Trabajo práctico 0: Algoritmo de Maximización de la Esperanza

Ph. D. Saúl Calderón Ramírez
Instituto Tecnológico de Costa Rica,
Escuela de Computación
PAttern Recongition and MAchine Learning Group (PARMA-Group)

3 de octubre de 2022

Estudiantes : Dennis Luna Acuña- Graciela Rivera Picado - Luis E. Vargas Porras

Fecha de entrega: Domingo 2 de Octubre del 2022.
Entrega: Un archivo .zip con el código fuente LaTeX o Lyx, el pdf, y un jupyter en Pytorch, debidamente documentado, con una función definida por ejercicio. A través del TEC-digital.
Modo de trabajo: Grupos de 3 personas.

Resumen

En el presente trabajo práctico se repasarán aspectos básicos del algebra lineal, relacionados con los conceptos a desarrollar a lo largo del curso, mezclando aspectos teóricos y prácticos, usando el lenguaje Python.

1. (100 puntos) Probabilidades: Algoritmo de Maximización de la Esperanza

A continuación, implemente el algoritmo de maximización de la esperanza (descrito en el material del curso), usando la definición y descripción de las siguientes funciones como base:

1. **(10 puntos)** Implemente la función *generate_data* la cual reciba la cantidad de observaciones unidimensionales total a generar N, y los parámetros correspondientes a $K = 2$ funciones de densidad Gaussianas. Genere los datos siguiendo tales distribuciones, y retorne tal matriz de datos $X \in \mathbb{R}^N$.

    a) Grafique los datos usando un *scatter plot* junto con las gráficas de la función de densidad de probabilidad, en la misma figura (gráfico), usando $\mu_1 = 10$, $\sigma_1 = 1{,}5$, $\mu_2 = 20$, $\sigma_2 = 3$.

R/ Para esta funcionalidad se implementaron 2 funciones, una llamada *createDataOneClass* encargada de construir una muestra Gaussiana dado los parámetros μ y σ. Seguidamente la función generate_data el cual recibe como parámetro el número de muestras por clase así como los parámetros iniciales $\mu_1 = 10, \sigma_1 = 1{,}5, \mu_2 = 20, \sigma_2 = 3$. Para ello se utilizaron tanto las funciones integradas de Pytorch como son *torch.distributions.Normal*, *torch.sample* para el caso de las muestras Gaussianas. Posteriormente se realiza la concatenación de las 2 muestras dando a la formación resultante de una muestra compuesta, para ello se utilizaron funcionalidades como torch.cat y scatter plots para la muestra del resultado.

En la figura 1 se puede apreciar la porción del código descrito anteriormente:

```python
def createDataOneClass(means, std, numberSamplesPerClass):
    normal_dist = torch.distributions.Normal(means,std)
    gaussian_sample = normal_dist.sample((numberSamplesPerClass,1)).squeeze()

    return gaussian_sample


def generate_data(numberSamplesPerClass, mean1, mean2 , stds1 , stds2 ):
    """
    Creates the data to be used for training, using a GMM distribution
    @param numberSamplesPerClass, the number of samples per class
    @param mean1, means for samples from the class 1
    @param mean2, means for samples from the class 2
    @param stds1, standard deviation for samples, class 1
    @param stds2, standard deviation for samples, class 2    """

    samplesClass1 = createDataOneClass(mean1, stds1, numberSamplesPerClass)
    samplesClass2 = createDataOneClass(mean2, stds2, numberSamplesPerClass)
    #Concatenates
    samplesAll = torch.cat((samplesClass1, samplesClass2), 0)
    print (samplesAll)

    # Plot Data
    y = np.zeros_like(samplesAll)
    plt.figure(figsize=(12,8))
    plt.scatter(samplesAll, y, label='Observations')
    #plt.title('Plotting)
    plt.xlabel('X Value')
    plt.ylabel('PDF')

    #Plot densities Functions
    plt.scatter(samplesClass1, norm.pdf(samplesClass1, mean1, stds1), color="green", label='Original Bi-Gaussian')
    plt.scatter(samplesClass2, norm.pdf(samplesClass2, mean2, stds2), color="green")
    plt.legend()
    plt.show()


    return (samplesAll, samplesClass1, samplesClass2)

'''
Creates data with gaussian distribution
'''

#Call Function generate Data
X_allSamples, X1_samples, X2_samples = generate_data(numberSamplesPerClass=100, mean1=10.0, mean2=20.0, stds1=1.5, stds2=3.0
```

