Homework 4

EEE482- Computational Neuroscience

Efe Acer
21602217

Bilkent University, CS

Contents

Question 1

Experimental stimuli are provided in the question contains 1000 face images downsampled to a 32x32 square grid. The question asks us to apply dimensionality reduction techniques that try to express the variance structure of the stimuli in less number of dimensions.

A sample stimulus is shown below:

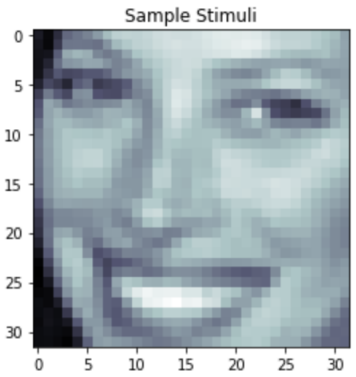

Figure 1: Sample stimulus

1.a) The first dimensionality reduction technique we use is *Principal Component Analysis (PCA)*. PCA basically applies eigenvalue decomposition to the covariance matrix of the data after an optional but recommended whitening procedure. Then, PCA selects the k most significant eigenpairs, which are the k largest eigenvalues and their corresponding eigenvectors. After this selection, PCA projects the data onto the space of the selected eigenpairs to obtain a lower dimensional representation of the data that still explains the variance structure well enough. The eigenvectors sorted with respect to the descending order of eigenvalues are said to be the *Principal Components (PCs)*.

In `Python`, `sklearn` library has a good and well documented implementation of `PCA` in its `decomposition` module. Hence, `sklearn`'s `PCA` is used in this part. The usage is as follows:

```
1 pca = PCA(100, whiten=True)
2 pca.fit(faces)
```

The `faces` array given as an input to `pca.fit()` is a matrix that contains the 1024 pixels of each image as its rows. Once `fit` is called on the `pca` object the decomposition is performed and the results are ready to be accessed. Note that the decomposition only computed the first 100 PCs.

`pca.explained_variance_ratio_` gives an array containing the proportion of variance explained by each individual PC. The figure below shows a plot of these values:

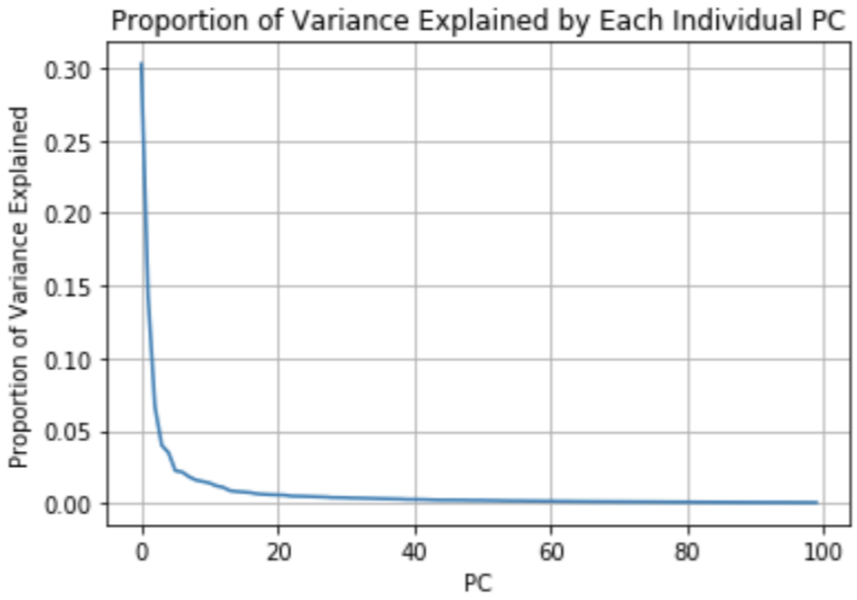

Figure 2: Proportion of Variance Explained by Each Individual PC

It can be seen from the figure that the first few PCs can actually explain the variance structure of our data fairly well. This is what makes PCA a very important tool, we can reduce the number of features significantly while keeping the variance structure of the data unchanged. Notice that the proportion of variance explained cumulatively adds up to 1 as we use more PCs. This is normal since we need to use all PCs to completely explain the original variance structure.

The code below is the `Python` equivalent of the `dispImArray.m` given for `Matlab`:

```
plt.figure(figure_num)
figure_num += 1
fig, axes = plt.subplots(5, 5, figsize=(8,8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.01, wspace=0.01))
for i, ax in enumerate(axes.flat):
    ax.imshow(pca.components_[i].reshape(32, 32).T, cmap=plt.cm.bone)
```

The figure at the beginning of the next page is obtained using the code above. It is a visualization of the first 25 PCs stored in the `pca.components_` array. The PCs are also referred to as eigenfaces.

Figure 3: The first 25 PCs

These 25 PCs are the directions in the original feature space that best describe the variance structure of the data. We can see that the PCs capture some facial areas such as eyes, nose and lips.

1.b) Let X denote the matrix containing our data (`faces` array) and V denote the matrix containing the largest k eigenvectors as its columns. Then, $Z = (X - \mu_X) \cdot V$ gives the projection of X onto the space defined by the k most important PCs, where μ_X is the mean of X subtracted due to whitening. Note that X is 1000x1024 and the Z is 1000 x k, since Z contains the PCA projections.

To map the projections back to the original feature space (the one having 1024 features) we need to multiply Z with V^T. μ_X must also be added to $Z \cdot V^T$ since we have enabled whitening. Final reconstruction of the $faces$ array, $\hat{X}$ is given by:

$$\hat{X} = Z \cdot V^T + \mu_X = (X - \mu_X) \cdot VV^T + \mu_X \tag{1}$$

Reconstructions based on the first 10, 25 and 50 PCs can be obtained using the following code:

```
1 # Obtain the reconstructions
2 faces_PCA_10 = (faces - pca.mean_).dot(pca.components_[0:10].T).dot(pca.
    components_[0:10]) + pca.mean_
```

```
3 faces_PCA_25 = (faces - pca.mean_).dot(pca.components_[0:25].T).dot(pca.
      components_[0:25]) + pca.mean_
4 faces_PCA_50 = (faces - pca.mean_).dot(pca.components_[0:50].T).dot(pca.
      components_[0:50]) + pca.mean_
```

Following figures show the first 36 original images and their reconstructions based on the first 10, 25 and 50 PCs:

Figure 4: The First 36 Original Images

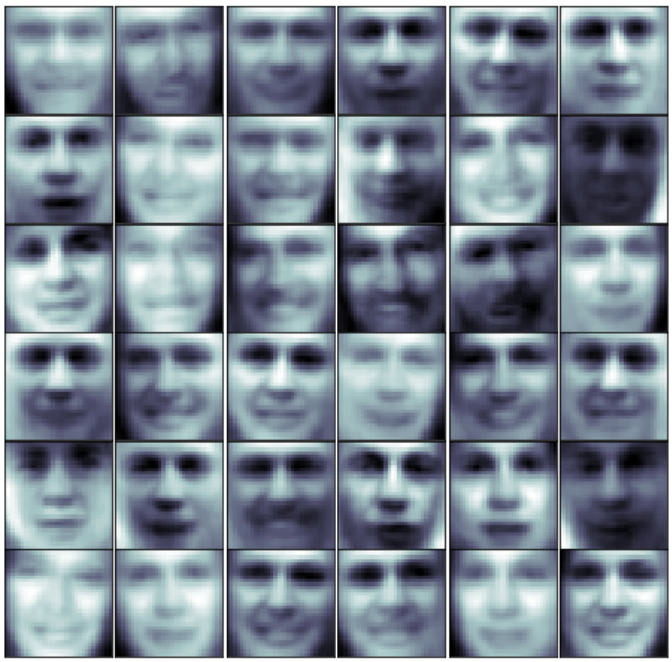

Figure 5: The First 36 Reconstructions Based on the First 10 PCs

Figure 6: The First 36 Reconstructions Based on the First 25 PCs

Figure 7: The First 36 Reconstructions Based on the First 50 PCs

The reconstructions we obtained are obviously not lossless. Hence, we compute the mean squared errors (MSE) between the original and reconstructed images, and report the mean and standard deviation of MSE across 1000 images. The code provided below performs the necessary calculations:

```
# Compute the mean and standard deviation of the reconstruction losses
losses_PCA_10 = (faces - faces_PCA_10) ** 2
MSE_PCA_10, std_PCA_10 = np.mean(losses_PCA_10), np.std(np.mean(losses_PCA_10,
    axis=1))
losses_PCA_25 = (faces - faces_PCA_25) ** 2
MSE_PCA_25, std_PCA_25 = np.mean(losses_PCA_25), np.std(np.mean(losses_PCA_25,
    axis=1))
losses_PCA_50 = (faces - faces_PCA_50) ** 2
MSE_PCA_50, std_PCA_50 = np.mean(losses_PCA_50), np.std(np.mean(losses_PCA_50,
    axis=1))
```

The results are as follows:

```
Reconstruction loss (10 PCs): mean of MSEs = 523.241745 , std of MSEs = 257.641201
Reconstruction loss (25 PCs): mean of MSEs = 332.256492 , std of MSEs = 153.110285
Reconstruction loss (50 PCs): mean of MSEs = 198.425252 , std of MSEs = 84.180230
```

The results are as expected since the increased number of PCs result in lower MSE values with less variability. Although first 10 PCs contain the most of the variability in the images, they are not sufficient to obtain a precise reconstruction of the original images. This is because even the least important PCs have details embedded in them. We can observe this by looking at the images above and seeing that they become more detailed as we use more PCs.

1.c) Another dimensionality reduction technique that we consider is *Independent Component Analysis (ICA)*. ICA aims to separate independent sources that are linearly mixed. Unlike PCA, the ICA algorithm involves some randomness; in other words ICA is not deterministic. `FastICA` of `sklearn.decomposition` is used in this questions. Different models are initialized for 10, 25 and 50 ICs as follows:

```
ica_10 = FastICA(10, whiten=True, random_state=np.random.seed(7))
ica_25 = FastICA(25, whiten=True, random_state=np.random.seed(7))
ica_50 = FastICA(50, whiten=True, random_state=np.random.seed(7))
ica_10.fit(faces)
ica_25.fit(faces)
ica_50.fit(faces)
```

The following figures show the ICs obtained by each model:

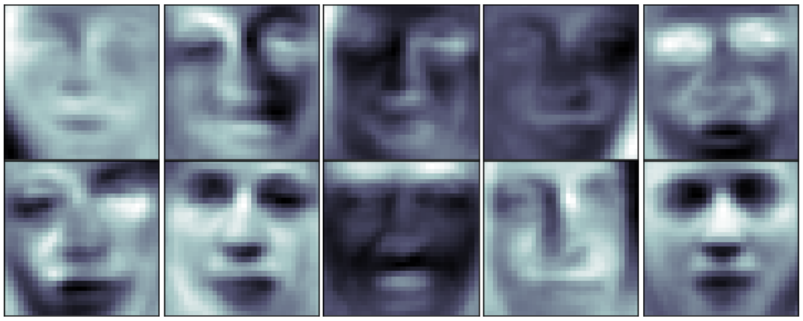

Figure 8: 10 ICs

Figure 9: 25 ICs

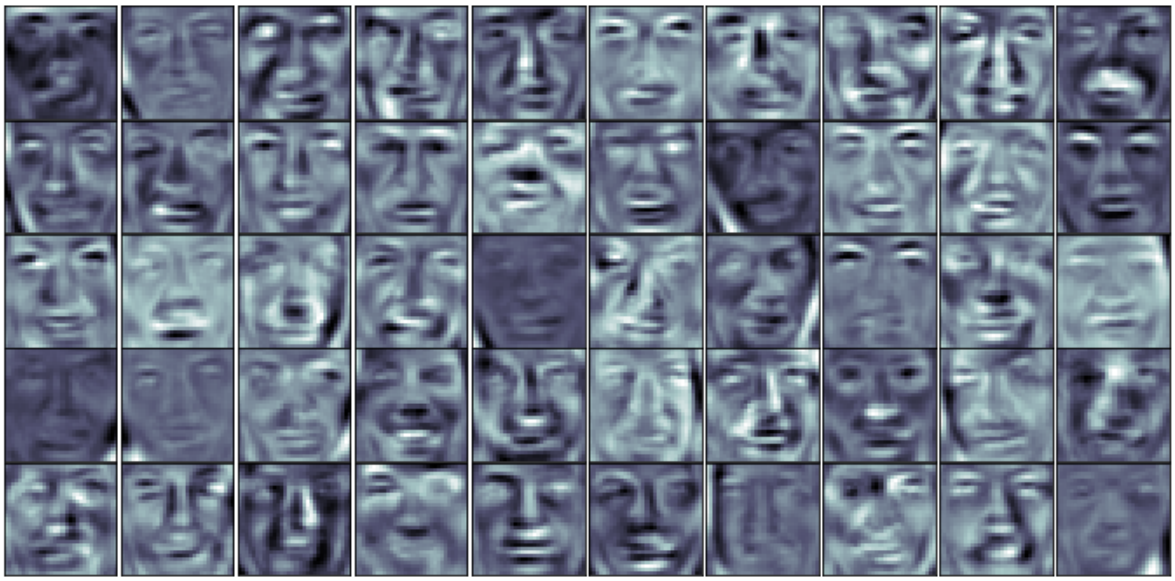

Figure 10: 50 ICs

The ICs show some facial parts such as eyes, nose and lips as PCs showed. However, they are harder to interpret possibly due to the randomness included in the `FastICA` implementation.

Reconstructions of the stimuli can be obtained by the ICA models as follows:

```
S_10 = ica_10.fit_transform(faces)
A_10 = ica_10.mixing_
faces_ICA_10 = S_10.dot(A_10.T) + ica_10.mean_
S_25 = ica_25.fit_transform(faces)
A_25 = ica_25.mixing_
faces_ICA_25 = S_25.dot(A_25.T) + ica_25.mean_
S_50 = ica_50.fit_transform(faces)
A_50 = ica_50.mixing_
faces_ICA_50 = S_50.dot(A_50.T) + ica_50.mean_
```

The reconstructed images are displayed below:

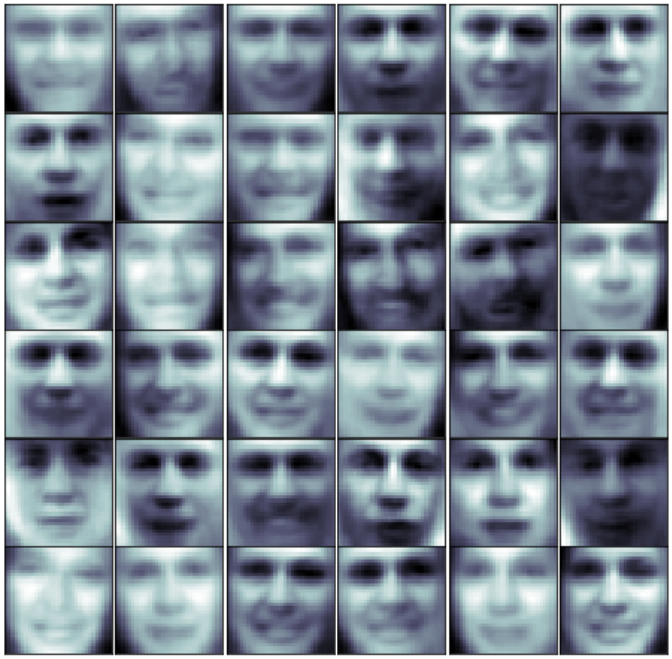

Figure 11: ICA Reconstructions of the First 25 Stimuli (10 IC model is used)

Figure 12: ICA Reconstructions of the First 25 Stimuli (25 IC model is used)

Figure 13: ICA Reconstructions of the First 25 Stimuli (50 IC model is used)

The code output shown below displays the reconstruction errors for the ICA models:

```
Reconstruction loss (10 ICs): mean of MSEs = 523.241745 , std of MSEs = 257.641200
Reconstruction loss (25 ICs): mean of MSEs = 332.256492 , std of MSEs = 153.110288
Reconstruction loss (50 ICs): mean of MSEs = 198.425067 , std of MSEs = 84.179966
```

The reconstruction errors are exactly the same with the PCA's reconstruction errors. This is because `sklearn.decomposition`'s `FastICA` does not accept a parameter that allows us to specify how many eigenvectors to use in the reconstruction. Thus, `FastICA` uses all by default; which yields a result very similar to PCA.

1.d) As the last dimensionality reduction technique we consider *Nonnegative Matrix Factorization (NNMF)*. NNMF tries to decompose the input matrix X as the multiplication of two lower rank matrices W and H. W is n x r and H is r x k; where n is the number of data points, k is the number of input features and r is the reduced number of features. NNMF can be summarized mathematically as:

$$\underset{W,H}{argmin}(||X - (W \cdot H)||) \text{ subject to } (W \cdot H) \geq 0 \tag{2}$$

This means that NNMF choses the decomposing matrices W and H such that the least squares error of the reconstruction is the smallest and the reconstructed matrix is nonnegative.

`sklearn`'s `NMF` is used to build the NNMF models. The initialization of the models is as follows:

```
nmf_faces = faces + np.abs(np.min(faces))
nmf_10 = NMF(n_components=10, solver="mu", max_iter=500)
W_10 = nmf_10.fit_transform(nmf_faces)
H_10 = nmf_10.components_
nmf_25 = NMF(n_components=25, solver="mu", max_iter=500)
W_25 = nmf_25.fit_transform(nmf_faces)
H_25 = nmf_25.components_
nmf_50 = NMF(n_components=50, solver="mu", max_iter=500)
W_50 = nmf_50.fit_transform(nmf_faces)
H_50 = nmf_50.components_
```

Input matrix given to the `NMF` initializer must be nonnegative, hence `np.abs(np.min(faces))` is added to all entries of the `faces` matrix before initialization.

The following figures display the MFs obtained by the `NMF` models.

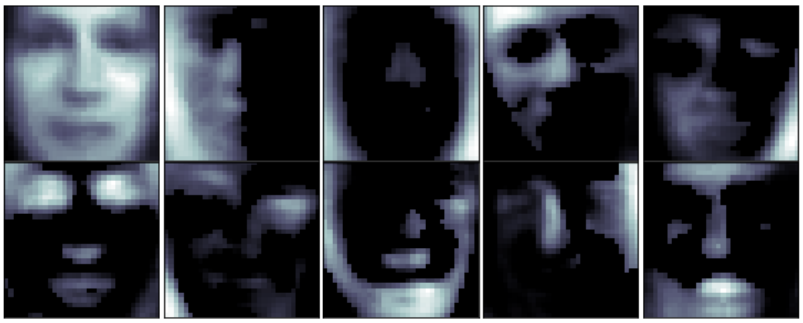

Figure 14: 10 MFs

Figure 15: 25 MFs

Figure 16: 50 MFs

The NNMF reconstruction can be easily obtained as follows:

```
faces_NMF_10 = W_10.dot(H_10) - np.abs(np.min(faces))
faces_NMF_25 = W_25.dot(H_25) - np.abs(np.min(faces))
faces_NMF_50 = W_50.dot(H_50) - np.abs(np.min(faces))
```

Note that the term `np.abs(np.min(faces))` must be subtracted to account for the transform we applied to `faces` during initialization of our models.

The figures that show the NNMF reconstructions follow starting from the next page.

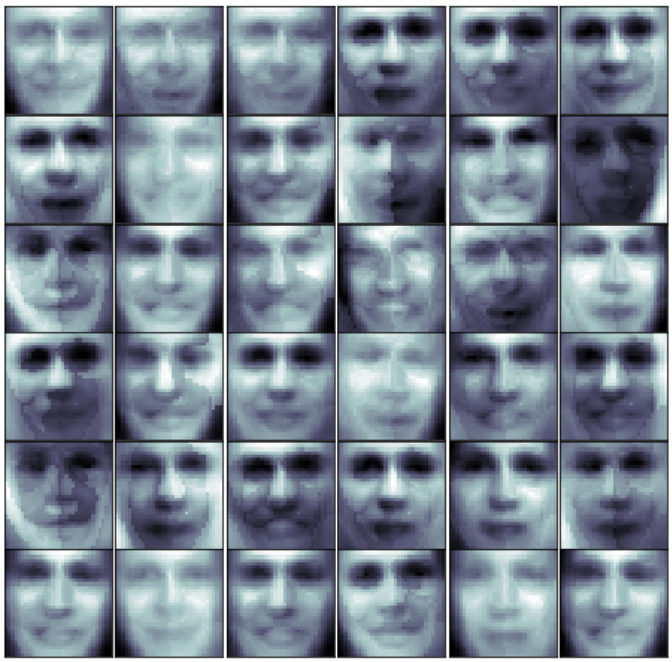

Figure 17: NNMF Reconstructions of the First 25 Stimuli (10 MF model is used)

Figure 18: NNMF Reconstructions of the First 25 Stimuli (25 MF model is used)

Figure 19: NNMF Reconstructions of the First 25 Stimuli (50 MF model is used)

Reconstruction losses for the models are as follows:

```
Reconstruction loss (10 MFs): mean of MSEs = 711.188931 , std of MSEs = 373.312780
Reconstruction loss (25 MFs): mean of MSEs = 547.534706 , std of MSEs = 275.813284
Reconstruction loss (50 MFs): mean of MSEs = 417.404509 , std of MSEs = 203.446847
```

We see from the output above that NNMF models perform the worse. However, NNMF provides better interpretability in most cases since it discovers some hidden latent features of the data. The reduced feature space of NNMF possibly includes features directly corresponding to the facial parts.

To sum up with, PCA is the most effective technique among the three dimensionality reduction techniques we considered. It is deterministic, fast and it gives the best reconstruction errors.

Question 2

The data given in this question contains BOLD responses evoked by two distinct auditory stimulus categories, human speech versus nature sounds, that are recorded across human auditory cortex. An array `stype` identifies the stimulus category as 1 (speech) or 2 (nature sound), and a matrix `vresp` represents the responses to each of 181 stimuli across 1626 voxels in auditory cortex.

2.a) In order to examine whether the two stimulus categories are discriminable based on the response patterns they elicit across auditory cortex, we compute the stimilarities of the response patterns across 181 stimuli.

