Homework 3

EEE482- Computational Neuroscience

Efe Acer
21602217

Bilkent University, CS

Contents

Question 1

The data given for this question contains the responses of a V1 neuron in an awake monkey. The `resp1` variable of the dataset describes the level of response during a passive fixation condition, whereas the variable `resp2` represents response levels during a memory task.

Various models will be built throughout this question. Our regressor variable in these models will be `resp1` and the output (regressand) variable will be `resp2`. In other words, `resp1` is the independent variable and `resp2` is dependent on `resp1`.

1.a) The first model built is a linear one that is constructed according to the *Ordinary Least Squares* (OLS) method. Mathematically, the model is $y = X \cdot w$, where y is a vector of size N (200) that contains the values of `resp2`, X is a matrix of shape $N \times K$ (200×2) that contains the values of `resp1` in its first column and 1's in its second column, and w is the weight vector $[a, b]^T$ of size K (2). This formulation means that $y_n = x_n^T \cdot w = a \cdot x_{n1} + b$ is the value our model predicts for the nth regressor. Note that the second column of 1's in X is necessary to have the offset b, the 1's in this column are referred to as *bias terms*.

`Python` is used to implement the models in this question. Since the dataset is given as a `v7.3 .mat` file, we need to import the `h5py` module to load it. After the dataset is loaded we initialize the following vector, constant and matrices:

```
y = data['resp2'] # output
N = np.size(y)
X_raw = data['resp1'] # regressor
# append the bias terms to the regressor
X = X_raw.reshape((N, 1))
X = np.concatenate((X, np.ones((N, 1))), axis=1)
```

In OLS, the error is defined as the Euclidian Distance (L2 Norm) between an observation and a prediction. OLS tries to minimize the sum of the squares of the errors which corresponds to a model that best fits to the data in an Euclidian sense. Minimizing the sum of the squared errors is equivalent to minimizing the well known *Mean Squared Error* (MSE) cost function. MSE is known to be convex in w, i.e. it has a well defined global minimum. Hence, we can formulate OLS as follows:

$$MSE(w) := \frac{1}{2} \sum_{n=1}^{N} (x_n^T \cdot w - y_n)^2 \tag{1}$$

$$w^\star = \underset{w}{argmin}\ MSE(w) \tag{2}$$

(1) gives our definition for the MSE cost function. $w^\star$ in (2) refers to the optimal weight vector that establishes the best fit line.

We can arrive at an analytical solution to (2) as follows:

$$\nabla_w MSE(w) = \begin{bmatrix} \frac{\partial MSE(w)}{\partial a} \\ \\ \frac{\partial MSE(w)}{\partial b} \end{bmatrix} = \begin{bmatrix} \sum_{n=1}^{N} (x_n^T \cdot w - y_n) \cdot x_{n1} \\ \\ \sum_{n=1}^{N} (x_n^T \cdot w - y_n) \cdot x_{n2} \end{bmatrix} = X^T \cdot (X \cdot w - y)$$

Since MSE is convex we can enforce optimality by:

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

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

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

$$X^T X \cdot w^\star = X^T \cdot y$$

Thus, the optimal solution is:

$$w^\star = (X^T X)^{-1} X^T \cdot y \tag{3}$$

A `Python` function that implements the OLS solution follows:

```
def OLS(y, X):
    """
    Given the matrix containing the data labels y, and the matrix
    containing the regressors X; computes the optimal weight vector
    such that the meas squared error (MSE) is minimized.
    Args:
        y: The data labels
        X: The regressors
    Returns
        w_optimal: The optimal weight vector
    """
    w_optimal = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
    return w_optimal
```

The function is used to compute the weights of the best fit line for our dataset:

```
w_optimal = OLS(y, X)
print('The optimal weight vector that minimizes the mean squared error '
      'for the linear model is:\n w_optimal = [a, b] =',
      w_optimal)
```

The output of the previous code segment is as follows:

```
The optimal weight vector that minimizes the mean squared error for the linear
   model is:
w_optimal = [a, b] = [ 0.29820552 67.03397361]
```

After this calculation we can simply obtain the model predictions by executing the following line:

```
1 pred = X.dot(w_optimal)
```

Now that we have our predictions in `pred` vector, original labels (outputs) in `y` vector and raw regressors in `X_raw` vector; we can plot the data together with the linear model. The resulting plot is given below:

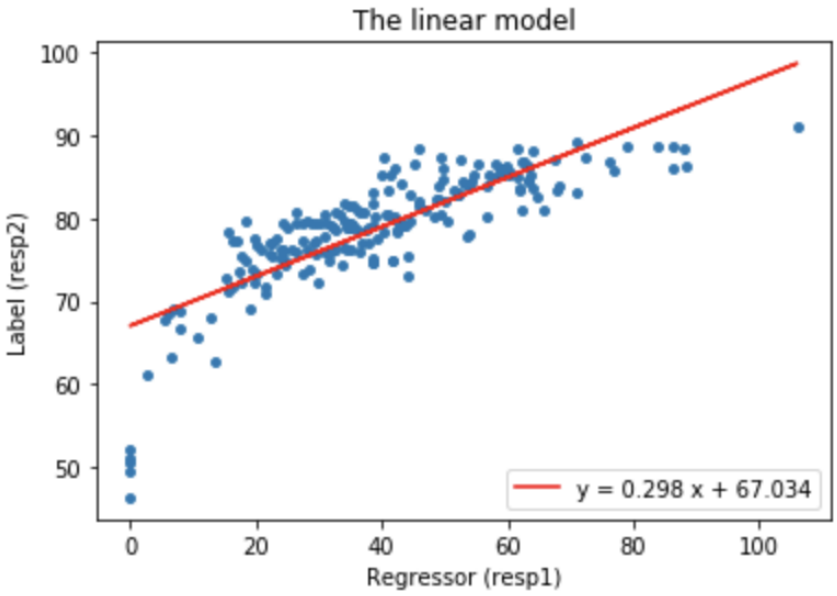

Figure 1: The linear model

What remains is to judge our model in terms of its accuracy. Since the value of the squared error itself is hard to interpret (it does not have units, etc.), we use the coefficient of determination R^2 to quantify model accuracy. R^2 is defined as the percent explained varance.

Total variance is simply:

$$Var_{total} = \frac{1}{N-1} \sum_{n=1}^{N} y_n - \overline{y} \tag{4}$$

Note that $\overline{y}$ in (4) is the sample mean of the labels.

We define unexplained variance as:

$$Var_{unexplained} = \frac{1}{N-1} \sum_{n=1}^{N} y_n - f(x_n) \tag{5}$$

Here, $f(x_n)$ is the output of our linear model given the nth regressor. In particular, $f(x_n) = x_n^T \cdot w$ for our linear model.

Using the definitions above, we can write R^2 as follows:

$$R^2 = 100 \times (1 - \frac{Var_{unexplained}}{Var_{total}}) \tag{6}$$

A `Python` function that computes and returns the explained variance, the unexplained variance and the coefficient of determination is provided below:

```
1  def test_model(y, pred):
2      """
3      Test a given linearized model by computing the coefficient
4      of determination (R^2). Returns the explained variance,
5      unexplained variance and R^2.
6      Args:
7          y: The data labels
8          pred: The predicted valus
9      Returns:
10         e_var: The explained variance
11         u_var: The unexplained variance
12         R2: The coefficient of determination
13     """
14     mean_y = np.mean(y)
15     N = np.size(y)
16     total_var = np.sum((y - mean_y) ** 2) / (N - 1)
17     u_var = np.sum((y - pred) ** 2) / (N - 1)
18     temp = total_var - u_var
19     R2 = 100 * (temp / total_var)
20     return R2 / 100, u_var, R2
```

Using the script below, we can test our linear model:

```
1  e_var, u_var, R2 = test_model(y, pred)
2  print('Explained variance of the linear model:', e_var)
3  print('Unexplained variance of the linear model:', u_var)
4  print('Coefficient of determination of the linear model:', R2)
```

The output of the previous script is:

```
Explained variance of the linear model: 0.6579235991547004
Unexplained variance of the linear model: 17.464077534973946
Coefficient of determination of the linear model: 65.79235991547004
```

Pearson Correlation Coefficient is a measure of the linear correlation between two variables. It is a number between -1 and 1, where -1 denotes a highly negative linear correlation and 1 denotes a highly positive linear correlation between the variables.

When we compare the square root of R^2, call it R, with the Pearson Correlation Coefficient between the regressor and the label we see that they are equal to each other apart from a scaling factor of 10. This scaling factor of 10 is there simply because of the factor of 100 in (6). The following code segment illustrates the case:

```
1 R = np.sqrt(R2)
2 pearson = np.corrcoef(y, X_raw)[0, 1]
3 print('The value of R:', R)
4 print('The value of pearson correlation between '
5       'the label and the regressor:', pearson)
```

The `Numpy` function `np.corrcoef` calculates the Pearson Correlation Coefficient. The output of the code is:

```
The value of R: 8.111248973830728
The value of pearson correlation between the label and the regressor:
   0.8111248973830725
```

This case where R equals the Pearson Correlation Coefficient between the regressor and the label does not generally hold but it is true in the case of pure linear regression.

1.b) Often times, the underlying distribution where the data is drawn from cannot be modeled by a linear function. This is particularly because the raw features of the regressors are usually weakly correlated with the output. One solution to this problem is *feature engineering*. This approach tries to generate new features using the existing ones and then fit a linear model where the feature space includes the newly generated features.

In our previous model, the features were the values of `resp1` (x_{n1}'s) and the bias terms (1's). Now as an additional feature, we add $x_{n2} = x_{n1}^2$ for all N data points. With the added feature column the nth row of the X matrix (x_n^T) becomes $[x^2, x, 1]$, where x denotes the nth `resp1` value.

Since we changed the dimension of the feature space, we should also reflect this change to our weight vector w. Size of w now becomes 3 instead of 2. w can be rewritten as $[a, b, c]^T$.

After this little trick our model predicts $y_n = x_n^T \cdot w = a \cdot x^2 + b \cdot x + c$ for the nth regressor. Although this is a second-order model, it is still linear in its feature space so we can still use the OLS solution in (3) to find the optimal w.

