EEE482 - Computational Neuroscience

Homework Assignment - 1

Berkan Ozdamar

21602353

Question 1

$$A = \begin{pmatrix} 1 & 0 & -1 & 2 \\ 2 & 1 & -1 & 5 \\ 3 & 3 & 0 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ 4 \\ 9 \end{pmatrix}$$

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

Part a

To find the general solution x_n to Ax = 0, reduced row echelon form of A must be found. To find rref(A), row operations must be performed.

$$\begin{pmatrix} 1 & 0 & -1 & 2 \\ 2 & 1 & -1 & 5 \\ 3 & 3 & 0 & 9 \end{pmatrix} \xrightarrow{-2r_1+r_2 \to r_2} \begin{pmatrix} 1 & 0 & -1 & 2 \\ 0 & 1 & 1 & 1 \\ 3 & 3 & 0 & 9 \end{pmatrix} \xrightarrow{-3r_1+r_3 \to r_3} \begin{pmatrix} 1 & 0 & -1 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 3 & 3 & 3 \end{pmatrix} \xrightarrow{-3r_2+r_3 \to r_3} \begin{pmatrix} 1 & 0 & -1 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Then we can write the equation as:

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

$x_1 - x_3 + 2x_4 = 0$

$x_2 + x_3 + x_4 = 0$

Since x_3 and x_4 are free variables, x_1 and x_2 should be written in terms of x_3 and x_4.

$x_1 = x_3 - 2x_4$

$x_2 = -x_3 - x_4$

Then we can write x_n as:

$$x_n = \begin{pmatrix} x_3 - 2x_4 \\ -x_3 - x_4 \\ x_3 \\ x_4 \end{pmatrix} \quad , \quad x_n = x_3 \begin{pmatrix} 1 \\ -1 \\ 1 \\ 0 \end{pmatrix} + x_4 \begin{pmatrix} -2 \\ -1 \\ 0 \\ 1 \end{pmatrix}$$

Python code is used to verify the results found.

[1]:
```
import numpy as np

A = np.array([[1, 0, -1, 2], [2, 1, -1, 5], [3, 3, 0, 9]])
b = np.array([1, 4, 9])
```

[2]:
```
# Part a

#Since x_3 and x_4 are free variables, we will assign them random numbers with␣
 ↪numpy.random
x_3 = np.random.random()
x_4 = np.random.random()

x_n = np.array([x_3 - 2*x_4, -x_3 - x_4, x_3, x_4])
print('Solution for Ax = 0')
print(x_n)
```

```
print('\n')
result = np.around(A.dot(x_n),6)
print('Confirming that Ax = 0')
print(result)
```

```
Solution for Ax = 0
[-0.24602768 -0.83691914  0.47593687  0.36098227]


Confirming that Ax = 0
[0. 0. 0.]
```

Part b

To find a particular solution to Ax = b, row operations must be performed to $A|b$. Reduced row echelon form of $A|b$ is as follows:

$$\left(\begin{array}{cccc|c} 1 & 0 & -1 & 2 & 1 \\ 2 & 1 & -1 & 5 & 4 \\ 3 & 3 & 0 & 9 & 9 \end{array}\right) \xrightarrow{-2r_1+r_2 \to r_2} \left(\begin{array}{cccc|c} 1 & 0 & -1 & 2 & 1 \\ 0 & 1 & 1 & 1 & 2 \\ 3 & 3 & 0 & 9 & 9 \end{array}\right) \xrightarrow{-3r_1+r_3 \to r_3} \left(\begin{array}{cccc|c} 1 & 0 & -1 & 2 & 1 \\ 0 & 1 & 1 & 1 & 2 \\ 0 & 3 & 3 & 3 & 6 \end{array}\right) \xrightarrow{-3r_2+r_3 \to r_3} \left(\begin{array}{cccc|c} 1 & 0 & -1 & 2 & 1 \\ 0 & 1 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Then we can write the equation as:

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

From the equation, we get;

$x_1 - x_3 + 2x_4 = 1$

$x_2 + x_3 + x_4 = 2$

Since x_3 and x_4 are free variables, x_1 and x_2 should be written in terms of x_3 and x_4.

$x_1 = 1 + x_3 - 2x_4$

$x_2 = 2 - x_3 - x_4$

For a particular solution, free variables x_3 and x_4 can be setted 0. From there, $x_1 = 1$ and $x_2 = 2$. So, $x_p = \begin{pmatrix} 1 \\ 2 \\ 0 \\ 0 \end{pmatrix}$ is a particular solution.

Python code confirmation of the part b is as:

[3]:
```
# Part b

#Since x_3 and x_4 are free variables, we will write solution set in terms of them.␣
 ↪And since we are looking for
# a particular solution, assigning x_3 and x_4 as 0 will make x_p = [1 2 0 0]^T.␣
 ↪Then check whether A.x_p = b where
# b is [1 4 9]^T
x_3 = 0
x_4 = 0

print('Solution for Ax = b')
x_p = np.array([ 1 + x_3 - 2*x_4, 2 -x_3 - x_4, x_3, x_4])
print(x_p)
```

```
print('\n')
print('Confirming that Ax = b')
result = A.dot(x_p)
print(result)
```

```
Solution for Ax = b
[1 2 0 0]


Confirming that Ax = b
[1 4 9]
```

Part c

For finding all solutions to Ax = b, equations found in the part b will be used. Those equations are:

