

Question 1

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 were necessary. The Python solution code for this question, assumes that the following lines are executed:

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

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

```
1 data = loadmat('XData.mat')
2 x_data = data['x'].flatten()
3 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:

```
1 mean = np.mean(x_data)
2 median = np.median(x_data)
3 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-quartile 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:

```
1 std = np.std(x_data)
2 q25, q75 = np.percentile(x_data, [25, 75])
3 iqr = q75 - q25
4 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 asked histograms is given below:

```
1 x_range = [70, 130]
2 num_bins = [3, 6, 12]
3 colors = ['g', 'r', 'b']
4
5 print('The histograms:')
6 for i in range(3):
7     plt.figure(figsize=(10, 6))
8     figure_num += 1
9     plt.title('Histogram of the aggregated responses')
10    plt.ylabel('Counts')
11    plt.xlabel('Aggregated responses')
12    plt.hist(x_data.flatten(), bins=num_bins[i], range=x_range, color=colors[i], edgecolor='black', linewidth=1.2)
13    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.

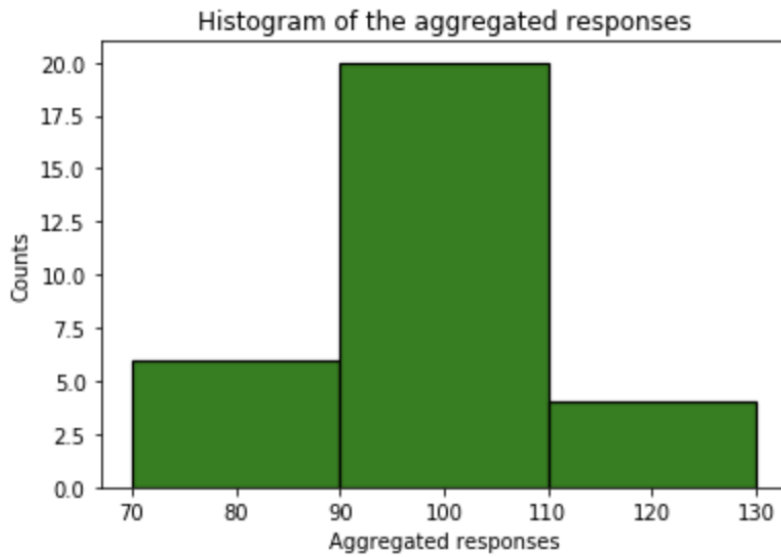


Figure 1: Histogram of the aggregated responses with 3 bins.

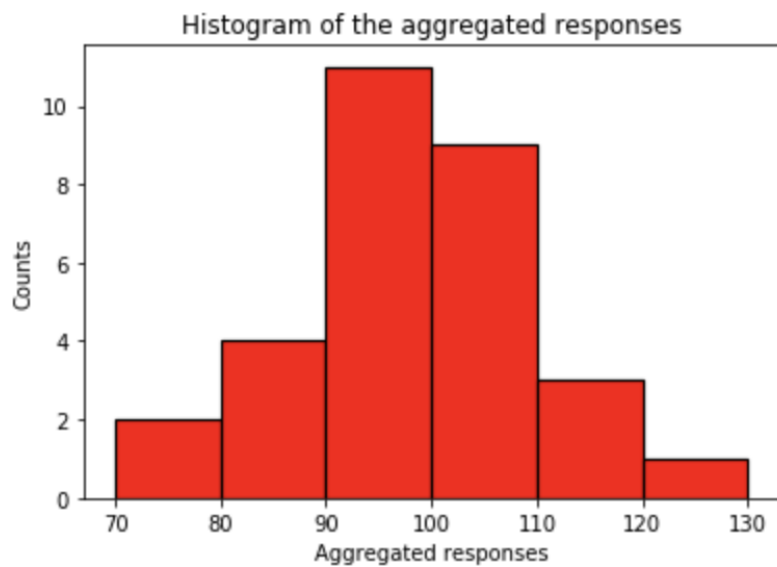


Figure 2: Histogram of the aggregated responses with 6 bins.

