TEC | Tecnológico de Costa Rica

Escuela de Ingeniería en Computación
Ingeniería en Computación
IC6200 - Inteligencia Artificial

KNN Algorithm:

Sebastián Bogantes Rodríguez
sebasbogantes6@estudiantec.cr
2020028437

Rohi Prendas Regalado
rd740112@estudiantec.cr
2019052258

Emmanuel López Ramírez
emma_1399@estudiantec.cr
2018077125

Alajuela, Costa Rica
8 de septiembre 2024

Índice general

Capítulo 1

Introducción

En este trabajo práctico se abordará el problema de la clasificación supervisada. Este problema requiere el uso de herramientas computacionales para la visualización y solución de funciones matemáticas complejas, además de la implementación de algoritmos en entornos de programación.

El **(K-NN)**, uno de los algoritmos de clasificación supervisada más utilizados en la ciencia de datos. Este algoritmo es conocido por su simplicidad y eficacia en la clasificación de datos multidimensionales. En este caso, se estudiarán las diferentes métricas de distancia (Euclidiana, Manhattan e Infinito) y se evaluará el rendimiento del algoritmo en un conjunto de datos generado artificialmente.

Este trabajo no solo permite comprender los conceptos matemáticos subyacentes en el análisis multivariable, sino también su aplicación directa en la resolución de problemas prácticos mediante el uso de herramientas computacionales como Pytorch. Los resultados obtenidos a partir de las simulaciones y la implementación del algoritmo proporcionarán una visión clara de las fortalezas y limitaciones de cada enfoque aplicado en este contexto.

Capítulo 2

Desarrollo

2.1. Implementación del algoritmo K-vecinos mas cercanos

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 $X_{train} \in R^{N \times D}$ y un arreglo de etiquetas $\vec{t} \in R^N$:

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

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

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

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.

(40 puntos) Implemente el algoritmo de K-vecinos más cercanos con la posibilidad de usar la distancia euclidiana, de Manhattan e Infinito en la función $d(\vec{x}_i - \vec{x}_j)$.

2.1.1. Creación de datos

Para la implementación de K-vecinos más cercanos se generan datos sintéticos aleatorios a través de unas medias y desviaciones estándar que se pasan por parámetro en la función **crearDatos**.

```
from __future__ import print_function
import torch
import numpy as np
import matplotlib.pyplot as plt
import time
from torch.distributions import multivariate_normal

""" Crea los datos que se utilizarán para el entrenamiento con dos clases y sus respectivas medias y desviaciones estándar.
    :param numeroMuestraClase: Número de muestras por clase.
    :param media1: Media para la clase 1.
    :param media2: Media para la clase 2.
    :param dv_estandar1: Desviaciones estándar para la clase 1.
    :param dv_estandar2: Desviaciones estándar para la clase 2.
    :return: labels_training, data_training: Etiquetas generadas y muestras de datos.
"""
def crearDatos(numeroMuestraClase=2, media1=[12, 12], media2=[20, 20], dv_estandar1=[3, 3], dv_estandar2=[2, 2]):
    # Class 1
    clase_media1 = torch.tensor(media1)
    matriz_covarianza_clase1 = torch.diag(torch.tensor(dv_estandar1))
    muestrasClase1 = crearDatosClase1(clase_media1, matriz_covarianza_clase1, numeroMuestraClase)
    # Class 2
    clase_media2 = torch.tensor(media2)
    matriz_covarianza_clase2 = torch.diag(torch.tensor(dv_estandar2))
    samplesClass2 = crearDatosClase1(clase_media2, matriz_covarianza_clase2, numeroMuestraClase)
    # Combine both classes
    data_training = torch.cat((muestrasClase1, samplesClass2), 0)
    # Create labels: 1 for class 1 and 0 for class 2
    clase1 = torch.ones(numeroMuestraClase, 1)
    clase2 = torch.zeros(numeroMuestraClase, 1)
    labels_training = torch.cat((clase1, clase2), 0)

    return labels_training, data_training


```

```python
"""Crea datos para una clase utilizando una distribución normal multivariante.
    :param medias: Vector de medias para la clase (centro de la distribución).
    :param matrizCovarianza: Matriz de covarianza para la clase (define la dispersión de los datos).
    :param numeroMuestras: Número de muestras a generar para la clase.
    :return: Muestras generadas a partir de la distribución normal multivariante.
"""
def crearDatosClase1(medias, matrizCovarianza, numeroMuestras):
    multiGaussGenerator = multivariate_normal.MultivariateNormal(medias.float(), matrizCovarianza.float())
    samples = multiGaussGenerator.sample(torch.Size([numeroMuestras]))
    return samples
```

a) Realice la implementación de forma completamente matricial, para cada observación $\vec{x}_i^{(test)}$ `evaluate_k_nearest_neighbors_observation(data_training, labels_training, test_observation, K = 7, p = 2)` **(Sin ciclos for)**.

- Para ello use funcionalidades de pytorch como `repeat`, `mode`, `sort`, etc.
- p indica el tipo de norma a utilizar. K corresponde a la cantidad de vecinos a evaluar.