Figura 1: Código para la generación de datos

En la figura 2 se puede apreciar de manera g´grafica el resultado de las curvas a partir de la muestra compuesta obtenida:

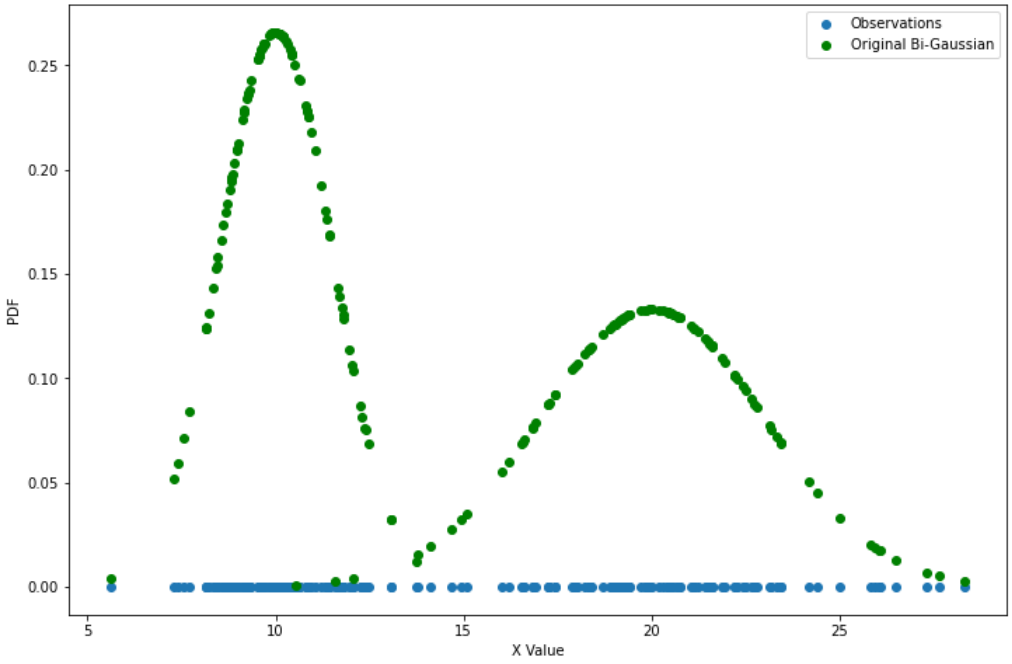


Figura 2: Graficación de muestra compuesta

2. **(10 puntos)** Implemente la función *init_random_parameters* la cual genere una matriz de $W \in \mathbb{R}^{K \times 2}$ dimensiones, con los parámetros de las funciones de densidad Gaussiana generados completamente al azar.

   a) Muestre un pantallazo donde verifique su funcionamiento correcto con los comentarios asociados.

   En la figura 3 se muestra el código donde se aprecia los detalles de implementación a nivel de código:

