Optimización y Programación Numérica
Escuela de Ingeniería en Computación
Maestría en Computación con Énfasis en Ciencias de la Computación

Trabajo práctico 1: Vectores, Matrices y Cálculo Multi-variable

Profesor: PhD. Saúl Calderón Ramírez

Realizado por

Andrés Chavarría Sibaja
Fernando Ugalde Green
Esteban Villalobos Gómez

Instituto Tecnológico de Costa Rica

23 de marzo, 2025

Tabla de contenidos

1 El producto punto y matrices

1.1 Norma infinita

1. Demuestra que la norma ℓ_∞ cumple la propiedad de la homogeneidad absoluta.

   (a) Muestre, usando 50 arreglos numéricos generados aleatoriamente esta propiedad (adjunte el código en pytorch)

Demostración:

Sea $\vec{x} = [x_1, x_2, x_3, ..., x_n] \in \mathbb{R}^n$ t.q.

$$\|\vec{x}\|_\infty = \max\{|x_i| : 1 \leq i \leq n\} \tag{1}$$

$\Rightarrow$ Sea α un escalar t.q. $\alpha \in \mathbb{R}$. Así tomando la ecuación (1), tenemos que:

$$\|\alpha\vec{x}\|_\infty = \max\{|\alpha x_i| : 1 \leq i \leq n\} \tag{2}$$

$\Rightarrow$ Donde por la propiedad $|\alpha x_i| = |\alpha| \cdot |x_i|$, tenemos que en la (2):

$$\|\alpha\vec{x}\|_\infty = \max\{|\alpha||x_i| : 1 \leq i \leq n\} \tag{3}$$

$\Rightarrow$ Así "sacando a factor común":

$$\|\alpha\vec{x}\|_\infty = |\alpha| \max\{|x_i| : 1 \leq i \leq n\} \tag{4}$$

$\Rightarrow$ Ahora, como $\max\{|x_i| : 1 \leq i \leq n\} = \|\vec{x}\|_\infty$, entonces tenemos que en la ecuación (4):

$$\|\alpha\vec{x}\|_\infty = |\alpha|\|\vec{x}\|_\infty \tag{5}$$

Por tanto queda demostrado según la ecuación (5)

Código Python:

[1]:
```
import torch
import random
```

[2]:
```
vect_list = [torch.randn(random.randint(1, 10)) for _ in range(50)]
alpha_list = [torch.randn(1).item() for _ in range(50)]
scal_vect_list  = [alpha * vector for alpha, vector in zip(alpha_list, vect_list)]
```

[3]:
```
for idx, (vector, alpha, vector_escalado) in enumerate(zip(vect_list, alpha_list,
scal_vect_list)):
```

```
    original_norm = torch.linalg.norm(vector, ord=float('inf'))
    scale_norm = torch.linalg.norm(vector_escalado, ord=float('inf'))
    cumple = torch.isclose(scale_norm, abs(alpha) * original_norm, atol=1e-5)

    print(f'[{idx+1}] Longitud: {vector.numel()}')
    print(f'Vector: {vector}')
    print(f'Alpha: {alpha}')
    print(f'Norma original L-inf: {original_norm.item()}')
    print(f'Norma escalada L-inf: {scale_norm.item()}')
    print(f'|alpha| * norma original: {abs(alpha) * original_norm}')
    print(f'Homogeneidad cumple: {cumple}\n')
```

Los resultados de la ejecución del código anterior pueden ser visto en el Jupyter Notebook adjunto en la tarea.

1.2 Producto punto

2. Demuestra que si dos vectores $\vec{u}$ y $\vec{v}$ son ortogonales, entonces:

$$\|\vec{u}+\vec{v}\| = \|\vec{u}\|^2 + \|\vec{v}\|^2 \tag{6}$$

   (a) Muestre, usando 50 arreglos numéricos generados aleatoriamente esta propiedad (adjunte el código en pytorch)

Demostración:

$\Rightarrow$ Sabiendo $\vec{u}\cdot\vec{v} = \vec{v}\cdot\vec{u} = 0$ pues $\vec{u} \perp \vec{v}$. Ahora tomando $\|\vec{u}+\vec{v}\|^2$ del lado izquierdo de la ecuación (6) tenemos que:

$$\|\vec{u}+\vec{v}\|^2 = (\vec{u}+\vec{v})\cdot(\vec{u}+\vec{v}) \tag{7}$$

$\Rightarrow$ Ahora tomando la propiedad distributiva del producto punto

$$\|\vec{u}+\vec{v}\|^2 = \vec{u}\cdot\vec{u} + \vec{u}\cdot\vec{v} + \vec{v}\cdot\vec{u} + \vec{v}\cdot\vec{v} \tag{8}$$

$\Rightarrow$ Ahora $\vec{u}\cdot\vec{v} = \vec{v}\cdot\vec{u} = 0$, $\vec{u}\cdot\vec{u} = \|\vec{u}\|^2$ y $\vec{v}\cdot\vec{v} = \|\vec{v}\|^2$ entonces: de la ecuación (8), tenemos:

$$\|\vec{u}+\vec{v}\|^2 = \vec{u}\cdot\vec{u} + \vec{v}\cdot\vec{v} = \|\vec{u}\|^2 + \|\vec{v}\|^2 \tag{9}$$

Por tanto $\|\vec{u}+\vec{v}\|^2 = \|\vec{u}\|^2 + \|\vec{v}\|^2$, quedando demostrado.

Código python:

[1]:
```
import torch
import random
```

Creamos una función de generación de vectores mutuamente ortogonales llamada generador ortogonal:

[2]:
```
def generador_ortogonal(u):
    v = torch.randn_like(u)#recibimos al vector u y calculamos una proyeccion
    proyeccion = (torch.dot(u, v) / torch.dot(u, u)) * u
    v_ortogonal = v - proyeccion #usando la proyeccion calculamos el vector v
ortogonal a u
    return v_ortogonal
```

Generamos 50 pares de vectores $\vec{u}$ y $\vec{v}$ para poder hacer el calculo de los ejemplos

[3]:
```
pares_ortogonales = [(u := torch.randn(random.randint(1, 10)),  # u aleatorio de
longitud aleatoria
                      v := generador_ortogonal(u))#generamos el vector v a partir
del vector u
                     for _ in range(50)]
```

[4]:
```
for idx, (u, v) in enumerate(pares_ortogonales):
    pp = torch.dot(u, v)

    u2_norm = torch.norm(u)**2
    v2_norm = torch.norm(v)**2
    sum2_norm = torch.norm(u + v)**2

    cumple = torch.isclose(sum2_norm, u2_norm + v2_norm, atol=1e-5)

    print(f'[{idx+1}] Longitud: {u.numel()}')
    print(f'u: {u}')
    print(f'v: {v}')
    print(f'Producto punto (debe ser 0): {pp.item()}')
    print(f'||u||^2: {u2_norm.item()}')
    print(f'||v||^2: {v2_norm.item()}')
    print(f'||u+v||^2: {sum2_norm.item()}')
    print(f'Suma de ||u||^2 + ||v||^2: {(u2_norm + v2_norm).item()}')
    print(f'Propiedad cumple: {cumple}\n')
```

Al ejecutar el código anterior se pueden obtener 50 ejemplos de comprobación de la ecuación (6) para vectores de entradas numéricas aleatorias y longitudes aleatorias

1.3 Cálculo y propiedades de matrices

3. Siendo las matrices $A, B \in \mathbb{R}^{n\times n}$ con $A + B = 1_n$ y $AB = 0_n$, demuestre $A^2 = A$ y $B^2 = B$

   (a) Genere al menos un ejemplo numérico que muestre si tal propiedad se da.