2.1.2. a) Implementación matricial sin ciclos `for`

La implementación de esta parte utiliza las funcionalidades de PyTorch, tales como `repeat`, `mode`, y `sort`. Se debe usar la función `evaluate_k_nearest_neighbors_observation` con los parámetros adecuados.

- p indica el tipo de norma a utilizar (Euclidiana, Manhattan, Infinito).
- K corresponde a la cantidad de vecinos a evaluar.

A continuación se presenta el código correspondiente:

```python
    """Esta función evalúa una observación de prueba usando el algoritmo KNN de manera completamente matricial.

        :param data_training: Matriz de datos de entrenamiento (N x d)
        :param labels_training: Vector de etiquetas de entrenamiento (N x 1)
        :param test_observation: Observación de prueba (1 x d)
```

```
        :param K: Número de vecinos más cercanos a considerar
        :param p: Tipo de norma Lp a utilizar
        :return: Etiqueta estimada para la observación de prueba
    """
    def evaluate_k_nearest_neighbors_observation(data_training, labels_training, test_observation, p , K=7):

        # Calcula la matriz de distancias utilizando la norma Lp
        distancias = construir_matriz_distancia(data_training, test_observation, p)

        # Ordena las distancias en orden ascendente y obtiene los índices correspondientes usando torch.sort
        sorted_distancias, sorted_indices = torch.sort(distancias)

        # Selecciona los K vecinos más cercanos
        k_nearest_indices = sorted_indices[:K]

        # Obtiene las etiquetas correspondientes a los K vecinos más cercanos
        k_nearest_labels = labels_training[k_nearest_indices]

        # Vota por la clase mayoritaria entre los vecinos utilizando torch.mode
        etiqueta_escogida = torch.mode(k_nearest_labels, 0).values.item()

        return etiqueta_escogida
```

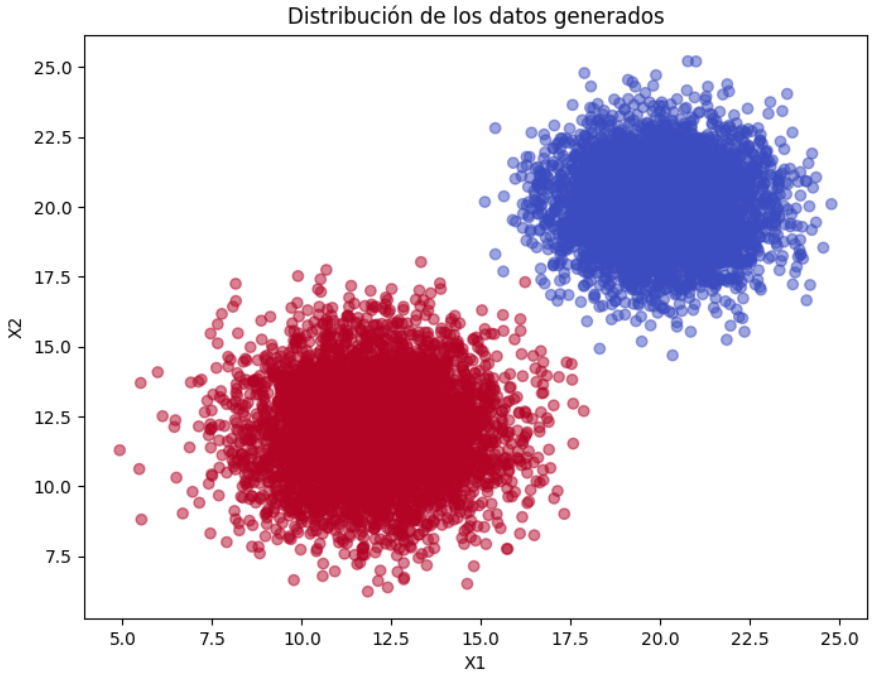
A continuación calculamos la distancia entre una observación de prueba y todas las muestras en data_training usando norma Lp.

```
        """ Calcula la distancia entre una observación de prueba y todas las muestras en data_training
        usando norma Lp.
    :param data_training: Matriz de todas las muestras de entrenamiento (N x d)
    :param test_observation: Observación de prueba (1 x d)
    :param p: Tipo de norma Lp a utilizar
    :return: Vector de distancias (N x 1)
"""
def construir_matriz_distancia(data_training, test_observation, p):
```

```
    # Expandimos la observación de prueba para que coincida con el tamaño de data_training
    test_observation_expanded = test_observation.repeat(data_training.size(0), 1)
    # Calculamos las diferencias absolutas entre las muestras y la observación de prueba
    diferencias = torch.abs(data_training - test_observation_expanded)
    if p == 1:
        # Distancia Manhattan (suma de las diferencias absolutas)
        distancias = torch.sum(diferencias, dim=1)
    elif p == 2:
        # Distancia Euclidiana (suma de cuadrados y raíz cuadrada usando PyTorch)
        distancias = torch.sqrt(torch.sum(torch.pow(diferencias, 2), dim=1))
    elif p == float('inf'):
        # Distancia Chebyshev (máxima diferencia absoluta usando torch.max)
        distancias = torch.max(diferencias, dim=1)[0]
    else:
        # Para otros valores de p, calculamos la norma Lp general utilizando torch.pow
        distancias = torch.pow(torch.sum(torch.pow(diferencias, p), dim=1), 1/p)

    return distancias
```

Y por ultimo probamos la función para una observación en el punto [x: 15.0, y: 17.0] como de la categoría 1 que corresponde al Rojo (0 seria la categoría Azul):




```
------------------------------------------------
Evaluando una observación individual:
La categoría predicha para el punto [[15.0, 17.0]] es: 1.0
```

Figura 2.1: Ejecución del Main: Gráfica de KNN

b) 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 $\vec{t}$ para todos los datos en X_{test}.

