# EEE482 - Computational Neuroscience

## Homework Assignment - 3

Berkan Ozdamar
21602353

## Question 1

In this question, we are given Blood-Oxygen Level Dependent (BOLD) responses of neurals in visual cortex. Data contains of Xn with size of (1000,100) and Yn with size (1000,1). Xn represents 1000 samples with 100 features and Yn represents the response of those 1000 samples.

[1]:
```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
import hdf5storage
import h5py
```

[2]:
```
with h5py.File('hw3_data2.mat', 'r') as file:
    Xn = list(file['Xn'])
with h5py.File('hw3_data2.mat', 'r') as file:
    Yn = list(file['Yn'])

Xn = np.array(Xn)
Yn = np.array(Yn)
Xn = Xn.T
Yn = Yn.flatten()

print(np.shape(Xn))
print(np.shape(Yn))
```

```
(1000, 100)
(1000,)
```

### Part a

We are asked to use Ridge Regression to model BOLD data. Ridge regression is actually an Ordinary Least Squared(OLS) method with regularization term which reduces overfitting.

$$L(w) := \frac{1}{2}\sum_{n=1}^{N}(x_n^T \cdot w - y_n)^2 + \frac{\lambda}{2}\sum_{k=1}^{K} w_k^2$$

The $\frac{\lambda}{2}||w||^2$ term is the regularization term that makes OLS method to Ridge Regression. To find the optimal weights in the Ridge Regression, we take the gradient of $L(w)$ and set it to zero. By doing this, we will find the optimal weight values which is:

$$\nabla L(w^\star) = 0$$

$$X^T \cdot (X \cdot w^\star - y) + \lambda w^\star = 0$$

$$X^T X \cdot w^\star - X^T \cdot y + \lambda w^\star = 0$$

$$(X^T X + \lambda I_K) \cdot w^\star = X^T \cdot y$$

$$w^\star = (X^T X + \lambda I_K)^{-1} X^T \cdot y$$

$w^*$ is $(X^TX + \lambda I)^{-1}X^Ty$ in ridge regression rather than $(X^TX)^{-1}X^Ty$ in OLS.

Python code for finding optimal weight in ridge regression is given below:

```
[3]: #Part A

     def ridge_regression(X, Y, lambdaa):
         weight = np.linalg.inv(X.T.dot(X) + lambdaa*(np.identity(np.shape(X)[1])).
      ↪dot(X.T).dot(Y)
         return weight
```

For measuring performance, we will use R2 on the test data. R2 will be computed from the square of the pearson correlation between predictions and outputs. Python code for R2 is given below:

```
[4]: def R2(Xtrain, Ytrain, Xtest, Ytest, lambdaa):
         weight = ridge_regression(Xtrain, Ytrain, lambdaa)
         prediction = Xtest.dot(weight)
         R2 = (np.corrcoef(Ytest, prediction)[0, 1]) ** 2
         return R2
```

And we are asked to use 10-Fold Cross Validation for training our data. Cross Validation is iteratively trains and then test the model for k times, in our case 10 times. It allows us to get better model estimations due to increased use of training and test data. Python code for the k-fold Cross Validation is given below:

```
[5]: def k_fold_cross_validation(X, Y, lambdaaK, foldNum):
         size = np.shape(X)[0]
         index  = int(size / foldNum)
         validX = np.zeros((100,np.shape(X)[1]))
         testX  = np.zeros((100,np.shape(X)[1]))
         trainX = np.zeros((800,np.shape(X)[1]))
         validY = np.zeros(100)
         testY  = np.zeros(100)
         trainY = np.zeros(800)

         r2_valid = []
         r2_test = []
         for i in range(foldNum):
             validStartindex = (i*index) % size
             validEndindex   = (i+1)*index % size
             testStartindex  = (i+1)*index % size
             testEndindex    = (i+2)*index % size
             trainStartindex = (i+2)*index % size
             trainEndindex   = (i+10)*index % size

             if (validEndindex == 0):
                 validEndindex = 1000
             elif(testEndindex == 0):
                 testEndindex = 1000

             validX = X[validStartindex : validEndindex][:]
             validY = Y[validStartindex : validEndindex]
             testX  = X[testStartindex  : testEndindex][:]
             testY  = Y[testStartindex  : testEndindex]

             if (trainStartindex >= trainEndindex):
                 trainX[0: size-trainStartindex-1 ][:]      = X[trainStartindex : -1][:]
                 trainX[size - trainStartindex-1  : -1][:] = X[0: trainEndindex][:]
                 trainY[0: size-trainStartindex -1]         = Y[trainStartindex : -1]
                 trainY[size - trainStartindex-1  : -1]     = Y[0: trainEndindex]
             else:
                 trainX = X[trainStartindex % size : trainEndindex % size][:]
                 trainY = Y[trainStartindex % size : trainEndindex % size]
```

```
        for lambdaa in lambdaaK:
            r2_valid.append(R2(trainX, trainY, validX, validY, lambdaa))
            r2_test.append(R2(trainX, trainY, testX, testY, lambdaa))

    return r2_valid, r2_test
```

We are going to find the $\lambda_{opt}$ for the given lambda values in range of $[0 - 10^{12}]$. We will run 10-fold Cross Validation on those lambda values and find the $\lambda_{opt}$ with R2 measurement.

