I- Description of the problem

According to the task n°1, we have implemented ourselves **the GMM algorithm** for the example one.
By definition, the inputs of the example 1 are also the ones of our assignment.
So, based on the course, we have to implement the Expectation-Maximization algorithm that can be divided into 2 steps:

1) E step
2) M step

to replace the Gaussian Mixture function. To do that we have to analyze the mathematical function of the E step.
Despite the fact that the GMM algorithm allows to resolve some clustering problems, the first and most important objective of this lab was to understand the mathematical background of the function. Thus, the problem was to study the mathematical behavior of the algorithm before implementing it with python. This means that all the different parameters that we will describe below must be perfectly understood.
So, as the first step of the E-M algorithm, we have the **E-step**:

- **E-step** (guess values of $z^{(i)}$):

$$\begin{aligned} w_j^{(i)} &= P\left(z^{(i)} = j \mid x^{(i)}; \phi, \mu, \Sigma\right) \\ &= \frac{P\left(x^{(i)} \mid z^{(i)} = j\right) P\left(z^{(i)} = j\right)}{\sum_{l=1}^{k} P\left(x^{(i)} \mid z^{(i)} = l\right) P\left(z^{(i)} = l\right)} \\ &= \frac{\frac{1}{(2\pi)^{n/2}|\Sigma_j|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu_j)^T \Sigma_j^{-1} (x^{(i)} - \mu_j)\right)\phi_j}{\sum_{l=1}^{k} \frac{1}{(2\pi)^{n/2}|\Sigma_l|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu_l)^T \Sigma_l^{-1} (x^{(i)} - \mu_l)\right)\phi_l}. \end{aligned} \tag{6}$$

$|\Sigma_j|$ the determinant of the covariance of the cluster j

μ_j The centroid of a cluster j

Σ_j^{-1} The inverse of the matrix of covariance

ϕ_j The probability that the point is on the j cluster

$|\Sigma_l|$ The determinant of the covariance as for j but for the different cluster and not a defined cluster as for the determinant for j

Then, we have the second that represented by M-Step (that also shares the parameters of E-step)

$$\phi_j = \frac{1}{m}\sum_{i=1}^{m} w_j^{(i)},$$

$$\mu_j = \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}},$$

$$\Sigma_j = \frac{\sum_{i=1}^{m} w_j^{(i)} \left(x^{(i)} - \mu_j\right)\left(x^{(i)} - \mu_j\right)^T}{\sum_{i=1}^{m} w_j^{(i)}}.$$

II- Method

1) Initialization:

```
# generate random sample, two components
np.random.seed(1)

# generate spherical data centered on (20, 20)
shifted_gaussian = np.random.randn(n_samples, dimension_de_x) + np.array([20, 20])

# generate zero centered stretched Gaussian data
C = np.array([[0., -0.7], [3.5, .7]])
stretched_gaussian = np.dot(np.random.randn(n_samples, dimension_de_x), C)

#probablities
w = np.random.randn(600, 2)

# concatenate the two datasets into the final training set
X_train = np.vstack([shifted_gaussian, stretched_gaussian])

sigma = np.array([np.random.rand(k,dimension_de_x), np.random.rand(k,dimension_de_x)])
mu = np.array([np.random.rand(1,dimension_de_x), np.random.rand(1,dimension_de_x)])
phi = np.matrix([[ 0.2, 0.8]])
```

Here we take the beginning of the example1's code (we only fix the seed at 1). This step corresponds to the data pre-processing, that will give us the input of our model, X_train.
Then we generate sigma and µ **randomly**.

- ∑ is a **2*2 matrix** because we have a 2 dimensions model. It represents the **covariance**
- µ is a the **centroïd** of each cluster. It is a **(2,1) matrix** where each element is a centroid so here a (1,2) matrix.
- For ϕ, we generate a **(1,2) matrix** that the sum will be 1. It is the **probability** for the different point to be in a cluster.

- W is generated **randomly** but is not used. The matrix will be updated at the first E-step with real data.