Demostración:

Tomando:

$$A + B = 1_n \tag{10}$$

Despejamos en la ecuación (10) para la matriz B t.q:

$$\Rightarrow B = 1_n - A \tag{11}$$

En la ecuación (11) hacemos producto matricial por A a la izquierda, y distribuimos, por tanto:

$$\Rightarrow AB = A(1_n - A) \tag{12}$$

$$\Rightarrow AB = A - A^2 \tag{13}$$

Como el producto AB=0_n entonces tenemos:

$$\Rightarrow 0_n = A - A^2 \tag{14}$$

Por lo tanto, de la ecuación (14) se tiene que despejando para la matriz A:

$$\Rightarrow A^2 = A \tag{15}$$

Quedando la primera proposición queda demostrada.

Ahora retomando la ecuación (11) y haciendo el producto matricial por B a la derecha, tenemos:

$$\Rightarrow BB = (1_n - A)B \tag{16}$$

Distribuyendo:

$$\Rightarrow B^2 = B - AB \tag{17}$$

Se tiene que por el enunciado $AB = O_n$, y por lo tanto despejando para la matriz B:

$$\Rightarrow B^2 = B \tag{18}$$

Quedando la segunda proposición queda demostrada en la ecuación (18).

Ejemplo generado:

Sean las siguientes matrices A y B, t.q:

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \tag{19}$$

$$B = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{20}$$

Verificación de $A^2 = A$

$$A^2 = A \cdot A = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = A \tag{21}$$

Verificación de $B^2 = B$

$$B^2 = B \cdot B = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = B \tag{22}$$

Verificación de $AB = 0$

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 0 + 0 \cdot 0 & 1 \cdot 0 + 0 \cdot 1 \\ 0 \cdot 0 + 0 \cdot 0 & 0 \cdot 0 + 0 \cdot 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} = 0 \tag{23}$$

4. Para la siguiente matriz:

$$A = s \begin{bmatrix} -1 & 2 & 2 \\ 2 & -1 & 2 \\ 2 & 2 & -1 \end{bmatrix} \tag{24}$$

Define un valor s que haga a la matriz ortonormal y verifíquelo haciendo $U^T = U^{-1}$

Verificación de Ortoginalidad de vectores: Primeramente realizamos una verificación rápida de la ortogonalidad de los vectores que conforma la matriz A:

$$\vec{a_1} \cdot \vec{a_2} = -1 \cdot 2 + 2 \cdot (-1) + 2 \cdot 2 = -2 + -2 + 4 = -4 + 4 = 0 \tag{25}$$

$$\vec{a_1} \cdot \vec{a_3} = -1 \cdot 2 + 2 \cdot 2 + 2 \cdot (-1) = -2 + 4 - 2 = 0 \tag{26}$$

$$\vec{a_2} \cdot \vec{a_3} = 2 \cdot 2 + (-1) \cdot 2 + 2 \cdot (-1) = 4 - 2 - 2 = 0 \tag{27}$$

Como vemos según las ecuaciones (25), (26) y (27), los vectores de la matriz A son ortogonales todos entre si. Además, vemos fácilmente que para sus filas es fácil darse cuenta que ocurre lo mismo pues es una A matriz simétrica. Por otro lado, ya que A es una matriz simétrica se tienen también que $A = A^T$

Sin embargo, los vectores de A deben de estar normalizados, por lo que calculamos la norma de cada uno. En este caso es suficiente calcular estas normas una única vez ya que es la misma norma para los 3 vectores. Así sea a la norma de los vectores columna de la matriz, tenemos que calculando para el primer vector columna $\vec{a}$ de la matriz A:

$$a_1 = \sqrt{(-1)^2 + (2)^2 + (2)^2} = \sqrt{1 + 4 + 4} = \sqrt{9} = 3 \tag{28}$$

De la ecuación (28) se sigue que al dividir todos los vectores entre 3. Así tomando $S = \frac{1}{3}$, tenemos:

$$A = \frac{1}{3}\begin{bmatrix} -1 & 2 & 2 \\ 2 & -1 & 2 \\ 2 & 2 & -1 \end{bmatrix} \tag{29}$$

Por lo que la matriz A en la ecuación (29) es ortonormal.

Podemos verificar por medio de $U^T = U^{-1} \Rightarrow UU^T = 1_n$

Para hacer esta comprobación calcularemos A^{-1} como sigue:

$$A^{-1} = \begin{bmatrix} -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \end{bmatrix} \Rightarrow \left[\begin{array}{ccc|ccc} -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} & 1 & 0 & 0 \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} & 0 & 1 & 0 \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} & 0 & 0 & 1 \end{array}\right]$$

$$3f_1 \Rightarrow \left[\begin{array}{ccc|ccc} -1 & 2 & 2 & 3 & 0 & 0 \\ 2 & -1 & 2 & 0 & 3 & 0 \\ 2 & 2 & -1 & 0 & 0 & 3 \end{array}\right] \xrightarrow{2f_1+f_2} \left[\begin{array}{ccc|ccc} -1 & 2 & 2 & 3 & 0 & 0 \\ 0 & 3 & 6 & 6 & 3 & 0 \\ 0 & 6 & 3 & 6 & 0 & 3 \end{array}\right]$$

$$\frac{1}{3}f_2 \Rightarrow \left|\begin{array}{ccc|ccc} -1 & 2 & 2 & 3 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ 0 & 2 & 1 & 2 & 0 & 1 \end{array}\right| \xrightarrow{-f_2+f_1} \left[\begin{array}{ccc|ccc} -1 & 0 & 1 & 1 & 0 & -1 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ 0 & 2 & 1 & 2 & 0 & 1 \end{array}\right]$$

$$-2f_3 + f_2 \Rightarrow \left[\begin{array}{ccc|ccc} -1 & 0 & -1 & 1 & 0 & -1 \\ 0 & -3 & 0 & -2 & 1 & -2 \\ 0 & 2 & 1 & 2 & 0 & 1 \end{array}\right] \xrightarrow{-\frac{1}{3}f_2} \left[\begin{array}{ccc|ccc} -1 & 0 & 1 & 1 & 0 & -1 \\ 0 & 1 & 0 & \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ 0 & 2 & 1 & 2 & 0 & 1 \end{array}\right]$$

$$-2f_2 + f_3 \Rightarrow \left[\begin{array}{ccc|ccc} -1 & 0 & -1 & 1 & 0 & -1 \\ 0 & 1 & 0 & \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ 0 & 0 & 1 & \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \end{array}\right] \xrightarrow{-f_3+f_1} \left[\begin{array}{ccc|ccc} -1 & 0 & 0 & \frac{1}{3} & -\frac{2}{3} & -\frac{2}{3} \\ 0 & 1 & 0 & \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ 0 & 0 & 1 & \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \end{array}\right]$$

$$-f_1 \Rightarrow \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ 0 & 1 & 0 & \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ 0 & 0 & 1 & \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \end{array}\right]$$

