Homework 3 Report

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

April 24, 2020

Contents

Question 1

This question presents the response samples (**Yn**) of a neural population which is Blood-oxygen level dependent (BOLD) and 100 regressors that would explain the responses given in (**Xn**). We use Ridge and Linear regression to build the models and use R^2 statistics to compare which we are exclusively required to compute as the square of the Pearson's correlation coefficient between the measured and predicted responses. The choice of programming language for this question and assignment is Python. The library imports that are used throughout are given below.

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
from scipy.stats import norm
```

Part A This part asks us to fit regularized linear models to predict the BOLD responses as the weighted sum of given regressors. It requires us to perform **10-Fold Cross Validation** to find the optimal Ridge parameter (λ) which we select logarithmically in the range $[0, 10^{12}]$. For each Cross-Validation fold, we are expected to **three-way split** the data as $800, 100, 100$ samples in training, validation and test respectively. Then, we were required to graph the Average R^2 errors versus Ridge parameter and find the optimal parameter.

In order to run the algorithm, we have first divided the data into the required folds. Hence, we have created the function given below in order to split the data into ten folds. It takes the regressors, responses and an index value, in this case $i \in 0, ..., 9$, to give the i^{th} partition of the data.

```
def cvSplit(X,y,turn):
    FOLD = 10
    num_sample = X.shape[0]
    num_fold = int(num_sample/FOLD)
    val_ind = turn*num_fold
    test_ind = (turn+1)*num_fold
    val_end = test_ind
    if(test_ind >= num_sample):
        test_ind = 0
    train_ind = test_ind + num_fold
    test_end = train_ind
    if(train_ind >= num_sample):
        train_ind = 0
    X_val_set = X[val_ind:val_end]
    y_val_set = y[val_ind:val_end]
    X_test_set = X[test_ind:test_end]
    y_test_set = y[test_ind:test_end]
    if(train_ind == 2*num_fold):
        X_train_set = X[train_ind:num_sample]
        y_train_set = y[train_ind:num_sample]
    elif(train_ind == 0):
        X_train_set = X[0:val_ind]
        y_train_set = y[0:val_ind]
    elif(train_ind == num_fold):
```

```
        X_train_set = X[train_ind:val_ind]
        y_train_set = y[train_ind:val_ind]
    else:
        X_train_set = np.vstack((X[0:val_ind], X[train_ind:
                                                num_sample]))
        y_train_set = np.vstack((y[0:val_ind], y[train_ind:
                                                num_sample]))
    return(X_train_set,y_train_set,X_val_set,y_val_set,X_test_set,
                                            y_test_set)
```

For further preparation, since we will be solving for the least squares solution for the Ridge regression in every model we fit, we derive the closed form solution. Normally, this would be computationally inefficient when compared with a Gradient Descent type approach however, since we only have 1000 data points, the computation time is not much of a concern. Hence, we define the least squares error term with regularization as follows.

$$E = \sum_{i=1}^{N} \left(y_i - \sum_{j=1}^{k} \omega_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{k} \omega_j^2 \tag{1}$$

We can rewrite this term in terms of matrices defined as follows.

$$X_{Nxk} = \begin{bmatrix} x_1^T \\ x_2^T \\ \dots \\ x_N^T \end{bmatrix} \tag{2}$$

$$y_{Nx1} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_N \end{bmatrix} \tag{3}$$

$$\omega_{kx1} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \dots \\ \omega_k \end{bmatrix} \tag{4}$$

$$E = (y - X\omega)^2 + \lambda \omega^T \omega \tag{5}$$

Then, we can simply use matrix algebra to readjust the terms as follows.

$$\begin{aligned} E &= (y - X\omega)^T (y - X\omega) + \lambda \omega^T \omega && (6)\\ &= (y^T - \omega^T X^T)(y - X\omega) + \lambda \omega^T \omega + \lambda \omega^T \omega && (7)\\ &= y^T y - y^T X \omega - \omega^T X^T y + \omega^T X^T X \omega + \lambda \omega^T \omega && (8)\\ &= y^T y - 2\omega^T X^T y + \omega^T X^T X \omega + \lambda \omega^T \omega && (9)\\ & && (10) \end{aligned}$$

Then, we take the derivative of the error function with respect to weights and equate it to zero in order to find the optimal weights.

$$\frac{\partial E}{\partial \omega} = -2X^T y + 2X^T X\omega + 2\lambda\omega = 0 \tag{11}$$

$$-X^T y + X^T X\omega + \lambda\omega = 0 \tag{12}$$

$$(X^T X + \lambda I_k)\omega = X^T y \tag{13}$$

$$\omega^* = (X^T X + \lambda I_k)^{-1} X^T y \tag{14}$$

if $X^T X + \lambda I_k$ is invertible, and we know it is for $\lambda > 0$. Hence, we define the helper function **findRidgeSol** to calculate the weights.

```
def findRidgeSol(X,y,r_coeff=0):
    xTx = np.matmul(X.T, X)
    xTy = np.matmul(X.T, y)
    return np.matmul(np.linalg.inv(xTx+r_coeff*np.identity(xTx.
                                        shape[0])), xTy)
```

Also, we define a helper function to find the R^2 error, which is simply the square of the Pearson Correlation found with **np.corrcoef**.

```
def calc_r2(y_true, y_pred):
    pearson = np.corrcoef(y_true.T,y_pred.T)[0,1]
    return pearson**2
```

Then we define the log-space for the Ridge parameter and for all parameters, run the cross validated Ridge solution and save the average R^2 error for each parameter in validation and test. The code is provided below.

```
ridge_params = np.logspace(0, 12, num=500, base=10)
r_2_list_val = []
r_2_list_test = []
for coeff in ridge_params:
    sum_r_2_val = 0
    sum_r_2_test = 0
    for i in range(10):
        X_train, y_train, X_val, y_val, X_test, y_test = cvSplit(
                                                xn,yn,i)
        w_ridge = findRidgeSol(X_train, y_train, coeff)
        test_pred = np.matmul(X_test, w_ridge)
        val_pred = np.matmul(X_val,w_ridge)
        r_sq_val = calc_r2(y_val, val_pred)
        r_sq_test = calc_r2(y_test, test_pred)
        sum_r_2_val += r_sq_val
        sum_r_2_test += r_sq_test
    r_2_list_val.append(sum_r_2_val/10)
    r_2_list_test.append(sum_r_2_test/10)
```

Then we graph these curves and find the optimal parameter value (λ_{opt}).

