Experiment No. 5
Implementation of Expectation-Maximization Algorithm
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Implementation of Expectation-Maximization Algorithm.

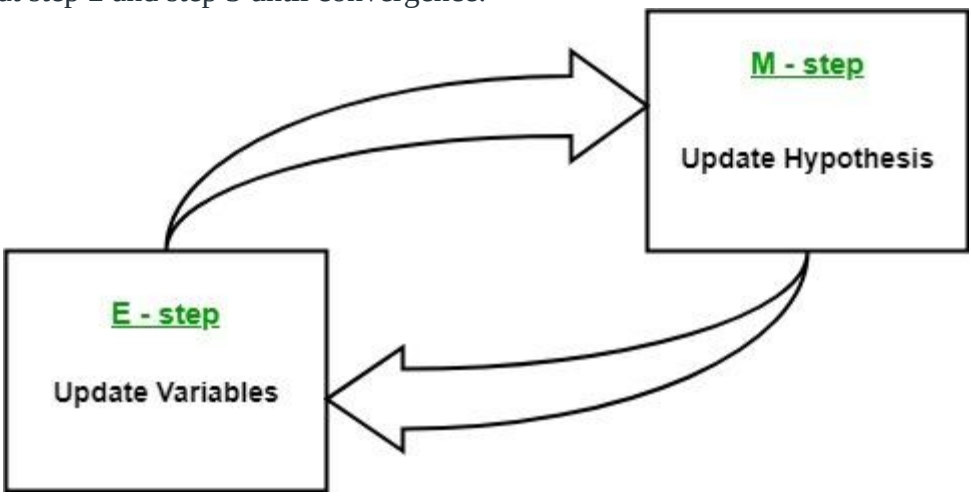
Objective: Able to interpret the how Expectation Maximization Algorithm is applied in unsupervised clustering algorithm to handle the non observable features.

Theory:

In the real-world applications of machine learning, it is very common that there are many relevant features available for learning but only a small subset of them are observable. So, for the variables which are sometimes observable and sometimes not, then we can use the instances when that variable is visible is observed for the purpose of learning and then predict its value in the instances when it is not observable.

On the other hand, ***Expectation-Maximization algorithm*** can be used for the latent variables (variables that are not directly observable and are actually inferred from the values of the other observed variables) too in order to predict their values with the condition that the general form of probability distribution governing those latent variables is known to us. This algorithm is actually at the base of many unsupervised clustering algorithms in the field of machine learning.
It was explained, proposed and given its name in a paper published in 1977 by Arthur Dempster, Nan Laird, and Donald Rubin. It is used to find the *local maximum likelihood parameters* of a statistical model in the cases where latent variables are involved and the data is missing or incomplete.

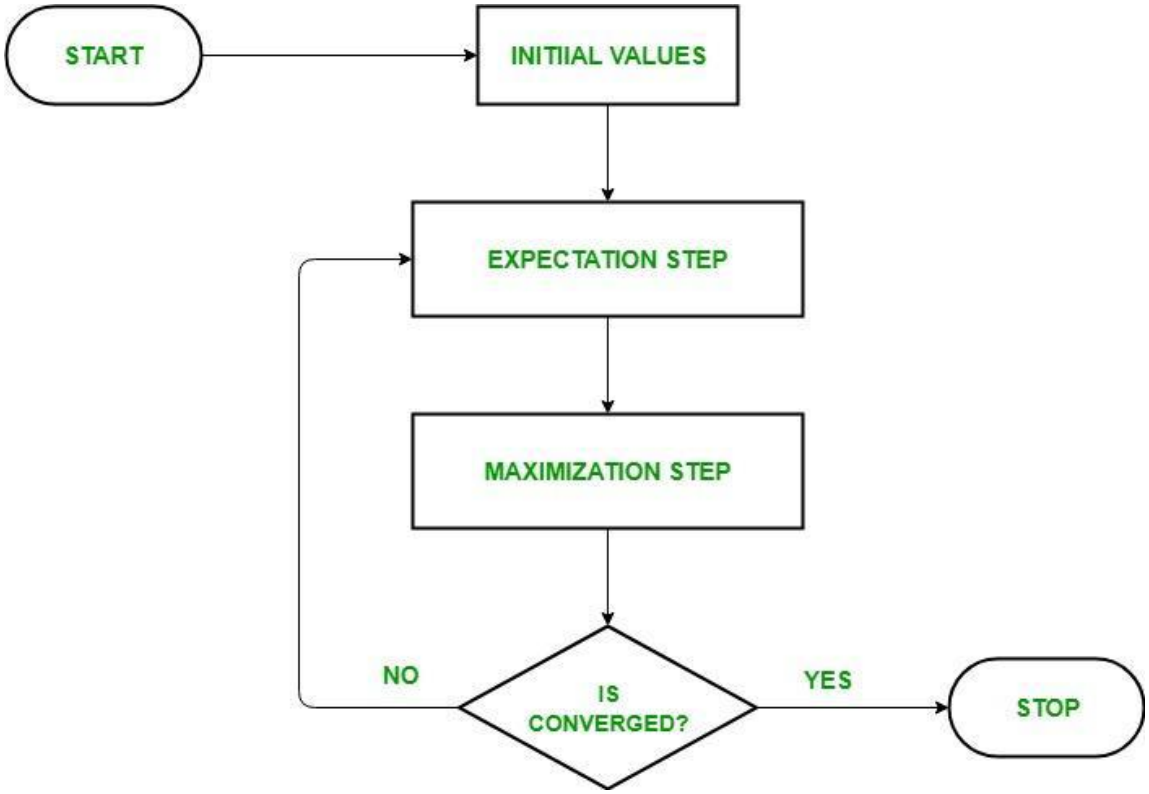
Algorithm:
1. Given a set of incomplete data, consider a set of starting parameters.
2. **Expectation step (E – step):** Using the observed available data of the dataset, estimate (guess) the values of the missing data.
3. **Maximization step (M – step):** Complete data generated after the expectation (E) step is used in order to update the parameters.
4. Repeat step 2 and step 3 until convergence.



