Batu Arda Düzgün 21802633
EEE-482 HW-1

Question 1:

Given the system of equations:

$$\begin{bmatrix}1 & 0 & -1 & 2\\ 2 & 1 & -1 & 5\\ 3 & 3 & 0 & 9\end{bmatrix}\cdot\begin{bmatrix}x_1\\ x_2\\ x_3\\ x_4\end{bmatrix}=\begin{bmatrix}1\\ 4\\ 9\end{bmatrix}$$

$$Ax = b$$

a) Find all solutions x_n to $Ax = 0$.

The question wants us to find the null space of A putting it differently.

To find all solutions of x_n I first used gaussian elimination to reduce the A matrix to its row echelon form.

$$\begin{bmatrix}1 & 0 & -1 & 2\\ 2 & 1 & -1 & 5\\ 3 & 3 & 0 & 9\end{bmatrix}\underset{R_3 = R_3 - 3R_1}{\xrightarrow{R_2 = R_2 - 2R_1}}\begin{bmatrix}1 & 0 & -1 & 2\\ 0 & 1 & 1 & 1\\ 0 & 3 & 3 & 3\end{bmatrix}\xrightarrow{R_3 = R_3 - 3R_2}\begin{bmatrix}1 & 0 & -1 & 2\\ 0 & 1 & 1 & 1\\ 0 & 0 & 0 & 0\end{bmatrix} \tag{1}$$

After calculating the row echelon form of A in equation 1, I put it back into the $Ax = 0$ equation. The row operation here actually are just multiplying both sides of the equations from left with a matrix which converts A to its row echelon form. Actually all row operations are multiplication with a matrix from left.

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

We can see from equation 2 that x_1 and x_2 are pivot variables and x_3 and x_4 are free variables. Now If we can write x_1 and x_2 in terms of x_3 and x_4 we can express all solutions of x_n to $Ax = 0$.

Then to do this I did the matrix multiplication in equation 2 to get the 2 equations in equation 3 and equation 4

$$x_1 - x_3 + 2x_4 = 0 \tag{3}$$

$$x_2 + x_3 + x_4 = 0 \tag{4}$$

Using equation 3 we can write x_1 as $x_1 = x_3 - 2x_4$ and using equation 4 we can write x_2 as $x_2 = -x_3 - x_4$ the we can write x_n as:

$$x_n = \begin{bmatrix}x_3 - 2x_4\\ -x_3 - x_4\\ x_3\\ x_4\end{bmatrix} = x_3\begin{bmatrix}1\\ -1\\ 1\\ 0\end{bmatrix} + x_4\begin{bmatrix}-2\\ -1\\ 0\\ 1\end{bmatrix}\ for\ x_3, x_4\ \in \mathbb{R} \tag{5}$$

Equation 5 is general solution of x_n for all solutions x_n to $Ax = 0$.

Then to verify the results I used python 3.8, python works almost fully on library's and without them it almost doesn't have any functions for what we want to use it for here. Because of this I imported some packages related to the tasks we are given. I paid attention to not use any library that doesn't already come with the Anaconda install so all this library's already should come with python and don't require an extra download. I wrote the import part of my code down below. If your python doesn't have the used packages and you have pip, you can install them missing packages with command in the command with command such as "python -m pip install numpy, python -m pip install scipy or python -m pip install matplotlib."

```
import sys # the primer already has it
import numpy as np # this brings simple matrix operations to python
import matplotlib.pyplot as plt # this brings nice loking plots to python

import random # for random number generation in some verifications

from scipy import linalg as lg # this brings complicated matrix operations to pyt
hon and elivates it to the level of matlab

import math as math # this brings mathematical operations to python
```

Now if we return to the question given in part a. Python has a function to calculate the null space of a given matrix (null_space(A)), this is what we are asked for in part a so it is perfect for the situation. But there is a problem null_space(A) return the null space matrix as an orthonormal matrix. To make the output of the function look same as what I calculated by hand I also did some column operation on the output of the null_space(A) function. My code for this part is down below

```

    A = np.array([ [1, 0, -1, 2] , [2, 1, -1, 5] , [3, 3, 0, 9]])
# this is the instensiation of the A matrix given in the question

    NS = lg.null_space(A) # this finds the null space of the given matrix

    CNS = NS.copy() # I am doing these steps to put the null space in to a format
#close to what I found by hand.

    C0 = CNS[-1,0]
    C1 = CNS[-1,1]

    CNS[: ,0] -= CNS[: , 1] * C0 / C1

    C0 = CNS[-2,0]

    CNS[: ,0] /= C0
```

```
C0 = CNS[-2,0]
C1 = CNS[-2,1]

CNS[: ,1] -= CNS[: , 0] * C1 / C0

C1 = CNS[-1,1]

CNS[: ,1] /= C1

print("The null space of the matrix is ")
print(CNS)
```

And it gives the output of:

```
[[ 1. -2.]

 [-1. -1.]

 [ 1.  0.]

 [ 0.  1.]]
```

Which is same as what I calculated by hand. If we multiply this matrix with $\begin{bmatrix} x_3 \\ x_4 \end{bmatrix}$ from right we can get the same general solution as the one in equation 5.

b) Find a particular solution x_p to $Ax = b$

In this part I will use a similar method to what I used in part a, I will use gaussian elimination to reduce the $[A\ |b]$ augmented matrix to its row echelon form.

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

After calculating the row echelon form of $[A\ |b]$ augmented matrix in equation 6, I put it back into the $Ax = b$ equation. We can do this again because the row operation here actually are just multiplying both sides of the equations from left with a matrix which converts A to its row echelon form.

$$\begin{bmatrix} 1 & 0 & -1 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} \tag{7}$$

Then I did the matrix multiplication in equation 7 to get the 2 equations in equation 8 and equation 9

$$x_1 - x_3 + 2x_4 = 1 \tag{8}$$

$$x_2 + x_3 + x_4 = 2 \tag{9}$$

The question asked us to find only one answer because of this I choose $x_3 = x_4 = 0$ to get a simple and clean answer. Then equation 8 and equation 9 simplified to $x_1 = 1, x_2 = 2$. Then We can express a particular solution x_p to $Ax = b$ as:

$$x_p = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} \tag{10}$$

I can not verify this part the same way I verified part a because there are infinitely many different particular solutions so making python calculate a particular solution is meaningless. So instead to verify I will place the x_p I found in equation 10 inside of the $Ax = b$ equation and verify that it satisfies it. I wrote the code used for this calculation down below

```
    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the 
#question

    x_p = np.array([ [1] , [2] , [0] , [0] ]) # this is the instensiation of the 
#x_p I calculated by hand

    b = A.dot(x_p)

    print("For the particular solution I found " + str(x_p.transpose()) + "^T \
 the result is " + str(b.transpose())+ "^T which means it satisfys the equation")
```

And it gives the output of:

```
[[1]
 [4]
 [9]]
```

Which is same with the b vector given in the question. This means the particular solution I calculated satisfies $Ax_p = b$.

c) Find all solutions to $Ax = b$

We know all solutions to $Ax = b$ can be expressed as a particular solution plus a vector from the null space of A because of the superposition. I already calculated the null space of the A matrix in part a and I found a particular solution of $Ax = b$ in part b. Now I can express the answer of this part just as the sum the previous 2 parts answers.

$$x = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} 1 \\ -1 \\ 1 \\ 0 \end{bmatrix} + x_4 \begin{bmatrix} -2 \\ -1 \\ 0 \\ 1 \end{bmatrix} \quad for\ x_3, x_4 \ \in \mathbb{R} \tag{11}$$

Like in part b I used python to verify my solution because we can express this solution set with infinitely many different particular solution and infinitely many different null spaces and all of them would be correct so there is no way to make the software give us the result I calculated. The code I wrote for this part chooses random $x_3, x_4 \ \in \mathbb{R}$ then placed them in equation 11 to calculate the x for those values. Then places the x inside of Ax to see if it gives b as an output. We can run the code as many times as we want to se that it always gives b. I included the code down below.

```
    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the
#question
    x_p = np.array([ [1] , [2] , [0] , [0] ]) # this is the instensiation of the
#x_p I calculated by hand

    x3 = random.random() # this generates a random number between 0 and 1. we can
# multipley 1000 to get a number generation between 0 and 1000 but I didn't do it
# because it shoudln't change the realizim of the test

    x4 = random.random()

    x3x4 = np.array([ [x3] , [x4] ])

    null_space = lg.null_space(A) # this finds the null space of the given matrix

    x = x_p + null_space.dot(x3x4) # creating the random x from our solution set.

    b = A.dot(x)

    print( "for x_3 = " + str(x3) + " and x_4 = " + str(x4) + " b is equal to: ")
    print( b )
```

d) Find the pseudo-inverse of A.

A is not a square matrix so we cannot get it's invers matrix but we can still find its persuade matrix which is what this question is asking us to do.

Persuade inverse is not like the normal inverse. Normal invers satisfies equation 11 but persuade inverse doesn't satisfy equation 11 but it still satisfies equation 12.

$$AA^{-1} = \mathrm{I} \tag{11}$$

$$x = A^{+}b\ for\ Ax = b \tag{12}$$

Because a is not full column rank and not full row rank, we need to use singular value decomposition to calculate A^{+}

Once we calculate SVD of A we can use equation 13 to calculate A^{+}.

$$A = U\Sigma V^{T}$$
$$A^{+} = V\Sigma^{+}U^{T} \tag{13}$$

In equation 13 there is a Σ^{+}term. This is just equal to reciprocal of every none zero term in Σ.

Now first I want to calculate the eigen values of AA^{T} then from that I will compute vectors of U.

$$AA^{T} = \begin{bmatrix} 1 & 0 & -1 & 2 \\ 2 & 1 & -1 & 5 \\ 3 & 3 & 0 & 9 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 3 \\ -1 & -1 & 0 \\ 2 & 5 & 9 \end{bmatrix} = \begin{bmatrix} 6 & 13 & 21 \\ 13 & 31 & 54 \\ 21 & 54 & 99 \end{bmatrix} \tag{14}$$

Then I used the result of equation 14 to calculate the eigen values of AA^{T}

u will be the vectors of the U matrix which we will use in SVD

$$AA^{T}u - \lambda Iu = 0$$
$$(AA^{T} - \lambda I)u = 0 \tag{15}$$

For a non-trivial u to satisfy equation 15 we need $det(AA^{T} - \lambda I) = 0$.

$$AA^{T} - \lambda I = \begin{bmatrix} 6-\lambda & 13 & 21 \\ 13 & 31-\lambda & 54 \\ 21 & 54 & 99-\lambda \end{bmatrix}$$

$$det(AA^{T} - \lambda I) =$$

$$(6-\lambda)[(31-\lambda)(99-\lambda) - 54^{2}] - 13[13(99-\lambda) - 54 \cdot 21] + 21[13 \cdot 54 - 21(31-\lambda)]$$

$$det(AA^{T} - \lambda I) = -\lambda^{3} + 136\lambda^{2} - 323\lambda = 0$$

The solution to equation 16 is. (16)

$$\lambda_1 = 68 + \sqrt{4301}\,, \qquad \lambda_2 = 68 - \sqrt{4301}\,0,68\,, \qquad \lambda_3 = 0$$

$$\lambda_1 = 133.582\ , \qquad \lambda_2 = 2.418\ , \qquad \lambda_3 = 0 \tag{17}$$

Now that we know the eigen values, I will compute U and V matrices. I will first compute U matrix because it is computed with AA^T and I already computed AA^T.