2) E-step:

```
#numerateur
determinant = np.absolute(np.linalg.det(sigma[j]))
dataligne = np.matrix(np.array([data[i,]])) - np.matrix(np.array([mu[j,]]))
transpose = dataligne.transpose()

num = 1/((2*math.pi)*math.sqrt(determinant))
num = num*math.exp((-1/2)*((dataligne)*np.linalg.inv(sigma[j])*transpose))*phi[0,j]
```

During the E-step, we actually guess the values of z(i), that is our hidden data. This particularity exists because our input is actually unabled dataset. In other words, we are in unsupervised learning.
So, in order to that, we determine the value of W that represents the probability that the datapoint belongs to either the cluster number 1 or 2.
As you can see above, we generate the **numerator** thanks to the formula given in the course.
First, we need to find the **determinant** of sigma for the cluster that we got from the parameter j (we used the **absolute function** to be sure that the determinant is **positive**).
Then the variable dataligne represent **the distance to the centroid** for the line i of the dataset given by parameters.
Finally, we generate the transpose matrix of the dataligne variable.

Then we calculate the denominator of the Wj.

```
#denominateur
determinant = np.absolute(np.linalg.det(sigma[0]))
dataligne = np.matrix(np.array([data[i,]])) - np.matrix(np.array([mu[0,]]))
transpose = dataligne.transpose()

den1 = (1/((2*math.pi)*math.sqrt(determinant)))
den1 = den1*math.exp((-1/2)*((dataligne)*np.linalg.inv(sigma[0])*transpose))*phi[0,0]

determinant = np.absolute(np.linalg.det(sigma[1]))
dataligne = np.matrix(np.array([data[i,]])) - np.matrix(np.array([mu[1,]]))
transpose = dataligne.transpose()

den2 = (1/((2*math.pi)*math.sqrt(determinant)))
den2 = den2*math.exp((-1/2)*((dataligne)*np.linalg.inv(sigma[1])*transpose))*phi[0,1]

den=den1+den2
```

Once again, we generate the denominator thanks to the formula of the course.
As for the numerator, we get the determinant, the distance to the centroid and his transpose.
With that we can calculate the denominator by sum of the different denominator for each cluster.

Then we return the result of the operation. It will be used to fill the W matrix that represents **the probability to be in each cluster for each line of data**.

```
return num/den
```

$\longrightarrow$

```
test=Expectation(X_train,sigma,mu,phi,i,j)
w[i,j]=test
```

3) *M-step:*

We created a function for the 3 argument that we maximize

```python
def Maximisation_phi (w, phi):

    sommekluster1 = 0
    sommekluster2 = 0

    for i in range (600):

        #print("la valeur en i ",i,"et j 1 est ",w[i,0])
        sommekluster1 = sommekluster1 + w[i,0]

        #print("la valeur en i ",i,"et j 2 est ",w[i,1])
        sommekluster2 = sommekluster2 + w[i,1]

    #print("Cluster 1 ",sommekluster1/600," Cluster 2 ",sommekluster2/600)

    phi[0,0]=sommekluster1/600
    phi[0,1]=sommekluster2/600

    return phi
```

- For ϕ:

We sum the probability of each cluster and we divide by the number of sample (in our case 600)
It finally gives the **new probability to be set on a specific cluster**.

- For µ:

```python
def Maximisation_mu (w,data,mu):

    sommekluster1 = 0
    sommekluster2 = 0

    sommeproba1 = 0
    sommeproba2 = 0

    for i in range (600):

        #print(dataligne)
        sommekluster1=sommekluster1+w[i,0]*data[i]
        sommekluster2=sommekluster2+w[i,1]*data[i]

        sommeproba1=sommeproba1+w[i,0]
        sommeproba2=sommeproba2+w[i,1]

    mu[0]=sommekluster1/sommeproba1
    mu[1]=sommekluster2/sommeproba2

    return mu
```

To get the centroid of a cluster, we use mainly the same method that we used for the **K-means**:
We sum the value of each point, multiply his probability to belong to this cluster divided by the number of points that belong to this cluster. It finally gives the **updated centroid of the cluster**.