2. Create init_random_parameters

```python
def init_random_parameters(K=2, range_mean = [1,10], range_std = [1,5]):

    # Se pasan por por parámetro el rango donde se elegiran de manera aleatoria los valores.
    # K2 corresponde a la forma matricial para el acomodo de los elementos, en este caso las columna 0 de la matriz
    # corresponden a los means y la columna 1 a los sigma.


    mean = torch.randint(range_mean[0],range_mean[1],(K,1))   # Creación de arreglo de tensores que contiene los means o mu
    std  = torch.randint(range_std[0], range_std[1],(K,1))    # Creación de arreglo de tensores que contiene los sigma


    W = torch.cat((mean, std), 1)

    g= W.numpy()

    # Se procede a realizar la graficación para comprobar su funcionamiento

    numberSamplesPerClass=100
    mean1=g[0,0]
    mean2=g[1,0]
    stds1=g[0,1]
    stds2=g[1,1]

    samplesClass1 = createDataOneClass(mean1, stds1, numberSamplesPerClass)
    samplesClass2 = createDataOneClass(mean2, stds2, numberSamplesPerClass)
    #Concatenates
    samplesAll = torch.cat((samplesClass1, samplesClass2), 0)



    y = np.zeros_like(samplesAll)
    plt.figure(figsize=(12,8))

    plt.scatter(samplesAll, y, marker='.', label="Observations")
    mu1 = round(g[0,0].item(),2)
    mu2 = round(g[1,0].item(),2)
    sig1 = round(g[0,1].item(),2)
    sig2 = round(g[1,1].item(),2)

    plt.title("μ1=%s, σ1=%s, μ2=%s, σ2=%s"%(mu1, sig1, mu2, sig2))
    plt.xlabel('X Value')
    plt.ylabel('PDF')

    #Plot densities Functions
    plt.scatter(samplesClass1, norm.pdf(samplesClass1, mean1, stds1), color='green', label='Original Bi-Gaussian')
    plt.scatter(samplesClass2, norm.pdf(samplesClass2, mean2, stds2), color='green')

    plt.legend()
    plt.show()


    return W

#Test

W_parameters = init_random_parameters(K=2, range_mean = [1,50], range_std = [1,10])
W_parameters
```

Figura 3: Código de implementación para la inicialización aleatoria de parámetros

b) Muestre una grafica de las funciones de densidad con los parámetros inicializados aleatoriamente.

Seguidamente en la figura 4 se nota la salida resultante de la función descrita. Asi mismo se construye una nueva muestra compuesta para propósito de validación y depuración.

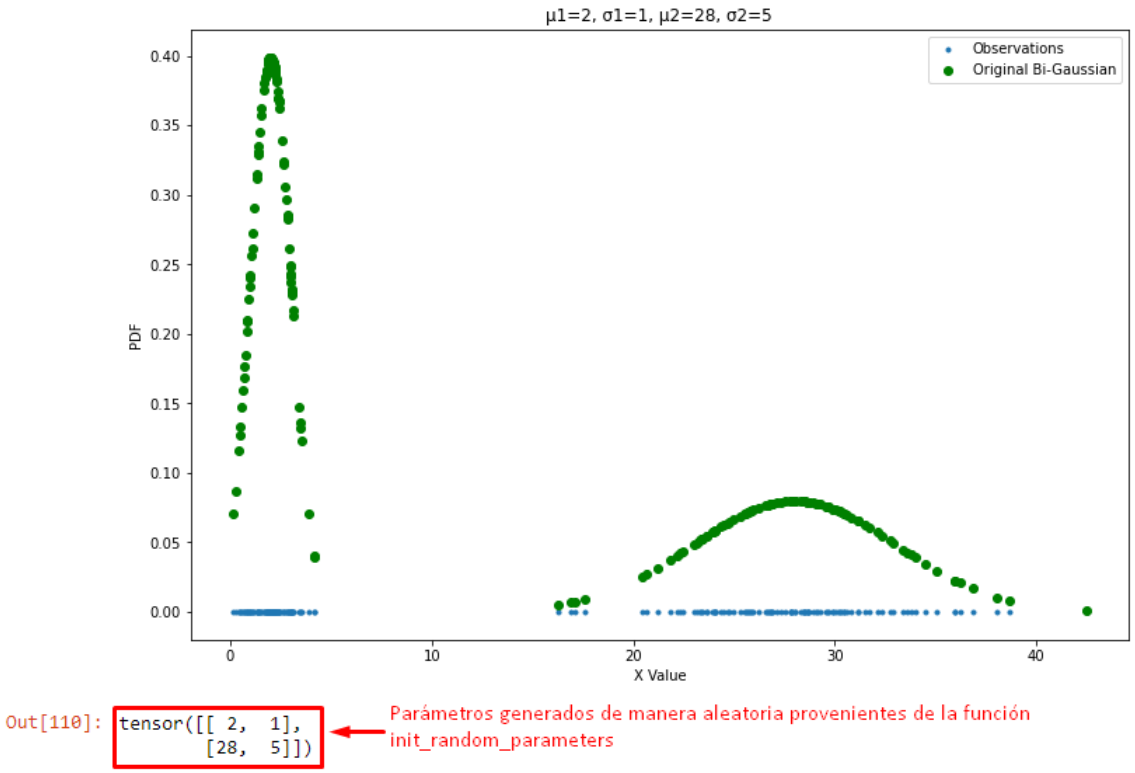


Figura 4: Resultado de la inicialización de parámetros aleatoria

3. **(10 puntos)**Implemente la función *calculate_likelihood_gaussian_observation(x_n, mu_k, sigma_k)* la cual calcule la verosimilitud de una observación específica x_n, para una función de densidad Gaussiana con parámetros μ_k y σ_k.

