len(x)):
        for k in range(len(mu)):
            loglikelihood += (response[k] - tuningCurves(A, x[i], mu[k], sigma)) **2
        loglikelihoods.append(loglikelihood)
        loglikelihood = 0
    min_index = np.argmin(loglikelihoods)
    est_stim = x[min_index]
    return est_stim
```

[11]:
```
est_MLE = []
error_MLE = []
for i in range(len(responses_B)):
    est_MLE.append(float(MLE_decoder(1, np.linspace(-5, 5, 500), mu, 1,␣
 ↪responses_B[i])))
    error_MLE.append(float(np.abs(stimuli[i] - est_MLE[i])))
error_MLE_mean = np.mean(error_MLE)
error_MLE_std = np.std(error_MLE)
```

[12]:
```
fig_num += 1
plt.figure(fig_num)
plt.xlabel('Trials')
plt.ylabel('Stimuli')
plt.title('Scatter of Actual and Estimated Stimuli \n(MLE Decoder)')
x_index = np.arange(0,numberOfTrials)
plt.scatter(x_index, stimuli, color='r', s=10)
plt.scatter(x_index, est_MLE, color='skyblue', s=10)
plt.legend(['actual', 'estimated'], loc='upper right')
plt.show(block=False)
```



From the figure, we can easily say that MLD did better than WTA in this experiment. Compute the error to compare two decoders.

[13]:
```
print('Mean of error:', error_MLE_mean)
print('Standard deviation of error:', error_MLE_std)
```

```
Mean of error: 0.04142448115458796
Standard deviation of error: 0.03183150244858742
```

Mean of error is 0.041 which is much lower than mean error of WTA. It is obvious both from figure and from error comparison that the MLD did much better than WTA.

Part d

In this part, we conduct the same experiment once more but this time with Maximum-a-Posterior(MAP) Decoder. MAP decoder maximizes posterior probability instead of likelihood. Thus, MAP decoder can use prior information on the input. Using Bayes' Rule:

$$P(x|r) \propto P(r|x)P(x) = L(x)P(x)$$

where L(x) is the likelihood function and P(x) is the prior.

$$\hat{x} = \underset{x}{\operatorname{argmax}}(L(x)P(x))$$

Like ML Decoder, calculating log(P(x | r)) is easier. We can express log(P(x | r)) as:

$$log(P(x|r)) \propto log(P(r|x)P(x)) = log(L(x)) + log(P(x)) = l(x) + log(P(x))$$

In the question, prior is given as a Gaussian with 0 mean and standart deviaton of 2.5. So we can write log(P(x)) as :

$$log(P(x)) = log(\frac{1}{\sqrt{2\pi}(2.5)} e^{\frac{0-x^2}{2(2.5)^2}})$$

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

We know l(x) from previous part and log(P(x)) is found. So we can write the log(P(x | r)) as:

$$log(P(x|r)) = -\frac{1}{2(\sigma/20)^2}\sum_{i=1}^{21}(r_i(x) - f_i(x))^2 - \frac{x^2}{2(2.5)^2}$$

Like we did in ML decoder, instead of maximizing log(P(x | r)), we can minimize -log(P(x | r)).

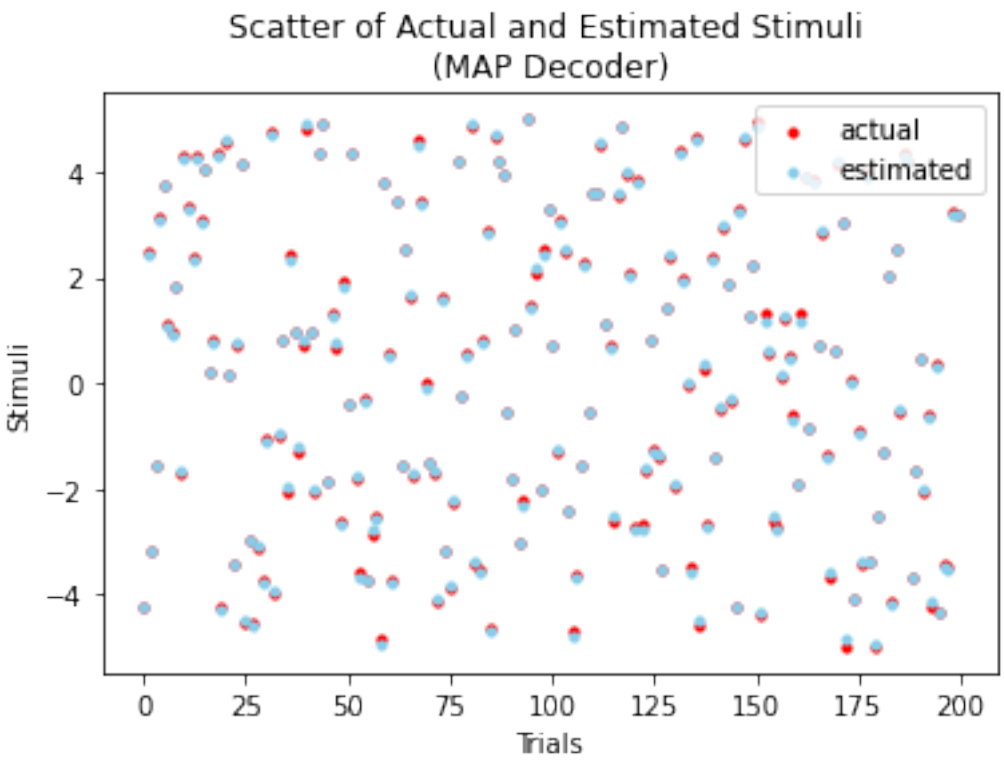
$$\hat{x} = \underset{x}{\text{argmax}}(-log(P(x|r))) = \underset{x}{\text{argmax}}(\frac{1}{2(\sigma/20)^2}\sum_{i=1}^{21}(r_i(x) - f_i(x))^2 + \frac{x^2}{2(2.5)^2})$$

The python code for that is given below:

[14]:
```
# Part D