[6]:
```
K_FOLD = 10
lambda_arr = np.logspace(0, 12, num=500, base=10)

r2_valid, r2_test = k_fold_cross_validation(Xn, Yn, lambda_arr, K_FOLD)
```

[7]:
```
# Get the mean of every 500th item in r2_valid and r2_test and store them in␣
 ↪r2_valid_final and r2_test_final,
# so we get the mean r2 valid and test values.

r2_valid_final = np.zeros(500)
r2_test_final = np.zeros(500)

temp = 0
for i in range(500):
    for j in range(K_FOLD):
        temp += r2_valid[i + j*500]
    r2_valid_final[i] = np.mean(temp)
    temp = 0

temp = 0
for i in range(500):
    for j in range(K_FOLD):
        temp += r2_test[i + j*500]
    r2_test_final[i] = np.mean(temp)
    temp = 0
```

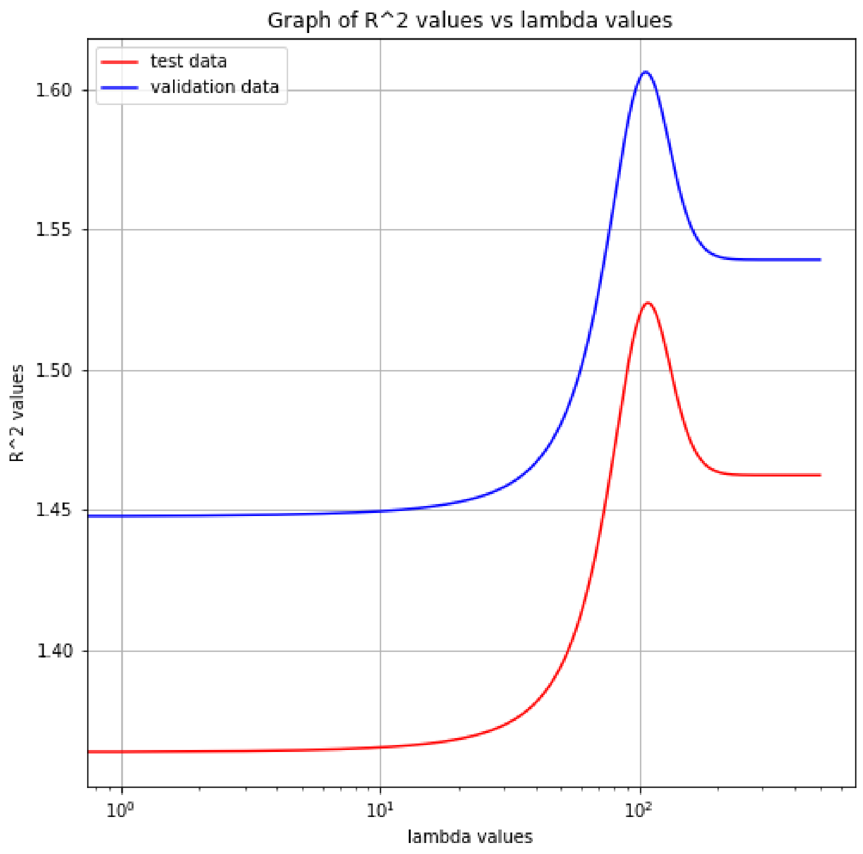
[8]:
```
# The optimal lambda value which makes R2 max for validation data
max_r2_valid = max(r2_valid_final)
for i in range(len(r2_valid_final)):
    if (r2_valid_final[i] == max_r2_valid):
        optimal_lambda_valid_index = i

optimal_lambda_valid = lambda_arr[optimal_lambda_valid_index]
print("The optimal lambda value for validation data is " +␣
 ↪str(optimal_lambda_valid))

# The optimal lambda value which makes R2 max for test data
max_r2_test = max(r2_test_final)
for i in range(len(r2_test_final)):
    if (r2_test_final[i] == max_r2_test):
        optimal_lambda_test_index = i

optimal_lambda_test = lambda_arr[optimal_lambda_test_index]
print("The optimal lambda value for test data is " + str(optimal_lambda_test))
```

We get the following optimal lambda values for both test and validation sets.

```
The optimal lambda value for validation data is 395.5436244734702
The optimal lambda value for test data is 354.077390896527
```

For all lambda values, graph of R2 values vs both test and validation sets is given below:

```
[9]: fig_num = 0
     plt.figure(fig_num, figsize=(8,8))
     t = np.logspace(0, 12, num=5000, base=10)
     plt.ylabel('R^2 values')
     plt.xlabel('lambda values')
     plt.title('Graph of R^2 values vs lambda values')
     plt.plot(r2_valid_final, color='r')
     plt.plot(r2_test_final, color='b')
     plt.grid()
     plt.legend(['validation data', 'test data',])
     plt.xscale('log')
     plt.show(block=False)
```



Those global maximum points for both validation and test sets are the previously reported optimal lambdas.

## Part b

In this part, we will determine 95% confidence intervals for the OLS model. We will use the ridge regression model but when we set the lambda to 0, we get the OLS model which is discussed previously.