$x_1 = 1 + x_3 - 2x_4$

$x_2 = 2 - x_3 - x_4$

From these equations, we can write the solution set as:

$$x_n = \begin{pmatrix} 1 + x_3 - 2x_4 \\ 2 - x_3 - x_4 \\ x_3 \\ x_4 \end{pmatrix} \quad , \quad x_n = \begin{pmatrix} 1 \\ 2 \\ 0 \\ 0 \end{pmatrix} + x_3 \begin{pmatrix} 1 \\ -1 \\ 1 \\ 0 \end{pmatrix} + x_4 \begin{pmatrix} -2 \\ -1 \\ 0 \\ 1 \end{pmatrix}$$

Confirming that the found general solution satisfies Ax = b with Python:

[4]:
```
# Part c

x_3 = np.random.random()
x_4 = np.random.random()

x_c = np.array([ 1 + x_3 - 2*x_4, 2 -x_3 - x_4, x_3, x_4])

print('Confirming that Ax = b')
result = A.dot(x_c)
print(result)
```

```
Confirming that Ax = b
[1. 4. 9.]
```

Part d

To find the pseudoinverse of A, we will first write the singular value decomposition (SVD) of A.
$A = U\Sigma V^T$
First, we will compute $A^T.A$ and from that square matrix, we will get eigenvalues and corresponding eigenvectors. Those eigenvectors will be the columns of V.

$$A = \begin{pmatrix} 1 & 0 & -1 & 2 \\ 2 & 1 & -1 & 5 \\ 3 & 3 & 0 & 9 \end{pmatrix} \quad , \quad A^T = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 3 \\ -1 & -1 & 0 \\ 2 & 5 & 9 \end{pmatrix} \quad , \quad A^T.A = \begin{pmatrix} 14 & 11 & -3 & 39 \\ 11 & 10 & -1 & 32 \\ -3 & -1 & 2 & -7 \\ 39 & 32 & -7 & 110 \end{pmatrix}$$

Now, we will find the eigenvalues of $A^T.A$

$\lambda_1 = 68 + \sqrt{4301} \approx 133.582$
$\lambda_2 = 68 - \sqrt{4301} \approx 2.418$

$\lambda_3 = 0$
$\lambda_4 = 0$

With these eigen values, i could not manage to find SVD correct by manually. But the procedure is as follows:

With found eigen values, find corresponding eigenvectors. For each vector, divide it by it's length to make each of them orthonormal. Each orthonormal $eigenvector_i$ acquired from $eigenvalue_i$, gives $column_i$ of V matrix.

After finding V matrix, U matrix must be found. $A.V = U.\Sigma.V.V^T$ which is equal to $A.V = U.\Sigma$
For each component of U_k, we will use $A.V_k = \sigma_k.U_k$, where $\sigma_k = \sqrt{\lambda_k}$. For example, $U_1 = (1/\sigma_1).A.V_1$ where $\sigma_1 = \sqrt{\lambda_1} = \sqrt{133.582} \approx 11.558$ and V_1 is the orthonormal eigenvector found by $eigenvelue_1 = \lambda_1$.

Finally, $\Sigma[i,j]$ is the matrix with same shape of A and when i=j, $\Sigma[i,j] = \sigma_i$

V,U and sigma calculation with Python is as below:

V and V^T is:

$$V = \begin{pmatrix} -0.3217 & -0.2641 & 0.0576 & -0.9074 \\ 0.2702 & -0.5322 & -0.8023 & 0.0082 \\ 0.8900 & 0.2200 & 0.1499 & -0.3700 \\ 0.1772 & -0.7737 & 0.5749 & 0.1988 \end{pmatrix} \quad , \quad V^T = \begin{pmatrix} -0.3217 & 0.2702 & 0.8900 & 0.1772 \\ -0.2640 & -0.5322 & 0.2201 & -0.7737 \\ 0.0576 & -0.8023 & 0.1499 & 0.5749 \\ -0.9074 & 0.0082 & -0.3700 & 0.1988 \end{pmatrix}$$

U matrix is:

$$U = \begin{pmatrix} -0.1898 & 0.7002 & -0.6882 \\ -0.4761 & 0.5474 & 0.6882 \\ -0.8587 & -0.4583 & -0.2294 \end{pmatrix}$$

Finally, we can write our matrix A as;

$$A = \begin{pmatrix} -0.1898 & 0.7002 & -0.6882 \\ -0.4761 & 0.5474 & 0.6882 \\ -0.8587 & -0.4583 & -0.2294 \end{pmatrix} \begin{pmatrix} 11.558 & 0 & 0 & 0 \\ 0 & 1.555 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} -0.3217 & 0.2702 & 0.8900 & 0.1772 \\ -0.2640 & -0.5322 & 0.2201 & -0.7737 \\ 0.0576 & -0.8023 & 0.1499 & 0.5749 \\ -0.9074 & 0.0082 & -0.3700 & 0.1988 \end{pmatrix}$$

First find U, Σ, V and Σ^+ with Python.

[5]:
```
# Part d

u, s, v = np.linalg.svd(A)

# Make sigma in the correct form which is 3x4 matrix
sigma = np.zeros((3,4))
for i in range(len(sigma[:,0])):
    for j in range(len(sigma[0])):
        if( i == j ):
            # Instead of zero, it assigns third sigma value as 2*10^-16, which␣
 ↪makes problem when taking reciprocal.
            # To resolve that issue, i have just assigned 0 instead 2*10^-16
            if(s[i] > 10 **-15):
                sigma[i,j] = s[i]
            else:
                sigma[i,j] = 0
```

```
# To find sigma_plus, take reciprocals of the non-zero values.
sigma_plus = np.zeros((3,4))
for i in range(len(sigma[:,0])):
    for j in range(len(sigma[0])):
         if( i == j and sigma[i,j] != 0):
            sigma_plus[i,j] = 1/sigma[i,j]

print('U is:')
print(u)
print('V is:')
print(v)
print('sigma is:')
print(sigma)
```

```
U is:
[[-0.1898465   0.70019575 -0.6882472 ]
 [-0.47607011  0.54742401  0.6882472 ]
 [-0.85867081 -0.45831524 -0.22941573]]
V is:
[[-0.32168832 -0.26407196  0.05761637 -0.90744861]
 [ 0.27016145 -0.53217213 -0.80233358  0.00815077]
 [ 0.89002517  0.22009547  0.14994474 -0.37004021]
 [ 0.17715703 -0.77370331  0.57485455  0.19884876]]
sigma is:
[[11.55776837  0.          0.          0.        ]
 [ 0.          1.55498883  0.          0.        ]
 [ 0.          0.          0.          0.        ]]
```