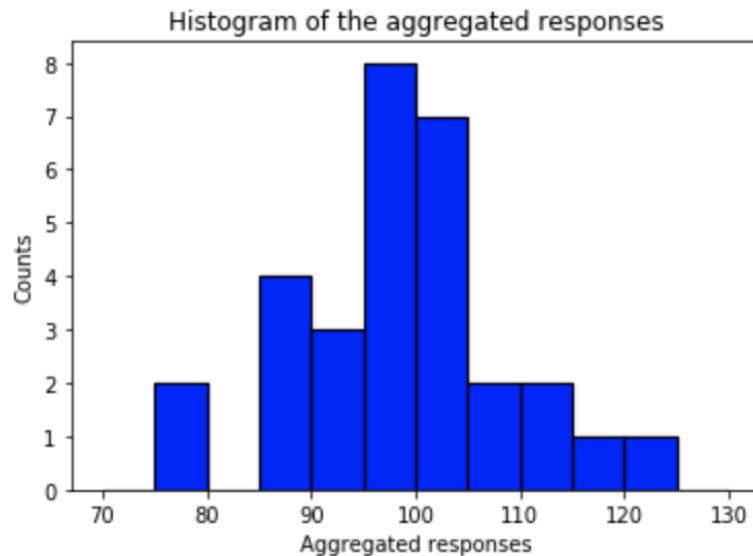


Figure 3: Histogram of the aggregated responses with 12 bins.

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. [1]

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

```
1 # Python equivalent of the Matlab Normal Quantile Plot code given in the assignment
2 y = np.sort(x_data)
3 n = np.size(x_data)
4 f = (np.arange(1, n + 1) - 3 / 8) / (n + 1 / 4)
5 q = 4.91 * (f ** 0.14 - (1 - f) ** 0.14)
6 plt.grid()
7 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.

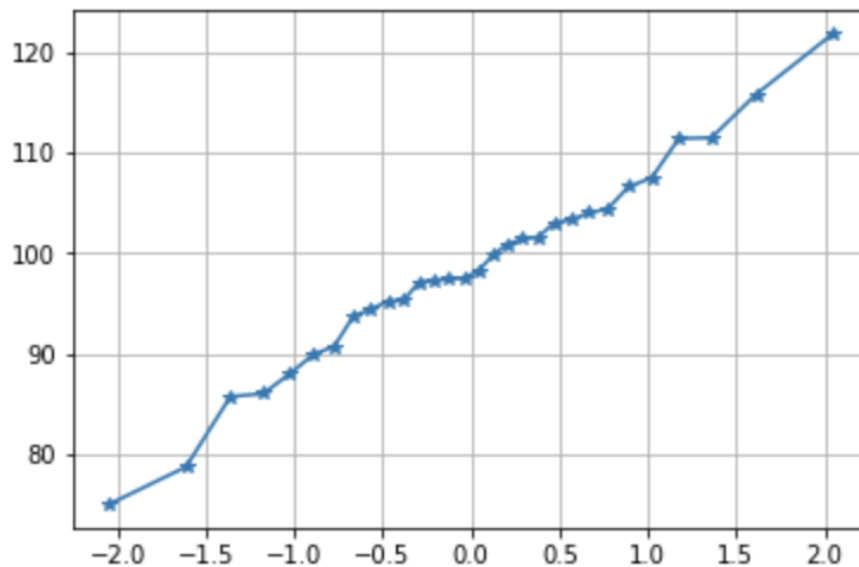


Figure 4: Histogram of the aggregated responses with 12 bins.

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. [2]

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

```

1  # Function to perform bootstrap resampling
2  def bootstrap_resampling(data, bootstraps, stat_func):
3      """
4      Given the data, number of bootstrapping iterations and the statistic of
5      interest; this function returns a list of statistics obtained from each
6      bootstrapped sample of the original data.
7      Args:
8          data: The original data
9          bootstraps: The number of bootstrapping iterations
10         stat_func: The function to compute the statistic of interest
11                   (e.g.: np.mean, np.std, etc.)
12     Returns:
13         bootstrap_replicate: The mean of all bootstrap replicates
14         standard_error: The standard error of the bootstrap replicate
15         bootstrap_replicates: The list of statistics obtained from the
16                               bootstrapped samples
17     """

```

```

18     bootstrap_replicates = []
19     sample_size = np.size(data)
20     for _ in range(bootstraps):
21         bootstrap_sample = np.random.choice(data, sample_size, replace=True)
22         bootstrap_replicates.append(stat_func(bootstrap_sample))
23     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:

```

1  # Function to compute the confidence interval of data samples
2  def compute_confidence_interval(data, confidence):
3      """
4      Given the data and the confidence level, computes the confidence interval
5      of the data samples.
6      Args:
7          data: The given data
8          confidence: The confidence level, known as alpha (between 0 and 100)
9      Returns:
10         lower: The lowerbound of the confidence interval
11         upper: The upperbound of the confidence interval
12     """
13     sorted_data = np.sort(data)
14     lower = np.percentile(sorted_data, (100 - confidence) / 2)
15     upper = np.percentile(sorted_data, confidence + (100 - confidence) / 2)
16     return lower, upper