The additional feature column is appended to X using the following code:

```
X_order2 = (X_raw ** 2).reshape((N, 1))
X_order2 = np.concatenate((X_order2, X), axis=1)
```

Then the optimal weight vector $w^\star$ is computed by running the following code:

```
w_optimal_order2 = OLS(y, X_order2)
print('The optimal weight vector that minimizes the mean squared error for '
      'the linearized second order model is:\nw_optimal_order2 = [a, b, c] =',
      w_optimal_order2)
```

The output is:

```
The optimal weight vector that minimizes the mean squared error for the linearized
    second order model is:
w_optimal_order2 = [a, b, c] = [-4.42550345e-03 6.80811071e-01 6.04930517e+01]
```

We compute the predictions as `pred = X_order2.dot(w_optimal_order2)`, and plot the model together with the raw regressors. The resulting plot follows:

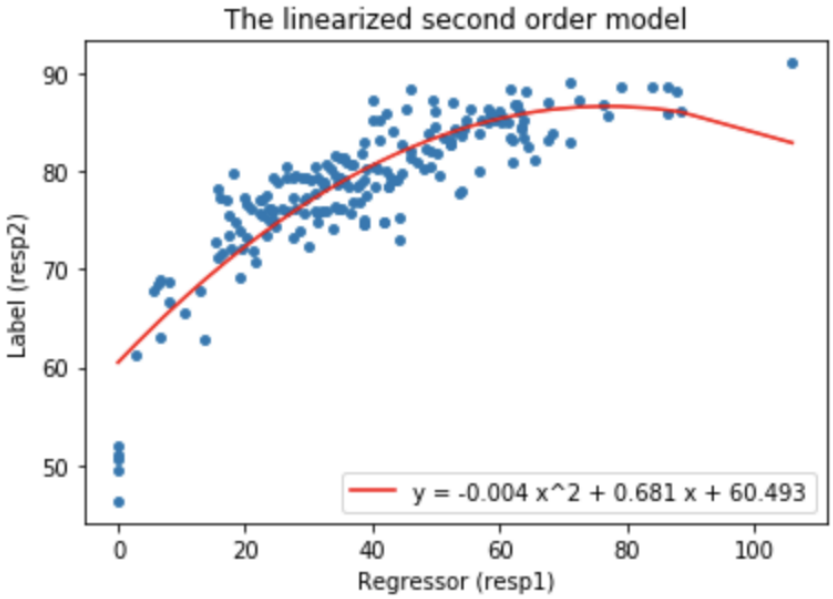

Figure 2: The linearized second order model

Again, we test the model accuracy by computing the explained variance, the unexplained variance and the coefficient of determination:

```
e_var, u_var, R2 = test_model(y, pred)
print('Explained variance of the linearized second order model:', e_var)
print('Unexplained variance of the linearized second order model:', u_var)
print('Coefficient of determination of the linearized second order:', R2)
```

The resulting values are:

```
Explained variance of the linearized second order model: 0.769297670953405
Unexplained variance of the linearized second order model: 11.778080428853912
Coefficient of determination of the linearized second order: 76.9297670953405
```

Just like Pearson Correlation Coefficient, *Spearman Correlation Coeffictient* measures the correlation between two variables. However, Spearman's correlation measures the strength and direction of the monotonic association between the variables rather than the linear association. A monotonic association is not as restrictive as a linear relationship. If there is a monotonic relationship between two variables, then as one variable increases, so does the other variable; or as one variable increases, the other variable decreases.

Recall that the first model had an R value that is equivalent to the Pearson Correlation Coefficient between the output and the input. Same applies for this model but here R corresponds to the Spearman Correlation Coefficient. The following code illustrates the case:

```
R = np.sqrt(R2)
spearman = spearmanr(y, X_raw)
print('The value of R:', R)
print('The value of spearman correlation coefficient between '
      'the labels and the regressor:', spearman.correlation)
```

The output is:

```
The value of R: 8.770961583278112
The value of spearman correlation coefficient between the labels and the regressor
    : 0.8488520877167255
```

Notice that the R value is more or less equal to the Spearman Correlation Coefficient, again disregarding the factor of 10. Hence, we can surely say that the model effectively captures the positive monotonic relation between `resp1` and `resp2`.

The Spearman Correlation Coefficient is computed using the `spearmanr` function of `scipy`'s `stats` module.

1.c) Another option is to fit a parametric nonlinear model to our dataset. In such a model, our prediction for the nth input x is $\hat{y_n} = a \cdot x^n + b$. There are various methods to tune the parameters a, n and b in this model. A commonly used method is to use least squares approximation, which tries to perform the following minimization:

$$\underset{\{a,n,b\}}{argmin} \sum_{n=1}^{N} (y_n - \hat{y_n})^2 \tag{7}$$

The `Python` equivalent of `Matlab`'s `lsqcurvefit` function given in the assignment description is the `curve_fit` function of `scipy`'s `optimize` module. The internal implementations of both methods use algorithms to perform the minimization above. Most probably, both methods use an iterative algorithm like *Stochastic Gradient Descent* (SGD) to perform the minimization. The behavior of the methods differ because of the differences in the implementation details such as the step size used in the iterative optimization procedure.

`curve_fit` function requires the parametric nonlinear model as an argument, hence we define:

```
def parametric_nonlinear_model(x, a, n, b):
    """
    An implementation of the parametric nonlinear model y = a * x^n + b
    that will be used in the curve_fit
    function.
    Args:
        x: The input value
        a, n, b: Parameters of the function that will be optimized
    Returns:
        result: The resulting y value
    """
    return a * (x ** n) + b
```

Then we tune the model parameters using the `curve_fit` method. First we set the starting point for the optimization as $\{a, n, b\} = \{1, 1, 0\}$ by providing the argument `p0=[1, 1, 0]` to the `curve_fit` function. The following code runs to find the values of the parameters:

```
w_optimal_parametric, _ = curve_fit(parametric_nonlinear_model,
                                    X_raw, y, p0=[1, 1, 0])
a_optimal = w_optimal_parametric[0]
n_optimal = w_optimal_parametric[1]
b_optimal = w_optimal_parametric[2]
print('The optimal values in the parametric nonlinear model (starting from (1, 1,
    0)):'
      '\n a = %f, n = %f, b = %f' % (a_optimal, n_optimal, b_optimal))
```

The output of this code is:

```
The optimal values in the parametric nonlinear model (starting from (1, 1, 0)):
 a = 9.575555, n = 0.313148, b = 49.812369
```

We repeat the same optimization procedure but this time we set the starting point as $\{a, n, b\} = \{10, 7, 100\}$:

```
w_optimal_parametric_, _ = curve_fit(parametric_nonlinear_model,
                                     X_raw, y, p0=[10, 7, 100])
a_optimal_ = w_optimal_parametric[0]
n_optimal_ = w_optimal_parametric[1]
b_optimal_ = w_optimal_parametric[2]
print('The optimal values in the parametric nonlinear model (starting from (10, 7, 100):'
      '\n a = %f, n = %f, b = %f' % (a_optimal_, n_optimal_, b_optimal_))
```

This code produces the output:

```
The optimal values in the parametric nonlinear model (starting from (10, 7, 100):
 a = 9.575555, n = 0.313148, b = 49.812369
```

Looking at two of the outputs above we see that there is no difference in the final values of the parameters. However, this is not the case when the same code is implemented in `Matlab` using the `lsqcurvefit` function. Again, the differences in the implementation details of the libraries make `lsqcurvefit` converge to different values for the two starting points, whereas `Python`'s `curve_fit` converges to the same set of values for both starting points. Since this question is implemented in `Python` through this part, we proceed with the `curve_fit` solution.

Once more, we compute the explained variance, the unexplained variance and the coefficient of determination for our model to judge its performance. The parameters of the final model are set to the values found above. We find the model predictions as follows:

```
pred = parametric_nonlinear_model(X_raw, a_optimal, n_optimal, b_optimal)
```

Then we find the statistics of interest by running the code:

```
e_var, u_var, R2 = test_model(Y, pred)
print('Explained variance of the parametric nonlinear model:', e_var)
print('Unexplained variance of the parametric nonlinear model:', u_var)
print('Coefficient of determination of the parametric nonlinear model:', R2)
```

The code outputs:

```
Explained variance of the parametric nonlinear model: 0.8558746842166239
Unexplained variance of the parametric nonlinear model: 7.358051252216542
Coefficient of determination of the parametric nonlinear model: 85.5874684216624
```

Up till now, we trained tree different models, namely the linear model, the linearized second-order model and the parametric nonlinear model. The R^2 values of these models where 65.79%, 76.93% and 85.59%, respectively. Thus, we can say that the linearized second-order model is better than the linear model and the parametric nonlinear model is the best among all three models in terms of how well the models fit to the given dataset.

The parametric nonlinear model is plotted together with the original data points. The plot is given below:

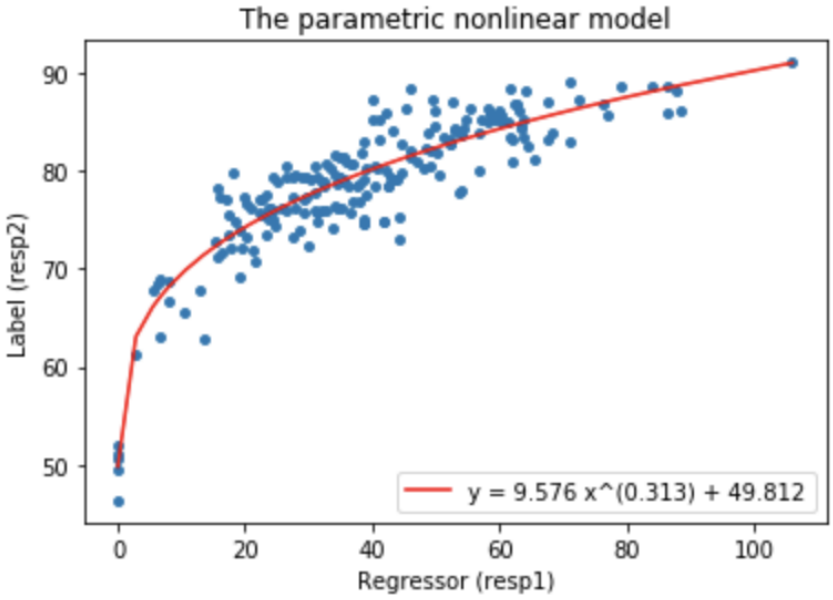

Figure 3: The parametric nonlinear model

The high value of R^2 reflects itself to the plot above, the parametric nonlinear model is clearly the best model so far in terms of capturing the overall shape of the data points.

