Homework 1

EEE482 - Computational Neuroscience

Efe Acer
21602217

Bilkent University, CS

Contents

Question 1

a) The first step to find the general solution x_n to the system $Ax = 0$ is to perform elementary row operations on A to obtain the reduced row echelon form of A, $rref(A)$:

$$A = \begin{bmatrix} 1 & 0 & -1 & 2 \\ 2 & 1 & -1 & 5 \\ 3 & 3 & 0 & 9 \end{bmatrix} \xrightarrow[R_3 \leftarrow R_3 - 3R_1]{R_2 \leftarrow R_2 - 2R_1,} \begin{bmatrix} 1 & 0 & -1 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 3 & 3 & 3 \end{bmatrix} \xrightarrow{R_3 \leftarrow R_3 - 3R_2} \begin{bmatrix} 1 & 0 & -1 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = rref(A)$$

Since the set of solutions to a linear system of equations is not affected by elementary row operations, the system can be rewritten as:

$$\begin{bmatrix} 1 & 0 & -1 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \iff \begin{matrix} x_1 - x_3 + 2x_4 = 0 \\ x_2 + x_3 + x_4 = 0 \end{matrix} \iff \begin{matrix} x_1 = x_3 - 2x_4 \\ x_2 = -x_3 - x_4 \end{matrix}$$

Here, x_1 and x_2 are pivot variables, whereas x_3 and x_4 are free variables. Hence, the solution should be written in terms of some arbitrary values of x_3 and x_4:

Let $x_3 = \alpha$ and $x_4 = \beta$, where $\alpha, \beta \in \mathbb{R}$. Then,

$$x_n = \begin{bmatrix} \alpha - 2\beta \\ -\alpha - \beta \\ \alpha \\ \beta \end{bmatrix} = \alpha \begin{bmatrix} 1 \\ -1 \\ 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} -2 \\ -1 \\ 0 \\ 1 \end{bmatrix} \quad \forall\ \alpha, \beta \in \mathbb{R}, \text{ is the general solution.}$$

This result can be confirmed by the following `Python` script, where the values of α and β are arbitrarily selected:

```
A = np.array([[1, 0, −1, 2],
              [2, 1, −1, 5],
              [3, 3, 0, 9]])
alpha = np.random.rand()
beta = np.random.rand()
x_n = np.array([alpha − 2 ∗ beta,
                −alpha − beta,
                alpha,
                beta])
print('A.x_n =', A.dot(x_n),
      '\nAs it can be seen the vector A.x_n is 0 for arbitrary alpha and beta.\n')
```

This script outputs:

```
A.x_n = [0. 0. 0.]
As it can be seen the vector A.x_n is 0 for arbitrary alpha and beta.
```

b) Similar to part **a**, we start by performing elementary row operations to solve the system $Ax = b$. However, the subject is the augmented matrix $[A|b]$ rather than A. We obtain $rref([A|b])$ with the following procedure:

$$[A|b] = \left[\begin{array}{cccc|c} 1 & 0 & -1 & 2 & 1 \\ 2 & 1 & -1 & 5 & 4 \\ 3 & 3 & 0 & 9 & 9 \end{array}\right] \xrightarrow[R_3 \leftarrow R_3 - 3R_1]{R_2 \leftarrow R_2 - 2R_1,} \left[\begin{array}{cccc|c} 1 & 0 & -1 & 2 & 1 \\ 0 & 1 & 1 & 1 & 2 \\ 0 & 3 & 3 & 3 & 6 \end{array}\right] \xrightarrow{R_3 \leftarrow R_3 - 3R_2}$$

$$\left[\begin{array}{cccc|c} 1 & 0 & -1 & 2 & 1 \\ 0 & 1 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] = rref([A|b])$$

Using $rref[A|b]$, we can rewrite the system as:

$$\begin{bmatrix} 1 & 0 & -1 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} \iff \begin{array}{c} x_1 - x_3 + 2x_4 = 1 \\ x_2 + x_3 + x_4 = 2 \end{array} \iff \begin{array}{c} x_1 = x_3 - 2x_4 + 1 \\ x_2 = -x_3 - x_4 + 2 \end{array}$$

The values $x_1 = 4$, $x_2 = 2$, $x_3 = 1$ and $x_4 = -1$ satisfies the simplified system. Hence, a particular solution, x_p, to the system is:

$$x_p^\top = \begin{bmatrix} 4 & 2 & 1 & -1 \end{bmatrix}$$

The following lines of code confirms the result:

```
1  b = np.array([1, 4, 9])
2  x_p = np.array([4, 2, 1, −1])
3  print('A.x_p =', A.dot(x_p),
4        '\nAs it can be seen the vector A.x_p equals the vector b.\n')
```

The output is:

```
A.x_p = [1 4 9]
As it can be seen the vector A.x_p equals the vector b.
```

c) Proceeding from the simplified system that we obtained in part **b**, we can construct a general solution in terms of arbitrary values of the free variables x_3 and x_4. Again:

$$\text{Let } x_3 = \alpha \text{ and } x_4 = \beta, \text{ where } \alpha, \beta \in \mathbb{R}.$$

Then, we can write our solution set, say S as:

$$S = \left\{ \begin{bmatrix} \alpha - 2\beta + 1 \\ -\alpha - \beta + 2 \\ \alpha \\ \beta \end{bmatrix}, \forall\ \alpha, \beta\ \in \mathbb{R} \right\} = \left\{ \alpha \begin{bmatrix} 1 \\ -1 \\ 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} -2 \\ -1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix}, \forall\ \alpha, \beta\ \in \mathbb{R} \right\}$$

Any vector, say $x_{n'}$, taken from the set S will be a valid solution. This can be confirmed by the following code:

```
alpha = np.random.rand()
beta = np.random.rand()
x_n_prime = np.array([alpha - 2 * beta + 1,
                      -alpha - beta + 2,
                      alpha,
                      beta])
print('A.x_n_prime =', A.dot(x_n_prime),
      '\nAs it can be seen the vector A.x_n_prime equals the vector b.\n')
```

The output is:

```
A.x_n_prime = [1. 4. 9.]
As it can be seen the vector A.x_n_prime equals the vector b.
```

d) The pseudo-inverse of A is denoted by A^+. It is a generalization of the inverse matrix and it is particularly useful for solving linear systems with a non-square coefficient matrix. A^+ can be computed from the Singular Value Decomposition (SVD) of A:

$$\begin{gathered} \text{SVD of } A \text{ gives: } A_{mxn} = U_{mxm}\Sigma_{mxn}V^{\top}_{nxn}, \\ \text{then pseudo-inverse is: } A^+ = V\Sigma^+U^{\top} \end{gathered}$$

One little note is that since SVD always exists, pseudo-inverse also exists for any matrix. In the expressions above, Σ contains the singular values as its diagonal entries in the descending order. U's ith row is the left-singular vector of A corresponding to the ith singular value in Σ, whereas V is the analogy for right-singular vectors.

The problem of finding right-singular vectors can be translated to the following eigen-value problem:

$$A^\top A v = \sigma^2 v, \text{ for nonzero v} \tag{1}$$

In a similar manner, the problem of finding left-singular vectors corresponds to the problem:

$$AA^\top u = \sigma^2 u, \text{ for nonzero u} \tag{2}$$

Since the transpose operation does not change the eigenvalues of a matrix, $A^\top A$ and $AA^\top$ share the same eigenvalues. Hence, σ^2 is common in (1) and (2). We need the left-singular and right-singular vectors together with the singular values to begin the computation of A^+, so the first step is to compute $A^\top A$ and $AA^\top$:

$$A^\top A = \begin{bmatrix} 14 & 11 & -3 & 39 \\ 11 & 10 & -1 & 32 \\ -3 & -1 & 2 & -7 \\ 39 & 32 & -1 & 110 \end{bmatrix}, \; AA^\top = \begin{bmatrix} 6 & 13 & 21 \\ 13 & 31 & 54 \\ 21 & 54 & 99 \end{bmatrix}$$

Now, we can rewrite the eigen-value problems (1) and (2) as:

$$(A^\top A - I_4 \, \sigma^2)v = 0 \iff det(A^\top A - I_4 \, \sigma^2) = 0 \tag{3}$$

$$(AA^\top - I_3 \, \sigma^2)v = 0 \iff det(AA^\top - I_3 \, \sigma^2) = 0 \tag{4}$$

Then, we proceed by computing the singular values using (4): (Let $\sigma^2 = \lambda$)

$$det(AA^\top - I_3 \, \lambda) = \begin{vmatrix} 6-\lambda & 13 & 21 \\ 13 & 31-\lambda & 54 \\ 21 & 54 & 99-\lambda \end{vmatrix}$$

$$= (6 - \lambda) \begin{vmatrix} 31-\lambda & 54 \\ 54 & 99-\lambda \end{vmatrix} - 13 \begin{vmatrix} 13 & 54 \\ 21 & 99-\lambda \end{vmatrix} + 21 \begin{vmatrix} 13 & 31-\lambda \\ 21 & 54 \end{vmatrix}$$

$$= (6 - \lambda)((31-\lambda)(99-\lambda) - 2916) - 13(13(99-\lambda) - 1134) + 21(702 - (31-\lambda)21)$$

$$= -\lambda^3 + 136\lambda^2 - 323\lambda$$

The roots of the characteristic polynomial of $AA^\top$ turns out to be 0, $68 - \sqrt{4301}$ and $68 + \sqrt{4301}$. These values introduce many numeric problems when trying to compute A^+ by hand. Thus, the exact same procedure is carried out in `Python`:

```
1  U, S, V_T = np.linalg.svd(A)
2  Sigma_plus = np.zeros(A.shape)
3  Sigma_plus[:len(S), :len(S)] = np.diag(np.concatenate((1 / S[0:2], np.array([0]))))
4  Sigma_plus = Sigma_plus.T
5  A_plus = V_T.T.dot(Sigma_plus).dot(U.T)
6  print(A.dot(A_plus).dot(A), '\n is an approximation of A.')
7  print('A_+:\n', A_plus, '\n')
```

The definition of pseudo inverse matrix states that $AA^+A = A$. [1] Hence, we can think of the difference between A and AA^+A as a measure of how "good" our pseudo inverse is. The code above has the following output:

```
[[ 1.00000000e+00 −4.99600361e−16 −1.00000000e+00 2.00000000e+00]
 [ 2.00000000e+00 1.00000000e+00 −1.00000000e+00 5.00000000e+00]
 [ 3.00000000e+00 3.00000000e+00 1.33226763e−15 9.00000000e+00]]
 is an approximation of A.
A_+:
 [[ 0.12693498 0.10835913 −0.05572755]
 [−0.23529412 −0.17647059 0.17647059]
 [−0.3622291 −0.28482972 0.23219814]
 [ 0.01857585 0.04024768 0.06501548]]
```

As it can be seen from the approximation of A , this method gives us a "good" pseudo inverse. Note that the 3rd line in the code above should be as it is. This is because of the fact that the singular value 0 has a numeric representation that is very small but not exactly 0. Thus, after the reciprocal operation the inverse of that singular value becomes very large and results in a biased A^+. For this reason, that singular value is taken as a 0.