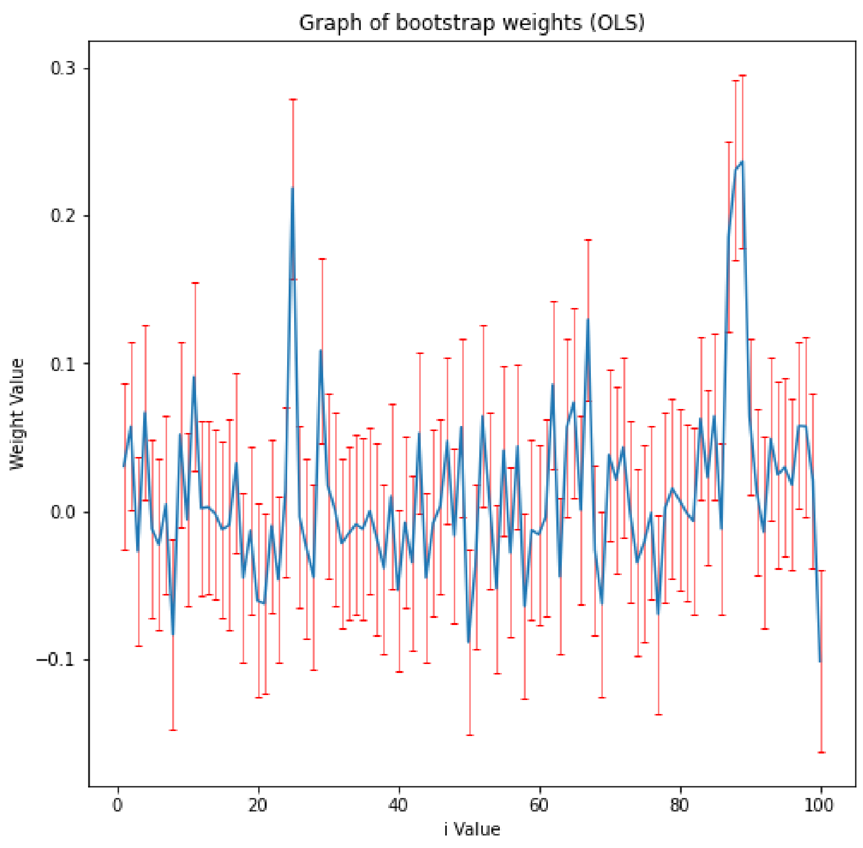
We are asked to use bootstrapping to generate more data since we do not have much data. For any given number of iterations in bootstrapping, we will generate that much of input and output samples from Xn and Yn. For each input and output pair, the OLS model will be fitted. Then we will take the mean of each parameter and compute the 95% confidence interval. Then standart deviation for each parameter will be computed and then finally, error bars that represents those confidence interval will be plotted. Python code for that procedure is given below:

[13]:
```
# Part B

def bootstrap(X, Y, lambdaa, number_iterations):
    size = np.shape(X)[0]
    x_bootstrap = np.zeros((np.shape(X)[0],np.shape(X)[1]))
    y_bootstrap = np.zeros(np.shape(Y))
    weights_bootstrap = np.zeros((100,500))

    for i in range(number_iterations):
        for j in range(size):
            x_bootstrap = np.array(x_bootstrap)
            y_bootstrap = np.array(y_bootstrap)
            index = np.random.randint(0, 1000, size=1)
            x_bootstrap[j,:] = X[index,:]
            y_bootstrap[j] = Y[index]
        weights_bootstrap[:,i] = ridge_regression(x_bootstrap, y_bootstrap, lambdaa)
        x_bootstrap = np.zeros((np.shape(X)[0],np.shape(X)[1]))
        y_bootstrap = np.zeros(np.shape(Y))
    weights_bootstrap = np.array(weights_bootstrap)
    weights_mean = np.mean(weights_bootstrap, axis = 1)
    weights_std = np.std(weights_bootstrap, axis = 1)
    return weights_mean, weights_std
```

[14]:
```
weights_mean_OLS, weights_std_OLS = bootstrap(Xn, Yn, 0, 500)
```

[21]:
```
fig_num += 1
plt.figure(fig_num, figsize=(8,8))
plt.title('Graph of bootstrap weights (OLS)')
plt.xlabel('i Value')
plt.ylabel('Weight Value')
plt.errorbar(np.arange(1, 101), weights_mean_OLS, 2 * weights_std_OLS,␣
 ↪ecolor='r',elinewidth=0.5, capsize=2)
plt.show(block=False)
```

The resulting error bar is shown as:



We can see from the plot that weights that are significantly different than 0 are so low in numbers. This means that 500 iterations for bootstrapping is enough for this model.

