AI lab : Assignment 6

Team : PUVBP

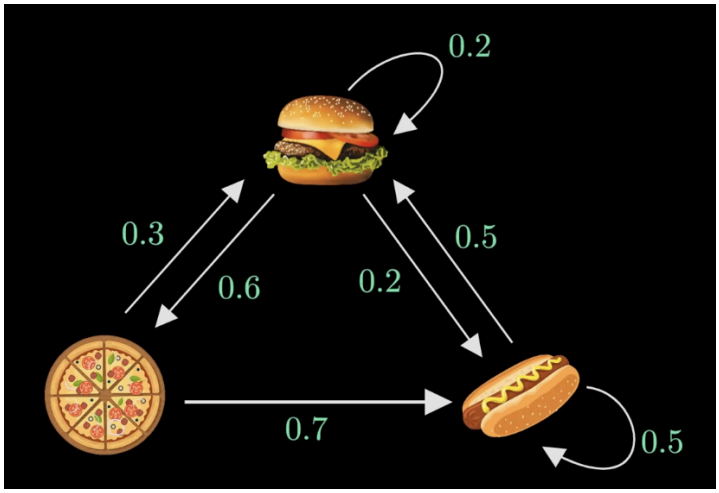
202151139	SANJEEVANI BALAJI LAKADE
202151156	SIDDHARTH GUPTA
202151171	TARUN MANOJ KUMAR SAHU
202151172	THORAT SPRUHA MOHAN
202151181	YASH KUMAR SINGH

Learning Objective:

To implement Expectation Maximization routine for learning parameters of a Hidden Markov Model, to be able to use the EM framework for deriving algorithms for problems with hidden or partial information.

Introduction:

1. Markov Chains:
   A Markov chain is a mathematical model that uses probability rules to describe a sequence of events. It predicts the probability of a sequence of events based on the most recent event.



Properties:

- The future state depends only on the current state and not on the earlier states
- Sum of the weights of the outgoing Arrow from any state is equal to 1.

2. Hidden Markov Model
   A hidden Markov model (HMM) is a Markov model in which the observations are dependent on a latent (or 'hidden') Markov process (referred to as X). An HMM requires that there be an observable process Y whose outcomes depend on the outcomes of X in a known way. Since X cannot be observed directly, the goal is to learn about the state of X by observing Y.

Components:

- **Hidden states:** The states that are not directly observable but influence the observable variables.
- **Observations:** These are the observable outcomes or measurements associated with each time step. Observations are influenced by the hidden states but are directly visible or measurable.
- **Transition Probabilities (A):** These are the probabilities of transitioning between hidden states. They represent the likelihood of moving from one hidden state to another in the next time step.
- **Emission Probabilities (B):** These are the probabilities of observing a particular output given the current hidden state. They define the likelihood of each observation being emitted from each hidden state.
- **Initial State Distribution (π):** These are the probabilities associated with starting the process in each possible hidden state. It represents the likelihood of the system starting in a particular hidden state.

Example:

$$\begin{array}{c} \\ H \\ C \end{array} \begin{array}{c} \begin{array}{cc} H & C \end{array} \\ \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \end{array}$$

$$\begin{array}{c} \\ H \\ C \end{array} \begin{array}{c} \begin{array}{ccc} S & M & L \end{array} \\ \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix} \end{array}.$$

The state transition matrix

$$A = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \tag{3}$$

comes from (1) and the observation matrix

$$B = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix}. \tag{4}$$

is from (2). In this example, suppose that the initial state distribution, denoted by π, is

$$\pi = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix}. \tag{5}$$

The matrix $A = \{a_{ij}\}$ is $N \times N$ with

$$a_{ij} = P(\text{state } q_j \text{ at } t+1 \,|\, \text{state } q_i \text{ at } t)$$

and A is row stochastic. Note that the probabilities a_{ij} are independent of t. The matrix $B = \{b_j(k)\}$ is an $N \times M$ with

$$b_j(k) = P(\text{observation } k \text{ at } t \,|\, \text{state } q_j \text{ at } t).$$

Our goal is to make effective and efficient use of the observable information so as to gain insight into various aspects of the Markov process.

Now consider a particular four-year period of interest from the distant past, for which we observe the series of tree rings S, M, S ,L. Letting 0 represent S, 1 represent M and 2 represent L, this observation sequence is : **O = (0, 1, 0, 2).**

$$P(X, \mathcal{O}) = \pi_{x_0} b_{x_0}(\mathcal{O}_0) a_{x_0,x_1} b_{x_1}(\mathcal{O}_1) a_{x_1,x_2} b_{x_2}(\mathcal{O}_2) a_{x_2,x_3} b_{x_3}(\mathcal{O}_3).$$

$$P(HHCC) = 0.6(0.1)(0.7)(0.4)(0.3)(0.7)(0.6)(0.1) = 0.000212.$$

Similarly we can calculate the probability for different **State Sequences.**
The most optimal sequence can be calculated in two ways :

- **DP sense**
- **HMM sense**

Table 1: State sequence probabilities

state	probability	normalized probability
$HHHH$	.000412	.042787
$HHHC$	.000035	.003635
$HHCH$	.000706	.073320
$HHCC$	.000212	.022017
$HCHH$	.000050	.005193
$HCHC$	.000004	.000415
$HCCH$	.000302	.031364
$HCCC$	.000091	.009451
$CHHH$	.001098	.114031
$CHHC$	.000094	.009762
$CHCH$	.001882	.195451

$CHCC$	.000564	.058573
$CCHH$	.000470	.048811
$CCHC$	.000040	.004154
$CCCH$	.002822	.293073
$CCCC$	.000847	.087963

Table 2: HMM probabilities

	element			
	0	1	2	3
$P(H)$	0.188182	0.519576	0.228788	0.804029
$P(C)$	0.811818	0.480424	0.771212	0.195971

The 3 problems which we can solve using HMM :

1. **Evaluation Problem(forward algorithm)**
   Given the model λ = (A, B, π) and a sequence of observations O, find P (O | λ). Here, we want to determine a score for the observed sequence O with respect to the given model λ.

$$\alpha_t(i) = \left[\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji}\right] b_i(\mathcal{O}_t)$$

2. **Decoding Problem (backward algorithm)**
   Given λ = (A, B, π) and an observation sequence O, find an optimal state sequence for the underlying Markov process. In other words, we want to uncover the hidden part of the Hidden Markov Model. This type of problem is discussed in some detail in Section 1, above.

$$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j).$$

3. **Learning Problem**
   Given an observation sequence O and the dimensions N and M, find the model λ = (A,B,π) that maximizes the probability of O. This can be viewed as training a model to best fit the observed data. Alternatively, we can view this as a (discrete) hill climb on the parameter space represented by A, B and π.