Another "Pythonic" way to compute the pseudo inverse is to simply call `pinv()` function from `numpy`:

```
1  A_plus = np.linalg.pinv(A)
2  print(A.dot(A_plus).dot(A),
3        '\n is also an approximation of A, but here pinv() is used to compute A_+.')
4  print('A_+:\n', A_plus, '\n')
```

Using `pinv()`, we get the following output:

```
[[ 1.00000000e+00 −3.33066907e−16 −1.00000000e+00 2.00000000e+00]
 [ 2.00000000e+00 1.00000000e+00 −1.00000000e+00 5.00000000e+00]
 [ 3.00000000e+00 3.00000000e+00 1.38777878e−15 9.00000000e+00]]
 is also an approximation of A, but here pinv() is used to compute A_+.
A_+:
 [[ 0.12693498 0.10835913 −0.05572755]
 [−0.23529412 −0.17647059 0.17647059]
 [−0.3622291 −0.28482972 0.23219814]
 [ 0.01857585 0.04024768 0.06501548]]
```

The lengthy procedure and the `pinv()` method produces the exact same pseudo inverse. One can use whichever is preferred.

e) The sparsest solution corresponds to a vector from the solution set that has the maximum number of zeros. In a general setting, finding the sparsest solution to a system is known as the *Sparsest Solution Vector* problem and it is NP-hard meaning that there is no polynomial upper bound limiting the time complexity of the solution algorithm. However, we have a constrained version of the problem where we know that:

$$x_n = \begin{bmatrix} \alpha - 2\beta + 1 \\ -\alpha - \beta + 2 \\ \alpha \\ \beta \end{bmatrix}$$

Here, it can be observed that when we set α and β to nonzero values, we are guaranteed to have two nonzero entries. This is particularly because α and β appearing as the 3rd and 4th variables of the vector. In the case where α and β are both zeros, we try to maximize the number of zeros by solving the system:

$$\begin{matrix} \alpha - 2\beta + 1 = 0 \\ -\alpha - \beta + 2 = 0 \end{matrix} \text{, where } \alpha, \beta \neq 0$$

The solution to this system can be easily obtained by summing up the equations, which gives:

$$\alpha = 1 \text{ and } \beta = 1 \iff \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Another possible configuration of α and β are such that they are both zero. However, we cannot set the 1st and the 2nd variables to zero in that case:

$$\alpha = 0 \text{ and } \beta = 0 \iff \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix}$$

The last two possible configurations are where α is zero and β is nonzero, and vice versa. The following are possible values of α and β that maximizes the number of zeros in those cases:

$$\alpha = 0 \text{ and } \beta = 1/2 \iff \begin{bmatrix} 0 \\ 3/2 \\ 0 \\ 1/2 \end{bmatrix}$$

$$\alpha = 0 \text{ and } \beta = 2 \iff \begin{bmatrix} -3 \\ 0 \\ 0 \\ 2 \end{bmatrix}$$

$$\alpha = -1 \text{ and } \beta = 0 \iff \begin{bmatrix} 0 \\ 3 \\ -1 \\ 0 \end{bmatrix}$$

$$\alpha = 2 \text{ and } \beta = 0 \iff \begin{bmatrix} 3 \\ 0 \\ 2 \\ 0 \end{bmatrix}$$

The six listed vectors are all sparsest solutions to the system. We can be sure that there does not exist a sparser solution, since we individually looked at all possible configurations of α and β.

We can confirm that these vectors indeed satisfy the system, by running the following script:

```
# the tuples (alpha, beta) that results in one of the sparsest solutions
sparsest_args = [(0, 0), (1, 1), (0, 1 / 2), (0, 2), (−1, 0), (2, 0)]

for alpha_sparse, beta_sparse in sparsest_args:
    x_sparsest = np.array([alpha_sparse − 2 ∗ beta_sparse + 1,
                           −alpha_sparse − beta_sparse + 2,
                           alpha_sparse,
                           beta_sparse])
    print('alpha = ', alpha_sparse, ' beta = ', beta_sparse)
    print('x_sparsest = ', x_sparsest)
    print('A.x_sparsest =', A.dot(x_n_prime),
          '\nx_sparsest satisfies the system as it can be seen.\n')
```

The output of this script is in agreement with our result:

```
alpha = 0 beta = 0
x_sparsest = [1 2 0 0]
A.x_sparsest = [1. 4. 9.]
x_sparsest satisfies the system as it can be seen.

alpha = 1 beta = 1
x_sparsest = [0 0 1 1]
A.x_sparsest = [1. 4. 9.]
x_sparsest satisfies the system as it can be seen.

alpha = 0 beta = 0.5
x_sparsest = [0. 1.5 0. 0.5]
A.x_sparsest = [1. 4. 9.]
x_sparsest satisfies the system as it can be seen.

alpha = 0 beta = 2
x_sparsest = [−3 0 0 2]
A.x_sparsest = [1. 4. 9.]
x_sparsest satisfies the system as it can be seen.

alpha = −1 beta = 0
x_sparsest = [ 0 3 −1 0]
A.x_sparsest = [1. 4. 9.]
x_sparsest satisfies the system as it can be seen.

alpha = 2 beta = 0
x_sparsest = [3 0 2 0]
A.x_sparsest = [1. 4. 9.]
x_sparsest satisfies the system as it can be seen.
```

f) Finding the least-norm solution corresponds to the constraint optimization problem where we are trying to minimize $\|x_n\|$ subject to:

$$x_n = \begin{bmatrix} \alpha - 2\beta + 1 \\ -\alpha - \beta + 2 \\ \alpha \\ \beta \end{bmatrix}$$

The first step to solve this problem is to specify the function to minimize:

$$\text{Let } f(\alpha,\beta) = \|x_n\| = \sqrt{(\alpha - 2\beta + 1)^2 + (-\alpha - \beta + 2)^2 + \alpha^2 + \beta^2}$$

$$= \sqrt{3\alpha^2 + 6\beta^2 - 2\alpha - 8\beta - 2\alpha\beta + 5}$$

$$= \sqrt{g(\alpha,\beta)}$$

To minimize $f(\alpha,\beta)$, we should first find the critical points where the first order partial derivatives are zero:

$$\tfrac{\partial f(\alpha,\beta)}{\partial \alpha} = \tfrac{1}{2}\ g(\alpha,\beta)^{\frac{-1}{2}}\ \tfrac{\partial g(\alpha,\beta)}{\partial \alpha} = 0 \text{ , and}$$

$$\tfrac{\partial f(\alpha,\beta)}{\partial \beta} = \tfrac{1}{2}\ g(\alpha,\beta)^{\frac{-1}{2}}\ \tfrac{\partial g(\alpha,\beta)}{\partial \beta} = 0$$

We know that the zero vector is not a solution to the system $Ax = b$, thus minimizing $f(\alpha,\beta)$ is exactly same as minimizing $g(\alpha,\beta)$, we find the critical values of α and β as:

$$\tfrac{\partial g(\alpha,\beta)}{\partial \alpha} = 6\alpha - 2\beta - 2 = 0 \text{ , and}$$

$$\tfrac{\partial g(\alpha,\beta)}{\partial \beta} = -2\alpha + 12\beta - 8 = 0$$

Solving these simultaneously, we get $\alpha = 10/17$ and $\beta = 13/17$. The only step that remains is to check whether this critical point defines a minimum or not. This can be done by checking the sign of the determinant of the Hessian matrix (H), which is essentially the second derivative test:

$$H = \begin{bmatrix} \frac{\partial^2 g}{\partial^2 \alpha} & \frac{\partial^2 g}{\partial \alpha \partial \beta} \\ \frac{\partial^2 g}{\partial \beta \partial \alpha} & \frac{\partial^2 g}{\partial^2 \beta} \end{bmatrix} = \begin{bmatrix} 6 & -2 \\ -2 & 12 \end{bmatrix},\ det(H) = 6.12 - (-2).(-2) = 68 > 0$$

This implies that the calculated values of α and β define a local minimum. Together with the fact that the polynomial $g(\alpha,\beta)$ being a convex function, we can be sure that we arrived at the global minimum.

The least-norm x_n calculated from the minimizing values $\alpha = 10/17$ and $\beta = 13/17$ is:

$$x_{least-norm} = \begin{bmatrix} 1/17 \\ 11/17 \\ 10/17 \\ 13/17 \end{bmatrix}$$

A faster way to find $x_{least-norm}$ is to use pseudo inverse of A, namely A^+. By definition pseudo inverse provides the least norm solution to a system. Hence, $x_{least-norm}$ can be simply computed using the lines:

```
# A_plus = np.linalg.pinv(A)
x_least_norm = A_plus.dot(b)

print('The least norm solution: ', x_least_norm)
```

The lines above produce the output:

```
The least norm solution: [0.05882353 0.64705882 0.58823529 0.76470588]
```

The vector in the output is a numeric approximation of the result we found by solving the constraint minimization problem, thus our result is confirmed.

Note that `Numpy` should be imported for the solution code for this question to work.