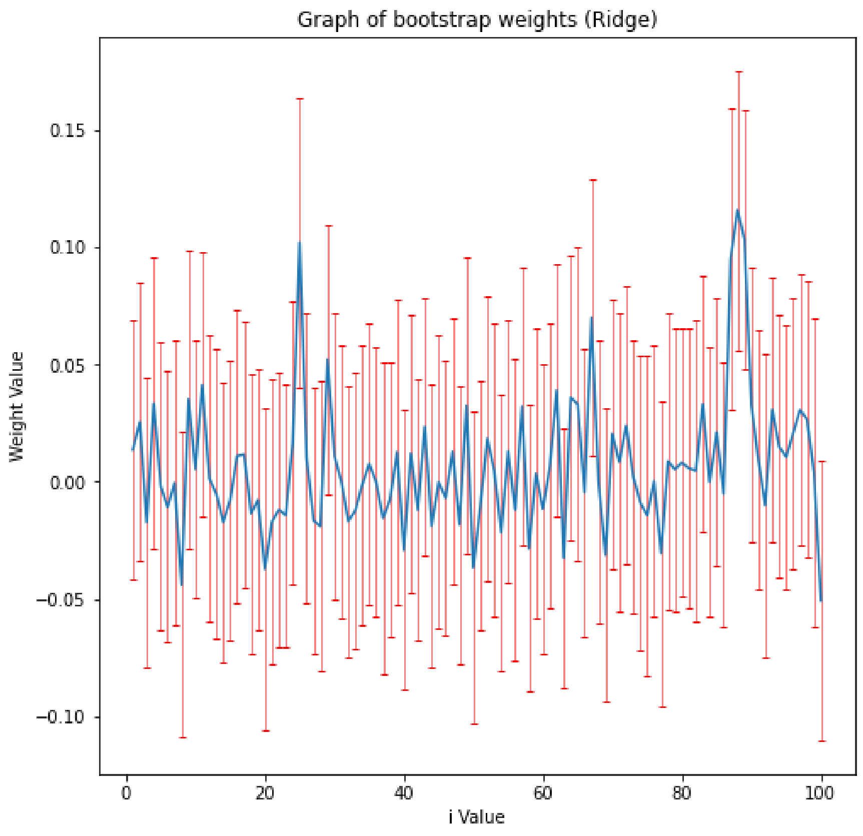
## Part c

In this part, we repeat the same procedure we have done in part b but the only difference is instead of an OLS model we will fit ridge regression model. To be able fit the ridge regression, instead of lambda equals zero, we will use the optimal lambda value that found in the part a. The Python code for that prodecure is given below:

[20]:
```
# Part C

weights_mean_ridge, weights_std_ridge = bootstrap(Xn, Yn, optimal_lambda_test, 500)
```

[23]:
```
fig_num += 1
plt.figure(fig_num, figsize=(8,8))
plt.title('Graph of bootstrap weights (Ridge)')
plt.xlabel('i Value')
plt.ylabel('Weight Value')
plt.errorbar(np.arange(1, 101), weights_mean_ridge, 2 * weights_std_OLS,␣
 ↪ecolor='r',elinewidth=0.5, capsize=2)
plt.show(block=False)
```

The resulting error bar is shown as:



As can be seen from the graph, weights that are trained with ridge regression are so close to zero. And when weights that are trained with ridge regression are more closer to zero than that are trained with OLS. This is due to regularization parameter of the ridge regression.

## Question 2

### Part a

In this question, responses from two populations of neurons is given. Those responses are stored in pop1 with size (7,1) and pop2 with size (5,1). We will test whether mean of responses differ by population or not. For that Hypothesis Testing. Basically, we will assume that these two different populations come from the same population. We can show this null hypothesis as:

$$H_0 = \mu_1 - \mu_2 = x = 0$$

$$H_A = \mu_1 - \mu_2 = x \neq 0$$

First we will apply bootstrapping since pop1 and pop2 has only 1 sample for each neuron. We will combine pop1 and pop2 into an array of size (12,1). Then from this combined array, we will generate 10000 samples using bootstrapping. For each 10000 samples, we will divide it into two random arrays with size (7,1) and (5,1) which are the original pop1 and pop2 sizes. Then we will calculate the mean of those two arrays. This procedure will be done for 10000 times which is the number of iterations. Python code for that is below:

[1]:
```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
import hdf5storage
import h5py
from scipy.stats import norm
```

[2]:
```
with h5py.File('hw3_data3.mat', 'r') as file:
    pop1 = list(file['pop1'])
with h5py.File('hw3_data3.mat', 'r') as file:
    pop2 = list(file['pop2'])

pop1 = np.array(pop1)
pop2 = np.array(pop2)
pop1.flatten()
pop2.flatten()

print("Shape of pop1 is: ")
print(np.shape(pop1))
print("Shape of pop2 is: ")
print(np.shape(pop2))
```