```

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

```

1  # Performing the bootstrap sampling and computing the bootstrap replicate
2  # of the mean together with its standard error
3  bootstrap_mean, standard_error_mean, bootstrap_means = bootstrap_resampling(x_data, 1000, np.mean)
4  print('The sample mean computed from %s bootstrap samples: %f' % (1000, bootstrap_mean))
5  print('The bootstrapped estimate for the standard error of the mean:', standard_error_mean)
6
7  # Computing the 95% confidence interval
8  lower, upper = compute_confidence_interval(bootstrap_means, 95)
9  confidence_interval = (lower, upper)
10 print('The 95% confidence interval of the mean:', confidence_interval)
11
12 print('Overall, mean of the dataset can be reported as: %.3f +- %.3f, (%.3f, %.3f)\n' %
13       (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: (95.122416666666664, 102.170500000000002)
Overall, mean of the dataset can be reported as: 98.438  $\pm$  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)
```

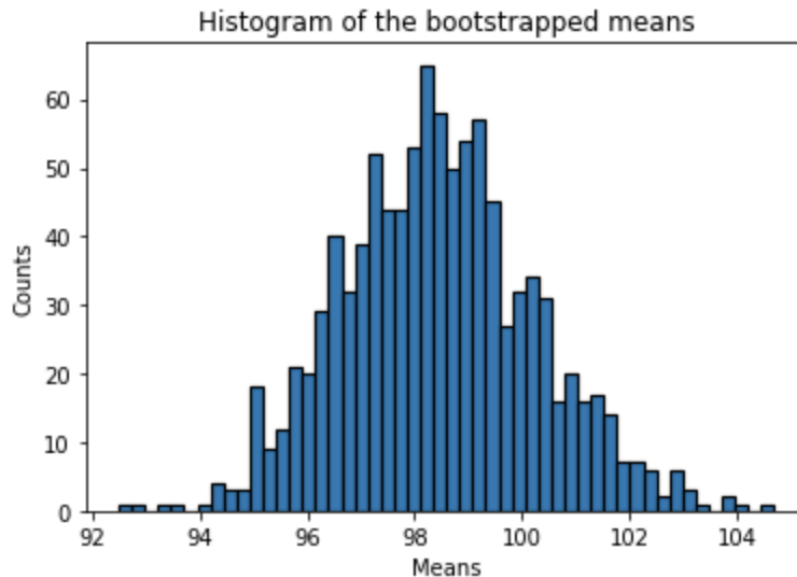


Figure 5: Histogram of the means of the bootstrap resamples with 50 bins.

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
```



```

3 bootstrap_std, standard_error_std, bootstrap_stds = bootstrap_resampling(x_data, 1000, np.std)
4 print('The sample standard deviation computed from %s bootstrap samples: %f' % (1000, bootstrap_std))
5 print('The bootstrapped estimate for the standard error of the standard deviation:', standard_error_std)
6
7 # Computing the 95% confidence interval
8 lower, upper = compute_confidence_interval(bootstrap_stds, 95)
9 confidence_interval = (lower, upper)
10 print('The 95% confidence interval of the standard deviation:', confidence_interval)
11
12 print('Overall, standard deviation of the dataset can be reported as: %.3f +- %.3f, (%.3f, %.3f)\n'
13       % (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)

```

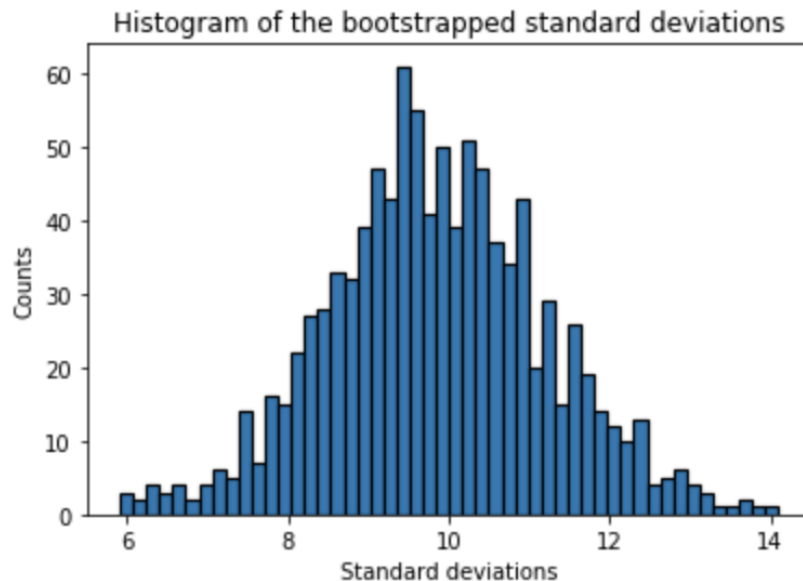


Figure 6: Histogram of the standard deviations of the bootstrap resamples with 50 bins.

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. [3]

A function that performs jackknife resampling is provided below:

```

1  # Function to perform Jackknife resampling
2  def jackknife_resampling(data, stat_func):
3      """
4      Given the data and the statistic of interest; this function returns a
5      list of statistics obtained from each jackknifed sample of the original data.
6      Args:
7          data: The original data
8          stat_func: The function to compute the statistic of interest
9                  (e.g.: np.mean, np.std, etc.)
10     Returns:
11         jackknife_replicate: The mean of all jackknife replicates
12         standard_error: The standard error of the jackknife replicate
13         jackknife_replicates: The list of statistics obtained from the
14                             jackknifed samples
15     """
16     jackknife_replicates = []
17     jackknifes = np.size(data)
18     for i in range(jackknifes):
19         jackknife_data = [elem for j, elem in enumerate(data) if i != j]
20         jackknife_replicate_i = stat_func(jackknife_data)
21         jackknife_replicates.append(jackknife_replicate_i)
22     return np.mean(jackknife_replicates), np.std(jackknife_replicates), jackknife_replicates

```

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

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

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:

```

1  # Performing the jackknife sampling and computing the jackknife replicate
2  # of the mean together with its standard error
3  jackknife_mean, standard_error_mean, jackknife_means = jackknife_resampling(x_data, np.mean)
4  print('The sample mean computed from jackknife samples:', jackknife_mean)
5  print('The jackknifed estimate for the standard error of the mean:', standard_error_mean)
6
7  # Computing the 95% confidence interval
8  lower, upper = compute_confidence_interval(jackknife_means, 95)
9  confidence_interval = (lower, upper)
10 print('The 95% confidence interval of the mean:', confidence_interval)
11
12 print('Overall, mean of the dataset can be reported as: %.3f +- %.3f, (%.3f, %.3f)\n' %
13       (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 \pm 0.347, (97.781, 99.152)

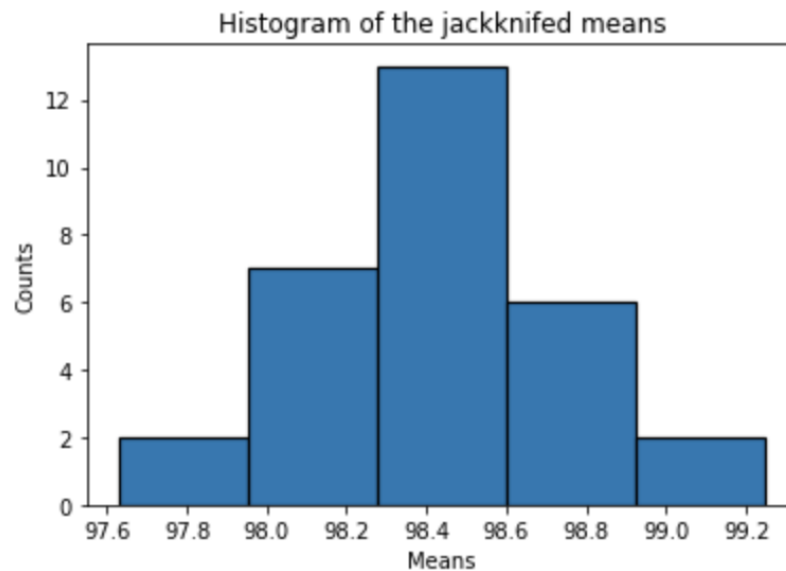


Figure 7: Histogram of the means of the jackknife resamples with 5 bins.

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:

```

1  # Performing the jackknife sampling and computing the jackknife replicate
2  # of the standard deviation together with its standard error
3  jackknife_std, standard_error_std, jackknife_stds = jackknife_resampling(x_data, np.std)
4  print('The sample standard deviation computed from jackknife samples:', jackknife_std)
5  print('The jackknifed estimate for the standard error of the standard deviation:', standard_error_std)
6
7  # Computing the 95% confidence interval
8  lower, upper = compute_confidence_interval(jackknife_stds, 95)
9  confidence_interval = (lower, upper)
10 print('The 95% confidence interval of the standard deviation:', confidence_interval)
11
12 print('Overall, standard deviation of the dataset can be reported as: %.3f +- %.3f, (%.3f, %.3f)\n' %
13       (jackknife_std, standard_error_std, lower, upper))

```

The output of these lines and the histogram of the standard deviations of the jackknife re-samples 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)