Concept : **Learning the parameters of HMM**

A. Read through the reference carefully. Implement routines for learning the parameters of HMM given in section 7. In section 8, “A not-so-simple example”, an interesting exercise is carried out. Perform a similar experiment on “War and Peace” by Leo Tolstoy.

```
# Importing
import numpy as np
import re
```

```
# Reading the Book and Preprocessing

# Reading the file
book = 'War_and_Peace.txt'
file = open(book, 'r', encoding='utf-8')
text = file.read()
file.close()

# Removing the punctuations and converting to lower case
text = re.sub(r'[^a-zA-Z]', " ", text)
text = " ".join(text.split()).lower()[:100000]

# Creating a dictionary of all the unique characters
dictionary = {}
```

```
for i in range(26):
   dictionary[chr(i + 97)] = i
dictionary[" "] = 26
# Initialize the parameters

# Observed sequence
O = np.zeros(len(text), dtype=int)

for i in range(len(text)):
   O[i] = dictionary[text[i]]
# Initial state distribution
pi = np.array(([0.525483, 0.474517]))
# Observable sequence
B = np.array([[0.03735, 0.03408, 0.03455, 0.03828, 0.03782,
0.03922, 0.03688, 0.03408, 0.03875, 0.04062, 0.03735,
0.03968, 0.03548, 0.03735, 0.04062, 0.03595, 0.03641,
0.03408, 0.04062, 0.03548, 0.03922, 0.04062, 0.03455,
0.03595, 0.03408, 0.03408, 0.03688],
             [0.03909, 0.03537,  0.03537, 0.03909, 0.03583,
0.03630, 0.04048, 0.03537, 0.03816, 0.03909, 0.03490,
0.03723, 0.03537, 0.03909, 0.03397, 0.03397, 0.03816,
0.03676, 0.04048, 0.03443, 0.03537, 0.03955, 0.03816,
0.03723,  0.03769, 0.03955, 0.03397]])
# Transition matrix
A = np.array([[0.47468, 0.52532], [0.51656, 0.48344]])
# Set of possible observations
V = np.array(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',
'v', 'w', 'x', 'y', 'z', ' '])
# Set of possible states, Q is hidden
# Number of observation symbols
M = len(V)
# Number of states in the model
```

```
N = len(A)
# Length of observation sequence
T = len(O)
```

Alpha Pass(forward algorithm)

```
# Alpha Pass

def alpha_pass(A1, B1, pi1, O1):
   c1 = np.zeros([T, 1])
    alpha1 = np.zeros([T, N])
    c1[0][0] = 0
    for x in range(N):
        alpha1[0][x] = pi1[x] * B1[x][O1[0]]
        c1[0][0] = c1[0][0] + alpha1[0][x]
    c1[0][0] = 1/c1[0][0]
    for x in range(N):
        alpha1[0][x] = c1[0][0] * alpha1[0][x]

    for t in range(1, T):
        c1[t][0] = 0
        for x in range(N):
            alpha1[t][x] = 0
            for y in range(N):
                alpha1[t][x] = alpha1[t][x] + alpha1[t-1][y] *
A1[y][x]
            alpha1[t][x] = alpha1[t][x] * B1[x][O1[t]]
            c1[t][0] = c1[t][0] + alpha1[t][x]
        c1[t][0] = 1/c1[t][0]
        for x in range(N):
            alpha1[t][x] = c1[t][0] * alpha1[t][x]
    return alpha1, c1
```

Beta pass(Backward algorithm)

```
# Beta Pass
def beta_pass(A1, B1, O1, c1):
   beta1 = np.zeros([T, N])
   for x in range(N):
       beta1[T-1][x] = c1[T-1][0]
   for t in range(T-2, -1, -1):
       for x in range(N):
           beta1[t][x] = 0
         for y in range(N):
               beta1[t][x] = beta1[t][x] + A1[x][y] *
B1[y][O1[t + 1]] * beta1[t + 1][y]
           beta1[t][x] = c1[t][0] * beta1[t][x]
   return beta1
```

Computing Gamma

```
# Compute Gamma(x,t) and Gamma(x,y,t)

def gamma_pass(alpha1, beta1, A1, B1, O1):
   gamma1 = np.zeros([T, N])
   di_gamma1 = np.zeros([T, N, N])
   for t in range(T-1):
       for x in range(N):
           gamma1[t][x] = 0
           for y in range(N):
               di_gamma1[t][x][y] = alpha1[t][x] * A1[x][y] *
B1[y][O1[t + 1]] * beta1[t + 1][y]
               gamma1[t][x] = gamma1[t][x] +
di_gamma1[t][x][y]
   for x in range(N):
       gamma1[T-1][x] = alpha1[T-1][x]
   return gamma1, di_gamma1
```

Re-estimating the parameters

```
# Re-estimate A, B, pi
def re_estimate(gamma1, di_gamma1, A1, B1, pi1):
# pi
 for x in range(N):
       pi1[x] = gamma1[0][x]
  # A1
for x in range(N):
      denominator = 0
       for t in range(T-1):
           denominator = denominator + gamma1[t][x]
       for y in range(N):
           numerator = 0
           for t in range(T-1):
               numerator = numerator + di_gamma1[t][x][y]
           A1[x][y] = numerator/denominator
# B1
 for x in range(N):
       denominator = 0
       for t in range(T):
           denominator = denominator + gamma1[t][x]
       for y in range(M):
           numerator = 0
           for t in range(T):
               if O[t] == y:
                   numerator = numerator + gamma1[t][x]
           B1[x][y] = numerator/denominator
   return A1, B1, pi1
```

Calculating Log-likelihood

```
# Compute log[P(O|lambda)]
```

```
def log_prob(c1):
    logProb1 = 0
    for x in range(T):
        logProb1 = logProb1 + np.log(c1[x][0])
    logProb1 = -logProb1
    return logProb1
```

Initial Values

```
# Values initially

oldLogProb = -10000000
print("A: \n", A)
print("B: \n", np.concatenate((V.reshape(1, M), B),
axis=0).T)
print("pi: ", pi)
print("logProb: ", oldLogProb)
```

Output

```
A:
 [[0.47468 0.52532]
 [0.51656 0.48344]]
B:
 [['a' '0.03735' '0.03909']
 ['b' '0.03408' '0.03537']
 ['c' '0.03455' '0.03537']
 ['d' '0.03828' '0.03909']
 ['e' '0.03782' '0.03583']
 ['f' '0.03922' '0.0363']
 ['g' '0.03688' '0.04048']
 ['h' '0.03408' '0.03537']
 ['i' '0.03875' '0.03816']
 ['j' '0.04062' '0.03909']
 ['k' '0.03735' '0.0349']
 ['l' '0.03968' '0.03723']
 ['m' '0.03548' '0.03537']
 ['n' '0.03735' '0.03909']
 ['o' '0.04062' '0.03397']
 ['p' '0.03595' '0.03397']
 ['q' '0.03641' '0.03816']
 ['r' '0.03408' '0.03676']
 ['s' '0.04062' '0.04048']
 ['t' '0.03548' '0.03443']
 ['u' '0.03922' '0.03537']
 ['v' '0.04062' '0.03955']
 ['w' '0.03455' '0.03816']
 ['x' '0.03595' '0.03723']
 ['y' '0.03408' '0.03769']
 ['z' '0.03408' '0.03955']
 [' ' '0.03688' '0.03397']]
pi:  [0.525483 0.474517]
logProb:  -10000000
```

```
# After first iteration

alpha, c = alpha_pass(A, B, pi, O)
beta = beta_pass(A, B, O, c)
gamma, di_gamma = gamma_pass(alpha, beta, A, B, O)
A, B, pi = re_estimate(gamma, di_gamma, A, B, pi)
logProb = log_prob(c)

print("A: \n", A)
print("B: \n", np.concatenate((V.reshape(1, M), np.round_(B,
decimals=7)), axis=0).T)
print("pi: ", np.round_(pi, decimals=7))
print("logProb: ", logProb)
```