The essence of Expectation-Maximization algorithm is to use the available observed data of the dataset to estimate the missing data and then using that data to update the values of the parameters.

- Initially, a set of initial values of the parameters are considered. A set of incomplete observed data is given to the system with the assumption that the observed data comes from a specific model.
- The next step is known as "Expectation" – step or *E-step*. In this step, we use the observed data in order to estimate or guess the values of the missing or incomplete data. It is basically used to update the variables.
- The next step is known as "Maximization"-step or *M-step*. In this step, we use the complete data generated in the preceding "Expectation" – step in order to update the values of the parameters. It is basically used to update the hypothesis.
- Now, in the fourth step, it is checked whether the values are converging or not, if yes, then stop otherwise repeat *step-2* and *step-3* i.e. "Expectation" – step and "Maximization" – step until the convergence occurs.

Flow chart for EM algorithm –



Usage of EM algorithm –

- It can be used to fill the missing data in a sample.
- It can be used as the basis of unsupervised learning of clusters.
- It can be used for the purpose of estimating the parameters of Hidden Markov Model (HMM).
- It can be used for discovering the values of latent variables.

Advantages of EM algorithm –

- It is always guaranteed that likelihood will increase with each iteration.
- The E-step and M-step are often pretty easy for many problems in terms of implementation.
- Solutions to the M-steps often exist in the closed form.

Disadvantages of EM algorithm –

- It has slow convergence.
- It makes convergence to the local optima only.
- It requires both the probabilities, forward and backward (numerical optimization requires only forward probability).

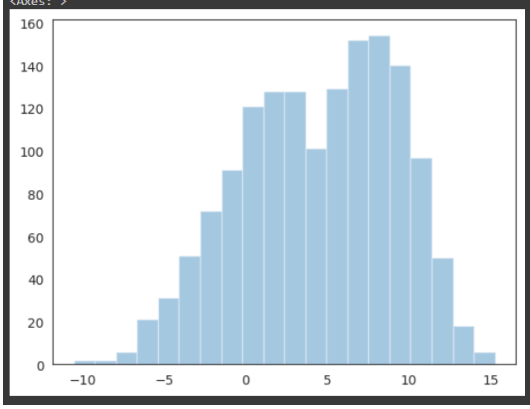
Implementation:

```
import matplotlib.pyplot as plt

import seaborn as sns

sns.set_style("white")

%matplotlib inline

#for matrix math

import numpy as np

#for normalization + probability density function computation

from scipy import stats

#for data preprocessing

import pandas as pd

from math import sqrt, log, exp, pi

from random import uniform

print("import done")
```

```
import done
```

```
random_seed=36788765
np.random.seed(random_seed)

Mean1 = 2.0  # Input parameter, mean of first normal probability distribution
Standard_dev1 = 4.0 #@param {type:"number"}
Mean2 = 9.0 # Input parameter, mean of second normal  probability distribution
Standard_dev2 = 2.0 #@param {type:"number"}

# generate data
y1 = np.random.normal(Mean1, Standard_dev1, 1000)
y2 = np.random.normal(Mean2, Standard_dev2, 500)
data=np.append(y1,y2)

# For data visiualisation calculate left and right of the graph
Min_graph = min(data)
Max_graph = max(data)
x = np.linspace(Min_graph, Max_graph, 2000) # to plot the data

print('Input Gaussian {:}: μ = {:.2}, σ = {:.2}'.format("1", Mean1, Standard_dev1))
print('Input Gaussian {:}: μ = {:.2}, σ = {:.2}'.format("2", Mean2, Standard_dev2))
sns.distplot(data, bins=20, kde=False)
```

```
Input Gaussian 1: μ = 2.0, σ = 4.0
Input Gaussian 2: μ = 9.0, σ = 2.0
<ipython-input-5-8d5c25c80112>:21: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(data, bins=20, kde=False)
<Axes: >
```