   a) Muestre un pantallazo donde verifique su funcionamiento correcto con los comentarios asociados, usando los datos anteriormente generados con $\mu_1 = 10$, $\sigma_1 = 1{,}5$, $\mu_2 = 20$, $\sigma_2 = 3$.
   En la figura 5 se puede apreciar 2 funciones de *calculate_likelihood_gaussian_observation(x_n, mu_k, sigma_k)* donde se utiliza el logaritmo como parte de la implementación y la otra función incluye dicho parámetro. Con base a las observaciones y resultados obtenidos se puede notar que para el caso donde no se utiliza logaritmo en el cálculo, se adquiere un resultado entre el rango 0 y 1 dando mas sentido a lo que la verosimilitud dictamina.

3. Likelihood Gaussian Observation Function

```
# Likelihood sin inserción de logaritmo
def calculate_likelihood_gaussian_observation(x_n,mu_k,sigma_k):

    p_x1 = prob_density = (1/np.sqrt((2.0*np.pi*sigma_k[0]))) * np.exp(-0.5*((x_n-mu_k[0])/sigma_k[0])**2)
    p_x2 = prob_density = (1/np.sqrt((2.0*np.pi*sigma_k[1]))) * np.exp(-0.5*((x_n-mu_k[1])/sigma_k[1])**2)
    #p_xAll = torch.Tensor([p_x1,p_x2])
    return p_x1,p_x2

calculate_likelihood_gaussian_observation(X_allSamples[0], mu_k = [10,20], sigma_k = [1.5,3])
```

```
(tensor(0.2453), tensor(0.0002))
```

← Resultado obtenido sin insertar logaritmo

```
# Likelihood con  inserción de logaritmo
def calculate_likelihood_gaussian_observationLOG(x_n,mu_k,sigma_k):

    p_x1 = prob_density = (1/np.sqrt((2.0*np.pi*sigma_k[0]))) * np.exp(-0.5*((x_n-mu_k[0])/sigma_k[0])**2)
    logp_x1= np.log(p_x1)

    p_x2 = prob_density = (1/np.sqrt((2.0*np.pi*sigma_k[1]))) * np.exp(-0.5*((x_n-mu_k[1])/sigma_k[1])**2)
    logp_x2 =  np.log(p_x2)

    return logp_x1,logp_x2

calculate_likelihood_gaussian_observationLOG(X_allSamples[0], mu_k = [10,20], sigma_k = [1.5,3])
```

```
(tensor(-1.2137), tensor(-7.7617))
```

← Resultado con inserción de logaritmo

```
# Conclusión, para efectos de los datos se determina utilizar likelihood sin logaritmo.
```

Figura 5: Cálculo de verosimilitud con o sin logaritmo

4. **(10 puntos)** Implemente la función *calculate_membership_dataset(X_dataset, Parameters_matrix)*, la cual, usando la matriz de parámetros W y la función anteriormente implementada *calculate_likelihood_gaussian_observation*, defina por cada observación $x_n \in X$ la pertenencia o membresía a cada cluster $k = 1, \ldots K$, en una matriz binaria $M \in \mathbb{R}^{N \times K}$. Retorne tal matriz de membresía M.

   a) Muestre un pantallazo donde verifique su funcionamiento correcto con los comentarios asociados, usando los datos de prueba anteriormente generados.