Then calculate A^+ with both $A^+ = U^T\Sigma^+V$ and np.linalg.pinv

[6]:
```
A_pseudo = (v.T).dot(sigma_plus.T).dot(u.T)
print('Pseudo inverse of A by SVD decomposition :')
print(A_pseudo)
print('\n')
print('To see how accurate our pseudo inverse of A, is we check A.A_pseudo.A = A')
print(A.dot(A_pseudo).dot(A))

# Finally, find pseudo inverse of A with numpy.linalg.pinv(A)
A_pseudo_2 = np.linalg.pinv(A)
print('\n')
print('Pseudo inverse of A with numpy.linalg.pinv(A)')
print(A_pseudo_2)
```

```
Pseudo inverse of A by SVD decomposition :
[[ 0.12693498  0.10835913 -0.05572755]
 [-0.23529412 -0.17647059  0.17647059]
 [-0.3622291  -0.28482972  0.23219814]
 [ 0.01857585  0.04024768  0.06501548]]


To see how accurate our pseudo inverse of A, is we check A.A_pseudo.A = A
[[ 1.00000000e+00 -1.66533454e-16 -1.00000000e+00  2.00000000e+00]
 [ 2.00000000e+00  1.00000000e+00 -1.00000000e+00  5.00000000e+00]
 [ 3.00000000e+00  3.00000000e+00  1.66533454e-16  9.00000000e+00]]
```

```
Pseudo inverse of A with numpy.linalg.pinv(A)
[[ 0.12693498  0.10835913 -0.05572755]
 [-0.23529412 -0.17647059  0.17647059]
 [-0.3622291  -0.28482972  0.23219814]
 [ 0.01857585  0.04024768  0.06501548]]
```

As can be seen from the results, $A^+ = U^T\Sigma^+V$ gives same result with np.linalg.pinv. And the accuracy of A^+ is checked with $AA^+A = A$, which is a good approximation of A^+

Part e

In this part, sparsest solution of Ax = b will be found. Sparsest solution to a set is the solution set containing most zeros. To find the sparsest solution to Ax = b, first, rewrite the x that satisfies Ax = b.

$x_1 = 1 + x_3 - 2x_4$

$x_2 = 2 - x_3 - x_4$

$$x_n = \begin{pmatrix} 1 + x_3 - 2x_4 \\ 2 - x_3 - x_4 \\ x_3 \\ x_4 \end{pmatrix}$$

Since solution set x has 2 free variables, x_3 and x_4, maximum number of zeros in the solution set can be 2. This can be done in 3 ways: 1. Set $x_3 = 0$ and $x_4 = 0$. Then the solution is:

$$\begin{pmatrix} 1 \\ 2 \\ 0 \\ 0 \end{pmatrix}$$

2. Set one of the free variables to zero. And set other free variable such that one of the pivots become 0. For example: $x_3 = 0$ and $x_4 = 1/2$. Then the solution set is:

$$\begin{pmatrix} 0 \\ 3/2 \\ 0 \\ 1/2 \end{pmatrix}$$

3. Set free variables such that both pivots are zero. An example for that is $x_3 = 1$ and $x_4 = 1$. Then the solution is:

$$\begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

So the sparsest solution is not unique and there may be other combinations of x_3 and x_4 which makes solution set with 2 zeros. Python code with those given examples are:

[7]:
```
# Part e

# 1. Set free variables to zero
x_3 = 0
x_4 = 0
x_sparsest = np.array([ 1 + x_3 - 2*x_4, 2 -x_3 - x_4, x_3, x_4])

print('Sparsest solution example for x is')
print(x_sparsest)

# 2. Set one of the free variables to zero. And set other free variable such that␣
 ↪one of the pivots become 0.
```

```
x_3 = 0
x_4 = 1/2
x_sparsest = np.array([ 1 + x_3 - 2*x_4, 2 -x_3 - x_4, x_3, x_4])

print('Sparsest solution example for x is')
print(x_sparsest)

# 3.Set free variables such that both pivots are zero.
x_3 = 1
x_4 = 1
x_sparsest = np.array([ 1 + x_3 - 2*x_4, 2 -x_3 - x_4, x_3, x_4])

print('Sparsest solution example for x is')
print(x_sparsest)
```

```
Sparsest solution example for x is
[1 2 0 0]
Sparsest solution example for x is
[0.  1.5 0.  0.5]
Sparsest solution example for x is
[0 0 1 1]
```

Part f

In this part, minimum Euclidian norm to Ax = b will be found. Minimum Euclidian form for x can be written as:

$$||x|| = \sqrt{\sum_{n=i} x_i^2}$$

To find the minimum Euclidan form, we can either find the least square norm or we can calculate $A^+.b$. Since we have found A^+ in part d, we can use it to easily calculate the minimum Euclidian form. Python code for that is given below:

[7]:
```
# Part f

L2 = A_pseudo.dot(b)
print('Least norm solution is:')
print(L2)
```

```
Least norm solution is:
[0.05882353 0.64705882 0.58823529 0.76470588]
```

Question 2

In this question, we are given fMRI reports about the activation of Broca's area in language involving tasks. We are given that Broca's area is activated in 103 out of 869 language involved tasks. Also, this area is activated in 199 out of 2353 tasks that do not involve language.