2.1.3. b) Evaluación del conjunto de datos de prueba

La función `evaluate_k_nearest_neighbors_test_dataset` implementa el algoritmo K-Nearest Neighbors (KNN) de manera matricial para evaluar un conjunto de observaciones de prueba. Esta función recibe como entrada los datos y etiquetas de entrenamiento, el conjunto de prueba, el número de vecinos más cercanos (K) y el tipo de distancia a utilizar (Euclidiana o Manhattan).

Primero, se selecciona la norma Lp en función del parámetro `is_euclidian`. A continuación, se calculan las distancias entre cada observación de prueba

y las muestras de entrenamiento utilizando una operación matricial. Estas distancias se ordenan y se seleccionan los K vecinos más cercanos. Con las etiquetas correspondientes a estos vecinos, se lleva a cabo una votación para determinar la clase mayoritaria mediante la función `torch.mode`. Finalmente, la función devuelve un vector con las etiquetas estimadas para cada observación del conjunto de prueba.

```
    """Evalúa todas las observaciones del conjunto de prueba usando el algoritmo KNN de manera completamente matricial
    utilizando la distancia Euclidiana (p=2) o la distancia de Manhattan (p=1) según el parámetro is_euclidian.

    :param data_training: Matriz de datos de entrenamiento (N x d)
    :param labels_training: Vector de etiquetas de entrenamiento (N x 1)
    :param test_dataset: Matriz de datos de prueba (M x d)
    :param K: Número de vecinos más cercanos a considerar (por defecto 3)
    :param is_euclidian: True para usar la distancia Euclidiana, False para usar la distancia Manhattan
    :return: Vector de etiquetas estimadas para el conjunto de prueba
"""
def evaluate_k_nearest_neighbors_test_dataset(data_training, labels_training, test_dataset, K = 3, is_euclidian=True):
    # Determinamos la norma Lp según el valor de is_euclidian
    if is_euclidian is True:
        p = 2  # Euclidiana
    elif is_euclidian is False:
        p = 1  # Manhattan
    else:
        p = float('inf')  # Chebyshev (por ejemplo, si is_euclidian = None)

    # Calculamos la distancia entre cada observación de prueba y todas las muestras de entrenamiento
    distancias = torch.cdist(test_dataset, data_training, p=p)
    # Ordenamos las distancias para cada observación de prueba y obtenemos los índices de los vecinos más cercanos
    sorted_distancias, sorted_indices = torch.sort(distancias, dim=1)
    # Seleccionamos los K vecinos más cercanos para cada observación de prueba
    k_nearest_indices = sorted_indices[:, :K]
    # Obtenemos las etiquetas correspondientes a los K vecinos más cercanos
```

```
    k_nearest_labels = labels_training[k_nearest_indices]
    # Votamos por la clase mayoritaria entre los K vecinos para cada observación de prueba
    etiqueta_escogidas, _ = torch.mode(k_nearest_labels, dim=1)

    return etiqueta_escogidas
```

c) Implemente la función `calcular_tasa_aciertos` la cual tome un arreglo de estimaciones $\vec{t}$ y un arreglo de etiquetas $\vec{x}_i^{(test)}$ y calcule la tasa de aciertos definida como $\frac{c}{N}$, donde c es la cantidad de estimaciones correctas. **(Sin ciclos for)**.

2.1.4. c) Tasa de Aciertos

La función **calcular_tasa_aciertos** se encarga de medir el rendimiento de un modelo de clasificación, calculando el porcentaje de predicciones correctas. Para ello, toma dos parámetros de entrada: las predicciones generadas por el modelo (denominadas **estimaciones_test**) y las etiquetas reales correspondientes a los datos de prueba (denominadas **test_labels**).

Primero, la función obtiene el número total de muestras presentes en las etiquetas de prueba para establecer la base de comparación. Luego, realiza una comparación directa entre las predicciones y las etiquetas reales, contando cuántas predicciones coinciden con las etiquetas correctas. Este resultado es la cantidad de predicciones acertadas.