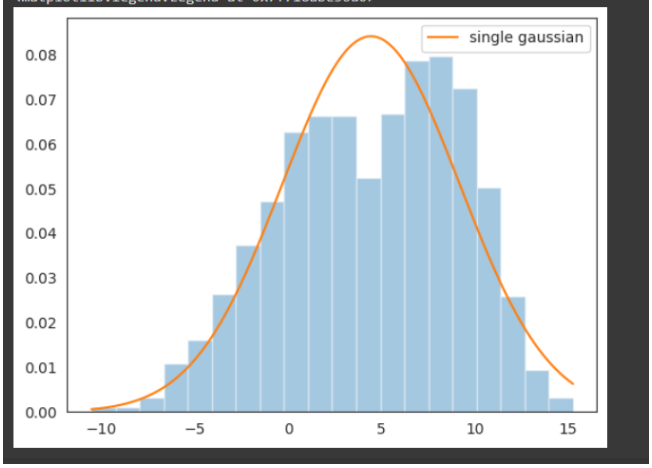



```
class Gaussian:
    "Model univariate Gaussian"
    def __init__(self, mu, sigma):
        #mean and standard deviation
        self.mu = mu
        self.sigma = sigma


    #probability density function
    def pdf(self, datum):
        "Probability of a data point given the current parameters"
        u = (datum - self.mu) / abs(self.sigma)
        y = (1 / (sqrt(2 * pi) * abs(self.sigma))) * exp(-u * u / 2)
        return y


    def __repr__(self):
        return 'Gaussian({0:4.6}, {1:4.6})'.format(self.mu, self.sigma)
print("done")
```

```
done
```

```
best_single = Gaussian(np.mean(data), np.std(data))
print('Best single Gaussian: μ = {:.2}, σ = {:.2}'.format(best_single.mu, best_single.sigma))
#fit a single gaussian curve to the data
g_single = stats.norm(best_single.mu, best_single.sigma).pdf(x)
sns.distplot(data, bins=20, kde=False, norm_hist=True)
plt.plot(x, g_single, label='single gaussian')
plt.legend()
```

```
Best single Gaussian: μ = 4.4, σ = 4.8
<ipython-input-7-0e8931e2d352>:5: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(data, bins=20, kde=False, norm_hist=True)
<matplotlib.legend.Legend at 0x7f718abe36d0>
```




```
class GaussianMixture_self:
    "Model mixture of two univariate Gaussians and their EM estimation"

    def __init__(self, data, mu_min=min(data), mu_max=max(data), sigma_min=1,
sigma_max=1, mix=.5):
        self.data = data
        #todo the Algorithm would be numerical enhanced by normalizing the data first, next do
all the EM steps and do the de-normalising at the end

        #init with multiple gaussians
        self.one = Gaussian(uniform(mu_min, mu_max),
```

```
                uniform(sigma_min, sigma_max))
        self.two = Gaussian(uniform(mu_min, mu_max),
                uniform(sigma_min, sigma_max))

        #as well as how much to mix them
        self.mix = mix

    def Estep(self):
        "Perform an E(stimation)-step, assign each point to gaussian 1 or 2 with a percentage"
        # compute weights
        self.loglike = 0. # = log(p = 1)
        for datum in self.data:
            # unnormalized weights
            wp1 = self.one.pdf(datum) * self.mix
            wp2 = self.two.pdf(datum) * (1. - self.mix)
            # compute denominator
            den = wp1 + wp2
            # normalize
            wp1 /= den
            wp2 /= den     # wp1+wp2= 1, it either belongs to gaussian 1 or gaussion 2
            # add into loglike
            self.loglike += log(den) #freshening up self.loglike in the process
            # yield weight tuple
            yield (wp1, wp2)

    def Mstep(self, weights):
        "Perform an M(aximization)-step"
        # compute denominators
        (left, rigt) = zip(*weights)
        one_den = sum(left)
        two_den = sum(rigt)
```

```
        # compute new means
        self.one.mu = sum(w * d  for (w, d) in zip(left, data)) / one_den
        self.two.mu = sum(w * d  for (w, d) in zip(rigt, data)) / two_den

        # compute new sigmas
        self.one.sigma = sqrt(sum(w * ((d - self.one.mu) ** 2)
                                  for (w, d) in zip(left, data)) / one_den)
        self.two.sigma = sqrt(sum(w * ((d - self.two.mu) ** 2)
                                  for (w, d) in zip(rigt, data)) / two_den)
        # compute new mix
        self.mix = one_den / len(data)

    def iterate(self, N=1, verbose=False):
        "Perform N iterations, then compute log-likelihood"
        for i in range(1, N+1):
            self.Mstep(self.Estep()) #The heart of the algorith, perform E-stepand next M-step
            if verbose:
                print('{0:2} {1}'.format(i, self))
        self.Estep() # to freshen up self.loglike

    def pdf(self, x):
        return (self.mix)*self.one.pdf(x) + (1-self.mix)*self.two.pdf(x)

    def __repr__(self):
        return 'GaussianMixture({0}, {1}, mix={2.03})'.format(self.one,
                                                              self.two,
                                                              self.mix)

    def __str__(self):
        return 'Mixture: {0}, {1}, mix={2:.03})'.format(self.one,
                                                        self.two,
                                                        self.mix)
```

```
print("done")
```

```
done
```