1.d) As the final alternative we use a model from the family of nonparametric nonlinear models. The model we use is constructed using a method named *Nearest-Neighbor Regression*. The method works as its name implies; in order to make a prediction, we first find the regressor that is closest to the input given to us, then we assign the label of the closest (nearest) regressor to the input and report this label as the prediction.

A `Python` function that implements this procedure is given at the beginning of the next page.

```python
def nearest_neighbor_regression(y, X):
    """
    Performs nonlinear regression based on the nearest neighbor
    approach, meaning that the function predicts the label of the
    closest regressor for each input.
    Args:
        y: The data labels
        X: The regressors
    Returns:
        pred: The predicted valus
    """
    pred = np.zeros(np.size(y))
    for index, x in enumerate(X):
        index_of_nearest = np.abs(X - x).argmin()
        pred[index] = y[index_of_nearest]
    return pred
```

We obtain the predictions of our model by running the line:

```python
pred = nearest_neighbor_regression(y, X_raw)
```

Then we plot the predictions of the model together with the regressors:

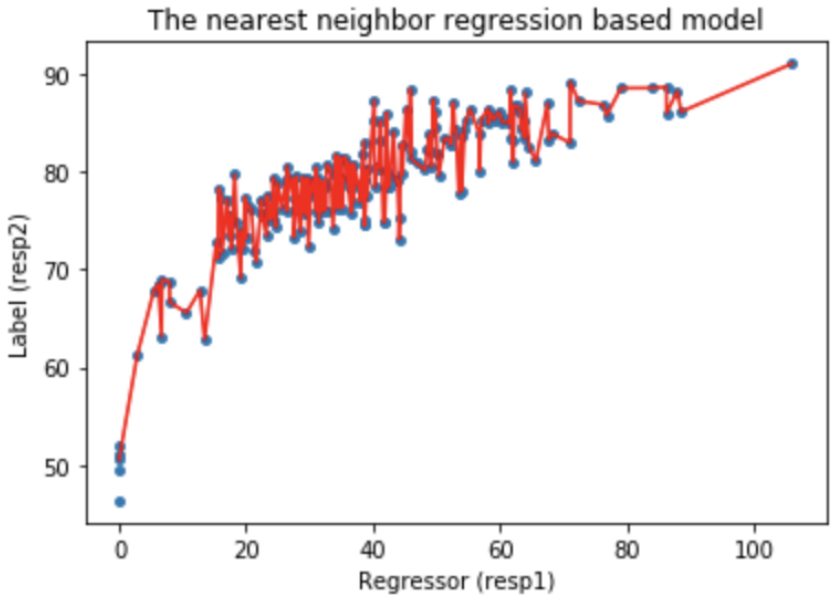

Figure 4: The nonparametric nonlinear model

Looking at the figure, it is clear that this model overfits. The reason is that we do not have a test set and we plotted the predictions of our model on the same data that it is trained on. In other words, the closest regressor to each input in our case is trivially that input itself. Hence, the model assigns the real label of the input instead of a value predicted by a meaningful estimator.

We can observe this overfitting numerically by measuring the explained variance, the unexplained variance and the coefficient of determination:

```
1 e_var, u_var, R2 = test_model(y, pred)
2 print('Explained variance of the nearest neighbor regression based model:', e_var)
3 print('Unexplained variance of the nearest neighbor regression based model:',
      u_var)
4 print('Coefficient of determination nearest neighbor regression based model:', R2)
```

The result are given below:

```
1 Explained variance of the nearest neighbor regression based model:
      0.9979301090423391
2 Unexplained variance of the nearest neighbor regression based model:
      0.10567445191836249
3 Coefficient of determination nearest neighbor regression based model:
      99.79301090423391
```

Notice that we have a very high R^2 value that is almost 100%. The model pretty much explains the whole variance structure of the data. This agrees with the previous explanation in which we stated that the model overfits by predicting the actual labels of the inputs that it is trained on. In practice, one should be suspicious when R^2 values exceed 90%, this is usually an indicator of overfitting.

Question 2

2.a) We conduct a *two-alternative forced choice* (2AFC) experiment, where in each trial a subject views two stimulus arrays of different intensities and determines the array containing the target stimulus. The probability that a subject gives a correct answer in any one of the trials is given by the *psychometric function*, which is:

$$p_c(I) = \frac{1}{2} + \frac{1}{2}\Phi(I; \mu, \sigma) \tag{8}$$

$\Phi(I; \mu, \sigma)$ is the cumulative density of a Gaussian with mean μ and standard deviation σ evaluated at the stimulus intensity I. A `Python` implementation of the function follows:

```
1  def psychometric_function(I, mu, sigma):
2      """
3      Implementation of the psychometric function.
4      Args:
5          I: Intensity value
6          mu: Mean of the normal distribution used
7          sigma: Standard deviation of the normal distribution used.
8      Returns:
9          p: The resulting probability value
10     """
11     return 0.5 + norm.cdf(I, loc=mu, scale=sigma) / 2
```

We plot the psychometric functions for μ, σ equal to 6, 3 and equal to 3, 4; for I $\in$[1 10]:

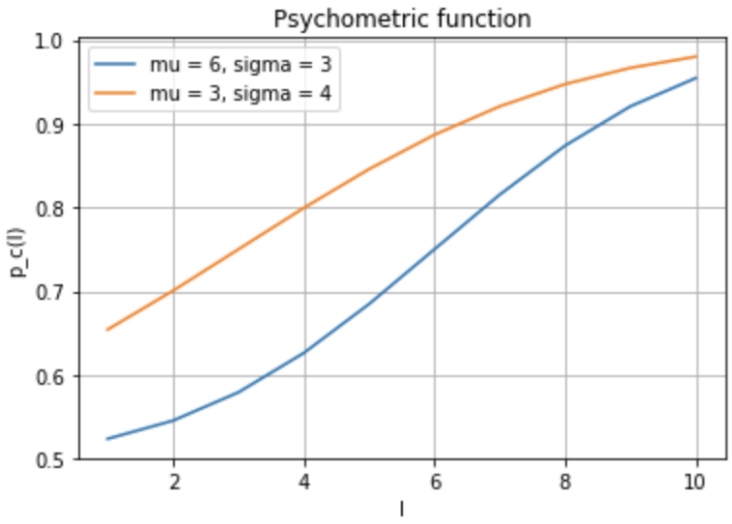

Figure 5: Psychometric function

The plot agrees with the expectations. First of all, we see that the values of the psychometric function never exceeds one. This must be the case since those values are interpreted as success probabilities. Another key observation is that the success probability increases when the stimulus intensity increases. This also makes sense since a subject is likelier to determine the array containing the target stimulus if the target stimulus' intensity is high. Lastly, the value of the psychometric function never goes below 0.5, this tells us that there is a lowerbound of 0.5 on success probability even when the stimulus intensity is very low.

We see from the plot that the psychometric function with the parameters $\{6, 3\}$ stays below the one with the parameters $\{3, 4\}$. This is simply because of the fact that a Gaussian with lower mean will give a larger cumulative density at a certain intensity than the same Gaussian with higher mean. Also, the slope of the function is affected by the σ values.

2.b) We must follow a down to top approach if we want to simulate the experiment. Each trial of the experiment is in fact a Beurnoulli trial, where the success probability is determined by the value of the psychometric function at a certain stimulus intensity. Hence, we implement a function that performs a single trial:

```
def beurnoulli(p):
    """
    Performs a beurnoulli trial with success probability p.
    Args:
        p: The success probability
    Returns:
        outcome: The outcome of the experiment
            (1 means success, 0 means failure)
    """
    rand_val = np.random.rand();
    if rand_val <= p:
        return 1
    return 0
```

A single trial is clearly not enough for our experiment. We have repeated trials, where the outcome of each trial is independent of the others. We assume that the success probability does not change throughout the trials. Thus, the experiment is a Binomial experiment with say n trials and p success probability. The following function simulates such an experiment:

```
def binomial(n, p):
    """
    Performs a binomial experiment that involves n independent
    trial where the success probability of each trial is p.
    Args:
        n: The number of trials
```

```
        p: The success probability of each trial
    Returns:
        success_count: Number of successes
        trials: An array containing the trial results
    """
    trials = [beurnoulli(p) for _ in range(n)]
    return np.count_nonzero(trials), np.array(trials)
```

We now write a function `simpsych` that takes two vectors `I` and `T` of the same length, containing the stimulus intensities and number of trials for each intensity, respectively. The function simulates random draws from $p_c(I)$ and outputs a vector `C` containing the number of trials correct out of `T` at each stimulus intensity `I`, and a matrix `E` containing the trial result at each stimulus intensity and each of `T` trials at that intensity. The function follows:

```
def simpsych(mu, sigma, I, T):
    """
    Simulates random draws using psychometric probabilities. Each draw itself
    is a binomial experiment.
    Args:
        mu: Normal distribution mean that will be given to psychometric function
        sigma: Normal distribution standard deviation that will be given to
            psychometric function
        I: An array containing intensity values
        T: An array containing values for number of trials
    Returns:
        C: An array containing the number of trials correct out of T at each
            stimulus intensity I
        E: A matrix containing the trial result at each stimulus intensity and
            each of T trials at that intensity
    """
    size = np.size(T)
    C = np.zeros(size)
    E = np.zeros((size, int(np.max(T))))
    for i in range(size):
        C[i], E[i] = binomial(int(T[i]), psychometric_function(I[i], mu, sigma))
    return C, E
```

We run the function for `mu = 5`, `sigma = 1.5`, `I = np.arange(1, 8)` and `T = np.ones(7) * 100`, and obtain the corresponding `C` vector and `E` matrix. Then, we show a scatter plot of `C/T` versus `I`, and a curve plot of the respective psychometric function on the same graph. The resulting figure is displayed at the beginning of the next page.