Part a

Conditional probability of activation given language and no language follows a Bernoulli distribution with probability p for active and 1-p for not active. For language involved tasks p = x_l and for tasks that do not involve language p = x_{nl}. Then, we can define the Bernoulli distribution as:

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

Here, data represents whether the Broca's area is active or not which is denoted by either 1 or 0. And x is the probability of activation as previously mentioned. We are also given that the data obtained from experiments are independently and identically distributed. With these informations, we can write the likelihoods as:

$$P(data|x_l) = \binom{869}{103} * x_l^{103} * (1-x_l)^{869-103} \qquad P(data|x_{nl}) = \binom{2353}{199} * x_{nl}^{199} * (1-x_{nl})^{2353-199}$$

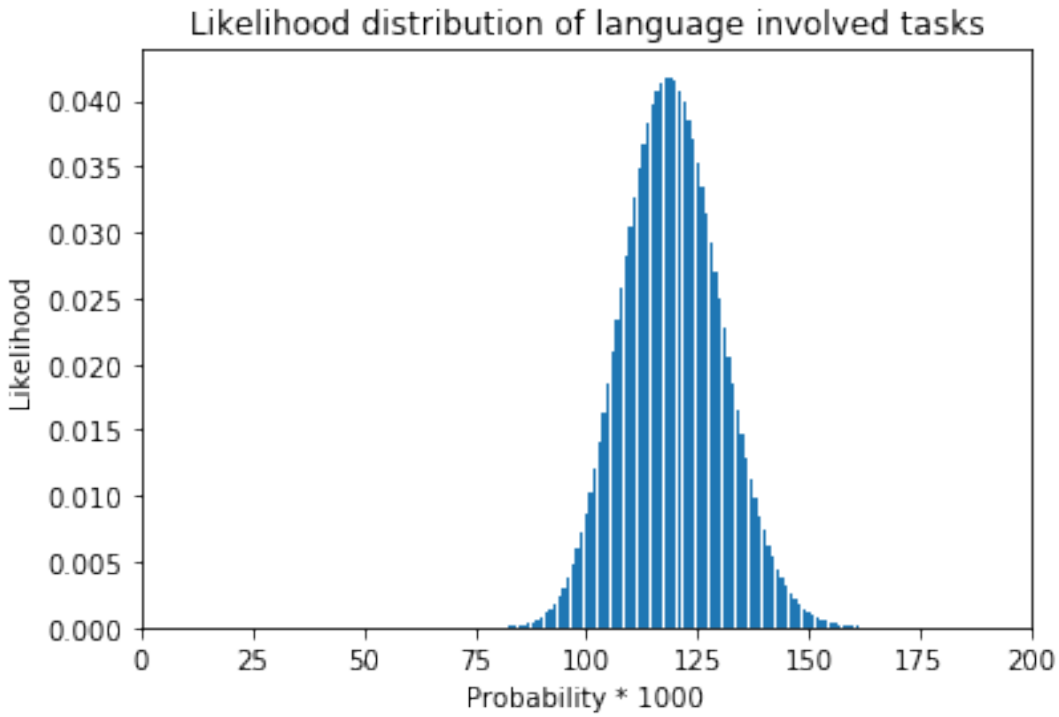
[1]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
```

[2]:
```
# Part a

def nChooseK(n,k):
    return math.factorial(n) / (math.factorial(k) * math.factorial(n-k))
```

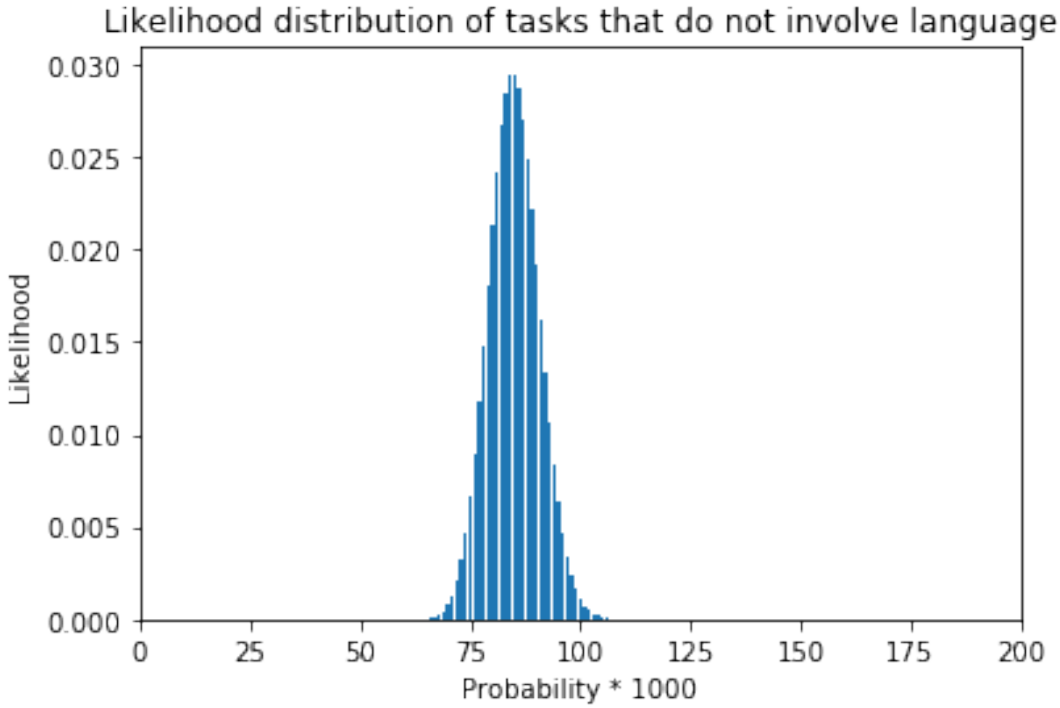
[3]:
```
def bernoulliDist(n,k,p):
    return nChooseK(n,k)*(p**k)*((1-p)**(n-k))
```

[4]:
```
x = np.arange(0, 1.001, 0.001)
likelihood_lang = bernoulliDist(869, 103, x)
likelihood_notlang = bernoulliDist(2353, 199, x)
```

[5]:
```
index = np.arange(0,1001)
plt.xlim(0,200)
plt.bar(index, likelihood_lang)
plt.xlabel('Probability * 1000')
plt.ylabel('Likelihood')
plt.title('Likelihood distribution of language involved tasks')
plt.show(block = False)
```



[6]:
```
plt.xlim(0,200)
plt.bar(index, likelihood_notlang)
plt.xlabel('Probability * 1000')
plt.ylabel('Likelihood')
plt.title('Likelihood distribution of tasks that do not involve language')
plt.show(block = False)
```



In the bar charts, I have limited the x axis between (0,200) to make datas seem more clear. For both charts, x_l and x_{nl} values are multiplied with 1000 for convenience.

Part b

In this part, we are asked to find the x_l and x_{nl} values which maximizes their respective likelihood functions. Since we plotted the both likelihood functions, we can easily find the max value from the likelihood functions and get the corresponding index to find asked x_l and x_{nl} values. Python code for that is:

[7]:
```
# Part b

