Trabajo Práctico 0: K-vecinos más cercanos

Andrey Arguedas Espinoza
2020426569
and12rx12@gmail.com

Heiner León Aguilar
2013006040
heiner@hey.com

Victor A Ortiz Ruiz
8705127
voruiz@gmail.com

1. Introducción

Para este práctica realizamos una implementación del algoritmo de K-vecinos más cercanos[1], el cuál será ejecutado sobre datos aleatoriamente generados a través de centroides definidos y variaciones estándares. Para generar los datos, se usa la función **createData**.

Para objetivos de la práctica, los datos generados pertenecen a 2 clases distintas como se puede observar en figura 1 usando las funciones de las figuras 2 y 3. Los elementos naranjas representan la clase 0 y los azules la clase 1

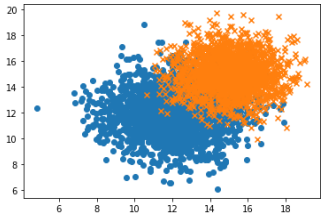


Figura 1: Ejemplo de una generación de datos.

```python
def createData(numberSamplesPerClass = 2, mean1 = [2, 2], mean2 = [26, 26], stds1 = [3, 3], stds2 = [2, 1]):
    """
    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)
    covarianceMatrix[0, 0] = stds1[0]
    covarianceMatrix[1, 1] = stds1[1]
    samplesClass1 = createDataOneClass(means, covarianceMatrix, numberSamplesPerClass)
    means[0] = mean2[0]
    means[1] = mean2[1]
    covarianceMatrix[0, 0] = stds2[0]
    covarianceMatrix[1, 1] = stds2[1]
    samplesClass2 = createDataOneClass(means, covarianceMatrix, 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 2: Función que genera los datos que se utilizarán en el modelo.

```
'''
Creates data with gaussian distribution
'''
def createDataOneClass(means, covarianceMatrix, numberSamples):
    # Inits the bi gaussian data generator
    multiGaussGenerator = multivariate_normal.MultivariateNormal(means, covarianceMatrix)
    # Takes the samples
    samples = multiGaussGenerator.sample(torch.Size([numberSamples]))

    return samples
```

Figura 3: Función que crea los elementos para una clase específica.

2. Definición del algoritmo KNN

El algoritmo de K-vecinos mas cercanos es un algoritmo de aprendizaje automático supervisado muy popular por su simplicidad. Dado un conjunto de datos en su forma matricial, con la matriz $X_{train} \in \mathbb{R}^{N \times D}$ y un arreglo de etiquetas $\vec{t} \in \mathbb{R}^N$:

$$X_{train} = \begin{bmatrix} - & \vec{x}_1 & - \\ & \vdots & \\ - & \vec{x}_{N_{train}} & - \end{bmatrix} \quad \vec{t} = \begin{bmatrix} t_1 \\ \vdots \\ t_{N_{train}} \end{bmatrix}$$

Para cada dato $\vec{t}_i^{(test)} \in X_{test}$ en un conjunto de datos de prueba o evaluación $X_{test} \in \mathbb{R}^{N_{test} \times D}$:

$$X_{test} = \begin{bmatrix} - & \vec{x}_1 & - \\ & \vdots & \\ - & \vec{x}_N & - \end{bmatrix}$$

se crea un conjunto de datos X_{KNN} con los K vecinos mas cercanos de la observación $\vec{x}_j$ en el conjunto de datos X_{train}, donde cada observación $\vec{x}_i \in X_{KNN}$ cumple que:

$$X_{KNN} = arg_{Kmin} min_j (d(\vec{x}_i^{(test)} - \vec{x}_j))$$

Luego de tomar los K vecinos mas cercanos de la observación $\vec{x}_i^{(test)}$ se realiza una votación según las etiquetas correspondientes $\vec{t}_i^{(test)}$, y se toma como estimación de la etiqueta $\widetilde{t}_j$ la etiqueta mas votada.

2.1. Funciones a definir

En este ejercicio debemos definir las siguientes funciones:

```
def evaluate_k_nearest_neighbors_observation(data_training, labels_training, test_observation, K = 3, is_euclidian = True):
    #TODO
    return t_estimated

def evaluate_k_nearest_neighbors_test_dataset(data_training, labels_training, test_dataset, K = 7, is_euclidian = True):
    #TODO

def calculate_accuracy(test_estimations, test_labels):
    #TODO
```

Figura 4: Funciones para desarrollar.

2.2. Implementación de KNN

1. **(50 puntos)** Implemente el algoritmo de K-vecinos mas cercanos con la posibilidad de usar la distancia euclidiana y la de Manhattan en la función $d(\overrightarrow{x}_i - \overrightarrow{x}_j)$.

   a) Realice la implementacion de forma completamente matricial, para cada observacion $\overrightarrow{x}_i^{(\text{test})}$ *evaluate_k_nearest_neighbors_observation(data_training, labels_training, test_observation, K = 7, is_euclidian = True)* (**No use ciclos for**).

   1) Para ello use funcionalidades de *pytorch* como repeat, mode, sort, etc.
   2) *is_euclidian* indica si se usara la distancia Euclidiana o la de Manhattan de lo contrario. K corresponde a la cantidad de vecinos a evaluar

Primero realizamos la función que realizará la concatenacion de las distancias de los vectores contra el punto de observación, esta distancias se añadiran a la matriz copia para poder manipularla más facilmente, con las distancias podremos ordenar para obtener los vecinos más cercanos del punto de observación ("test-observation") y hacer la votación de a que clase pertenece el punto de observación.

```
def evaluate_k_nearest_neighbors_observation(data_training, labels_training, test_observation, K = 3, is_euclidian = True):
    # We clone the data for better handling
    copy_data_training = torch.clone(data_training)
    # Create a column full of ones to append to the data training
    ones = torch.ones(data_training.shape[0], 1)
    # We append the column of ones
    distanced_data_training = torch.cat((copy_data_training, ones, ), 1)

    # Build the distances matrix depending on the algorithm (Euclidean or Manhattan)
    distance_matrix = build_distance_matrix_with_euclidean_distance(test_observation, data_training) if is_euclidian == True else build_distance_matrix_with_manhattann_distance(test_observation, data_training)

    # We update the ones in the column with the distances, so each row will have the format [x, y, distance_with_test_observation]
    distanced_data_training[:, 2].mul_(distance_matrix)

    # Sorts training points on the basis of distance
    sorted_values, sorted_index = torch.sort(distanced_data_training[:, 2])

    # Here we have the k neighbors
    k_neighbors_values = sorted_values[:K] # selects k-nearest neighbors
    k_neighbors_index = sorted_index[:K] # selects k-nearest neighbors

    # Now let's vote
    k_labels = labels_training[k_neighbors_index]
    #Here we get the most common category of the K nearest neighbors
    t_estimated = torch.mode(k_labels, 0).values.item()
    return t_estimated
```

Figura 5: Función evaluate_k_nearest_neighbors_observation.

Podemos observar el uso de funciones REPEAT, SORT, MODE, CAT, ONES...

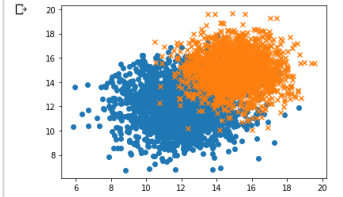
Seguidamente delegamos la construcción de las matrices de distancias a otras funciones que se encargaran de operar mediante matrices las distancias euclidiana y de Manhattan.

```
[ ]  def build_distance_matrix_with_euclidean_distance(observation, SamplesAll):
         # Here we are applying the mathematical operations of the Euclidean distance but operating through matrices
         observation_matrix = observation.repeat(SamplesAll.shape[0],1)
         distances_matrix = SamplesAll - observation_matrix
         distances_matrix_sum = torch.sum(distances_matrix, dim=1)
         distances_matrix_powed = torch.pow(distances_matrix_sum, 2)
         distances_matrix_sqrt = torch.sqrt(distances_matrix_powed)
         distances_matrix_final = np.round(distances_matrix_sqrt, 3)
         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_sum = torch.sum(distances_matrix, dim=1)
         distances_matrix_final = np.round(distances_matrix_sum, 3)
         return distances_matrix_final
```

Figura 6: Funciones para distancias euclidiana y Manhattann.

A continuación probamos la función para una sola observación :