Because AA^T is a symmetric matrix its eigen vectors will be orthogonal and for SVD formula to work we need U and V matrices to be orthonormal so this is really nice for us. Now I will compute u_1 which is for λ_1.

$$\begin{bmatrix} 6-133.582 & 13 & 21 \\ 13 & 31-133.582 & 54 \\ 21 & 54 & 99-133.582 \end{bmatrix} u_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -127.582 & 13 & 21 \\ 13 & -102.582 & 54 \\ 21 & 54 & -34.582 \end{bmatrix} u_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{18}$$

The normal u_1 vector which satisfies equation 18 is:

$$u_1 = \begin{bmatrix} 0.1898 \\ 0.4761 \\ 0.8587 \end{bmatrix} \tag{19}$$

I used a calculator to solve this step and all the other computations for the u and v vector calculations because all of them are with floating points. But if I wanted to do them by hand, I would have only done what I did in part b of this question.

Now I will compute u_2

$$\begin{bmatrix} 6-2.418 & 13 & 21 \\ 13 & 31-2.418 & 54 \\ 21 & 54 & 99-2.418 \end{bmatrix} u_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{20}$$

$$\begin{bmatrix} 3.582 & 13 & 21 \\ 13 & 28.582 & 54 \\ 21 & 54 & 96.582 \end{bmatrix} u_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The normal u_2 vector which satisfies equation 20 is:

$$u_2 = \begin{bmatrix} 0.7002 \\ 0.5474 \\ -0.4583 \end{bmatrix} \tag{21}$$

Now I will compute u_3

$$\begin{bmatrix} 6 & 13 & 21 \\ 13 & 31 & 54 \\ 21 & 54 & 99 \end{bmatrix} u_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{22}$$

The normal u_3 vector which satisfies equation 22 is:

$$u_3 = \begin{bmatrix} 0.6882 \\ -0.6882 \\ 0.2294 \end{bmatrix} \tag{23}$$

Then we can create the U matrix by writing equation 19, 21, 23 next to each other like a matrix

$$U = [u_1 \quad u_2 \quad u_3] = \begin{bmatrix} 0.1898 & 0.7002 & 0.6882 \\ 0.4761 & 0.5474 & -0.6882 \\ 0.8587 & -0.4583 & 0.2294 \end{bmatrix} \tag{24}$$

Something which I haven't said before but is very important is we need to put the eigen values and the eigen vectors in the same other. I will be careful to do this.

Now I will compute the vectors of V, to do this I need to compute $A^T A$

$$A^T A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 3 \\ -1 & -1 & 0 \\ 2 & 5 & 9 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 & 2 \\ 2 & 1 & -1 & 5 \\ 3 & 3 & 0 & 9 \end{bmatrix} = \begin{bmatrix} 14 & 11 & -3 & 39 \\ 11 & 10 & -1 & 32 \\ -3 & -1 & 2 & -7 \\ 39 & 32 & -7 & 110 \end{bmatrix} \tag{25}$$

I don't need to recalculate the eigenvalues of $A^T A$ because it is same with AA^T and one more 0 for the fourth eigenvalue

$$\lambda_1 = 133.582 \ , \qquad \lambda_2 = 2.418 \ , \qquad \lambda_3 = 0, \qquad \lambda_4 = 0$$

Now I will compute v_1 which is for λ_1.

$$\begin{bmatrix} 14 - 133.582 & 11 & -3 & 39 \\ 11 & 10 - 133.582 & -1 & 32 \\ -3 & -1 & 2 - 133.582 & -7 \\ 39 & 32 & -7 & 110 - 133.582 \end{bmatrix} v_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -119.582 & 11 & -3 & 39 \\ 11 & -123.582 & -1 & 32 \\ -3 & -1 & -131.582 & -7 \\ 39 & 32 & -7 & -23.582 \end{bmatrix} v_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{26}$$

The normal v_1 vector which satisfies equation 26 is:

$$v_1 = \begin{bmatrix} 0.3217 \\ 0.2641 \\ -0.0576 \\ 0.9074 \end{bmatrix} \tag{27}$$

Now I will compute v_2 which is for λ_2.

$$\begin{bmatrix} 14-2.418 & 11 & -3 & 39 \\ 11 & 10-2.418 & -1 & 32 \\ -3 & -1 & 2-2.418 & -7 \\ 39 & 32 & -7 & 110-2.418 \end{bmatrix} v_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -119.582 & 11 & -3 & 39 \\ 11 & -123.582 & -1 & 32 \\ -3 & -1 & -131.582 & -7 \\ 39 & 32 & -7 & -23.582 \end{bmatrix} v_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{28}$$

The normal v_2 vector which satisfies equation 26 is:

$$v_2 = \begin{bmatrix} 0.2702 \\ -0.5322 \\ -0.8023 \\ 0.0082 \end{bmatrix} \tag{29}$$

Now I will compute v_3 and v_4 which is for λ_3 and λ_4. I need to pick 2 solutions which are orthogonal to each other and v_1, v_2.

$$\begin{bmatrix} 14 & 11 & -3 & 39 \\ 11 & 10 & -1 & 32 \\ -3 & -1 & 2 & -7 \\ 39 & 32 & -7 & 110 \end{bmatrix} v_3 = \begin{bmatrix} 14 & 11 & -3 & 39 \\ 11 & 10 & -1 & 32 \\ -3 & -1 & 2 & -7 \\ 39 & 32 & -7 & 110 \end{bmatrix} v_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{30}$$

$$v_3 v_4 = 0 \tag{31}$$

The normal v_3 and v_4 vectors which satisfies equation 28 and equation 29 are:

$$v_3 = \begin{bmatrix} 0.7768 \\ 0.4699 \\ -0.0543 \\ -0.4156 \end{bmatrix}, \quad v_4 = \begin{bmatrix} 0.4691 \\ -0.6529 \\ 0.5916 \\ 0.0613 \end{bmatrix} \tag{32}$$

Then we can create the V matrix by writing equation 27, 29, 32 next to each other like a matrix

$$V = [v_1 \quad v_2 \quad v_3 \quad v_4] = \begin{bmatrix} 0.3217 & 0.2702 & 0.7768 & 0.4691 \\ 0.2641 & -0.5322 & 0.4699 & -0.6529 \\ -0.0576 & -0.8023 & -0.0543 & 0.5916 \\ 0.9074 & 0.0082 & -0.4156 & 0.0613 \end{bmatrix}$$

$$V^T = \begin{bmatrix} 0.3217 & 0.2641 & -0.0576 & 0.9074 \\ 0.2702 & -0.5322 & -0.8023 & 0.0082 \\ 0.7768 & 0.4699 & -0.0543 & -0.4156 \\ 0.4691 & -0.6529 & 0.5916 & 0.0613 \end{bmatrix} \tag{33}$$

Then finally Σ is the matrix with diagonal entries which is equal to square root of the diagonal entries in equation 17. Also, Σ has the same dimension as A

$$\Sigma = \begin{bmatrix} 11.558 & 0 & 0 & 0 \\ 0 & 1.555 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{34}$$

Now we can simply write the SVD using equation 24, equation 33 and equation 34

$$\begin{bmatrix} 0.1898 & 0.7002 & 0.6882 \\ 0.4761 & 0.5474 & -0.6882 \\ 0.8587 & -0.4583 & 0.2294 \end{bmatrix} \begin{bmatrix} 11.558 & 0 & 0 & 0 \\ 0 & 1.555 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0.3217 & 0.2641 & -0.0576 & 0.9074 \\ 0.2702 & -0.5322 & -0.8023 & 0.0082 \\ 0.7768 & 0.4699 & -0.0543 & -0.4156 \\ 0.4691 & -0.6529 & 0.5916 & 0.0613 \end{bmatrix}$$

Then we can compute $A^+ = V\Sigma^+U^T$

$$\Sigma^+ = \begin{bmatrix} 0.0865 & 0 & 0 \\ 0 & 0.6431 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad U^T = \begin{bmatrix} 0.1898 & 0.4761 & 0.8587 \\ 0.7002 & 0.5474 & -0.4583 \\ 0.6882 & -0.6882 & 0.2294 \end{bmatrix}$$

$$\begin{bmatrix} 0.3217 & 0.2702 & 0.7768 & 0.4691 \\ 0.2641 & -0.5322 & 0.4699 & -0.6529 \\ -0.0576 & -0.8023 & -0.0543 & 0.5916 \\ 0.9074 & 0.0082 & -0.4156 & 0.0613 \end{bmatrix} \begin{bmatrix} 0.0865 & 0 & 0 \\ 0 & 0.6431 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0.1898 & 0.4761 & 0.8587 \\ 0.7002 & 0.5474 & -0.4583 \\ 0.6882 & -0.6882 & 0.2294 \end{bmatrix}$$

$$A^+ = \begin{bmatrix} 0.1269 & 0.1084 & -0.0557 \\ -0.2353 & -0.1765 & 0.1765 \\ -0,3622 & -0.2848 & 0.2322 \\ 0.0186 & 0.0402 & 0.0650 \end{bmatrix} \tag{35}$$

Finally, the answer I computed by hand is equation 35. To verify my computation, I used the function in python to compute the pseudo inverse of a matrix. I added my code down below.

```
    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the 
#question

    pseudo_A = np.linalg.pinv(A) # this is the function to compute the pseudo 
#inverse of a matrix

    print("Pseudo invers of the matrix A is: ")
    print(pseudo_A)
```

This code gives the output:

```
[[ 0.12693498  0.10835913 -0.05572755]
 [-0.23529412 -0.17647059  0.17647059]
 [-0.3622291  -0.28482972  0.23219814]
 [ 0.01857585  0.04024768  0.06501548]]
```

This is same as my computation by hand which means my computations were correct.

e) Find the sparsest solution to $Ax = b$.