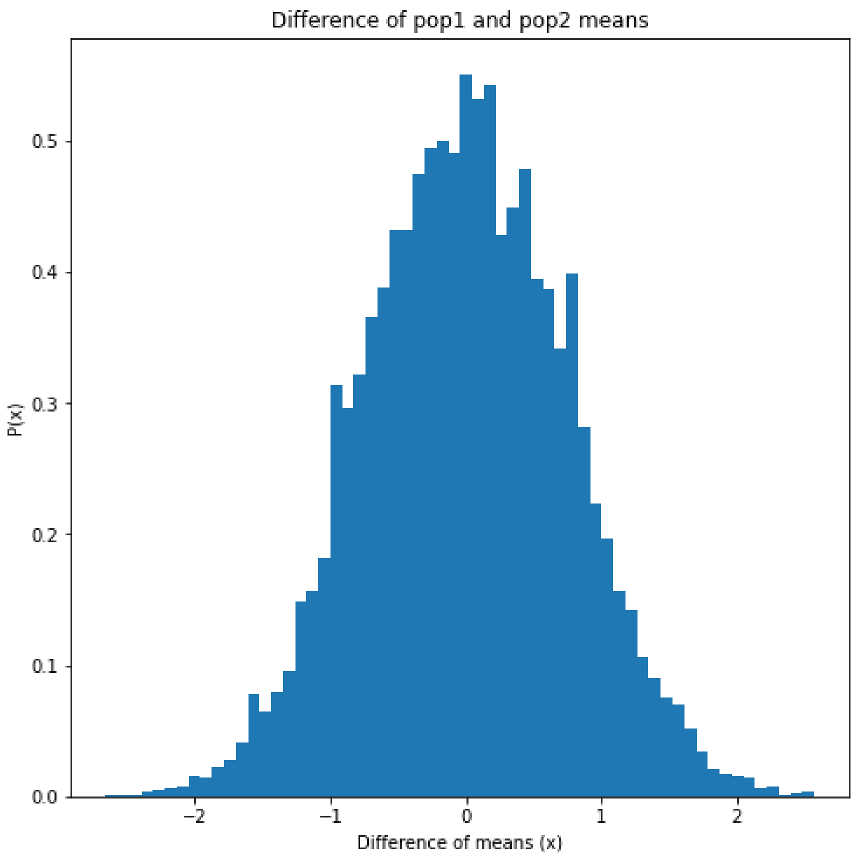
```
Shape of pop1 is:
(7, 1)
Shape of pop2 is:
(5, 1)
```

[3]:
```
# Part A

def bootstrap(data, number_iterations):
    np.random.seed(7)
    size = np.shape(data)[0]
    data_bootstrap = np.zeros(np.shape(data)[0])
    result_bootstrap = []

    for i in range(number_iterations):
        for j in range(size):
            data_bootstrap = np.array(data_bootstrap)
            index = np.random.randint(0, size, size=1)
            data_bootstrap[j] = data[index]
        result_bootstrap.append(data_bootstrap)
        data_bootstrap = np.zeros(np.shape(data)[0])
    result_bootstrap = np.array(result_bootstrap)
    return result_bootstrap
```

[4]:
```
def difference_of_means(pop1, pop2, number_iterations):
    pops = np.concatenate((pop1, pop2))
    pops_bootstrap = bootstrap(pops, number_iterations)
    pop1_bootstrap = []
    pop2_bootstrap = []
    for i in range(np.shape(pops_bootstrap)[1]):
        if (i < np.shape(pop1)[0]):
            pop1_bootstrap.append(pops_bootstrap[:,i])
        else:
            pop2_bootstrap.append(pops_bootstrap[:,i])
    pop1_bootstrap = np.array(pop1_bootstrap)
    pop2_bootstrap = np.array(pop2_bootstrap)
    mean1 = np.mean(pop1_bootstrap, axis=0)
    mean2 = np.mean(pop2_bootstrap, axis=0)
    diff_of_means = mean1 - mean2
    sigma = np.std(diff_of_means)
    mu = np.mean(diff_of_means)
    return diff_of_means, sigma, mu
```

[5]:
```
diff_of_means, sigma_pops, mu_pops = difference_of_means(pop1, pop2, 10000)
```

[6]:
```
fig_num = 0
plt.figure(fig_num, figsize=(8,8))
plt.xlabel('Difference of means (x)')
plt.ylabel('P(x)')
plt.title('Difference of pop1 and pop2 means')
plt.hist(diff_of_means, bins=60, density=True)
plt.show(block=False)
```



Then we will calculate the z and two-sided p values.

$$z = \frac{\overline{x} - \mu_0}{\sigma_0}$$

$$p = 2 * P(Z > z|H_0) = 2 * (1 - P(Z \leq z|H_0))$$

[7]:
```
def z_and_p_values(line, sigma, mu):
    z = (line - mu) / sigma
    p = 2 * (1 - norm.cdf(np.abs(z)))
    return z,p
```

[8]:
```
pops_line = np.mean(pop1) - np.mean(pop2)
z,p = z_and_p_values(pops_line, sigma_pops, mu_pops)
print("z-value is : ")
print(z)
print("\n")
print("Two side p-value is : ")
print(p)
```

```
z-value is :
2.5328129326156494


Two side p-value is :
0.0113151320630549
```

Two sided p-values is small, so the null hypothesis that these two populations have same mean is wrong. We can say that pop1 and pop2 are two distinct populations.