```
opt_lamda = ridge_params[np.argmax(r_2_list_val)]
print('Optimal Lambda Value:', opt_lamda)
```

```
print('Best Average R^2 Value for Validation', r_2_list_val[np.
                                        argmax(r_2_list_val)])
print('Best Average R^2 Value for Test', r_2_list_test[np.argmax(
                                        r_2_list_val)])

plt.xscale('log')
plt.xlabel('$\lambda$ (log)')
plt.ylabel('$R^2$')
plt.title('Average $R^2$ vs Ridge Coefficient Values')
plt.grid('on')
plt.plot(ridge_params,np.asarray(r_2_list_val))
plt.plot(ridge_params,np.asarray(r_2_list_test))
plt.show()
```

```
Optimal Lambda Value: 395.5436244734702
Best Average R^2 Value for Validation 0.1525988778486
Best Average R^2 Value for Test 0.1604206104492847
```

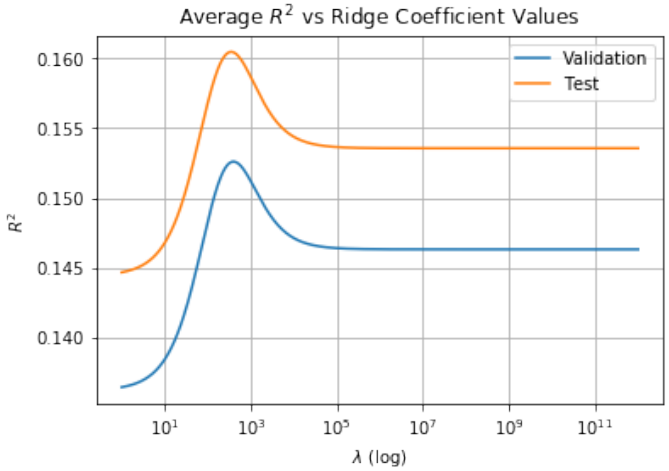

Figure 1: Average R^2 vs λ for Validation and Test (10-Fold CV)

We observe that both of the curves have a unique global maximum. meaning Average R^2 is concave in λ domain. The λ_{opt} value is found as 359.54 from the validation set. and the corresponding R^2 Averages for this value of lambda is given as 0.153 and 0.160 in validation and test respectively. Usually, we expect to find the test error to be higher than validation since we optimize the hyperparameter sets according to validation statistics. However, here, the R^2 score was much higher in the test set, which was a surprise founding.

Part B This part asks us to compute the confidence intervals from the OLS (Ordinary Least Squares) model in Part A which corresponds to the $\lambda = 0$ case in Ridge Regression.

$$E_{OLS} = \sum_{i=1}^{N} \left(y_i - \sum_{j=1}^{k} \omega_j x_{ij} \right)^2 \tag{15}$$

which gives the solution,

$$\omega^* = (X^T X)^{-1} X^T y \tag{16}$$

For this part, we should have generated 500 bootstrap iterations which is means taking repeatable subsamples from the original data. The code used to generate bootstrap samples is provided below.

```
def gen_bootstrap(X,y):
    num_sample = X.shape[0]
    seq = [np.random.randint(0,num_sample) for i in range(
                                            num_sample)]
    X_boot = X[seq]
    y_boot = y[seq]
    return (X_boot, y_boot)
```

Then, we run the same algorithm we have in the previous part and determine the confidence intervals for each weight (ω_i) and plot on a ω_i versus i plot.

```
BOOTSTRAP_ITER = 500
w_boots = []
for i in range(BOOTSTRAP_ITER):
    x_boot, y_boot = gen_bootstrap(xn,yn)
    w_boot = findRidgeSol(x_boot,y_boot)
    w_boots.append(w_boot)
w_boots = np.asarray(w_boots)
w_means = np.mean(w_boots, axis=0)
w_stds = np.std(w_boots, axis=0)
x_ = np.arange(100)+1
plt.errorbar(x_, w_means[:,0], yerr=2*w_stds[:,0], ecolor='r',
                                        elinewidth=0.4, capsize=2)
plt.title('Model Weights - OLS')
plt.xlabel('$i$')
plt.ylabel('Average Value of $w_i$')
plt.show()
```

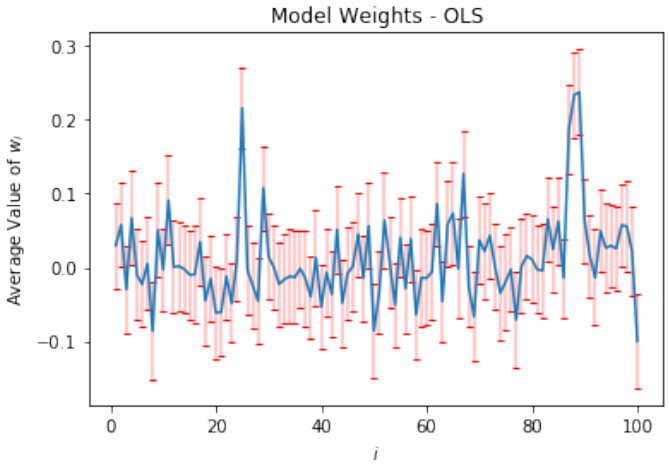

Figure 2: ω_i vs i (OLS - 500 Bootstrap Iterations)

Furthermore, we want obtain the information of which weight parameters are significantly different than 0 at a significance level of $p < 0.05$. For that, we create a hypothesis testing procedure where we obtian the p value of each weight parameters and take the ones that are smaller than the given threshold.

```
z = w_means / w_stds
p = 2*(1-norm.cdf(np.abs(z)))
significant_OLS = np.argwhere(p < 0.05)
significant_OLS = significant_OLS[significant_OLS != 0]
print('Indices of i different than 0:\n', significant_OLS)
```

```
Indices of i different than 0:
[ 3  7 10 20 24 28 49 51 57 61 64 66 68 76 82 84 86 87 88 89 96 97 99]
```

Part C This part follows tha same procedure that we have followed with the previous part, however this time we use the λ_{opt} value that we have found in Part A for each bootstrap sample. The following code is used to obtain the given results.