The sparsest solution means the solution with the least amount of non-zero terms. I will select my solution from the solution space which I calculated in part c. For convenience I will re write equation 11 here once more:

$$x = \begin{bmatrix}1\\2\\0\\0\end{bmatrix} + x_3\begin{bmatrix}1\\-1\\1\\0\end{bmatrix} + x_4\begin{bmatrix}-2\\-1\\0\\1\end{bmatrix} \ \ for\ x_3, x_4 \ \in \mathbb{R} \tag{11}$$

We can see choosing $x_3 = 0\ and\ x_4 = 0$ gives quite a spares solution with only 2 none zero entries.

$$x_{sparsest} = \begin{bmatrix}1\\2\\0\\0\end{bmatrix} + 0\begin{bmatrix}1\\-1\\1\\0\end{bmatrix} + 0\begin{bmatrix}-2\\-1\\0\\1\end{bmatrix} = \begin{bmatrix}1\\2\\0\\0\end{bmatrix}$$

We can also find another sparse solution by taking $x_3 = -1\ and\ x_4 = 0$

$$x_{sparsest} = \begin{bmatrix}1\\2\\0\\0\end{bmatrix} - 1\begin{bmatrix}1\\-1\\1\\0\end{bmatrix} + 0\begin{bmatrix}-2\\-1\\0\\1\end{bmatrix} = \begin{bmatrix}0\\3\\-1\\0\end{bmatrix}$$

Another sparse solution comes from taking $x_3 = 2\ and\ x_4 = 0$

$$x_{sparsest} = \begin{bmatrix}1\\2\\0\\0\end{bmatrix} + 2\begin{bmatrix}1\\-1\\1\\0\end{bmatrix} + 0\begin{bmatrix}-2\\-1\\0\\1\end{bmatrix} = \begin{bmatrix}3\\0\\2\\0\end{bmatrix}$$

Then we can start taking x_3 as zero and find another spare solution from $x_3 = 0\ and\ x_4 = 1/2$

$$x_{sparsest} = \begin{bmatrix}1\\2\\0\\0\end{bmatrix} + 0\begin{bmatrix}1\\-1\\1\\0\end{bmatrix} + \frac{1}{2}\begin{bmatrix}-2\\-1\\0\\1\end{bmatrix} = \begin{bmatrix}0\\1.5\\0\\0.5\end{bmatrix}$$

Another sparse solution comes from taking $x_3 = 0 \; and \; x_4 = 2$

$$x_{sparsest} = \begin{bmatrix}1\\2\\0\\0\end{bmatrix} + 0\begin{bmatrix}1\\-1\\1\\0\end{bmatrix} + 2\begin{bmatrix}-2\\-1\\0\\1\end{bmatrix} = \begin{bmatrix}-3\\0\\0\\2\end{bmatrix}$$

one final spares solution can be found by $x_3 = 1 \; and \; x_4 = 1$

$$x_{sparsest} = \begin{bmatrix}1\\2\\0\\0\end{bmatrix} + 1\begin{bmatrix}1\\-1\\1\\0\end{bmatrix} + 1\begin{bmatrix}-2\\-1\\0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\1\\1\end{bmatrix}$$

I wrote the sparest solution I found once again in equation 36 so they are next to each other.

$$x_{sparsest} = \begin{bmatrix}1\\2\\0\\0\end{bmatrix}, \begin{bmatrix}0\\3\\-1\\0\end{bmatrix}, \begin{bmatrix}3\\0\\2\\0\end{bmatrix}, \begin{bmatrix}0\\1.5\\0\\0.5\end{bmatrix}, \begin{bmatrix}-3\\0\\0\\2\end{bmatrix}, \begin{bmatrix}0\\0\\1\\1\end{bmatrix} \tag{36}$$

These might not be all the sparest solutions; these are just some sparest solutions I found easily by looking at equation11. We could have chosen different vectors from the null space of A. End wrote our solution space differently, then we could have found different sparest solutions. But I can say that none of these different solutions could have less then 2 non zero entries. Because that would require for A matrix to have column equal to a scale of the b vector. We can see none of the columns of A matrix is a scale of the b vector, so the solutions in equation 36 are actually the sparest solutions.

Calculating the sparsest solution is something extremely complex with soft ware even on MATLAB doing it requires download of external packages. The MATLAB function A\b doesn't return the sparsest solution it only returns a solution with non-zero term count equal to rank of A. Because of this just like in part c, I will just prove the sparsest solutions I found in equation 36 satisfy $Ax = b$. The code which does this is down below.

```
    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the
#question

    x_sparsest_1 = np.array([ [1] , [2] , [0] , [0] ]) # I calculated this by
#hand

    x_sparsest_2 = np.array([ [0] , [3] , [-1] , [0] ])
# I calculated this by hand

    x_sparsest_3 = np.array([ [3] , [0] , [2] , [0] ]) # I calculated this by
#hand
```

```
    x_sparsest_4 = np.array([ [0] , [1.5] , [0] , [0.5] ]) # I calculated this by
# hand

    x_sparsest_5 = np.array([ [-3] , [0] , [0] , [2] ])
# I calculated this by hand

    x_sparsest_6 = np.array([ [0] , [0] , [1] , [1] ])
# I calculated this by hand

    b1 = A.dot(x_sparsest_1)
    b2 = A.dot(x_sparsest_2)
    b3 = A.dot(x_sparsest_3)
    b4 = A.dot(x_sparsest_4)
    b5 = A.dot(x_sparsest_5)
    b6 = A.dot(x_sparsest_6)

    print(str(x_sparsest_1.transpose()) + "^T as a x vector gives the output of "
 + str(b1.transpose()) + "^T \n")
    print(str(x_sparsest_2.transpose()) + "^T as a x vector gives the output of "
 + str(b2.transpose()) + "^T \n")
    print(str(x_sparsest_3.transpose()) + "^T as a x vector gives the output of "
 + str(b3.transpose()) + "^T \n")
    print(str(x_sparsest_4.transpose()) + "^T as a x vector gives the output of "
 + str(b4.transpose()) + "^T \n")
    print(str(x_sparsest_5.transpose()) + "^T as a x vector gives the output of "
 + str(b5.transpose()) + "^T \n")
    print(str(x_sparsest_6.transpose()) + "^T as a x vector gives the output of "
 + str(b6.transpose()) + "^T \n")
```

this code gave the output of:

```
[[1 2 0 0]]^T as a x vector gives the output of [[1 4 9]]^T

[[ 0  3 -1  0]]^T as a x vector gives the output of [[1 4 9]]^T

[[3 0 2 0]]^T as a x vector gives the output of [[1 4 9]]^T

[[0.  1.5 0.  0.5]]^T as a x vector gives the output of [[1. 4. 9.]]^T

[[-3  0  0  2]]^T as a x vector gives the output of [[1 4 9]]^T

[[0 0 1 1]]^T as a x vector gives the output of [[1 4 9]]^T
```

Which verifies my solutions by hand still satisfy the given linear system of equations.

f) Find the least-norm solution to $Ax = b$.

Least norm solution means the solution $x_{LeastNorm}$ with minimum $\|x\|$. Something which is really nice is the solution to Ax = b if we us the A^{+}, the pseudo inverse of A, is $x_{LeastNorm}$. Because we already calculated pseudo invers of the A matrix, we can just use equation 37 to calculate $x_{LeastNorm}$.

$$x_{LeastNorm} = A^{+}b \tag{37}$$

$$x_{LeastNorm} = \begin{bmatrix} 0.1269 & 0.1084 & -0.0557 \\ -0.2353 & -0.1765 & 0.1765 \\ -0,3622 & -0.2848 & 0.2322 \\ 0.0186 & 0.0402 & 0.0650 \end{bmatrix}$$

$$x_{LeastNorm} = \begin{bmatrix} 0.0588 \\ 0.6471 \\ 0.5882 \\ 0.7647 \end{bmatrix} \tag{38}$$

To validate my answer in equation 38 I computed the least-norm solution to $Ax = b$ in phyton. The code for this is down below.

```
    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the
#question
    b = np.array([ [1] , [4] , [9] ]) # this is the instensiation of the b vector
# given in the question

    S = lg.lstsq(A, b) # this is the function to compute the least norm solution
#of Ax = b

    print("least-norm solution to Ax = b is")
    print(S[0]) # lstsq returns a list of answers but we only want the first term
# of that lis which is the least norm solution.
```

this gives the output of:

```
[[0.05882353]

 [0.64705882]

 [0.58823529]

 [0.76470588]]
```

This is same as equation 35 this means my computations by hand are correct.

Question 2:

- Broca's area was reported to be activated in 103 out of 869 fMRI tasks involving engagement of language.
- Broca's area was reported to be activated in 199 out of 2353 fMRI tasks not involving engagement of language

a) We are told the probability of activation of Broca's area given language involvement and probability of activation of Broca's area given no language involvement follow a Bernoulli distribution. We are asked to compute the likelihood of observed frequencies of activation in literature, for different Bernoulli parameters for the range of p=[0:.001:1]. Then plot them on separate bar charts.

The 2 function, activation of Broca's area given language involvement and activation of Broca's area given no language involvement respectively have their respective Bernoulli probability parameters $p = x_l$ and $p = x_{nl}$.