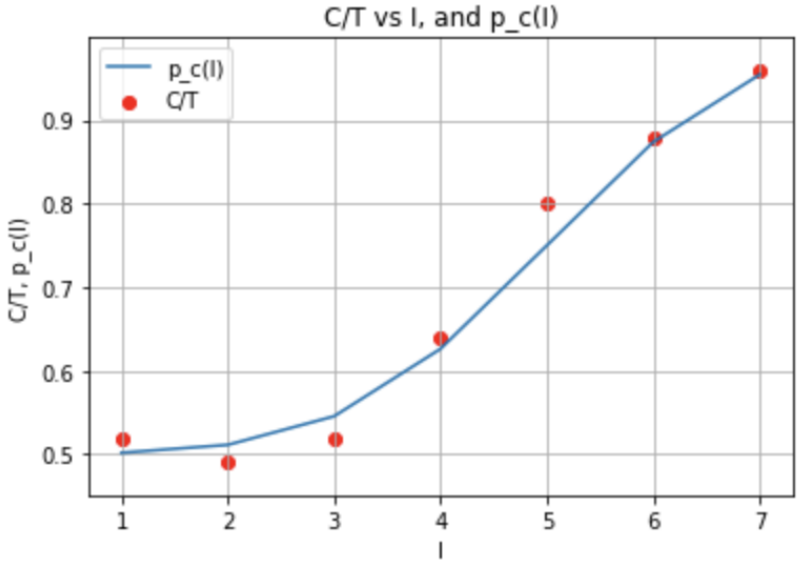

Figure 6: `C/T` and psychometric function versus `I`

It can be seen that the points representing the `C/T` values align with the psychometric function curve. The reason for this is that the expected value of $C \sim Bin(T, p_c(I))$ is $T \cdot p_c(I)$ and it roughly equals the success count `C`. Then, `C/T` gives:

$$\texttt{C/T} \simeq E[C/T] = \frac{E[C]}{T} = \frac{T \cdot p_c(I)}{T} = p_c(I)$$

2.c) Now we are interested in the likelihood that we observe the data we obtained in the previous part (vector `C` and matrix `E`). For this task, we implement a function `nloglik` that returns the negative log-likelihood that we observe the data we obtained given the Gaussian parameters μ and $sigma$. Before implementing the function, we derive an expression for the negative log-likelihood. We have:

$$p_c(I)_i = \frac{1}{2} + \frac{1}{2}\Phi(I_i; \mu, \sigma)$$

The subscript i denotes the ith indexed values in the argument arrays of the `simpsych` function. Likelihood can be written as follows:

$$L(\mu, \sigma) = \prod_i [p_c(I)_i^{C_i} \cdot (1 - p_c(I)_i)^{(T_i - C_i)}] \tag{9}$$

Multiplication usually introduces numeric problems in scientific computing, we take the natural logarithm of the likelihood to change the multiplications in (9) to additions.

$$ln(L(\mu,\sigma)) = \sum_i [C_i \cdot ln(p_c(I)_i) + (T_i - C_i) \cdot ln(1 - p_c(I)_i)]$$

Our objective will be maximizing the likelihood, which is equal to minimizing the negative of it. For convenience we chose minimization over maximization and write:

$$-ln(L(\mu,\sigma)) = -\sum_i [C_i \cdot ln(p_c(I)_i) + (T_i - C_i) \cdot ln(1 - p_c(I)_i)] \tag{10}$$

The function `nloglik` just computes the numeric value of (10) for its inputs. Its implementation is given below:

```
1  def nloglik(pp, I, T, C):
2      """
3      Computes and return the negative log-likelihood that we observe
4      a certain data.
5      Args:
6          pp: A size 2 array that contains the mean and the standard
7              deviation of the normal distribution that will be given to
8              the psychometric function
9          I: An array containing intensity values
10         T: An array containing values for number of trials
11         C: An array containing the number of trials correct out of T at each
12             stimulus intensity I
13     Returns:
14         nll: The negative log-likelihood that we observe the argument data
15     """
16     mu = pp[0]
17     sigma = pp[1]
18     size = np.size(T)
19     ll = 0
20     for i in range(size):
21         p_i = psychometric_function(I[i], mu, sigma)
22         ll += C[i] * np.log(p_i) + (T[i] - C[i]) * np.log(1 - p_i)
23     return -ll
```

We draw a contour plot of `nloglik` values for μ values `mu_est = np.arange(2, 8.1, 0.1)`, sigma values σ `sigma_est = np.arange(0.5, 4.6, 0.1)` and `I, T, C` values obtained in the previous part. The figure on the next page shows the contour with 50 levels.

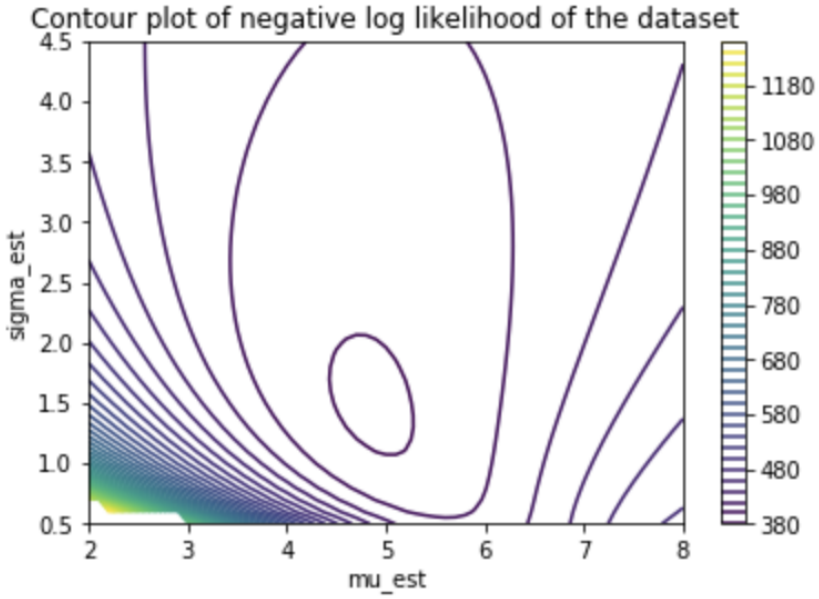

Figure 7: Contour plot of `nloglik` values

The best fitting (minimizing) parameters will roughly be $\mu = 5$ and $\sigma = 1.5$ by visual inspection on the contour plot.

2.d) To find close estimates to the exact values of the minimizing parameters we can use the `fmin` function in `scipy`'s `optimize` module. To find the best fitting parameters we run the following code segment:

```
mu_best, sigma_best = fmin(func=lambda pp: nloglik(pp, I, T, C), x0=[2, 2])
print('Best mu_est:', mu_best)
print('Best sigma_est:', sigma_best)
```

The code gives the minimum `nloglik` value and the best fitting parameters by printing:

```
Optimization terminated successfully.
        Current function value: 377.354195
        Iterations: 44
        Function evaluations: 85
Best mu_est: 4.8630993962982325
Best sigma_est: 1.471239681344411
```

2.e) At this point, we want to find confidence intervals for the best fitting parameter estimates we obtained in the previous part. To do this, we resample the matrix `E` that contains the experiment results using bootstrapping. The function given below is used to compute a single resample of `E`:

```
def resample_matrix(matrix):
    """
    Resamples a matrix using bootstrapping on each row.
    Args:
        matrix: The matrix that will be resampled
    Returns:
        resample_matrix: The resampled matrix
    """
    num_rows, num_cols = np.shape(matrix)
    resampled_matrix = np.zeros((num_rows, num_cols))
    indices = np.arange(num_cols)
    for i in range(num_rows):
        bootstrap_indices = np.random.choice(indices, num_cols)
        resampled_matrix[i] = matrix[i][bootstrap_indices]
    return resampled_matrix
```

200 resamples are enough for accurate results. For each resample of the experiment matrix we build our probabilistic model `nloglik` and find the best fitting parameters using `fmin`. The following script performs the required procedure:

```
NUM_RESAMPLES = 200
mu_resamples = []
sigma_resamples = []
np.random.seed(7) # to be able to reproduce results
for i in range(NUM_RESAMPLES):
    E_resample = resample_matrix(E)
    C_resample = np.sum(E_resample, axis=1)
    mu_resample, sigma_resample = fmin(
        func=lambda pp: nloglik(pp, I, T, C_resample), x0=[2, 2], disp=False)
    mu_resamples.append(mu_resample)
    sigma_resamples.append(sigma_resample)
```

We now have sampling distributions for best fitting μ and σ values, each of size 200. The histograms on the next page show these distributions.