```
BOOTSTRAP_ITER = 500
w_boots = []
for i in range(BOOTSTRAP_ITER):
    x_boot, y_boot = gen_bootstrap(xn,yn)
    w_boot = findRidgeSol(x_boot,y_boot,opt_lamda)
    w_boots.append(w_boot)
w_boots = np.asarray(w_boots)
w_means = np.mean(w_boots, axis=0)
w_stds = np.std(w_boots, axis=0)
x_ = np.arange(100)+1
plt.errorbar(x_, w_means[:,0], yerr=2*w_stds[:,0], ecolor='r',
                                        elinewidth=0.4, capsize=2)
plt.title('Model Weights - Ridge w/$\lambda_{opt}$')
plt.xlabel('$i$')
plt.ylabel('Average Value of $w_i$')
plt.show()
```

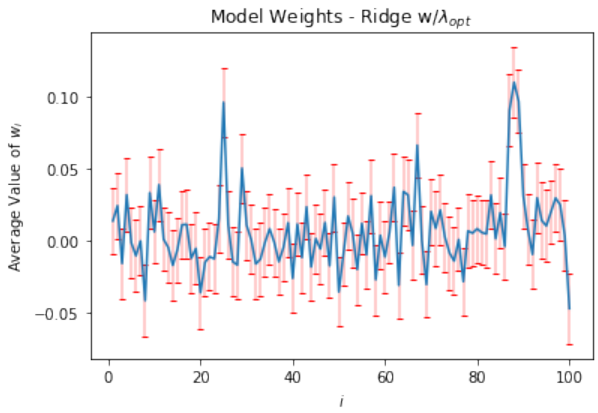

Figure 3: ω_i vs i - (Ridge with λ_{opt} - 500 Bootstrap Iterations)

Then, we again find the weight indices that are significantly closer to zero. Here, we expect the number of parameters that are closer to zero to be more than the previous part since ridge regression punishes the weights that are bigger with its square and moves the weight distribution to zero as much as possible. The same code with the previous part is run and the results are as follows.

```
z = w_means / w_stds
p = 2*(1-norm.cdf(np.abs(z)))
significant_OLS = np.argwhere(p < 0.05)
significant_OLS = significant_OLS[significant_OLS != 0]
print('Indices of i that are different than 0:\n', significant_OLS
                                                  )
```

```
Indices of i that are different than 0:
[ 1  3  7  8 10 19 24 28 39 42 48 49 56 57 61 62 63 64 66 68 76 82 86 87
88 89 92 96 97 99]
```

In parallel with our assumption, we see that there are more weights that are closer to zero in weight distribution since we know that Ridge penalty forces the model weights to smaller L2 norms. However, this procedure also increases the standard deviation of the parameters due to the added complexity.

Question 2

In this question, there are pairs of neural response measurements are provided. The question asks us to convey the relationships between the pairs using known techniques such as Hypothesis Testing and Confidence İntervals. The provided data is summarized as follows and each structure has **float** as its data type.

1. **pop1:** (7x1) Neural population with 7 neurons.
2. **pop2:** (5x1) Neural population with 5 neurons.
3. **vox1:** (50x1) BOLD (Blood-Oxygen Level Dependent) responses in a voxel.
4. **vox2:** (50x1) BOLD responses in a voxel.
5. **building:** (20x1) Average BOLD responses in a face-seşective region of brain to images of buildings.
6. **face:** (20x1) Average BOLD responses in a face-selective region of brain to images of faces.

The methods that we are going to use in this problem are Hypothesis Testing and Confidence Intervals, which are dual problems. Before, starting the solutions, we will explain Hypothesis testing since the Confidence Intervals are explained through the previous question.

Hypothesis Testing

Hypothesis testing is a method of statistical inference. Usually, two statistical datasets are compared, or a dataset obtained by sampling is compared to a data synthesized from an ideal distribution/model. The tests are used to determine if the outcomes of a study (data) are accepted/rejected under the null hypothesis according to a predetermined significance level (p). The procedure that is followed is given as follows.

1. Find the observed value of the test statistic T, T_{obs}. In our case, we use the Z-statistics, where we assume that the distribution of the data is Gaussian and try to decide if the means are different given that the variances are the same and the sample size is large.
2. Calculate the values, which is the probability under the null hypothesis.
3. Reject, the null hypothesis (H_0), in favor of alternate hypothesis (H_A) if and only if the p-value is less than or equal to the significance level threshold, which is 0.05 in our case.

The imports that are used in this question are provided below.

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
from scipy.stats import norm
```

Part A In this part, we use **pop1** and **pop2**, and the objective is to determine if the mean responses of the two populations are significantly differentthan each other. We are required to bootstrap 10000 samples and find the two-tailed p-value for under the null hypothesis that two datasets have the same distribution. In order to do that, we first define the null and alternate hypothesis.

$$H_0 := \mu_1 = \mu_2 \rightarrow \mu_1 - \mu_2 = 0 \tag{1}$$
$$H_A := \mu 1 \neq \mu_2 \rightarrow \mu_1 - \mu_2 \neq 0 \tag{2}$$

Here H_0 is what we initially assume and check about our datasets, which is the hypothesis that the means of the two populations' responses are similar. The alternate hypothesis is the exact opposite. In order to reach a conclusion, we need to obtain the probabilities under the null hypothesis: $P(x|H_0)$. In order to find, we combine the two populations' responses as if they come from the same population and generate 10000 bootstrap samples from the combined responses. Then, we divide each sample according to the number of neurons in each population. We compute individual means of each populations' bootstrap samples. Then we find the difference of means for each individual bootstrap samples and project them onto a histogram. The code provided below is used to implement this part. We use the same bootstrap function that we have in the previous question.

```
BOOTSTRAP_ITER = 10000
x_comb = np.vstack((pop1,pop2))
mean_diffs = []
for i in range(BOOTSTRAP_ITER):
    x_boot = gen_bootstrap(x_comb)
    pop1_boot = x_boot[0:pop1.shape[0]]
    pop2_boot = x_boot[pop1.shape[0]:x_boot.shape[0]]
    mean_diffs.append(np.mean(pop1_boot)-np.mean(pop2_boot))
plt.title('Difference of Means (10000 Bootstrap Iterations)')
plt.xlabel('$(\mu_1-\mu_2)| H_0$')
plt.ylabel('$P(x|H_0)$')
plt.hist(mean_diffs, bins=60, density=True)
plt.show()
```