Output :

```
A:
 [[0.48026894 0.51973106]
 [0.52229627 0.47770373]]
B:
 [['a' '0.0650332' '0.0696181']
 ['b' '0.0103124' '0.0109893']
 ['c' '0.0214131' '0.0224695']
 ['d' '0.0345775' '0.0362065']
 ['e' '0.1003477' '0.0971644']
 ['f' '0.0150525' '0.0143257']
 ['g' '0.0155683' '0.0174964']
 ['h' '0.0509223' '0.0541456']
 ['i' '0.062085' '0.0625161']
 ['j' '0.0007938' '0.0007862']
 ['k' '0.0067203' '0.006439']
 ['l' '0.0327946' '0.0315423']
 ['m' '0.0185224' '0.0189587']
 ['n' '0.0554752' '0.0594546']
 ['o' '0.062998' '0.0539197']
 ['p' '0.0194886' '0.0188698']
 ['q' '0.0005581' '0.000602']
 ['r' '0.0461758' '0.0510963']
 ['s' '0.0488063' '0.0498764']
 ['t' '0.0695088' '0.0691503']
 ['u' '0.021594' '0.019962']
 ['v' '0.0114156' '0.0114044']
 ['w' '0.0167512' '0.0189944']
 ['x' '0.004376' '0.0046447']
 ['y' '0.0144712' '0.0164336']
 ['z' '0.000347' '0.0004132']
 [' ' '0.1938911' '0.1825209']]
pi:  [0.5333767 0.4666233]
logProb:  -330338.2213654224
```

```
# Output

maxIter = 100
for ite in range(maxIter):
    alpha, c = alpha_pass(A, B, pi, O)
    beta = beta_pass(A, B, O, c)
    gamma, di_gamma = gamma_pass(alpha, beta, A, B, O)
    A, B, pi = re_estimate(gamma, di_gamma, A, B, pi)
    logProb = log_prob(c)

print("A: \n", A)
print("B: \n", np.concatenate((V.reshape(1, M), np.round_(B,
decimals=7)), axis=0).T)
print("pi: ", np.round_(pi, decimals=5))
print("logProb: ", logProb)
```

Output :

```
A:
 [[0.28002118 0.71997882]
 [0.70820408 0.29179592]]
B:
 [['a' '0.1336603' '0.0020659']
 ['b' '0.0' '0.0211256']
 ['c' '1.37e-05' '0.0435073']
 ['d' '1e-07' '0.0702005']
 ['e' '0.1991638' '2e-07']
 ['f' '0.0' '0.0291395']
 ['g' '0.0006941' '0.0321066']
 ['h' '0.0012589' '0.1029616']
 ['i' '0.1249793' '0.000647']
 ['j' '0.0' '0.0015671']
 ['k' '1e-07' '0.0130522']
 ['l' '0.0056878' '0.0582186']
 ['m' '0.0' '0.0371732']
 ['n' '6.9e-06' '0.1139724']
 ['o' '0.1179133' '0.0']
 ['p' '0.0009427' '0.0371186']
 ['q' '0.0' '0.0011505']
 ['r' '0.0' '0.0964637']
 ['s' '1e-07' '0.0978721']
 ['t' '5.43e-05' '0.1374714']
 ['u' '0.0359264' '0.0058816']
 ['v' '0.0' '0.0226332']
 ['w' '0.0' '0.0354474']
 ['x' '0.0002303' '0.0087197']
 ['y' '1.6e-06' '0.0306455']
 ['z' '0.0' '0.0007538']
 [' ' '0.3794664' '0.000105']]
pi:  [0. 1.]
logProb:  -275231.1599944892
```

Concept : **Expectation Maximization**

The Expectation-Maximization (EM) algorithm is an iterative optimization technique used to estimate parameters for probabilistic models when some of the variables are not observed (i.e., they are hidden or latent variables). The algorithm aims to find the maximum likelihood (ML)

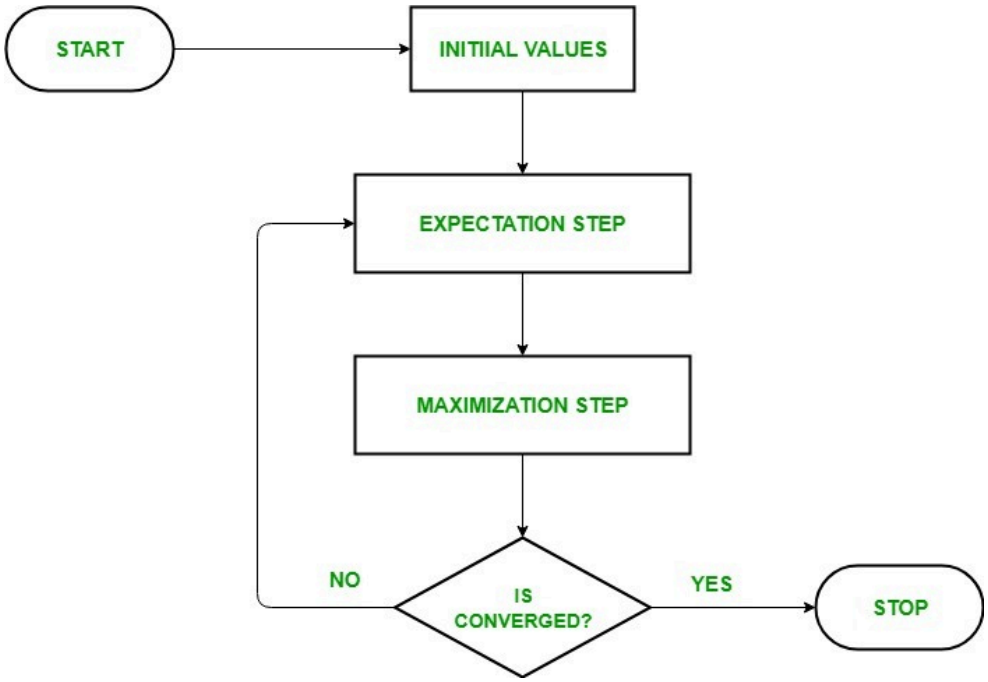
Steps :

1) **Initialization** : EM starts with an initial guess of the parameters of the model.
2) **Expectation step (E-step)** :
   - We use the observed data in order to estimate or guess the values of the missing or incomplete data.
   - Estimate the missing or incomplete data values using the current parameter estimates
   - Compute the log-likelihood of the observed data based on the current parameter and estimated missing data.
3) **Maximization step (M-step)**:

- This step updates the parameters of the model using the complete data (including the observed data and the estimated missing data) obtained from the E-step.
- Update the parameters of the model by maximizing the expected complete data log-likelihood obtained from the E-step.
- The EM algorithm involves solving optimization problems to maximize the log-likelihood, with the specific technique chosen based on the problem's nature and the model's characteristics.

