

Trabajo Práctico 2: Kmeans y Corte de Grafos

Andrey Arguedas Espinoza
2020426569
and12rx12@gmail.com

Heiner León Aguilar
2013006040
heiner@hey.com

Victor A Ortiz Ruiz
8705127
voruiz@gmail.com

1. Introducción

En el presente documento hacemos un resumen de nuestra implementación para el algoritmo de Kmedias, desde la fase de la generación de datos como el algoritmo y la graficación del movimiento de centroides y generación de clusters.

2. Implementación del algoritmo K-medias

1. Genere un conjunto de datos $X \in \mathbb{R}^{D \times N}$ con $D = 2$ y $N = 200$, de $K = 2$ etiquetas, donde:
 - a) 100 observaciones correspondan a un *cluster* y las otras 100 a otro, generados por una VA Gaussiana con dos medias $\vec{\mu}_1$, $\vec{\mu}_2$ y matrices de covarianza Σ_1 y Σ_2 (escoja los valores de forma que haya covarianzas no nulas, y varianzas diferentes en cada dimension, haciendo que los datos roten).
 - 1) Genere un caso en el que las medias y las covarianzas faciliten la separación de los datos, X_1 .
 - 2) Genere otro caso en el que las medias y covarianzas generen datos traslapados, X_2 .
 - b) Grafique las muestras, con las etiquetas correctas y sin las etiquetas, usando una simbología de su preferencia.

2.1. Creación de los datos

Para generar los datos utilizaremos la función `createData`, a la cual le agregaremos la funcionalidad de usar matriz de covarianzas distintas.

```
def createData(numberSamplesPerClass = 2, mean1 = [2, 2], mean2 = [26, 26], stds1 = [3, 3], stds2 = [2, 1], covarianceMatrix1 = torch.eye(2), covarianceMatrix2 = torch.eye(2)):  
    """  
    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  
    """  
    means = torch.zeros(2)  
    # Ones to concatenate for bias  
    ones = torch.ones(numberSamplesPerClass, 1)  
    means[0] = mean1[0]  
    means[1] = mean1[1]  
    # Covariance matrix creation with identity  
    #covarianceMatrix = torch.eye(2)  
    covarianceMatrix1[0, 0] = stds1[0]  
    covarianceMatrix1[1, 1] = stds1[1]  
    samplesClass1 = createDataOneClass(means, covarianceMatrix1, numberSamplesPerClass)  
    means[0] = mean2[0]  
    means[1] = mean2[1]  
    covarianceMatrix2[0, 0] = stds2[0]  
    covarianceMatrix2[1, 1] = stds2[1]  
    samplesClass2 = createDataOneClass(means, covarianceMatrix2, numberSamplesPerClass)  
    # Concatenates the ones for the bias  
    samplesClass1Bias = torch.cat((ones, samplesClass1), 1)  
    samplesClass2Bias = torch.cat((ones, samplesClass2), 1)  
    samplesAll = torch.cat((samplesClass1, samplesClass2), 0)  
    plt.scatter(samplesClass1[:, 0], samplesClass1[:, 1])  
    plt.scatter(samplesClass2[:, 0], samplesClass2[:, 1], marker = 'x')  
    plt.show()  
    #Create samples without bias  
    samplesAll = torch.cat((samplesClass1, samplesClass2), 0)  
  
    #Create targets  
    targetsClass1 = torch.ones(numberSamplesPerClass, 1)  
    targetsClass2 = torch.zeros(numberSamplesPerClass, 1)  
    targetsAll = torch.cat((targetsClass1, targetsClass2), 0)  
  
    return (targetsAll, samplesAll)
```