def MAP_decoder(A, x, mu, sigma, response):
    logPosterior = 0
    logPosteriors = []
    for i in range(len(x)):
        for k in range(len(mu)):
            logPosterior += (response[k] - tuningCurves(A, x[i], mu[k], sigma)) **2
        logPosterior = (logPosterior / (2 * (sigma / 20) ** 2)) + (x[i] ** 2) / (2␣
 ↪* 2.5 ** 2)
        logPosteriors.append(logPosterior)
        logPosterior = 0
    min_index = np.argmin(logPosteriors)
    est_stim = x[min_index]
    return est_stim
```

[15]:
```
est_MAP = []
error_MAP = []
for i in range(len(responses_B)):
    est_MAP.append(float(MAP_decoder(1, np.linspace(-5, 5, 500), mu, 1,␣
 ↪responses_B[i])))
    error_MAP.append(float(np.abs(stimuli[i] - est_MAP[i])))
error_MAP_mean = np.mean(error_MAP)
error_MAP_std = np.std(error_MAP)
```

[16]:
```
fig_num += 1
plt.figure(fig_num)
plt.xlabel('Trials')
plt.ylabel('Stimuli')
plt.title('Scatter of Actual and Estimated Stimuli \n(MAP Decoder)')
x_index = np.arange(0,numberOfTrials)
plt.scatter(x_index, stimuli, color='r', s=10)
plt.scatter(x_index, est_MAP, color='skyblue', s=10)
plt.legend(['actual', 'estimated'], loc='upper right')
plt.show(block=False)
```



We can see like ML decoder, MAP also does a great job of estimating. Calculation of errors in MAP given below:

[17]:
```
print('Mean of error:', error_MAP_mean)
print('Standard deviation of error:', error_MAP_std)
```

```
Mean of error: 0.04149586321863232
Standard deviation of error: 0.03127356853566808
```

Mean of error for MAP is 0.04149. It is slightly higher than ML Decoder's error mean. However standart deviation of ML Decoder's error is slightly higher than standart deviation of MAP's error. MAP and ML Decoder performs similar. In theory, MAP supposed to perform better since there is additional prior information. In conclusion, both MAP and ML decoders perform well and they are both better than WTA in this experiment.

[18]:
```
# Part E

In this part, we are asked to perform 200 trials of stimulus intensity for␣
↪different sigma values. Then ML estimates of the stimulus will be found and␣
↪errors for each sigma value will be calculated and commented. Python code for␣
↪that is given below:

sigmas = [0.1, 0.2, 0.5, 1, 2, 5]
```

[19]:
```
numberOfTrials = 200
responses_E = []
stimuli_E = []
est_MLE_E = []
error_MLE_E = []
errors_MLE_E = []
np.random.seed(5)
```

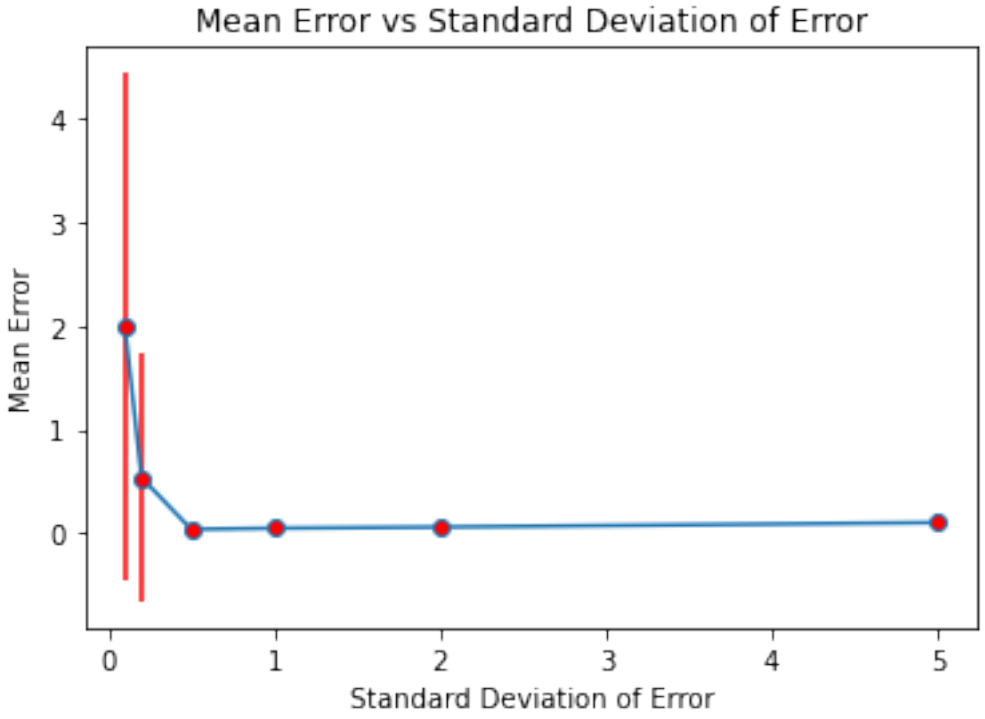
```
for i in range(numberOfTrials):
    response_E = []
    random = 10 * np.random.random_sample() - 5
    stimuli_E.append(random)
    error_MLE_E = []
    for k,sigma in enumerate(sigmas):
        response_E = (tuningCurves(1,stimuli_E[i], mu,sigma)) + np.random.normal(0,␣
↪0.05, 21)
        est_MLE_E.append(MLE_decoder(1, np.linspace(-5, 5, 500), mu, sigma,␣
↪response_E))
        error_MLE_E.append(np.abs(stimuli_E[i] - float(est_MLE_E[i*6 + k])))
        responses_E.append(response_E)
    errors_MLE_E.append(error_MLE_E)
errors_MLE_E = np.array(errors_MLE_E)
est_MLE_E = np.array(est_MLE_E)
responses_E = np.array(responses_E)
stimuli_E = np.array(stimuli_E)
```

[20]:
```
errors_MLE_E_mean = []
errors_MLE_E_std = []
for i in range(len(sigmas)):
    error_MLE_E_mean = np.mean(errors_MLE_E[:,i])
    error_MLE_E_std = np.std(errors_MLE_E[:,i])
    print('sigma = %.1f' % sigmas[i])
    print('Mean of errors', error_MLE_E_mean)
    print('Standard deviation of errors ', error_MLE_E_std)
    print('\n')
    errors_MLE_E_mean.append(error_MLE_E_mean)
    errors_MLE_E_std.append(error_MLE_E_std)
errors_MLE_E_mean = np.array(errors_MLE_E_mean)
errors_MLE_E_std = np.array(errors_MLE_E_std)
```

```
sigma = 0.1
Mean of errors 1.9890657370175808
Standard deviation of errors  2.448906135786968


sigma = 0.2
Mean of errors 0.533970544896922
Standard deviation of errors  1.1950532407851018


sigma = 0.5
Mean of errors 0.03569769588945639
Standard deviation of errors  0.032079236966028025


sigma = 1.0
Mean of errors 0.04641636808837287
Standard deviation of errors  0.03708667913017074


sigma = 2.0
Mean of errors 0.05729291259648481
Standard deviation of errors  0.040685873500029966


sigma = 5.0
```

```
Mean of errors 0.10065260782811584
Standard deviation of errors  0.07667936215460809
```

[21]:
```
fig_num += 1
plt.figure(fig_num)
plt.xlabel('Standard Deviation of Error')
plt.ylabel('Mean Error')
plt.title('Mean Error vs Standard Deviation of Error')
plt.errorbar(sigmas, errors_MLE_E_mean, yerr=errors_MLE_E_std,
             marker='o', markerfacecolor='r', ecolor='r')

plt.show(block=False)
```



Narrow tuning curves are more effected by the noise. Wider tuning curves are better for dealing with noise since they are less effected. We can see from the error calculations that the best result is obtained when sigma = 0.5. Also, we can conclude from the errors that, as sigma values move away from the 0.5 both mean and standard deviation of error starts to increase.