Question 2

In order to perform statistical computations and data visualization (also to load the data in the `.mat` format into the `Python` environment) some libraries are necessary. The `Python` solution code for this question, assumes that the following lines are executed:

```
# Necassary imports
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat # to be able to use .mat file in the python environment
```

The `loadmat` function in the `scipy.io` package is necessary to load the dataset, this can be done by executing the lines:

```
data = loadmat('XData.mat')
x_data = data['x'].flatten()
print('The dataset: ', x_data, '\n')
```

a) Mean and median tells about where the data is centered. Depending on the specifics of a dataset such as the frequency of the outliers, which are essentially the data points that are distant from others, one of these measures may be more preferable. Overall, these measures establish a solid baseline for the underlying distribution that generated the data.

To compute the sample mean and the sample median of the dataset, numpy functions `np.mean()` and `np.median()` are used:

```
mean = np.mean(x_data)
median = np.median(x_data)
print('Sample mean: %f \nSample median: %f\n' % (mean, median))
```

The output of the script gives the results:

```
Sample mean: 98.436667
Sample median: 97.850000
```

b) One can also analyze the dispersion of a dataset to gain further understanding about the underlying distribution. The standard deviation and the inter-quantile range are descriptive measures that are used to quantify the dispersion of the data points.

Similarly the sample standard deviation is computed using `np.std()`. For the sample inter-quantile range computation, the first and the third quantiles are found with the help of the function `np.percentile()` and their difference is computed. The script is provided below:

```
std = np.std(x_data)
q25, q75 = np.percentile(x_data, [25, 75])
iqr = q75 - q25
print('Sample standard deviation: %f \nSample inter-quartile range: %f\n' % (std, iqr))
```

The output is:

```
Sample standard deviation: 10.056523
Sample inter-quartile range: 9.950000
```

c) An histogram is a type of plot that visually represents the frequency distribution of data points. It correlates to the likelihood function since frequently observed data points are likelier to occur in the future.

`Matplotlib`, a 2D plotting library in `Python` is used to draw the histograms. The function calls in the library are self-explanatory. The code that plots the required histograms is given below:

```
x_range = [70, 130]
num_bins = [3, 6, 12]
colors = ['g', 'r', 'b']

print('The histograms:')
for i in range(3):
    plt.figure(figure_num)
    figure_num += 1
    plt.title('Histogram of the aggragated responses')
    plt.ylabel('Counts')
    plt.xlabel('Aggragated responses')
    plt.hist(x_data.flatten(), bins=num_bins[i], range=x_range, color=colors[i], edgecolor='black', linewidth
        =1.2)
    plt.show(block=False)
```

Note that the very last line `plt.show(block=False)` has the `block=False` argument, because when `block=True` (the default value) the execution stops and the program waits for the user to press the `Enter` key. To prevent this behavior, `plt.show()` line is added at the very end of the solution code of Question 2; this way the program executes completely and the plots are displayed together afterwards.

The histograms drawn by the script above follow. Notice that as the number of bins increases the histograms become more detailed; also they give the impression that the data is normally distributed. Under the assumption that we have enough data, it looks like the histogram will converge to a Gaussian as the number of bins approaches infinity.

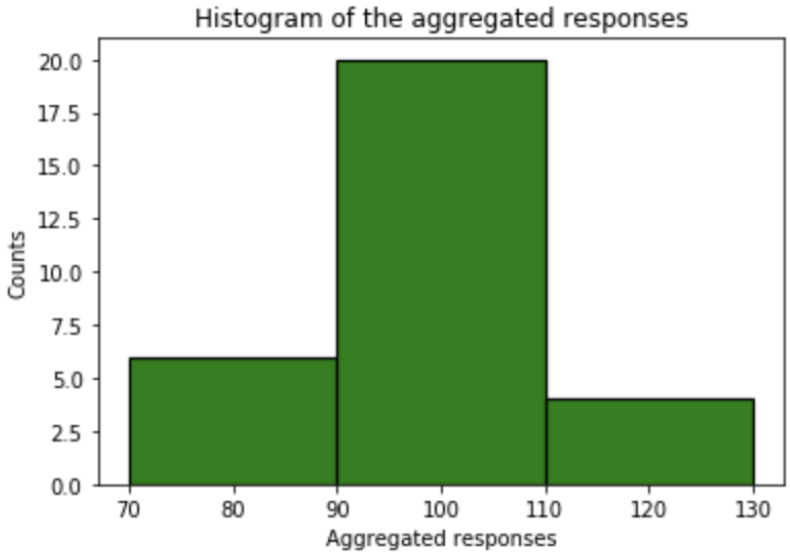

Figure 1: Histogram of the aggregated responses with 3 bins.

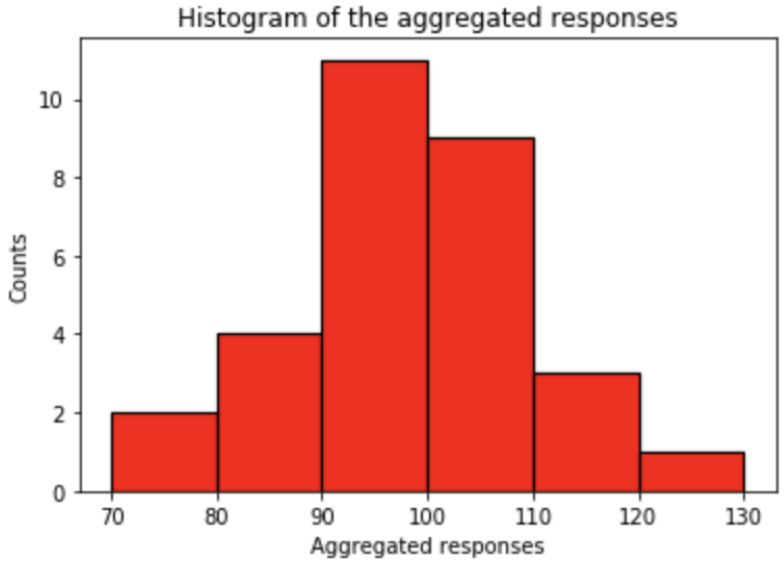

Figure 2: Histogram of the aggregated responses with 6 bins.

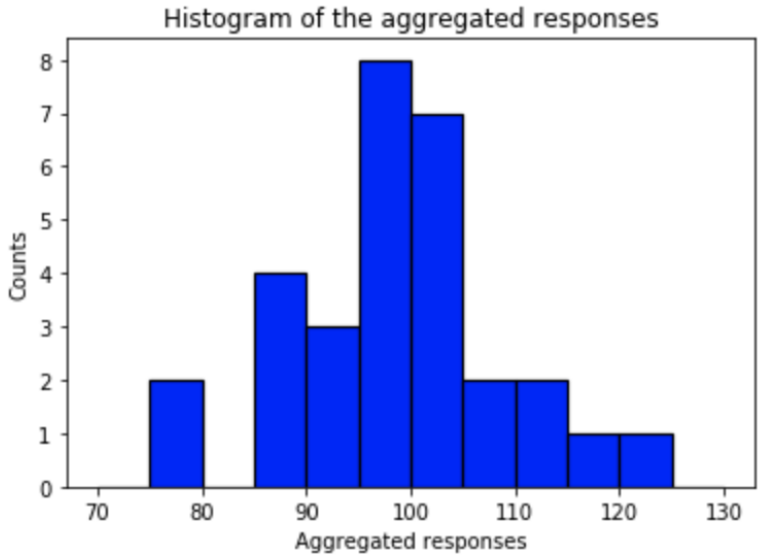

Figure 3: Histogram of the aggregated responses with 12 bins.

d) The Normal Quantile Plot is basically a procedure to visually check whether the data came from the Normal distribution or not. It simply sorts the sample data in ascending order and plots the quantiles against the quantiles drawn from the Normal distribution. If the resulting scatter plot forms a roughly straight line, it means that the both set of quantiles came from the Normal distribution. [2]

The `Matlab` Normal Quantile Plot code given in the assignment can be translated into `Python` as:

```
# Python equivalent of the Matlab Normal Quantile Plot code given in the assignment
y = np.sort(x_data)
n = np.size(x_data)
f = (np.arange(1, n + 1) − 3/ 8) / (n + 1 / 4)
q = 4.91 ∗ (f ∗∗ 0.14 − (1 − f) ∗∗ 0.14)
plt.grid()
plt.plot(q, y, '∗−')
```

The plot produced by this script is given in the next page. This plot conforms with the expectation that the data is normally distributed since the points form a roughly straight line.

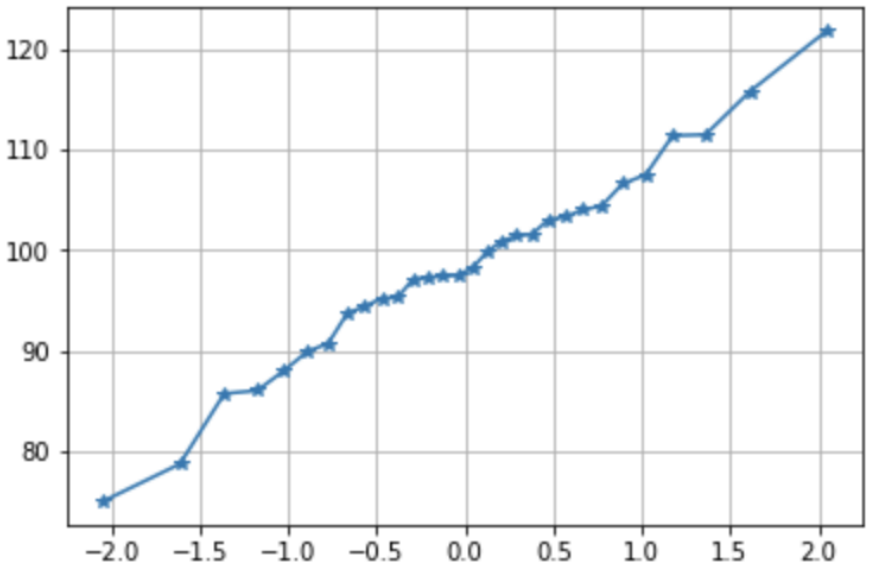

Figure 4: Histogram of the aggregated responses with 12 bins.