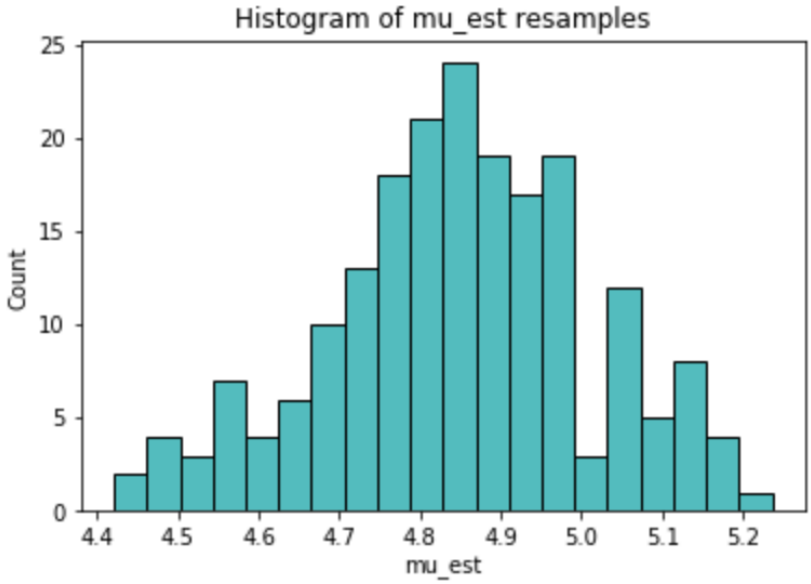

Figure 8: Sampling distribution of best fitting μ values

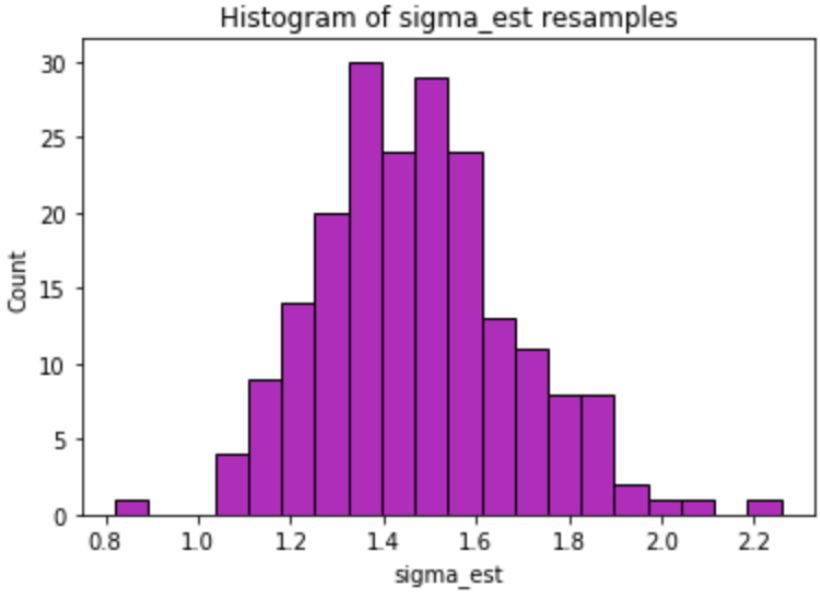

Figure 9: Sampling distribution of best fitting σ values

Using the sampling distributions we compute the 95% confidence interval for both of the parameters. The function given below computes the lower and upper end of the confidence interval of a variable:

```
def compute_confidence_interval(data, confidence):
    """
    Given the data and the confidence level, computes the confidence interval
    of the data samples.
    Args:
        data: The given data
        confidence: The confidence level, known as alpha (between 0 and 100)
    Returns:
        lower: The lowerbound of the confidence interval
        upper: The upperbound of the confidence interval
    """
    sorted_data = np.sort(data)
    lower = np.percentile(sorted_data, (100 - confidence) / 2)
    upper = np.percentile(sorted_data, confidence + (100 - confidence) / 2)
    return lower, upper
```

We run the following code to compute the required intervals:

```
mu_lower, mu_upper = compute_confidence_interval(mu_resamples, 95)
print('95%% Confidence interval for mu_est: (%1.5f, %1.5f)' % (mu_lower, mu_upper)
    )
sigma_lower, sigma_upper = compute_confidence_interval(sigma_resamples, 95)
print('95%% Confidence interval for sigma_est: (%1.5f, %1.5f)' % (sigma_lower,
    sigma_upper))
```

The resulting 95% confidence intervals for the best fitting parameters are:

```
95% Confidence interval for mu_est: (4.49845, 5.15409)
95% Confidence interval for sigma_est: (1.12135, 1.88954)
```

Question 3

Data given in this question contains the Blood-oxygen level dependent (BOLD) responses of a neural population in the human visual cortex. The array `Yn` includes 1000 response samples, whereas `Xn` includes 1000 input samples that contain 100 features each.

3.a) We use *Ridge Regression* to model `Yn` as a weighted sum of the features given in `Xn`. Ridge regression is a natural extension of *Ordinary Least Squares* method, which introduces an additional regularization term to the least squares cost to reduce overfitting. The formulation of Ridge regression follows:

$$\text{Let } X \text{ be the input matrix } \implies X_{NxK} = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_N^T \end{bmatrix}$$

$$\text{Let } y \text{ be the output vector } \implies y_{Nx1} = \begin{bmatrix} y_1 & y_2 & \dots & y_N \end{bmatrix}^T$$

$$\text{Let } w \text{ be the weight vector } \implies w_{Kx1} = \begin{bmatrix} w_1 & w_2 & \dots & w_K \end{bmatrix}^T$$

We add the regularization term $\frac{\lambda}{2}\|w\|_2^2$ to the cost function used in OLS to constraint the L2 norm of w and enforce simpler models that have a lower chance of overfitting. The loss function becomes:

$$L(w) \coloneqq \frac{1}{2}\sum_{n=1}^{N}(x_n^T \cdot w - y_n)^2 + \frac{\lambda}{2}\sum_{k=1}^{K} w_k^2 \tag{11}$$

The regularization term is convex in w; since the rest is already a convex function in w and addition of two convex functions is convex, we know that $L(w)$ is convex in w. Thus we take the gradient of $L(w)$ with respect to w and set it to zero, which will lead us to the optimal value of w:

$$\underset{w}{\nabla} L(w) = \begin{bmatrix} \frac{\partial L(w)}{\partial w_1} \\ \frac{\partial L(w)}{\partial w_2} \\ \vdots \\ \frac{\partial L(w)}{\partial w_k} \end{bmatrix} = \begin{bmatrix} \sum_{n=1}^{N}(x_n^T \cdot w - y_n) \cdot x_{n1} + \lambda w_1 \\ \sum_{n=1}^{N}(x_n^T \cdot w - y_n) \cdot x_{n2} + \lambda w_2 \\ \vdots \\ \sum_{n=1}^{N}(x_n^T \cdot w - y_n) \cdot x_{nK} + \lambda w_K \end{bmatrix} = X^T \cdot (X \cdot w - y) + \lambda w$$

Optimal w, call it $w^\star$ should satisfy $\nabla L(w^\star) = 0$

We derive $w^\star$ as follows:

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

Thus:

$$w^\star = (X^T X + \lambda I_K)^{-1} X^T \cdot y \tag{12}$$

Notice that we now have $(X^T X + \lambda I_K)^{-1}$ rather than $(X^T X)^{-1}$, which we had in OLS. This is very good since $X^T X$ can be hard to invert (ill-conditioned) or even not invertible. Here, $X^T X + \lambda I_K$ is positive definite for $\lambda > 0$, hence it is always invertible.

A `Python` function that implements the Ridge regression formulation is given below:

```
1  def ridge(y, X, lambda_):
2      """
3      Given data labels and regressors, learns an optimal weight
4      vector according to the ridge regression formulation.
5      Args:
6          y: The data labels
7          X: The regressors
8          lambda_: The regularization parameter
9      Returns
10         w_optimal: The optimal weight vector
11     """
12     K = np.shape(X)[1]
13     temp = np.linalg.inv(X.T.dot(X) + lambda_ * np.eye(K))
14     w_optimal = temp.dot(X.T).dot(y)
15     return w_optimal
```

As we did in the first question, we will again use the coefficient of determination R^2 to measure the performance of the models that we are going to fit to the dataset. R^2 will be computed as the square of the pearson correlation coefficient between the model predictions and the outputs since we will only consider linear models. A function that performs the R^2 computation follows:

```
1  def compute_R2(Y, pred):
2      """
```

```
    Tests a given linearized model by computing the coefficient
    of determination (R^2). R^2 is computed as the square of the
    Pearson correlation between the labels and the predictions.
    Args:
        Y: The data labels
        pred: The predicted valus
    Returns:
        R2: The coefficient of determination
    """
    pearson = np.corrcoef(Y, pred)[0, 1]
    R2 = pearson ** 2
    return R2
```

We use *cross validation* to ensure that our model is stable, meaning that it does not overfit to the training set. Cross validation is a procedure in which we iteratively train and test our model over varying partitions of training and test set. Due to the increased number of training and test sets, we become able to better estimate the model's true error and hence prevent overfitting to a particular partition of the dataset. A function that performs this procedure follows:

```
def cross_validation(y, X, k_fold, lambda_arr):
    """
    Performs k fold cross validation with three way split in each
    iteration. The aim is to tune the ridge regression's regularizer,
    lambda. Hence each value in an array of lambda values is integrated
    into the model and coefficient of determination (R^2) is calculated
    for each case.
    Args:
        y: The data labels
        X: The regressors
        k_fold: Number of folds in cross validation
        lambda_arr: The regularization parameters to select from
    Returns:
        dict_valid: The R^2 values calculated in the validation
            set for each fold and lambda value, stored as a dictionary.
        dict_test: The R^2 values calculated in the test set
            for each fold and lambda value, stored as a dictionary.
    """
    N = np.size(y)
    idx_unit = int(N / k_fold)
    dict_valid = dict()
    dict_test = dict()
    for i in range(k_fold):
        valid_start = i * idx_unit
```

```
        test_start = (i + 1) * idx_unit
        train_start = (i + 2) * idx_unit
        valid_indices = np.arange(valid_start, test_start) % N
        test_indices = np.arange(test_start, train_start) % N
        train_indices = np.arange(train_start, N + valid_start) % N
        y_valid = y[valid_indices]
        X_valid = X[valid_indices]
        y_test = y[test_indices]
        X_test = X[test_indices]
        y_train = y[train_indices]
        X_train = X[train_indices]
        for lambda_ in lambda_arr:
            w = ridge(y_train, X_train, lambda_)
            dict_valid.setdefault(lambda_, []).append(compute_R2(y_valid, X_valid.
                dot(w)))
            dict_test.setdefault(lambda_, []).append(compute_R2(y_test, X_test.dot(
                w)))
    dict_valid = dict((k, np.mean(v)) for k, v in dict_valid.items())
    dict_test = dict((k, np.mean(v)) for k, v in dict_test.items())
    return dict_valid, dict_test
```

The function above implements k-fold cross validation, meaning k different partitions of the dataset are used as training sets and test sets. The function also introduces *validation sets*, which serves to the purpose of selecting from the trained models. Note that the trained models can also be selected using the test set, however this may result in a biased model since the selection itself may help overfitting to the test set by favoring its characteristics.