Donde por lo tanto A^{-1} es tal que:

$$A^{-1} = \begin{bmatrix} -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \end{bmatrix} = \frac{1}{3}\begin{bmatrix} -1 & 2 & 2 \\ 2 & -1 & 2 \\ 2 & 2 & -1 \end{bmatrix} = A \tag{30}$$

Por lo tanto tenemos que como $AA^T = 1_n$, pues $A^{-1} = A^T = 1_n$, tenemos que al realizar el calculo del producto de AA^T:

$$AA^T = s^2 \begin{bmatrix} -1 & 2 & 2 \\ 2 & -1 & 2 \\ 2 & 2 & -1 \end{bmatrix}\begin{bmatrix} -1 & 2 & 2 \\ 2 & -1 & 2 \\ 2 & 2 & -1 \end{bmatrix}$$

$$= s^2 \begin{bmatrix} 1+4+4 & -2-2+4 & -2+4-2 \\ -2-2+4 & 4+1+4 & 4-2-2 \\ -2+4-2 & 4-2-2 & 4+4-1 \end{bmatrix}$$

$$= s^2 \begin{bmatrix} 9 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 9 \end{bmatrix}$$

De lo anterior se deduce que para que la matriz resultante sea la matriz unitaria, entonces s debe ser una contante tal que:

$$AA^T = s^2 \begin{bmatrix} 9 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 9 \end{bmatrix} \tag{31}$$

Tal que:

$$\Rightarrow \quad s^2 = \frac{1}{9} \Rightarrow s = \frac{1}{\sqrt{9}} = \frac{1}{3} \tag{32}$$

Por lo que por la ecuación (32), queda demostrado nuevamente el valor de la constante s

2 Funciones multivariable

2.1 Funciones lineales multivariable

Un hiperplano definido en un espacio $\mathbb{R}^{n+1}$ se puede expresar como una función con dominio $\overrightarrow{x} \in \mathbb{R}^n$ y codominio en $\mathbb{R}$ como sigue: $z = f(\overrightarrow{x}) = \overrightarrow{x} \cdot \overrightarrow{w}$, con $\overrightarrow{w} \in \mathbb{R}^n$ el arreglo de coeficientes de tal funcional.

1. Tómese $\overrightarrow{w}_1 = \begin{bmatrix} 0.5 \\ 0.2 \end{bmatrix}$ para la función f_1 y $\overrightarrow{w}_2 = \begin{bmatrix} -0.1 \\ 0.05 \end{bmatrix}$ para la función f_2, (funciones con dominio en $\mathbb{R}^2$ y codominio en $\mathbb{R}$). Grafique ambos planos en Pytorch.

2. Para cada plano, grafique el vector normal en el punto $P = (1, 1)$ y una curva de nivel perpendicular a tal vector normal. Calcule el vector gradiente en tal punto, y demuestre que ese vector gradiente es perpendicular a la curva de nivel previamente dibujada.

   Para demostrar que ∇f_1 es perpendicular a la curva de nivel en $(1, 1)$, basta con mostrar que el producto punto entre ∇f_1 y $[1, 1, f_1(1, 1)]$ da como resultado 0.

$$f_1(\overrightarrow{x}) = f_1(x, y) = 0.5x + 0.2y = z \Rightarrow 0.5x + 0.2y - z = 0 \tag{33}$$

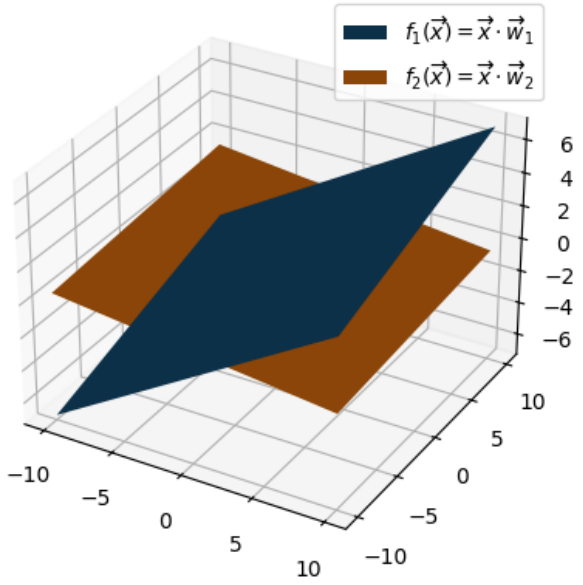

Figura 1: Planos en $\mathbb{R}^2 \to \mathbb{R}$

$$\begin{aligned} \nabla f_1 \cdot [1, 1, f_1(1,1)] &= \left[\tfrac{df_1}{dx}, \tfrac{df_1}{dy}, \tfrac{df_1}{dz}\right] \cdot [1, 1, 0.5 + 0.2] \\ &= [0.5, 0.2, -1] \cdot [1, 1, 0.7] \\ &= 0.5 \cdot 1 + 0.2 \cdot 1 - 1 \cdot 0.7 \\ &= 0 \end{aligned} \tag{34}$$

Para demostrar que ∇f_2 es perpendicular a la curva de nivel en $(1, 1)$, basta con mostrar que el producto punto entre ∇f_2 y $[1, 1, f_2(1, 1)]$ da como resultado 0.

$$f_2(\vec{x}) = f_2(x, y) = -0.1x + 0.05y = z \Rightarrow -0.1x + 0.05y - z = 0 \tag{35}$$

$$\begin{aligned} \nabla f_2 \cdot [1, 1, f_2(1,1)] &= \left[\tfrac{df_2}{dx}, \tfrac{df_2}{dy}, \tfrac{df_2}{dz}\right] \cdot [1, 1, -0.1 + 0.05] \\ &= [-0.1, 0.05, -1] \cdot [1, 1, -0.05] \\ &= -0.1 \cdot 1 + 0.05 \cdot 1 + 1 \cdot 0.05 \\ &= 0 \end{aligned} \tag{36}$$

2.2 El vector gradiente

Para cada una de las siguientes funciones multivariable:

$$\begin{aligned} f_1(x, y) &= (1.5 - x + xy)^2 + (2.25 - x + xy^2) + (2.625 - x + xy^3) \\ f_2(x, y) &= (x + 2y - 7)^2 + (2x + y - 5)^2 \\ f_3(x, y) &= 100\sqrt{|y - 0.01x^2|} + 0.01|x + 10| \end{aligned} \tag{37}$$

1. Grafique su superficie con dominio entre -10 y 10.
2. Calcule el vector gradiente manualmente, evalúelo y grafique el vector unitario en la dirección del gradiente para los dos puntos especificados (en la misma figura de la superficie).