   En la siguiente figura se puede observar la matriz de membresía la cual corresponde a la matriz binaria $M \in \mathbb{R}^{N \times K}$que logra clasificar cada una de las observaciones en k=2 clusters según la verosimilitud calculada con la función realizada en la sección 3. Por cada observación se determina cúal es el cluster más cercano de acuerdo a la probabilidades calculadas. La siguiente estructura lógica se muestra a continuación:

```python
def calculate_membership_dataset(X_dataset,Parameters_matrix):
    #crear una matriz con ceros
    N = len(X_dataset)
    k = 2
    M = torch.zeros((N, k))

    #calculate_likelihood
    likelihood1,likelihood2 = calculate_likelihood_gaussian_observation(
                                    X_dataset,Parameters_matrix[:,0],Parameters_matrix[:,1])

    #print("likelihood2:", likelihood2.reshape([N, 1]))
    likelihoodALL = torch.cat((likelihood1.reshape([N, 1]), likelihood2.reshape([N, 1])), 1)

    M[:,0] = likelihoodALL[:,0] > likelihoodALL[:,1]
    M[:,1] = likelihoodALL[:,0] < likelihoodALL[:,1]

    return M

#fix_param = torch.tensor([[10,1.5],[20,3]])
Membership_Matrix = calculate_membership_dataset(X_allSamples,W_parameters)
Membership_Matrix

```

```
tensor([[1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.],
```

Figura 6: Matriz de membresía M

Es importante mencionar que se utiliza el criterio visto en clase el cuál se representa en la línea 14 y 15 del código donde se define el criterio de pertenencia de cada observación. Además, se realiza directamente la comparación en la matriz de verosimilitud de la campana 1 y 2 respectivamente, para evitar el uso de estructuras de repetición.

5. **(20 puntos)** Implemente la función *recalculate_parameters(X_dataset, Membership_data)*, la cual recalcule los parámetros de las funciones de densidad Gaussianas representandas en la matriz W, de acuerdo a lo representado en la matriz de membresía M. Debe retornar la matriz con los paráramетros W.

   a) Use las funciones *mean* y *std* de pytorch para ello. Intente prescindir al máximo de estructuras de repetición tipo *for*.

   b) Muestre el resultado para un conjunto de datos de prueba y comente los resultados.

   Para esta sección se usa un set de datos generados utilizando las funciones de las secciones anteriores 1 y 2, contemplando los siguientes parámetros $\mu_1 = 10$, $\sigma_1 = 1{,}5$, $\mu_2 = 20$, $\sigma_2 = 3$. La estructura ló-

gica que se utiliza para la generación de esta función se muestra a continuación:

5. Recalculate_parameters

```
def recalculate_parameters(X_dataset,Membership_data):

    N = len(X_dataset)

    X1 = torch.cat((X_dataset.reshape([N, 1]),Membership_data[:,0].reshape([N, 1])),1)
    X2 = torch.cat((X_dataset.reshape([N, 1]),Membership_data[:,1].reshape([N, 1])),1)

    X1_filtrada = X1[X1[:,1] == 1][:,0]
    X2_filtrada = X2[X2[:,1] == 1][:,0]

    mu_x1 = torch.mean(X1_filtrada)
    mu_x2 = torch.mean(X2_filtrada)

    sigma_x1 = torch.std(X1_filtrada)
    sigma_x2 = torch.std(X2_filtrada)

    new_W = torch.tensor([[mu_x1,sigma_x1],[mu_x2,sigma_x2]])

    return new_W

recalculate_parameters(X_allSamples,Membership_Matrix)

```

```
tensor([[10.1907,  5.9349],
        [16.4188,  4.8510]])
```

Figura 7: Matriz de recálculo de parámetros

c) Es imporante recalcar que la lógica mostrada utiliza una función que hace que por cada cluster se recalculen los parámetros(mu y sigma) usando la pertenencia de los datos previamente calculados. Lo anterior resulta una matriz new_W $\in \mathbb{R}^{K\times 2}$, en este caso con este set de datos, se logra como resultado unos medias de 10.190 y 16.4188 y unas desviaciones estándar de 5.93 y 4.85. Es importante notar que la desviación estándar de la campana 1 tiene una diferencia significativamente más alta en comparación con el desviación estándar inicial. Lo anterior, puede verse como una mejora en el algoritmo en la fase de inicialización de parámetros, ya que el éxito de el E-M algoritmo depende de una buena incialización de parámetros (S. Calderón, 2022) , con el fin de que la convergencia hacia los valores reales no solo se realice con una mayor exactitud, sino que se realice en un menor tiempo computacional al final de cada corrida.