### Part b

In this part, we are given vox1 and vox2 which are two distinct brain voxels. Again, we will compare the similarity of these two voxels using Pearson coefficient. Since the size of the voxels are (50,1), we will use bootstrapping with 10000 number of iterations. We will also compute mean and compute the 95% confidence interval. The python code for that is:

[9]:
```
# Part B

with h5py.File('hw3_data3.mat', 'r') as file:
    vox1 = list(file['vox1'])
with h5py.File('hw3_data3.mat', 'r') as file:
    vox2 = list(file['vox2'])

vox1 = np.array(vox1)
vox2 = np.array(vox2)

print("Shape of vox1 is: ")
print(np.shape(vox1))
print("Shape of vox2 is: ")
print(np.shape(vox2))
```

```
Shape of vox1 is:
(50, 1)
Shape of vox2 is:
(50, 1)
```

[10]:
```
def correlation(vox1, vox2, number_iterations, PartC = False):
    size = np.shape(vox1)[0]
    vox1_bootstrap = bootstrap(vox1, number_iterations)
    vox2_bootstrap = bootstrap(vox2, number_iterations)
    if (PartC == True):
        np.random.seed(15)
        np.random.shuffle(vox2_bootstrap)

    result_bootstrap = np.zeros(number_iterations)

    for i in range(number_iterations):
        result_bootstrap[i] = np.corrcoef(vox1_bootstrap[i], vox2_bootstrap[i])[0,␣
 ↪1]
    result_bootstrap = np.array(result_bootstrap)
    return result_bootstrap
```

[11]:
```
cor_bootstrap = correlation(vox1, vox2, 10000)
```

[12]:
```
sorted_cor_bootstrap = np.sort(cor_bootstrap)
corr_mean = np.mean(sorted_cor_bootstrap)
lowerPercentile = np.percentile(sorted_cor_bootstrap, 5 / 2)
upperPercentile = np.percentile(sorted_cor_bootstrap, 95 + (5 / 2))
print("Mean if correlation is: " + str(corr_mean))
print("\n")
print('95 Percent Confidence Interval : (%1.4f, %1.4f)' %␣
 ↪(lowerPercentile,upperPercentile))
```

```
Mean of correlation is: 0.5575702100845678
```

```
95 Percent Confidence Interval : (0.3206, 0.7576)
```

To find the 95% confidence interval we get the values of $250^{th}$ and $750^{th}$ percentiles. Also correlation coefficient of vox1 and vox2 is 0.5576.

## Part c

In this part, we are still using the vox1 and vox2 arrays. However, the definition of our null hypothesis is different. We assume vox1 and vox2 are uncorrelated and coefficient of correlation is zero. We can define it as:

$$H_0 = p_{corr} = 0$$

$$H_A = p_{corr} < 0$$

The code for that is given below:

[13]:
```
# Part C
cor_bootstrap_c = correlation(vox1, vox2, 10000, True)
```

[14]:
```
fig_num += 1
plt.figure(fig_num, figsize=(8,8))
plt.yticks([])
plt.xlabel('Correlation (y)')
plt.ylabel('P(y)')
plt.title('Correlation between vox1 and vox2')
plt.hist(cor_bootstrap_c, bins=60, density=True)
plt.show(block=False)
```

We are also asked to compute z value and one sided p value. We can calculate z value from:

$$z = \frac{\overline{x} - 0}{std(p|H_0)}$$

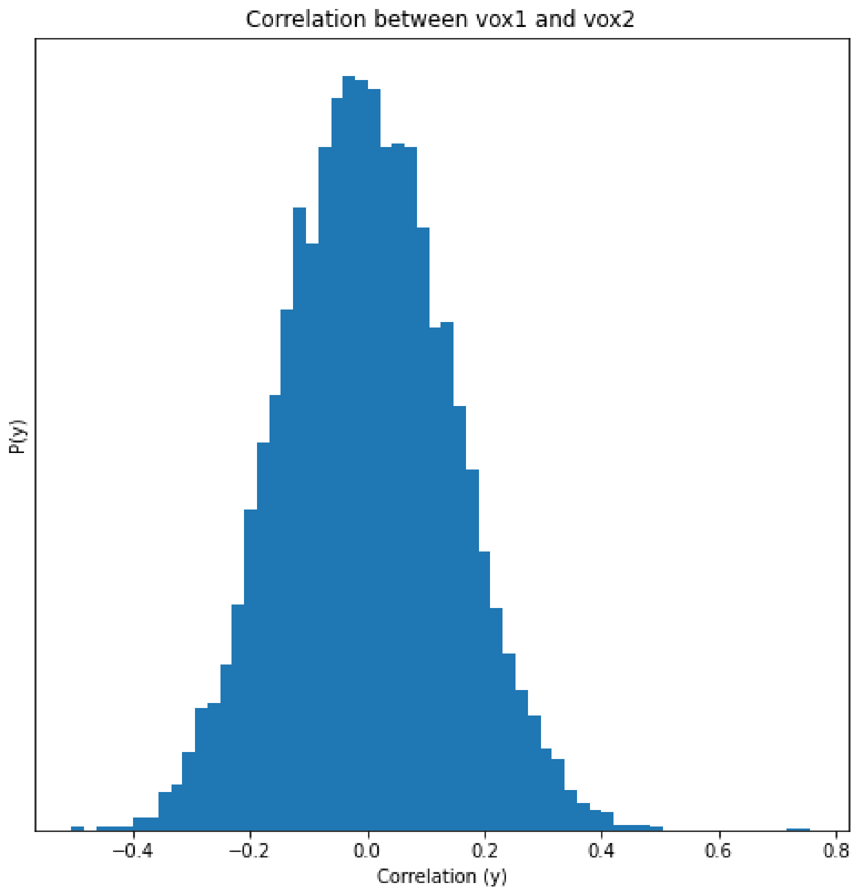
where $\overline{x}$ = 0.5576 from part b. Therefore, z = 3.982.