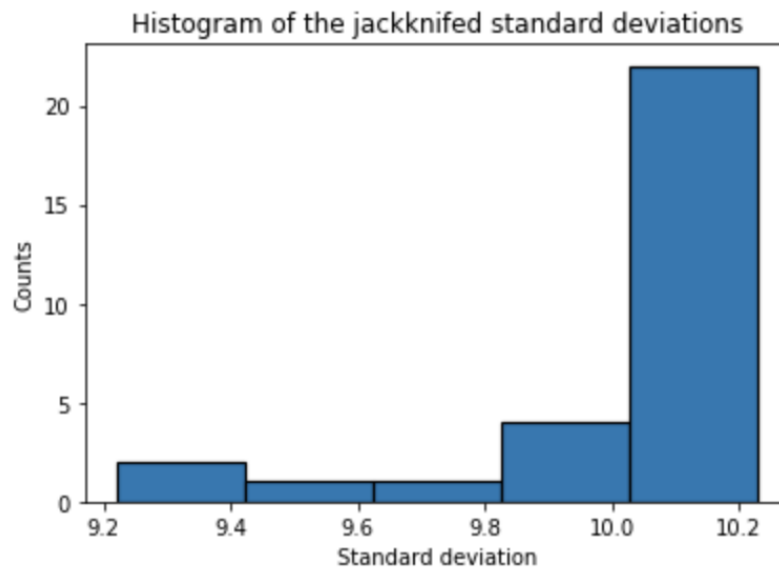


Figure 8: Histogram of the standard deviations of the jackknife resamples with 5 bins.

Question 3

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

```
1 # Necessary imports
2 import numpy as np
3 import matplotlib.pyplot as plt # for bar plots
4 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:

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

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 there 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:

$$L(data|x_l) = \binom{809}{103} \prod_{data_i} P(data_i|x_l) = \binom{809}{103} x_l^{103} (1 - x_l)^{809-103} = \frac{809!}{766! 103!} x_l^{103} (1 - x_l)^{766} \quad (2)$$

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

$$L(data|x_{nl}) = \binom{2353}{119} \prod_{data_i} P(data_i|x_{nl}) = \binom{2353}{119} x_{nl}^{119} (1 - x_{nl})^{2353-119} = \frac{2353!}{2234! 119!} x_{nl}^{119} (1 - x_{nl})^{2234} \quad (3)$$

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:

```

1 def likelihood_beurnoulli(prob, num_positive, total):
2     """
3     Likelihood function of the beurnoulli distribution.
4     Args:
5         prob: The probability that a binary event has a postive outcome
6         num_positive: Number of positive outcomes
7         total: Total number of observations/trials
8     Returns:
9         likelihood: The likelihood of having the beurnoulli distribution
10        with the specified arguments
11    """
12    num_negative = total - num_positive
13    likelihood = (prob ** num_positive) * ((1 - prob) ** num_negative)
14    likelihood *= comb(total, num_positive)
15    return likelihood

```

The likelihood functions given in (2) and (3) are computed for the asked values of x_l and x_{nl} using the following lines:

```

1 x = np.arange(0, 1.001, 0.001) # probability values to try
2 # Compute the likelihood vectors for language invloving tasks and others
3 likelihoods_l = likelihood_beurnoulli(x, ACTIVE_L, TOTAL_L)
4 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 function at some discrete values of x_l and x_{nl} .

A bar chart is useful to visualize these discretized likelihood function and it can be plotted for `likelihoods_l` using the lines:

```

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

```

The output of these lines is given below:

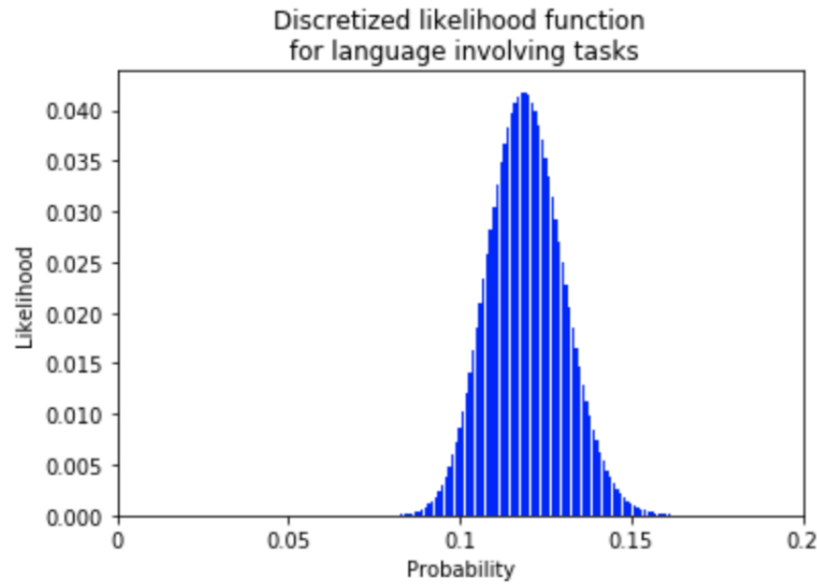


Figure 9: Discretized likelihood function for language involving tasks (`likelihoods_l`)