Para calcular la tasa de aciertos, la función divide el número de predicciones correctas por el número total de muestras, obteniendo un valor que va de 0 a 1, donde 1 indica una precisión del 100 %. Finalmente, utiliza ›item()‘ para convertir el resultado a un número flotante estándar de Python, lo que facilita su manejo en otras partes del código. La salida final es la tasa de aciertos o precisión del modelo.

```
    """ Parámetros de entrada:
    :param estimaciones_test: Tensor de las predicciones generadas por el modelo (dimensión N x 1)
    :param test_labels: Tensor con las etiquetas reales de las muestras (dimensión N x 1)
    :return Tasa de aciertos: Un número flotante que representa el porcentaje de predicciones correctas, entre 0 y 1.
"""
# Calcula la tasa de aciertos
def calcular_tasa_aciertos(estimaciones_test, test_labels):
    # Obtiene la cantidad total de muestras en las etiquetas de prueba
```

```
    cantidad_muestras_totales = test_labels.shape[0]
    # Calcula cuántas predicciones fueron correctas comparando
    ↪  estimaciones con las etiquetas reales
    estimaciones_totales_correctas = (estimaciones_test ==
    ↪  test_labels).sum()
    # Calcula la tasa de aciertos dividiendo el número de
    ↪  predicciones correctas entre el total de muestras
    return (estimaciones_totales_correctas /
    ↪  cantidad_muestras_totales).item()
```

Y por ultimo podemos ver una gráfica de una ejecución con las diferentes distancias.

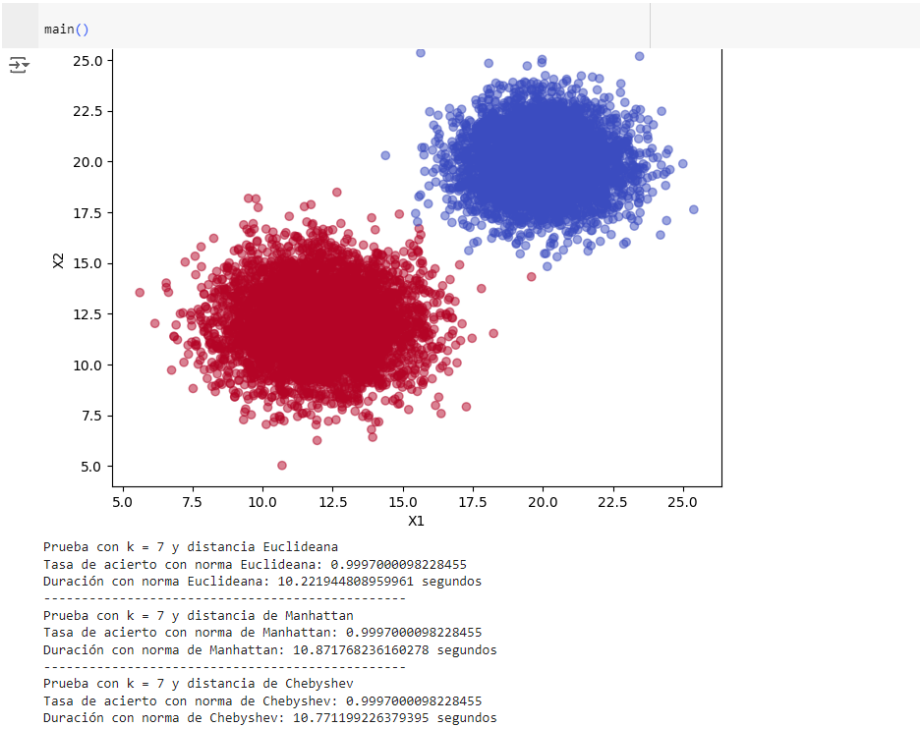


Figura 2.2: Resultado de ejecución con la distancias, euclidiana, manhattan y chebyshev

2. Para un conjunto de datos de $N = 10000$ (5000 observaciones por clase) genere un conjunto de datos con medias $\mu_1 = [12, 12]^T$, $\mu_2 = [20, 20]^T$, y desviaciones estándar $\sigma_1 = [3, 3]^T$, $\sigma_2 = [2, 2]^T$. Grafique los datos y muestre las figuras.