We want to find the optimal regularization parameter λ_{opt} using the function above. We are given that the λ values should be in the range $[010^12]$. We run a 10 fold cross validation on this range of values to find the optimal lambda (the one that maximizes R^2 on the validation set):

```
K_FOLD = 10
lambda_arr = np.logspace(0, 12, num=500, base=10)
# Takes about a minute to execute
dict_valid, dict_test = cross_validation(Yn, Xn, K_FOLD, lambda_arr)
lambda_optimal = max(dict_valid, key=lambda k: dict_valid[k])
print('Optimal lambda:', lambda_optimal,
      '\nCorresponding R^2 in validation set:', dict_valid[lambda_optimal],
      '\nCorresponding R^2 in test set:', dict_test[lambda_optimal])
```

We get the following output when the previous script runs:

```
Optimal lambda: 395.5436244734702
Corresponding R^2 in validation set: 0.15259887784859996
Corresponding R^2 in test set: 0.16042061044928463
```

For better interpretations, we plot R^2 curves obtained on testing and validation data for all λ values (note that x axis is logarithmic):

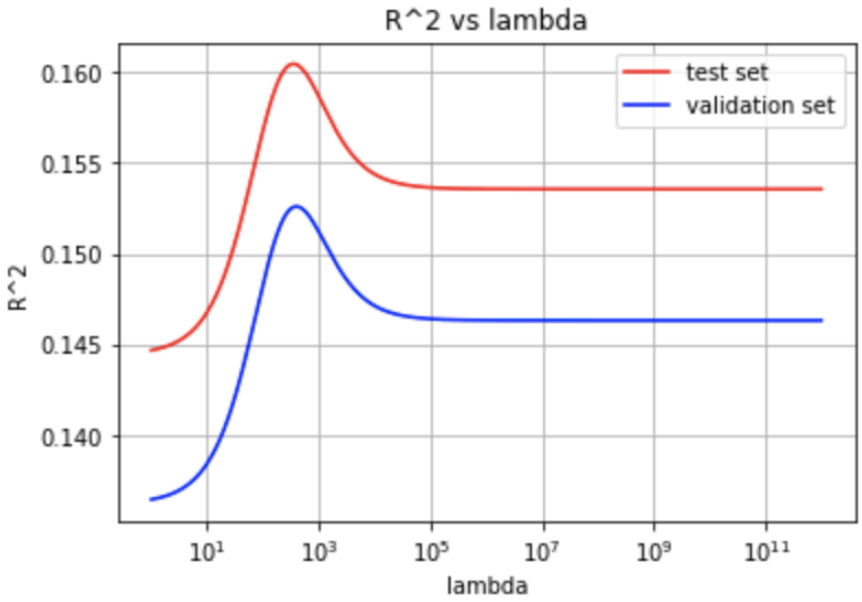

Figure 10: R^2 vs λ

We can see that both of the R^2 curves are concave in λ in the given domain. It means that we have a global maximum in the given domain, which is the optimal lambda reported above.

Interestingly, R^2 curve of the test set is above that of the validation set. This is a rare case in practice, however it is not impossible since we did not use the test set in any phase of the training procedure. It can be thought as a dataset that the model meets for the first time. Hence, it is hard to make comments about the performance of the model on the test set.

3.b) This part requires us to determine 95% confidence intervals for parameters (model weights) for the OLS model from the first part. Recall that the OLS model can be considered as a specific type of Ridge model where $\lambda = 0$.

Since we do not have that much data available, we use bootstrapping to generate more. We identically sample from `Yn` and `Xn` to generate 500 input and output resamples in total. We fit the OLS model to each pair of input, output resamples to obtain 500 sets of model parameters. We plot the mean of each parameter and draw the 95% confidence interval. Since 95% confidence interval corresponds to $\pm$ 2 standard deviations from the center, we compute the standard deviation for each parameter and draw error bars that represent the confidence intervals accordingly. The following code is used to carry out the procedure:

```
NUM_ITER = 500 # number of bootstrap iterations
N = np.size(Yn) # number of samples
np.random.seed(7) # to be able to reproduce the results
w_bootstrap_OLS = []
for _ in range(NUM_ITER):
    # draw N samples with replacement
    indices = np.arange(N)
    bootstrap_indices = np.random.choice(indices, N)
    # genereate the respective bootstrap labels and regressors
    y_bootstrap = Yn[bootstrap_indices]
    X_bootstrap = Xn[bootstrap_indices]
    w_OLS = ridge(y_bootstrap, X_bootstrap, 0)
    w_bootstrap_OLS.append(w_OLS)
w_bootstrap_OLS = np.array(w_bootstrap_OLS).T # now rows indicate regressors
x_vals = np.arange(1, 101)
w_mean_OLS = np.mean(w_bootstrap_OLS, axis=1)
w_std_OLS = np.std(w_bootstrap_OLS, axis=1)
```

After the code above runs and populates the necessary arrays, the code below should run to do the plotting:

```
plt.figure(figure_num)
figure_num += 1
plt.errorbar(x_vals, w_mean_OLS, yerr=2 * w_std_OLS, ecolor='r',
             elinewidth=0.5, capsize=2)
plt.title('Model weights, w (OLS)')
plt.xlabel('i')
plt.ylabel('value of w_i')
plt.show(block=False)
```

The resulting figure is shown below:

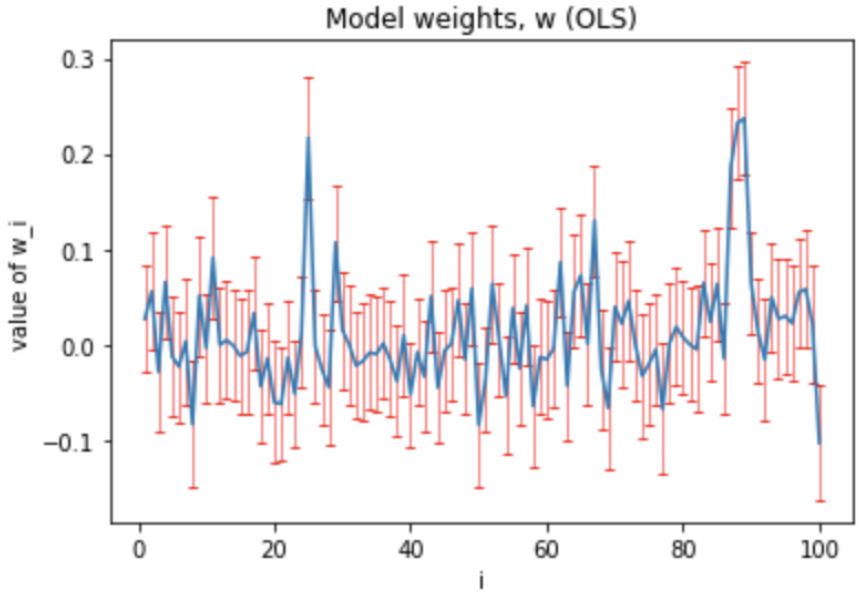

Figure 11: Model weights, w (OLS)

We also want to know which parameters are significantly different than 0 (at a significance level of $p < 0.05$). For this we perform a small hypothesis testing procedure for each parameter. Hypothesis testing is explained in detail in the last section of the report. Thus, the lengthy explanation is omitted here and the code that computes the significant parameter indices is given:

```
z_vals = w_mean_OLS / w_std_OLS
p_vals = 2 * (1 - norm.cdf(np.abs(z_vals)))
significant_OLS = np.argwhere(p_vals < 0.05).flatten()
print('Indices of the parameters that are significantly different than 0:\n',
      significant_OLS)
```

The code produces the output:

```
Indices of the parameters that are significantly different than 0:
 [ 3 7 10 20 24 28 48 49 51 57 61 64 66 68 76 82 84 86 87 88 89 99]
```

3.c) Now, we repeat the exact same procedure we have carried out in the previous part. However, this time we fit the Ridge model with λ_{opt} (found in the first part) to each bootstrap resample. The following code performs the calculations:

```
np.random.seed(7) # to be able to reproduce the results
w_bootstrap_ridge = []
for _ in range(NUM_ITER):
    # draw N samples with replacement
    indices = np.arange(N)
    bootstrap_indices = np.random.choice(indices, N)
    # genereate the respective bootstrap labels and regressors
    y_bootstrap = Yn[bootstrap_indices]
    X_bootstrap = Xn[bootstrap_indices]
    w_ridge = ridge(y_bootstrap, X_bootstrap, lambda_optimal)
    w_bootstrap_ridge.append(w_ridge)
w_bootstrap_ridge = np.array(w_bootstrap_ridge).T # now rows indicate regressors
w_mean_ridge = np.mean(w_bootstrap_ridge, axis=1)
w_std_ridge = np.std(w_bootstrap_ridge, axis=1)
```

After the arrays required for the plot is populated by the code above, the following figure is generated:

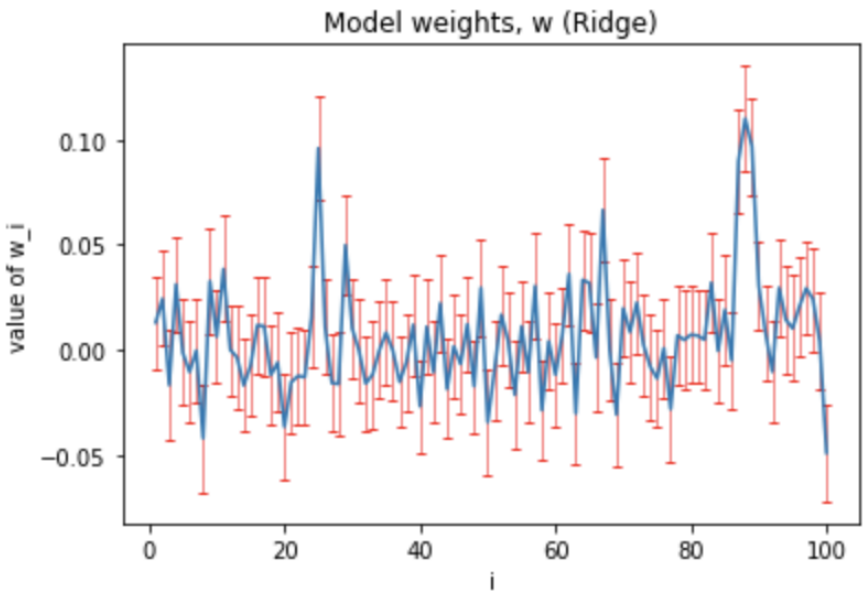

Figure 12: Model weights, w (Ridge)

Parameters that are significantly different than 0 are computes using the code:

```
z_vals = w_mean_ridge / w_std_ridge
p_vals = 2 * (1 - norm.cdf(np.abs(z_vals)))
significant_ridge = np.argwhere(p_vals < 0.05).flatten()
print('Indices of the parameters that are significantly different than 0:\n',
      significant_ridge)
```

Th result is:

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

Remember that the regularization term in Ridge Regression forces the model weights (parameters) to have smaller L2 norms. This in turn makes the model parameters closer to 0. However, the regularization term also increases the standard deviation of the parameters since it is an added complexity to the model. That is the main reason why we have more parameters that are significantly different than 0 when we incorporate the requlrization term into the OLS model.