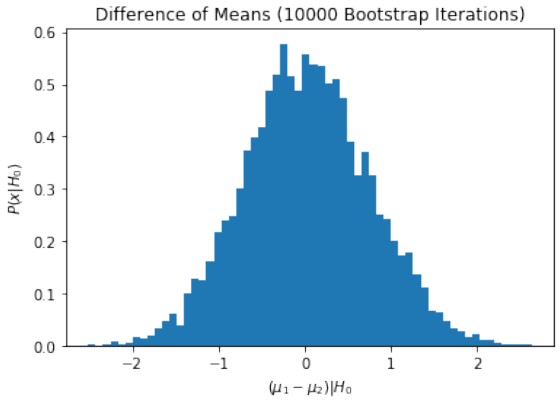

Figure 4: $P(x|H_0)$ - Difference of Means Under Null Hypothesis (10000 Resamples)

We can see that the distribution resembles a Gaussian. If we define the real means of **pop1** and **pop2** as μ_1^* and μ_2^* and their difference as $\overline{x} = \mu_1^* - \mu_2^*$, we are interested in the probability $P(|x| > \overline{x}|H_0)$ since we would like to know how likely the difference of means falls outside the observed $\overline{x}$ given H_0 and take absolute value since we are looking for the two-sided p-value. Hence, we use Z-Statistics shown as follows to find the p-values.

$$P(|x| > \overline{x}|H_0) = P(|Z| > \frac{\overline{x} - \mu_0}{\sigma_0}|H_0) \tag{3}$$

As we think of x as Gaussian, Z becomes the standard normal, μ_0 is the mean and σ_0 is the standard deviation of the sampling distribution. Hence, we define the z-value as,

$$z = \frac{\overline{x} - \mu_0}{\sigma_0} \tag{4}$$

which makes the probability given as,

$$P(|x| > \overline{x}|H_0) = P(|Z| > z|H_0) \tag{5}$$

Then, we can define the two-sided p-value only with minor probability properties.

$$p = P(|Z| > z|H_0) = 2P(Z > |H_0) = 2(1 - P(Z \leq z|H_0)) = 2(1 - F_Z(z)) \tag{6}$$

Here, we can see we can express the p-value in terms of the cumulative distribution of the standard normal random variable, which is easy to compute. The code is as follows.

```
x_bar = np.mean(pop1) - np.mean(pop2)
std = np.std(mean_diffs)
mean = np.mean(mean_diffs)
z = (x_bar - mean)/std
print('z-value: ', z)
p = 2*(1-norm.cdf(z))
print('Two-Sided p-value: ', p)
```

```
z-value:  2.581616441356418
Two-Sided p-value:  0.009833881096261932
```

The two-sided p-value is found to be very small. We can understand from this that the defined null hypothesis is very unlikely. Here, we can conclude that **pop1** and **pop2** are samples from different populations.

Part B In this part, we are required to use **vox1** and **vox2**, to measure their correlations. We are required to run 10000 bootstrap iterations and find the mean and 95% confidence interval of the correlation, and the percentage of the zero correlations in the bootstrapped sample space. Here, we measure similarity using Pearson correlation as we have in the previous question. Here, we need to decide whether we should resample the data identically or independently. The null hypothesis

presumes that the two are correlated, hence, we need to consider the order in which we resample the data. This corresponds to identical resampling. Hence, we define the hypotheses as follows.

$$H_0 := \rho_{vox1,vox2} \neq 0 \tag{7}$$
$$H_A := \rho_{vox1,vox2} = 0 \tag{8}$$

Then we define the function below, to run the identical resampling, meaning that we run the same order when resampling both **vox1** and **vox2**.

```
def gen_bootstrap_vox(x,y):
    num_sample = x.shape[0]
    seq = [np.random.randint(0,num_sample) for i in range(
                                            num_sample)]
    x_boot = x[seq]
    y_boot = y[seq]
    return (x_boot,y_boot)
```

Then, we run the bootstrap operation 10000 times, calculate and save the Pearson Correlations between the samples, then plot on a histogram as follows.

```
corr_vector = []
for i in range(BOOTSTRAP_ITER):
    vox1_temp, vox2_temp = gen_bootstrap_vox(vox1,vox2)
    corr_temp = np.corrcoef(vox1_temp.T, vox2_temp.T)
    corr_vector.append(corr_temp[0,1])
plt.title('Identical Resampling (10000 Bootstrap Iterations)')
plt.xlabel('$\\rho | H_0$')
plt.ylabel('$P(x|H_0)$')
plt.hist(corr_vector, bins=60, density=False)
plt.show()
print('Mean correlation:', np.mean(corr_vector))
```

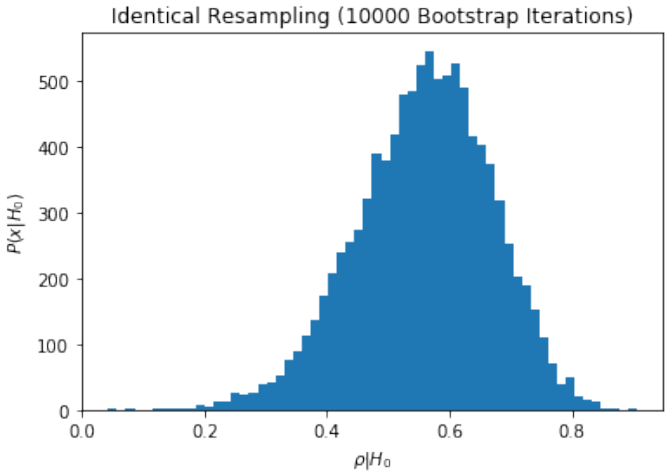

Figure 5: $P(x|H_0)$ - Correlations under Identical Resampling (10000 Resamples)