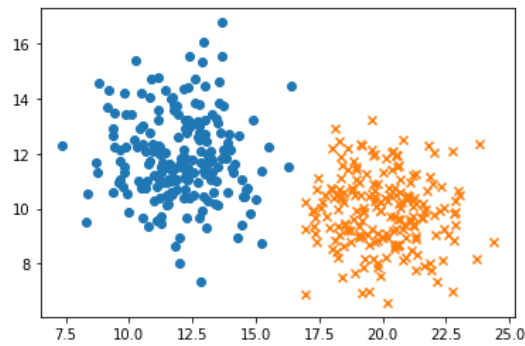


Figura 1: Clusters separados

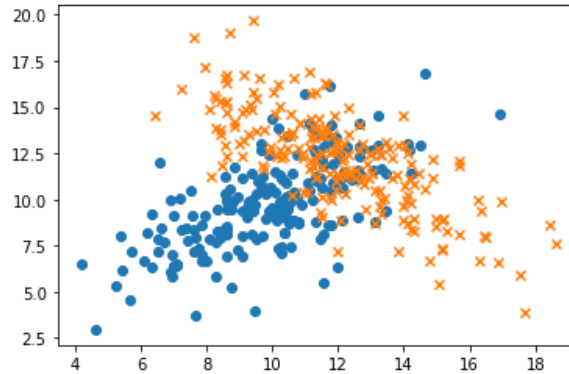


Figura 2: Clusters con direcciones opuestas

1. Implemente alguno de los algoritmos de aprendizaje no supervisado vistos en clase (K-medias), para realizar la clasificación de los dos conjuntos X_1 y X_2 . **Hagalo de forma completamente matricial usando Pytorch.**
 - a) Realice una implementación usando la distancia Euclidiana para medir la disimilitud de las muestras.
 - b) Proponga e implemente el algoritmo usando otra distancia, como la distancia L1 o de Mahalanobis.
 - c) Reporte los resultados midiendo la tasa de aciertos (media y desviación estándar), con gráficas que muestren las etiquetas generadas por el algoritmo. Realicelo para 10 corridas.
 - 1) Reporte además el tiempo de ejecución promedio y la desviación estándar para cada variante con cada distancia.

2.2. Implementación de Kmedias

Primero generamos las funciones para cálculos de distancias tanto euclidianas como de Manhattan: Seguido podemos implementar la función del kmedias la cual consiste en el movimiento de los centroides para la generación de clusters.

```
def build_distance_matrix_with_euclidean_distance(observation, centroids):
    # Here we are applying the mathematical operations of the Euclidean distance but operating through matrices
    observation_matrix = observation.repeat(centroids.shape[0],1)
    distances_matrix = centroids - observation_matrix
    distances_matrix_sum = torch.sum(distances_matrix, dim=1)
    distances_matrix_powed = torch.pow(distances_matrix_sum, 2)
    distances_matrix_final = torch.sqrt(distances_matrix_powed)
    return distances_matrix_final

def build_distance_matrix_with_manhattann_distance(observation, SamplesAll):
    # Here we are applying the mathematical operations of the Manhattan distance but operating through matrices
    observation_matrix = observation.repeat(SamplesAll.shape[0],1)
    distances_matrix = SamplesAll - observation_matrix
    distances_matrix = torch.abs(distances_matrix)
    distances_matrix_final = torch.sum(distances_matrix, dim=1)
    return distances_matrix_final
```