4) **Convergence Check**: Convergence refers to the condition when the EM algorithm has reached a stable solution.

Flow chart :



Advantages of EM algorithm

1) Easy implementation .
2) It is mostly guaranteed that likelihood will enhance after each iteration, ensuring that the algorithm converges towards a locally optimal solution

Disadvantages of EM algorithm

1) It has slow convergence.
2) It converges to the local optima only.

Question 2:

Ten bent (biased) coins are placed in a box with unknown bias values. A coin is randomly picked from the box and tossed 100 times. A file containing results of five hundred such instances is presented in tabular form with 1 indicating head and 0 indicating tail. Find out the unknown bias values. (2020_ten_bent_coins.csv)

Code:

```
#imports
import pandas as pd
import numpy as np
from scipy.special import comb
```

```
df = pd.read_csv('./2020_ten_bent_coins.csv').transpose()
print(df)
```

```
      0   1   2   3   4   5   6   7   8   9   ...  490  491  492  \
1     1   0   1   0   1   1   0   1   1   0  ...    0    0    1
2     1   0   0   0   0   1   0   0   1   0  ...    1    1    1
3     1   0   1   0   0   1   0   0   1   0  ...    0    1    0
4     1   0   1   0   1   1   0   1   0   0  ...    1    0    1
5     1   0   0   0   0   1   1   0   0   0  ...    0    1    0
..   ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...
96    1   0   0   0   0   1   1   0   0   1  ...    1    1    1
97    1   1   1   0   1   1   0   0   0   0  ...    1    1    1
98    1   0   1   0   1   0   1   0   1   0  ...    1    1    1
99    1   0   0   1   1   1   0   0   0   0  ...    1    1    1
100   1   0   0   0   1   1   1   0   0   0  ...    1    1    0

      493  494  495  496  497  498  499
1       0    1    0    0    1    0    1
2       0    0    0    0    1    1    0
3       0    0    1    0    0    0    1
4       0    1    0    0    1    0    0
5       0    0    1    0    1    0    0
..    ...  ...  ...  ...  ...  ...  ...
96      0    0    0    0    0    0    0
97      0    1    1    0    1    0    0
98      0    1    0    0    0    0    1
99      0    0    1    0    1    1    0
100     1    1    0    0    1    1    0

[100 rows x 500 columns]
```

```python
num_of_tosses = 100
num_of_coins = 10
num_of_obs = 500

# number of heads and tails in each instance of 100 tosses
heads = df.sum().to_numpy()
tails = num_of_tosses - heads
# print(heads)
# print(tails)

np.random.seed(0)
# creating an array of 500 values with each one having value ranging from
1 to 10
selected_coin = np.random.randint(0, num_of_coins, size=num_of_obs)
print("Selected coin : \n" ,selected_coin)
print()
```

```
# frequency of coins selected from the box
_, freq_selected_coins = np.unique(selected_coin, return_counts=True)
print("Frequency of coins: ",freq_selected_coins)
print()
# maximum likelihood estimation
# Initialize MLE_vector
heads_of_selected_coin = np.zeros(num_of_coins)

# Update the count of heads for the selected coin
for i, j in zip(heads, selected_coin):
    heads_of_selected_coin[j] += i

p_heads = heads_of_selected_coin/(freq_selected_coins * 100)
print("Probability of getting heads(inital):")
for i, j in enumerate(p_heads):
    print(f"Coin {i+1} : {j:.4f}")


#A function compute_likelihood is defined to calculate the
#likelihood of a given observation (number of heads) given the number of
tosses and the estimated bias value.
def compute_likelihood(obs, n, pheads):
    likelihood = comb(n, obs,
exact=True)*(pheads**obs)*(1.0-pheads)**(n-obs)
    return likelihood
```

```
Selected coin :
 [5 0 3 3 7 9 3 5 2 4 7 6 8 8 1 6 7 7 8 1 5 9 8 9 4 3 0 3 5 0 2 3 8 1 3 3 3
 7 0 1 9 9 0 4 7 3 2 7 2 0 0 4 5 5 6 8 4 1 4 9 8 1 1 7 9 9 3 6 7 2 0 3 5 9
 4 4 6 4 4 3 4 4 8 4 3 7 5 5 0 1 5 9 3 0 5 0 1 2 4 2 0 3 2 0 7 5 9 0 2 7 2
 9 2 3 3 2 3 4 1 2 9 1 4 6 8 2 3 0 0 6 0 6 3 3 8 8 8 2 3 2 0 8 8 3 8 2 8 4
 3 0 4 3 6 9 8 0 8 5 9 0 9 6 5 3 1 8 0 4 9 6 5 7 8 8 9 2 8 6 6 9 1 6 8 8 3
 2 3 6 3 6 5 7 0 8 4 6 5 8 2 3 9 7 5 3 4 5 3 3 7 9 9 9 7 3 2 3 9 7 7 5 1 2
 2 8 1 5 8 4 0 2 5 5 0 8 1 1 0 3 8 8 4 4 0 9 3 7 3 2 1 1 2 1 4 2 5 5 5 2 5
 7 7 6 1 6 7 2 3 1 9 5 9 9 2 0 9 1 9 0 6 0 4 8 4 3 3 8 8 7 0 3 8 7 7 1 8 4
 7 0 4 9 0 6 4 2 4 6 3 3 7 8 5 0 8 5 4 7 4 1 3 3 9 2 5 2 3 5 7 2 7 1 6 5 0
 0 3 1 9 9 6 6 7 8 8 7 0 8 6 8 9 8 3 6 1 7 4 9 2 0 8 2 7 8 4 4 1 7 6 9 4 1
 5 9 7 1 3 5 7 3 6 6 7 9 1 9 6 0 3 8 4 1 4 5 0 3 1 4 4 4 0 0 8 4 6 9 3 3 2
 1 2 1 3 4 1 1 0 7 8 4 3 5 6 3 2 9 8 1 4 0 8 3 9 5 5 1 7 8 6 4 7 3 5 3 6 4
 7 3 0 5 9 3 7 5 5 8 0 8 3 6 9 3 2 7 0 3 0 3 6 1 9 2 9 4 9 1 3 2 4 9 7 4 9
 4 1 2 7 2 3 9 7 6 6 2 3 6 0 8 0 7 6 5]

Frequency of coins:  [50 43 45 71 52 44 41 49 54 51]

Probability of getting heads(inital):
Coin 1 : 0.5026
Coin 2 : 0.4707
Coin 3 : 0.4487
Coin 4 : 0.3813
Coin 5 : 0.4581
Coin 6 : 0.4898
Coin 7 : 0.4432
Coin 8 : 0.4349
Coin 9 : 0.5281
Coin 10 : 0.3833
```

```python
# Initialize random seed for reproducibility
np.random.seed(0)

EM_vector = np.zeros((num_of_tosses, num_of_coins))
# Initialize EM_vector with MLE estimates for the first iteration
EM_vector[0] = p_heads
print(EM_vector[0])

# Initialize variables for convergence criteria
improvement = float('inf')
iteration = 0
```

```
while improvement > 0.01:
    # E-step: Calculate the expected values of the number of heads and
tails for each coin based on the current MLE estimates.
    expectation = np.zeros((num_of_coins, num_of_obs, 2))

    # Iterate over each observation
    for obs in range(num_of_obs):
    # Compute the likelihood of each coin generating the current
observation
        likelihood = np.zeros(num_of_coins)
        for coin in range(num_of_coins):
            likelihood[coin] = compute_likelihood(heads[obs],
num_of_tosses, EM_vector[iteration][coin])

        # Compute the weights based on the likelihoods
        weights = likelihood / np.sum(likelihood)

        # Calculate the expected number of heads and tails for each coin
        for coin in range(num_of_coins):
            expectation[coin, obs, 0] = weights[coin] * heads[obs]
            expectation[coin, obs, 1] = weights[coin] * tails[obs]

    # M-step: Update the MLE estimates based on the expected values
    theta = np.zeros(num_of_coins)
    for coin in range(num_of_coins):
        # Estimate the probability of heads for each coin by dividing the
sum of expected heads by the sum of total expected outcomes
        theta[coin] = np.sum(expectation[coin, :, 0]) /
(np.sum(expectation[coin, :, 0]) + np.sum(expectation[coin, :, 1]))

    # Update EM_vector with the new MLE estimates
    EM_vector[iteration + 1] = theta
    # Calculate the improvement in MLE estimates for this iteration
    improvement = np.max(np.abs(EM_vector[iteration + 1] -
EM_vector[iteration]))

    # Increment iteration counter
    iteration += 1
```

```
    # Print the improvement and the updated MLE estimates for this
iteration
    print(f"Improvement in iteration {iteration}: {improvement}")
    print(EM_vector[iteration])
```

```
[0.5026     0.47069767 0.44866667 0.38126761 0.45807692 0.48977273
 0.44317073 0.43489796 0.52814815 0.38333333]
Improvement in iteration 1: 0.23860856413950138
[0.60897994 0.49311016 0.44434609 0.15758219 0.46424838 0.55081991
 0.43276627 0.41470214 0.76675671 0.17198189]
Improvement in iteration 2: 0.07417951775022445
[0.63564358 0.49743786 0.41955561 0.08340267 0.45503609 0.56678289
 0.39816477 0.36753685 0.82150997 0.15767246]
Improvement in iteration 3: 0.04064313691765714
[0.66731579 0.50396594 0.41048675 0.0443469  0.45995283 0.57669719
 0.37508937 0.32689371 0.84414716 0.1780762 ]
Improvement in iteration 4: 0.02516057298486518
[0.69247636 0.50852047 0.41027426 0.03454228 0.46571767 0.58313661
 0.36590042 0.30641643 0.8560499  0.16648171]
Improvement in iteration 5: 0.01987525757022157
[0.71235162 0.5125195  0.4104946  0.02635663 0.46975828 0.5909991
 0.35995338 0.28996373 0.86706257 0.1504688 ]
Improvement in iteration 6: 0.014544242550698416
[0.72689586 0.51605565 0.41021762 0.019608   0.47255645 0.6010668
 0.35591478 0.27565695 0.87685431 0.13732344]
Improvement in iteration 7: 0.012602185566432833
[0.73747109 0.51921444 0.40938569 0.01440537 0.47477499 0.61229069
 0.35296495 0.26305477 0.88349182 0.1274426 ]
Improvement in iteration 8: 0.011304899669348
[0.74551482 0.52225875 0.4081346  0.0120125  0.47688515 0.62330167
 0.35028898 0.25174987 0.88736708 0.12100168]
Improvement in iteration 9: 0.00996105395962743
[0.75189754 0.52542539 0.40662818 0.01122201 0.47902288 0.63326272
 0.34736875 0.24187619 0.88966218 0.11650509]
```

```
# Print the final MLE estimates after convergence
print(f"Final MLE estimates after {iteration} iterations:")

# Iterate over each coin's MLE estimate
```