max_l = np.amax(likelihood_lang)
max_nl = np.amax(likelihood_notlang)

max_l_prob = 0
max_nl_prob = 0
for i in range(len(likelihood_lang)):
    if (likelihood_lang[i] == max_l):
        #Dividing the i by 1000 since the probability is scaled by 1000.
        max_l_prob = i/1000

    if (likelihood_notlang[i] == max_nl):
        max_nl_prob = i/1000

print('The probability that maximizes the likelihood of language involving tasks:␣
 ↪(probability, likelihood of that probability)')
print(max_l_prob, max_l)
print('The probability that maximizes the likelihood of not language involving␣
 ↪tasks: (probability, likelihood of that probability)')
print(max_nl_prob, max_nl)
```

```
The probability that maximizes the likelihood of language involving tasks:
(probability, likelihood of that probability)
0.119 0.0417952478261316
The probability that maximizes the likelihood of not language involving tasks:
(probability, likelihood of that probability)
0.085 0.02946375315559796
```

Since in the bar charts, I have scaled both x's around (0,1000), after finding the appropriate index, that x value is divided by 1000 to get real p=x which is scaled back to interval (0,1).

Part c

In part a , we found the likelihood function $P(data|X)$. In this part, we need to find posterior function $P(X|data)$. In order to make that transition between likelihood and posterior, Bayes' rule will be implemented.

$$P(data|X) * P(X) = P(X|data) * P(data) \qquad P(X|data) = P(data|X) * P(X)/P(data)$$

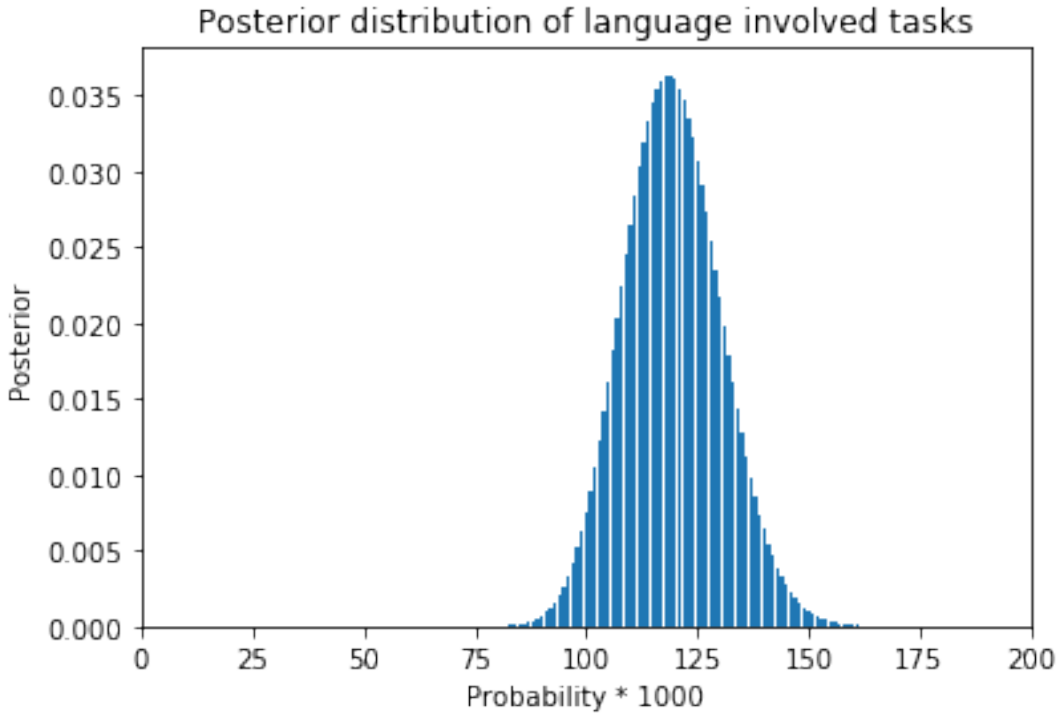
where $P(X)$ = prior and $P(data)$ = normalizer. To find prior, we are given that P(X) is uniformly distributed. Thus, prior can be written as: P(X) = prior = $1/1001$ which is the size of both x_l and x_{nl}. For $P(data)$ can be calculated by computing the $\sum_x P(data|X) * P(X)$. Therefore, $P(X|data)$ can be written as:

$$P(X|data) = P(data|X) * P(X) / \sum_x P(data|X) * P(X)$$

where P(X) = prior = 1/1001. For calculating the posterior $P(data|x_l)$ and $P(data|x_{nl})$, Python code is given below:

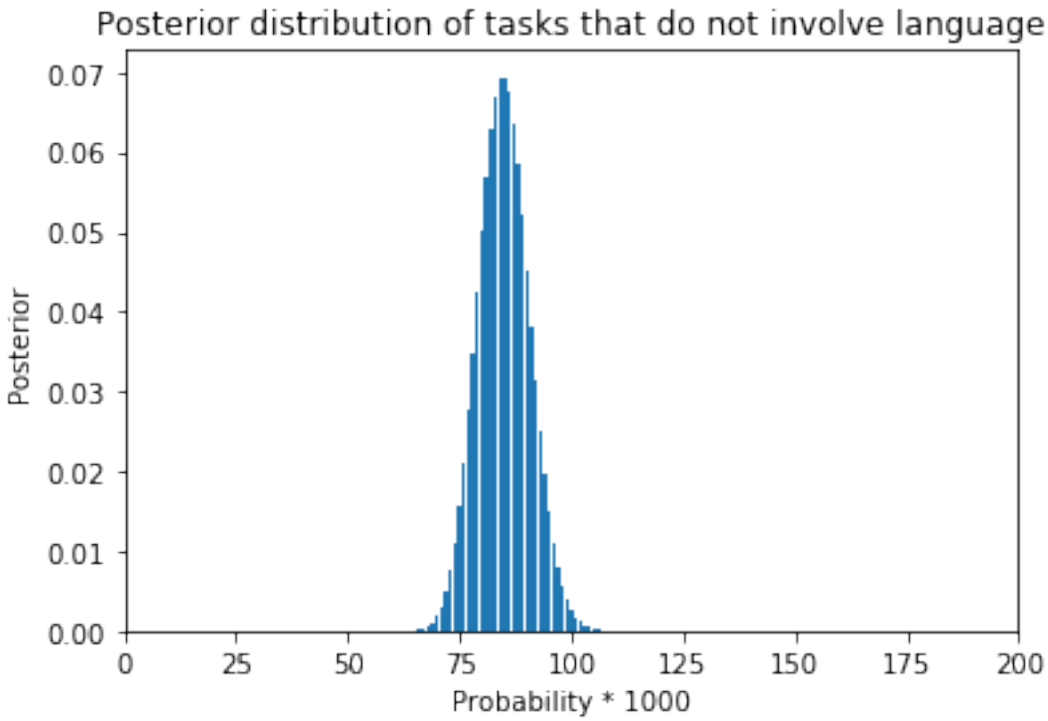
[8]:
```
# Part c