```
"""
Main function
"""
def main():
    (TargetsAll, SamplesAll) = createData(numberSamplesPerClass = 2000, mean1 = [12,12], mean2 = [15, 15], stds1 = [3, 3], stds2 = [2, 2])

    test_observation_1 = SamplesAll[0, :].unsqueeze(0)

    estimatedValue = evaluate_k_nearest_neighbors_observation(SamplesAll, TargetsAll, test_observation_1)
    print("The category value for", test_observation_1, "is------->", estimatedValue)

main()
```




```
The category value for tensor([[14.3342,  9.9424]]) is-------> 1.0
```

Figura 7: Función Main/Principal.

Como podemos observar nuestro modelo categoriza la observación [x: 14.3342, y: 9.9424] como parte de la categoría 1, la cual son los elementos azules del gráfico.

2.3. Resultados de KNN para todas las observaciones del dataset

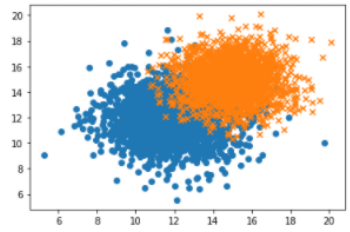
1. **(10 puntos)** Para todo el conjunto de datos X_{test} implemente la función *evaluate_k_nearest_neighbors_test_dataset(data_training, labels_training, test_dataset, K = 3, is_euclidian = True)*, la cual utilice la función previamente construida *evaluate_k_nearest_neighbors_observation* para calcular el arreglo de estimaciones $\overrightarrow{\overrightarrow{t}}$ para todos los datos en X_{test}.

2. **(10 puntos)** Implemente la función *calcular_tasa_aciertos* la cual tome un arreglo de estimaciones $\overrightarrow{\overrightarrow{t}}$ y un arreglo de etiquetas $\overrightarrow{x}_i^{(\text{test})}$ y calcule la tasa de aciertos definida como $\frac{c}{N}$ donde c es la cantidad de estimaciones correctas. (**No use ciclos for**).

```python
def evaluate_k_nearest_neighbors_test_dataset(data_training, labels_training, test_dataset, K = 7, is_euclidian = True):
    register = 0
    observation_estimates = torch.ones(data_training.shape[0], 1)
    for sample_point in test_dataset:
        observation_estimate = evaluate_k_nearest_neighbors_observation(data_training, labels_training, sample_point, K, is_euclidian)
        observation_estimates[register, 0].mul_(observation_estimate)
        register += 1
    return observation_estimates

def calculate_accuracy(test_estimations, test_labels):
    total_amount_samples = test_labels.shape[0]
    total_amount_right_estimations = (test_estimations == test_labels).sum()
    return (total_amount_right_estimations / total_amount_samples).item()
```

Figura 8: Funciones para la evaluación de estimaciones.

```python
"""
Main function
"""
def main():
    (TargetsAll, SamplesAll) = createData(numberSamplesPerClass = 2000, mean1 = [12,12], mean2 = [15, 15], stds1 = [3, 3], stds2 = [2, 2])

    print("Testing KNN with K = 7 and Euclidean distance")
    start_time = time.time()
    test_estimations_all = evaluate_k_nearest_neighbors_test_dataset(SamplesAll, TargetsAll, SamplesAll, 7, is_euclidian = True)

    # Calculate accuracy
    accuracy = calculate_accuracy(test_estimations_all, TargetsAll)
    print("Accuracy is -> :", accuracy)

    elapsed_time = time.time() - start_time

    print("Time duration :", elapsed_time, "seconds")


    print("Testing KNN with K = 7 and Manhattan distance")
    start_time = time.time()
    test_estimations_all_manhattan = evaluate_k_nearest_neighbors_test_dataset(SamplesAll, TargetsAll, SamplesAll, 7, is_euclidian = False)
    # Calculate accuracy
    accuracy = calculate_accuracy(test_estimations_all_manhattan, TargetsAll)
    print("Accuracy is -> :", accuracy)

    elapsed_time = time.time() - start_time

    print("Time duration :", elapsed_time, "seconds")