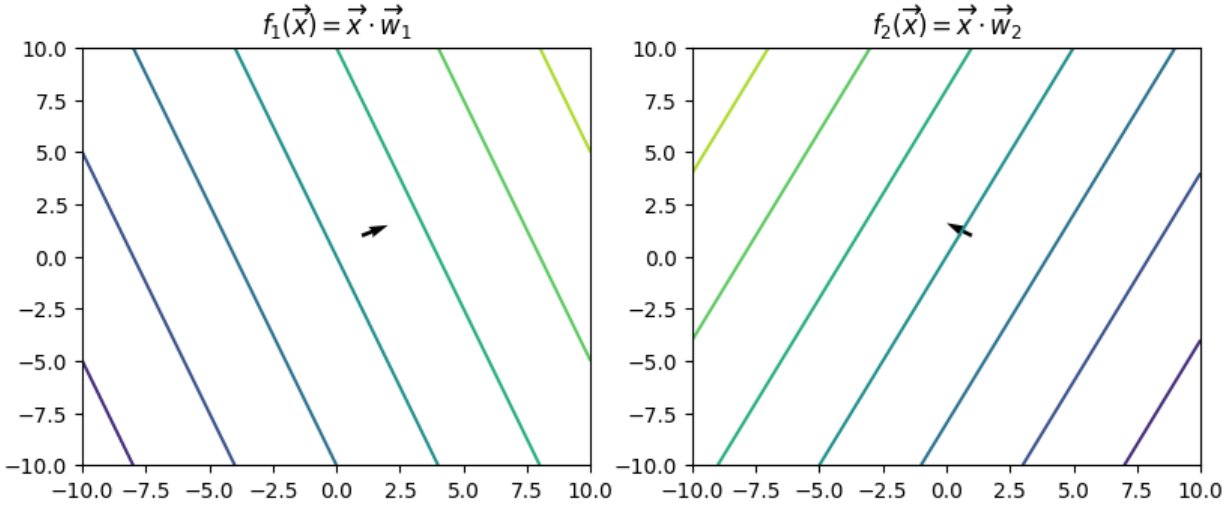

Figura 2: Curvas de nivel y vectores normales.

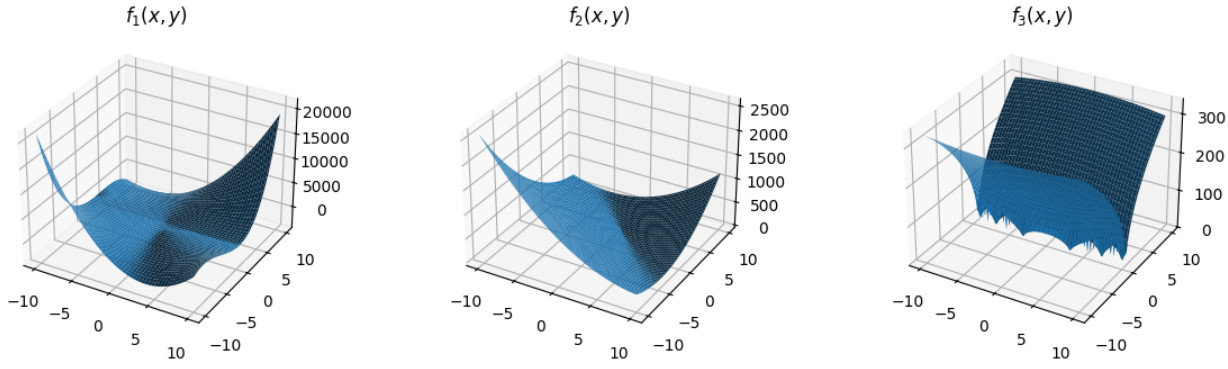

Figura 3: Superficies en el dominio entre -10 y 10.

$$
\begin{aligned}
\nabla f_1 &= \left[\tfrac{df_1}{dx}, \tfrac{df_1}{dy}\right] \\
\tfrac{df_1}{dx} &= 2(1.5 - x + xy)(y - 1) + (y^2 - 1) + (y^3 - 1) \\
\tfrac{df_1}{dy} &= 2(1.5 - x + xy)x + 2xy + 3xy^2 \\
\nabla f_1(0, 0) &= [-5, 0] \\
\nabla f_1(2, 2) &= [17, 46]
\end{aligned}
\tag{38}
$$

$$
\begin{aligned}
\nabla f_2 &= \left[\tfrac{df_2}{dx}, \tfrac{df_2}{dy}\right] \\
\tfrac{df_2}{dx} &= 2(x + 2y - 7) + 4(2x + y - 5) \\
\tfrac{df_2}{dy} &= 4(x + 2y - 7) + 2(2x + y - 5) \\
\nabla f_2(1.5, -5.5) &= [-63, -81] \\
\nabla f_2(-7, -7) &= [-160, -164]
\end{aligned}
\tag{39}
$$

$$
\begin{aligned}
\nabla f_3 &= \left[\tfrac{df_3}{dx}, \tfrac{df_3}{dy}\right] \\
u &= y - 0.01x^2 \\
\tfrac{df_3}{dx} &= -x\tfrac{\text{sgn}(u)}{\sqrt{|u|}} + 0.01 \cdot \text{sgn}(x+10) \\
\tfrac{df_3}{dy} &= 50\tfrac{\text{sgn}(u)}{\sqrt{|u|}} \\
\nabla f_3(-4,-2) &\approx [-2.7117, -34.0207] \\
\nabla f_3(-2,3) &\approx [1.1725, 29.0619]
\end{aligned}
\tag{40}
$$

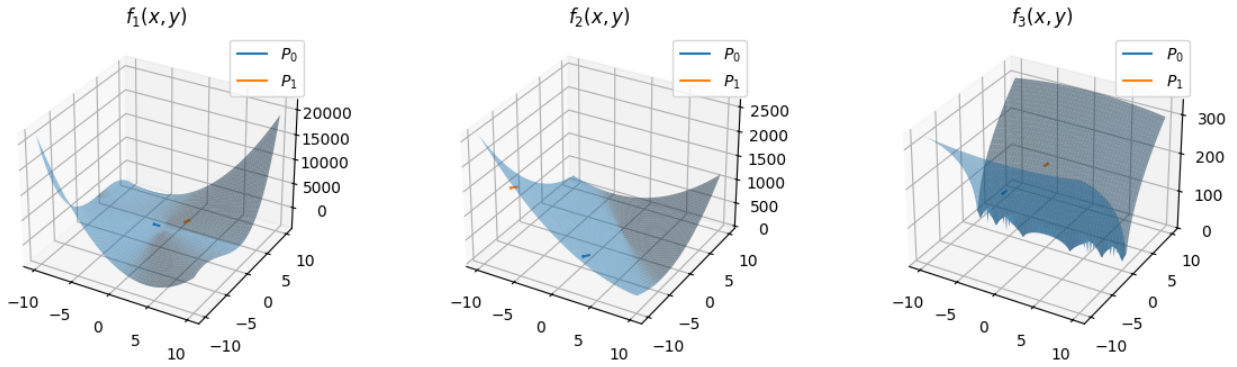

Figura 4: Superficies con vectores unitarios en la dirección del gradiente para cada punto especificado.