[15]:
```
vox1 = np.array(vox1)
vox2 = np.array(vox2)
vox1 = vox1.flatten()
vox2 = vox2.flatten()

c_line = np.corrcoef(vox1, vox2)[0,1]
sigma_c = np.std(cor_bootstrap_c )
mu_c = np.mean(cor_bootstrap_c )

z = (c_line - mu_c) / sigma_c
p = 1 - norm.cdf(z)
print("z-value is : ")
print(z)
print("\n")
print("p-value is : ")
print(p)
```

```
z-value is :
3.971119824778546
```

```
p-value is :
3.576779294101051e-05
```

We found the z values as 3.971 which is really close to calculated z value whixh is 3.982. One sided p value is very small so we can say that our assumption is incorrect. This result confirms the part b. We can also see the correlation between vox1 and vox2 from graph below:



## Part d

In this part, we are given building and face arrays that stores mean BOLD responses of neuron responses to building and face images. We build our null hypothesis similar to the one in part a.

$$H_0 = \mu_1 - \mu_2 = x = 0$$

$$H_A = \mu_1 - \mu_2 = x \neq 0$$

where $\mu_1$ and $\mu_2$ are the mean BOLD responses to building and face images. Hypothesis states that if there is no difference between the response than the mean difference should be zero. Given two images, respond can be in 4 ways which are: 1. Face - Face 2. Face - Building 3. Building - Face 4. Building - Building

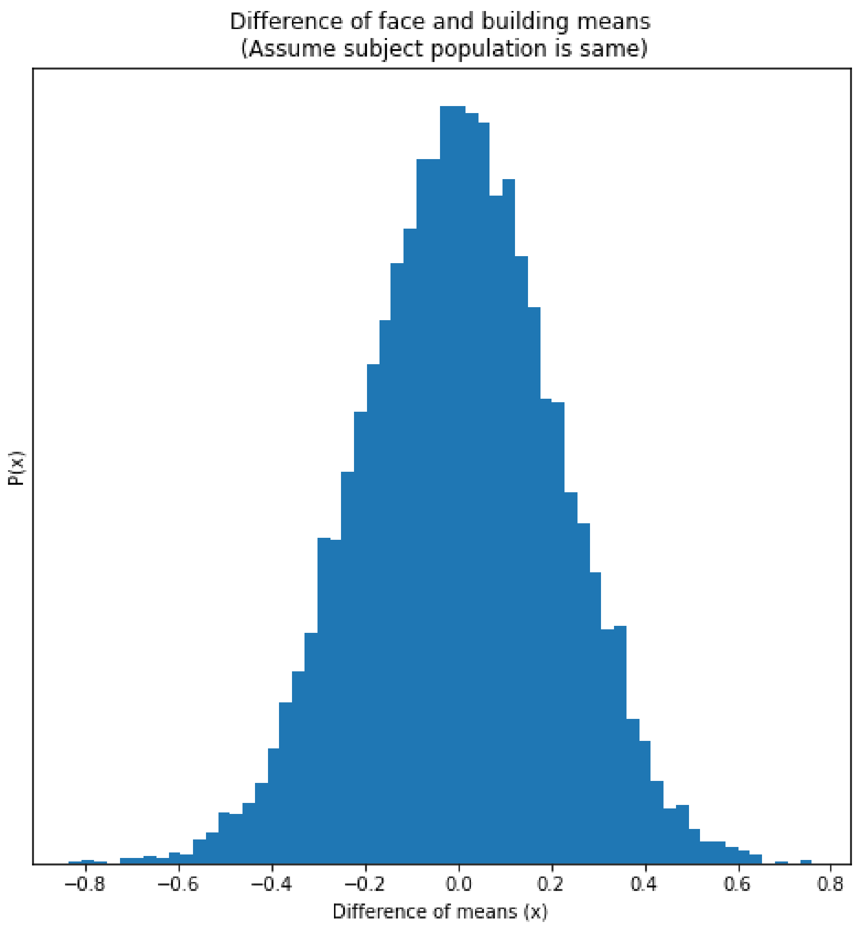
Since we make an assumption of responses are same, we pick a random outcome and record the mean difference. For 20 subjects, we record the mean difference of randomly selected outcome and record the mean of each subject's difference of means. This procedure will be done for 10000 iterations of bootstrapping. The python code for that procedure is given below:

[16]:
```
# Part D

with h5py.File('hw3_data3.mat', 'r') as file:
    building = list(file['building'])
with h5py.File('hw3_data3.mat', 'r') as file:
    face = list(file['face'])

building = np.array(building)
face = np.array(face)

print("Shape of building is: ")
print(np.shape(building))
print("Shape of face is: ")
print(np.shape(face))
```

```
Shape of building is:
(20, 1)
Shape of face is:
(20, 1)
```