For ∑:

```
def Maximisation_sigma (w,data,mu,sigma):

    sommekluster1 = 0
    sommekluster2 = 0
    temp1 = 0
    temp2 = 0
    sommeproba1 = 0
    sommeproba2 = 0

    for i in range (600):

        dataligne = [data[i,]]
        dataligne = np.matrix(np.array(dataligne))
        dataligne = dataligne - np.matrix(np.array([mu[0,]]))
        transpose = dataligne.transpose()

        temp1=w[i,0]*transpose*dataligne

        sommekluster1=sommekluster1+temp1
        sommeproba1=sommeproba1+w[i,0]

        sigma[0]=sommekluster1/sommeproba1

        dataligne = [data[i,]]
        dataligne = np.matrix(np.array(dataligne))
        dataligne = dataligne - np.matrix(np.array([mu[1,]]))
        transpose = dataligne.transpose()

        temp2=w[i,1]*transpose*dataligne

        sommekluster2=sommekluster2+temp2
        sommeproba2=sommeproba2+w[i,1]
        sigma[1]=sommekluster2/sommeproba2


    return sigma
```

To get the **covariance maximization** we use the formula below:

$$\Sigma_j = \frac{\sum_{i=1}^{m} w_j^{(i)} \left(x^{(i)} - \mu_j\right)\left(x^{(i)} - \mu_j\right)^T}{\sum_{i=1}^{m} w_j^{(i)}}.$$

So, as we did for the Expectation step, we calculate **the difference between the point and the centroid of the cluster then we multiply by his transpose and by the probability of this point to be in the cluster**. This numerator is divided by the sum of probability to be in this cluster.

4) The linkelihood and convergence

The E-M algorithm is not based on only 1 iteration of its 2 steps. Indeed, the algorithm should run as long as it does not converge. So, it as an iterative algorithm.
As explained in the website *quora.com*, E represents a local maximum. If E represents a local maximum, and M represents a normal distribution, the function necessarily converges at point E (local maximum).
The E step is a chosen point, and the M step is a normal distribution that maximizes the likelihood of E.

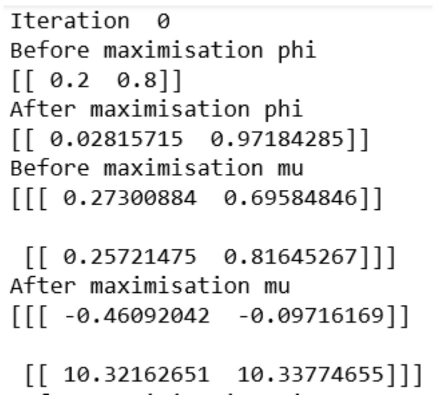
Problem: How can we know when the algorithm converges?

Of course, we can specify a value for the number of iterations and check the results every time in order to know whether the convergence happened or not.
However, it not really an efficient way, so we have decided to choose another option:
Basically, what we need to do in order to stop the iterative algorithm when it converges it to determine a tolerance. The tolerance is a value that will allow us to compare the old likelihood with the new one, and then take decision about stopping the algorithm or not.
In our case, we decided to put the tolerance equals to 0.001, that is pretty much a small and good value according to our model and dataset.
So as long as the difference between the new likelihood and the old one is not higher that the tolerance, then we iterate the E-M algorithm. At the opposite, we stop it.

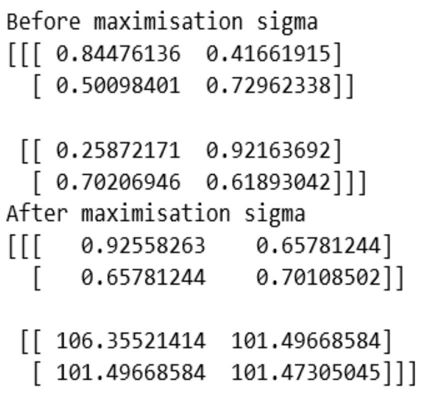
In the next section, we will display the "Negative log-likelihood predicted by a GMM" by following the steps defined in the example 1.