The bar chart for `likelihoods_n1` is plotted using the same script where very few of the arguments are changed, the resulting output follows:

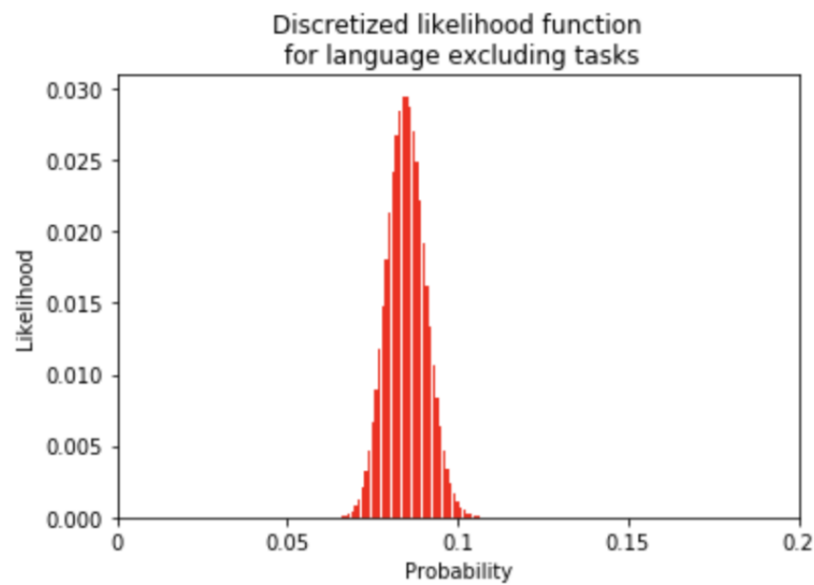


Figure 10: Discretized likelihood function for language excluding tasks (`likelihoods_n1`)

b) We are asked to find the values of x_l and x_{nl} that maximizes the *discretized* counterparts of (2) and (3), so it suffices to find the probabilities that maximizes 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:

```

1 x_l_max = x[np.argmax(likelihoods_l)]
2 print('The probability that maximizes the discretized likelihood of the data ' +
3       'obtained from the language involving tasks: %.3f, %.5f' % (x_l_max,
4       likelihood_beurnoulli(x_l_max, ACTIVE_L, TOTAL_L)))
5 x_nl_max = x[np.argmax(likelihoods_nl)]
6 print('The probability that maximizes the discretized likelihood of the data ' +
7       'obtained from the language excluding tasks: %.3f, %.5f' % (x_nl_max,
8       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:

$$P(X|data)P(data) = P(data|X)P(X) \quad (4)$$

$$P(X|data) = \frac{P(data|X)P(X)}{P(data)} \quad (5)$$

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

$$P(X|data) = \frac{P(data|X)P(X)}{\sum_x P(data|x)p(x)} \quad (6)$$

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

$$P(X) = \frac{1}{|S_X|} = \frac{1}{1001} = P(X = x), \forall x \in S_X \quad (7)$$

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:

$$P(X|data) = \frac{P(data|X)}{\sum_x P(data|x)} \quad (8)$$

$P(data|X)$ is the likelihood that is already computed, $P(X|data)$ can be easily calculated from (8). However, the code below calculates $P(X|data)$ for language involving tasks using (6) and (7) because of the fact that (8) is not a generic formula but rather a special case. The code follows:

```

1 # Assumption is a uniform prior
2 uniform_prior = 1 / len(x)
3 # Marginalize the prior out in the conditional distribution to compute
4 # the normalizer
5 normalizer_l = np.sum(likelihoods_l * uniform_prior)
6 # Apply Bayes rule
7 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:

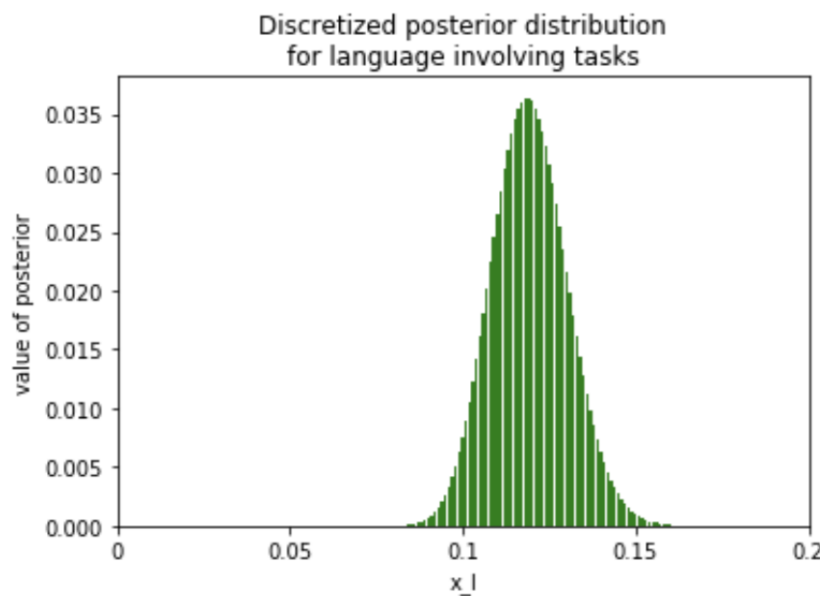


Figure 11: Discretized posterior distribution for language including tasks (`posterior_l`)