[17]:
```
difference_in_means = []
size = np.shape(face)[0]
sample = []
for i in range(10000):
    for j in range(size):
        choices = []
        index = np.random.randint(0, size, size=1)
        choices.append(building[index] - face[index])
        choices.append(face[index] - building[index])
        for k in range(2):
            choices.append(0)
        chosenOne = int(np.random.randint(0, len(choices), size=1))
        sample.append(choices[chosenOne])
    difference_in_means.append(np.mean(sample))
    sample = []
difference_in_means = np.array(difference_in_means)
difference_in_means = difference_in_means.flatten()
```

[18]:
```
fig_num += 1
plt.figure(fig_num, figsize=(8,8))
plt.yticks([])
plt.xlabel('Difference of means (x)')
plt.ylabel('P(x)')
plt.title('Difference of face and building means\n (Assume subject population is␣
 ↪same)')
plt.hist(difference_in_means, bins=60, density=True)
plt.show(block=False)
```



```
[19]: mu_fb = np.mean(difference_in_means)
      sigma_fb = np.std(difference_in_means)
      fb_line = np.mean(building) - np.mean(face)

      z,p = z_and_p_values(fb_line, sigma_fb, mu_fb)
      print("z-value is : ")
      print(z)
      print("\n")
      print("Two side p-value is : ")
      print(p)
```

```
z-value is :
-3.5887412544409685


Two side p-value is :
0.00033227841905580924
```

Again, the p value is very small indicating that the assumption made earlier is not true. That means the difference face and building responses are not zero.

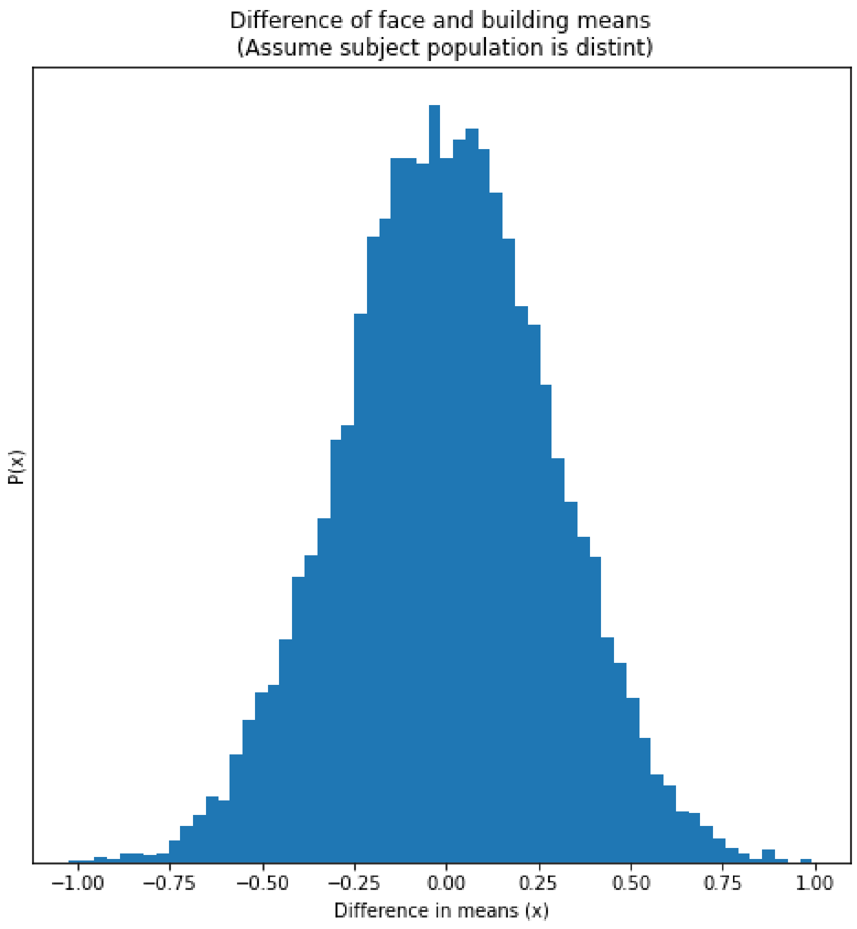
## Part e

This part is almost same with the previous part but this time we let the populations to be distinct. So the bootstrapping procedure done to data is similar to part a. The Python code for that is:

[20]:
```
# Part E
diff_of_means_fb, sigma_fb_2, mu_fb_2 = difference_of_means(face, building, 10000)
```

[21]:
```
fig_num += 1
plt.figure(fig_num, figsize=(8,8))
plt.yticks([])
plt.xlabel('Difference in means (x)')
plt.ylabel('P(x)')
plt.title('Difference of face and building means\n (Assume subject population is␣
  ↪distint)')

plt.hist(diff_of_means_fb, bins=60, density=True)
plt.show(block=False)
```

Graph of difference in means of building and face arrays:



[22]:
```
mu_fb_2 = np.mean(diff_of_means_fb)
sigma_fb_2 = np.std(diff_of_means_fb)
fb_line_2 = np.mean(building) - np.mean(face)

z,p = z_and_p_values(fb_line_2, sigma_fb_2, mu_fb_2)
print("z-value is : ")
print(z)
print("\n")
print("Two side p-value is : ")
print(p)
```

```
z-value is :
-2.651567544945334


Two side p-value is :
0.00801190860639811
```

We see that the p value is really small, so that the assumption made is incorrect. Which means there is a difference in responses of face and building.