e) Bootstrap resampling is a method to approximate the properties of a distribution using a generalized approximation of the dataset. The method begins by randomly picking samples from the dataset with replacement. Then the properties of interest are computed again for these resamples. Later, the means of the properties together with their standard errors can be reported as valid approximations. This method is quite useful in practice since it can artificially produce up to n^n resamples, where n is the number of samples. [3]

A function to perform bootstrap resampling can be implemented as follows:

```
# Function to perform bootstrap resampling
    def bootstrap_resampling(data, bootstraps, stat_func):
    """
    Given the data, number of bootstrapping iterations and the statistic of
    interest; this function returns a list of statistics obtained from each
    bootstrapped sample of the original data.
    Args:
        data: The original data
        bootstraps: The number of bootstrapping iterations
        stat_func: The function to compute the statistic of interest
            (e.g.: np.mean, np.std, etc.)
    Returns:
        bootstrap_replicate: The mean of all bootstrap replicates
        standard_error: The standard error of the bootstrap replicate
        bootstrap_replicates: The list of statistics obtained from the
            bootstrapped samples
        """
```

```
        bootstrap_replicates = []
            sample_size = np.size(data)
            for _ in range(bootstraps):
                bootstrap_sample = np.random.choice(data, sample_size, replace=True)
                bootstrap_replicates.append(stat_func(bootstrap_sample))
            return np.mean(bootstrap_replicates), np.std(bootstrap_replicates), bootstrap_replicates
```

Another important statistical concept is confidence interval. Confidence interval refers to the range of values that is likely to include an unseen data point generated by the underlying distribution. In particular, 95% confidence interval refers to the range in which you can be 95% percent sure that your statistics are sound. In other words, it is the range where your results will lie in 95% of your experiments.

An implementation of a function that computes the confidence interval for a dataset is given:

```
# Function to compute the confidence interval of data samples
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
```

The mean of the distribution can be approximated from 1000 bootsrap resamples using the functions above as:

```
# Performing the bootstrap sampling and computing the bootstrap replicate
# of the mean together with its standard error
bootstrap_mean, standard_error_mean, bootstrap_means = bootstrap_resampling(x_data, 1000, np.mean)
print('The sample mean computed from %s bootstrap samples: %f' % (1000, bootstrap_mean))
print('The bootstrapped estimate for the standard error of the mean:', standard_error_mean)

# Computing the 95% confidence interval
lower, upper = compute_confidence_interval(bootstrap_means, 95)
confidence_interval = (lower, upper)
print('The 95% confidence interval of the mean:', confidence_interval)

print('Overall, mean of the dataset can be reported as: %.3f +- %.3f, (%.3f, %.3f)\n' %
      (bootstrap_mean, standard_error_mean, lower, upper))
```

`np.std()` is used to compute the standard error, since it is simply the standard deviation of the sampling distribution. The output of the script is:

```
The sample mean computed from 1000 bootstrap samples: 98.438220
The bootstrapped estimate for the standard error of the mean: 1.798037377080295
The 95% confidence interval of the mean: (95.12241666666664, 102.17050000000002)
Overall, mean of the dataset can be reported as: 98.438 +- 1.798, (95.122, 102.171)
```

An histogram of the means of the bootstrap resamples can be plotted using the same procedures used in the previous parts. Such an histogram with 50 bins is shown below together with the lines to plot it:

```
1 plt.title('Histogram of the bootstrapped means')
2 plt.ylabel('Counts')
3 plt.xlabel('Means')
4 plt.hist(bootstrap_means, bins=50, edgecolor='black', linewidth=1.2)
5 plt.show(block=False)
```

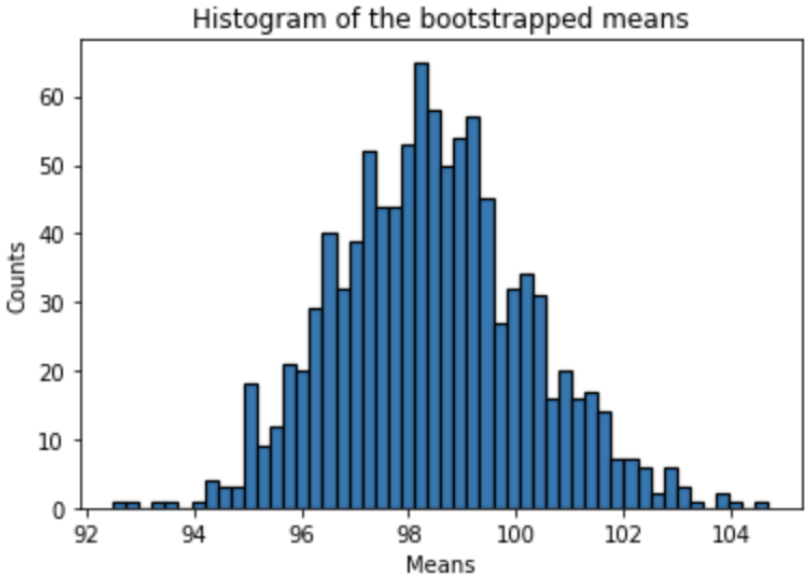

Figure 5: Histogram of the means of the bootstrap resamples with 50 bins.

f) The exact same steps are followed to estimate the standard deviation of the underlying distribution. The only change is that the statistic of interest is now the standard deviation instead of the mean. The corresponding script follows:

```
1 # Performing the bootstrap sampling and computing the bootstrap replicate of the
2 # standard deviation together with its standard error
```

```
bootstrap_std, standard_error_std, bootstrap_stds = bootstrap_resampling(x_data, 1000, np.std)
print('The sample standard deviation computed from %s bootstrap samples: %f' % (1000, bootstrap_std))
print('The bootstrapped estimate for the standard error of the standard deviation:', standard_error_std)

# Computing the 95% confidence interval
lower, upper = compute_confidence_interval(bootstrap_stds, 95)
confidence_interval = (lower, upper)
print('The 95% confidence interval of the standard deviation:', confidence_interval)

print('Overall, standard deviation of the dataset can be reported as: %.3f +- %.3f, (%.3f, %.3f)\n'
      % (bootstrap_std, standard_error_std, lower, upper))
```

The output of this script and the histogram of the standard deviations of the bootstrap resamples with 50 bins are:

```
The sample standard deviation computed from 1000 bootstrap samples: 9.833668
The bootstrapped estimate for the standard error of the standard deviation: 1.3571556300969863
The 95% confidence interval of the standard deviation: (7.173218592238309, 12.5811392567476)
Overall, standard deviation of the dataset can be reported as: 9.834 +- 1.357, (7.173, 12.581)
```

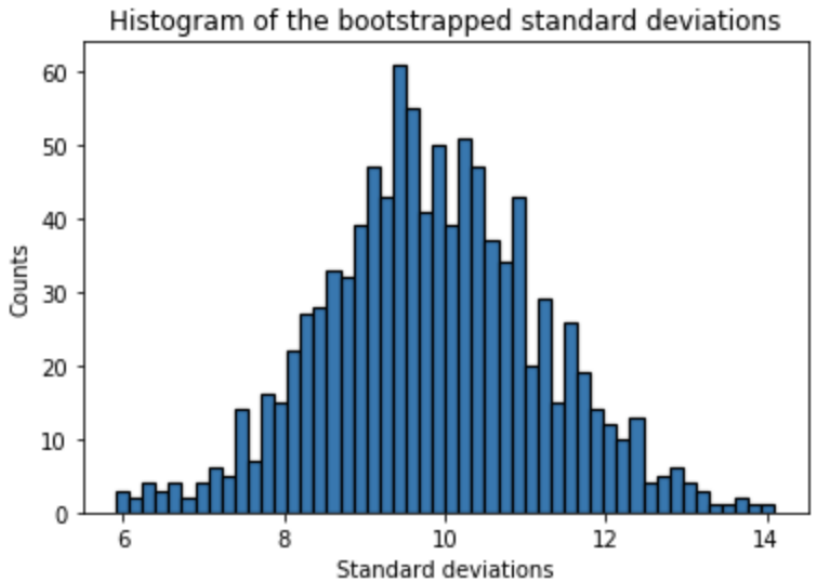

Figure 6: Histogram of the standard deviations of the bootstrap resamples with 50 bins.

g) Jackknife resampling is yet another resampling procedure to obtain valid approximations for statistics of interest. It pre-dates most of the resampling methods and it is relatively simpler. It operates by systematically leaving out one sample and considering the resulting subsample as a resample. Then, it computes the statistics of interest on these resamples. The number of resamples it can produce has an upper bound of n, where n is again the number of samples. [4]

A function that performs jackknife resampling is provided below:

```
# Function to perform Jacknife resampling
def jackknife_resampling(data, stat_func):
    """
    Given the data and the statistic of interest; this function returns a
    list of statistics obtained from each jacknifed sample of the original data.
    Args:
        data: The original data
        stat_func: The function to compute the statistic of interest
            (e.g.: np.mean, np.std, etc.)
    Returns:
        jackknife_replicate: The mean of all jackknife replicates
        standard_error: The standard error of the jackknife replicate
        jackknife_replicates: The list of statistics obtained from the
            jackknifed samples
    """
    jackknife_replicates = []
    jackknifes = np.size(data)
    for i in range(jackknifes):
        jackknife_data = [elem for j, elem in enumerate(data) if i != j]
        jackknife_replicate_i = stat_func(jackknife_data)
        jackknife_replicates.append(jackknife_replicate_i)
        return np.mean(jackknife_replicates), np.std(jackknife_replicates), jackknife_replicates
```

The jackknife approximation of the sample mean can be neatly formulated as:

$$\bar{x}_i = \frac{1}{n-1} \sum_{j=1, j \neq i}^{n} x_j, \qquad i = 1, \ldots, n.$$

The previous parts in which mean and standard deviation are approximated can be repeated again with the little change where `jackknife_resampling` method is called instead of `bootstrap_resampling`.