Then, we are required to calculate the confidence interval for the correlation vector. To find the confidence interval, we among the sorted vector, found the 2.5% and

the 97.5% section. For, the we have used the **np.percentile** function and defined the helper function givne bwlow.

```
def confidence_interval(x, percent=95):
    low_end = np.percentile(x, (100-percent)/2)
    high_end = np.percentile(x, percent+ (100-percent)/2)
    return low_end,high_end
```

Then, we have used this function to calculate the lower and upper bounds for the correlation vector.

```
CONFIDENCE = 95
lower_bound, upper_bound = confidence_interval(np.asarray(
                                        corr_vector))
print('%2d%% confidence interval: (%1.5f, %1.5f)' % (CONFIDENCE,
                                        lower_bound, upper_bound))
```

```
95% confidence interval: (0.32349, 0.75850)
```

The last thing we have done is to calculate the percentile of the resamples with zero correlation.

```
corr_zero = len(np.where(np.isclose(corr_vector, 0))[0])
print('Number of zero-correlation values:', corr_zero)
```

```
Number of zero-correlation values: 0
```

This results is expected since the correlation between **vox1** and **vox2** is 0.5645 and a near-same value (0.5576), can be observed with Figure 5. Also, since identical resampling is used, there is a somewhat forced correlation between the bootstrapped samples, which explains why there isn't any zero-correlation sample.

Part C In this part, again ve use **vox1** and **vox2**, but this time we place the null hypothesis on the responses having zero correlation. Then, we are required to find the one-tailed p-value for two voxel responses having zero or negative correlation. The hyoptheses are formulated below.

$$H_0 := \rho_{vox1,vox2} \leq 0 \tag{9}$$
$$H_A := \rho_{vox1,vox2} > 0 \tag{10}$$

This time, we are resampling the vectors to break the correlation between them which corresponds to independent resampling, meaning, we bootstrap the vectors completely independent form each other, disregarding the index relation that we have not in the previous part. Then, we save the correlations of the bootstrap samples and put on the histogram. The code is provided below.

```
corr_vector = []
for i in range(BOOTSTRAP_ITER):
    vox1_temp = gen_bootstrap(vox1)
    vox2_temp = gen_bootstrap(vox2)
    corr_temp = np.corrcoef(vox1_temp.T, vox2_temp.T)
    corr_vector.append(corr_temp[0,1])
plt.title('Independent Resampling (10000 Bootstrap Iterations)')
plt.xlabel('$\\rho | H_0$')
plt.ylabel('$P(x|H_0)$')
plt.hist(corr_vector, bins=60, density=False)
plt.show()
print('Mean correlation:', np.mean(corr_vector))
```

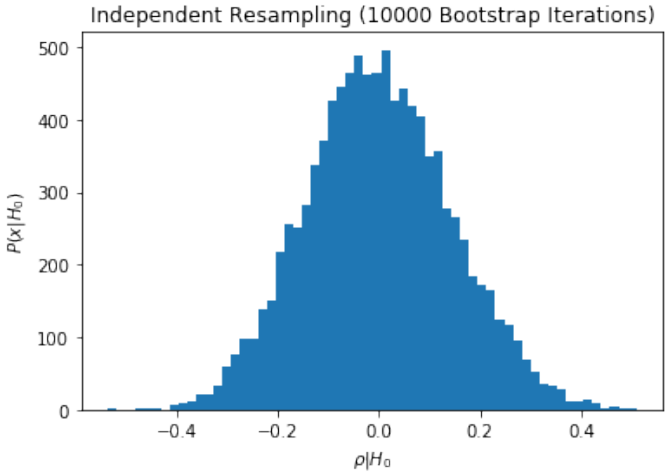

Figure 6: $P(x|H_0)$ - Correlations under Independent Resampling (10000 Resamples)

Then, we mathematically define the one-tailed p-value as follows.

$$p = P(x > \overline{x}|H_0) = P(Z > z|H_0),\ where\, z = \frac{\overline{x} - \mu_0}{\sigma_0} \tag{11}$$

$$p = 1 - P(Z \leq z|H_0) = 1 - F_Z(z) \tag{12}$$

Hence, we find the one-tailed p-value for the correlation vector.

```
x_bar = np.corrcoef(vox1.T,vox2.T)[0,1]
std = np.std(corr_vector)
mean = np.mean(corr_vector)
z = (x_bar - mean)/std
print('z-value: ', z)
p = 1-norm.cdf(z)
print('One-Sided p-value: ', p)
```

```
z-value:  3.9148195911090977
One-Sided p-value:  4.5235914834518276e-05
```

The p-value is very small, meaning we should reject the null hypothesis which conveys a zero or negative correlation between the voxel responses. This result also correlates with the results that we have found in Part B, which resulted with a positive correlation, here we see the hypothesis of zero or negative correlation is unlikely. This outcome makes sense as Hypothesis Testing and Confidence Intervals are dual problems of each other. We conclude that the voxel responses are positively correlated even though they are resampled independently.

Part D Here, we are given two set of responses, namely **face** and **building** and we build on the null hypothesis that there is no difference between the responses. Hence, we can define the null and alternate hypotheses as follows.

$$H_0 := \mu_{face} - \mu_{building} = 0 \tag{13}$$
$$H_A := \mu_{face} - \mu_{building} \neq 0 \tag{14}$$

Here, since we know that the subjects in both experiment are the same by index, we use a different approach while resampling the data. For each resample, we choose randomly from the four different options for a random subject as,

- Building, Building
- Face, Face
- Building, Face and
- Face, Building.