`Python` equivalent of the `pdist2.m` function, `cdist` is used to compute the pairwise distances and build similarity matrices based on Euclidean, Cosine and Correlation distance metrics. The similarity matrices are computed as follows:

```
1 sim_euclidean = cdist(vresp, vresp, metric='euclidean')
2 sim_cosine = cdist(vresp, vresp, metric='cosine')
3 sim_correlation = cdist(vresp, vresp, metric='correlation')
```

The figures provided next displays the similarity matrices:

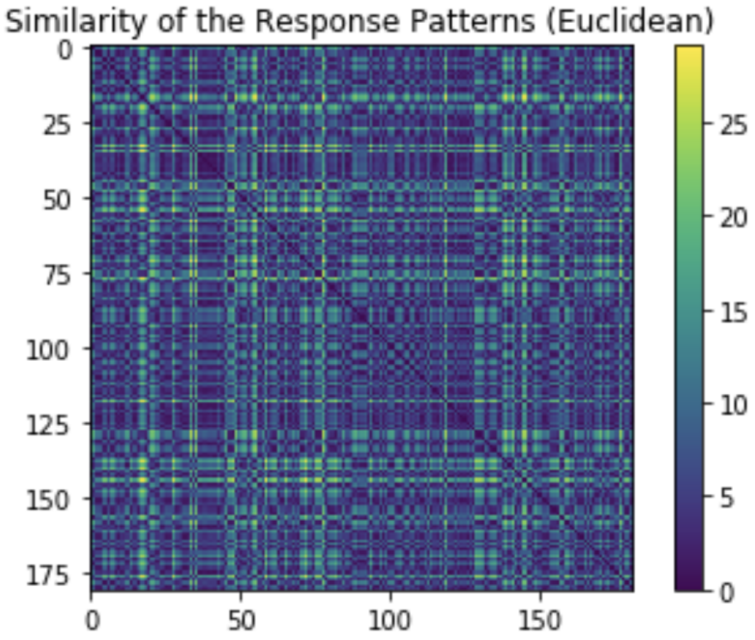

Figure 20: Similarity of the Response Patterns (Euclidean)

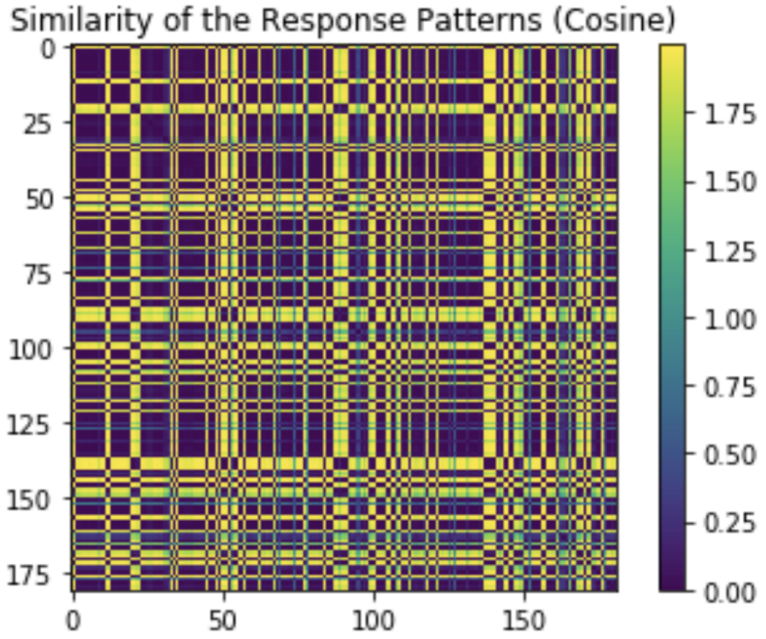

Figure 21: Similarity of the Response Patterns (Cosine)

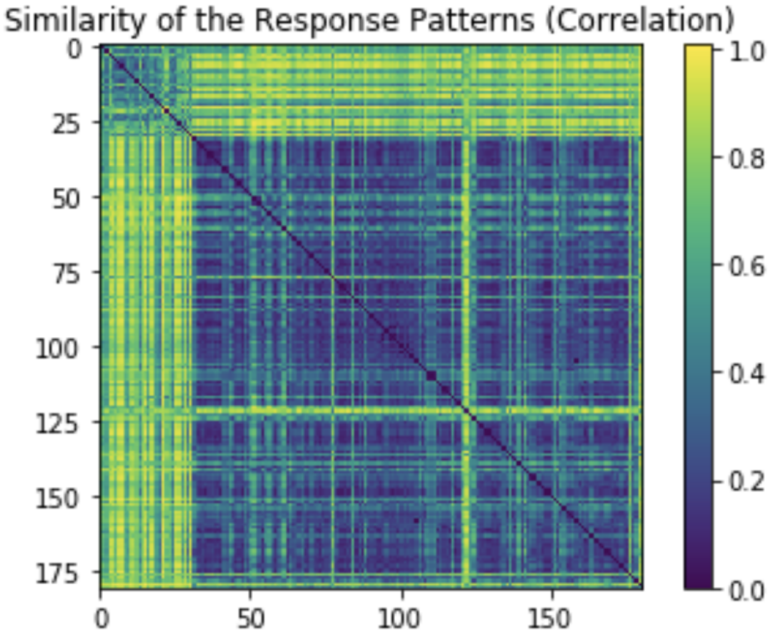

Figure 22: Similarity of the Response Patterns (Correlation)

By inspection on the figures showing the similarities, we can say that the Correlation metric provides the best separation. Euclidean and Cosine metrics both show homogeneous response patterns, where there is no clear separation, however Correlation metric perfectly separates the first 30 responses from the rest, since these responses are highly correlated. It can thus be concluded that the Correlation metric discriminates the responses best among the three metrics.

2.b) *Classical Multi-Dimensional Scaling (CMDS)* is a technique that allows us to represent data in a lower (usually 2) dimensional space while preserving the distances between the data points. Since there is no well documented `Python` equivalent of `Matlab`'s `cmdscale` function, a `Python` function that implements the `cmdscale` according to the algorithm discussed in the lecture slides is provided:

```python
def cmdscale(D):
    """
    Implementation of the classical multidimensional scaling (MDS) algorithm.
    Args:
        D: The symmetric matrix containing the distances between n
            objects in p dimensions
    Returns:
        X: Coordinates of n objects in the new space
    """
    N = D.shape[0]
    # Double centering procedure
    J = np.eye(N) - np.ones((N, N)) / N
    B = - J.dot(D ** 2).dot(J) / 2 # = X.X^T
    # Diagonalization
    evals, evecs = np.linalg.eigh(B)
    # Sort eigenpairs according to the descending order of eigenvalues
    idx = np.argsort(evals)[::-1]
    evals = evals[idx]
    evecs = evecs[:, idx]
    # Extract the positive eigenvalues
    pos_idx = np.where(evals > 0)[0]
    L = np.diag(np.sqrt(evals[pos_idx]))
    E = evecs[:, pos_idx]
    X = E.dot(L)
    return X
```

Thus, the following code is used to project the similarity matrices to the CMDS space:

```python
MDS_euclidean = cmdscale(sim_euclidean)
MDS_cosine = cmdscale(sim_cosine)
```

```
MDS_correlation = cmdscale(sim_correlation)
```

Since we only need 2 components in our new space (referred to as MDs) to visualize the data, we extract the first 2 MDs:

```
X_euclidean = MDS_euclidean[:, 0:2]
X_cosine = MDS_cosine[:, 0:2]
X_correlation = MDS_correlation[:, 0:2]
```

When visualizing the data, we also want to show the original labels; hence we compute the indices of each label and separate the data using these indices:

```
SPEECH_INDICES = np.where(stype == 1)[0]
NATURE_SOUND_INDICES = np.where(stype == 2)[0]
# Separate data with different labels
X_speech_euclidean = X_euclidean[SPEECH_INDICES]
X_nature_euclidean = X_euclidean[NATURE_SOUND_INDICES]
X_speech_cosine = X_cosine[SPEECH_INDICES]
X_nature_cosine = X_cosine[NATURE_SOUND_INDICES]
X_speech_correlation = X_correlation[SPEECH_INDICES]
X_nature_correlation = X_correlation[NATURE_SOUND_INDICES]
```

We now have the necessary data to obtain the visualizations. The following scatter plots show the responses for different stimuli categories in the 2-dimensional CMDS space:

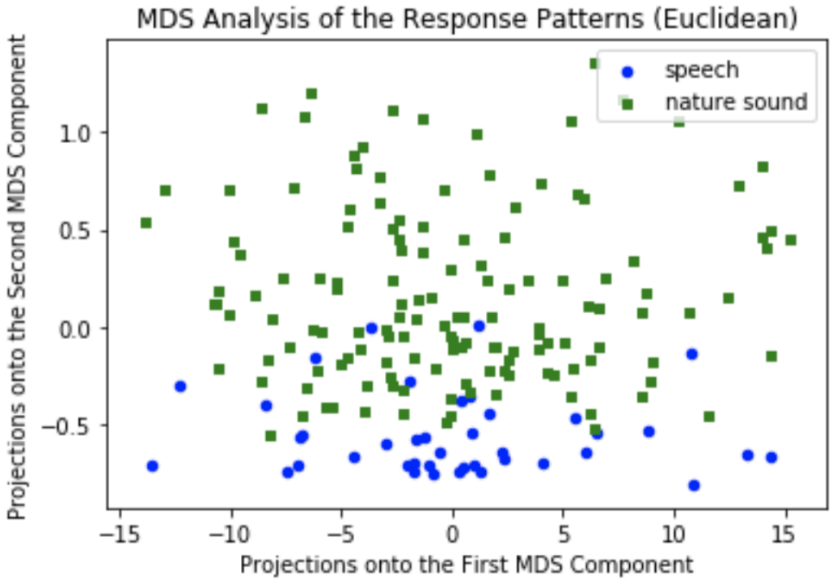

Figure 23: MDS Analysis of the Response Patterns (Euclidean)

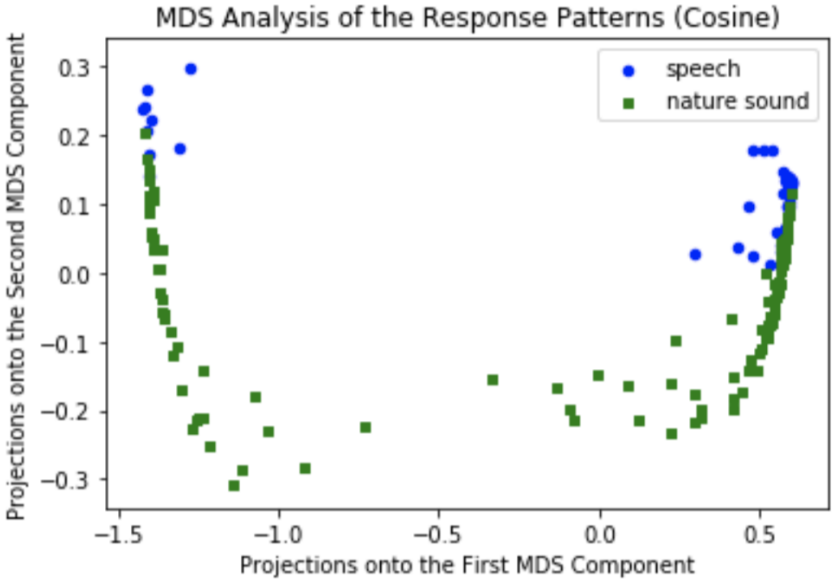

Figure 24: MDS Analysis of the Response Patterns (Cosine)

