Trabajo práctico 0:
Algoritmo de Maximización de la Esperanza

Marco Ferraro, Ricardo Chacón, Gabriel Valentine

Septiembre 23, 2023

Abstract

En el presente trabajo práctico se introduce al algoritmo de maximización de esperanza, y su aplicación para la segmentación de imágenes.

1 Algoritmo de Maximización de la Esperanza con datos artificiales

1. (15 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 $X \in R^{N \times 1}$.

   a) Grafique los datos usando un scatter plot junto con las gráficas de loshistogramas de los datos y las funciones de densidad de probabilidad Gaussianas usando los parámetros para inicializar los datos, en la misma figura (gráfico).

```python
import numpy as np

#Genera un conjunto de datos aleatorio a partir de una combinacion de distribuciones normales
def generate_data(N, means: np.ndarray, stds: np.ndarray, K=2):
  if len(means) != len(stds): # Comprobar si las longitudes de 'means' y 'stds' son diferentes
    print("Error␣in␣dimensions")
    return None
  else:
    data = []
# Generar muestras a partir de una distribucion normal estandar y ajustarlos
    for i in range(K): as
      y = np.sqrt(-2 * np.log(np.random.rand(N // K))) * np.cos(2 * np.pi * np.random.rand(N // 2))
      y = (y * stds[i]) + means[i]

      data.append(y)

    X = np.concatenate(data)

    # Mezclar datos para que no esten ordenados
    np.random.shuffle(X)

    return X.reshape(-1, 1)
#Ejemplo:
    N = 100

means = [3.0, 7.0]
stds = [0.5, 0.8]

data = generate_data(N=100, means=means, stds=stds)
data
```

```
array([[3.378], [3.167], [2.663], [6.278], [2.892], [6.98 ], [1.962], [3.404],
[7.043], [7.125], [6.351], [2.439], [7.26 ], [7.515], [7.253], [7.42 ], [3.113],
[3.516], [7.768], [7.389], [3.111], [2.607], [1.876], [7.359], [2.248], [7.135],
[6.556], [8.456], [6.518], [7.324], [3.276], [3.365], [9.047], [3.045], [3.22 ],
[2.829], [7.697], [7.558], [4.039], [4.689], [2.595], [3.293], [6.867], [3.643],
[3.346], [2.748], [3.117], [2.772], [7.513], [2.607], [6.139], [7.31 ], [2.714],
[6.412], [6.906], [2.704], [3.858], [3.251], [7.552], [2.995], [7.943], [6.075],
[2.605], [1.817], [2.466], [7.436], [6.643], [3.826], [6.892], [7.75 ], [3.6 ],
[7.143], [3.167], [8.019], [3.105], [2.582], [6.866], [6.339], [6.237], [6.231],
[3.908], [5.837], [7.49 ], [7.888], [2.225], [3.645], [7.41 ], [3.484], [7.024],
[2.775], [7.527], [5.623], [7.558], [5.451], [3.543], [3.044], [6.93 ], [2.891],
[7.286], [3.486]])
```

```python
#Graficamos
import matplotlib.pyplot as plt
import numpy as np

def plot_data(data, title='Two␣Normal␣Distributions'):
    hist_values, bin_edges = np.histogram(data, bins=20)
    fig, ax = plt.subplots(figsize=(8, 6))
    ax.bar(bin_edges[:-1], hist_values, width=np.diff(bin_edges), color='skyblue', alpha=0.7, label='Frequency')
    ax.scatter(data, np.zeros_like(data), marker='o', s=30, color='b', label='Data')

    y_min, y_max = ax.get_ylim()
    ax.set_ylim(y_min - 0.5, y_max + 0.1)

    ax.set_xlabel('Value')
    ax.set_ylabel('Frequency')
    ax.set_title(title)
    ax.grid(True, which='both')

    ax.legend()
    plt.show()

    plot_data(data)
```