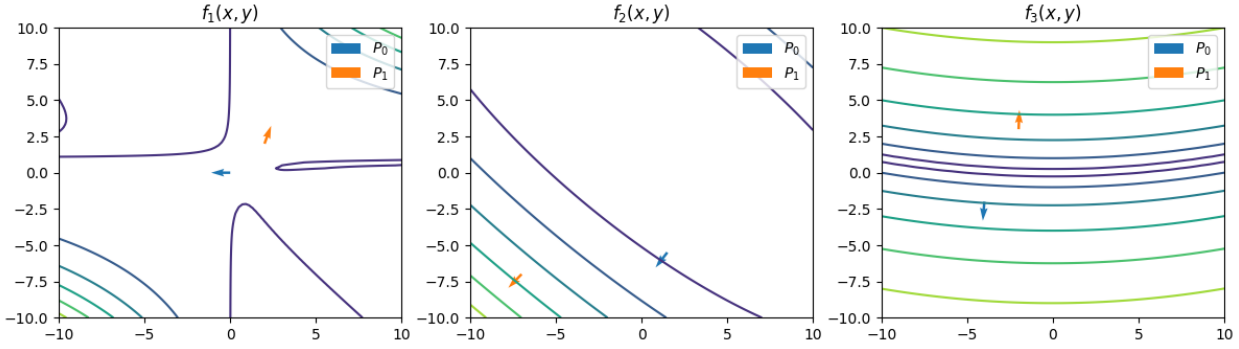

Figura 5: Curvas de nivel con vectores unitarios en la dirección del gradiente para cada punto especificado.

3. Calcule la magnitud de tal vector gradiente en cada punto.

$$
\begin{aligned}
||\nabla f_1(0,0)||_2 &= \sqrt{(-5)^2+0^2} &&= 5 \\
||\nabla f_1(2,2)||_2 &= \sqrt{17^2+46^2} &&\approx 49.0408 \\
||\nabla f_2(1.5,-5.5)||_2 &= \sqrt{(-63)^2+(-81)^2} &&\approx 102.6158 \\
||\nabla f_2(-7,-7)||_2 &= \sqrt{(-160)^2+(-164)^2} &&\approx 229.1201 \\
||\nabla f_3(-4,-2)||_2 &\approx \sqrt{(-2.7117)^2+(-34.0207)^2} &&\approx 34.1286 \\
||\nabla f_3(-2,3)||_2 &\approx \sqrt{1.1725^2+29.0619^2} &&\approx 29.0856
\end{aligned}
\tag{41}
$$

4. Calcule lo que se conoce como la matriz Hessiana.

$$
\begin{aligned}
H_{f(x,y)} &= \begin{bmatrix} \frac{d^2 f}{dx^2} & \frac{d^2 f}{dxy} \\ \frac{d^2 f}{dyx} & \frac{d^2 f}{dy^2} \end{bmatrix} \\
H_{f_1(x,y)} &= \begin{bmatrix} 2(y-1)^2 & 2(1.5-2x+2xy)+2y+3y^2 \\ 2(1.5-2x+2xy)+2y+3y^2 & 2x^2+2x+6xy \end{bmatrix} \\
H_{f_2(x,y)} &= \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix} \\
u &= y - 0.01x^2 \\
H_{f_3(x,y)} &= \begin{bmatrix} \frac{-\text{sgn}(u)}{\sqrt{|u|}} - \frac{0.01x^3}{|u|^{1.5}} & \frac{x}{2|u|^{1.5}} \\ \frac{x}{2|u|^{1.5}} & \frac{-25}{|u|^{1.5}} \end{bmatrix}
\end{aligned}
\tag{42}
$$

3 Implementación del algoritmo K-vecinos más cercanos

El algoritmo de K-vecinos más cercanos es un algoritmo de aprendizaje automático supervisado muy popular por su simplicidad. Dado un conjunto de datos representado matricialmente en la matriz $X_{\text{train}} \in \mathbb{R}^{N \times D}$ y un arreglo de etiquetas $\overrightarrow{t} \in \mathbb{R}^N$:

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

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

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

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

$$
X_{\text{KNN}} = \arg \min_{K \; \min j} \left(d\left(\overrightarrow{x}_i^{(\text{test})} - \overrightarrow{x}_j \right) \right)
$$

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

1. Implemente el algoritmo de K-vecinos mas cercanos con la posibilidad de usar la distancia euclidiana, de Manhattan, Infinito en la función $d\left(\overrightarrow{x}_i - \overrightarrow{x}_j\right)$.

   (a) Realice la implementación 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, p = 2)* (**No use ciclos *for***).

      i. Para ello use funcionalidades de *pytorch* como *repeat*, *mode*, *sort*, etc.

      ii. p indica el tipo de norma a utilizar. K corresponde a la cantidad de vecinos a evaluar.

      iii. Diseñe al menos 2 pruebas unitarias para esta función.

      Implementación i, ii:

[ ]:
```
#1.a
def evaluate_k_nearest_neighbors_observation(data_training,
labels_training, test_observation, k=7, p=2.0):
```

```
  # Calcula la distancia entre la observacion de entrada y el resto del
dataset.
  if p == float('inf'):
    dist = torch.max(torch.abs(data_training - test_observation), dim=1).
 ↪values.unsqueeze(0)
  else:
    dist = torch.pow(torch.sum(torch.pow(torch.abs(data_training -
test_observation), p), dim=1), 1/p).unsqueeze(0)

  # Se seleccionan los K indices con las menores distancias
  k_vecinos = dist.topk(k, largest=False)

  # Extraccion de los labels de los vecinos mas cercanos
  target_labels = labels_training[k_vecinos.indices[0]]

  # La moda se puede usar para obtener el valor más repetido
  # Este valor se convierte en la etiqueta estimada
  return torch.mode(target_labels, 0).values.unsqueeze(0)
```

Pruebas Unitarias iii:

[ ]:
```
# Datos de prueba
ut_data = torch.tensor([[1.0, 2.0], [2.0, 3.0], [3.0, 4.0], [6.0, 7.0], [7.
 ↪0, 8.0]])
ut_labels = torch.tensor([0, 0, 0, 1, 1]).unsqueeze(1)  # Dos clases: 0 y 1

#Puntos a evaluar
punto1 = torch.tensor([2.5, 3.5]) # clase 0
punto2 = torch.tensor([5.5, 6.5]) # clase 1
punto3 = torch.tensor([1.5, 2.5]) # clase 0

print(f'ut_data shape:\t\t{ut_data.shape}\nut_labels shape:\t{ut_labels.
 ↪shape}')

# Plotting datos de prueba
plt.scatter(ut_data[:, 0], ut_data[:, 1], c=ut_labels.squeeze())

# Plotting puntos especificos de pruebas
plt.scatter(punto1[0], punto1[1], color='red', marker='x', label='Punto de
prueba 1')
plt.scatter(punto2[0], punto2[1], color='green', marker='x', label='Punto
de prueba 2')
plt.scatter(punto3[0], punto3[1], color='blue', marker='x', label='Punto de
prueba 3')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Observaciones para unit tests')
plt.legend()
plt.show()
```

```
ut_data shape:          torch.Size([5, 2])
ut_labels shape:        torch.Size([5, 1])
```

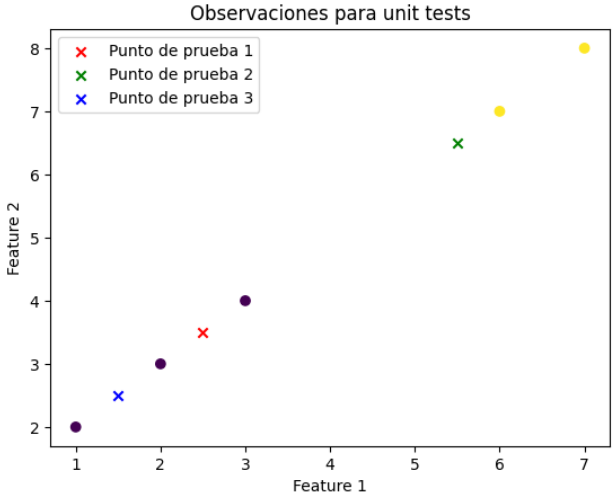

Figura 6: Dataset de pruebas unitarias con 5 observaciones, 3 de clase 0 y 2 de clase 1. Las x muestran los puntos de prueba (pivote) para probar el algoritmo de KNN.