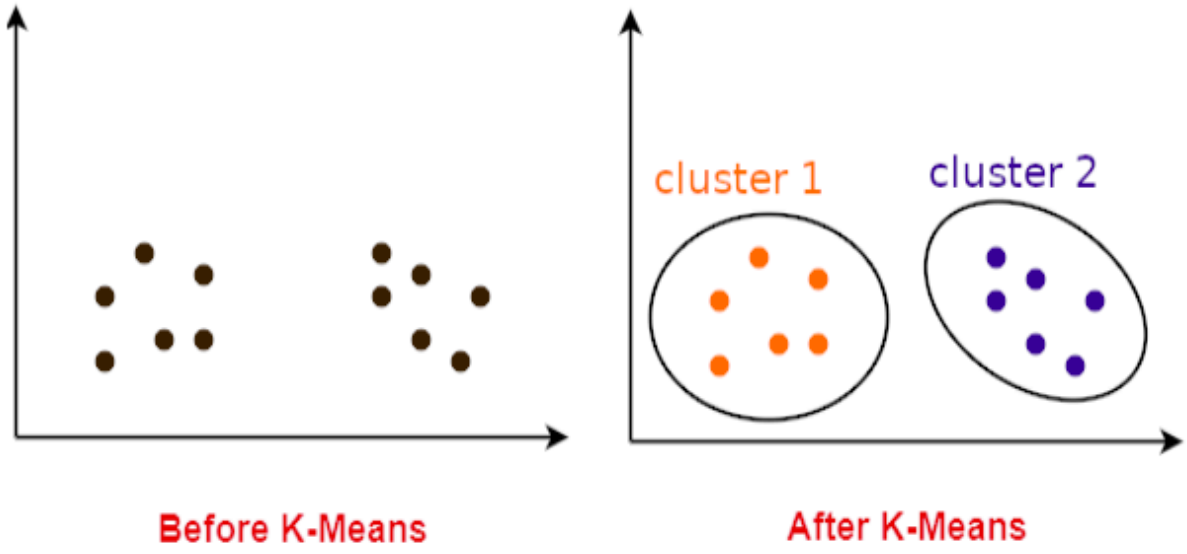
```python
for i, j in enumerate(EM_vector[iteration]):
    print(f"Coin {i+1} : {j:.6f}")
```

```
Final MLE estimates after 9 iterations:
Coin 1 : 0.751898
Coin 2 : 0.525425
Coin 3 : 0.406628
Coin 4 : 0.011222
Coin 5 : 0.479023
Coin 6 : 0.633263
Coin 7 : 0.347369
Coin 8 : 0.241876
Coin 9 : 0.889662
Coin 10 : 0.116505
```

Question 3:

C. A point set with real values is given in 2020_em_clustering.csv. Considering that there are two clusters, use EM to group together points belonging to the same cluster. Try and argue that k-means is an EM algorithm.