After 20 responses are resampled, which is the original number of results in both vectors, we save the difference in their means and save them to again use a histogram chart to evaluate further. The following function is created for the bootstrapping procedure.

```
def gen_bootstrap_fb(x,y):
    num_sample = x.shape[0]
    seq = [np.random.randint(0,num_sample) for i in range(
                                            num_sample)]
    x_boot = x[seq]
    y_boot = y[seq]
    dif_vector = []
    for i in range(num_sample):
        x_temp = x_boot[i]
        y_temp = y_boot[i]
        x_dif = x_temp - y_temp
        opt = [0,0,x_dif,-1*x_dif]
        dif_vector.append(np.random.choice(opt))
    return np.asarray(dif_vector)
```

Hence, we use it as we have described.

```
mean_diffs = []
for i in range(BOOTSTRAP_ITER):
    dif_vect = gen_bootstrap_fb(face,building)
    mean_diffs.append(np.mean(dif_vect)[0])
```

```
plt.title('Difference of Means Pairwise Resampling (10000
                                        Bootstrap Iterations)')
plt.xlabel('$(\mu_{face}-\mu_{building})| H_0$')
plt.ylabel('$P(x|H_0)$')
plt.hist(mean_diffs, bins=60, density=True)
plt.show()
```

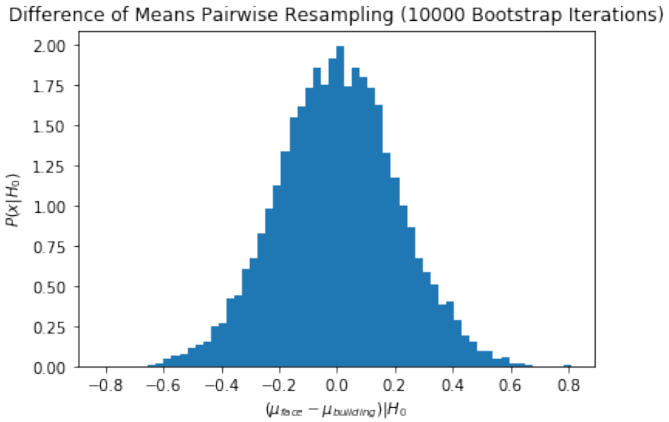

Figure 7: $P(x|H_0)$ - Mean Differences under Pairwise Resampling (10000 Iterations)

Then, again we find the two-sided p-value as we have in Part A.

```
x_bar = np.mean(face) - np.mean(building)
std = np.std(mean_diffs)
mean = np.mean(mean_diffs)
z = (x_bar - mean)/std
print('z-value: ', z)
p = 2*(1-norm.cdf(np.abs(z)))
print('Two-Sided p-value: ', p)
```

```
z-value:  3.560863658919425
Two-Sided p-value:  0.0003696369895145324
```

Here, we observe again a very small p-value, meaning that we should reject the null hypothesis in favor of the alternate hypothesis. The actual difference in means that was obtained from the sample in hand was much greater than 0, and the standard deviation of the population was large considering the number of samples, which resulted in a smaller p-value. Hence, we should conclude that the face and image responses are not the same with each other statistically.

Part E This part again requires us to use **face** and **building** however this time we convey the idea that the subject in the experiments are distinct. Hence, we can use the same bootstrapping technique that we have in Part A. The defined hypotheses are given below.

$$H_0 := \mu_{face} - \mu_{building} = 0 \tag{15}$$

$$H_A := \mu_{face} - \mu_{building} \neq 0 \tag{16}$$

The code is provided.

```
x_comb = np.vstack((face,building))
mean_diffs = []
for i in range(BOOTSTRAP_ITER):
    x_boot = gen_bootstrap(x_comb)
    face_boot = x_boot[0:face.shape[0]]
    build_boot = x_boot[face.shape[0]:x_boot.shape[0]]
    mean_diffs.append(np.mean(face_boot)-np.mean(build_boot))
plt.title('Difference of Means (10000 Bootstrap Iterations)')
plt.xlabel('$(\mu_1-\mu_2)| H_0$')
plt.ylabel('$P(x|H_0)$')
plt.hist(mean_diffs, bins=60, density=True)
plt.show()
```

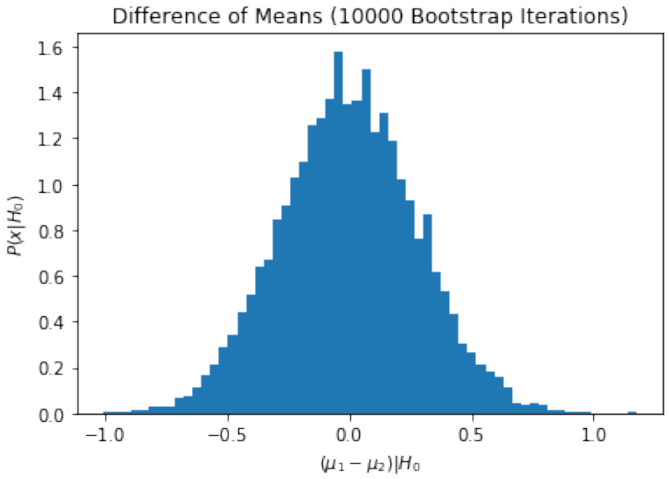

Figure 8: $P(x|H_0)$ - Mean Differences under Independent Resampling (10000 Iterations)

Then, we calculate the two-sided p-value.

```
x_bar = np.mean(face) - np.mean(building)
std = np.std(mean_diffs)
mean = np.mean(mean_diffs)
z = (x_bar - mean)/std
print('z-value: ', z)
p = 2*(1-norm.cdf(np.abs(z)))
print('Two-Sided p-value: ', p)
```

```
z-value:  2.6775845393867996
Two-Sided p-value:  0.0074155136243534425
```

The p-value is again small, so we reject the null hypothesis in favor of the alternate hypothesis. Hence, we understand that even if the subjects are different in the two experiments, face responses show a separate distribution than the building responses.