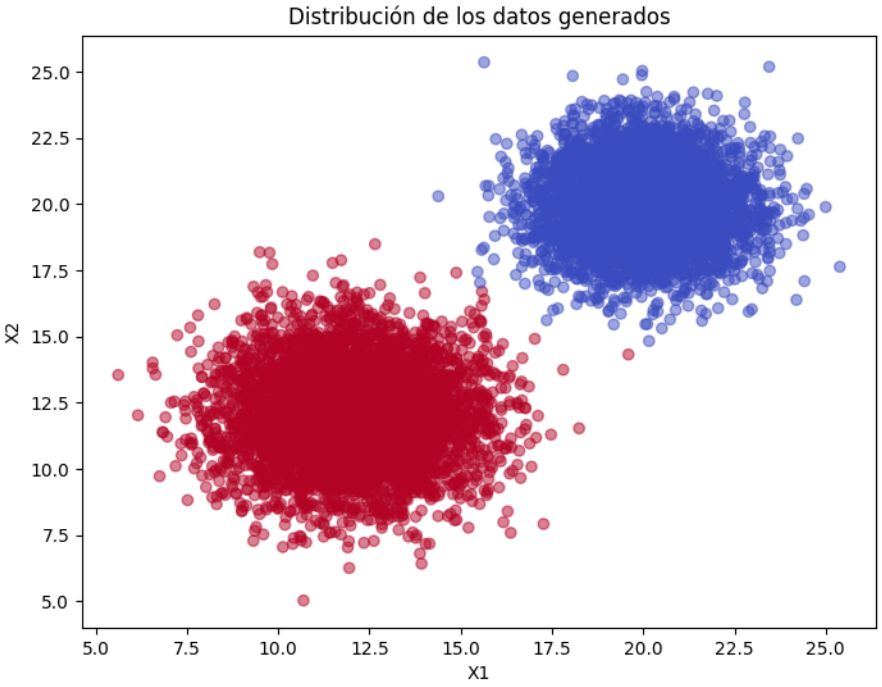


Figura 2.3: Gráfico de los datos generados

3. Compruebe y compare para las dos distancias implementadas, usando el dataset anterior, y $K = 7$:

a) **(20 puntos)** La tasa de aciertos, definida como $\frac{c}{N}$ donde c es la cantidad de estimaciones correctas, usando el mismo conjunto de datos X_{train} como conjunto de prueba X_{test}. Documente los resultados y coméntelos. Puede probar otros valores de medias que faciliten la separabilidad de los datos para facilitar la explicación.

2.1.5. Tasa de Aciertos con el mismo Conjunto de Datos

Aquí se evaluara la tasa de aciertos utilizando el mismo conjunto de datos X_{train} como conjunto de prueba X_{test}. La métrica utilizada es la distancia euclidiana, y se realizaron tres pruebas para observar el rendimiento del algoritmo KNN. Los resultados de las pruebas son los siguientes:

- **Primera Prueba:**

```
# Punto 2.3.a
print("-----------------------------------------------------")
# Prueba con diferentes medias para observar la separabilidad de los datos
print("\nPruebas con diferentes medias para observar separabilidad:")
(labels_training_diff_means, data_training_diff_means) = crearDatos(numeroMuestraClase=1000, media1=[9, 9], media2=[25,25], dv_estandar1=[10, 10], dv_estandar2=[6, 6])
# Graficamos los datos con medias diferentes y desviaciones estándar
graficar_datos(data_training_diff_means, labels_training_diff_means)
# Pruebas con diferentes medias y desviaciones estándar
print("Pruebas con diferentes medias y distancia euclidiana")
start_time = time.time()
test_estimations_diff_means = evaluate_k_nearest_neighbors_test_dataset(data_training_diff_means, labels_training_diff_means, data_training_diff_means, K=7, is_euclidian=True)
accuracy_diff_means = calcular_tasa_aciertos(test_estimations_diff_means, labels_training_diff_means)
print("Tasa de acierto Euclideana con diferentes medias:", accuracy_diff_means)
elapsed_time = time.time() - start_time
print("Duración con norma Euclideana:", elapsed_time, "segundos")
```