[ ]:
```
# Test bench

def ut_p(test_point, expected_class, p):
  test_estimations_all = evaluate_k_nearest_neighbors_observation(ut_data,
ut_labels, test_point, k = 2, p=p, debug=True)
  print("Clase estimada: ", test_estimations_all.squeeze(1))
  print("Clase esperada: ", torch.tensor([0]))
  if torch.equal(test_estimations_all.squeeze(1), expected_class):
    print(f"Test {p} passed")
    return True

  print(f"Test {p} failed")
  return False
```

Ejecución de pruebas

[ ]:
```
pruebas = []
print("Unit test 1")
pruebas.append(ut_p(punto1, torch.tensor([0]), 1.0))
print("Unit test 2")
pruebas.append(ut_p(punto2, torch.tensor([1]), 1.0))
print("Unit test 3")
pruebas.append(ut_p(punto3, torch.tensor([0]), 1.0))
```

```
Prueba       Passed
 Prueba 1     True
 Prueba 2     True
 Prueba 3     True
```

(b) Para todo el conjunto de datos X_{test} implemente la funcion *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 $\vec{\hat{t}}$ para todos los datos en X_{test}.

```
[ ]: #1.b
def evaluate_k_nearest_neighbors_test_dataset(data_training, labels_training,
test_dataset, k=3, p=float('inf')):
  """
  Se elimina la variable de la especificacion is_euclidean, pues se deben
soportar tres normas:
  p = [1, 2, float('inf')]
  """
  # calcula el knn para cada observacion del test_dataset
  t_estimated_dataset = [
    evaluate_k_nearest_neighbors_observation(data_training, labels_training,
test_observation, k, p)
      for test_observation in test_dataset
    ]

  # retorna un tensor en la misma arquitectura que el tensor de data_training
  return torch.tensor(t_estimated_dataset).to(data_training.device)
```

(c) Implemente la función *calcular_tasa_aciertos* la cual tome un arreglo de estimaciones $\overrightarrow{t}$ 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*)**.

```
[ ]: #1.c
def calcular_tasa_aciertos(test_estimations, test_labels):
  """ Calcula la tasa de aciertos """
  comparison = torch.eq(test_estimations, test_labels)
  return torch.count_nonzero(comparison, dim=0).item() / test_labels.shape[0]
```

2. Para un conjunto de datos de $N = 1000000$ (500000 observaciones por clase)[1] 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.

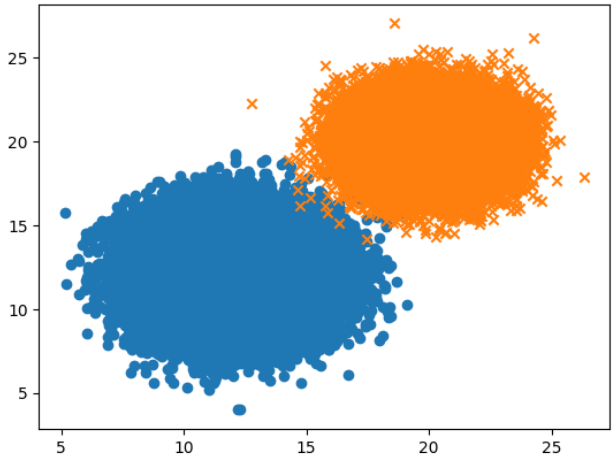

Figura 7: Dataset de prueba con cien mil observaciones, 50% para cada clase: clase 0 azul $\mu_1 = [12, 12]^T$ y $\sigma_1 = [3, 3]^T$, clase 1 naranja $\mu_2 = [20, 20]^T$ y $\sigma_2 = [2, 2]^T$.

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

[1] Se utilizaron un total de 100000 observaciones, 50000 por clase, después de consultarlo con el profesor.

(a) 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.

Distancia	**Tasa de Aciertos**
ℓ_1	0.99978
ℓ_2	0.99979
ℓ_∞	0.99982

Tabla 1: Tasa de aciertos del algoritmo KNN aplicado a todo el conjunto de datos, utilizando $K = 7$, y las tres distancias ℓ_1,ℓ_2 y ℓ_∞, la tabla muestra los resultados de 1 corrida para cada distancia.

Comentario: Las tres distancias parecen generar una tasa muy alta de aciertos, las cuales son muy similares entre sí. Esto puede deberse a que si se ve la distribución de las clases en la **Figura** 7, se puede ver que hay poco traslape entre ellas, por lo que el algoritmo, sin importar la distancia utilizada, hace una clasificación efectiva. El hecho de que en ninguna ocasión se obtuvo una tasa de aciertos del 100%, es positivo, pues algunas pocas observaciones no son fácilmente clasificables, pues se encuentran traslapadas por completo.

(b) 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 código.

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

ii. Calcule la tasa de aciertos para las 3 configuraciones (distancia ℓ_1,ℓ_2 y ℓ_∞) 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.

	Tasa de Aciertos					
	Manhattan		Euclidean		Infinity	
Corrida	GPU	CPU	GPU	CPU	GPU	CPU
1	0.999800	0.999767	0.999800	0.999800	0.999767	0.999733
2	0.999900	0.999833	0.999867	0.999833	0.999867	0.999800
3	0.999667	0.999833	0.999700	0.999800	0.999733	0.999767
4	0.999833	0.999800	0.999833	0.999767	0.999833	0.999733
5	0.999700	0.999733	0.999733	0.999733	0.999667	0.999733
6	0.999800	0.999600	0.999800	0.999633	0.999800	0.999667
7	0.999700	0.999833	0.999700	0.999800	0.999700	0.999800
8	0.999867	0.999767	0.999833	0.999800	0.999800	0.999800
9	0.999800	0.999700	0.999833	0.999700	0.999833	0.999700
10	0.999833	0.999667	0.999833	0.999633	0.999833	0.999600
Media	**0.999790**	**0.999753**	**0.999793**	**0.999750**	**0.999783**	**0.999733**
St. dev.	**0.000077**	**0.000079**	**0.000060**	**0.000072**	**0.000065**	**0.000065**

Tabla 2: Tasa de aciertos de 10 corridas en GPU con las tres distancias implementadas. Las pruebas se efectuaron en ambientes de Google Colab, utilizando un GPU A100, mientras que el CPU utilizado fue el provisto en la instancia L4 (alta memoria). Todas las corridas en GPU y CPU utilizaron semillas aleatorias diferentes.