Python Code

```
import sys
import numpy as np
import matplotlib.pyplot as plt
import h5py
from scipy.stats import norm

question = sys.argv[1]

def ayhan_okuyan_21601531_hw3(question):
    if question == '1' :
        print('Question 1')
        with h5py.File('hw3_data2.mat', 'r') as f:
            print(f.keys())
            yn = f['Yn'].value.T
            xn = f['Xn'].value.T
            yn = yn.astype(float)
            xn = xn.astype(float)
        print('Yn:', yn.shape)
        print('Xn:', xn.shape)

        print('Part A')
        ridge_params = np.logspace(0, 12, num=500, base=10)

        r_2_list_val = []
        r_2_list_test = []
        for coeff in ridge_params:
            sum_r_2_val = 0
            sum_r_2_test = 0
            for i in range(10):
                X_train, y_train, X_val, y_val, X_test, y_test = cvSplit
                                                        (xn, yn, i)
                w_ridge = findRidgeSol(X_train, y_train, coeff)
                test_pred = np.matmul(X_test, w_ridge)
                val_pred = np.matmul(X_val, w_ridge)
                r_sq_val = calc_r2(y_val, val_pred)
                r_sq_test = calc_r2(y_test, test_pred)
                sum_r_2_val += r_sq_val
                sum_r_2_test += r_sq_test

            r_2_list_val.append(sum_r_2_val / 10)
            r_2_list_test.append(sum_r_2_test / 10)

        opt_lamda = ridge_params[np.argmax(r_2_list_val)]
        print('Optimal Lambda Value:', opt_lamda)
        print('Best Average R^2 Value for Validation', r_2_list_val[np.
                                            argmax(r_2_list_val)])
        print('Best Average R^2 Value for Test', r_2_list_test[np.argmax
                                            (r_2_list_val)])

        plt.xscale('log')
        plt.xlabel('$\lambda$ (log)')
        plt.ylabel('$R^2$')
```

```
plt.title('Average $R^2$ vs Ridge Coefficient Values')
plt.grid('on')
plt.plot(ridge_params, np.asarray(r_2_list_val))
plt.plot(ridge_params, np.asarray(r_2_list_test))
plt.legend(['Validation', 'Test'])
plt.show()

print('Part B')
BOOTSTRAP_ITER = 500
w_boots = []
for i in range(BOOTSTRAP_ITER):
    x_boot, y_boot = gen_bootstrap(xn, yn)
    w_boot = findRidgeSol(x_boot, y_boot)
    w_boots.append(w_boot)
w_boots = np.asarray(w_boots)
print(w_boots.shape)

w_means = np.mean(w_boots, axis=0)
w_stds = np.std(w_boots, axis=0)

print(w_means.shape, w_stds.shape)

x_ = np.arange(100) + 1
plt.errorbar(x_, w_means[:, 0], yerr=2 * w_stds[:, 0], ecolor='r
                                    ', elinewidth=0.4, capsize=2
                                    )
plt.title('Model Weights - OLS')
plt.xlabel('$i$')
plt.ylabel('Average Value of $w_i$')
plt.show()

z = w_means / w_stds
p = 2 * (1 - norm.cdf(np.abs(z)))
significant_OLS = np.argwhere(p < 0.05)
significant_OLS = significant_OLS[significant_OLS != 0]
print('Indices of i different than 0:\n', significant_OLS)

print('Part C')
BOOTSTRAP_ITER = 500
w_boots = []
for i in range(BOOTSTRAP_ITER):
    x_boot, y_boot = gen_bootstrap(xn, yn)
    w_boot = findRidgeSol(x_boot, y_boot, opt_lamda)
    w_boots.append(w_boot)
w_boots = np.asarray(w_boots)

w_means = np.mean(w_boots, axis=0)
w_stds = np.std(w_boots, axis=0)

x_ = np.arange(100) + 1
plt.errorbar(x_, w_means[:, 0], yerr=2 * w_stds[:, 0], ecolor='r
                                    ', elinewidth=0.4, capsize=2
                                    )
plt.title('Model Weights - Ridge w/$\lambda_{opt}$')
plt.xlabel('$i$')
```

```
        plt.ylabel('Average Value of $w_i$')
        plt.show()

        z = w_means / w_stds
        p = 2 * (1 - norm.cdf(np.abs(z)))
        significant_OLS = np.argwhere(p < 0.05).flatten()
        significant_OLS = significant_OLS[significant_OLS != 0]
        print('Indices of i that are different than 0:\n',
                                              significant_OLS)

    elif question == '2' :
        print('Question 2')
        with h5py.File('hw3_data3.mat', 'r') as f:
            print(f.keys())
            pop1 = f['pop1'].value
            pop2 = f['pop2'].value
            vox1 = f['vox1'].value
            vox2 = f['vox2'].value
            building = f['building'].value
            face = f['face'].value
        print('pop1:\n', pop1.T)
        print('pop2:\n', pop2.T)
        print('vox1:\n', vox1.T)
        print('vox2:\n', vox2.T)
        print('building:\n', building.T)
        print('face:\n', face.T)

        print('Part A')
        BOOTSTRAP_ITER = 10000
        x_comb = np.vstack((pop1, pop2))
        mean_diffs = []
        for i in range(BOOTSTRAP_ITER):
            x_boot = gen_bootstrap_v2(x_comb)
            pop1_boot = x_boot[0:pop1.shape[0]]
            pop2_boot = x_boot[pop1.shape[0]:x_boot.shape[0]]
            mean_diffs.append(np.mean(pop1_boot) - np.mean(pop2_boot))
        plt.title('Difference of Means (10000 Bootstrap Iterations)')
        plt.xlabel('$(\mu_1-\mu_2)| H_0$')
        plt.ylabel('$P(x|H_0)$')
        plt.hist(mean_diffs, bins=60, density=True)
        plt.show()
        x_bar = np.mean(pop1) - np.mean(pop2)
        std = np.std(mean_diffs)
        mean = np.mean(mean_diffs)
        z = (x_bar - mean) / std
        print('z-value: ', z)
        p = 2 * (1 - norm.cdf(z))
        print('Two-Sided p-value: ', p)

        print('Part B')
        corr_vector = []
        for i in range(BOOTSTRAP_ITER):
            vox1_temp, vox2_temp = gen_bootstrap_vox(vox1, vox2)