III- Results and discussion

1) *For the first iteration*



```
Iteration  0
Before maximisation phi
[[ 0.2  0.8]]
After maximisation phi
[[ 0.02815715  0.97184285]]
Before maximisation mu
[[[ 0.27300884  0.69584846]]

 [[ 0.25721475  0.81645267]]]
After maximisation mu
[[[ -0.46092042  -0.09716169]]

 [[ 10.32162651  10.33774655]]]
```

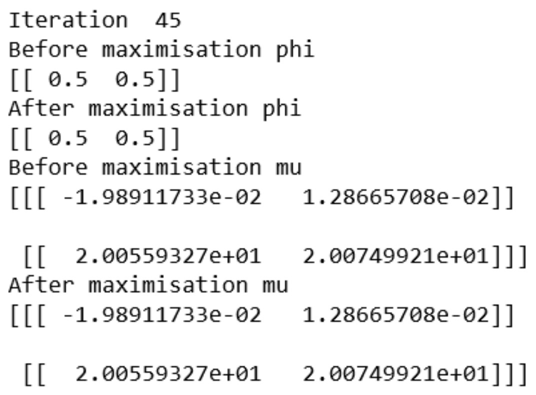


```
Before maximisation sigma
[[[ 0.84476136  0.41661915]
  [ 0.50098401  0.72962338]]

 [[ 0.25872171  0.92163692]
  [ 0.70206946  0.61893042]]]
After maximisation sigma
[[[   0.92558263    0.65781244]
  [   0.65781244    0.70108502]]

 [[ 106.35521414  101.49668584]
  [ 101.49668584  101.47305045]]]
```

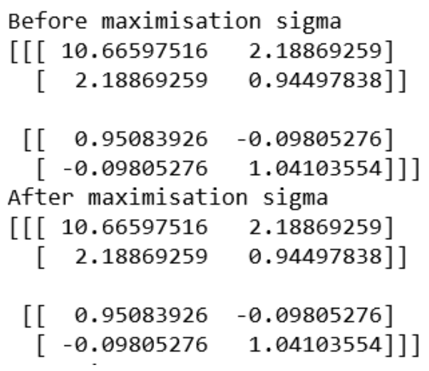
As we used **random values** to generate μ and ∑ and manually configure ϕ to be a 0.2 probability for first cluster and 0.8 for the second one, this part is not relevant. Nevertheless, it a good start for showing the evolution of our model.

We can see as result that the ∑ is high for the second cluster (located on the coordinated 20;20 because we randomly create the centroid around 0 as you can see in μ matrix)
We can also analyze that μ started to converge to the good centroid position and for ϕ, the probability is totally unbalanced due to the randomly centroids initialization that pushes far point to a specific cluster.

2) *After 45 iterations*



```
Iteration  45
Before maximisation phi
[[ 0.5  0.5]]
After maximisation phi
[[ 0.5  0.5]]
Before maximisation mu
[[[ -1.98911733e-02   1.28665708e-02]]

 [[  2.00559327e+01   2.00749921e+01]]]
After maximisation mu
[[[ -1.98911733e-02   1.28665708e-02]]

 [[  2.00559327e+01   2.00749921e+01]]]
```



```
Before maximisation sigma
[[[ 10.66597516   2.18869259]
  [  2.18869259   0.94497838]]

 [[  0.95083926  -0.09805276]
  [ -0.09805276   1.04103554]]]
After maximisation sigma
[[[ 10.66597516   2.18869259]
  [  2.18869259   0.94497838]]

 [[  0.95083926  -0.09805276]
  [ -0.09805276   1.04103554]]]
```

We have the results we **expected**:

ϕ represents the probability of a point to be in each cluster (because we have 300 points around (0,0) and 300 points around (20,20)

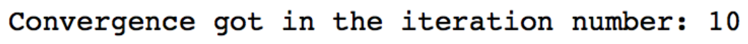
μ has finally converged to the expected centroids (0,0) and (20,20)

For the ∑, the cluster around (0,0) as a covariance higher than the one around (20,20)
It is because data pre-processing of the example1 **streched** the data around this centroid regarding to the other centroid.

3) *What is about Likelihood?*

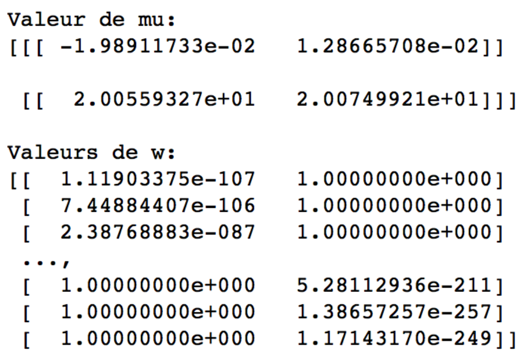
As mentioned in the previous part, we worked on the likelihood concept: how to know when the E-M algorithm converges, and how to display the Gaussian distributions for our models.