6. **(20 puntos)** Ejecute 5 corridas diferentes del algoritmo, donde por cada una documente los parámetros a los que se arribó. Ejecute cada corrida con $R = 20$ iteraciones.

   a) Grafique las funciones de densidad de probabilidad a las que con-

vergió el algoritmo. Puede graficar también las funciones de densidad obtenidas en 2 o 3 pasos intermedios.

b) Comente los resultados, usando algún criterio de correctitud de los mismos.

Los resultados de las 5 corridas con $R = 20$ iteraciones se muestran en la siguiente tabla. En cada una de ellas se registra cada uno de los parametros obtenidos en quinta iteración y la décima iteración, con el fin de observar los pasos intermedios de la convergencia.

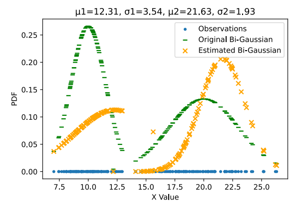
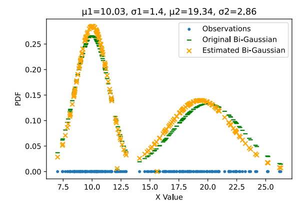
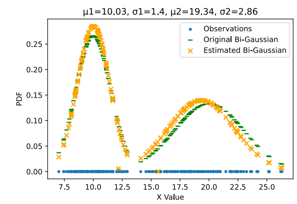
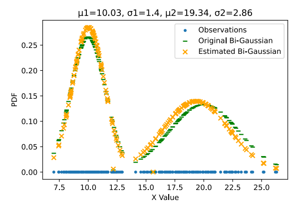
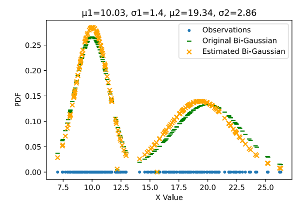
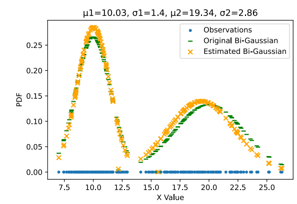
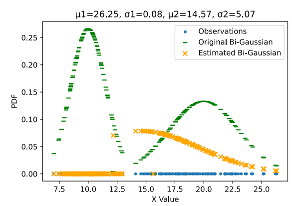
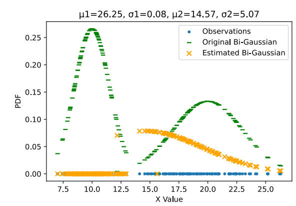
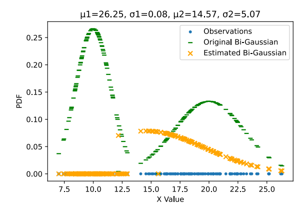
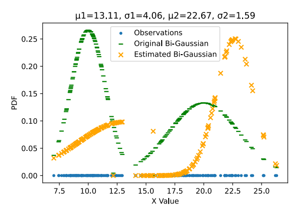
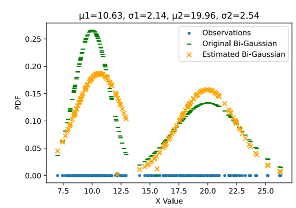
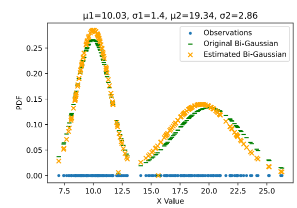
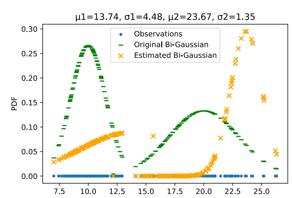
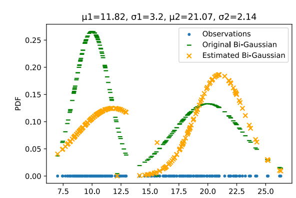
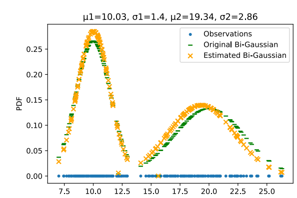
#Corridas	iteración_5	iteración_10	iteración_20
1			
2			
3			
4			
5			

Figura 8: Tabla de resultados de las corridas: E-M Algoritmo.