Question 4

4.a) In this part of the question, we are provided with responses from two separate populations of neurons. The responses of the first population is stored in the `pop1` array, whereas the responses of the second is stored in the `pop2` array. First population contains 7 neurons and the second contains 5 neurons, so `pop1` is of size 7 and `pop2` is of size 5.

We want to determine whether the mean response of these populations differ significantly or not. For that we use *Hypothesis Testing* and build up the assumption that the values in `pop1` and `pop2` came from the same population. This assumption leads us to the null hypothesis, which is denoted by H_0:

$$H_0 = \mu_1 = \mu_2 \iff H_0 = \mu_1 - \mu_2 = x = 0 \tag{13}$$

In words, null hypothesis is what we assume about our data. Here, the null hypothesis states that the mean response of the first population μ_1 is equal to that of the second population μ_2 since both populations are assumed to come from the same population. It is conventionally required to define the alternative hypothesis, denoted by H_A, which is the exact opposite of the null hypothesis:

$$H_A = \mu_1 \neq \mu_2 \iff H_0 = \mu_1 - \mu_2 = x \neq 0 \tag{14}$$

What follows is to determine when to reject and when to accept the null hypothesis, for that we need $P(x|H_0)$.

Since, `pop1` and `pop2` contain only one sample for each neuron, we use bootstrapping to generate more samples. A `Python` function is implemented to perform this bootstrapping operation to make future steps easier, the implementation follows:

```
1  def bootstrap(arr, num_samples, seed=7):
2      """
3      Resamples an array using the bootstrap technique.
4      Args:
5          arr: The array that will be resampled
6          num_samples: Number of samples that will be generated
7          seed: The random seed to be able to reproduce the results
8      Returns:
9          arr_bootstrap: Numpy array containing the new samples
10     """
11     arr_bootstrap = []
12     arr_size = np.size(arr)
13     indices = np.arange(arr_size)
14     np.random.seed(seed) # to be able to reproduce the results
15     for _ in range(num_samples):
16         bootstrap_indices = np.random.choice(indices, arr_size)
```

```
17          arr_resample = arr[bootstrap_indices]
18          arr_bootstrap.append(arr_resample)
19      return np.array(arr_bootstrap)
```

x is the difference of the means of the two populations but we are interested in the distribution of x given the null hypothesis. Thus:

- We combine the `pop1` and `pop2` arrays into one, the combined array includes responses of $7 + 5 = 12$ neurons.
- We generate 10000 samples from the combined array using bootstrapping
- We randomly select 7 values from each sample and interpret the values as first populations' responses. Similarly we interpret the remaining 5 values of each sample as second populations' responses.
- We compute the mean of the 7 values selected and assigned to the first population for each sample, we do the same for the 5 values assigned to the second population.
- We compute the difference of the means calculated for the first and the second population for each bootstrap sample.

What we end up with after the above steps is 10000 x values obtained under the assumption that the null hypothesis is correct. We can then discretize these values and create an histogram with density normalization to approximate $P(x|H)$. A `Python` function that performs these steps is given below:

```
1  def difference_in_means(arr1, arr2, num_samples, bins=60, seed=7):
2      """
3      Generates a sampling distribution of the difference
4      in means of two individual distributions. Uses bootstrapping
5      to generate samples from the combined distribution.
6      Args:
7          arr1: The first distribution
8          arr2: The second distribution
9          num_samples: Number of samples to generate when computing
10             the sampling distribution
11         bins: Number of bins in the discretized distribution
12         seed: The random seed to be able to reproduce the results
13     Returns:
14         diff_in_means: The sampling distribution of the difference
15             in means of the given distributions
16         vals: The discretized interval of values
17         probs: The probabilities that the discretized interval of values
18             can be seen
```

```
    """
    arr = np.concatenate((arr1, arr2))
    arr_bootstrap = bootstrap(arr, num_samples, seed)
    samples1 = arr_bootstrap[:, :np.size(arr1)]
    samples2 = arr_bootstrap[:, np.size(arr1):]
    means1 = np.mean(samples1, axis=1)
    means2 = np.mean(samples2, axis=1)
    diff_in_means = means1 - means2
    probs, vals = np.histogram(diff_in_means, bins=bins, density=True)
    return diff_in_means, vals, probs
```

We run the function and store the results in the following arrays, where `NUM_SAMPLES` is 10000:

```
diff_in_means, vals, probs = difference_in_means(pop1, pop2, NUM_SAMPLES)
```

The figure below is created by plotting the discretized distribution $P(x|H_0)$ obtained above:

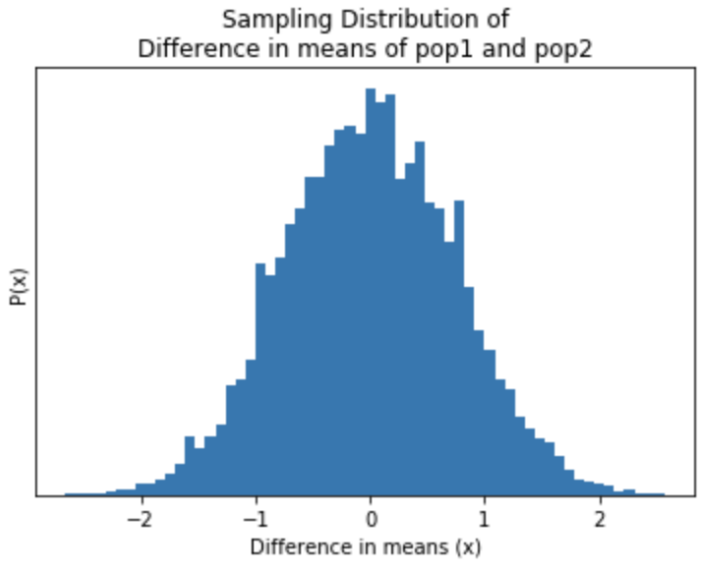

Figure 13: Sampling distribution of difference of means of `pop1` and `pop2`

Now we can find the probability that the null hypothesis is true given `pop1` and `pop2`. The actual difference of means of `pop1` and `pop2` can be computed by the statement "`x_overline = np.mean(pop1) - np.mean(pop2)`" and denoted by $\overline{x}$. We are interested in the probability $P(|x| > \overline{x}|H_0)$ because we want to know how probable it is for the difference of means to fall outside the observed value $\overline{x}$ under the null hypothesis. We take the absolute value of x since the order in which the difference of means is computed is not important.

$P(x|H_0)$ looks like it is normally distributed. Hence, to calculate the probability $P(|x| > \overline{x}|H_0)$, we perform the following normalization:

$$P(|x| > \overline{x}|H_0) = P(|Z| > \frac{\overline{x} - \mu_0}{\sigma_0}|H_0) \tag{15}$$

In (15), Z is the standard normal, μ_0 is the mean of the sampling distribution, is also called the null distribution, and σ_0 is the standard deviation of the sampling distribution. In fact, we define *z-value* as:

$$z = \frac{\overline{x} - \mu_0}{\sigma_0} \tag{16}$$

Then, the probability of interest becomes $P(|Z| > z|H_0)$ and we define the *two sided p-value* as:

$$p = P(|Z| > z|H_0) = 2 * P(Z > z|H_0) = 2 * (1 - P(Z \leq z|H_0)) \tag{17}$$

It is easy to compute the z-value and p-value can be calculated using the cumulative distribution of the standard normal random variable Z. The following code does the calculations:

```
x_overline = np.mean(pop1) - np.mean(pop2)
sigma_0 = np.std(diff_in_means)
mu_0 = np.mean(diff_in_means)
z = (x_overline - mu_0) / sigma_0
p = 2 * (1 - norm.cdf(z))
print('The z-value is:', z)
print('The two sided p-value is:', p)
```

The code prints:

```
The z-value is: 2.5328129326156494
The two sided p-value is: 0.0113151320630549
```

We have a quite small two sided p-value. Thus, we can say that the probability that our null hypothesis is correct is very small. In conclusion, `pop1` and `pop2` are samples from different populations.

4.b) In this part we are given Blood-oxygen level dependent (BOLD) responses recorded in two voxels in the human brain, the data is stored in the arrays `vox1` and `vox2` respectively. We are interested in the similarity between `vox1` and `vox2`, so we measure the correlation between them in terms of Pearson correlation coefficient.

`vox1` and `vox2` contains only a single response vector of size 50, so there is a need for more samples. We generate more samples, again, using the `bootstrap` function. The bootstrap samples are obtained and stored as follows:

```
vox1_bootstrap = bootstrap(vox1, NUM_SAMPLES)
vox2_bootstrap = bootstrap(vox2, NUM_SAMPLES)
```

NUM_SAMPLES is again 10000. Now, we calculate the correlation value for each of these 10000 response vectors. The code below performs the calculation:

```
corr_bootstrap = np.zeros(NUM_SAMPLES)
for i in range(NUM_SAMPLES):
    corr_bootstrap[i] = np.corrcoef(vox1_bootstrap[i],
                                    vox2_bootstrap[i])[0, 1]
```

We compute some other statistics over the calculated correlation values stored in the corr_bootstrap array. These statistics are the mean and the 95% confidence interval. The following code runs to compute the statistics:

```
corr_mean = np.mean(corr_bootstrap)
corr_lower, corr_upper = compute_confidence_interval(corr_bootstrap, 95)
print('Mean correlation value:', corr_mean)
print('95%% confidence interval of the correlation values: (%1.5f, %1.5f)' %
      (corr_lower, corr_upper))
```

The output is:

```
Mean correlation value: 0.5575702100845678
95% confidence interval of the correlation values: (0.32057, 0.75761)
```

The compute_confidence_interval function used in the calculations simply sorts the data, looks at the appropriate percentiles and returns the endpoints of the confidence interval.