Figura 3: Implementación distancias

```
def KmeansImplementation(data, num_clusters, num_iterations, is_euclidian = True):
    #Primero debemos escoger al azar num_clusters centroides
    indices = torch.randint(0, data.shape[0], (num_clusters,))
    centroids = data[indices]

    #cluster_association lo usaremos como una columna que haremos append al dataset para poner el cluster al que pertenece cada observacion
    cluster_association = torch.zeros(data.shape[0], 1)

    for iter in range(0,num_iterations):
        plot_centroids_clusters(centroids, data, False)

        #Creamos matriz de pesos
        weight_matriz = torch.zeros(data.shape[0], centroids.shape[0])

        w_idx = 0
        for observation_point in data :
            distances = build_distance_matrix_with_euclidean_distance(observation_point, centroids) if is_euclidian == True else build_distance_matrix_with_manhattann_distance(observation_point, centroids)
            # El centroide mas cercano al punto de observacion
            closest_centroid_observation = torch.argmin(distances).item()
            # Ponemos un 1 en la columna del centroide mas cercano
            weight_matriz[w_idx, closest_centroid_observation] = 1.0
            w_idx += 1

        # Set first centroid position
        set_new_centroid_position(data, weight_matriz, centroids, 0, cluster_association)
        #Set second centroid position
        set_new_centroid_position(data, weight_matriz, centroids, 1, cluster_association)

    final_clustered_data = torch.cat((cluster_association, data), 1)
    plot_centroids_clusters(centroids, final_clustered_data)
    return final_clustered_data
```

Figura 4: Implementación del Kmedias

Seguido de eso corremos el algoritmo sobre el dataset cuyos datos estan bien separados y luego sobre el de direcciones opuestas:

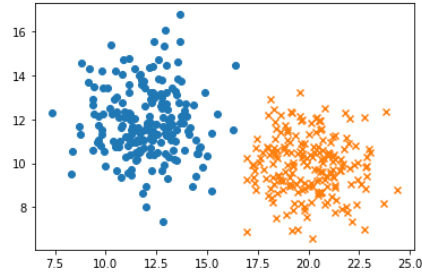


Figura 5: Clusters separados sobre el que se ejecutará el algoritmo

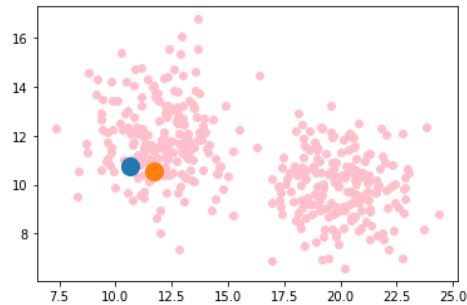


Figura 6: Primera iteración donde los centroides caen al azar en el dataset

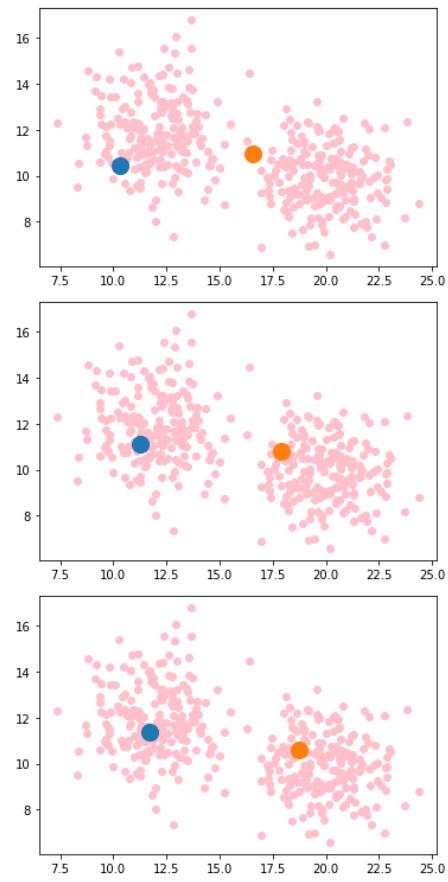


Figura 7: Sigüientes iteraciones del kmedias

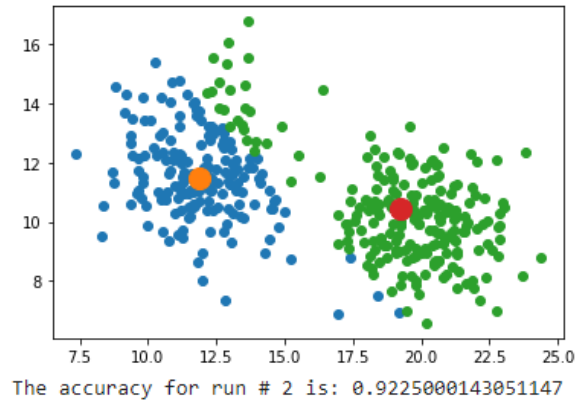


Figura 8: Resultado final de Kmedias

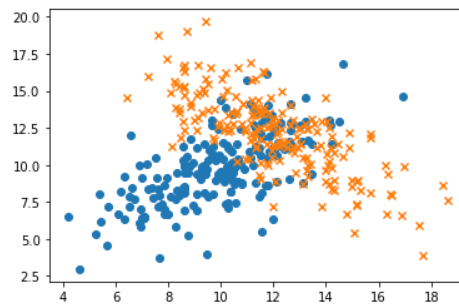


Figura 9: Clusters con direcciones opuestas

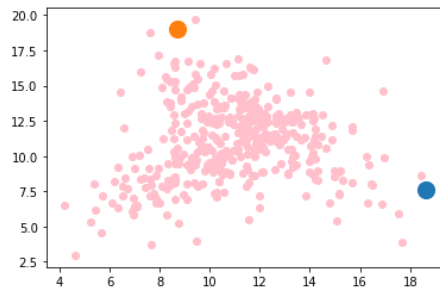


Figura 10: Primera iteración donde los centroides caen al azar en el dataset

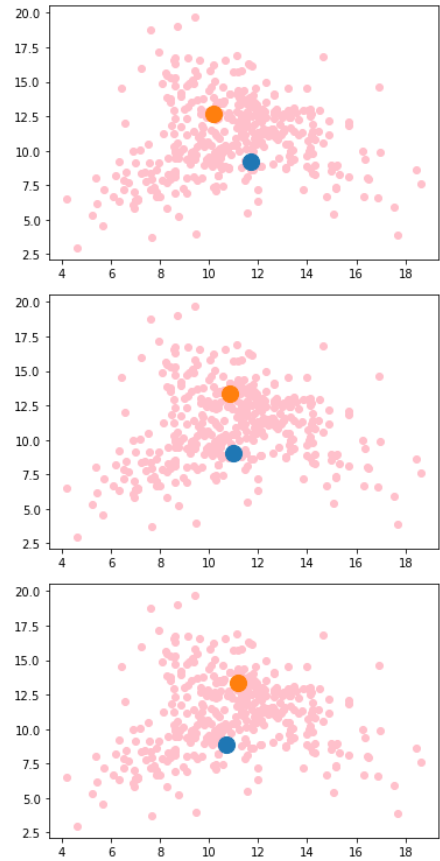
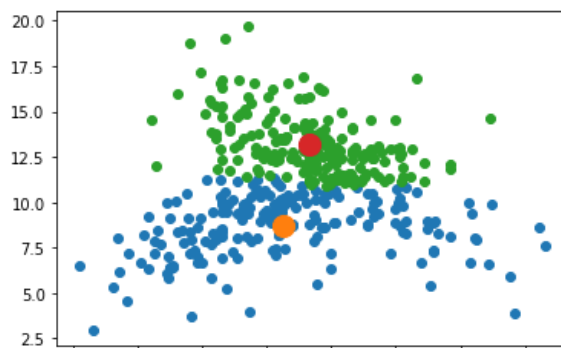


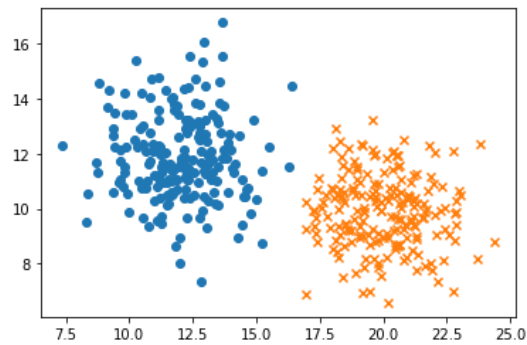
Figura 11: Sigüientes iteraciones del kmedias