Figura 2.4: Prueba 1

- Tasa de acierto: 1,0 Duración: 0,3539 segundos

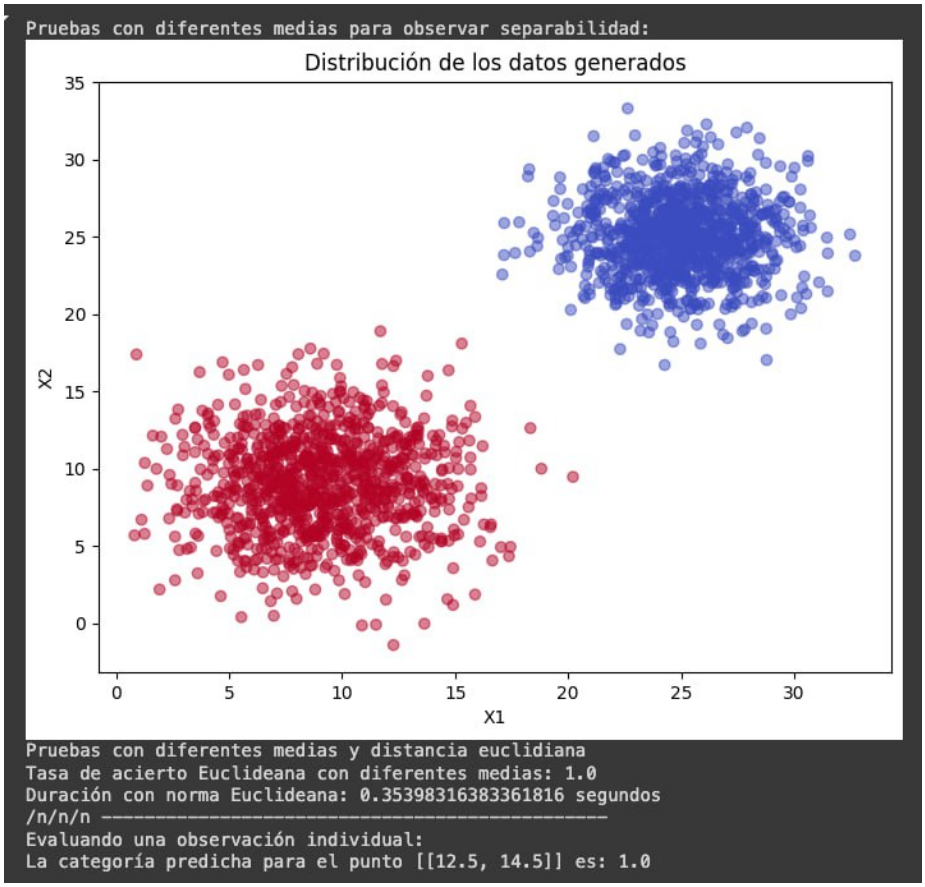


Figura 2.5: Resultado 1

- **Segunda Prueba:**

```
# Punto 2.3.a
print("-----------------------------------------------------")
# Prueba con diferentes medias para observar la separabilidad de los datos
print("\nPruebas con diferentes medias para observar separabilidad:")
(labels_training_diff_means, data_training_diff_means) = crearDatos(numeroMuestraClase=1000, media1=[9, 9], media2=[25,25], dv_estandar1=[[11, 11], dv_estandar2=[9, 9])
# Graficamos los datos con medias diferentes y desviaciones estándar
graficar_datos(data_training_diff_means, labels_training_diff_means)
# Pruebas con diferentes medias y desviaciones estándar
print("Pruebas con diferentes medias y distancia euclidiana")
start_time = time.time()
test_estimations_diff_means = evaluate_k_nearest_neighbors_test_dataset(data_training_diff_means, labels_training_diff_means, data_training_diff_means, K=7, is_euclidian=True)
accuracy_diff_means = calcular_tasa_aciertos(test_estimations_diff_means, labels_training_diff_means)
print("Tasa de acierto Euclideana con diferentes medias:", accuracy_diff_means)
elapsed_time = time.time() - start_time
print("Duración con norma Euclideana:", elapsed_time, "segundos")
```

Figura 2.6: Prueba 2

- Tasa de acierto: 1,0 Duración: 0,3712 segundos

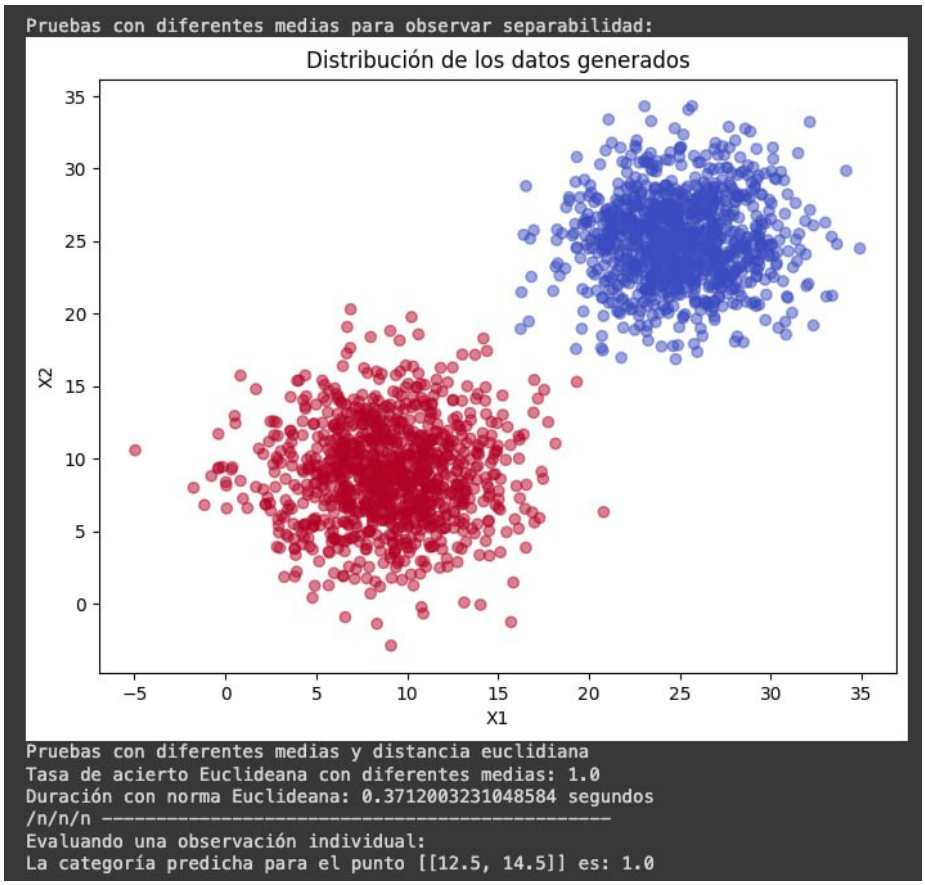


Figura 2.7: Resultado 2

- **Tercera Prueba:**

```python
# Punto 2.3.a
print("-----------------------------------------------")
# Prueba con diferentes medias para observar la separabilidad de los datos
print("\nPruebas con diferentes medias para observar separabilidad:")
(labels_training_diff_means, data_training_diff_means) = crearDatos(numeroMuestraClase=1000, media1=[10, 10], media2=[21,21], dv_estandar1=[11, 11], dv_estandar2=[9, 9])
# Graficamos los datos con medias diferentes y desviaciones estándar
graficar_datos(data_training_diff_means, labels_training_diff_means)
# Pruebas con diferentes medias y desviaciones estándar
print("Pruebas con diferentes medias y distancia euclidiana")
start_time = time.time()
test_estimations_diff_means = evaluate_k_nearest_neighbors_test_dataset(data_training_diff_means, labels_training_diff_means, data_training_diff_means, K=7, is_euclidian=True)
accuracy_diff_means = calcular_tasa_aciertos(test_estimations_diff_means, labels_training_diff_means)
print("Tasa de acierto Euclideana con diferentes medias:", accuracy_diff_means)
elapsed_time = time.time() - start_time
print("Duración con norma Euclideana:", elapsed_time, "segundos")
```