This code approximates the mean of the underlying distribution using jackknife resampling:

```
# Performing the jackknife sampling and computing the jackknife replicate
# of the mean togerther with its standard error
jackknife_mean, standard_error_mean, jackknife_means = jackknife_resampling(x_data, np.mean)
print('The sample mean computed from jackknife samples:', jackknife_mean)
print('The jackknifed estimate for the standard error of the mean:', standard_error_mean)

# Computing the 95% confidence interval
lower, upper = compute_confidence_interval(jackknife_means, 95)
confidence_interval = (lower, upper)
print('The 95% confidence interval of the mean:', confidence_interval)

print('Overall, mean of the dataset can be reported as: %.3f +- %.3f, (%.3f, %.3f)\n' %
      (jackknife_mean, standard_error_mean, lower, upper))
```

The output of this code and the histogram of the means of jackknife resamples with 5 bins are:

```
The sample mean computed from jackknife samples: 98.43666666666667
The jackknifed estimate for the standard error of the mean: 0.3467766563828598
The 95% confidence interval of the mean: (97.78103448275863, 99.15232758620692)
Overall, mean of the dataset can be reported as: 98.437 +− 0.347, (97.781, 99.152)
```

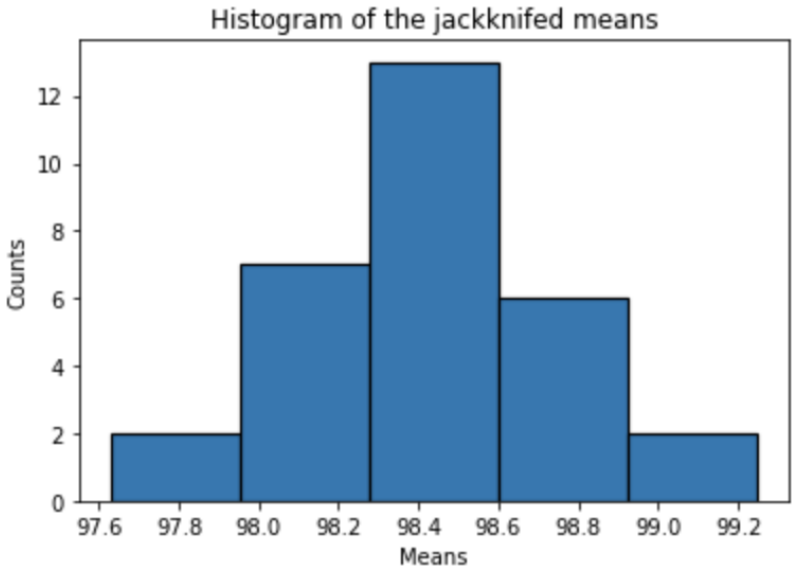

Figure 7: Histogram of the means of the jackknife resamples with 5 bins.

One can follow the same steps to approximate the standard deviation using jackknife resampling, the only change is to pass `np.std` as the function to compute the statistic of interest to the `jackknife_resampling` function:

```
# Performing the jackknife sampling and computing the jackknife replicate
# of the standard deviation togerther with its standard error
jackknife_std, standard_error_std, jackknife_stds = jackknife_resampling(x_data, np.std)
print('The sample standard deviation computed from jackknife samples:', jackknife_std)
print('The jackknifed estimate for the standard error of the standard deviation:', standard_error_std)

# Computing the 95% confidence interval
lower, upper = compute_confidence_interval(jackknife_stds, 95)
confidence_interval = (lower, upper)
print('The 95% confidence interval of the standard deviation:', confidence_interval)

print('Overall, standard deviation of the dataset can be reported as: %.3f +− %.3f, (%.3f, %.3f)\n' %
      (jackknife_std, standard_error_std, lower, upper))
```

The output of these lines and the histogram of the standard deviations of the jackknife resamples with 5 bins follow:

```
The sample standard deviation computed from jackknife samples: 10.046791101023341
The jackknifed estimate for the standard error of the standard deviation: 0.27457253785040076
The 95% confidence interval of the standard deviation: (9.225836041965325, 10.22730576358885)
Overall, standard deviation of the dataset can be reported as: 10.047 +− 0.275, (9.226, 10.227)
```

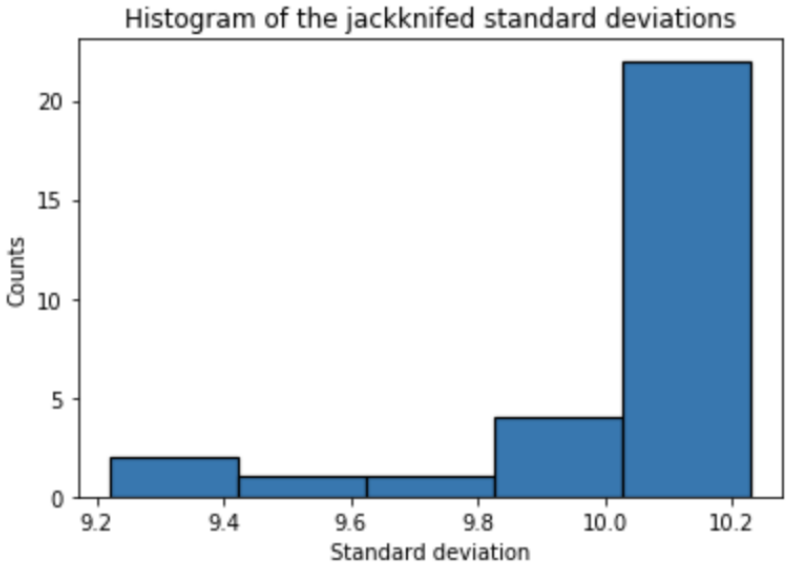

Figure 8: Histogram of the standard deviations of the jackknife resamples with 5 bins.

Question 3

The `Python` code that solves the question assumes that the following lines are executed to import the necessary libraries:

```
# Necessary imports
import numpy as np
import matplotlib.pyplot as plt # for bar plots
from scipy.special import comb # for efficient combination computation
```

a) It is given that the conditional probability of Broca activation follows the Beurnoulli distribution, which is defined as:

$$P(data|x) = \begin{cases} x & \text{if } data = 1 \\ 1-x & \text{if } data = 0 \end{cases} \tag{5}$$

Here, *data* refers to a binary variable that can only take on the values $\{0, 1\}$, 1 meaning that the Broca's area is active whereas 0 means the Broca's area is inactive. x simply refers to the probability of activation.

For the language involving tasks, we are given that the Broca's area was active 103 out of 809 experiments. We assume that the data obtained from those experiments are independently and identically distributed ($i.i.d$), that is the outcome of an experiment does not effect the outcome of the future experiments and the distribution from which the data is drawn is static. Then, using the multiplication rule, we can write the likelihood to observe the specified data as:

$$L(data|x_l) = \binom{809}{103} \prod_{data_i} P(data_i|x_l) = \binom{809}{103} x_l^{103}(1-x_l)^{809-103} = \frac{809!}{766!\ 103!} x_l^{103}(1-x_l)^{766} \tag{6}$$

In a similar way, we can write the likelihood to observe 199 activations out of 2353 experiments not involving language as:

$$L(data|x_{nl}) = \binom{2353}{119} \prod_{data_i} P(data_i|x_{nl}) = \binom{2353}{119} x_{nl}^{119}(1-x_{nl})^{2353-119} = \frac{2353!}{2234!\ 119!} x_{nl}^{119}(1-x_{nl})^{2234} \tag{7}$$

Note that the Binomial coefficients play a significant role in both of the equations, since there is no information about the order of experiments in which the data is obtained.

A function to compute the numeric values of these likelihoods can be implemented as:

```
def likelihood_beurnoulli(prob, num_positive, total):
    """
    Likelihood function of the beurnoulli distribution.
    Args:
        prob: The probability that a binary event has a postive outcome
        num_positive: Number of positive outcomes
        total: Total number of observations/trials
    Returns:
        likelihood: The likelihood of having the beurnoulli distribution
            with the specified arguments
    """
    num_negative = total - num_positive
    likelihood = (prob ** num_positive) * ((1 - prob) ** num_negative)
    likelihood *= comb(total, num_positive)
    return likelihood
```

The likelihood functions given in (6) and (7) are computed for the required values of x_l and x_{nl} using the following lines:

```
x = np.arange(0, 1.001, 0.001) # probability values to try
# Compute the likelihood vectors for language invloving tasks and others
likelihoods_l = likelihood_beurnoulli(x, ACTIVE_L, TOTAL_L)
likelihoods_nl = likelihood_beurnoulli(x, ACTIVE_NL, TOTAL_NL)
```

In the code segment above `ACTIVE_L` is 103, `TOTAL_L` is 809, `ACTIVE_NL` is 119 and `TOTAL_NL` is 2353.

The vectors `likelihoods_l` and `likelihoods_nl` can indeed be interpreted as discretized likelihood functions, since they contain samples from the continuous likelihood functions at some discrete values of x_l and x_{nl}.