Another way to say what is asked from us in this part is, we need to calculate the probability of getting the data in the literature by taking all x_l and x_{nl} values between 0 and 1 with a step size of 0.001 for the Bernoulli parameters. They are given in equation 39 and 40.

$$P(data \mid x_l) \tag{39}$$

$$P(data \mid x_{nl}) \tag{40}$$

Because their probability of activation is given as a Bernoulli distribution. We can represent the probability of random experiment causing activation or not, as a probability distribution. These are given in equation 41 for a task involving language and equation 42 for a task not involving language.

$$P(Activation_i \mid x_l) = \begin{cases} x_l\,, & Activation_i = 1 \\ 1 - x_l\,, & Activation_i = 0 \end{cases} \tag{41}$$

$$P(Activation_i \mid x_{nl}) = \begin{cases} x_{nl}\,, & Activation_i = 1 \\ 1 - x_{nl}\,, & Activation_i = 0 \end{cases} \tag{42}$$

Because we are going to do this experiment n number times and we are expecting get a activation k number of times this hole experiment becomes a binomial distribution like the one in equation 43 the likelihood of observed frequencies of activation in literature, for different Bernoulli parameters can be written as in equation 44 and equation 45

$$P(k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{43}$$

$$n = 869, k = 103\ ;\ P(data_l \mid x_l) = \binom{869}{103} x_l^{103}(1 - x_l)^{766} \quad (44)$$

$$n = 2353, k = 199\ ;\ P(data_{nl} \mid x_{nl}) = \binom{2353}{199} x_{nl}^{199}(1 - x_{nl})^{2154} \quad (45)$$

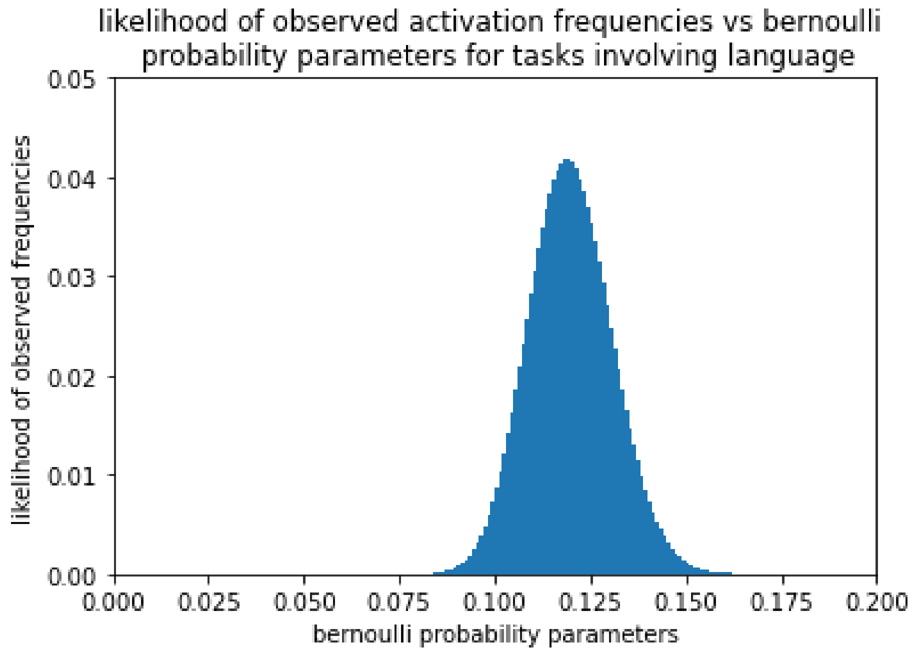
This is something which is extremely hard to compute by hand so I wrote a code to compute the binomial distribution for a set of given $n, k, p\ (x_l\ , x_{nl}\ in\ here)$ parameters. The code is given below.

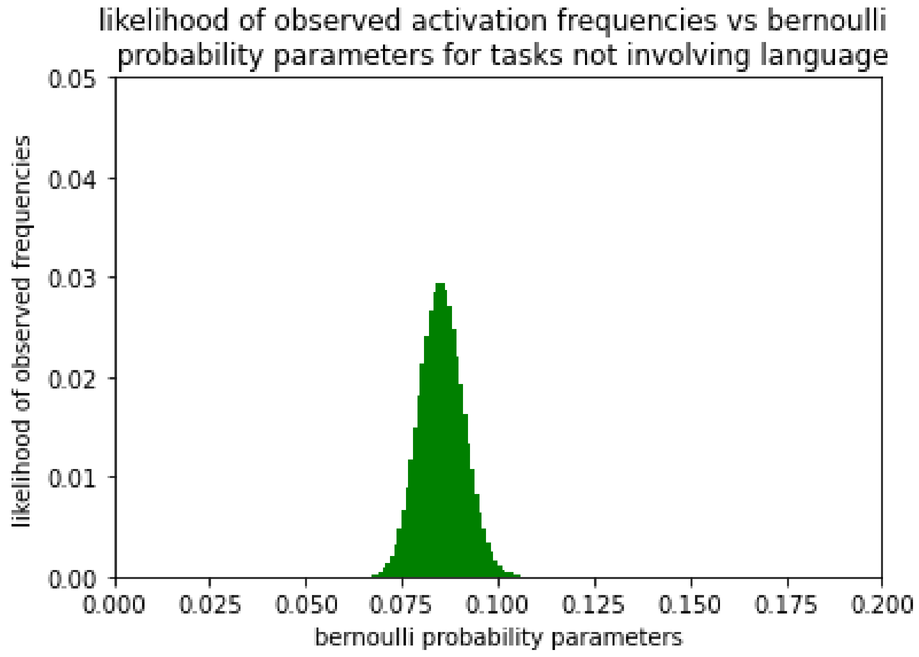
```
def Bernuoli_distribution(number_of_trys , number_of_ones , p):

    return ( math.factorial(number_of_trys)/(math.factorial(number_of_ones) * \
math.factorial(number_of_trys - number_of_ones)) * (p ** number_of_ones) *  \
((1 - p) ** (number_of_trys - number_of_ones)) ) # this is the formula of a
#binomial distrubution
```

Then I created an array of numbers 0 to 1 with a step size of 0.001 to represent different Bernoulli parameters and placed it in the above equation for the 2 cases and draw their results bar charts. The code for this is given down below.

```
    print("All the plots will be displayed at the end")

    Xaxis = np.arange(0, 1.001, 0.001) # python doesn't take 1.001 because it is
#the last elemnent so I had to stop at 1.001 to take 1 as the last element

    Prob_data_language_given_x = Bernuoli_distribution(869, 103, Xaxis) # this
#places each value of the array in the funtion and creats an array out of its
#results

    Prob_data_nonlanguage_given_x = Bernuoli_distribution(2353, 199, Xaxis)
# this places each value of the array in the funtion and creats an array out of
#its results

    plt.figure()
    plt.bar(Xaxis, Prob_data_language_given_x, align = "edge", width = 0.001)
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('likelihood of observed frequencies')
    plt.title('likelihood of observed activation frequencies vs bernoulli \n \
 probability parameters for tasks involving language')
    plt.axis([0, 0.2, 0, 0.05])

    plt.figure()
    plt.bar(Xaxis, Prob_data_nonlanguage_given_x, align = "edge", width = 0.001,
facecolor='g')
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('likelihood of observed frequencies')
    plt.title('likelihood of observed activation frequencies vs bernoulli \n \
 probability parameters for tasks not involving language')
    plt.axis([0, 0.2, 0, 0.05])
```

this gives the output of:





I cut the parts of the graphs for Bernoulli parameters which are higher than 0.2 because the likelihood of the observed frequencies are almost zero there, so drawing them wouldn't add anything to the graph and only lower the resolution of the graph.

b) Find the values of x_l and x_{nl} that maximize their respective discretized likelihood functions.

The x_l and x_{nl} values which maximize their respective discretized likelihood functions are their graphs peaks respectively. Finding peak of the graphs from the previous part is extremely easy. Because all the values the function takes are neatly placed in an array already so I only need to find the maximum int in each array to find the x_l and x_{nl} values. The code that does that is given below.

```
    Xaxis = np.arange(0, 1.001, 0.001) # python doesn't take 1.001 because it is
#the last elemnent so I had to stop at 1.001 to take 1 as the last element

    Prob_data_language_given_x = Bernuoli_distribution(869, 103, Xaxis) # this
#places each value of the array in the funtion and creats an array out of its
#results

    Prob_data_nonlanguage_given_x = Bernuoli_distribution(2353, 199, Xaxis)
#this places each value of the array in the funtion and creats an array out of
#its results


    print("maximum likelihood of x_l = " + str( round( \
max(Prob_data_language_given_x) , 4 )) + " and the Bernoulli paremeter that \
achives this is " + str(np.argmax(Prob_data_language_given_x) / 1000))
    print("")
    print("maximum likelihood of x_nl = " + str( round( \
max(Prob_data_nonlanguage_given_x) , 4 )) + " and the Bernoulli paremeter that \
achives this is " + str(np.argmax(Prob_data_nonlanguage_given_x) / 1000))
```

This code repeats some part of the code in part a because I want the functions for each part of the question to be able to work without first running the previous parts of the same question.

This code finds the x_l and x_{nl} values which maximize their respective discretized likelihood functions as:

$$x_{l,maximum\ likelyhood} = 0.0418 \text{ for } \mathrm{x_l} = 0.119$$

$$x_{nl,maximum\ likelyhood} = 0.0294 \text{ for for } \mathrm{x_{nl}} = 0.085$$

c) Using the likelihood functions computed for discrete x, compute and plot the discrete posterior distributions $P(X|data)$ and the associated cumulative distributions $P(X \leq x|data)$ for both processes, assuming $P(x)$ uniform. Then compute (discrete approximations to) upper and lower 95% confidence bounds on each proportion (xl, nl) using the cumulative distributions.

To compute $P(X|data)$ I am going to use the Bayes' rule. Bayes' rule is given in equation 46.

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{46}$$

If we implement the Bayes' rule to the current problem, we get equation 47

$$P(x \mid data) = \frac{P(data \mid x)P(x)}{P(data)} \tag{47}$$

We are told to take $P(x)$ uniform and there are 1001 different x values so:

$$P(x) = \frac{1}{1001} \tag{48}$$

Then we can compute $P(data)$ using total probity theorem.