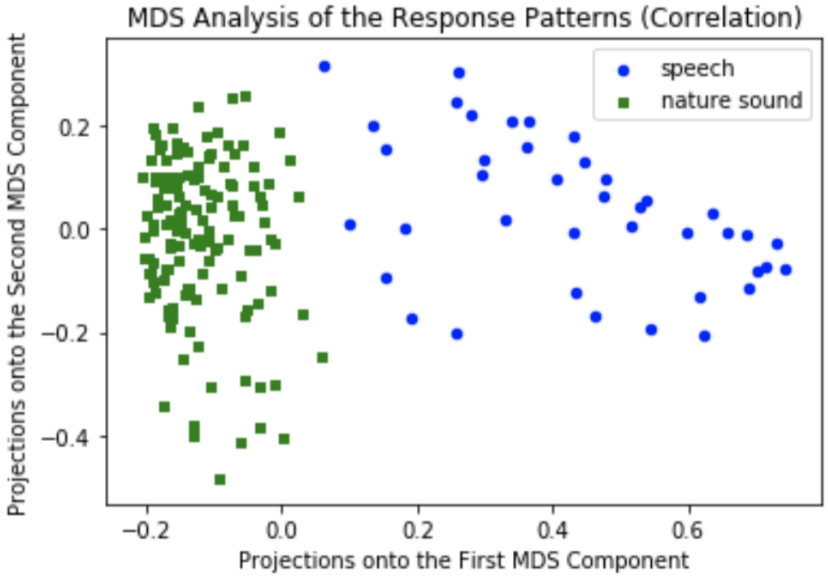

Figure 25: MDS Analysis of the Response Patterns (Correlation)

In Figure 10, we see that there is no clear separation between the responses from different categories. The *speech* responses tend to stay below the *nature sound* responses looking at the projections onto the second MD, however they still look like they are distributed over one big cluster.

In Figure 11, we clearly see that there are 2 clusters, but the problem is that the responses from different categories are both distributed over the two clusters. Thus, there is again no clear separation.

In Figure 12, there are 2 clusters and the *speech* responses are distributed over one cluster while the *nature sound* responses are distributed over the other. Hence, there exists a clear separation.

MDS analysis also showed that Correlation is the best distance metric for our data.

2.c) In this part, we try to make the cluster analysis we did by inspection more formal by using the *k-means* algorithm. An implementation of the k-means algorithm is provided. The first step in the algorithm is to initialize k cluster centers (centroids). The easiest way to do this is to select the centroids uniformly random:

```
def init_clusters(data, k, seed=7):
    """
    Initializes k cluster centers (means).
    Args:
        data: The input data
        k: Preferred number of clusters
        seed: Random seed for reproducibility (default is 7)
    Returns:
        centers: Array of k cluster centers
    """
    N = data.shape[0] # number of samples
    np.random.seed(seed)
    cluster_idx = np.random.choice(N, k)
    centers = data[cluster_idx]
    return centers
```

In each iteration of the k-means algorithm, we need to compute the distance between each data point and each centroid. Thus, we define the following method:

```
def build_dist_matrix(data, centers):
    """
    Builds a distance matrix containing the distance
    of each point to each cluster center.
    Args:
        data: The input data
```

```
        centers: Array of k cluster centers
    Returns:
        dist_matrix: The distance matrix, entry (i, j)
            represents the distance of the ith data point
            to the jth cluster center.
    """
    N = data.shape[0]
    K = centers.shape[0]
    dist_matrix = []
    for k in range(K):
        dist_matrix.append(np.sum((data - centers[k]) ** 2, axis=1))
    return np.array(dist_matrix).T
```

After the distances of the data points to the clusters are computed, k-means assigns each data point to the cluster for which the centroid is closest to the data point. Then the centroids are updated to be the center of the modified clusters. This updates can be performed by the following funtion:

```
def param_update_kmeans(data, old_centers):
    """
    Performs the parameter updates in a kmeans iteration.
    Args:
        data: The input data
        old_centers: Array of k cluster centers before the update
    Returns:
        losses: Loss computed for each data point
        assignments: Assignment vector indicating which data
            point belongs to which cluster
        centers: Array of k cluster centers after the update
    """
    N, F = data.shape
    K = old_centers.shape[1]
    dist_matrix = build_dist_matrix(data, old_centers)
    assignments = np.argmin(dist_matrix, axis=1)
    losses = []
    for i in range(N):
        assignment = assignments[i]
        point_cluster = old_centers[assignment]
        losses.append(np.sum((data[i] - point_cluster) ** 2))
    centers = []
    for k in range(K):
        cluster_idx = np.where(assignments == k)[0]
        centers.append(np.mean(data[cluster_idx], axis=0))
    return np.array(losses), assignments, np.array(centers)
```

Note that the function above also returns an array of losses containing the squared distance of each data point to the center of the cluster it is assigned to. We want to minimize the average of these losses so we run the parameter updates until the average loss converges somewhere. This procedure is implemented as follows:

```
def kmeans(data, k, max_iters=100, threshold=1e-5):
    """
    Implementation of the kmeans clustering algorithm.
    Args:
        data: The input data
        k: Preferred number of clusters
        max_iters: Maximum number of iterations to run the algorithm
            (default is 100)
        threshold: The threshold in the change in kmeans' loss metric, the
            algorithm converges when the threshold is reached (default is 1e-5)
    Returns:
        assignments: Assignment vector indicating which data
            point belongs to which cluster
        centers: Array of final k cluster centers
    """
    centers_old = init_clusters(data, k)
    avg_losses = []
    for i in range(max_iters):
        # Perform iteration updates
        losses, assignments, centers = param_update_kmeans(data, centers_old)
        avg_loss = np.mean(losses)
        avg_losses.append(avg_loss)
        # Check for convergence
        if i > 0 and np.abs(avg_losses[-1] - avg_losses[-2]) <= threshold:
            break
        # Prepare for the next iteration
        centers_old = centers
    return assignments, centers_old
```

Now that we have our `kmeans`, we run it on the similarity data projected onto the 2-dimensional CMDS space:

```
assignments_euclidean, centers_euclidean = kmeans(X_euclidean, 2)
assignments_cosine, centers_cosine = kmeans(X_cosine, 2)
assignments_correlation, centers_correlation = kmeans(X_correlation, 2)
```

We separate the data belonging to different clusters by running the code:

```
X_0_euclidean = X_euclidean[np.where(assignments_euclidean == 0)[0]]
X_1_euclidean = X_euclidean[np.where(assignments_euclidean == 1)[0]]
X_0_cosine = X_cosine[np.where(assignments_cosine == 0)[0]]
X_1_cosine = X_cosine[np.where(assignments_cosine == 1)[0]]
X_0_correlation = X_correlation[np.where(assignments_correlation == 0)[0]]
X_1_correlation = X_correlation[np.where(assignments_correlation == 1)[0]]
```

Using the obtained arrays, we show the same scatter plots in the previous part but this time we use different colors for the data points belonging to different clusters instead of the data points having different labels. The following figures display the results, and cluster centers are shown below each of them:

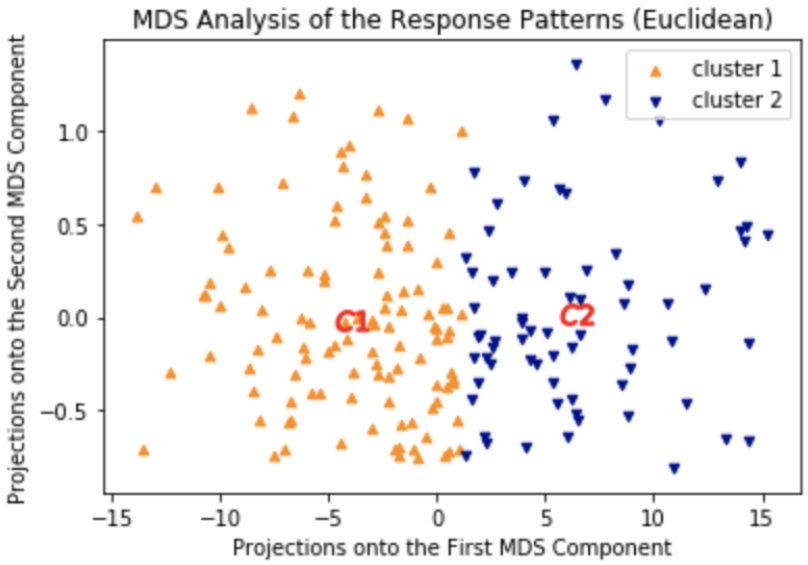

Figure 26: Population Response to the Stimulus $x = -1$

```
Cluster centers (Euclidean)
Center of cluster 1: (-3.967307, -0.011665)
Center of cluster 2: (6.439686, 0.018934)
```

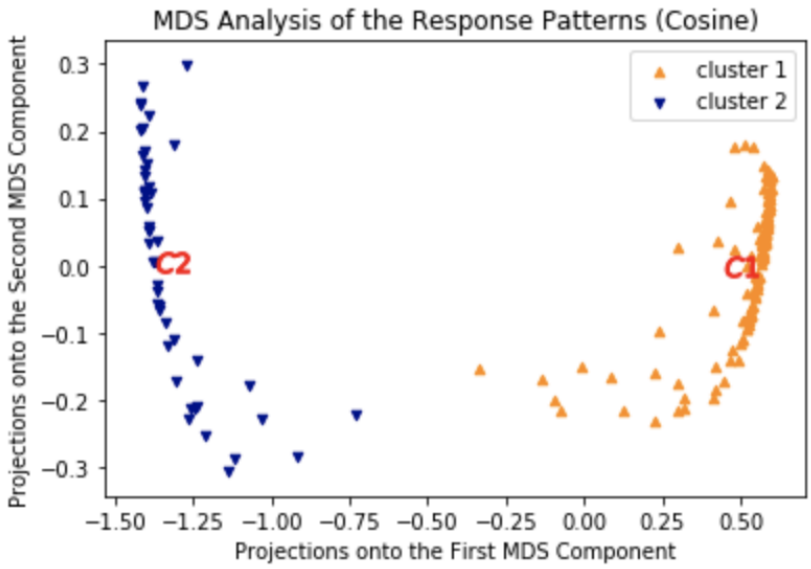

Figure 27: Population Response to the Stimulus $x = -1$

```
Cluster centers (Cosine)
Center of cluster 1: (0.503391, -0.002436)
Center of cluster 2: (-1.318885, 0.006383)
```

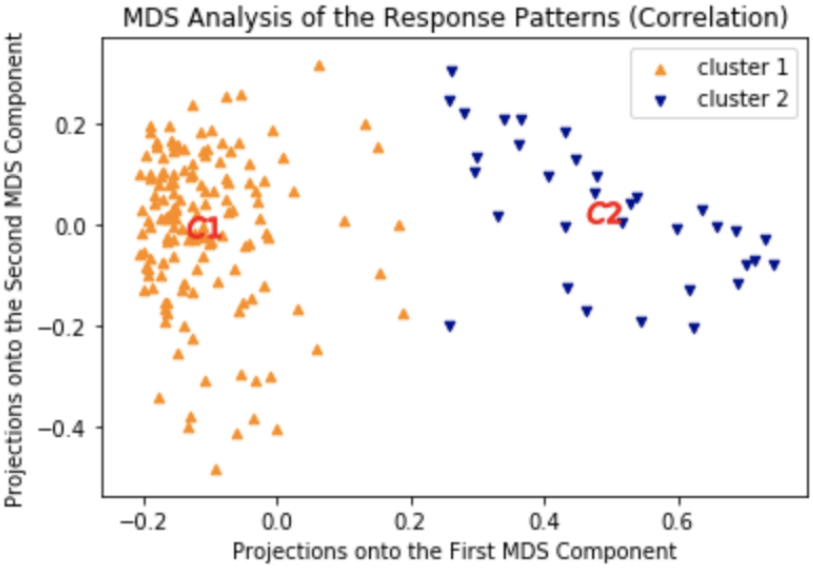

Figure 28: Population Response to the Stimulus $x = -1$

```
Cluster centers (Correlation)
Center of cluster 1: (-0.108990, -0.005833)
```

```
Center of cluster 2: (0.488804, 0.026160)
```

`kmeans` successfully found clusters for the similarity data. However, the clusters it found for the Euclidean and Cosine distance metrics are not consistent with the ground truth, meaning the clusters and the stimulus categories do not coincide. For Correlation, the clusters `kmeans` found and the actual stimulus categories are more or less the same. This implies once again that Correlation is the right metric to use.