Figura 2.8: Prueba 3

- Tasa de acierto: 0,993 Duración: 0,3689 segundos

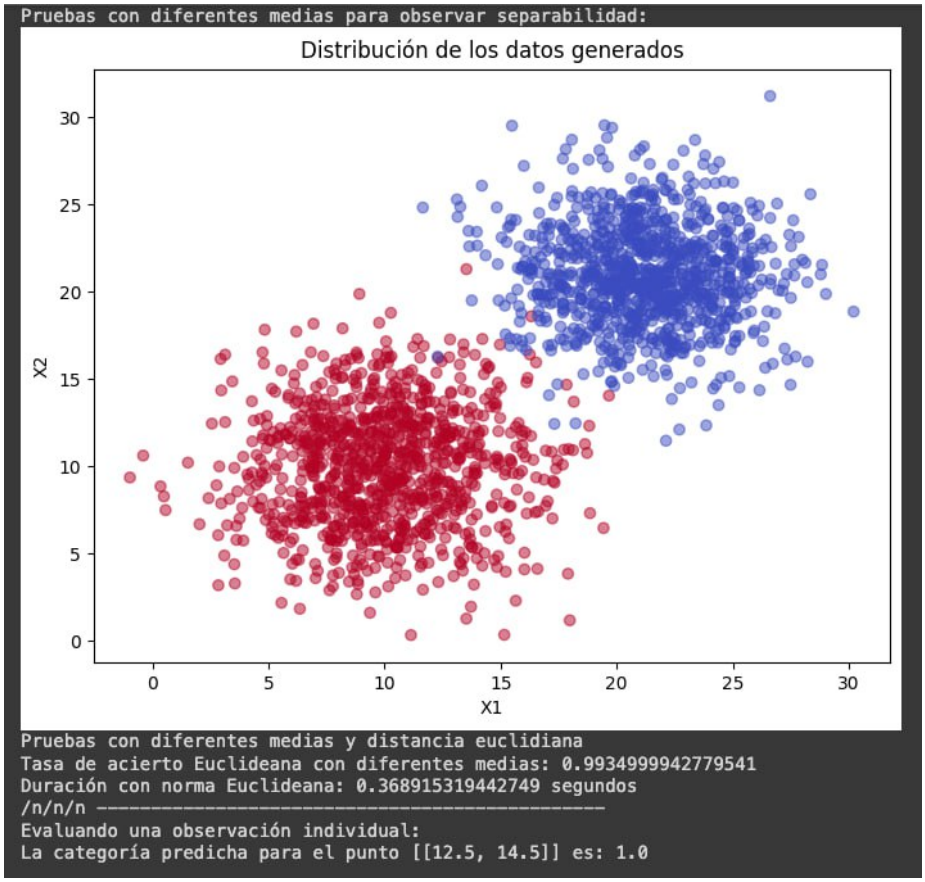


Figura 2.9: Resultado 3

- **Cuarta Prueba:**

```python
# Punto 2.3.a
print("-----------------------------------------------")
# Prueba con diferentes medias para observar la separabilidad de los datos
print("\nPruebas con diferentes medias para observar separabilidad:")
(labels_training_diff_means, data_training_diff_means) = crearDatos(numeroMuestraClase=1000, media1=[15, 15], media2=[20,20], dv_estandar1=[11, 11], dv_estandar2=[11, 11])
# Graficamos los datos con medias diferentes y desviaciones estándar
graficar_datos(data_training_diff_means, labels_training_diff_means)
# Pruebas con diferentes medias y desviaciones estándar
print("Pruebas con diferentes medias y distancia euclidiana")
start_time = time.time()
test_estimations_diff_means = evaluate_k_nearest_neighbors_test_dataset(data_training_diff_means, labels_training_diff_means, data_training_diff_means, K=7, is_euclidian=True)
accuracy_diff_means = calcular_tasa_aciertos(test_estimations_diff_means, labels_training_diff_means)
print("Tasa de acierto Euclideana con diferentes medias:", accuracy_diff_means)
elapsed_time = time.time() - start_time
print("Duración con norma Euclideana:", elapsed_time, "segundos")
```