Bar charts are useful to visualize these discretized likelihood functions and one can be plotted for `likelihoods_l` using the lines:

```
figure_num = 1
plt.figure(figure_num)
figure_num += 1
plt.bar(np.arange(len(x)), likelihoods_l, color='b')
plt.xlim(0, 200)
plt.xticks(np.arange(0, 201, step=50), (0, 0.05, 0.1, 0.15, 0.2))
plt.xlabel('Probability')
plt.ylabel('Likelihood')
plt.title('Discretized likelihood function\n for language involving tasks')
plt.show(block=False)
```

The output of these lines is given below:

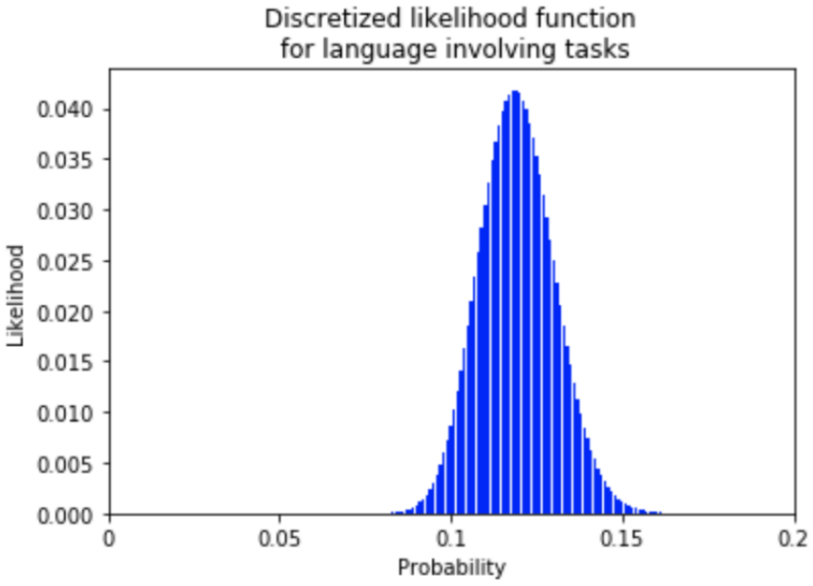

Figure 9: Discretized likelihood function for language involving tasks (`likelihoods_l`)

The bar chart for `likelihoods_nl` is plotted using the same script where very few of the arguments are changed, the resulting output follows:

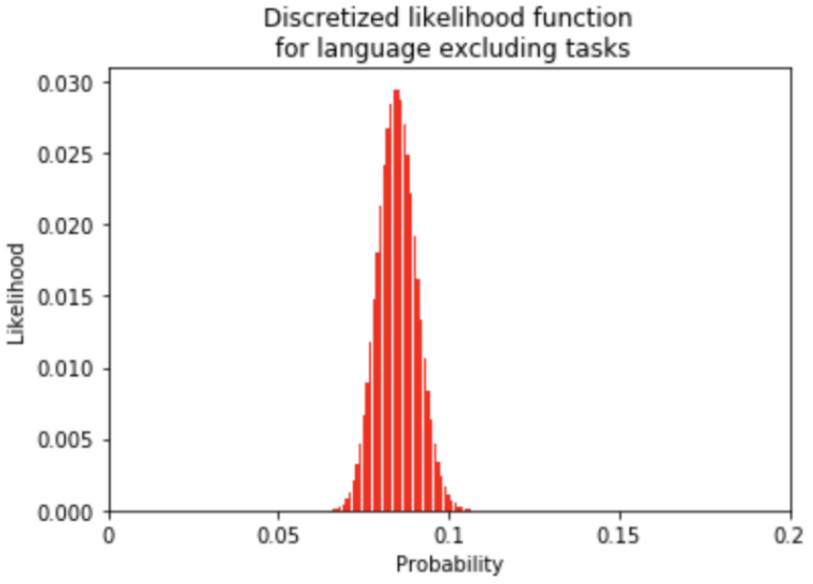

Figure 10: Discretized likelihood function for language excluding tasks (`likelihoods_nl`)

b) We are asked to find the values of x_l and x_{nl} that maximize the *discretized* counterparts of (6) and (7), so it suffices to find the probabilities that maximize the likelihood values in the vectors `likelihoods_l` and `likelihoods_nl`. Since the ordering of the values in the vectors correspond to the ascending ordering of the probabilities, we can easily find the maximizing probabilities by executing the lines:

```
x_l_max = x[np.argmax(likelihoods_l)]
print('The probability that maximizes the discretized likelihood of the data ' +
      'obtained from the language involving tasks: %.3f, %.5f' % (x_l_max,
      likelihood_beurnoulli(x_l_max, ACTIVE_L, TOTAL_L)))
x_nl_max = x[np.argmax(likelihoods_nl)]
print('The probability that maximizes the discretized likelihood of the data ' +
      'obtained from the language excluding tasks: %.3f, %.5f' % (x_nl_max,
      likelihood_beurnoulli(x_nl_max, ACTIVE_NL, TOTAL_NL)))
```

The output of the script gives the maximizing values of x_l and x_{nl} together with the corresponding likelihood values:

```
The probability that maximizes the discretized likelihood of the data obtained from the language involving
    tasks: 0.119, 0.04180
The probability that maximizes the discretized likelihood of the data obtained from the language excluding
    tasks: 0.085, 0.02946
```

c) In order to make the transition from $P(data|X)$ to $P(X|data)$, we need to use the Baye's Theorem. Baye's Theorem or Baye's Rule states that:

$$P(X|data)P(data) = P(data|X)P(X) \tag{8}$$

$$P(X|data) = \frac{P(data|X)P(X)}{P(data)} \tag{9}$$

$P(data)$ is not explicitly given, hence we need to calculate it by marginalizing X out in $P(data|X)$. Thus, we write:

$$P(X|data) = \frac{P(data|X)P(X)}{\sum_x P(data|x)p(x)} \tag{10}$$

We are given the assumption that the prior distribution, $P(X)$ is a uniform distribution; X can take on values from the set $S_X = \{0, 0.001, 0.002, ..., 0.999, 1\}$, hence:

$$P(X) = \frac{1}{|S_X|} = \frac{1}{1001} = P(X = x), \forall x \in S_X \tag{11}$$

Since $P(X) = P(x)$ is a constant, we can move $P(x)$ out of the summation in the denominator and cancel it with the $P(X)$ in the numerator. We obtain:

$$P(X|data) = \frac{P(data|X)}{\sum_x P(data|x)} \tag{12}$$

$P(data|X)$ is the likelihood that is already computed, $P(X|data)$ can be easily calculated from (12). However, the code below calculates $P(X|data)$ for language involving tasks using (10) and (11) because of the fact that (12) is not a generic formula but rather a special case. The code follows:

```
# Assumption is a uniform prior
uniform_prior = 1 / len(x)
# Marginalize the prior out in the conditional distribution to compute
# the normalizer
normalizer_l = np.sum(likelihoods_l * uniform_prior)
# Apply Bayes rule
posterior_l = likelihoods_l * uniform_prior / normalizer_l
```

$P(X|data)$ for language excluding tasks (`posterior_nl`) can be calculated using the same procedure for `likelihoods_nl`.

The following figures give bar charts for `posterior_l` and `posterior_nl` to visualize the discretized posterior distributions:

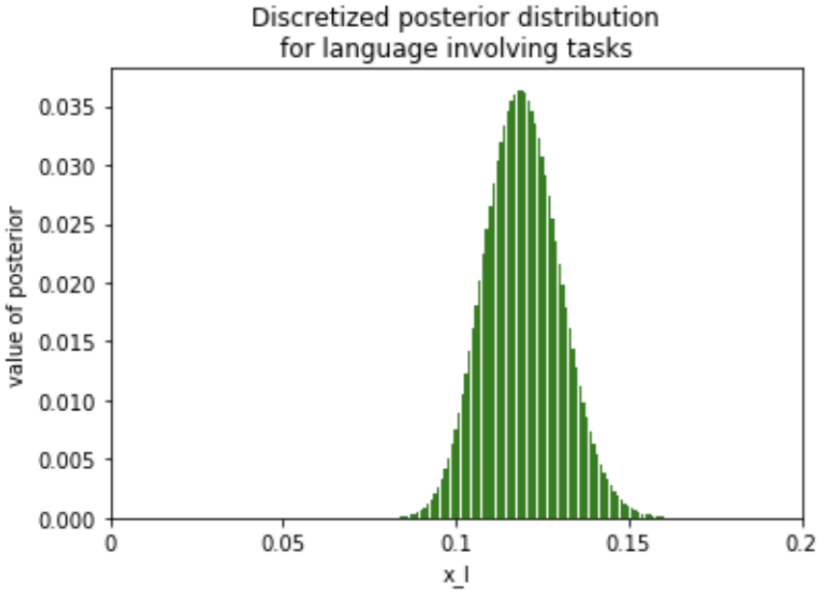

Figure 11: Discretized posterior distribution for language including tasks (`posterior_l`)

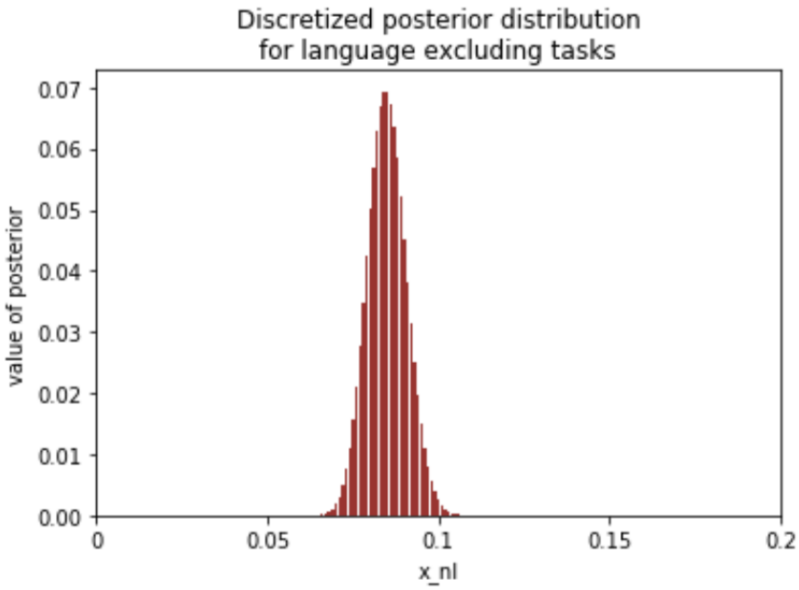

Figure 12: Discretized posterior distribution for language excluding tasks (`posterior_nl`)

By definition, to find the 95% confidence intervals of x_l and x_{nl}, we should find the 2.5 and 97.5 percentiles. 2.5 percentile is the value which is located on the 2.5th percent of the total coverage of the distribution. Same argument applies for 97.5 percentile. The conditional cumulative distribution function (c.d.f) is given as:

$$P(X \leq x|data) = F(X|data) = \sum_{x_i \leq x} P(x|data) \tag{13}$$

This expression implies that the value of the c.d.f, $F(x|data)$, corresponds to the area under the posterior distribution (essentially its integral in the continuous case) from 0 to x. Hence, the area under the posterior distribution computed at a given value x gives how much of the distribution is covered up to and including the value x. Since we want to find the 2.5th and 97.5th percentiles, it suffices to find the values of x where $F(x|data)$ equals 0.025 (2.5%) and 0.975 (97.5%).