Question 3

In this question, we perform stimulus reconstruction based on the response patterns collected from a neural population. The population consists of 21 neurons having Gaussian shaped tuning curves characterized by:

$$f_i(x) = A\dot{e}^{-(x-\mu_i)^2/(2\sigma_i^2)} \tag{3}$$

The tuning curves of the neurons have an amplitude (A) and standard deviation (σ) of 1. The centers of the curves are evenly spaced between -10 and 10, meaning μ_i is $-10 + i$ for the ith neuron.

3.a) The `Python` function below implements the tuning curve:

```
def tuning_curve(x, A, mu, sigma):
    """
    Gaussian shaped tuning function of a population of neurons.
    Args:
        x: The input stimulus
        A: Amplitude of the Gaussian-shaped tuning curve
        mu: Mean of the Gausssian-shaped tuning curve
        sigma: Standard deviation of the Gaussian-shaped tuning curve
    Returns:
        response: Resulting neural response
    """
    response = A * np.exp(-((x - mu) ** 2) / (2 * sigma ** 2))
    return response
```

The constants are initialized as explained above:

```
A = 1 # amplitude
SIGMA = 1 # standard deviation
NUM_NEURONS = 21
MU_VALS = np.arange(-10, 11)
STIMULI = np.linspace(-15, 16, 500)
```

The population activity is then calculated as follows:

```
activities = []
for mu in MU_VALS:
    activities.append(tuning_curve(STIMULI, A, mu, SIGMA))
```

Each array in the `activities` list represents the neural activity of a single neuron of the population. Thus, we iteratively plot the items of the `activities` list to visualize the population behavior. The resulting figure gives all tuning curves in the population:

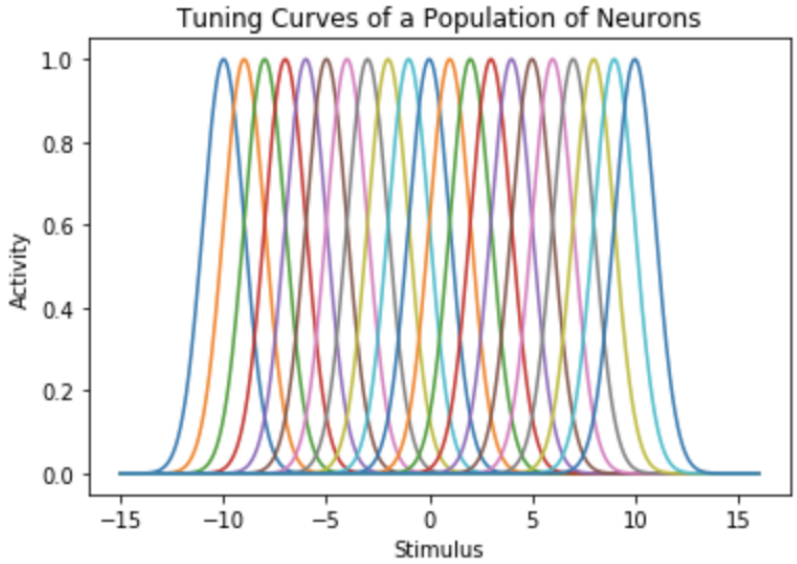

Figure 29: Tuning Curves of the Population

It can be seen that each neurons preferred stimulus value is the mean of its tuning curve. The following figure displays the population response to the stimulus $x = -1$:

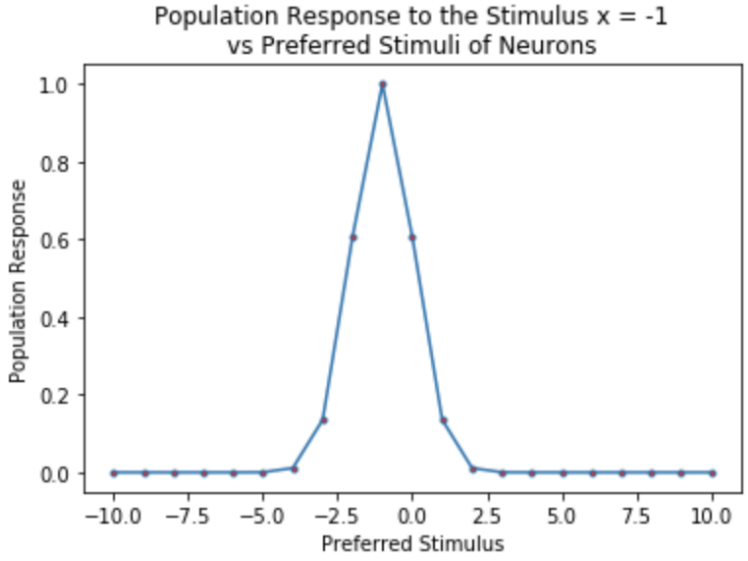

Figure 30: Population Response to the Stimulus $x = -1$

The previous figure is obtained by executing the line `responses = tuning_curve(X, A, MU_VALS, SIGMA)`, where `X = -1`. We clearly see that neurons with a preferred stimulus value closer to the input stimulus elicit a higher response. This observation plays a key role in *Winner Take All Decoders* which we discuss next.

3.b) In this part, we perform an experiment of 200 trials. In each trial, we sample an input stimulus from the interval $[-5, 5]$ and simulate the population response. The population response is modelled as the response generated from the tuning curve with an additional Gaussian noise with 1/20 standard deviation.

We implement a Winner Take All (WTA) decoder that estimates the input stimulus in each experiment trial. The idea behind the WTA decoder is to estimate the input stimulus as the preferred stimulus of the neuron in the population that elicited the highest response. This can be summarized with the equation:

$$\hat{x} = \mu_i : i = \underset{i}{argmax}(r_i) \tag{4}$$

In the equation above, $\hat{x}$ represents our estimation for the input stimulus and r_i represents the simulates response of the ith neuron.

The function below implements (4):

```
def winner_take_all_decoder(preferred_stimuli, response):
    """
    Given a population response and preferred stimuli of the
    neurons, estimates the actual stimulus as the preferred
    stimulus of the neuron exhibiting the highest response.
    Args:
        preferred_stimuli: The preferred stimuli of the neurons
        response: The response the population exhibits
    Returns:
        stimulus: the estimated input stimulus
    """
    highest_idx = np.argmax(response)
    stimulus = preferred_stimuli[highest_idx]
    return stimulus
```

Then, we perform the experiment by running the code below:

```
np.random.seed(17) # for reproducibility
responses = []
stimuli = []
estimated_stimuli_WTA = []
errors_WTA = []
```

```
for i in range(NUM_TRIALS):
    stimulus = np.random.choice(STIMULI_RANGE)
    response = tuning_curve(stimulus, A, MU_VALS, SIGMA)
    noise = np.random.normal(0, SIGMA / 20, NUM_NEURONS)
    response += noise
    estimated_stimulus_WTA = winner_take_all_decoder(MU_VALS, response)
    error_WTA = np.abs(stimulus - estimated_stimulus_WTA)
    responses.append(response)
    stimuli.append(stimulus)
    estimated_stimuli_WTA.append(estimated_stimulus_WTA)
    errors_WTA.append(error_WTA)
```

In the code above `NUM_TRIALS = 200` and `STIMULI_RANGE` is initialized to `np.linspace(-5, 5, 500)`. Input stimuli and response data are stored in arrays for further use. The stimuli estimations in `estimated_stimuli_WTA` are shown in the figure below together with the actual inputs:

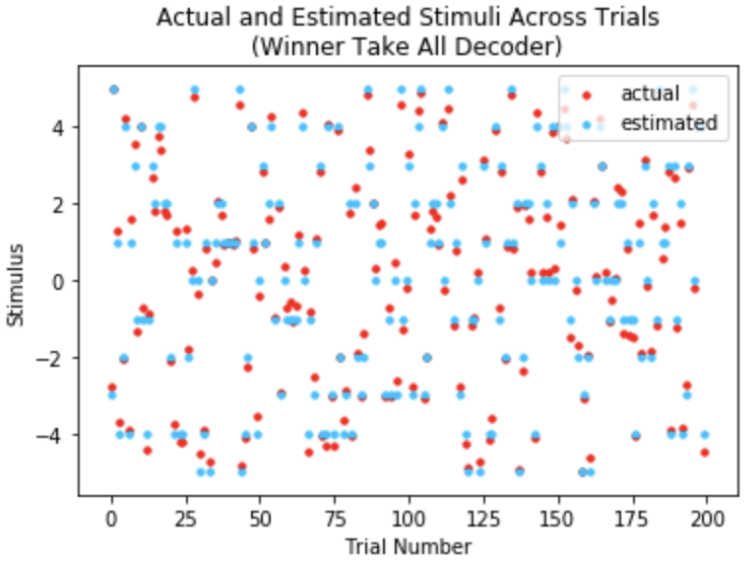

Figure 31: Actual and Estimated Stimuli Across Trials (WTA Decoder)

Absolute error is used to measure the performance of the decoder. In other words error in each trial is taken as the distance between the estimated and the actual stimuli. The code below computes, then prints mean and standard deviation of the errors made in the experiment:

```
mean_error_WTA = np.mean(errors_WTA)
std_error_WTA = np.std(errors_WTA)
print('Error Statistics for Winner Take All Decoder')
print('Mean of errors in stimuli estimation (absolute error is used):',
```

```
    mean_error_WTA)
print('Standard deviation of errors in stimuli estimation (absolute error is used)
    :', std_error_WTA)
```

The output of the code is as follows:

```
Error Statistics for Winner Take All Decoder
Mean of errors in stimuli estimation (absolute error is used): 0.22927855711422843
Standard deviation of errors in stimuli estimation (absolute error is used):
    0.1526404626792791
```

The mean absolute error (MAE) is fairly high in this case but this is not surprising since WTA Decoder is limited to a small set of values when choosing its estimations. For example in this experiment, WTA Decoder can only output preferred stimuli of the neurons that are between -5 and 5.