Al analizar la siguiente tabla podemos observar la evolución que tiene cada una las iteraciones para llegar a una convergencia óptima. Se puede observar que en la mayoría de los casos se inicia con parámetros muy lejanos a los parámetros reales, por lo que confirma nuestra hipótesis de que el algoritmo puede mejorar su proceso de inicialización, ya que las medias y desviaciones estándar seleccionadas aleatoriamente puede afec-

tar la convergencia. Un ejemplo se puede observar en la corrida número 3, donde los puntos de color amarillo (Estimated Bi-Gaussian), nunca se aproximó a las observaciones de color verde (Original Bi-Gaussian), lo cual se puede verse impacto por una desviación estándar muy baja de aproximadamente cero para la primera campana y una desviación estándar alta para la segunda campana.

Otra observación importante que se analiza es la inestabilidad de las 5 corridas (en algunas corridas convergen más lentas que otras e inclusive algunas no convergen), por lo que se probó como criterio de correctitud la aplicación del logaritmo para la función de densidad de probabilidad, sin embargo los resultados se mostraron similares, lo que conviene estudiar y analizar una mejor heurística para la etapa de inicialización de parámetros como posible alternativa.

7. **(20 puntos)** Proponga una mejor heurística para inicializar los parámetros del modelo aleatoriamente.

   a) Compruebe la mejora obtenida con el método propuesto, corriendo las pruebas del punto anterior.

   La heurística propuesta para la búsqueda de una mejora en la convergencia, se escogió como base las iteraciones restrictivas para la incialización de los parámetros. Según (E. Shireman, D. Steinley, M. J. Brusco) es una técnica que aplica las iteraciones de manera controlada utilizando una mezcla de valores iniciales en conjunto con el modelo mixto Gaussiano, normalmente esto se conoce como M_Step donde se busca la maximización de la verosimilitud. Tomando como referencia el modelo Gaussiano multivariable se realizan iteraciones controladas e implementación base que describe (S. Causevic). Por lo que se tomaron las siguientes ecuaciones para el cálculo de parámetros:

   $\mu_b = \frac{b_1 x_1 + b_2 x_2 + ... + b_n x_n}{b_1 + b_2 + ... + b_n}$

   $\sigma_b^2 = \frac{b_1 (x_1 - \mu_b)^2 + ... + b_1 (x_n - \mu_b)^2}{b_1 + b_2 + ... + b_n}$

   $\mu_a = \frac{a_1 x_1 + a_2 x_2 + ... + a_n x_n}{b_1 + a_n + ... + a_n}$

   $\sigma_a^2 = \frac{a_1 (x_1 - \mu_a)^2 + ... + a_1 (x_n - \mu_a)^2}{a_1 + a_2 + ... + a_n}$

   Sin embargo a nivel de implementación se utilizó como base la función multivariable Gaussiana proveniente de la biblioteca SciPy, en donde dicha función genera una matriz de covarianza para el cálculo de sigmas y posteriormente realizar las operaciones para la obtención de la máxima verosimilitud. Sin embargo, dado a que se está evaluando en un ecosistema unidimensional se aplica una segunda heurística propuesta por el equipo de trabajo, donde se toma el valor absoluto minimo y posteriormente se aplicó una escalación de 10^{15}veces con el fin de obtener convergencia con base a los datos generados en la primera parte, ya que en primera instancia se

obtienen valores muy bajos de sigmas que impiden una muestra de convergencia poco conveniente.