Figura 2.10: Prueba 4

- Tasa de acierto: 0,865 Duración: 0,3882 segundos

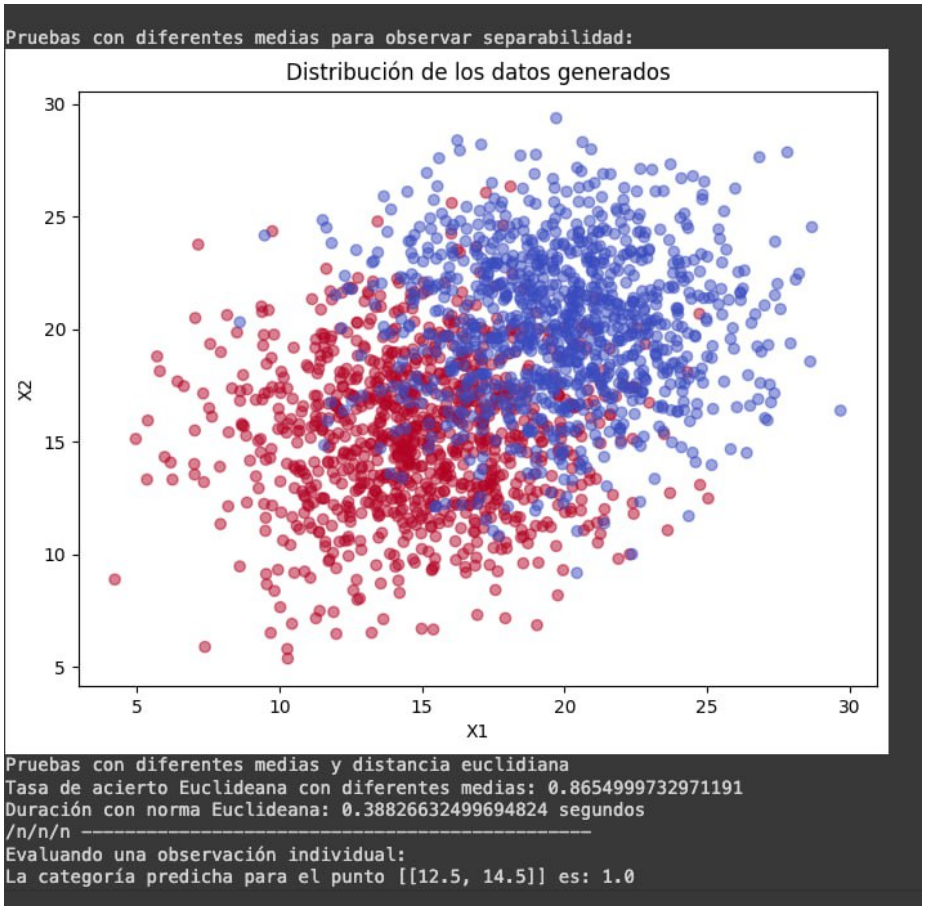


Figura 2.11: Resultado 4

En las cuatro iteraciones, se observa cómo la variación en las medias y desviaciones estándar de los datos afecta la tasa de acierto y el tiempo de ejecución del KNN. En las primeras dos iteraciones, con clases claramente separadas, la tasa de acierto es del 100 % y el tiempo de ejecución es bajo, lo que indica una clasificación eficiente.

En las últimas dos iteraciones, con clases más cercanas, la tasa de acierto disminuye (99.35 % y 86.55 %), lo que refleja la dificultad del modelo para distinguir entre clases menos separadas. A pesar de esto, el tiempo de ejecución solo aumenta ligeramente, mostrando que el KNN sigue siendo eficiente en términos de tiempo.

En resumen, el KNN funciona muy bien con clases separadas, pero su precisión disminuye cuando las clases están más cercanas, aunque el tiempo de ejecución se mantiene eficiente.

Capítulo 3

Conclusiones

Este proyecto se centró principalmente en la implementación del algoritmo K-vecinos más cercanos (KNN), donde se exploraron distintas estrategias para mejorar su eficiencia y precisión. Se implementó una versión matricial de KNN, optimizando el cálculo de distancias y la clasificación de puntos con el uso de PyTorch, lo que permitió realizar operaciones complejas de manera eficiente. Los resultados obtenidos demostraron que KNN puede alcanzar altas tasas de acierto en conjuntos de datos bien separados, mientras que su rendimiento disminuye cuando los datos presentan menor separabilidad, un desafío común en problemas de clasificación.

Además, la optimización fue clave para mejorar la eficiencia del algoritmo. Se evitó el uso de ciclos for, lo que, combinado con funciones nativas de PyTorch como 'cdist' para el cálculo de distancias y 'mode' para la clasificación, permitió mejorar significativamente el tiempo de ejecución, lo que es crucial en aplicaciones que involucran grandes volúmenes de datos. Esta optimización computacional no solo ayudó a reducir el tiempo de procesamiento, sino que también facilitó la implementación del algoritmo en entornos de producción.

En resumen, este proyecto no solo demostró la efectividad de KNN en la clasificación de datos, sino que también resaltó la importancia de optimizar algoritmos desde el punto de vista computacional para abordar problemas complejos de manera eficiente. Esta combinación de optimización matemática y computacional es fundamental en la resolución de problemas reales, especialmente en áreas donde el manejo de grandes volúmenes de datos y la precisión en la clasificación son esenciales.