The c.d.f of the posterior distribution with tasks including language tasks can be calculated using (13) with the following line:

```
1 cdf_l = np.array([np.sum(posterior_l[:i]) for i in range(1, len(x) + 1)])
```

Similarly, c.d.f of the posterior distribution with tasks excluding language can be calculated with the line:

```
1 cdf_nl = np.array([np.sum(posterior_nl[:i]) for i in range(1, len(x) + 1)])
```

Then to plot a bar chart that visualizes `cdf_l`, the given lines can be executed:

```
plt.figure(figure_num)
figure_num += 1
plt.bar(np.arange(len(x)), cdf_l, color='navy')
plt.xticks(np.arange(len(x), step=100),
           (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1))
plt.xlabel('x_l')
plt.ylabel('value of cumulative posterior')
plt.title('Cumulative posterior distribution\nfor language involving tasks')
plt.show(block=False)
```

The output figure is:

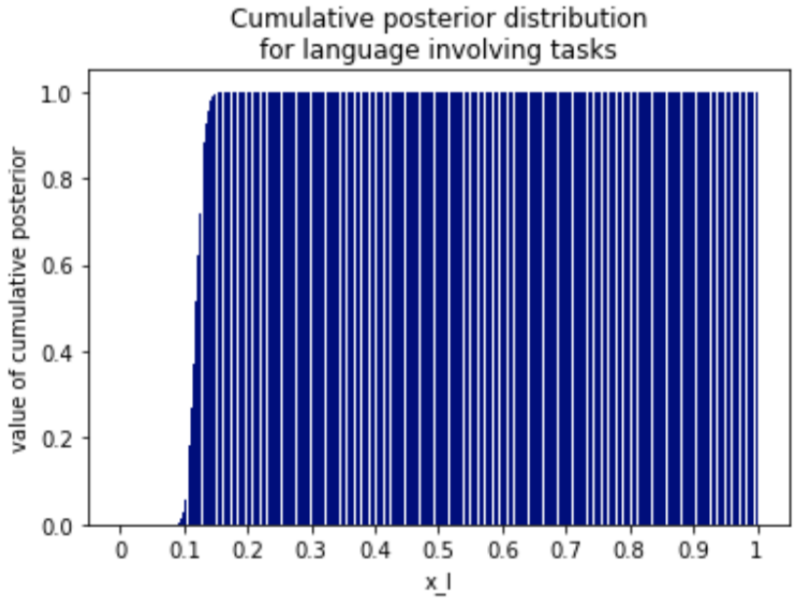

Figure 13: Cumulative posterior distribution for language involving tasks (`cdf_l`)

The counterpart of the script above plots the bar chart for `cdf_nl`. The chart follows:

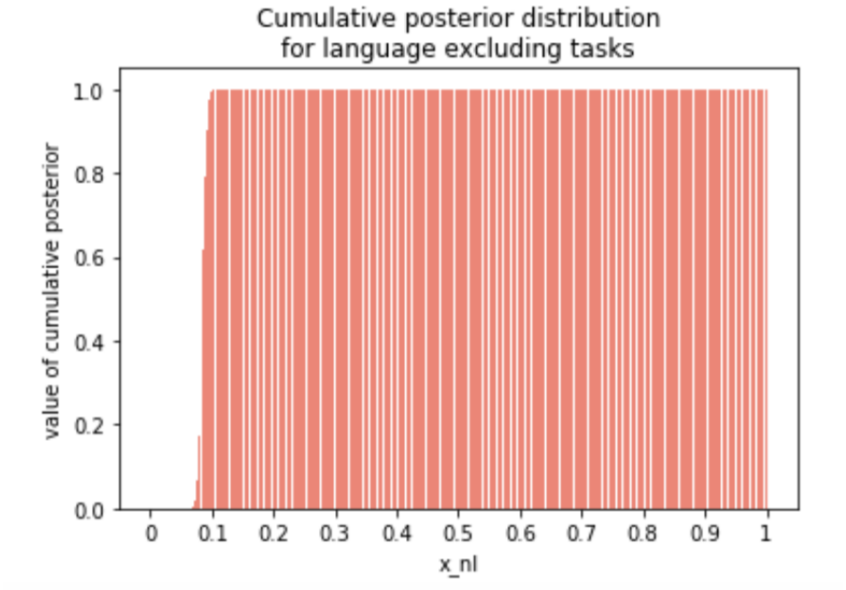

Figure 14: Cumulative posterior distribution for language excluding tasks (`cdf_nl`)

To find the confidence intervals of x_l and x_{nl}, we find probabilities corresponding to $F(X_l|data) = 0.025$, $F(X_l|data) = 0.975$, $F(X_{nl}|data) = 0.025$ and $F(X_{nl}|data) = 0.975$. The scripts to find these values for `cdf_l` and `cdf_nl` and their outputs are given below:

```
# Approximating the lower and higher 95% confidence bounds using the cdf
index_lower = np.argmin(np.abs(cdf_l - 0.025))
index_upper = np.argmin(np.abs(cdf_l - 0.975))
print('lower and higher 95%% confidence bounds of x_l respectively: (%.3f, %.3f)' %
      (x[index_lower], x[index_upper]))
```

```
lower and higher 95% confidence bounds of x_l respectively: (0.098, 0.141)
```

```
index_lower = np.argmin(np.abs(cdf_nl - 0.025))
index_upper = np.argmin(np.abs(cdf_nl - 0.975))
print('lower and higher 95%% confidence bounds of x_nl respectively: (%.3f, %.3f)' %
      (x[index_lower], x[index_upper]))
```

```
lower and higher 95% confidence bounds of x_nl respectively: (0.073, 0.096)
```

d) As it is indicated in the assignment, the joint distribution $P(X_l, X_{nl}|data)$ must be computed as the outer product of `posterior_l` and `posterior_nl`. This can be achieved using the `outer` function in the `numpy` library:

```
joint_posterior = np.ma.outer(posterior_l, posterior_nl)
```

Similar to `Matlab`'s `imagesc`, `Python`'s `imshow` is used to visually represent the joint distribution. The code and the output follows:

```
plt.figure(figure_num)
figure_num += 1
plt.imshow(joint_posterior)
plt.colorbar()
plt.title('The joint posterior distribution')
plt.xlabel('x_nl')
plt.ylabel('x_l')
plt.xticks(np.arange(len(x), step=100),
           (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1))
plt.yticks(np.arange(len(x), step=100),
           (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1))
plt.show(block=False)
```

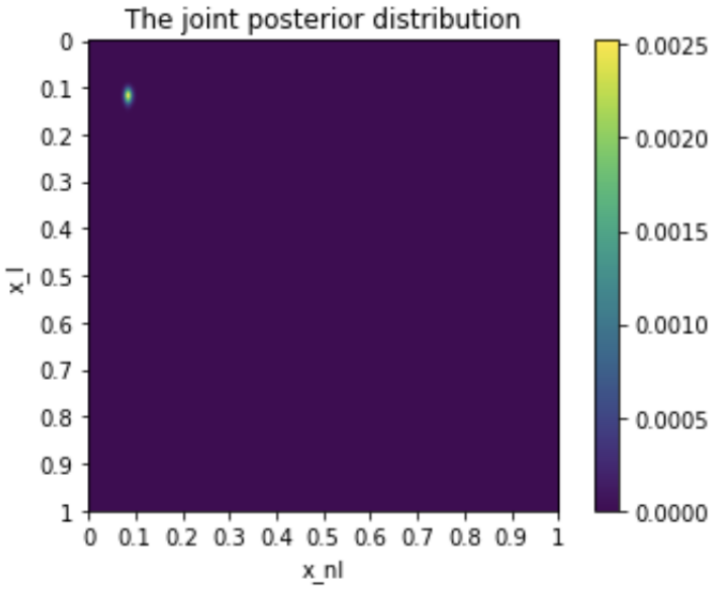

Figure 15: The joint distribution (`joint_posterior`)

The value of $P(X_l > X_{nl}|data)$ can be computed by summing the lower triangular part of the joint distribution matrix. This is because of the fact that $X_l > X_{nl}$ defines the lower triangular region of the matrix. In a similar manner, $P(X_l \leq X_{nl}|data)$ can be computed by summing the upper triangular part and the diagonal of the joint distribution matrix.

Masking is a common technique to access certain regions of matrices in `numpy`, the following code does the job:

```
# x_l > x_nl corresponds to the lower triangular part
mask = np.zeros((len(x), len(x)), dtype=bool)
mask[np.tril_indices(len(x), k=-1)] = True
sum_x_l_greater = np.sum(joint_posterior[mask])
print('Sum of the posterior probabilities such that x_l > x_nl: ', sum_x_l_greater)
# x_l <= x_nl corresponds to the diagonal and upper triangular parts
sum_x_l_less_equal = np.sum(joint_posterior[np.logical_not(mask)])
print('Sum of the posterior probabilities such that x_l <= x_nl: ', sum_x_l_less_equal)
```

The resulting output is given below:

```
Sum of the posterior probabilities such that x_l > x_nl: 0.9978520275861361
Sum of the posterior probabilities such that x_l <= x_nl: 0.002147972413864108
```

e) Again, Baye's Rule should be applied to find $P(language|activation)$, where we are given the value of $P(language)$. Using the rule as in (9), we can write:

$$P(language|activation) = \frac{P(activation|language)P(language)}{P(activation)} \tag{14}$$

Using the additivity axiom of probability, which states the probability of a certain event, say A, is the sum of the probabilities of mutually exclusive events whose union gives A, we can write $P(activation)$ as:

$$\begin{aligned} P(activation) = {} & P(activation|language)P(language) + \\ & P(activation|no\ language)P(no\ language) \end{aligned} \tag{15}$$

It is given that $P(language) = 0.5$, thus $P(no\ language)$ is simply $1 - 0.5 = 0.5$. It is also given that $P(activation|language) = max(x_l)$ and $P(activation|no\ language) = max(x_{nl})$, which are already computed in part **b**. The only work is to plug these values in the right hand side of (15).