K MEANS CLUSTERING :

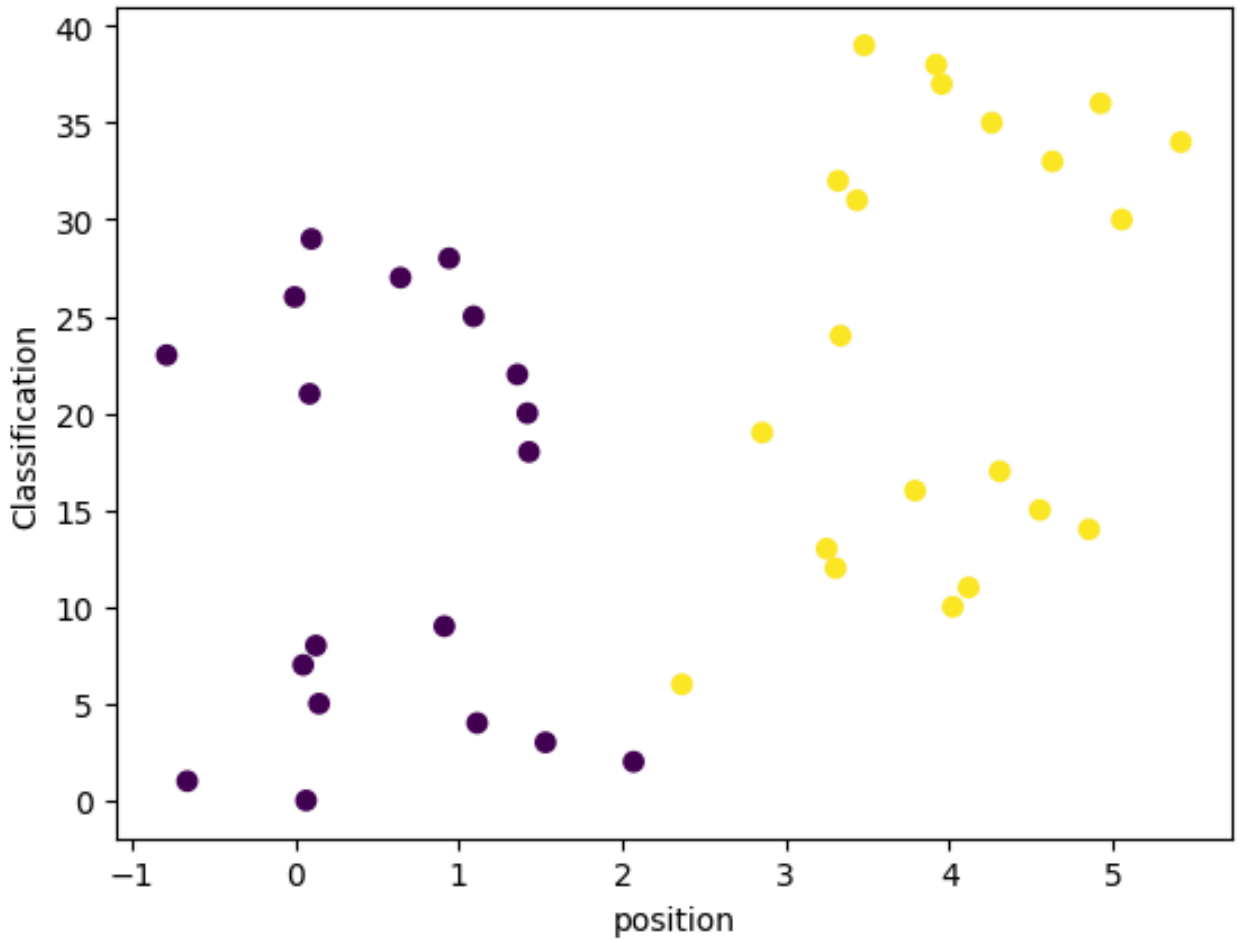


Code :

```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
```

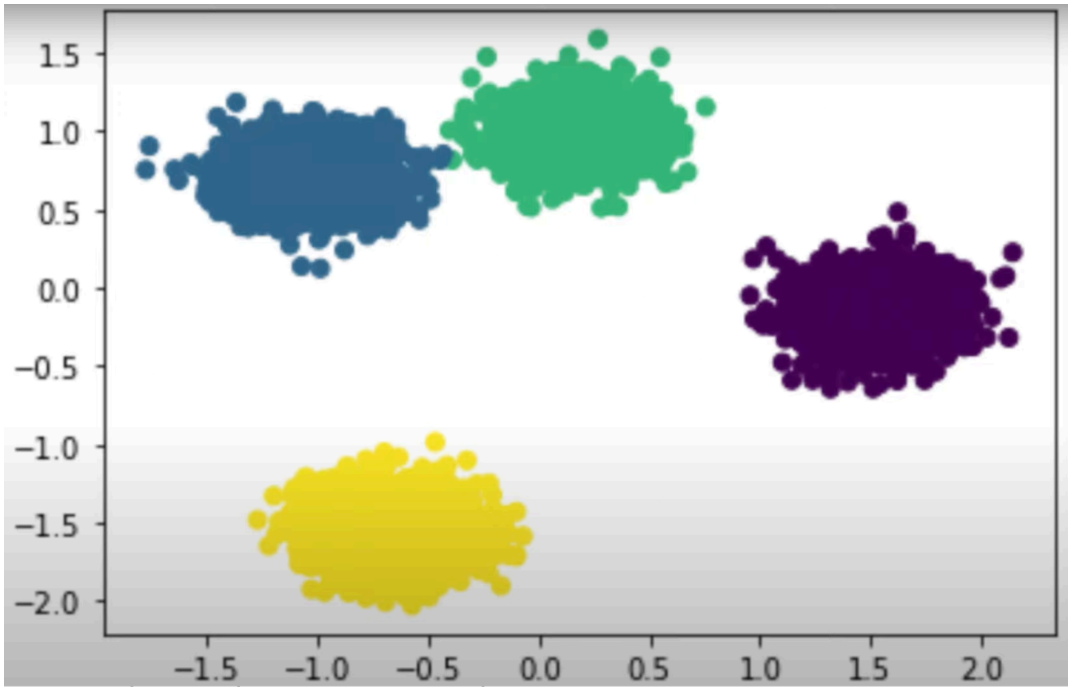
```
# Reading the data from csv file
df=pd.read_csv("/content/2020_em_clustering.csv", sep=',',
header=None)
df=df.transpose()

# Defining the number of clusters
kmeans=KMeans(n_clusters=2)

# Fitting the model
kmeans.fit(df)

# Predicting the clusters
predictions=kmeans.predict(df)

# Plotting the clusters
plt.scatter(df.iloc[:,0], [i for i in range(df.shape[0])],
c=predictions)
plt.xlabel("position")
plt.ylabel("Classification")
plt.show()
```