main()
```




```
Testing KNN with K = 7 and Euclidean distance
Accuracy is -> : 0.9212499856948853
Time duration : 3.165518283843994 seconds
Testing KNN with K = 7 and Manhattan distance
Accuracy is -> : 0.9132500290870667
Time duration : 3.1343042850494385 seconds
```

Figura 9: Resultados de ejecución con distancia euclidiana y Manhattan con K = 7.

Obtenemos que la diferencia entre la distancia euclidiana y la Manhattan son prácticamente iguales tanto en tasa de aciertos como en tiempo de duración.

2.4. Uso de funciones de partición

1. **(20 puntos)** Usando las funciones de partición de datos del paquete sklearn necesarias, implemente la partición de datos del conjunto de datos original X para crear las particiones X_{train} y X_{test}. Cree 10 particiones distintas, para ejecutar 10 veces el algoritmo. Use N=4000 (2000 observaciones por clase).

   a) Utilice 70 % de los datos para entrenamiento y el resto para prueba.

   b) Calcule la tasa de aciertos para las 2 configuraciones (distancia ℓ_1 y ℓ_2) probadas, usando X_{train} para entrenamiento y X_{test} para prueba. Reporte los resultados en una tabla, junto con su media y desviación estándar y coméntelos.

   c) Calcule además el tiempo de ejecución por corrida para cada configuración. Reporte los resultados en una tabla, junto con su media y desviación estándar y coméntelos. Cual distancia resulto mas eficiente? Hay algún costo en cuanto a la tasa de aciertos? Explique el porque de la diferencia en la tasa de aciertos, si la hay.

```
(TargetsAll, SamplesAll) = createData(numberSamplesPerClass = 2000, mean1 = [12,12], mean2 = [15, 15], stds1 = [3, 3], stds2 = [2, 2])

for i in range(10):
  X_train, X_test, y_train, y_test = model_selection.train_test_split(SamplesAll, TargetsAll, train_size=0.7,test_size=0.3)

  print("Testing KNN with K = 7 and Euclidean distance for partition #", i)
  start_time = time.time()
  test_estimations_all_euclidean = evaluate_k_nearest_neighbors_test_dataset(X_train, y_train, X_test, 7, is_euclidian = True)
  #calculate accuracy
  accuracy = calculate_accuracy(test_estimations_all_euclidean[:y_test.shape[0]], y_test)
  print("Accuracy of partition # ", i," with Euclidean distance is: ", accuracy)

  # your code
  elapsed_time = time.time() - start_time

  print("Time duration :", elapsed_time, "seconds")

  print("Testing KNN with K = 7 and Manhattan distance")
  start_time = time.time()
  test_estimations_all_manhattan = evaluate_k_nearest_neighbors_test_dataset(X_train, y_train, X_test, 7, is_euclidian = False)
  #calculate accuracy
  accuracy = calculate_accuracy(test_estimations_all_manhattan[:y_test.shape[0]], y_test)
  print("Accuracy of partition # ", i," with Manhattan distance is: ", accuracy)

  # your code
  elapsed_time = time.time() - start_time

  print("Time duration :", elapsed_time, "seconds")

  print("--------------------------")
```

Figura 10: Funciones de partición.

Configuración	Tasa de aciertos (media)	Tasa de aciertos (desviación estándar)
Distancia euclidiana	**0.9049999714**	**0.006188911866**
Distancia de Manhattan	0.90291664	0.007414133932

La diferencia entre ambas configuraciones es prácticamente nula con el dataset. Una razón puede ser debido a que los datos están sumamente cerca entre ellos, haciendo que la diferencia de los valores de distancia no sea significativa. Intuitivamente se podría asumir que un punto va a tener menor distancia a otro usando un método, pero esto va a afectar la distancia de otros puntos, como aumentando su distancia, resultando que el promedio de valores dé similar.

Un experimento para trabajar a futuro sería con un conjunto de datos más disperso, y con los clusters más separados, podríamos observar mayor diferencias entre las dos configuraciones usadas.

Configuración	Tiempo en segs (media)	Tiempo en segs (desviación estándar)
Distancia euclidiana	0.7635270357	0.1019599702
Distancia de Manhattan	**0.7491422892**	**0.03359455466**

Con respecto a los tiempos, la distancia euclidiana es la que representa un mayor tiempo y puede deberse a tener que hacer un calculo extra comparado a la de Manhattan. De igual forma, la diferencia también es mínima. Una razón es que las operaciones utilizadas por ambas distancias son comunes y suelen estar optimizadas a nivel de hardware.

Referencias

[1] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.