The accuracy for run # 4 is: 0.7124999761581421

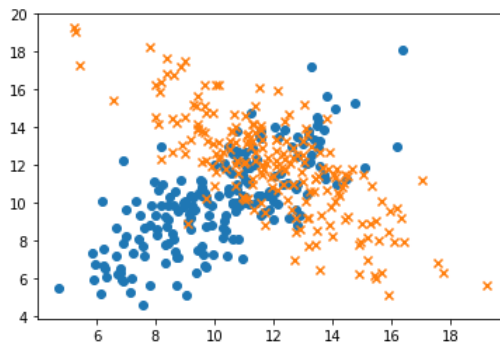
Figura 12: Resultado final de Kmedias

2.3. Resultados promedios finales



```
The accuracy for run # 0 is: 0.9049999713897705
The accuracy for run # 1 is: 0.8675000071525574
The accuracy for run # 2 is: 0.9049999713897705
The accuracy for run # 3 is: 0.8999999761581421
The accuracy for run # 4 is: 0.9049999713897705
The accuracy for run # 5 is: 0.9049999713897705
The accuracy for run # 6 is: 0.9049999713897705
The accuracy for run # 7 is: 0.9024999737739563
The accuracy for run # 8 is: 0.9100000262260437
The accuracy for run # 9 is: 0.8999999761581421
```

```
accuracy mean: 0.9004999816417694
accuracy std: 0.01133577823331307
```



```
The accuracy for run # 0 is: 0.6650000214576721
The accuracy for run # 1 is: 0.6225000023841858
The accuracy for run # 2 is: 0.6725000143051147
The accuracy for run # 3 is: 0.6625000238418579
The accuracy for run # 4 is: 0.6025000214576721
The accuracy for run # 5 is: 0.6600000262260437
The accuracy for run # 6 is: 0.6274999976158142
The accuracy for run # 7 is: 0.6650000214576721
The accuracy for run # 8 is: 0.6625000238418579
The accuracy for run # 9 is: 0.6899999976158142
```

```
accuracy mean: 0.6530000150203705
accuracy std: 0.025293280715493652
```

** Nota: Para la presentación de este documento fueron cortados los movimientos de los centroides por el espacio del documento, los movimientos completos pueden ser vistos en el jupyter notebook.

3. (20 puntos extra) Implementación del algoritmo de Corte de grafos

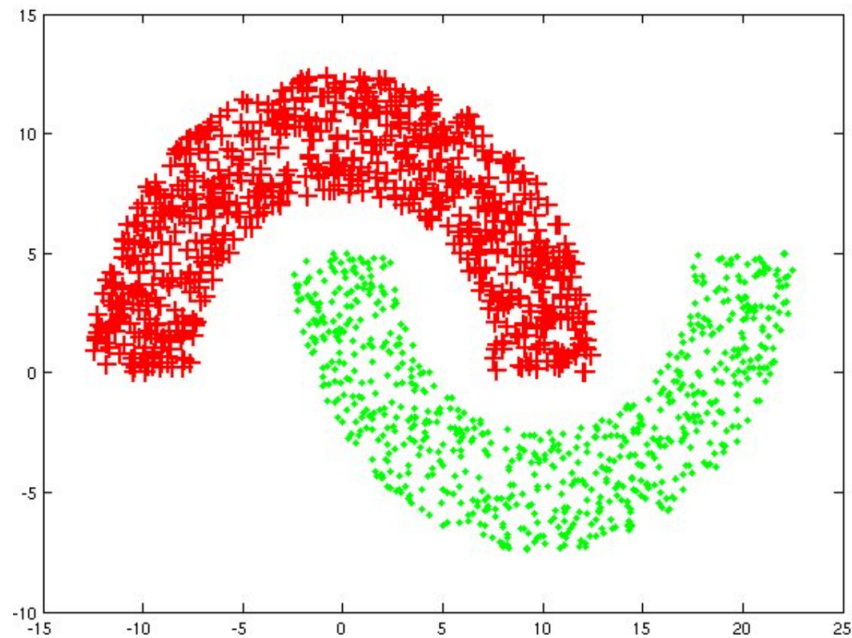


Figura 13: Datos en configuracion de *dos medias lunas*.