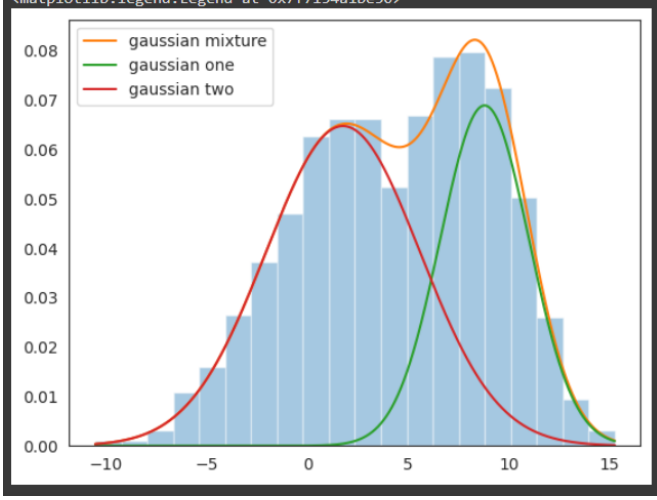
```
n_iterations = 20
best_mix = None
best_loglike = float('-inf')
mix = GaussianMixture_self(data)
for _ in range(n_iterations):
    try:
        #train!
        mix.iterate(verbose=True)
        if mix.loglike > best_loglike:
            best_loglike = mix.loglike
            best_mix = mix

    except (ZeroDivisionError, ValueError, RuntimeWarning): # Catch division errors from bad starts, and just throw them out...
        print("one less")
        pass
```

```
1 Mixture: Gaussian(-5.81052, 1.27375), Gaussian(4.83277, 4.39033), mix=0.036)
1 Mixture: Gaussian(-5.30703, 1.23684), Gaussian(4.73186, 4.51064), mix=0.0281)
1 Mixture: Gaussian(-5.04445, 1.20674), Gaussian(4.7003, 4.54811), mix=0.0257)
1 Mixture: Gaussian(-4.88152, 1.20146), Gaussian(4.68792, 4.56395), mix=0.0249)
1 Mixture: Gaussian(-4.76296, 1.21164), Gaussian(4.68347, 4.57134), mix=0.0247)
1 Mixture: Gaussian(-4.66629, 1.23092), Gaussian(4.6833, 4.57443), mix=0.025)
1 Mixture: Gaussian(-4.58114, 1.25563), Gaussian(4.68588, 4.57496), mix=0.0255)
1 Mixture: Gaussian(-4.50225, 1.28362), Gaussian(4.69047, 4.57376), mix=0.0262)
1 Mixture: Gaussian(-4.42685, 1.31357), Gaussian(4.69664, 4.5713), mix=0.0271)
1 Mixture: Gaussian(-4.35343, 1.3447), Gaussian(4.70413, 4.56787), mix=0.0281)
1 Mixture: Gaussian(-4.28121, 1.3765), Gaussian(4.71277, 4.56364), mix=0.0292)
1 Mixture: Gaussian(-4.20982, 1.40866), Gaussian(4.72243, 4.55874), mix=0.0305)
1 Mixture: Gaussian(-4.13909, 1.44099), Gaussian(4.73304, 4.55326), mix=0.0319)
1 Mixture: Gaussian(-4.06903, 1.47337), Gaussian(4.74451, 4.54725), mix=0.0334)
1 Mixture: Gaussian(-3.99968, 1.50573), Gaussian(4.75682, 4.54077), mix=0.0351)
1 Mixture: Gaussian(-3.93113, 1.53806), Gaussian(4.7699, 4.53384), mix=0.0368)
1 Mixture: Gaussian(-3.86347, 1.57032), Gaussian(4.78374, 4.5265), mix=0.0386)
1 Mixture: Gaussian(-3.79679, 1.60253), Gaussian(4.79829, 4.51876), mix=0.0405)
1 Mixture: Gaussian(-3.73116, 1.63468), Gaussian(4.81353, 4.51064), mix=0.0426)
1 Mixture: Gaussian(-3.66663, 1.6668), Gaussian(4.82946, 4.50216), mix=0.0447)
```

```
n_iterations = 300
n_random_restarts = 4
best_mix = None
best_loglike = float('-inf')
print('Computing best model with random restarts...\n')
for _ in range(n_random_restarts):
    mix = GaussianMixture_self(data)
    for _ in range(n_iterations):