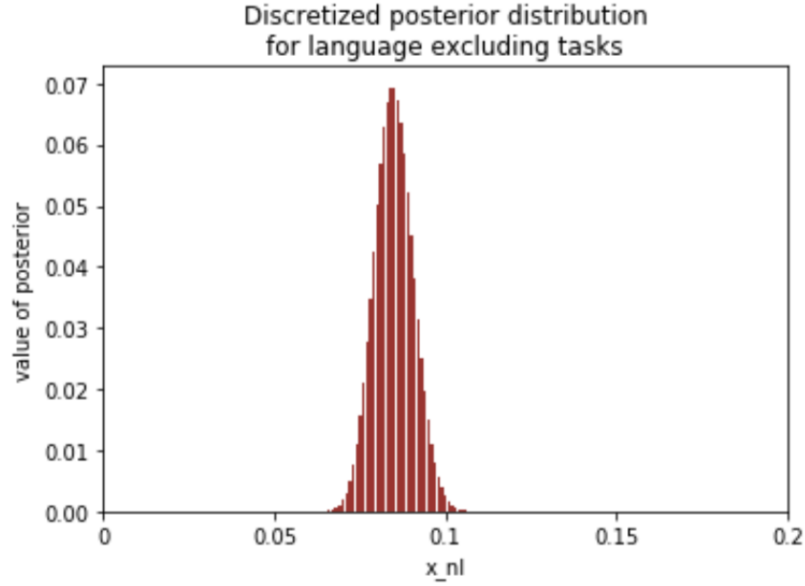


Figure 12: Discretized posterior distribution for language excluding tasks (`posterior_nl`)

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 97.5th percentile. The conditional cumulative distribution function (c.d.f) is given as:

$$P(X \leq x|data) = F(X|data) = \sum_{x_i \leq x} P(x|data) \quad (9)$$

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 (10) 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:

```
1 plt.figure(figsize=(10, 6))
2 figure_num += 1
3 plt.bar(np.arange(len(x)), cdf_l, color='navy')
4 plt.xticks(np.arange(0, 1, step=0.1),
5            (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1))
6 plt.xlabel('x_l')
7 plt.ylabel('value of cumulative posterior')
8 plt.title('Cumulative posterior distribution\nfor language involving tasks')
9 plt.show(block=False)
```

The output figure is:

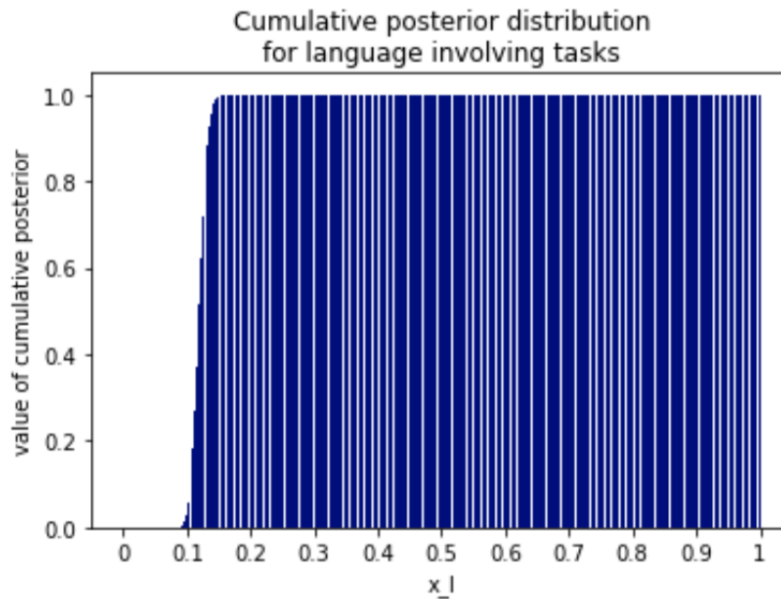


Figure 13: Cumulative posterior distribution for language involving tasks (`cdf_l`)