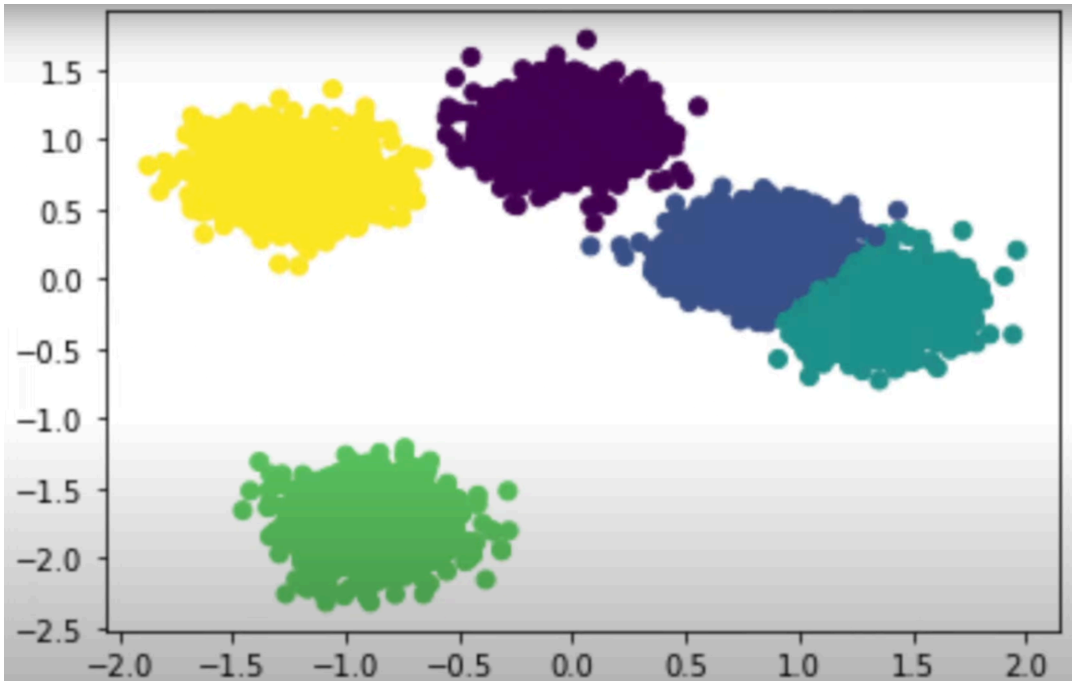
Output :



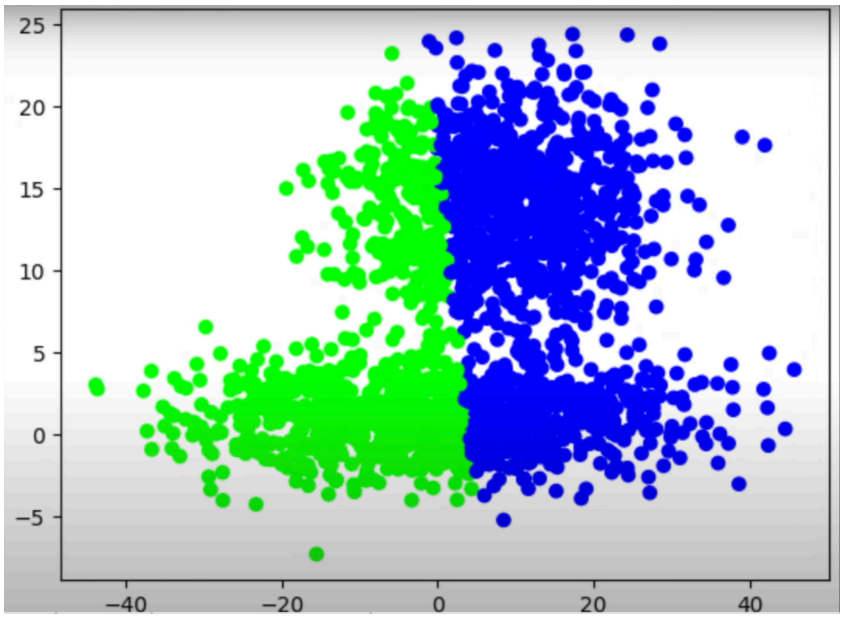
LIMITATIONS :

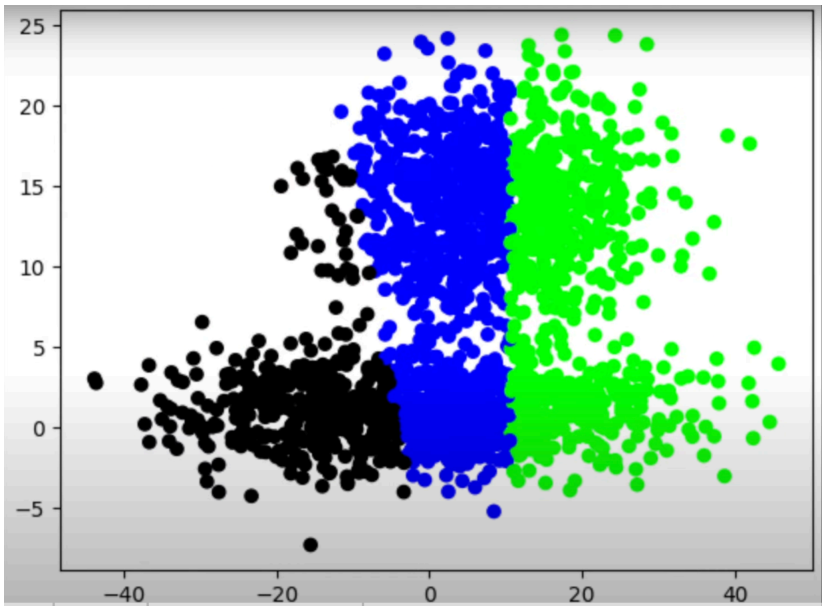
K means work great on these toy data sets





But when it comes to real world problems it cannot handle **noise** or **overlapping clusters** :





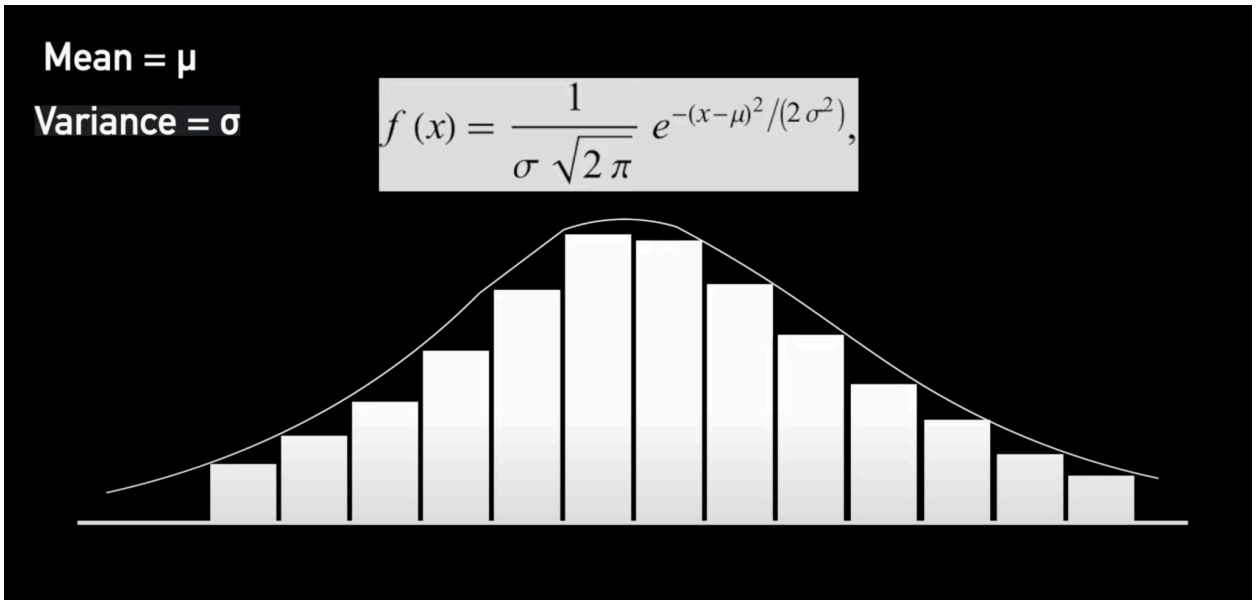
Gaussian Mixture Model :