Firstly, in our case, the E-M algorithm stops after the 10th iteration:

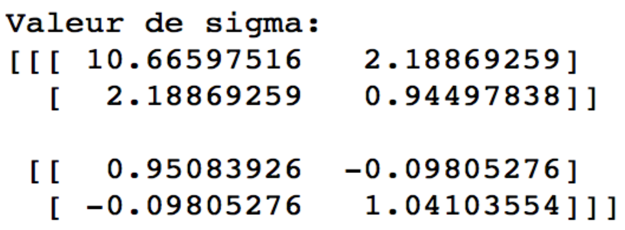


```
Convergence got in the iteration number: 10
```

Indeed, as you can see below, the parameters seem to fit to our model:



```
Valeur de mu:
[[[ -1.98911733e-02   1.28665708e-02]]

 [[  2.00559327e+01   2.00749921e+01]]]

Valeurs de w:
[[   1.11903375e-107   1.00000000e+000]
 [   7.44884407e-106   1.00000000e+000]
 [   2.38768883e-087   1.00000000e+000]
 ...,
 [   1.00000000e+000   5.28112936e-211]
 [   1.00000000e+000   1.38657257e-257]
 [   1.00000000e+000   1.17143170e-249]]
```



```
Valeur de sigma:
[[[ 10.66597516   2.18869259]
  [  2.18869259   0.94497838]]

 [[  0.95083926  -0.09805276]
  [ -0.09805276   1.04103554]]]
```

- The coordinates are still approximately the same as the 45th iteration, the centroids are around the coordinated (20;20) and (0;0).
- We can have the same analogy for ∑.
- W, that is the probability that the datapoint belongs to the cluster i (i representing the columns of W), then we can notice that it is pretty much clear. The model knows whether it belongs to the cluster 1 or 2, because we always have a probability close to 1 for one side, et 0 to the other.

Finally, in order to compare our results with the GMM function implemented in the sklearn library, we have decided to display the Gaussian distribution for both.

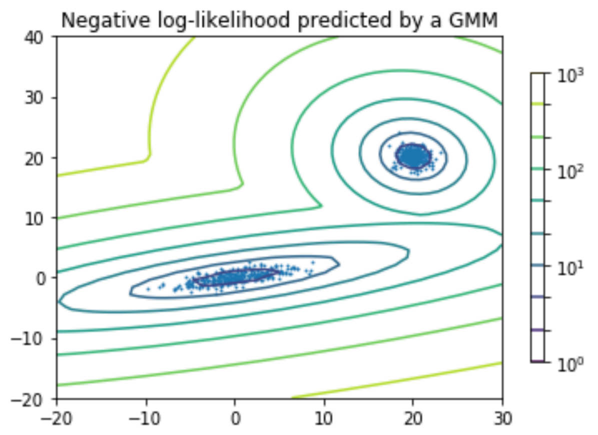


Figure 1: Figure of our own implementation of GMM

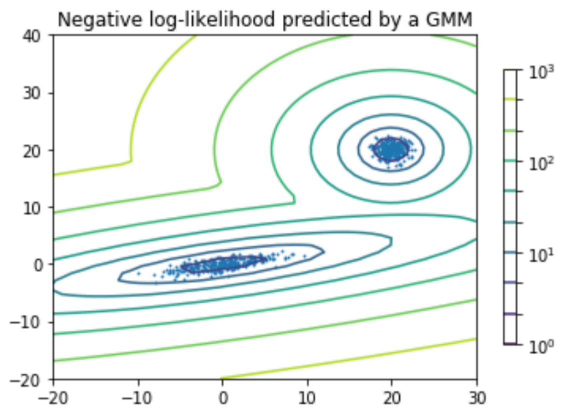


Figure 2: Figure of the Sklearn'GMM

We can conclude that the model we got from our own implementation of GMM is really close to what we got from the sklearn library. We have clearly 2 Gaussian distributions, with centroids and covariances **that match with the two groups of datapoints**.
Thus, the parameters ∑ and µ of the models are really close.
So, we can honestly judge that **our algorithm works well for this problem**.