1. Implemente el algoritmo de corte de grafos visto en clase.
2. Investigue como generar el conjunto de datos conocido como las *dos medias lunas* e implementelo en *pytorch*, como se muestra en la Figura 13.
3. Compruebe el resultado de realizar el agrupamiento usando el algoritmo de corte de grafos, usando el conjunto de datos de las *dos medias lunas*, usando una separacion *grande* y otra *pequena*.
 - a) Grafique los resultados del agrupamiento.
 - b) Reporte la tasa de aciertos respecto a una particion distinta de datos (particion de *test*) generada con la misma distribucion.
 - c) Realice los dos pasos anteriores para el algoritmo de K-medias previamente implementado. Compare y comente los resultados.

3.1. Generación del conjunto de datos para dos medias lunas.

Para efectos de la generación de datos, debemos tener en cuenta que los datos fluyen con dos curvas muy conocidas en la trigonometría. Esas curvas son el equivalente de las funciones Seno y Coseno [1].

La gráfica de la función de Seno se puede apreciar en la figura 14, donde se muestra tanto el círculo trigonométrico como la gráfica con los valores desde $x = 0$ hasta $x = 2\pi$.

Dentro de la función python, que crea una de las media lunas, se utiliza esta función para la generación de datos:

```
outer_circ_y = torch.sin(torch.linspace(0, math.pi, samples_out))
inner_circ_y = 1 - torch.sin(torch.linspace(0, math.pi, samples_in)) - 0.5
```

Similarmente, la función del coseno se puede apreciar en la figura 15, que al igual que la del seno, muestra el círculo trigonométrico y los valores que toma desde $x = 0$ hasta $x = 2\pi$.

Esta función también es utilizada para la generación de datos, como muestra el siguiente código:

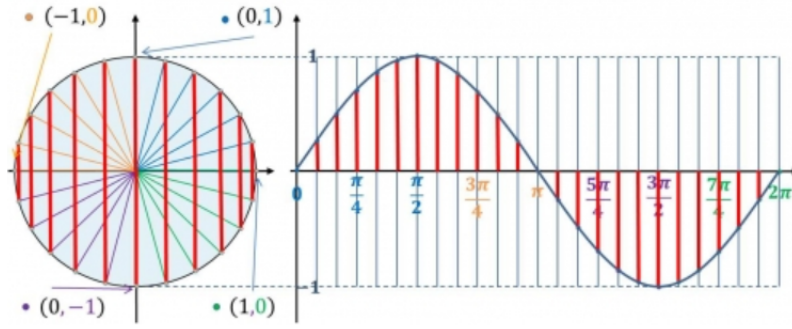


Figura 14: Gráfica de la *función de Seno*. [1]

```
outer_circ_x = torch.cos(torch.linspace(0, math.pi, samples_out))
inner_circ_x = 1 - torch.cos(torch.linspace(0, math.pi, samples_in))
```

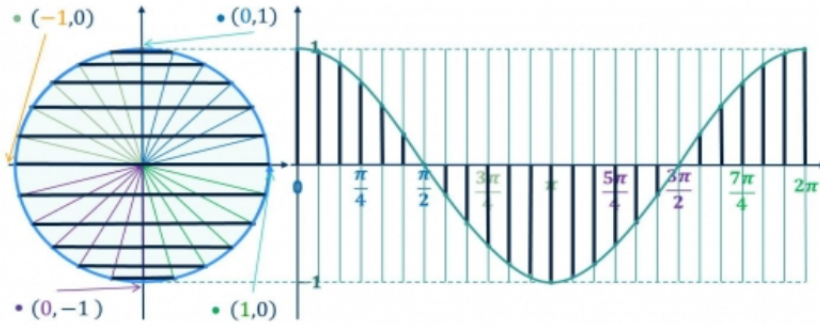


Figura 15: Gráfica de la *función de Coseno*. [1]