The calculation can be done with the following script:

```
PROB_L = 0.5 # probability of language involvement
prob_nl = 1 - PROB_L
prob_active = x_l_max * PROB_L + x_nl_max * prob_nl # additivity axiom
# Apply Bayes Rule
prob_l_given_active = x_l_max * PROB_L / prob_active
print('P(language|activation) is computed as: ', prob_l_given_active)
```

This script produces the output:

```
P(language|activation) is computed as: 0.5833333333333334
```

In conclusion, reverse inference comes in very handy in this case. We can say that whenever the Broca's area is active there is a language involving task with 58.3% probability.

Question 4

a) Given many samples from a multidimensional Gaussian with zero mean and unit covariance; adjusting the samples such that they give the impression that they came from a Gaussian with a specified mean, is a relatively easy task. We denote the specified mean with μ, then shifting each sample by μ is enough to change the sample mean.

However, adjusting the samples so that they have a specified sample covariance is relatively hard. As it is given in the question directive, we need to multiply each sample with a matrix Y such that the covariance matrix, *cov*, has the decomposition $Y^\top Y$. There are various ways of computing such a Y matrix, we proceed with a way that uses Singular Value Decomposition (SVD).

Before, we begin introducing the procedure to find Y; we state that the matrix *cov* is positive semi-definite. The condition for a matrix, say A, to be postive semi-definite is:

$$u^\top Au \geq 0, \text{ for any arbitrary vector } u$$

According to this condition *cov* is positive semi-definite, this can be proved using the definition of a covariance matrix:

$$\text{Recall that: } cov = E[(x-\mu)(x-\mu)^\top] \text{ , then}$$

$$\mathrm{u}^\top(cov)u = u^\top E[(x-\mu)(x-\mu)^\top]u$$

$$= \mathrm{E}[\mathrm{u}^\top(x-\mu)(x-\mu)^\top u]$$

$$= \mathrm{E}[((\mathrm{x}\text{ - }\mu)u)^2] \geq 0$$

$A = U\Sigma V^\top$ for any matrix A. However, if A is a symmetric positive definite matrix, its eigenvectors are orthogonal and the decomposition becomes $A = Q\Lambda Q^\top$ (this is by the Spectral Theorem [5]). *cov* matrix is trivially symmetric, and positive "semi-definite" as proven above. We can exclude the corner case where $u^\top(cov)u = 0$ since this case is not possible in practice and write:

$$cov = U\Sigma V^\top = Q\Lambda Q^\top$$

Here Λ gathers the eigenvalues as its diagonal entries and Q gathers the corresponding eigenvectors. Since Λ is a diagonal matrix:

$$cov = Q\sqrt{\Lambda}\sqrt{\Lambda}Q^\top = Y^\top Y, \text{ where } Y = \sqrt{\Lambda}Q^\top$$

A `Python` method to find the decomposing matrix Y can be written as follows:

```
def decompose(A):
    """
    Given a positive definite matrix A, computes and returns a
    matrix Y such that Y^T.Y decomposes A. This decomposition
    is a special case of SVD for positive definite matrices.
    Args:
        A: A positive definite matrix
    Returns:
        Y: A matrix such that A = Y^T.Y
    """
    Q, L, Q_T = np.linalg.svd(A)
    L_sqrt = np.sqrt(np.diag(L))
    Y = L_sqrt.dot(Q_T)
    return Y
```

Then following the procedure mentioned above, we can write the function `ndRandn(mean, cov, num)` as follows:

```
def ndRandn(mean, cov, num=1):
    """
    Generates a set of samples drawn from a multidimensional
    Gaussian distribution with the specified mean (an N-vector)
    and covariance (an NxN matrix). The parameter num specifies
    the number of samples to return (default is 1).
    Args:
        mean: The mean of the multidimensional Gaussian
            distribution (an N-vector)
        cov: The covariance of the multidimensional Gaussian
            distribution (an NXN matrix)
        num: Number of samples to return (default is 1)
    Returns:
        samples: A set of samples drawn from a multidimensional
            Gaussian distribution (stored as a 2D numpy array)
    """
    samples = []
    N = np.size(cov, 0)
    zero_mean = np.zeros(N)
    ident_cov = np.identity(N)
    for _ in range(num):
        # Prepare a zero mean, unit covariance sample, that will later be adjusted
        sample_ident_cov = np.random.multivariate_normal(zero_mean, ident_cov)
        # We can apply a special case of SVD to decompose A as Y^T.Y
        # Y = np.linalg.cholesky(cov).T # alternative library function
        Y = decompose(cov)
        # Adjust the sample to have the specified mean and covariance
        sample = sample_ident_cov.dot(Y)
        sample += mean
```

```
30          samples.append(sample)
31      return np.array(samples)
```

b) What remains is to plot a standard deviational ellipse that captures our samples.

$x^\top(cov)x = c$, defines a multidimensional standard deviational ellipse, where c is a constant

Again using the spectral theorem, we can write:

$$x^\top(cov)x = x^\top Q\Lambda Q^\top x = y^\top \Lambda y$$

This is basically a projection of the x and y axes to the eigenvectors of cov. The following `Python` function makes us of this projection to find the rotation angle, height and width the standard deviational ellipse:

```
1  def get_standard_deviational_ellipse(sample_mean, sample_cov):
2      """
3      Generates the standard deviational ellipse
4      for the data drawn from a particular 2-dimensional Gaussian
5      Distribution.
6      Args:
7          sample_mean: The mean of the 2-dimensional Gaussian
8              distribution samples (a 2-vector)
9          sample_cov: The covariance of the 2-dimensional Gaussian
10             distribution samples(a 2X2 matrix)
11     Returns:
12         error_ellipse: The Ellipse object representing
13             the standard deviational ellipse
14     """
15     eigvals, eigvecs = np.linalg.eig(sample_cov)
16     descending_order = eigvals.argsort()[::-1]
17     eigvals = eigvals[descending_order]
18     eigvecs = eigvecs[descending_order]
19     theta = np.rad2deg(-np.arctan2(eigvecs[0, 1], eigvecs[0, 0]))
20     width = np.sqrt(eigvals[0]) * 2 * 2
21     height = np.sqrt(eigvals[1]) * 2 * 2
22     error_ellipse = Ellipse(xy=sample_mean, width=width, height=height,
23                             angle=theta, edgecolor='r', fc='None', lw=2)
24     return error_ellipse
```

The endpoints of the ellipse given by $y^\top \Lambda y = c$ (an ellipse with axes aligning with the eigenvectors) are given by $y_i = \pm c\sqrt{\lambda_i}$ for each direction i. This is the reason we computed the square roots of the eigenvalues in the 20th and 21th lines. Note that $c = 2$ since we want to trace out the points that are two standard deviations away.

Now that we have all the necessary **Python** functions ready, we can test `ndRandn` with the following script:

```
mean = np.array([2, 10])
cov = np.array([[1, −1],
                [−1, 4]])
samples = ndRandn(mean, cov, 1000)
sample_mean = np.mean(samples, axis=0)
print('Mean of the samples is:\n', sample_mean)
sample_cov = np.cov(samples, rowvar=False)
print('Covariance of the samples is:\n', sample_cov)
x = [sample[0] for sample in samples]
y = [sample[1] for sample in samples]
plt.scatter(x, y, s=30)
ellipse = get_standard_deviational_ellipse(sample_mean, sample_cov)
plt.figure(1)
plt.title('The Standard Deviational Ellipse')
plt.gca().add_patch(ellipse)
plt.show()
```

The output of the script and the figure it plots are as follows:

```
Mean of the samples is:
 [2.03628352 9.99956127]
Covariance of the samples is:
 [[ 0.86961785 −0.85409448]
 [−0.85409448 3.92467049]]
```

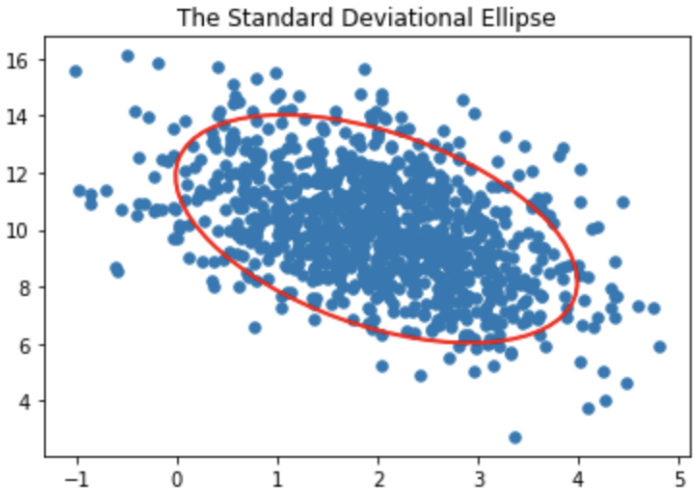

Figure 16: The Standard Deviational Ellipse

As it can be seen, the sample mean and the sample covariance are very close to the specified values and the standard deviational ellipse captures the shape of the samples. This confirms that the `ndRandn` function works properly.

Note that some functions from `Numpy` and `Matplotlib` libraries are used in the solution code and they should be imported.

References

[1] "Moorepenrose inverse," En.wikipedia.org, 2019, [Accessed: 24- Feb- 2019]. [Online]. Available: https://en.wikipedia.org/wiki/Moore-Penrose_inverse

[2] "Understanding q-q plots — university of virginia library research data services + sciences," Data.library.virginia.edu, 2019, [Accessed: 18- Feb- 2019]. [Online]. Available: https://data.library.virginia.edu/understanding-q-q-plots/

[3] "Bootstrapping (statistics)," En.wikipedia.org, 2019, [Accessed: 18- Feb- 2019]. [Online]. Available: https://en.wikipedia.org/wiki/Bootstrapping_(statistics)

[4] "Jackknife resampling," En.wikipedia.org, 2019, [Accessed: 18- Feb- 2019]. [Online]. Available: https://en.wikipedia.org/wiki/Jackknife_resampling

[5] "Spectral theorem," En.wikipedia.org, 2019, [Accessed: 25- Feb- 2019]. [Online]. Available: https://en.wikipedia.org/wiki/Spectral_theorem