We are also interested in the percentage of correlation values that are 0, if this percentage is high then we can say that most of the samples are uncorrelated. We find this value as follows:

```
corr_zero_percentage = 100 * np.size(np.where(np.isclose(corr_bootstrap, 0))) /
    NUM_SAMPLES
print('Percentage of zero correlation values:', corr_zero_percentage)
```

The code prints:

```
Percentage of zero correlation values: 0.0
```

Here, we see that there are no 0 valued correlations in the corr_bootstrap array. This implies none of our bootstrap samples are uncorrelated. However, we must be careful about

our resampling procedure. Our resampling procedure is some kind of *identical* sampling. When we sample from `vox1` and `vox2` identically:

- We first choose the indices from the range of integers from 0 to 9999 (with replacement).
- We then select the values corresponding to the chosen indices from `vox1` and `vox2`.

Since the chosen indices are the same for both `vox1` and `vox2` (seed value is the same (7) for the two calls to the `bootstrap` method), we force some kind of correlation between the newly generated samples. This is one reason why we had no zero valued correlation between our samples.

4.c) This time, we construct a null hypothesis which states that the correlation values between the bootstrap samples of `vox1` and `vox2` are less than zero. Formally:

$$H_0 = \text{Two voxel responses have zero or negative correlation.}$$

$$H_A = \text{Two voxel responses have positive correlation.}$$

Let y denote the correlation values, we want to be able to calculate $P(y|H_0)$. We need to sample from `vox1` and `vox2` in a way that the correlation enforced by identical sampling is no longer a problem. For that, we use *independent* sampling. In *independent* sampling the indices we chose to select values from `vox1` and `vox2` can possibly be different. We can independently sample from the arrays by using different seed values in our `bootstrap` calls. Our resamples can now be obtained as follows:

```
vox1_independent = bootstrap(vox1, NUM_SAMPLES, seed=17)
vox2_independent = bootstrap(vox2, NUM_SAMPLES, seed=51)
```

Then the correlation values y are computed:

```
y = np.zeros(NUM_SAMPLES)
for i in range(NUM_SAMPLES):
    y[i] = np.corrcoef(vox1_independent [i], vox2_independent [i])[0, 1]
```

The probability that the two voxel responses have zero or negative correlation is asked under the null hypothesis. Thus, we define the *one side p-value*:

$$p = P(y > \overline{y}|H_0) = P(Z > \frac{\overline{y} - \mu_0}{\sigma_0}|H_0) = P(Z > z|H_0) \tag{18}$$

In (18), μ_0 and σ_0 are again the mean and standard deviation of the sampling distribution respectively, also $\overline{y}$ is the actual correlation between `vox1` and `vox2` themselves. Z is standard normal as it was before.

We calculate the one sided p-value following the same steps we have followed in part **a**. The following code calculates the z-value and the one sided p-value:

```
y_overline = np.corrcoef(vox1, vox2)[0, 1]
sigma_0 = np.std(y)
mu_0 = np.mean(y)
z = (y_overline - mu_0) / sigma_0
p = 1 - norm.cdf(z)
print('The z-value is:', z)
print('The one sided p-value is:', p)
```

The output of the code is as follows:

```
The z-value is: 3.984898491468329
The one sided p-value is: 3.375448908071732e-05
```

The one sided p-value is very small, which implies that our null hypothesis is incorrect. The alternative hypothesis stating that the voxel responses have positive correlation is in fact correct. This result agrees with part **b** since the 95% confidence interval found for the correlation values clearly indicates a positive correlation between the voxel responses.

The sampling distribution of correlation between `vox1` and `vox2` is provided in the figure below for clearity:

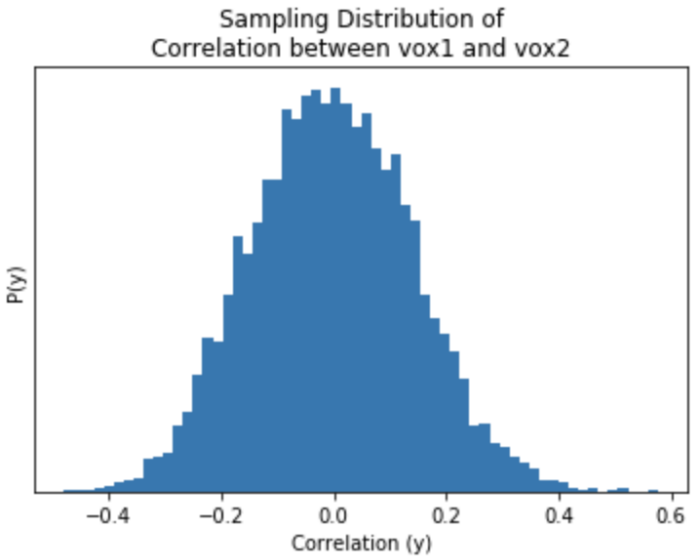

Figure 14: Sampling distribution of correlation between `vox1` and `vox2`

4.d) We are given two arrays `building` and `face` that contains the average BOLD responses in a face-selective brain region to building images (1st experiment) and face images (2nd experiment) for 20 subjects. We build up the hypothesis:

$$H_0 = \text{There is no difference between the building and face responses}$$

$$H_A = \text{There is a difference between the building and face responses}$$

We assume that the subjects are the same for both experiments and rewrite the hypotheses as:

$$H_0 = \mu_1 - \mu_2 = x = 0$$
$$H_1 = \mu_1 - \mu_2 = x \neq 0$$

μ_1 is the average BOLD response to building images and μ_2 is the average BOLD response to face images. Obviously μ_1 should be equal to μ_2 if there is no difference between the responses.

We resample the data and compute the difference of means, x. Since we have an assumption that the subjects are the same for both experiments, we use a different resampling procedure. To generate a single difference of means resample we perform the following:

- We iteratively select a subject in random.
- For the selected subject we simulate the experiment once more. The subject can respond to two images as:
  - Face, Face
  - Building, Building
  - Face, Building
  - Building, Face
- We pick an outcome for the experiment uniformly random and record the difference of two responses.
- After 20 response differences are recorded, we calculate and save the mean of the response differences.

The following code performs this procedure to generate 10000 x values:

```
1 diff_in_means = []
2 np.random.seed(7)
3 for i in range(NUM_SAMPLES):
4     resample = []
```

```
    for j in range(np.size(face)):
        indices = np.random.choice(np.size(face))
        options = [0, 0]
        options.append(building[j] - face[j])
        options.append(face[j] - building[j])
        resample.append(np.random.choice(options))
    diff_in_means.append(np.mean(resample))
diff_in_means = np.array(diff_in_means)
```

The resulting sampling distribution is the following:

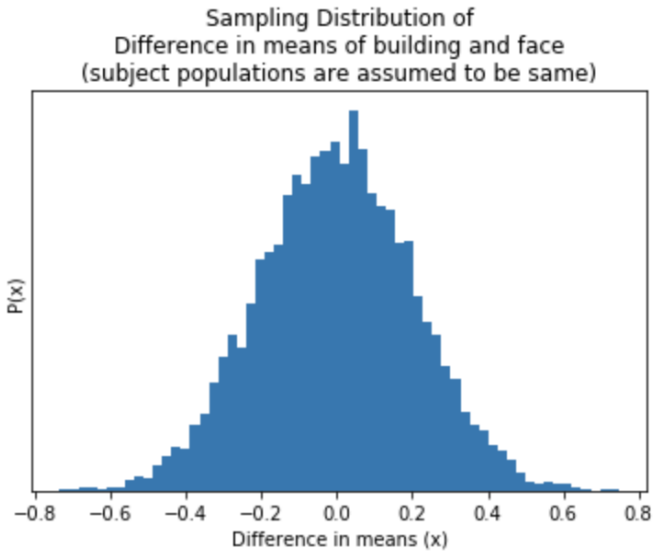

Figure 15: Sampling distribution of difference in means of `building` and `face` (subject populations are assumed to be same)

We now calculate the probability $P(x > \overline{x}|H_0)$, hence we define the z-value and the two sided p-value as:

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

$$p = P(|Z| > z|H_0)$$

$\overline{x}$ is the actual difference in means of `building` and `face`, the other variables are the same as usual.

These values can be computed using the following code:

```
x_overline = np.mean(building) - np.mean(face)
sigma_0 = np.std(diff_in_means)
mu_0 = np.mean(diff_in_means)
z = (x_overline - mu_0) / sigma_0
p = 2 * (1 - norm.cdf(np.abs(z)))
print('The z-value is:', z)
print('The two sided p-value is:', p)
```

The results are:

```
The z-value is: -3.6447496655297713
The two sided p-value is: 0.0002676521981408442
```

The p value is very small, so we reject the null hypothesis in favor of the alternative hypothesis. Meaning that there is a difference between the building and face responses assuming that the subjects are the same for both experiments.

4.e) Now, we repeat the previous part but we allow the subject populations to be distinct. Thus, we can resample the values just like we did in the first part of the question. The following line performs the resampling:

```
diff_in_means, vals, probs = difference_in_means(building, face, NUM_SAMPLES)
```

The resulting sampling distribution is the following:

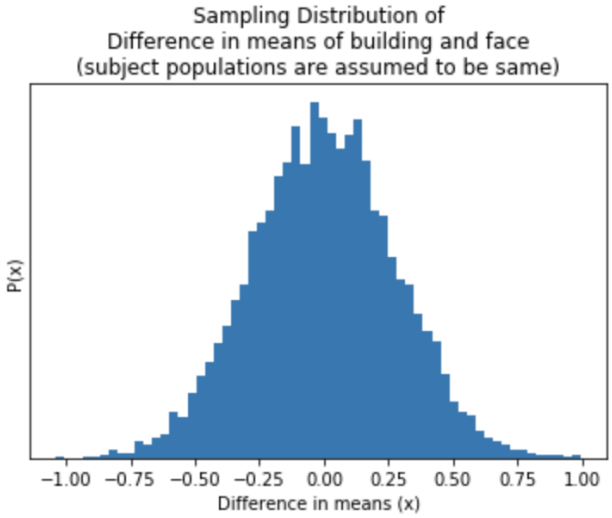

Figure 16: Sampling distribution of difference in means of `building` and `face` (subject populations are assumed to be distinct)

The z-value and the two sided p-value are computed as:

```
The z-value is: -2.686851787380054
The two sided p-value is: 0.00721289501971123
```

The p-value is again quite small, so we reject the null hypothesis that there is no difference between the building and face responses once more.