$$P(data) = \sum_{x} P(data \mid x)P(x) \tag{49}$$

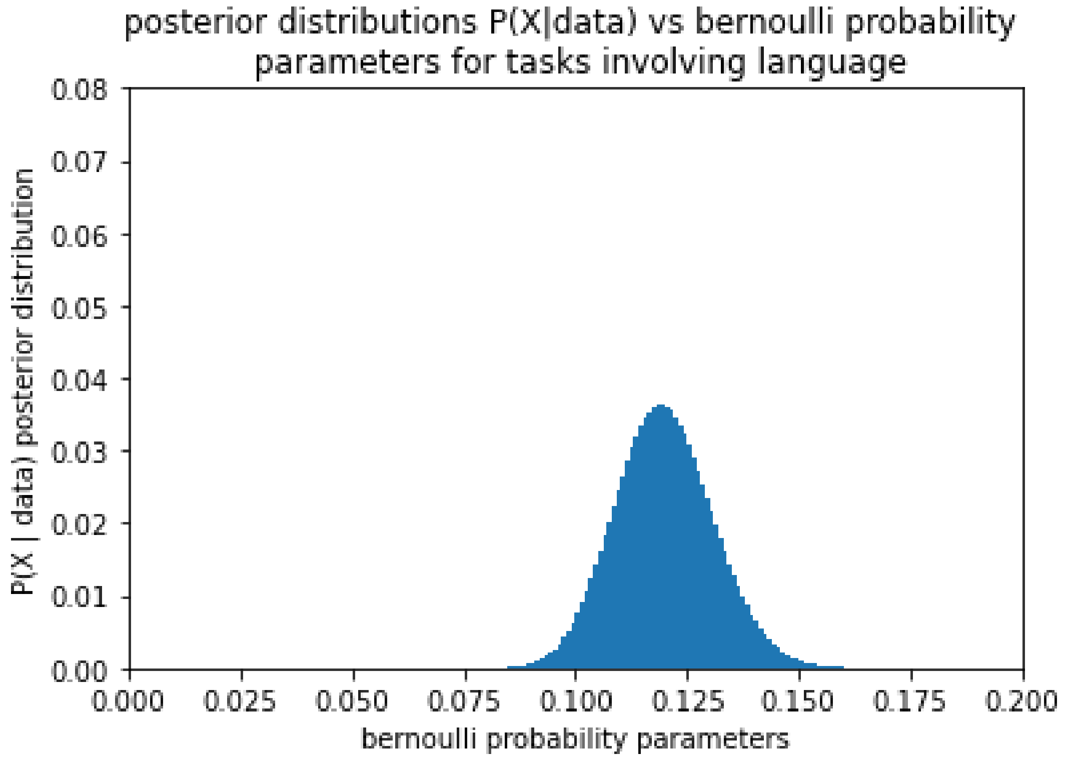
Then by putting equation 48 and equation 49 inside of equation 47 we get:

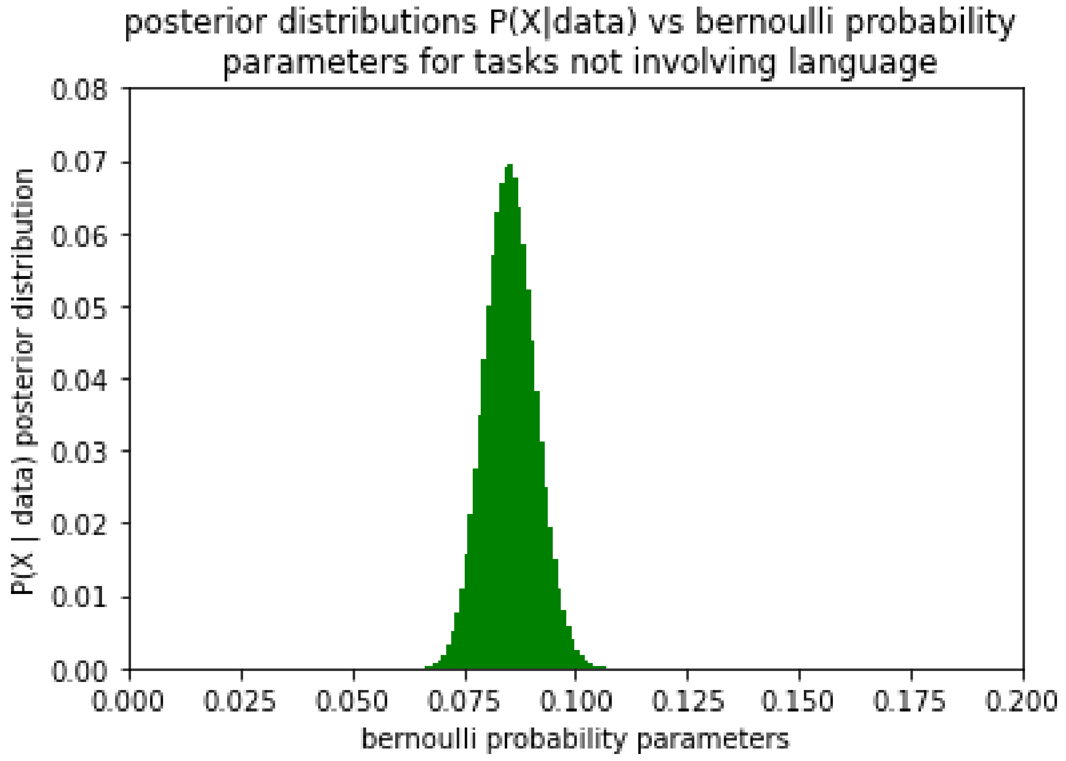
$$P(x \mid data) = \frac{P(data \mid x)\frac{1}{1001}}{\sum_x P(data \mid x)\frac{1}{1001}} \tag{50}$$

The code which computes equation 50 for all the x_l and x_{nl} of the 2 processes is given below.

```
    Xaxis = np.arange(0, 1.001, 0.001) # python doesn't take 1.001 because it is
#the last elemnent so I had to stop at 1.001 to take 1 as the last element

    Prob_data_language_given_x = Bernuoli_distribution(869, 103, Xaxis) # this
#places each value of the array in the funtion and creats an array out of its
#results

    Prob_data_nonlanguage_given_x = Bernuoli_distribution(2353, 199, Xaxis)
# this places each value of the array in the funtion and creats an array out of
#its results

    prob_x = 1 / 1001
    Prob_data_language = sum(Prob_data_language_given_x) * prob_x

    Prob_data_nonlanguage = sum(Prob_data_nonlanguage_given_x) * prob_x


    X_given_data_language = prob_x * Prob_data_language_given_x / \
Prob_data_language # bayes' rule formula

    X_given_data_nonlanguage = prob_x * Prob_data_nonlanguage_given_x / \
Prob_data_nonlanguage # bayes' rule formula

    plt.figure()
    plt.bar(Xaxis, X_given_data_language, align = "edge", width = 0.001)
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('P(X | data) posterior distribution')
    plt.title('posterior distributions P(X|data) vs bernoulli probability \n \
parameters for tasks involving language')
    plt.axis([0, 0.2, 0, 0.08])

    plt.figure()
    plt.bar(Xaxis, X_given_data_nonlanguage, align = "edge", width = 0.001, \
facecolor='g')
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('P(X | data) posterior distribution')
    plt.title('posterior distributions P(X|data) vs bernoulli probability  \
parameters for tasks not involving language')
    plt.axis([0, 0.2, 0, 0.08])
```

this gives the output of:



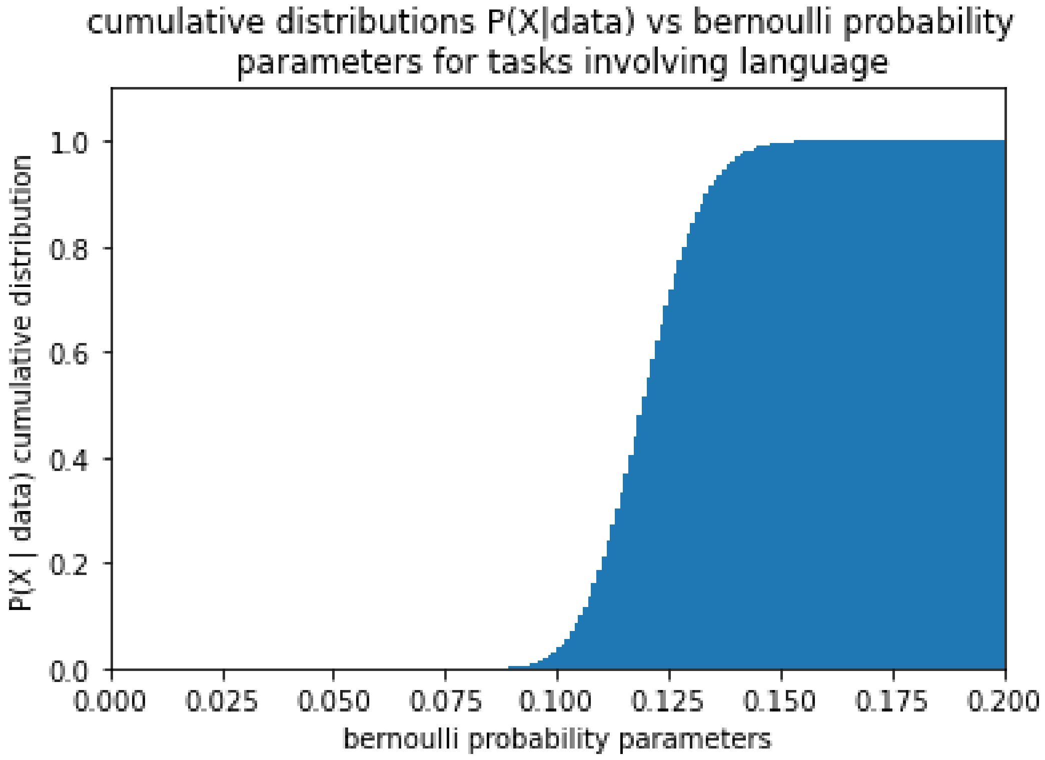


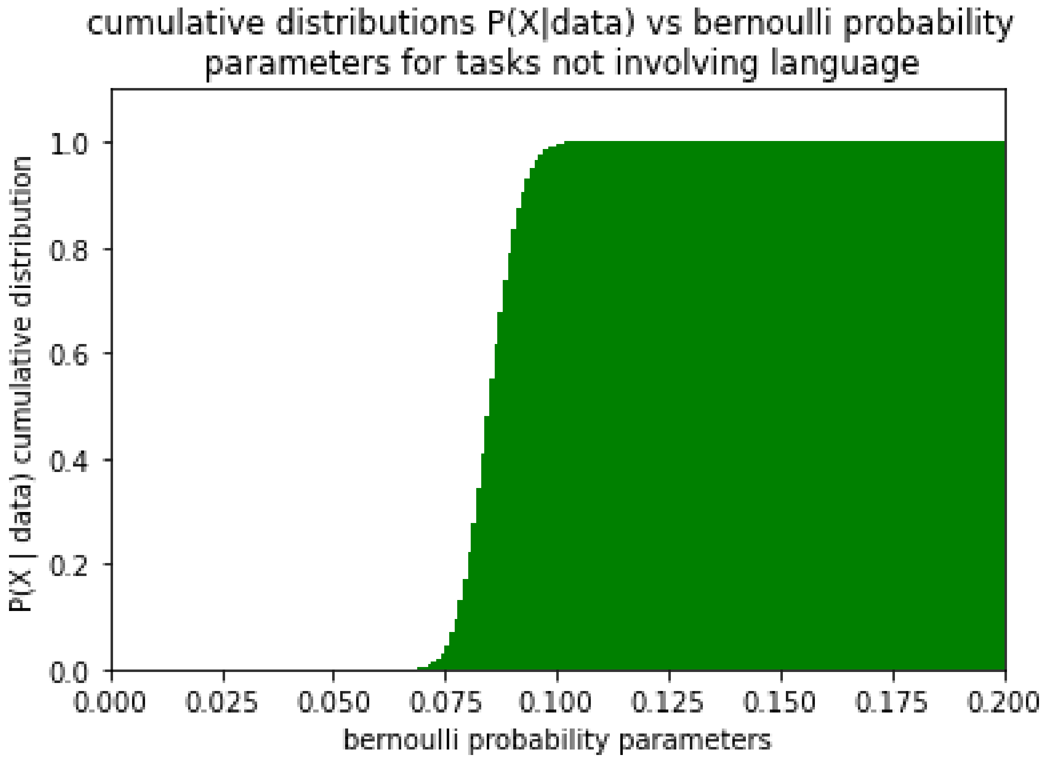
Once again, I cut these graphs at p = 0.2 because after that point there is nothing meaningful, we could look at.