# x has 1001 values, and since it is given that uniformly distributed, prior P(X) =␣
 ↪1/1001
prior = 1/1001

normalizer_l = 0
posterior_l = np.zeros(len(likelihood_lang))
normalizer_nl = 0
posterior_nl = np.zeros(len(likelihood_notlang))

for i in range(len(likelihood_lang)):
    normalizer_l += likelihood_lang[i] * prior
    normalizer_nl += likelihood_notlang[i] * prior

posterior_l[:] = likelihood_lang[:] * prior / normalizer_l
posterior_nl[:] = likelihood_notlang[:] * prior / normalizer_nl
```

[9]:
```
plt.xlim(0,200)
plt.bar(index, posterior_l)
plt.xlabel('Probability * 1000')
plt.ylabel('Posterior')
plt.title('Posterior distribution of language involved tasks')
plt.show(block = False)
```



[10]:
```
plt.xlim(0,200)
plt.bar(index, posterior_nl)
plt.xlabel('Probability * 1000')
```

```
plt.ylabel('Posterior')
plt.title('Posterior distribution of tasks that do not involve language')
plt.show(block = False)
```



After finding posteriors, we are asked to find 95% confidence intervals for both tasks that involve language and do not involve language. For finding the lower and upper limits for that confidence interval, 2.5th and 97.5th percentiles must be found. Those percentiles are the lower and upper limits of 95% confidence interval. For this, we will use cumulative distribution (CDF). To find the limits, we will use:

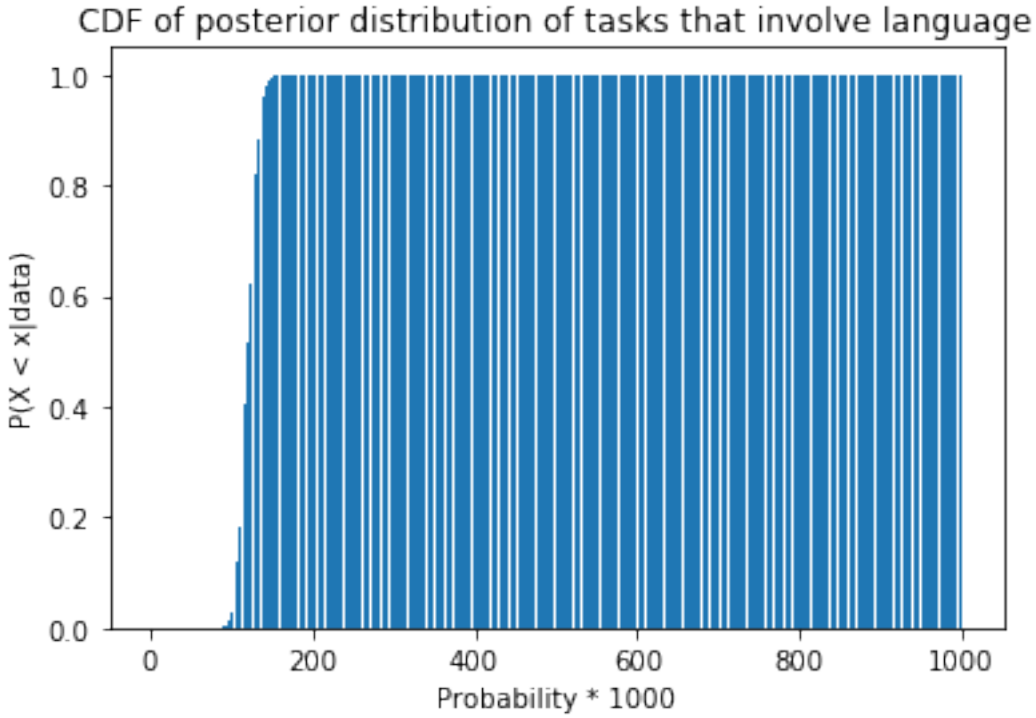
$$P(X < x_{lower}|data) > 0.025 \qquad P(X < x_{upper}|data) < 0.975$$

Python code for CDF's of tasks that involve language and do not involve language are given below. Also, 95% confidence intervals for both tasks are also implemented.

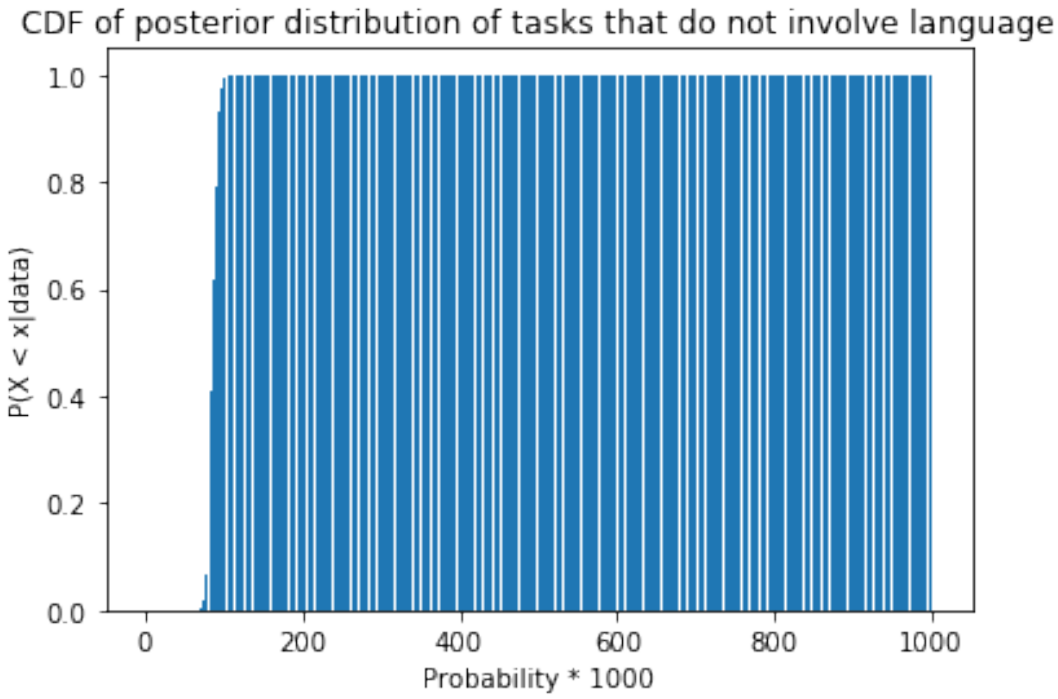
[11]:
```
def pdf_to_cdf(posterior):
    lowerbound = 0
    temp_min = 0
    upperbound = np.inf
    temp_max = np.inf
    cdf = np.zeros(len(posterior) + 1)
    for i in range(1, len(posterior) +1):
        for j in range(i):
            cdf[i] += posterior[j]
            # Since i have iterated i by 1, when finding lowerbound and upperbound,␣
 ↪i decrease 1 from i.
        if (cdf[i] >= 0.025 and lowerbound <= temp_min):
            temp_min = cdf[i]
            lowerbound = (i-1)/1000
        if (cdf[i] >= 0.975 and upperbound >= temp_max):
            temp_max = cdf[i]
```