        try:
            mix.iterate()
            if mix.loglike > best_loglike:
                best_loglike = mix.loglike
                best_mix = mix
        except (ZeroDivisionError, ValueError, RuntimeWarning): # Catch division errors from bad starts, and just throw them out...
            pass
#print('Best Gaussian Mixture : μ = {:.2}, σ = {:.2} with μ = {:.2}, σ = {:.2}'.format(best_mix.one.mu, best_mix.one.sigma, best_mix.two.mu, best_mix.two.sigma))

print('Input Gaussian {:}: μ = {:.2}, σ = {:.2}'.format("1", Mean1, Standard_dev1))
print('Input Gaussian {:}: μ = {:.2}, σ = {:.2}'.format("2", Mean2, Standard_dev2))
print('Gaussian {:}: μ = {:.2}, σ = {:.2}, weight = {:.2}'.format("1", best_mix.one.mu, best_mix.one.sigma, best_mix.mix))
print('Gaussian {:}: μ = {:.2}, σ = {:.2}, weight = {:.2}'.format("2", best_mix.two.mu, best_mix.two.sigma, (1-best_mix.mix)))
#Show mixture
sns.distplot(data, bins=20, kde=False, norm_hist=True)
g_both = [best_mix.pdf(e) for e in x]
plt.plot(x, g_both, label='gaussian mixture')
g_left = [best_mix.one.pdf(e) * best_mix.mix for e in x]
plt.plot(x, g_left, label='gaussian one')
g_right = [best_mix.two.pdf(e) * (1-best_mix.mix) for e in x]
```

```
plt.plot(x, g_right, label='gaussian two')
plt.legend()
```

```
Computing best model with random restarts...

Input Gaussian 1: μ = 2.0, σ = 4.0
Input Gaussian 2: μ = 9.0, σ = 2.0
Gaussian 1: μ = 8.8, σ = 2.2, weight = 0.38
Gaussian 2: μ = 1.8, σ = 3.8, weight = 0.62
<ipython-input-12-8906b55554ce>:23: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(data, bins=20, kde=False, norm_hist=True)
<matplotlib.legend.Legend at 0x7f7154a1be50>
```




```
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components = 2, tol=0.000001)
gmm.fit(np.expand_dims(data, 1)) # Parameters: array-like, shape (n_samples, n_features), 1
dimension dataset so 1 feature
Gaussian_nr = 1
print('Input Gaussian {:}: μ = {:.2}, σ = {:.2}'.format("1", Mean1, Standard_dev1))
print('Input Gaussian {:}: μ = {:.2}, σ = {:.2}'.format("2", Mean2, Standard_dev2))
for mu, sd, p in zip(gmm.means_.flatten(), np.sqrt(gmm.covariances_.flatten()),
gmm.weights_):
    print('Gaussian {:}: μ = {:.2}, σ = {:.2}, weight = {:.2}'.format(Gaussian_nr, mu, sd, p))
    g_s = stats.norm(mu, sd).pdf(x) * p
    plt.plot(x, g_s, label='gaussian sklearn')
    Gaussian_nr += 1
```