Comentario: Las tres distancias mantienen la tendencia vista anteriormente en la tabla 1, donde todas las distancias presentan una tasa muy alta de aciertos, en este

caso con una baja desviación estándar, lo cual parece indicar que el KNN presenta casi los mismos resultados sin importar la distancia utilizada. No parece existir una diferencia significativa entre los resultados de GPU y CPU.

iii. Calcule además el **tiempo de ejecución** por corrida para cada las tres distancias tanto en CPU como en GPU. Reporte los resultados en una tabla para las 10 corridas, junto con su media y desviación estándar y coméntelos.

Duración en segundos						
	Manhattan		Euclidean		Infinity	
Corrida	GPU	CPU	GPU	CPU	GPU	CPU
1	8.086052	21.610155	7.471702	24.006129	6.696899	18.805519
2	8.071934	20.520020	7.450645	24.019213	6.677324	18.755920
3	8.194681	20.791444	7.468749	24.013317	6.674734	18.379076
4	8.067917	20.350141	7.519531	23.636163	6.690659	18.966814
5	8.088926	20.234142	7.452440	24.011520	6.705777	19.095105
6	8.115149	20.754730	7.476169	24.103350	6.715930	18.729428
7	8.072658	20.381359	7.662809	23.968350	6.503448	19.104312
8	8.052628	20.497603	7.686365	23.902102	6.516738	18.150882
9	8.092654	20.341320	7.627234	23.946782	6.501912	18.267952
10	8.062700	20.645990	7.618219	23.823238	6.699250	18.936353
Media	**8.090530**	20.612691	**7.543386**	23.943016	**6.638267**	18.719136
St. dev.	**0.040645**	0.395726	**0.094283**	0.131642	**0.091217**	0.341449

Tabla 3: Duración en segundos de 10 corridas tanto en GPU como en CPU, con sus medias y desviaciones estándar. Las pruebas se efectuaron en ambientes de Google Colab, utilizando un GPU A100, mientras que el CPU utilizado fue el provisto en la instancia L4 (alta memoria). Todas las corridas en GPU y CPU utilizaron semillas aleatorias diferentes.

Comentario: El KNN en GPU parece ser más rápido que en CPU, por al menos 1 orden de magnitud. Para todas las distancias utilizadas, los resultados en GPU son superiores, lo cual es consecuente con la implementación matricial del algoritmo KNN y el hecho que los GPUs pueden realizar operaciones matriciales de forma más eficiente y paralela que en CPU.

A. Realice una gráfica comparativa.

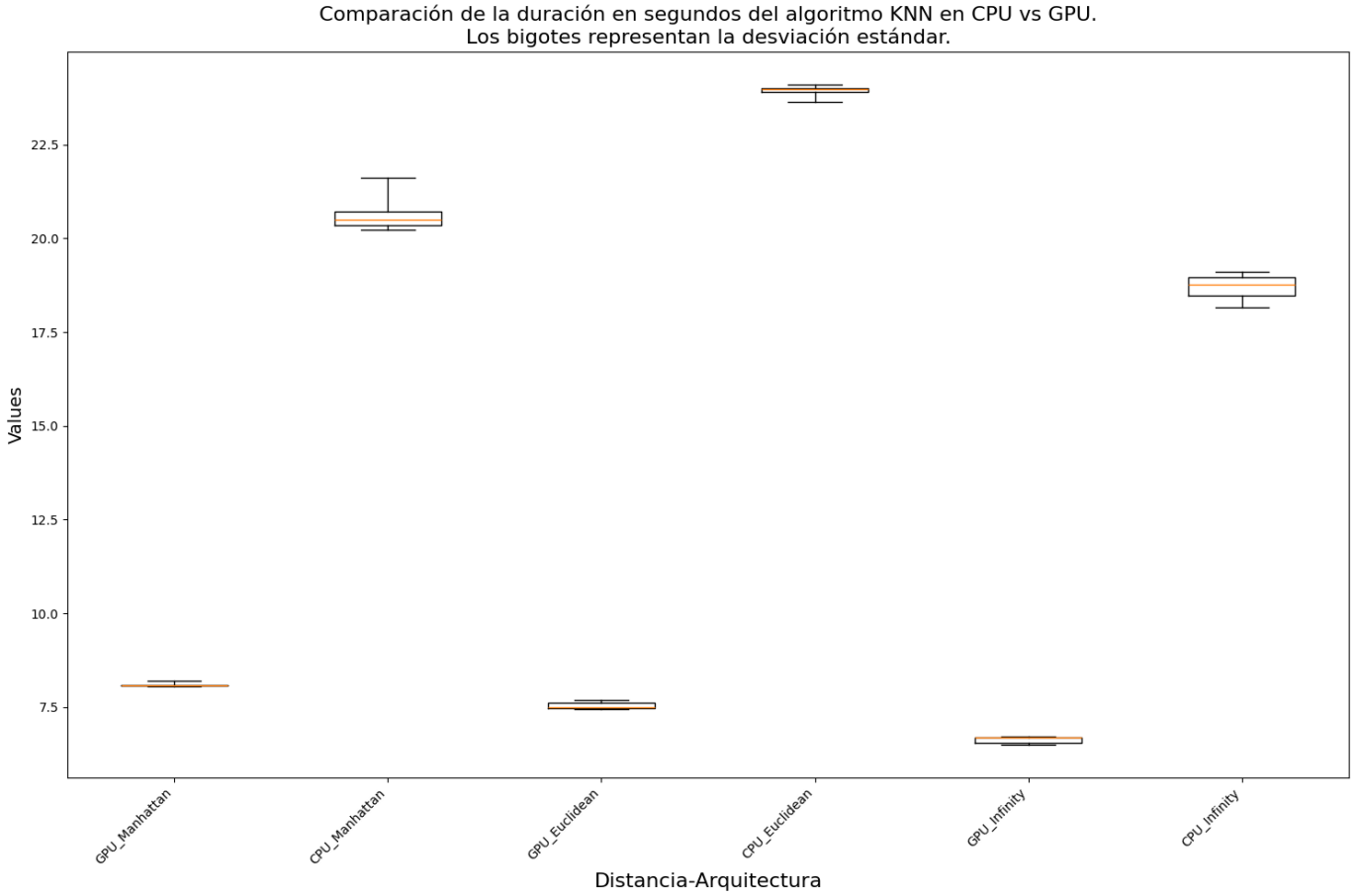

Figura 8: Comparación de la duración en segundos del algoritmo KNN ejecutado en GPU (A100) versus CPU (L4) en Google Colab. Los bigtotes representan la desviación estándar para cada corrida.

B. ¿Cuál distancia resulto más eficiente?

R/ La distancia ℓ_∞ (infinita) parece ser más eficiente con respecto a la duración en segundos, tanto en GPU como en CPU.

C. Hay algún costo en cuanto a la tasa de aciertos? Explique el porque de la diferencia en la tasa de aciertos, si la hay.

R/ No se puede determinar con los resultados obtenidos que se haya sacrificado la tasa de aciertos de ninguna forma, ni por la distancia utilizada ni por el tipo de arquitectura (GPU o CPU). Para todas las configuraciones del experimento se obtuvieron resultados que parecen ser equivalentes entre sí, data sus altas tasas de aciertos superiores a 0.999.