```
            upperbound = (i-1)/1000
    return cdf,lowerbound, upperbound

cdf_l,lower_l, upper_l = pdf_to_cdf(posterior_l)
cdf_nl, lower_nl, upper_nl = pdf_to_cdf(posterior_nl)
```

[12]:
```
index2 = np.arange(0,1002)
plt.bar(index2, cdf_l)
plt.xlabel('Probability * 1000')
plt.ylabel('P(X < x|data)')
plt.title('CDF of posterior distribution of tasks that involve language')
plt.show(block = False)
```



[13]:
```
plt.bar(index2, cdf_nl)
plt.xlabel('Probability * 1000')
plt.ylabel('P(X < x|data)')
plt.title('CDF of posterior distribution of tasks that do not involve language')
plt.show(block = False)
```



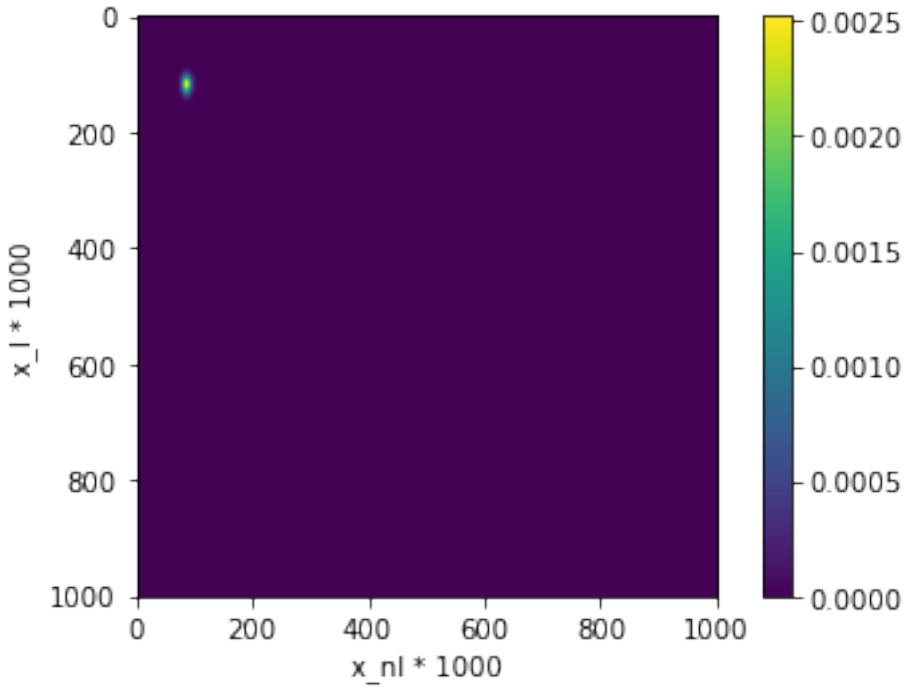
[14]:
```
print('Confidence interval of x_l is (', lower_l, ',', upper_l, ')' )
print('Confidence interval of x_nl is (', lower_nl, ',', upper_nl, ')' )
```

```
Confidence interval of x_l is ( 0.099 , 0.142 )
Confidence interval of x_nl is ( 0.074 , 0.096 )
```

Part d

In this part, we will find the joint distribution $P(X_l, X_{nl}|data)$. We are given that these two frequencies are independent, so the joint distribution of these two functions is given by the outer product of these two marginals. Python code for the joint distribution is given below. (Imshow is used for plotting the joint distribution.)

[15]:
```
# Part d

matrixPosterior_l  = np.matrix(posterior_l)
matrixPosterior_nl = np.matrix(posterior_nl)
joint_dist = (matrixPosterior_l.T).dot(matrixPosterior_nl)

plt.imshow(joint_dist)
plt.xlabel('x_nl * 1000')
plt.ylabel('x_l * 1000')
plt.colorbar()
plt.show(block = False)
```



Using CDF, we will compute the $P(X_l > X_{nl}|data)$ and $P(X_l <= X_{nl}|data)$. For $P(X_l > X_{nl}|data)$, we will sum points where condition i > j satisfies. Because $X_l > X_{nl}$ corresponds to data points in the CDF[i,j] where i > j. Alternatively, we can say that for $X_l > X_{nl}$, we will sum data point in the lower triangular part in the CDF. Similarly, for $P(X_l <= X_{nl}|data)$, data point where i <= j will be summed.

[16]:
```
x_l_greater = 0
x_l_notgreater = 0
for i in range(len(joint_dist)):
    for j in range(len(joint_dist)):
        if ( i > j ):
            x_l_greater += joint_dist[i,j]
        else:
            x_l_notgreater += joint_dist[i,j]

print('Sum of posteriors such that x_l > x_nl')
print(x_l_greater)
print('Sum of posteriors such that x_l <= x_nl')
print(x_l_notgreater)
```

```
Sum of posteriors such that x_l > x_nl
0.9978520275861253
Sum of posteriors such that x_l <= x_nl
0.002147972413864104
```

Part e

We are given prior $P(language) = 0.5$ and asked to compute $P(language|activation)$. For that, Bayes' rule will be implemented which gives:

$$P(language|activation) * P(activation) = P(activation|language) * P(language)$$
$$P(language|activation) = P(activation|language) * P(language) / P(activation)$$

For $P(activation)$, we need to extend it to:

$$P(activation) = P(activation|language) * P(language) + P(activation|notLanguage) * P(notLanguage)$$

where, $P(language)$ = 0.5 and $P(notLanguage)$ = 1 - $P(language)$ = 1 - 0.5 = 0.5. For both $P(activation|language)$ and $P(activation|notLanguage)$, their corresponding ML estimate values will be used. Thus, we can say that $P(activation|language) = max(x_l)$ and $P(activation|notLanguage) = max(x_{nl})$.

Python code for that is:

[17]:
```
# Part e

PROB_LANG = 0.5

prob_lang_active = ( max_l_prob * PROB_LANG ) / ( ( max_l_prob * PROB_LANG ) +␣
 ↪max_nl_prob * (1 - PROB_LANG) )
print('P(language | activation) is :')
print(prob_lang_active)
```

```
P(language | activation) is :
0.5833333333333333
```

References

[1] Strang and Gilbert, “Lecture 21: Eigenvalues and eigenvectors,” MIT OpenCourseWare, Massachusetts Institute of Technology. [Online]. Available: https://ocw.mit.edu/courses/mathematics/18-06-linear-algebra-spring-2010/video-lectures/lecture-21-eigenvalues-and-eigenvectors/. [Accessed: 22-Feb-2020].

[2] Strang and Gilbert, “Lecture 33: Left and right inverses; pseudoinverse,” MIT OpenCourseWare, Massachusetts Institute of Technology. [Online]. Available: https://ocw.mit.edu/courses/mathematics/18-06-linear-algebra-spring-2010/video-lectures/lecture-33-left-and-right-inverses-pseudoinverse/. [Accessed: 22-Feb-2020].