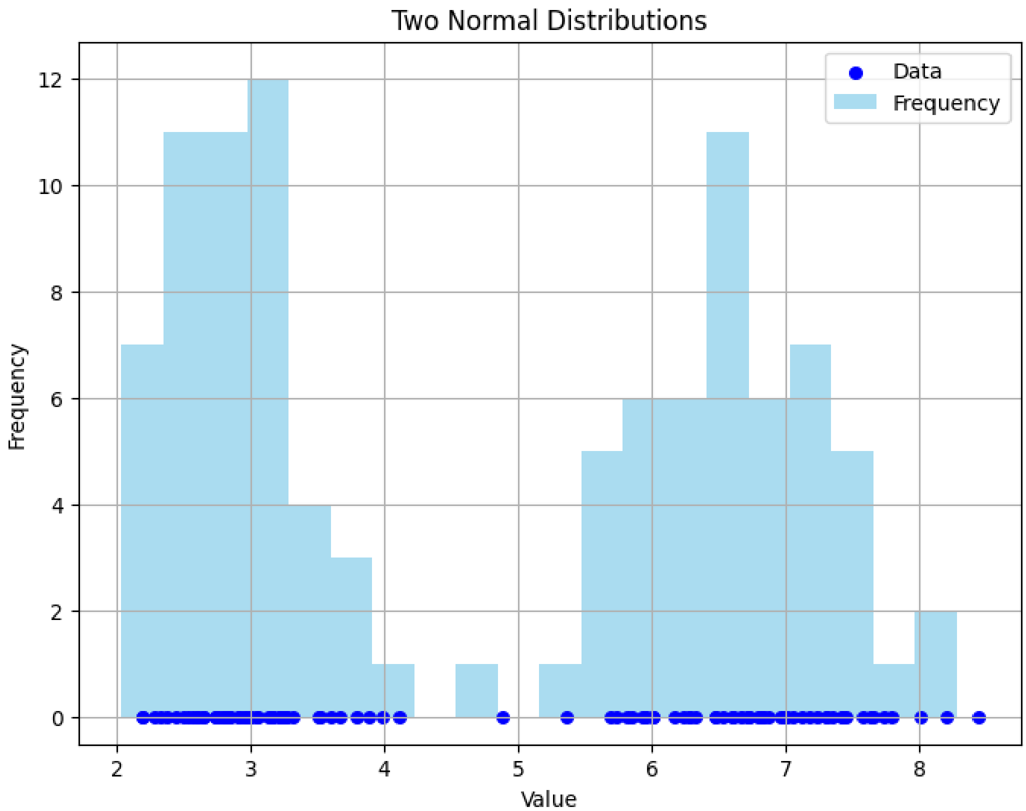

Figure 1: Análisis de los datos

2. Implemente la función `init_random_parameters` la cual genere una matriz $P \in R^{K \times 2}$ de dimensiones, con los parametros de las funciones de densidad Gaussiana generados completamente al azar.

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

```python
# El asterisco es para desempacar la tupla
def init_random_parameters(K, mean_range=(0, 10), std_range=(0, 1)):
  means = np.random.uniform(*mean_range, size=K)
  stds = np.random.uniform(*std_range, size=K)

  params = np.column_stack((means, stds)).tolist()

  return params

  init_random_parameters(3)
```

```
[[7.065079293907514, 0.23386200242383015],
 [8.359163627456542, 0.481536877138774],
 [3.791742857806563, 0.45383778855691903],
 [1.9012224822214685, 0.17143595210428797],
 [7.437920981438372, 0.8207967987216976]]
```

Figure 2: Parámetros de las funciones de densidad Gaussiana generados al azar

3. 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. Realice la corrección pertinente al cálculo de la función de verosimilitud para evitar el problema de underflow.

   a) Diseñe y ejecute una prueba unitaria donde verifique su funcionamiento correcto con los comentarios asociados.

```python
import numpy as np

def calculate_likelihood_gaussian_observation(x_n, mu_k, sigma_k):
    log_likelihood = -0.5 * np.log(2 * np.pi * sigma_k**2) - 0.5 * ((x_n - mu_k) / sigma_k)**2
    return log_likelihood

# Prueba Unitaria
x_n = 3.0
mu_k = 2.0
sigma_k = 1.0

likelihood = calculate_likelihood_gaussian_observation(x_n, mu_k, sigma_k)

# Comprobacion
print(f'Likelihood:␣{likelihood}')
```

```
Likelihood: -1.4189385332046727
```

4. Implemente la función `calculate_membership_dataset(X_dataset, Parameters_matrix)`, la cual, usando la matriz de parámetros P 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 $M \in R^{N \times K}$. Retorne tal matriz de membresía M.

```
import numpy as np

def calculate_membership_dataset(X_dataset, Parameters_matrix):
    N, K = X_dataset.shape[0], Parameters_matrix.shape[0]
    M = np.zeros((N, K))

    for i in range(N):
        for k in range(K):
            mu_k, sigma_k = Parameters_matrix[k]
            log_likelihood = calculate_likelihood_gaussian_observation(X_dataset[i], mu_k, sigma_k)
            M[i, k] = log_likelihood

    # Aplicar normalizacion softmax para obtener valores entre 0 y 1
    M = np.exp(M - np.max(M, axis=1, keepdims=True))
    M /= np.sum(M, axis=1, keepdims=True)

    return M

# Prueba Unitaria
X_dataset = np.array([[2.5], [3.0], [1.8], [2.2]])  # Ejemplo de dataset (4 observaciones)
Parameters_matrix = np.array([[2.0, 0.5], [3.0, 0.3]])
# Ejemplo de matriz de parametros (2 clusters)

membership_matrix = calculate_membership_dataset(X_dataset, Parameters_matrix)

# Comprobacion
print("Matriz␣de␣Membresia:")
print(membership_matrix)
```

```
Matriz de Membresía:
[[5.93405901e-01 4.06594099e-01]
 [7.51027396e-02 9.24897260e-01]
 [9.99394696e-01 6.05303929e-04]
 [9.50955067e-01 4.90449332e-02]]
```

Figure 3: Prueba unitaria

5. Implemente la función `recalculate_parameters(X_dataset, Membership_data)`, la cual recalcule los parámetros de las funciones de densidad Gaussianas representadas en la matriz P, de acuerdo a lo representado en la matriz de membresía M.