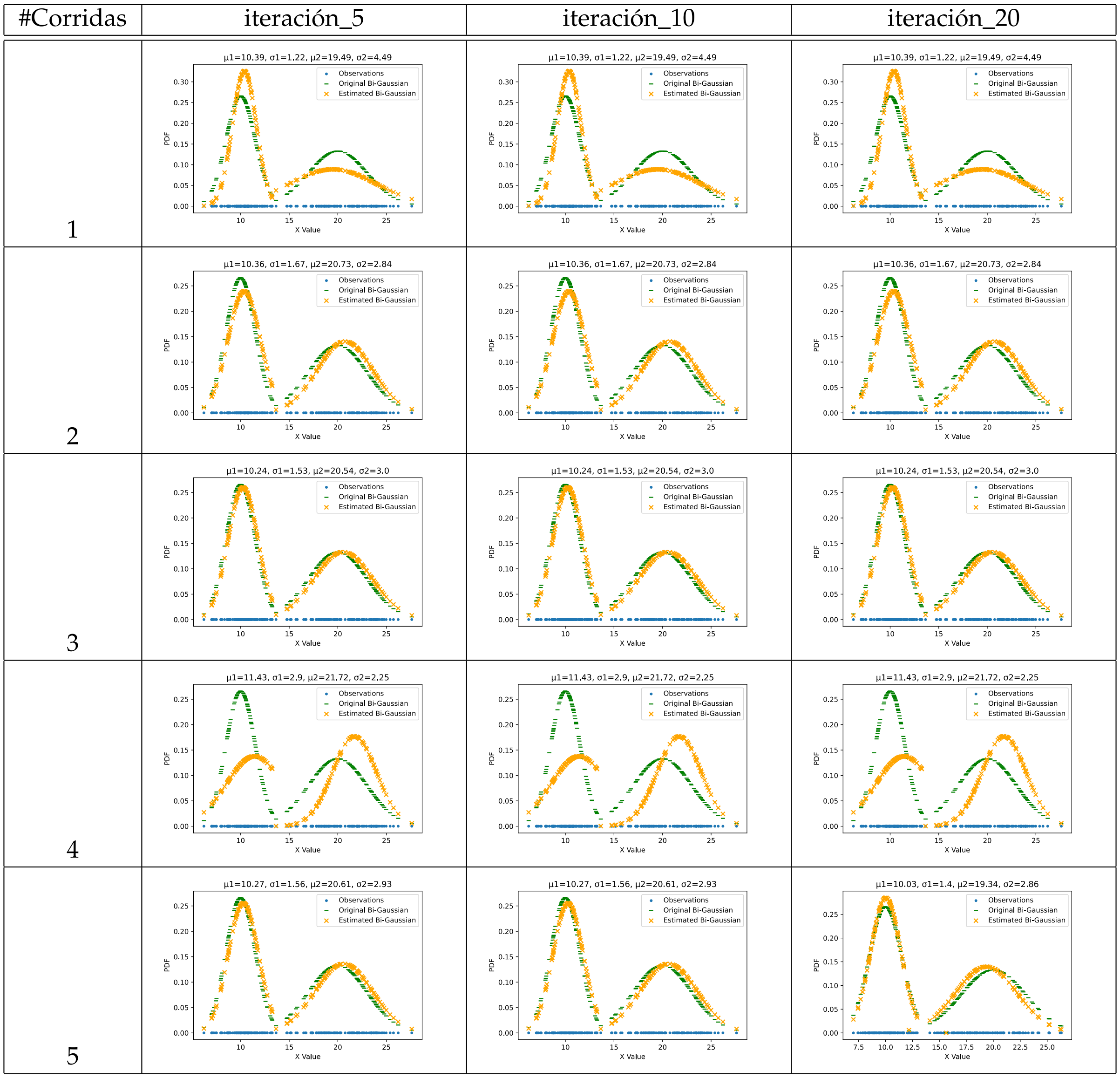


Figura 9: Tabla de resultados aplicando Heuristica Propuesta

En la figura 9 se muestra la serie de iteraciones obtenidas de la heurística descrita, adicionalmente se concluye que la convergencia es alta en comparación con la inicialización aleatoria sin ningún tratamiento previo. Sin embargo a nivel de implementación se necesitan

realizar ajustes mayoritamente en la selección de la matriz de covarianza.

Referencias:

[1] E. Shireman, D. Steinley, and M. J. Brusco, "Examining the effect of initialization strategies on the performance of Gaussian mixture modeling," Behav. Res. Methods, vol. 49, no. 1, pp. 282–293, 2017.

[2] S. Causevic, "Implement expectation-maximization(EM) in python," Towards Data Science, 26-Nov-2020.

[Online]. Available: https://towardsdatascience.com/implement-expectation-maximization-em-algorithm-in-python-from-scratch-f1278d1b9137. [Accessed: 01-Oct-2022].