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

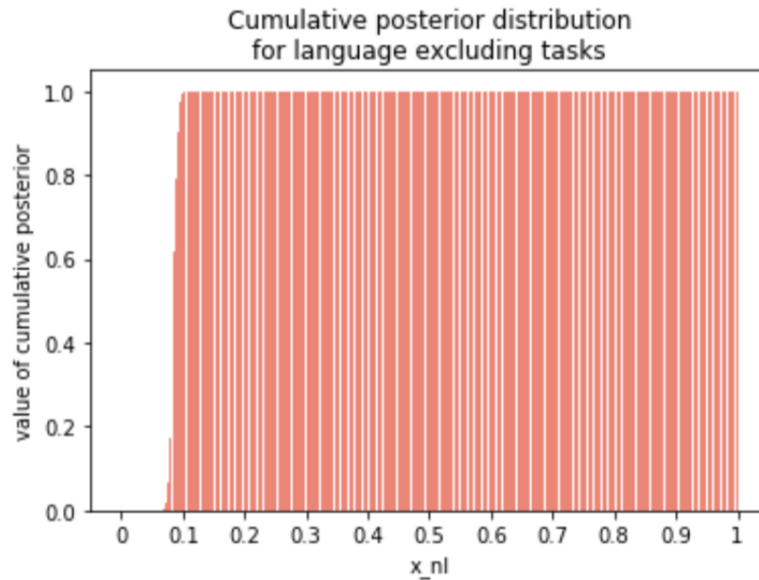


Figure 14: Cumulative posterior distribution for language excluding tasks (`cdf_nl`)

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:

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

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

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

lower and higher 95% confidence bounds of x_{nl} respectively: (0.073, 0.096)

d) As it is said 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:

```
1 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:

```
1 plt.figure(figsize=(10, 10))
2 figure_num += 1
3 plt.imshow(joint_posterior)
4 plt.colorbar()
5 plt.title('The joint posterior distribution')
6 plt.xlabel('x_nl')
7 plt.ylabel('x_l')
8 plt.xticks(np.arange(0, 1, step=0.1),
9            (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1))
10 plt.yticks(np.arange(0, 1, step=0.1),
11            (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1))
12 plt.show(block=False)
```

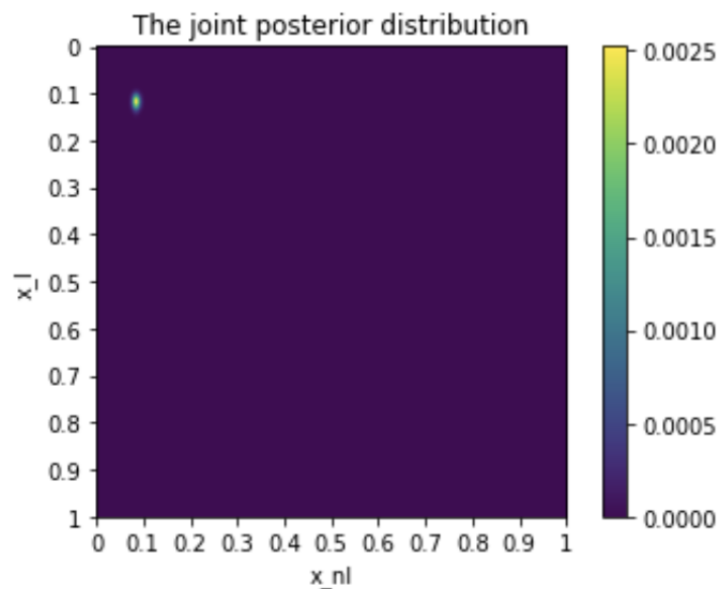


Figure 15: The joint distribution (`joint_posterior`)

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:

```

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

$$P(language|activation) = \frac{P(activation|language)P(language)}{P(activation)} \quad (10)$$

Using the additivity axiom of probability, which states that 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:

$$P(activation) = P(activation|language)P(language) + P(activation|no language)P(no language) \quad (11)$$

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 (11).

The calculation can be done with the following script:

```
1 PROBL = 0.5 # probability of language involvement
2 prob_nl = 1 - PROBL
3 prob_active = x_l_max * PROBL + x_nl_max * prob_nl # additivity axiom
4 # Apply Bayes Rule
5 prob_l_given_active = x_l_max * PROBL / prob_active
6 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

References

- [1] “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/>
- [2] “Bootstrapping (statistics),” En.wikipedia.org, 2019, [Accessed: 18- Feb- 2019]. [Online]. Available: [https://en.wikipedia.org/wiki/Bootstrapping_\(statistics\)](https://en.wikipedia.org/wiki/Bootstrapping_(statistics))
- [3] “Jackknife resampling,” En.wikipedia.org, 2019, [Accessed: 18- Feb- 2019]. [Online]. Available: https://en.wikipedia.org/wiki/Jackknife_resampling