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

```python
import torch

def recalculate_parameters(X_dataset, Membership_data):
    X_torch = torch.tensor(X_dataset, dtype=torch.float32)
    Membership_torch = torch.tensor(Membership_data, dtype=torch.float32)

    # Calcular nuevas medias
    new_means = torch.matmul(Membership_torch.T, X_torch) / torch.sum(Membership_torch, dim=0, keepdim=True)

    # Calcular nuevas desviaciones est ndar
    deviations = X_torch.unsqueeze(1) - new_means
    squared_deviations = deviations ** 2
    weighted_squared_deviations = squared_deviations * Membership_torch.unsqueeze(2)
    new_std = torch.sqrt(torch.sum(weighted_squared_deviations, dim=0) / torch.sum(Membership_torch, dim=0, keepdim=True))

    return new_means.numpy(), new_std.numpy()

# Ejemplo de uso
X_dataset = np.array([[2.5], [3.0], [1.8], [2.2]], dtype=np.float32)
Membership_data = np.array([[0.2, 0.8], [0.6, 0.4], [0.3, 0.7], [0.9, 0.1]], dtype=np.float32)

new_means, new_std = recalculate_parameters(X_dataset, Membership_data)

print("Nuevas␣Medias:")
print(new_means)

print("\nNuevas␣Desviaciones␣Estandar:")
print(new_std)
```

```
Nuevas Medias:
[[2.41 2.41]
 [2.34 2.34]]

Nuevas Desviaciones Estándar:
[[0.4253234 0.4253234]
 [0.4476606 0.4476606]]
```

Figure 4: Parámetros de las funciones de densidad Gaussianas

6. Ejecute 5 corridas diferentes del algoritmo, donde por cada una documente los parámetros a los que se arribó.

   a) Grafique las funciones de densidad de probabilidad a las que convergió el algoritmo. Puede graficar también las funciones de densidad obtenidas en 2 o 3 pasos intermedios. Presente una tabla de gráficas donde en cada entrada se identifique el número de iteración y los parámetros iniciales.

      Vamos a correr el algoritmo por 5 épocas. La idea es que a lo largo de cada época, las gráficas se asemejen a la gráfica real.

      El primer paso es inicializar las variables. Tenemos unos parámetros reales y unos aleatorios. Nótese que los datos los vamos a generar con los parámetros reales.

      ```
      EPOCHS = 5
      ```

```
K = 2

P = init_random_parameters(K=K, mean_range=(0, 50), std_range=(0,
P = np.array(P)

true_P = init_random_parameters(K=K, mean_range=(0, 50), std_range=
true_P = np.array(true_P)
X = generate_data(N=100, means=true_P[:, 0], stds=true_P[:, 1], K=

true_P
plot_data(X)
plot_multiple_normal_distributions(means=P[:, 0], stds=P[:, 1])

for i in range(EPOCHS):
    membership_matrix = calculate_membership_dataset(X, P)
    P = recalculate_parameters(X, membership_matrix)
    plot_multiple_normal_distributions(means=P[:, 0], stds=P[:, 1]
```

b) Comente los resultados.

7. 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.

   Vamos a correr el algoritmo por 5 épocas. La idea es que a lo largo de cada época, las gráficas se asemejen a la gráfica real.

   El primer paso es inicializar las variables. Inicializaremos los parámetros con medias que estén relativamente muy distantes entre sí, manteniendo el mismo valor de desviación estándar.

```
EPOCHS = 5
K = 2

def init_fixed_parameters(K, mean_values=None, std_value=15.0):
    if mean_values is None:
        mean_values = [1.0] * K

    means = np.array(mean_values)
    stds = np.full(K, std_value)
```

```
    P = np.column_stack((means, stds)).tolist()

    return P

mean_values = [5.0, 55.0]

P = init_fixed_parameters(K=K, mean_values=mean_values)
P = np.array(P)

plot_multiple_normal_distributions(means=P[:, 0], stds=P[:, 1])

for i in range(EPOCHS):
    membership_matrix = calculate_membership_dataset(X, P)
    P = recalculate_parameters(X, membership_matrix)
    plot_multiple_normal_distributions(means=P[:, 0], stds=P[:, 1], i
```