Calculating the associated cumulative distributions $P(X \leq x|data)$ for both processes can be done by cumulatively summing the values of P(X | data) posterior distribution array. The we can draw their bar charts too. the code which does this is given below. This code is the continuum of the code in this part so it has access to that part of codes local variables.

```
    cumulative_X_given_data_language = np.cumsum(X_given_data_language)
    cumulative_X_given_data_nonlanguage = np.cumsum(X_given_data_nonlanguage)

    plt.figure()
    plt.bar(Xaxis, cumulative_X_given_data_language, align = "edge", width = \
0.001)
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('P(X | data) cumulative distribution')
    plt.title('cumulative distributions P(X|data) vs bernoulli probability \n \
parameters for tasks involving language')
    plt.axis([0, 0.2, 0, 1.1])

    plt.figure()
    plt.bar(Xaxis, cumulative_X_given_data_nonlanguage, align = "edge", width = \
 0.001 , facecolor='g')
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('P(X | data) cumulative distribution')
    plt.title('cumulative distributions P(X|data) vs bernoulli probability \n \
parameters for tasks not involving language')
    plt.axis([0, 0.2, 0, 1.1])
```

this gives the output of:





Once again, I cut these graphs at p = 0.2 because after that point there is nothing meaningful, we could look at (it is always almost 1).

To calculate the upper and lower 95% confidence bounds on each proportion (xl, nl) using the cumulative distributions. I only need to loop trough the values of $P(X \leq x|data)$ cumulative distributions and find the first value exceeding 0.025 for the lower bound and the first value exceeding 0.975 for the upper bound. The code which does this is given below and once a again this code is the continuum of the code in this part.

```
    for i in range(0 , 1002): # we will loop trough all the values of the array

        if cumulative_X_given_data_language[i] >= 0.025: # break out of the loop
#once we find the lower confidence bound

            x_language_lower_95_confidence = Xaxis[i]

            break

    for i in range(0 , 1002): # we will loop trough all the values of the array

        if cumulative_X_given_data_language[i] >= 0.975: # break out of the loop
#once we find the upper confidence bound

            x_language_upper_95_confidence = Xaxis[i]

            break


    for i in range(0 , 1002): # we will loop trough all the values of the array

        if cumulative_X_given_data_nonlanguage[i] >= 0.025: # break out of the
#loop once we find the lower confidence bound

            x_non_language_lower_95_confidence = Xaxis[i]

            break


    for i in range(0 , 1002): # we will loop trough all the values of the array

        if cumulative_X_given_data_nonlanguage[i] >= 0.975: # break out of the
#loop once we find the upper confidence bound

            x_non_language_upper_95_confidence = Xaxis[i]
            break

    print("lower 95% confidence bound of x_language: " + \
 str(x_language_lower_95_confidence))
    print("upper 95% confidence bound of x_language: " + \
str(round(x_language_upper_95_confidence , 3)))
    print("")
    print("lower 95% confidence bound of x_nonlanguage: " + \
str(x_non_language_lower_95_confidence))
    print("upper 95% confidence bound of x_nonlanguage: " + \
str(x_non_language_upper_95_confidence))
```

This code gives the output of:

$$x_{l,lower\ 95\%\ confidence} = 0.099\ ,x_{l,upper\ 95\%\ confidence} = 0.142$$

$$x_{nl,lower\ 95\%\ confidence} = 0.074\ ,x_{nl,upper\ 95\%\ confidence} = 0.096$$

d) Calculate $P(X_l, X_{nl} \mid data)$ over x_l and x_{nl}, the Bernoulli probability parameters for the language and non-language contrasts. Given that these two frequencies are independent, then plot it. Then compute the posterior probabilities that $P(X_l > X_{nl}|data)$ and conversely that $P(X_l \leq X_{nl}|data)$.

The 2 frequencies are given as independent, then we can calculate $P(X_l, X_{nl} \mid data)$ by just doing equation 47.

$$P(X_l, X_{nl} \mid data) = P(\, X_{nl} \mid data)P(X_l \mid data)^{\,T} \tag{47}$$

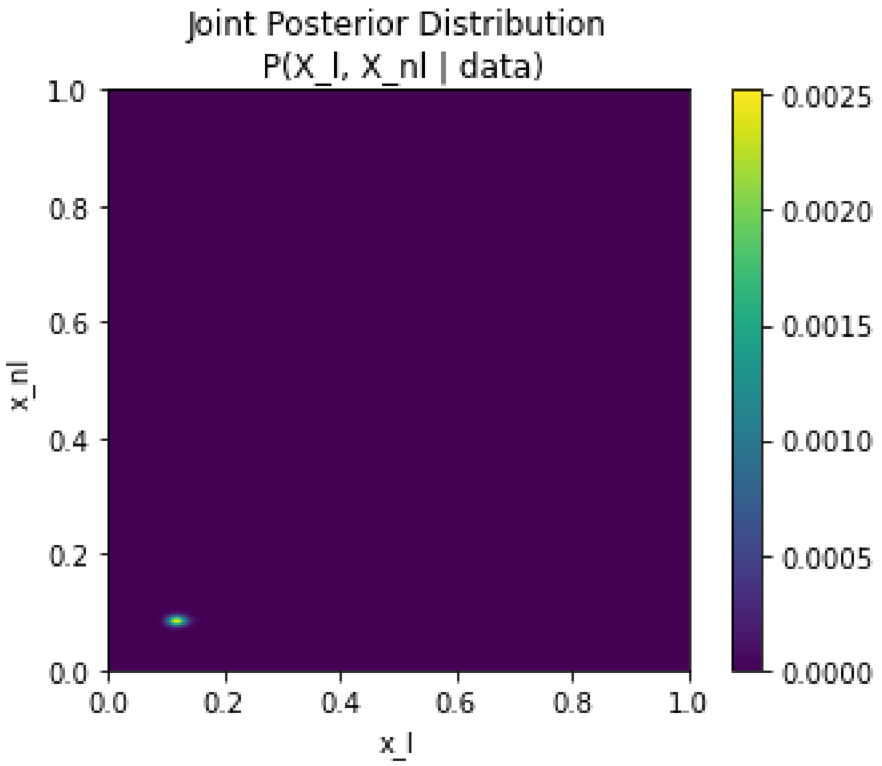
in equation 47 $P(\, X_{nl} \mid data)\ ,P(X_l \mid data)$ represent a vector of probabilities and $P(X_l, X_{nl} \mid data)$ represent a matrix of probabilities.

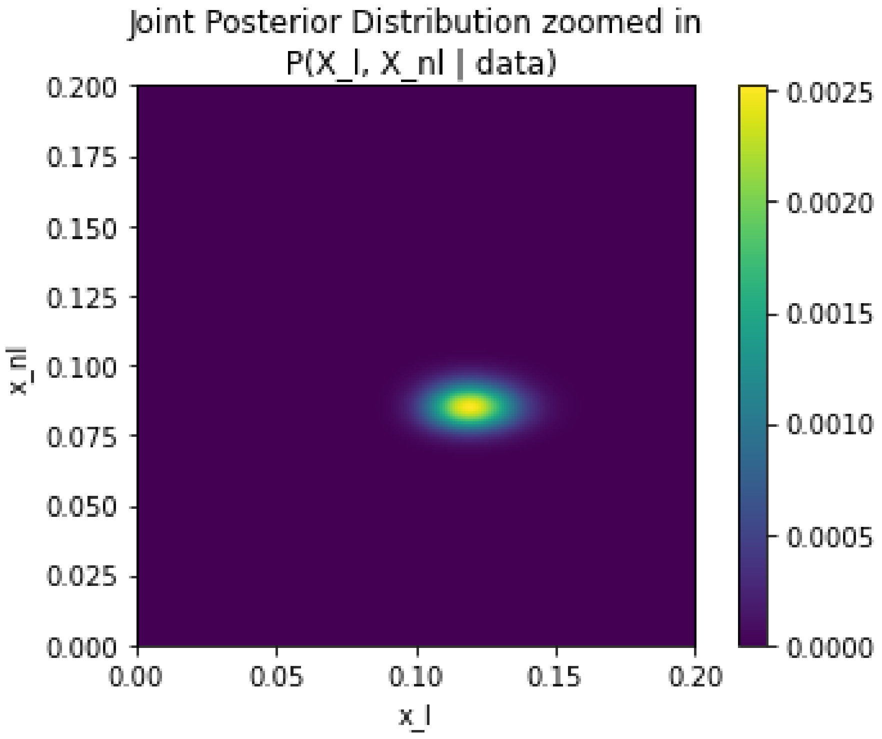
To plot $P(X_l, X_{nl} \mid data)$ as a heatmap the homework suggest to use "imagesc" but phyton doesn't come with "imagesc" and has other function which does the same thing. I used one of those function so that during testing the tester doesn't need to download anything additional. The code for this is given below.

```
    Xaxis = np.arange(0, 1.001, 0.001) # python doesn't take 1.001 because it is
#the last elemnent so I had to stop at 1.001 to take 1 as the last element

    Prob_data_language_given_x = Bernuoli_distribution(869, 103, Xaxis) # this
#places each value of the array in the funtion and creats an array out of its
#results
```

```
    Prob_data_nonlanguage_given_x = Bernuoli_distribution(2353, 199, Xaxis)
# this places each value of the array in the funtion and creats an array out of
#its results

    prob_x = 1 / 1001

    Prob_data_language = sum(Prob_data_language_given_x) * prob_x

    Prob_data_nonlanguage = sum(Prob_data_nonlanguage_given_x) * prob_x


    X_given_data_language = prob_x * Prob_data_language_given_x / \
Prob_data_language # bayes' rule formula

    X_given_data_nonlanguage = prob_x * Prob_data_nonlanguage_given_x / \
Prob_data_nonlanguage # bayes' rule formula

    X_l_X_nl = np.outer(X_given_data_nonlanguage, X_given_data_language)

    plt.figure()
    plt.imshow(X_l_X_nl , origin='lower' , extent=[0,1,0,1]) #this is the
#function to plot a heat map
    plt.colorbar()
    plt.xlabel("x_l")
    plt.ylabel("x_nl")
    plt.title("Joint Posterior Distribution\n P(X_l, X_nl | data)")

    plt.figure()
    plt.imshow(X_l_X_nl , origin='lower' , extent=[0,1,0,1]) #this is the
#function to plot a heat map
    plt.axis([0, 0.2, 0, 0.2])
    plt.colorbar()
    plt.xlabel("x_l")
    plt.ylabel("x_nl")
    plt.title("Joint Posterior Distribution zoomed in\n P(X_l, X_nl | data)")
```

This code gives the output of:





This time I drew the hole plot too, because I wanted to show the part which I wasn't drawing since the beginning of this question was not interesting.