4. Implemente el algoritmo Condensed Nearest Neighbors (CNN) para implementar de forma mas rápida el algoritmo de K-vecinos más cercano. El objetivo del algoritmo es eliminar instancias redundantes manteniendo solo los puntos críticos para preservar la estructura de decisión, y realizar inferencia mas rápido.

   (a) El algoritmo funciona como sigue, usando un conjunto de datos de entrenamiento T:

      i. Comienza con un conjunto vacío condensado C.

      ii. Escoge aleatoriamente K elementos y se agregan a C.

      iii. Para cada dato $\overrightarrow{x}_i \in T$, que al ejecutarse el algoritmo KNN usando los datos en C, es incorrectamente clasificado, usando como etiqueta la estimación de KNN usando T (**inconsistencia con el conjunto de datos de entrenamiento**), se agrega su vecino más cercano en T, hasta que la inconsistencia se arregle. De esta forma se podan los elementos que no aportan mayor información.

[ ]:
```python
def get_random_sample(data, labels, K=7):
  # data: dataset de features
  # labels: etiquetas de los features
  # k: numero de observaciones a seleccionar aleatoriamente
  # return: 3 tensores, uno con los features seleccionados, las etiquetas y
los indices seleccionados

  # Genera una permutacion aleatoria de todas las observaciones
  random_indices = torch.randperm(data.shape[0])

  # Selecciona los primeros K índices aleatorios
  selected_indices = random_indices[:K]

  return data[selected_indices], labels[selected_indices], selected_indices

def get_distancia(X, v, p=float('inf')):
  if p == float('inf'):
      return torch.max(torch.abs(X - v), dim=1).values.unsqueeze(0)
  return torch.pow(torch.sum(torch.pow(torch.abs(X - v), p), dim=1), 1/p).
↪unsqueeze(0)
```

[ ]:
```python
def cnn_clasify_observation(C, C_targets, test_observation, p, K):
  # Calcula la distancia entre la observacion de entrada y el dataset de
entrenamiento
  dist = get_distancia(C, test_observation, p)

  # Se seleccionan los K indices con las menores distancias al conjunto
condensado C
  k_vecinos = dist.topk(K, largest=False)

  # Extraccion de los labels de los vecinos mas cercanos
  target_labels = C_targets[k_vecinos.indices[0]]

  # La moda se puede usar para obtener el valor más repetido
  # Este valor se convierte en la etiqueta estimada
  return torch.mode(target_labels, 0).values.unsqueeze(0)
```

[ ]:
```python
def evaluate_cnn_test_dataset(data_training, labels_training, test_dataset,
k=7, p=float('inf')):
```

```
  # calcula el cnn para cada observacion del test_dataset
  t_estimated_dataset = []
  C, C_targets, indices_agregados = get_random_sample(data_training,
labels_training, k)
  mascara = torch.ones(data_training.shape[0], dtype=bool)
  mascara[indices_agregados] = False

  for i, test_observation in tqdm(enumerate(test_dataset)):
    t_consistente = False
    test_target = None
    test_target = cnn_clasify_observation(C, C_targets, test_observation, p,
k)
    t_consistente = torch.equal(test_target, labels_training[i])

    if not t_consistente:
      training_filtrado = data_training[mascara]
      labels_filtrado = labels_training[mascara]
      dist = get_distancia(training_filtrado, test_observation, p)
      k_vecinos = dist.topk(2, largest=False)

      # Busca el siguiente vecino mas cercano que no sea la misma observacion
      nuevo_vecino = training_filtrado[k_vecinos.indices.squeeze()[-1]].
↪unsqueeze(0)
      etiqueta_nuevo_vecino = labels_filtrado[k_vecinos.indices.
↪squeeze()[-1]].unsqueeze(1)

      # Revisa que el nuevo vecino no este en C antes de agregarlo
      if not torch.any(torch.all(torch.eq(C, nuevo_vecino), dim=1)):
        C = torch.cat((C, nuevo_vecino), dim=0)
        C_targets = torch.cat((C_targets, etiqueta_nuevo_vecino), dim=0)

      mascara[k_vecinos.indices.squeeze()[-1]] = False

    # Agrega valor estimado
    t_estimated_dataset.append(test_target)
  # retorna el tensor condensado, sus etiquetas y las etiquetas estimadas
  return C, C_targets, torch.tensor(t_estimated_dataset).to(data_training.
↪device)
```

(b) Pruebe usando los mismos datos de prueba de la función anterior, las 3 distancias usadas anteriormente. Use $K = 7$. Documente el tamaño del dataset generado S. Hágalo para 30 corridas y compare los resultados con el algoritmo original tanto en tasa de aciertos como tiempo de ejecución en inferencia.

Tamaño conjunto S

Distancia	Manhattan	Euclidean	Infinity
Media	23089.73	23066.03	23102.30
Std. Dev.	61.8161	58.6112	73.6118

Tabla 4: Tamaño del conjunto condensado S generado por el algoritmo de CNN, para cada una de las métricas de distancias implementadas.

Tasa de Aciertos						
Distancia	Manhattan		Euclidean		Infinity	
	CNN	KNN	CNN	KNN	CNN	KNN
Media	0.999671	0.999790	0.999691	0.999793	0.999700	0.999783
Std. Dev	0.000086	0.000077	0.000069	0.000060	0.000090	0.000065

Tabla 5: Comparación de las tasas de aciertos promedio con s.d. obtenidas por CNN (30 corridas) y KNN (10 corridas), ambas corridas en GPU, para cada una de las métricas de distancia.

	Duración en segundos					
Distancia	Manhattan		Euclidean		Infinity	
	CNN	KNN	CNN	KNN	CNN	KNN
Media	132.150336	**8.090530**	141.161563	**7.543386**	128.240465	**6.638267**
Std. Dev.	0.918917	**0.040645**	1.417119	**0.094283**	0.896052	**0.091217**

Tabla 6: Comparación de la duración en segundos promedio con s.d. obtenida tanto para CNN (30 corridas), como para KNN (10 corridas), ambas corridas en GPU, para cada una de las métricas de distancia.

Comentario: La tabla 4 evidencia como con aproximadamente un 76% de los datos se puede realizar la clasificación del conjunto de datos original, lo cual cumple con la ventaja propuesta del algoritmo de clasificación Condensed Nearest Neighbors (CNN).

Al comparar las tasas de aciertos obtenidas tanto por CNN como por KNN, según muestra la tabla 5, no se puede determinar que un algoritmo sea mejor que el otro, lo cual parece indicar que el algoritmo de CNN, utilizando menos datos puede ser equivalente al KNN.

Al analizar el tiempo de ejecución de ambos algoritmos, según la tabla 6, se puede observar como nuestra implementación del CNN, fue mucho más lento que el KNN, creemos que esto es porque la implementación del CNN es menos paralelizada que el KNN, pues debemos analizar por cada observación individualmente para determinar si el conjunto S es **consistente con el conjunto de entrenamiento**.

Sin embargo, una vez entrenado el algoritmo de CNN, creemos que se pueden realizar inferencias extremadamente veloces (si no se necesita actualizar S), y el modelo consume menos memoria que el KNN, debido a que $S \subseteq T$, donde T es el conjunto de entrenamiento. En el peor de los casos, si $S \equiv T$, el consumo de memoria de CNN es igual al del KNN.