3.c) Now, we repeat the same procedure we followed in the previous part with the only being the decoder we use. WTA decoder is now changed to *Maximum Likelihood (ML) Decoder*. ML decoder considers the likelihood function of the input stimulus, meaning the probability that the input stimulus elicited the simulated response; and selects the stimulus value that maximizes this likelihood as its output. This can be expressed as:

$$\hat{x} = \underset{x}{argmax}(L(x)) = \underset{x}{argmax}(P(\mathbf{r}|x))) \tag{5}$$

In (5) $\mathbf{r}$ is the response vector containing the responses of all 21 neurons in the population. We proceed by deriving a simplified expression for $L(x)$:

$$\text{Given the response vector } \mathbf{r} = [r_1, r_2, ..., r_2 1]^T, \text{ where}$$

$$r_i(x) = f_i(x) + Normal(0, (\sigma/20)^2) \text{ for } i = 1, 2, ..., 21$$

$$\text{Then, } r_i(x) \sim Normal(f_i(x), (\sigma/20)^2)), \text{ therefore}$$

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

Using the independence of the neuron responses, we can write:

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

Multiplication of small numbers often introduces numeric problems so we consider the natural logarithm of $L(x)$ since maximization process is not affected by this simple transform:

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

We can further simplify $l(x)$ as shown below:

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

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

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

By convention, minimization is preferred over maximization. Hence, we consider the negative log-likelihood $-l(x)$, and arrive at:

$$\hat{x} = \underset{x}{argmin}(-l(x)) = \underset{x}{argmin}\Big(\sum_{i=1}^{21} (r_i(x) - f_i(x))^2\Big) \tag{6}$$

This result is an expected one since ML estimators relate to least squares cost when applied to Normal distributions.

The following `Python` function implements $-l(x)$:

```python
def nlogLL(x, response, A, mu_vals, sigma):
    """
    Given the input stimulus x, the response elicited by the neuron
    population and the tuning curve parameters; computes the negative
    of the log-likelihood of seeing the given population response.
    Disregards some constant terms for simplicity.
    Args:
        x: The input stimulus
        response: The response elicited by the neuron population
        A: Amplitude of the tuning curve
        mu_vals: Preferred stimulus value of each neuron in the population
        sigma: Standard deviation of the tuning curve
    Returns:
        nlogLL: The negative log-likelihood to see the given response
    """
    nlogLL = 0
    for r_i, mu_i in zip(response, mu_vals):
        nlogLL += (r_i - tuning_curve(x, A, mu_i, sigma)) ** 2
    return nlogLL
```

The function below implements the estimation procedure given by (6):

```
def MLE_decoder(response, A, mu_vals, sigma, stimuli_range):
    """
    Estimates the input stimulus to a neuron population by maximizing
    the likelihood to see the given population response.
    Args:
        response: The response elicited by the neuron population
        A: Amplitude of the tuning curve
        mu_vals: Preferred stimulus value of each neuron in the population
        sigma: Standard deviation of the tuning curve
        stimuli_range: Range of stimuli to consider
    Returns:
        est_stimulus: The estimated input stimulus
    """
    nlogLL_vals = []
    for stimulus in stimuli_range:
        nlogLL_vals.append(nlogLL(stimulus, response, A, mu_vals, sigma))
    min_idx = np.argmin(nlogLL_vals)
    est_stimulus = stimuli_range[min_idx]
    return est_stimulus
```

We can run the experiment once more using the ML decoder with the lines below:

```
estimated_stimuli_MLE = []
errors_MLE = []
for response, stimulus in zip(responses, stimuli):
    estimated_stimulus_MLE = MLE_decoder(response, A, MU_VALS, SIGMA,
        STIMULI_RANGE)
    error_MLE = np.abs(stimulus - estimated_stimulus_MLE)
    estimated_stimuli_MLE.append(float(estimated_stimulus_MLE))
    errors_MLE.append(float(error_MLE))
```

The figure located at the beginning of the next page shows a scatter plot of the actual stimuli input and the stimuli estimated by the ML decoder.

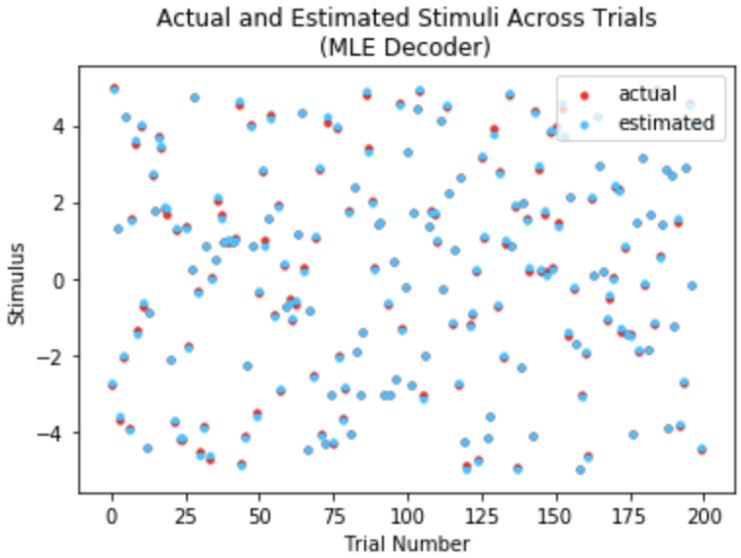

Figure 32: Actual and Estimated Stimuli Across Trials (ML Decoder)

We have the absolute errors made in each trial in the `errors_MLE` array, we compute the mean absolute error and the standard deviation of all errors. The results are provided below:

```
Error Statistics for MLE Decoder
Mean of errors in stimuli estimation (absolute error is used): 0.04228456913827656
Standard deviation of errors in stimuli estimation (absolute error is used):
    0.03423218782611939
```

It is obvious that ML decoder outperforms the WTA decoder. One reason for this is that ML decoder is not limited to a small set of output values as WTA decoder is. We have a very small MAE, which shows how powerful probabilistic decoders can be.

3.d) We change our decoder once more and use *Maximum-a-Posteriori (MAP) Decoder* in the same experiment setting. MAP decoder makes use of *Bayesian Estimation* to select the stimulus value that maximizes the posterior probability instead of the likelihood. This way MAP decoder can incorporate any prior information on the input stimulus to its estimation. Bayes rule tells us that:

$$P(x|\mathbf{r}) \propto P(\mathbf{r}|x)\dot{P}(x) = L(x)\dot{P}(x) \tag{7}$$

Thus, MAP decoder's estimation is given by:

$$\hat{x} = \underset{x}{argmax}(L(x)\dot{P}(x)) \tag{8}$$

where $P(x)$ is the prior of the stimulus value x.

Again, there is no difference between maximizing $P(x|\mathbf{r})$ and $log(P(x|\mathbf{r}))$:

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

It is given in the question that the prior on x follows a Gaussian with 0 mean and 2.5 standard deviation, so we can write:

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

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

We derived in the previous part that:

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

Now, we can combine these expressions, disregard the constant terms and write:

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

Once more, we choose minimization of $-log(P(x|\mathbf{r})$ for convenience. We arrive at:

$$\hat{x} = \underset{x}{argmin}(-log(P(x|\mathbf{r}))) = \underset{x}{argmin}\Big(\frac{1}{2(\sigma/20)^2} \sum_{i=1}^{21} \Big((r_i(x) - f_i(x))^2\Big) + \frac{x^2}{2(2.5)^2}\Big) \quad (9)$$

This expression is more or less the same with (6), the only difference is the additional regularization term due to the prior.

The function below implements $-log(P(x|\mathbf{r})$:

```
def nlogPosterior(x, response, A, mu_vals, sigma):
    """
    Given the input stimulus x, the response elicited by the neuron
    population and the tuning curve parameters; computes the negative
    of the log-posterior probability of seeing the input stimulus
    given population response. Disregards some constant terms for simplicity.
    Assumes that the prior follows a Gaussian with 0 mean and 2.5 standard
    deviation.
    Args:
        x: The input stimulus
        response: The response elicited by the neuron population
```

```
        A: Amplitude of the tuning curve
        mu_vals: Preferred stimulus value of each neuron in the population
        sigma: Standard deviation of the tuning curve
    Returns:
        nlogPosterior: The negative log-posterior to see the input stimulus
    """
    nlogPosterior = 0
    for r_i, mu_i in zip(response, mu_vals):
        nlogPosterior += ((r_i - tuning_curve(x, A, mu_i, sigma)) ** 2)
    nlogPosterior /= (2 * (sigma / 20) ** 2)
    nlogPosterior += (x ** 2) / (2 * 2.5 ** 2)
    return nlogPosterior
```

The following function implements the estimation procedure in (9):

```
def MAP_decoder(response, A, mu_vals, sigma, stimuli_range):
    """
    Estimates the input stimulus to a neuron population by maximizing
    the posterior probability to see the input stimulus given the
    population response.
    Args:
        response: The response elicited by the neuron population
        A: Amplitude of the tuning curve
        mu_vals: Preferred stimulus value of each neuron in the population
        sigma: Standard deviation of the tuning curve
        stimuli_range: Range of stimuli to consider
    Returns:
        est_stimulus: The estimated input stimulus
    """
    nlogPosterior_vals = []
    for stimulus in stimuli_range:
        nlogPosterior_vals.append(nlogPosterior(stimulus, response, A, mu_vals,
            sigma))
    min_idx = np.argmin(nlogPosterior_vals)
    est_stimulus = stimuli_range[min_idx]
    return est_stimulus
```

We re-run the experiment using the MAP decoder as follows:

```
estimated_stimuli_MAP = []
errors_MAP = []
for response, stimulus in zip(responses, stimuli):
    estimated_stimulus_MAP = MAP_decoder(response, A, MU_VALS, SIGMA,
        STIMULI_RANGE)
```

```
5       error_MAP = np.abs(stimulus - estimated_stimulus_MAP)
6       estimated_stimuli_MAP.append(float(estimated_stimulus_MAP))
7       errors_MAP.append(float(error_MAP))
```

The figure shown below displays the actual stimuli and the stimuli estimated by the MAP decoder:

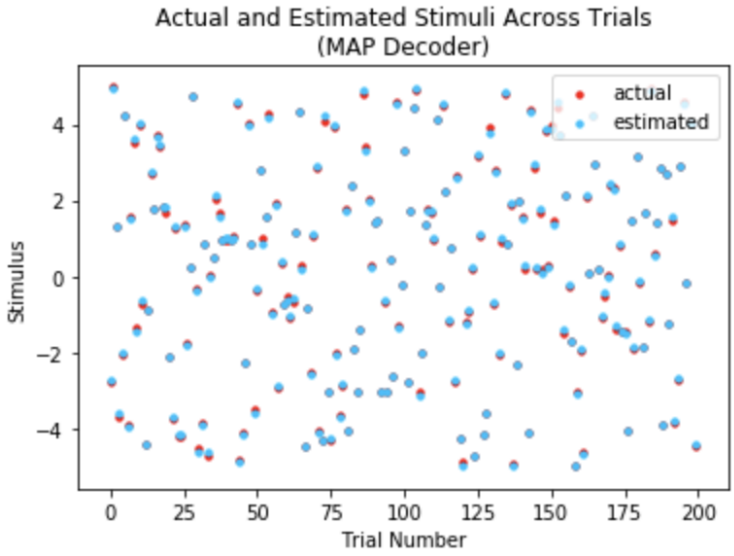

Figure 33: Actual and Estimated Stimuli Across Trials (MAP Decoder)

Similar to the previous parts, we compute the MAE and the standard deviation of the absolute errors. The resulting error statistics for MAP decoder are the following:

```
Error Statistics for MAP Decoder
Mean of errors in stimuli estimation (absolute error is used):
    0.041783567134268555
Standard deviation of errors in stimuli estimation (absolute error is used):
    0.03469756285083548
```

We can see that MAP decoder has a slightly smaller MAE than the ML decoder. However, standard deviation of errors is a little higher in the MAP decoder. Thus, we can say that the two performs more or less the same.