            corr_temp = np.corrcoef(vox1_temp.T, vox2_temp.T)
            corr_vector.append(corr_temp[0, 1])
```

```
plt.title('Identical Resampling (10000 Bootstrap Iterations)')
plt.xlabel('$\\rho | H_0$')
plt.ylabel('$P(x|H_0)$')
plt.hist(corr_vector, bins=60, density=False)
plt.show()
print('Mean correlation:', np.mean(corr_vector))
CONFIDENCE = 95
lower_bound, upper_bound = confidence_interval(np.asarray(
                                        corr_vector))
print('%2d%% confidence interval: (%1.5f, %1.5f)' % (CONFIDENCE,
                                         lower_bound, upper_bound))
corr_zero = len(np.where(np.isclose(corr_vector, 0))[0])
print('Number of zero-correlation values:', corr_zero)

print('Part C')
corr_vector = []
for i in range(BOOTSTRAP_ITER):
    vox1_temp = gen_bootstrap_v2(vox1)
    vox2_temp = gen_bootstrap_v2(vox2)
    corr_temp = np.corrcoef(vox1_temp.T, vox2_temp.T)
    corr_vector.append(corr_temp[0, 1])
plt.title('Independent Resampling (10000 Bootstrap Iterations)')
plt.xlabel('$\\rho | H_0$')
plt.ylabel('$P(x|H_0)$')
plt.hist(corr_vector, bins=60, density=False)
plt.show()
print('Mean correlation:', np.mean(corr_vector))
x_bar = np.corrcoef(vox1.T, vox2.T)[0, 1]
std = np.std(corr_vector)
mean = np.mean(corr_vector)
z = (x_bar - mean) / std
print('z-value: ', z)
p = 1 - norm.cdf(z)
print('One-Sided p-value: ', p)

print('Part D')
mean_diffs = []
for i in range(BOOTSTRAP_ITER):
    dif_vect = gen_bootstrap_fb(face, building)
    mean_diffs.append(np.mean(dif_vect)[0])
plt.title('Difference of Means Pairwise Resampling (10000
                                        Bootstrap Iterations)')
plt.xlabel('$(\mu_{face}-\mu_{building})| H_0$')
plt.ylabel('$P(x|H_0)$')
plt.hist(mean_diffs, bins=60, density=True)
plt.show()
x_bar = np.mean(face) - np.mean(building)
std = np.std(mean_diffs)
mean = np.mean(mean_diffs)
z = (x_bar - mean) / std
print('z-value: ', z)
p = 2 * (1 - norm.cdf(np.abs(z)))
print('Two-Sided p-value: ', p)

print('Part E')
```

```
        x_comb = np.vstack((face, building))
        mean_diffs = []
        for i in range(BOOTSTRAP_ITER):
            x_boot = gen_bootstrap_v2(x_comb)
            face_boot = x_boot[0:face.shape[0]]
            build_boot = x_boot[face.shape[0]:x_boot.shape[0]]
            mean_diffs.append(np.mean(face_boot) - np.mean(build_boot))
        plt.title('Difference of Means (10000 Bootstrap Iterations)')
        plt.xlabel('$(\mu_1-\mu_2)| H_0$')
        plt.ylabel('$P(x|H_0)$')
        plt.hist(mean_diffs, bins=60, density=True)
        plt.show()
        x_bar = np.mean(face) - np.mean(building)
        std = np.std(mean_diffs)
        mean = np.mean(mean_diffs)
        z = (x_bar - mean) / std
        print('z-value: ', z)
        p = 2 * (1 - norm.cdf(np.abs(z)))
        print('Two-Sided p-value: ', p)



#Question 1 Functions
def findRidgeSol(X, y, r_coeff=0):
    xTx = np.matmul(X.T, X)
    xTy = np.matmul(X.T, y)
    return np.matmul(np.linalg.inv(xTx + r_coeff * np.identity(xTx.shape
                                        [0])), xTy)

def cvSplit(X,y,turn):
    FOLD = 10
    num_sample = X.shape[0]
    num_fold = int(num_sample/FOLD)
    val_ind = turn*num_fold
    test_ind = (turn+1)*num_fold
    val_end = test_ind
    if(test_ind >= num_sample):
        test_ind = 0
    train_ind = test_ind + num_fold
    test_end = train_ind
    if(train_ind >= num_sample):
        train_ind = 0
    X_val_set = X[val_ind:val_end]
    y_val_set = y[val_ind:val_end]
    X_test_set = X[test_ind:test_end]
    y_test_set = y[test_ind:test_end]
    if(train_ind == 2*num_fold):
        X_train_set = X[train_ind:num_sample]
        y_train_set = y[train_ind:num_sample]
    elif(train_ind == 0):
        X_train_set = X[0:val_ind]
        y_train_set = y[0:val_ind]
    elif(train_ind == num_fold):
        X_train_set = X[train_ind:val_ind]
        y_train_set = y[train_ind:val_ind]
```

```
    else:
        X_train_set = np.vstack((X[0:val_ind], X[train_ind:num_sample]))
        y_train_set = np.vstack((y[0:val_ind], y[train_ind:num_sample]))
    return(X_train_set,y_train_set,X_val_set,y_val_set,X_test_set,
                                        y_test_set)

def calc_r2(y_true, y_pred):
    pearson = np.corrcoef(y_true.T,y_pred.T)[0,1]
    return pearson**2

def gen_bootstrap(X,y):
    num_sample = X.shape[0]
    seq = [np.random.randint(0,num_sample) for i in range(num_sample)]
    X_boot = X[seq]
    y_boot = y[seq]
    return (X_boot, y_boot)

#Question 2 Functions
def gen_bootstrap_v2(X):
    num_sample = X.shape[0]
    seq = [np.random.randint(0,num_sample) for i in range(num_sample)]
    X_boot = X[seq]
    return X_boot

def gen_bootstrap_vox(x,y):
    num_sample = x.shape[0]
    seq = [np.random.randint(0,num_sample) for i in range(num_sample)]
    x_boot = x[seq]
    y_boot = y[seq]
    return (x_boot,y_boot)

def confidence_interval(x, percent=95):
    low_end = np.percentile(x, (100-percent)/2)
    high_end = np.percentile(x, percent+ (100-percent)/2)
    return low_end,high_end

def gen_bootstrap_fb(x,y):
    num_sample = x.shape[0]
    seq = [np.random.randint(0,num_sample) for i in range(num_sample)]
    x_boot = x[seq]
    y_boot = y[seq]
    dif_vector = []
    for i in range(num_sample):
        x_temp = x_boot[i]
        y_temp = y_boot[i]
        x_dif = x_temp - y_temp
        opt = [0,0,x_dif,-1*x_dif]
        dif_vector.append(np.random.choice(opt))
    return np.asarray(dif_vector)

ayhan_okuyan_21601531_hw3(question)
```