Then I calculated $P(X_l > X_{nl}|data)$ summing the upper triangle entries (excluding the diagonal line) of the matrix I drew and $P(X_l \leq X_{nl}|data)$ is the sum of the rest of the elements of that matrix. Something to note to not get confused; the plots I drew were flipped so that the (0, 0) is on the bottom left corner. The code that calculates $P(X_l > X_{nl}|data)$ and $P(X_l \leq X_{nl}|data)$ is given below.

```
    xl_bigger_xnl = 0
    xnl_bigger_xl = 0
    for i in range(0 , 1001): # this loops trough all the matrix and put its
#upper and lower triangles entries to the respective sums.
        for j in range(0 , 1001):
            if j > i :
                xl_bigger_xnl += X_l_X_nl[i , j]
            else:
                xnl_bigger_xl += X_l_X_nl[i , j]


    print("P(Xl > Xnl|data) = " + str(round(xl_bigger_xnl , 4)))
    print("P(Xl ≤ Xnl|data) = " + str(round(xnl_bigger_xl , 4)))
```

this code gives the output of:

$$P(X_l > X_{nl}|data) = 0.9979 \; and \; P(X_l \leq X_{nl}|data) = 0.0021$$

e) Using the estimates from part b as the relevant conditional probabilities, and assuming the prior that a contrast engages language, P(language) = 0.5, compute the probability P(language|activation). Then comment on reverse inference technic.

I will use Bayes' rule to compute P(language|activation), just like I used it in part c of this question. If we implement Bayes' rule shown in equation 46 to this situation, we get 48.

$$P(language \mid activation) = \frac{P(activation \mid language)P(language)}{P(activation)} \tag{48}$$

In the question $P(language) = 5$ is given and $P(activation)$ can be found from total probability theorem as:

$$P(activation) =$$

$$P(activation \mid language)P(language) + P(activation \mid nonlanguage)P(nonlanguage)$$

Then the question tells us to take the $P(activation \mid language)$ and $P(activation \mid nonlanguage)$ from the answer of part b, the $P(activation)$ is:

$$P(activation \mid language) = 0.119$$

$$P(activation \mid nonlanguage) = 0.085$$

$$P(activation) = 0.119 \cdot 0.5 + 0.085 \cdot 0.5$$

the we can find the terms we found back into equation 48 and find:

$$P(language \mid activation) = \frac{0.119 \cdot 0.5}{0.119 \cdot 0.5 + 0.085 \cdot 0.5}$$

The code which does this computation is given below.

```
    Xaxis = np.arange(0, 1.001, 0.001) # python doesn't take 1.001 because it is 
#the last elemnent so I had to stop at 1.001 to take 1 as the last element

    Prob_data_language_given_x = Bernuoli_distribution(869, 103, Xaxis) # this 
#places each value of the array in the funtion and creats an array out of its 
#results

    Prob_data_nonlanguage_given_x = Bernuoli_distribution(2353, 199, Xaxis) 
# this places each value of the array in the funtion and creats an array out of 
#its results

    P_activation_given_language = max(Prob_data_language_given_x)
    P_activation_given_nonlanguage = max(Prob_data_nonlanguage_given_x)

    P_language = 0.5 # this is given the question.

    P_activation = P_language * P_activation_given_language + ( 1 - P_language )\
* P_activation_given_nonlanguage

    P_language_given_activation = P_activation_given_language * P_language / \
P_activation # Bayes' rule formula

    print ("P(language|activation) = " + str(round(P_language_given_activation \
, 4)))
```

this code gives the output of:

$$P(language|activation) = 0.5865$$

Saying “observing activation in this area implies engagement of language processes” is not interlay correct because $P(language|activation) = 0.5865$ so only slightly more than half of the activations are caused by a process involving language. An other way to say this is when ewer there is an activation in the area there is a 41.35% chance it was caused by a task not involving language. So, we almost can’t be confident at all in implicating language if we observe activity in Broca’s area. But still if we were to guess if an activity in the area was caused by a language involving task or non-language involving task, we should still guess a language involving task as its probability is bigger than 50%. But we shouldn’t forget to be unconfident in our guess.

CODE:

As a formality I am adding all of my code as it is at the end of the report:

```
import sys # the primer already has it
import numpy as np # this brings simple matrix operations to python
import matplotlib.pyplot as plt # this brings nice loking plots to python

import random # for random number generation in some verifications

from scipy import linalg as lg # this brings complicated matrix operations to pyt
hon and elivates it to the level of matlab

import math as math # this brings mathematical operations to python


question = sys.argv[1]

def Batu_Arda_Düzgün_21802633_hw1(question):
    if question == '1' :
        print("Answer of question 1.) \n")
        question1_part_a()
        question1_part_b()
        question1_part_c()
        question1_part_d()
        question1_part_e()
        question1_part_f()
    elif question == '2' :
        print("Answer of question 2.) \n")
        question2_part_a()
        question2_part_b()
        question2_part_c()
        question2_part_d()
        question2_part_e()


def question1_part_a():

    print("\nAnswer of question 1 part a: \n")

    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the q
uestion

    NS = lg.null_space(A) # this finds the null space of the given matrix
    CNS = NS.copy() # I am doing these steps to put the null space in to a format
 close to what I found by hand.

    C0 = CNS[-1,0]
    C1 = CNS[-1,1]

    CNS[: ,0] -= CNS[: , 1] * C0 / C1

    C0 = CNS[-2,0]

    CNS[: ,0] /= C0

    C0 = CNS[-2,0]
    C1 = CNS[-2,1]

    CNS[: ,1] -= CNS[: , 0] * C1 / C0

    C1 = CNS[-1,1]

    CNS[: ,1] /= C1

    print("The null space of the matrix is ")
    print(CNS)



def question1_part_b():

    print("\nAnswer of question 1 part b: \n")

    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the q
uestion

    x_p = np.array([ [1] , [2] , [0] , [0] ]) # this is the instensiation of the
x_p I calculated by hand

    b = A.dot(x_p)

    print("For the particular solution I found " + str(x_p.transpose()) + "^T the
 result is " + str(b.transpose()) + "^T which means it satisfys the equation")
```

```
def question1_part_c():

    print("\nAnswer of question 1 part c: \n")

    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the q
uestion

    x_p = np.array([ [1] , [2] , [0] , [0] ]) # this is the instensiation of the
x_p I calculated by hand

    x3 = random.random() # this generates a random number between 0 and 1. we can
 multipley 1000 to get a number generation between 0 and 1000 but I didn't do it
because it shoudln't change the realizim of the test

    x4 = random.random()

    x3x4 = np.array([ [x3] , [x4] ])

    null_space = lg.null_space(A) # this finds the null space of the given matrix

    x = x_p + null_space.dot(x3x4) # creating the random x from our solution set.

    b = A.dot(x)

    print( "for x_3 = " + str(x3) + " and x_4 = " + str(x4) + " b is equal to: ")
    print( b )


def question1_part_d():

    print("\nAnswer of question 1 part d: \n")

    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the q
uestion

    pseudo_A = np.linalg.pinv(A) # this is the function to compute the pseudo inv
erse of a matrix

    print("Pseudo invers of the matrix A is: ")
    print(pseudo_A)
```

```
def question1_part_e():

    print("\nAnswer of question 1 part e: \n")

    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the q
uestion

    x_sparsest_1 = np.array([ [1] , [2] , [0] , [0] ]) # I calculated this by han
d

    x_sparsest_2 = np.array([ [0] , [3] , [-
1] , [0] ]) # I calculated this by hand

    x_sparsest_3 = np.array([ [3] , [0] , [2] , [0] ]) # I calculated this by han
d

    x_sparsest_4 = np.array([ [0] , [1.5] , [0] , [0.5] ]) # I calculated this by
 hand

    x_sparsest_5 = np.array([ [-
3] , [0] , [0] , [2] ]) # I calculated this by hand

    x_sparsest_6 = np.array([ [0] , [0] , [1] , [1] ]) # I calculated this by han
d

    b1 = A.dot(x_sparsest_1)
    b2 = A.dot(x_sparsest_2)
    b3 = A.dot(x_sparsest_3)
    b4 = A.dot(x_sparsest_4)
    b5 = A.dot(x_sparsest_5)
    b6 = A.dot(x_sparsest_6)

    print(str(x_sparsest_1.transpose()) + "^T as a x vector gives the output of "
 + str(b1.transpose()) + "^T \n")
    print(str(x_sparsest_2.transpose()) + "^T as a x vector gives the output of "
 + str(b2.transpose()) + "^T \n")
    print(str(x_sparsest_3.transpose()) + "^T as a x vector gives the output of "
 + str(b3.transpose()) + "^T \n")
    print(str(x_sparsest_4.transpose()) + "^T as a x vector gives the output of "
 + str(b4.transpose()) + "^T \n")
    print(str(x_sparsest_5.transpose()) + "^T as a x vector gives the output of "
 + str(b5.transpose()) + "^T \n")
```

```
    print(str(x_sparsest_6.transpose()) + "^T as a x vector gives the output of "
 + str(b6.transpose()) + "^T \n")

def question1_part_f():

    print("\nAnswer of question 1 part f: \n")

    A = np.array([ [1, 0, -1, 2] , [2, 1, -
1, 5] , [3, 3, 0, 9]]) # this is the instensiation of the A matrix given in the q
uestion
    b = np.array([ [1] , [4] , [9] ]) # this is the instensiation of the b vector
 given in the question

    S = lg.lstsq(A, b) # this is the function to compute the least norm solution 
of Ax = b

    print("least-norm solution to Ax = b is")
    print(S[0]) # lstsq returns a list of answers but we only want the first term
 of that lis which is the least norm solution.

def question2_part_a():

    print("\nAnswer of question 2 part a: \n")

    print("All the plots will be displayed at the end")

    Xaxis = np.arange(0, 1.001, 0.001) # python doesn't take 1.001 because it is 
the last elemnent so I had to stop at 1.001 to take 1 as the last element

    Prob_data_language_given_x = Bernuoli_distribution(869, 103, Xaxis) # this pl
aces each value of the array in the funtion and creats an array out of its result
s

    Prob_data_nonlanguage_given_x = Bernuoli_distribution(2353, 199, Xaxis) # thi
s places each value of the array in the funtion and creats an array out of its re
sults

    plt.figure()
    plt.bar(Xaxis, Prob_data_language_given_x, align = "edge", width = 0.001)
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('likelihood of observed frequencies')
    plt.title('likelihood of observed activation frequencies vs bernoulli \n prob
ability parameters for tasks involving language')
    plt.axis([0, 0.2, 0, 0.05])
    plt.figure()
    plt.bar(Xaxis, Prob_data_nonlanguage_given_x, align = "edge", width = 0.001, 
facecolor='g')
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('likelihood of observed frequencies')
    plt.title('likelihood of observed activation frequencies vs bernoulli \n prob
ability parameters for tasks not involving language')
    plt.axis([0, 0.2, 0, 0.05])


def Bernuoli_distribution(number_of_trys , number_of_ones , p):

    return ( math.factorial(number_of_trys)/(math.factorial(number_of_ones) * mat
h.factorial(number_of_trys - number_of_ones)) * (p ** number_of_ones) * ((1 - p) 
** (number_of_trys - number_of_ones)) ) # this is the formula of a binomial distr
ubution


def question2_part_b():

    print("\nAnswer of question 2 part b: \n")

    Xaxis = np.arange(0, 1.001, 0.001) # python doesn't take 1.001 because it is 
the last elemnent so I had to stop at 1.001 to take 1 as the last element

    Prob_data_language_given_x = Bernuoli_distribution(869, 103, Xaxis) # this pl
aces each value of the array in the funtion and creats an array out of its result
s

    Prob_data_nonlanguage_given_x = Bernuoli_distribution(2353, 199, Xaxis) # thi
s places each value of the array in the funtion and creats an array out of its re
sults


    print("maximum likelihood of x_l = " + str( round( max(Prob_data_language_giv
en_x) , 4 )) + " and the Bernoulli paremeter that achives this is " + str(np.argm
ax(Prob_data_language_given_x) / 1000))
    print("")
    print("maximum likelihood of x_nl = " + str( round( max(Prob_data_nonlanguage
_given_x) , 4 )) + " and the Bernoulli paremeter that achives this is " + str(np.
argmax(Prob_data_nonlanguage_given_x) / 1000))
```

```
def question2_part_c():

    print("\nAnswer of question 2 part c: \n")

    Xaxis = np.arange(0, 1.001, 0.001) # python doesn't take 1.001 because it is
the last elemnent so I had to stop at 1.001 to take 1 as the last element

    Prob_data_language_given_x = Bernuoli_distribution(869, 103, Xaxis) # this pl
aces each value of the array in the funtion and creats an array out of its result
s

    Prob_data_nonlanguage_given_x = Bernuoli_distribution(2353, 199, Xaxis) # thi
s places each value of the array in the funtion and creats an array out of its re
sults

    prob_x = 1 / 1001

    Prob_data_language = sum(Prob_data_language_given_x) * prob_x

    Prob_data_nonlanguage = sum(Prob_data_nonlanguage_given_x) * prob_x


    X_given_data_language = prob_x * Prob_data_language_given_x / Prob_data_langu
age # bayes' rule formula

    X_given_data_nonlanguage = prob_x * Prob_data_nonlanguage_given_x / Prob_data
_nonlanguage # bayes' rule formula

    plt.figure()
    plt.bar(Xaxis, X_given_data_language, align = "edge", width = 0.001)
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('P(X | data) posterior distribution')
    plt.title('posterior distributions P(X|data) vs bernoulli probability \n para
meters for tasks involving language')
    plt.axis([0, 0.2, 0, 0.08])

    plt.figure()
    plt.bar(Xaxis, X_given_data_nonlanguage, align = "edge", width = 0.001, facec
olor='g')
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('P(X | data) posterior distribution')
    plt.title('posterior distributions P(X|data) vs bernoulli probability \n para
meters for tasks not involving language')
    plt.axis([0, 0.2, 0, 0.08])
    cumulative_X_given_data_language = np.cumsum(X_given_data_language)
    cumulative_X_given_data_nonlanguage = np.cumsum(X_given_data_nonlanguage)

    plt.figure()
    plt.bar(Xaxis, cumulative_X_given_data_language, align = "edge", width = 0.00
1)
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('P(X | data) cumulative distribution')
    plt.title('cumulative distributions P(X|data) vs bernoulli probability \n par
ameters for tasks involving language')
    plt.axis([0, 0.2, 0, 1.1])

    plt.figure()
    plt.bar(Xaxis, cumulative_X_given_data_nonlanguage, align = "edge", width = 0
.001 , facecolor='g')
    plt.xlabel('bernoulli probability parameters')
    plt.ylabel('P(X | data) cumulative distribution')
    plt.title('cumulative distributions P(X|data) vs bernoulli probability \n par
ameters for tasks not involving language')
    plt.axis([0, 0.2, 0, 1.1])

    for i in range(0 , 1002): # we will loop trough all the values of the array

        if cumulative_X_given_data_language[i] >= 0.025: # break out of the loop
once we find the lower confidence bound

            x_language_lower_95_confidence = Xaxis[i]

            break

    for i in range(0 , 1002): # we will loop trough all the values of the array

        if cumulative_X_given_data_language[i] >= 0.975: # break out of the loop
once we find the upper confidence bound

            x_language_upper_95_confidence = Xaxis[i]

            break


    for i in range(0 , 1002): # we will loop trough all the values of the array

        if cumulative_X_given_data_nonlanguage[i] >= 0.025: # break out of the lo
op once we find the lower confidence bound
```

```
            x_non_language_lower_95_confidence = Xaxis[i]

            break


    for i in range(0 , 1002): # we will loop trough all the values of the array

        if cumulative_X_given_data_nonlanguage[i] >= 0.975: # break out of the lo
op once we find the upper confidence bound

            x_non_language_upper_95_confidence = Xaxis[i]

            break


    print("lower 95% confidence bound of x_language: " + str(x_language_lower_95_
confidence))
    print("upper 95% confidence bound of x_language: " + str(round(x_language_upp
er_95_confidence , 3)))
    print("")
    print("lower 95% confidence bound of x_nonlanguage: " + str(x_non_language_lo
wer_95_confidence))
    print("upper 95% confidence bound of x_nonlanguage: " + str(x_non_language_up
per_95_confidence))


def question2_part_d():

    print("\nAnswer of question 2 part d: \n")

    Xaxis = np.arange(0, 1.001, 0.001) # python doesn't take 1.001 because it is
the last elemnent so I had to stop at 1.001 to take 1 as the last element

    Prob_data_language_given_x = Bernuoli_distribution(869, 103, Xaxis) # this pl
aces each value of the array in the funtion and creats an array out of its result
s

    Prob_data_nonlanguage_given_x = Bernuoli_distribution(2353, 199, Xaxis) # thi
s places each value of the array in the funtion and creats an array out of its re
sults

    prob_x = 1 / 1001

    Prob_data_language = sum(Prob_data_language_given_x) * prob_x
    Prob_data_nonlanguage = sum(Prob_data_nonlanguage_given_x) * prob_x


    X_given_data_language = prob_x * Prob_data_language_given_x / Prob_data_langu
age # bayes' rule formula

    X_given_data_nonlanguage = prob_x * Prob_data_nonlanguage_given_x / Prob_data
_nonlanguage # bayes' rule formula

    X_l_X_nl = np.outer(X_given_data_nonlanguage, X_given_data_language)

    plt.figure()
    plt.imshow(X_l_X_nl , origin='lower' , extent=[0,1,0,1]) #this is the functio
n to plot a heat map
    plt.colorbar()
    plt.xlabel("x_l")
    plt.ylabel("x_nl")
    plt.title("Joint Posterior Distribution\n P(X_l, X_nl | data)")

    plt.figure()
    plt.imshow(X_l_X_nl , origin='lower' , extent=[0,1,0,1]) #this is the functio
n to plot a heat map
    plt.axis([0, 0.2, 0, 0.2])
    plt.colorbar()
    plt.xlabel("x_l")
    plt.ylabel("x_nl")
    plt.title("Joint Posterior Distribution zoomed in\n P(X_l, X_nl | data)")

    xl_bigger_xnl = 0
    xnl_bigger_xl = 0


    for i in range(0 , 1001): # this loops trough all the matrix and put its uppe
r and lower triangles entries to the respective sums.
        for j in range(0 , 1001):
            if j > i :
                xl_bigger_xnl += X_l_X_nl[i , j]
            else:
                xnl_bigger_xl += X_l_X_nl[i , j]


    print("P(Xl > Xnl|data) = " + str(round(xl_bigger_xnl , 4)))
    print("P(Xl ≤ Xnl|data) = " + str(round(xnl_bigger_xl , 4)))
```

```
def question2_part_e():

    print("\nAnswer of question 2 part e: \n")

    Xaxis = np.arange(0, 1.001, 0.001) # python doesn't take 1.001 because it is
the last elemnent so I had to stop at 1.001 to take 1 as the last element

    Prob_data_language_given_x = Bernuoli_distribution(869, 103, Xaxis) # this pl
aces each value of the array in the funtion and creats an array out of its result
s

    Prob_data_nonlanguage_given_x = Bernuoli_distribution(2353, 199, Xaxis) # thi
s places each value of the array in the funtion and creats an array out of its re
sults

    P_activation_given_language = max(Prob_data_language_given_x)
    P_activation_given_nonlanguage = max(Prob_data_nonlanguage_given_x)

    P_language = 0.5 # this is given the question.

    P_activation = P_language * P_activation_given_language + ( 1 - P_language )
* P_activation_given_nonlanguage

    P_language_given_activation = P_activation_given_language * P_language / P_ac
tivation # Bayes' rule formula

    print ("P(language|activation) = " + str(round(P_language_given_activation ,
4)))

    plt.show() # this privents the plot from closing themselfs at the end

Batu_Arda_Düzgün_21802633_hw1(question)
```