MAE is expected be smaller for MAP decoder, as it is the case here, since MAP decoder must theoretically be better because of the additional information it has about the prior (assuming that it has the correct information).

In conclusion, both of the probabilistic decoders perform very well but MAP is expected to be slightly better as we gain more prior knowledge about our stimuli data.

3.e) In the last part, we perform another experiment with 200 trials of stimulus intensity. The experiment is the same with part **3.b** except that we simulate the population response for different standard deviation σ_i values. We consider the following σ_i's:

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

We run the our experiment using the code below:

```
print('Takes a while...')
np.random.seed(7) # for reproducibility
errors_std_MLE = []
for i in range(NUM_TRIALS):
    stimulus = np.random.choice(STIMULI_RANGE)
    error_std_MLE = []
    for sigma in SIGMA_VALS:
        response_std = tuning_curve(stimulus, A, MU_VALS, sigma)
        noise = np.random.normal(0, 1 / 20, NUM_NEURONS)
        response_std += noise
        estimated_stimulus_std_MLE = MLE_decoder(response_std, A, MU_VALS, sigma,
            STIMULI_RANGE)
        error_std_MLE.append(np.abs(stimulus - float(estimated_stimulus_std_MLE)))
    errors_std_MLE.append(np.array(error_std_MLE))
errors_std_MLE = np.array(errors_std_MLE)
```

We wonder whether it is better to have narrower or wider tuning curves, hence we measure and record the absolute error made in each trial and each sigma value in the code above. We must obviously use a fixed decoder and ML decoder is used above. The resulting error statistics can be printed by running the lines below:

```
mean_errors_std_MLE = []
std_errors_std_MLE = []
for i, sigma in enumerate(SIGMA_VALS):
    mean_error_std_MLE = np.mean(errors_std_MLE[:, i])
    std_error_std_MLE = np.std(errors_std_MLE[:, i])
    print('\nError Statistics for MLE Decoder (sigma = %.1f for the tuning curve)'
        % sigma)
    print('Mean of errors in stimuli estimation (absolute error is used):',
        mean_error_std_MLE)
    print('Standard deviation of errors in stimuli estimation (absolute error is
        used):', std_error_std_MLE)
    mean_errors_std_MLE.append(mean_error_std_MLE)
    std_errors_std_MLE.append(std_error_std_MLE)
```

The code prints:

```
Error Statistics for MLE Decoder (sigma = 0.1 for the tuning curve)
Mean of errors in stimuli estimation (absolute error is used): 1.6338677354709419
Standard deviation of errors in stimuli estimation (absolute error is used):
   2.4175380499447843

Error Statistics for MLE Decoder (sigma = 0.2 for the tuning curve)
Mean of errors in stimuli estimation (absolute error is used): 0.4335671342685371
Standard deviation of errors in stimuli estimation (absolute error is used):
   1.1435024578429585

Error Statistics for MLE Decoder (sigma = 0.5 for the tuning curve)
Mean of errors in stimuli estimation (absolute error is used):
   0.030861723446893797
Standard deviation of errors in stimuli estimation (absolute error is used):
   0.031100241658623403

Error Statistics for MLE Decoder (sigma = 1.0 for the tuning curve)
Mean of errors in stimuli estimation (absolute error is used): 0.04108216432865732
Standard deviation of errors in stimuli estimation (absolute error is used):
   0.02860541567859759

Error Statistics for MLE Decoder (sigma = 2.0 for the tuning curve)
Mean of errors in stimuli estimation (absolute error is used): 0.06182364729458916
Standard deviation of errors in stimuli estimation (absolute error is used):
   0.0519025502590642

Error Statistics for MLE Decoder (sigma = 5.0 for the tuning curve)
Mean of errors in stimuli estimation (absolute error is used): 0.09859719438877754
Standard deviation of errors in stimuli estimation (absolute error is used):
   0.0730669127571741
```

It is hard to interpret the results looking at the huge printout, hence a plot that summarizes the printout is provided at the beginning of the next page.

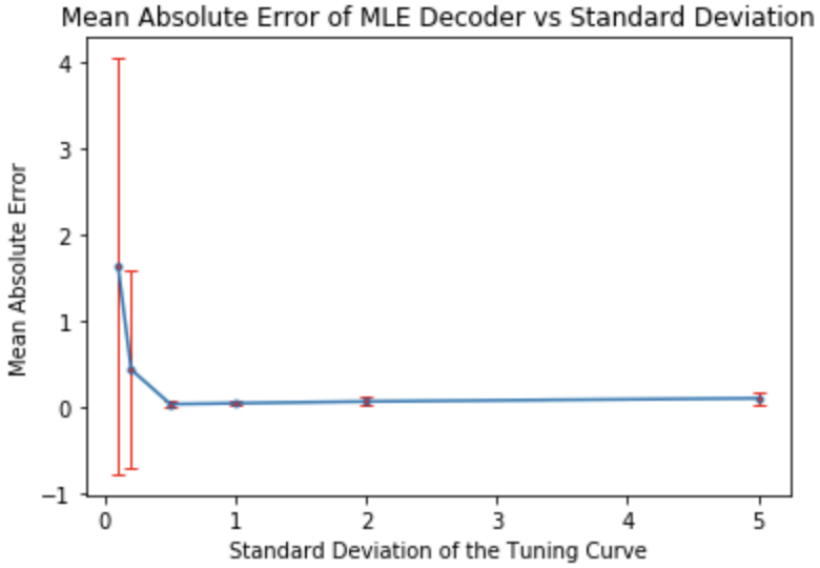

Figure 34: MAE of ML decoder vs Standard Deviation of the Tuning Curve

Looking at the plot we can see that wider tuning curves give better results since they are less affected by noise. Narrow tuning curves on the other hand are more sensitive to noise. As the tuning curves become narrower, the noise becomes more and more significant.

The printout tells us that the optimal σ_i value from the values we considered is 0.5. This implies that making the tuning curves wider also has a negative effect on the decoder performance.

We can conclude that there exists an optimal standard deviation for the tuning curves that results in the best decoder performance and having σ_i's away from the optimal reduces the performance; however wider tuning curves tend to be more robust in general.

Question 4

This question asks us to build a classifier that can distinguish response patterns elicited by *face* and *building* stimuli that have been recorded across 1626 voxels in ventral-temporal cortex. The `stype` array contains the category of each stimulus 1 (*face*) or 2 (*building*), and the matrix `vresop` represents the responses to 181 stimuli across 1626 voxels.

4.a) The classifier we implement will rely on *Linear Discriminant Analysis (LDA)*. A LDA classifier tries to build a linear decision boundary on the input feature space that maximizes the separability among known classes. When the linear decision boundary is being established, the LDA classifier tries to maximize the distance between the means of the classes and minimize the variation within each class.

In our case the data we want to classify is first reduced to a 2-dimensional MDS space using the `cmdscale` function implemented in the second question. Euclidean distance is used to compute the similarity matrix for the response patterns, and the resulting matrix is given as an input to the `cmdscale` function as before.

The theory behind the LDA classifier is to compute the probability of a data point belonging to a Gaussian given the class labels. LDA then assigns a data point to the class that maximizes this probability. Formally:

$$P(Y = k|X = x) \propto P(X = x|Y = k) \cdot P(y) = f_k(x) \cdot \pi_k \tag{10}$$

where $f_k(x)$ is the multivariate Gaussian with mean, mu_k and covariance Σ. mu_k represents the mean of the data points belonging to the kth class, Σ is the common covariance matrix of the entire dataset and π_k is the prior on class k.

When we plug in the multivariate Gaussian density to (10), take the logorithm of the expression and get rid of some constant terms, we arrive at:

$$\delta_k(x) = log\pi_k - \frac{1}{2}\mu_k^T\Sigma\mu_k + x^T\Sigma\mu_k \tag{11}$$

$\delta_k(x)$ is our objective which we will try to maximize because it is indeed a simplified expression for the probability of x belonging to the class k. Thus, a data point x will get assigned to the class k that maximizes $\delta_k(x)$.

For the case of binary LDA classification, the decision boundary is:

$$\delta_0(x) = \delta_1(x) \tag{12}$$

It turns out that (12) defines a linear decision boundary as claimed before.

LDA is implemented from scratch according to the (11) and (12). Note that the LDA functions in sophisticated libraries use complicated implementation methods that yield better results, so the performance of our classifier is a little worse than them.

A function to compute the class means, μ_k's, is given below:

```
def compute_class_means(data, labels, num_classes):
    """
    Computes the mean of each class.
    Args:
        data: The input data
        labels: The given classes of each data point
        num_classes: Number of classes
    Returns:
        means: Means of the classes
    """
    means = []
    for i in range(num_classes):
        class_idx = np.where(labels == i)[0]
        means.append(np.mean(data[class_idx], axis=0))
    return np.array(means)
```

A function to compute an unbiased estimate for the common covariance matrix follows:

```
def compute_cov(data, labels, class_means):
    """
    Computes an unbiased estimator for the common covariance matrix.
    Args
        data: The input data
        labels: The given classes of each data point
        class_means: Means of the classes
    Returns:
        cov: Unbiased estimator for the common covariance matrix
    """
    K = np.size(class_means) # number of classes
    N, F = data.shape
    cov = np.zeros((F, F))
    for k in range(K):
        for x_i in data[np.where(labels == k)[0]]:
            cov += x_i.reshape(-1, 1).dot(x_i.reshape(-1, 1).T)
    cov /= (N - K)
    return cov
```

Now that we have the functions necessary to compute μ_k's and Σ, we can provide the function that computes $\delta_k(x)$ values. The function is given at the beginning of the next page.

```python
def LDA_predictor(x, cov, mean_k, k, labels):
    """
    Computes the class discriminant.
    Args:
        x: The input data point
        cov: Covariance matrix of the data
        mean_k: Mean of the input class
        k: The input class
        labels: The input data labels
    Returns:
        delta: Class discriminant
    """
    # Compute the prior probability of class k
    pi_k = np.size(labels[labels == k]) /np.size(labels)
    if np.size(x) == 1:
        cov_inv = np.array([[1 / cov]])
    else:
        cov_inv = np.linalg.inv(cov)
    delta = np.log(pi_k) - mean_k.T.dot(cov_inv).dot(mean_k) / 2
    delta += x.T.dot(cov_inv).dot(mean_k)
    return delta
```