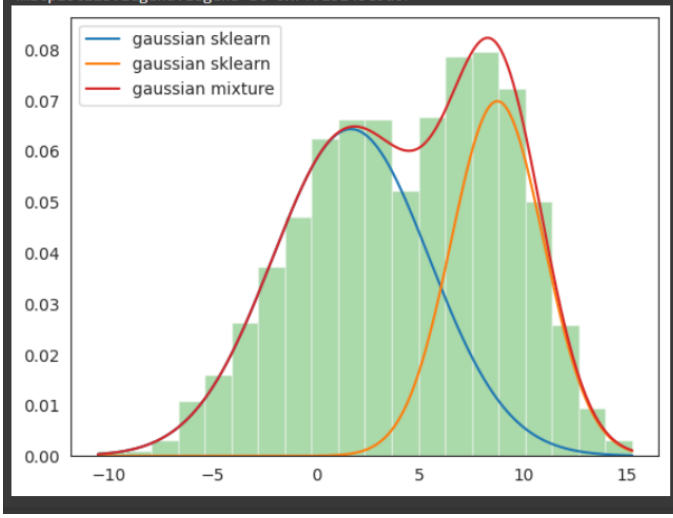
sns.distplot(data, bins=20, kde=False, norm_hist=True)

gmm_sum = np.exp([gmm.score_samples(e.reshape(-1, 1)) for e in x]) #gmm gives log probability, hence the exp() function

plt.plot(x, gmm_sum, label='gaussian mixture')

plt.legend()

```
Input Gaussian 1: μ = 2.0, σ = 4.0
Input Gaussian 2: μ = 9.0, σ = 2.0
Gaussian 1: μ = 1.7, σ = 3.8, weight = 0.61
Gaussian 2: μ = 8.8, σ = 2.2, weight = 0.39
<ipython-input-13-b6a32aee3662>:12: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(data, bins=20, kde=False, norm_hist=True)
<matplotlib.legend.Legend at 0x7f715143c9d0>
```



Conclusion:

1) What is the use of Expectation-Maximization Algorithm?

   The Expectation-Maximization (EM) algorithm is a powerful statistical method used for estimating parameters in probabilistic models when dealing with incomplete or missing data. It iteratively alternates between two main steps: the E-step (Expectation), where it calculates the expected values of latent variables given the observed data and current parameter estimates, and the M-step (Maximization), where it updates the parameters based on the expected values obtained in the E-step. EM is particularly useful in scenarios where data is partially observed or latent variables are involved, such as in clustering algorithms like Gaussian Mixture Models (GMM) or in fitting models with hidden variables like Hidden Markov Models (HMM). Its

versatility and ability to handle missing data make it invaluable in various fields, including machine learning, bioinformatics, and natural language processing.

2) How it can be used to handle missing data?

The Expectation-Maximization (EM) algorithm is particularly useful for handling missing data because of its ability to iteratively estimate parameters in probabilistic models while accommodating incomplete or partially observed data. Here's how it can be used to handle missing data:

1)E-step (Expectation): In the E-step of the EM algorithm, the missing data points are treated as latent variables. The algorithm calculates the expected values of these missing data points based on the observed data and the current estimates of the model parameters.

2)M-step (Maximization): In the M-step, the algorithm updates the parameters of the model based on the expected values obtained in the E-step, along with the observed data. This step involves maximizing the likelihood function, taking into account both the observed and imputed values.

3)Iterative Process: The EM algorithm iteratively alternates between the E-step and M-step until convergence, where the parameters stabilize. During each iteration, the algorithm refines its estimates of both the model parameters and the missing data values.

4)Imputation of Missing Values: Through this iterative process, the EM algorithm effectively imputes missing values by estimating their most likely values based on the observed data and the model parameters. These imputed values can then be used for subsequent analyses or modeling tasks.