Gaussian distribution or The Bell Curve :

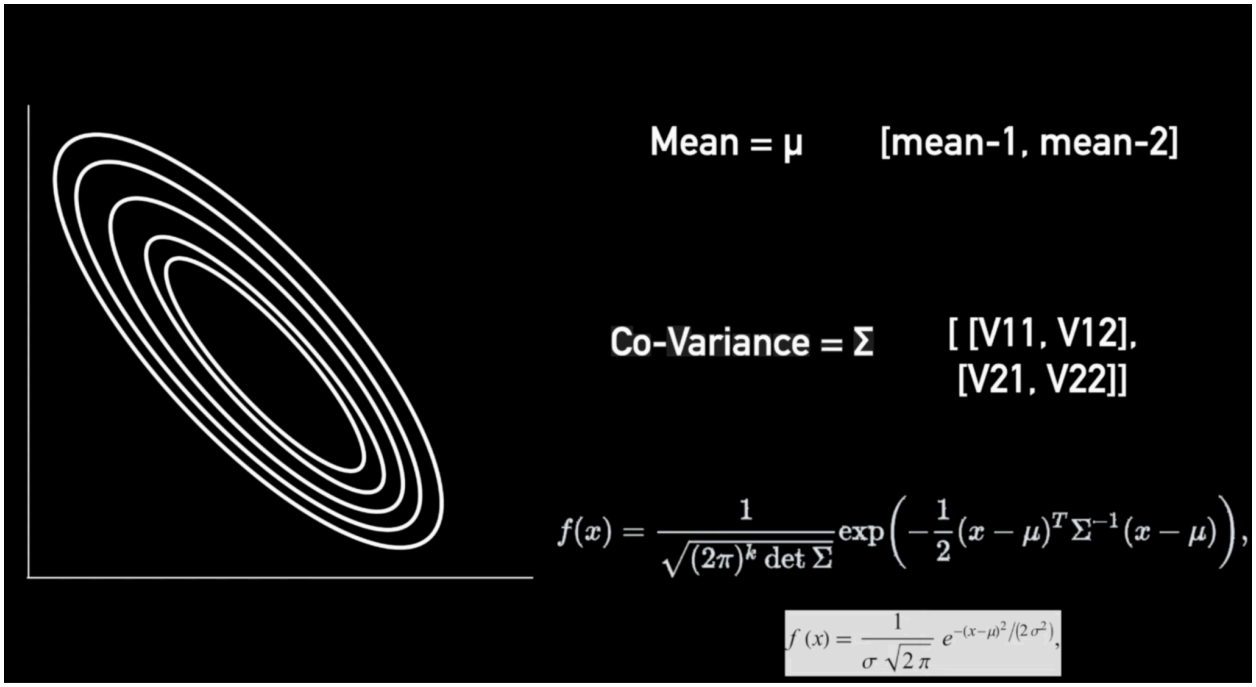
To solve this problem we need a more robust algorithm and one of those is the Gaussian mixture model. To understand this we first need to understand the gaussian distribution or the bell curve. To describe a bell curve we need 2 things :-

1.Mean (around which the distribution is centered)

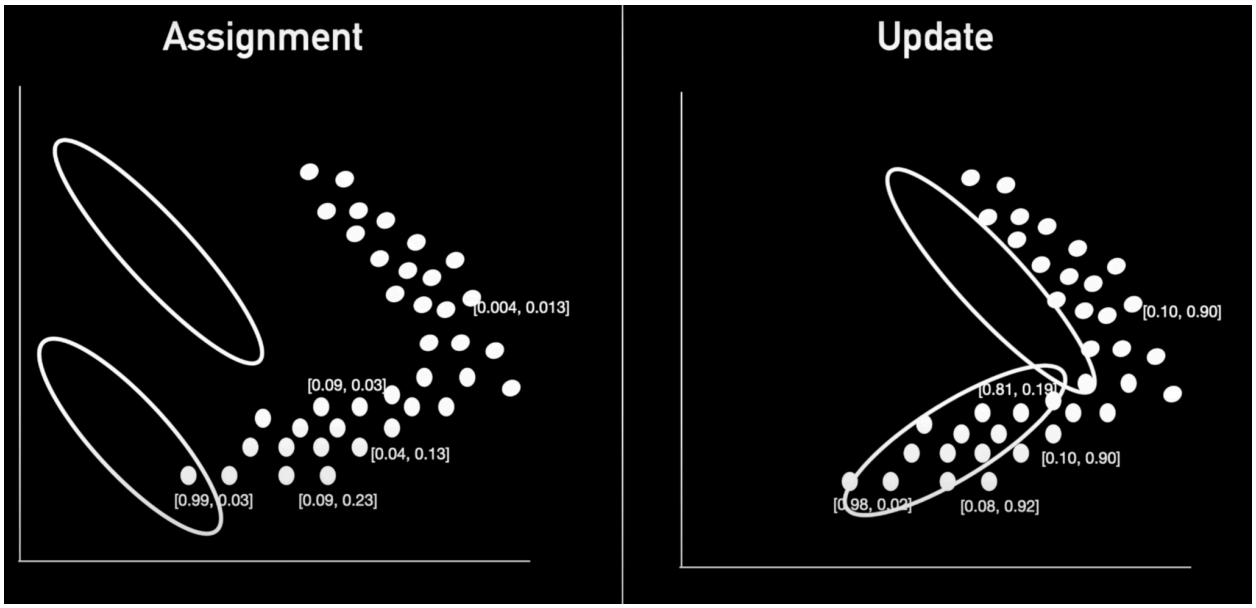
2.Variance (which describes how sharp or flat the bell curve is)



We can also have a bell curve in more than one dimension. [2-D]



Just like any other clustering algorithm GMM is iterative and has 2 steps :



Code :

```
# Reading the data from csv file
df=pd.read_csv("/content/2020_em_clustering.csv", sep=',',
header=None)
df=df.transpose()

# Defining the number of clusters
em=GaussianMixture(n_components=2)

# Fitting the model,
"""
Fitting the model: The fit method is used to fit the EM
algorithm to the data.
This step involves iteratively estimating the parameters of
the Gaussian distribution for each cluster until convergence.
"""

em.fit(df)

"""
Predicting the clusters: The predict method is used to make
predictions on which cluster each data point belongs to.
"""
predictions=em.predict(df)

# Plotting the clusters
plt.scatter(df.iloc[:,0], [i for i in range(df.shape[0])],
c=predictions)
plt.xlabel("position")
plt.ylabel("Classification")
plt.show()
```

OUTPUT :