We can compute the points that define the linear decision boundary numerically by solving the equation $\delta_0(x) = \delta_1(x)$. The `bisect` method of `scipy`'s `optimize` module can be helpful when solving for (12). The function that returns the linear decision boundary follows:

```python
def LDA_binary_boundary(data, labels):
    """
    Computes a LDA-based binary classification boundary.
    Args:
        data: The input data
        labels: The class of each data point
    Returns:
        x_vals: x coordinate values of the linear boundary
        y_vals: y coordinate values of the linear boundary
    """
    means = compute_class_means(data, labels, 2)
    cov = compute_cov(data, labels, means)
    min_x, max_x = np.min(data[:, 0]), np.max(data[:, 0])
    min_y, max_y = np.min(data[:, 1]), np.max(data[:, 1])
    x_vals = np.linspace(min_x, max_x, 500)
    y_vals = []
    for x in x_vals:
```

```
        y_vals.append(bisect(
            lambda y: LDA_predictor(np.array([x, y]), cov, means[0], 0, labels) -
            LDA_predictor(np.array([x, y]), cov, means[1], 1, labels),
            min_y - np.sign(min_y) * 10 * min_y, max_y + np.sign(max_y) * 10 *
                max_y))
    return x_vals, np.array(y_vals)
```

Finally, the classification function that predicts the class k for data point x, which maximizes $\delta_k(x)$ is given below:

```
def LDA_classifier(x, train, labels, num_classes=2):
    """
    Computes class prediction for an input using LDA-based binary classification.
    Args:
        x: The input data point
        train: The training data
        labels: The class of each training data point
        num_classes: Number of classes in the prediction process (default is 2)
    Returns:
        class_: Class predicted for input based on binary LDA classifier
    """
    means = compute_class_means(train, labels, num_classes)
    cov = compute_cov(train, labels, means)
    deltas = []
    for k in range(num_classes):
        deltas.append(LDA_predictor(x, cov, means[k], k, labels))
    class_ = np.argmax(np.array(deltas))
    return class_
```

A method that performs leave one out cross validation is also provided below to be able to measure model performance:

```
def CV_leave_one_out(data, labels):
    """
    Performs leave one out cross validation and returns the count
    of correct classifications.
    Args:
        data: The input data
        labels: The original labels of the data points
    Returns:
        correct_count: The number of correct classifications
    """
    correct_count = 0
    for i, test in enumerate(data):
```

```
        idx = np.arange(data.shape[0]) != i
        train = data[idx]
        train_labels = labels[idx]
        class_of_test = LDA_classifier(test, train, train_labels)
        correct_count += (class_of_test == labels[i])
    return correct_count
```

We can now compute the linear decision boundary for the 2-dimensional Euclidean MDS representation of our data using the line below:

```
x_vals_euclidean, y_vals_euclidean = LDA_binary_boundary(X_euclidean, stype - 1)
```

The decision boundary together with the data points colored with respect to their labels are shown in the figure below:

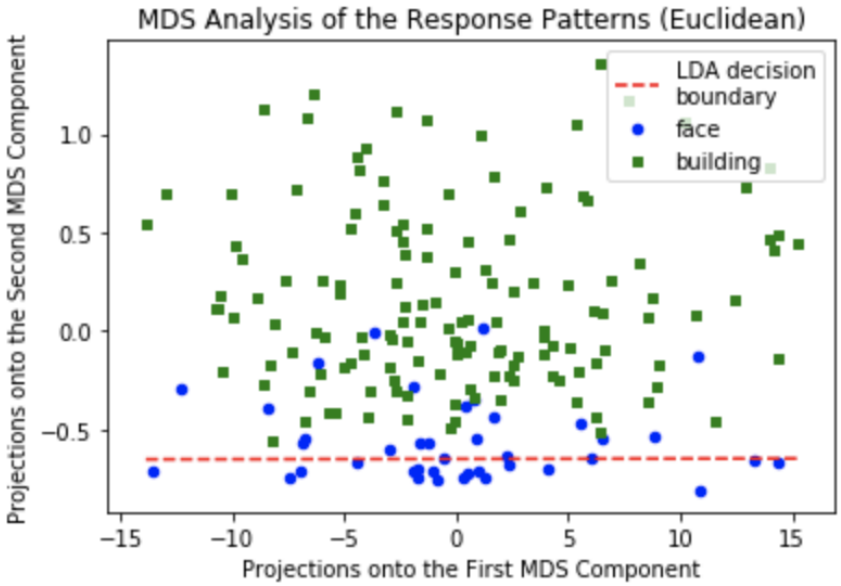

Figure 35: LDA Decision Boundary for the Data in the Euclidean MDS space

The classification accuracy is computed by dividing the correct classification count returned by `CV_leave_one_out` to the total number of data points. The result is displayed below:

```
Leave one out cross validation results (euclidean):
157 class labels predicted correctly out of 181
Prediction accuracy: 86.74%
```

The classifier performs fairly well as the points below the decision boundary are all responses to *face* stimuli, whereas points above are responses to *building* stimuli except a few outliers.

Note that the results would be better if a library function was used.

4.b) In this part, we use Correlation distance metric to build our similarity matrix and then project the matrix to the 2-dimensional MDS space. After this, we repeat the exact same procedure we carried out in the previous part. The data points together with their labels and the linear decision boundary produced by the LDA classifier are displayed in the figure below:

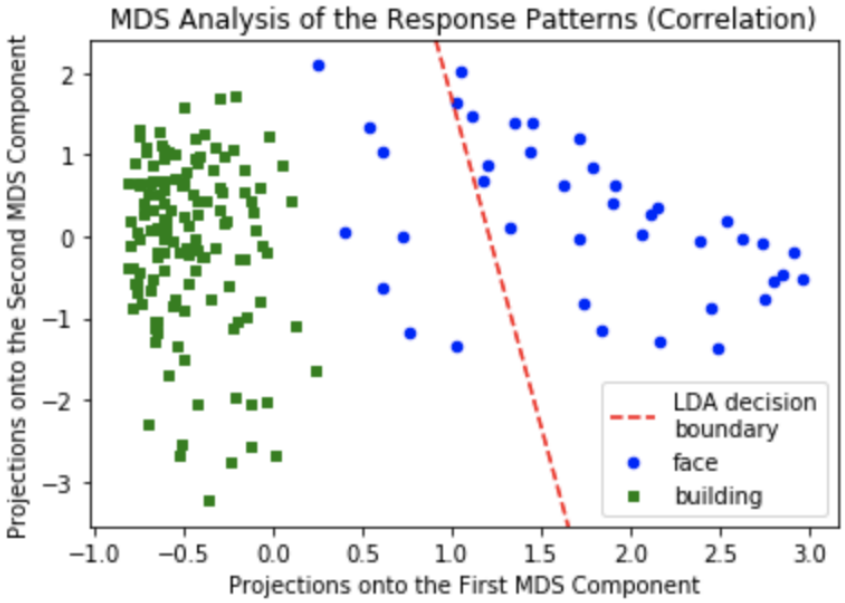

Figure 36: LDA Decision Boundary for the Data in the Correlation MDS space

The classification accuracy when Correlation is used as the distance metric is given below:

```
Leave one out cross validation results (correlation):
172 class labels predicted correctly out of 181
Prediction accuracy: 95.03%
```

We see that classification accuracy increased quite a lot when we changed the distance metric from Euclidean to Correlation. The points on the right-hand side of the decision boundary are all responses to *face* stimuli and there are much fewer outliers on the other side. This implies that Correlation is a more effective distance metric for this particular dataset.

4.c) We wonder the effect of the dimension of the MDS projection on the classification performance. Hence, we run the LDA classification in the 1, 2, 3, 4 and 5 dimensional MDS spaces. We keep using Correlation as our distance metric. We execute `CV_leave_one_out` for each dimension number to measure classifier performance. The code below performs the experiment:

```
accuracies = []
for i in range(1, 6):
    X_corr = MDS_correlation[:, 0:i]
    correct_count_correlation = CV_leave_one_out(X_corr, stype - 1)
    print('Leave one out cross validation results on %d dimensional MDS
        representation (correlation):' % i)
    print('%d class labels predicted correctly out of %d' % (
        correct_count_correlation, N))
    prediction_accuracy = 100 * correct_count_correlation / N
    print('Prediction accuracy: %.2f%%\n' % prediction_accuracy)
    accuracies.append(prediction_accuracy)
accuracies = np.array(accuracies).flatten()
```

The code produces the following output:

```
Leave one out cross validation results on 1 dimensional MDS representation (
    correlation):
168 class labels predicted correctly out of 181
Prediction accuracy: 92.82%

Leave one out cross validation results on 2 dimensional MDS representation (
    correlation):
172 class labels predicted correctly out of 181
Prediction accuracy: 95.03%

Leave one out cross validation results on 3 dimensional MDS representation (
    correlation):
170 class labels predicted correctly out of 181
Prediction accuracy: 93.92%

Leave one out cross validation results on 4 dimensional MDS representation (
    correlation):
170 class labels predicted correctly out of 181
Prediction accuracy: 93.92%

Leave one out cross validation results on 5 dimensional MDS representation (
    correlation):
170 class labels predicted correctly out of 181
```

```
Prediction accuracy: 93.92%
```

An histogram showing the classification accuracies is shown in the figure below:

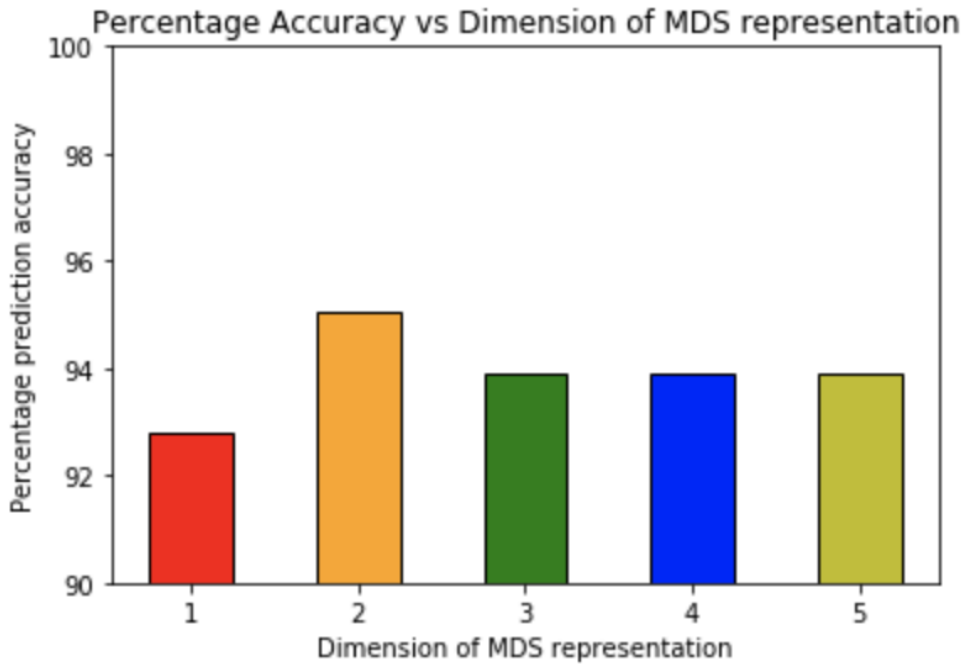

Figure 37: Prediction Accuracy vs MDS Space Dimension

We see from the figure above that 1 dimensional MDS space provides the worse classification performance and 2 dimensional MDS space provides the best. Intiutively, we expect the prediction performance to increase as the MDS space dimension increases. However, this is not the case here since increased number of dimensions result in inaccuracies in the numerical calculations. Looking at the figure we can infer that 2 dimensions are sufficient. However, the sufficient number of MDS dimensions may depend on the use of the dataset; since higher dimensions may provide better interpretability.