Una vez que se tiene el conjunto de puntos para el eje x y el eje y , tenemos que juntar los datos. Primero debemos juntar los puntos externos con los internos del eje x , lo cual lo logramos concatenando los tensores utilizando la dimensión de columnas ($\text{dim}=1$) y luego hacemos lo mismo con los datos para el eje y .

Pero ahora tenemos dos tensores de tipo fila, con la cantidad de datos que se pidió en el parametro de entrada "*Samples*".

Para tener una matriz, volvemos a concatenar utilizando los dos vectores fila resultante, pero en la dimensión de las filas ($\text{dim}=0$), con lo cual obtenemos una matriz $X \in \mathbb{R}^{2 \times D}$. Para poder darle vuelta a las dimensiones, utilizamos la función transpuesta de pytorch. Una vez que se hace transpuesta, obtenmos la matriz deseada $X \in \mathbb{R}^{D \times 2}$.

```
X = torch.cat((
    torch.cat((torch.unsqueeze(outer_circ_x,0),
               torch.unsqueeze(inner_circ_x,0)), 1),
    torch.cat((torch.unsqueeze(outer_circ_y,0),
               torch.unsqueeze(inner_circ_y,0)), 1)
),
0
).T
```

Ahora nos queda crear el vector de las etiquetas para los dos conjuntos de datos creados con las funciones seno y coseno. Para efectos de las etiquetas, nos aseguramos de tener valores enteros utilizando `dtype=torch.uint8`.

```
y = torch.cat((
    torch.zeros(samples_out, dtype=torch.uint8),
    torch.ones(samples_in, dtype=torch.uint8)), -1)
```


El resultado obtenido se puede apreciar en la figura 16.

Demostración

```
1 X, y = make_half_moons((200, 200) , noise=0.1)
2
3 print('X: ', X.shape)
4 print('y: ', y.shape)
5
6 plt.scatter(X[y==0, 0], X[y==0, 1], linewidths = 2, color = "blue", marker='o')
7 plt.scatter(X[y==1, 0], X[y==1, 1], linewidths = 2, color = "red", marker='+')
8 plt.show()
9
```

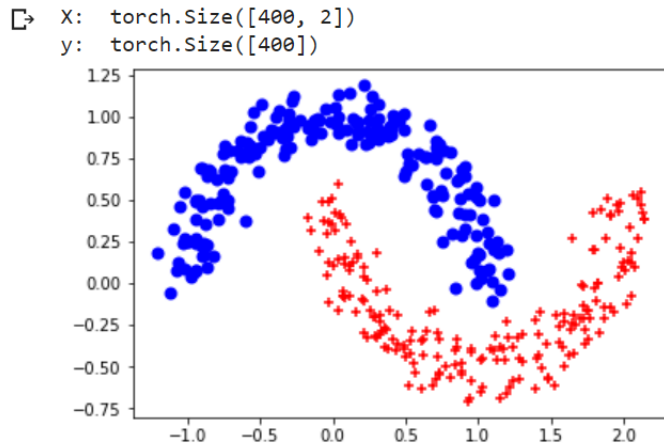


Figura 16: Demostración de las dos *medias lunas* generadas.

Referencias

- [1] Gráficas funciones trigonométricas. Matematicas Puerto Rico — L2DJ Temas de Matemáticas, Inc. [Online]. Available: <http://matematicaspr.com/l2dj/blog/graficas-funciones-trigonometricas>