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

Trabajo práctico 1:

Vectores y Calculo Multi-variable

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 abordarán dos problemas fundamentales dentro del campo del análisis multivariable y la clasificación supervisada. Ambos problemas requieren 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 primer problema trata sobre el análisis de funciones lineales multivariable y el cálculo del gradiente. Este estudio es esencial en diversas áreas de las matemáticas aplicadas, como la optimización y el aprendizaje automático. Mediante el uso de **Pytorch**, se graficarán planos correspondientes a funciones lineales multivariables en R^2, así como los vectores normales asociados. Además, se calculará el vector gradiente y su magnitud en puntos específicos para diferentes funciones, junto con la matriz Hessiana que describe la curvatura de estas superficies. La comprensión y manipulación de estas herramientas permiten un análisis detallado de cómo varían las funciones en su dominio.

El segundo problema involucra la implementación del algoritmo de K-vecinos más cercanos **(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. Funciones multivariable

(10 puntos) Funciones lineales multivariable: un hiper-plano definido en un espacio R^{n+1} se puede expresar como una función con dominio $\vec{x} \in R^n$ y condominio en R como sigue:

$$z = f(\vec{x}) = \vec{x} \cdot \vec{w}, \text{ con } \vec{w} \in R^n \text{ el arreglo de coeficientes de tal funcional.}$$

2.1.1. Subproblema *a*

Tómese $\vec{w}_1 = \begin{bmatrix} 0{,}5 \\ 0{,}2 \end{bmatrix}$ para la función f_1 y $\vec{w}_2 = \begin{bmatrix} -0{,}1 \\ 0{,}05 \end{bmatrix}$ para la función f_2, ambas con dominio en R^2 y condominio en R.

Se consideran dos funciones lineales, cada una definida por un hiper-plano en R^2 con vectores normales $\vec{w}_1$ y $\vec{w}_2$. Las funciones están dadas por:

$$f_1(\vec{x}) = \vec{w}_1 \cdot \vec{x}, \quad f_2(\vec{x}) = \vec{w}_2 \cdot \vec{x}$$

donde $\vec{x} = (x_1, x_2)$.

Vectores Normales

Los vectores normales correspondientes a cada hiper-plano son:

$$\vec{w}_1 = \begin{bmatrix} 0{,}5 \\ 0{,}2 \end{bmatrix}$$
$$\vec{w}_2 = \begin{bmatrix} -0{,}1 \\ 0{,}05 \end{bmatrix}$$

Ecuaciones de los Planos

Las ecuaciones de los planos están dadas por:

- Para $f_1(\vec{x})$:
$$z = 0{,}5x_1 + 0{,}2x_2$$
- Para $f_2(\vec{x})$:
$$z = -0{,}1x_1 + 0{,}05x_2$$

Gráfico de los Planos en PyTorch

Los planos fueron graficados en un sistema de coordenadas 3D, utilizando PyTorch y Matplotlib. Los gráficos incluyen la visualización de los planos en el espacio (x_1, x_2, z).

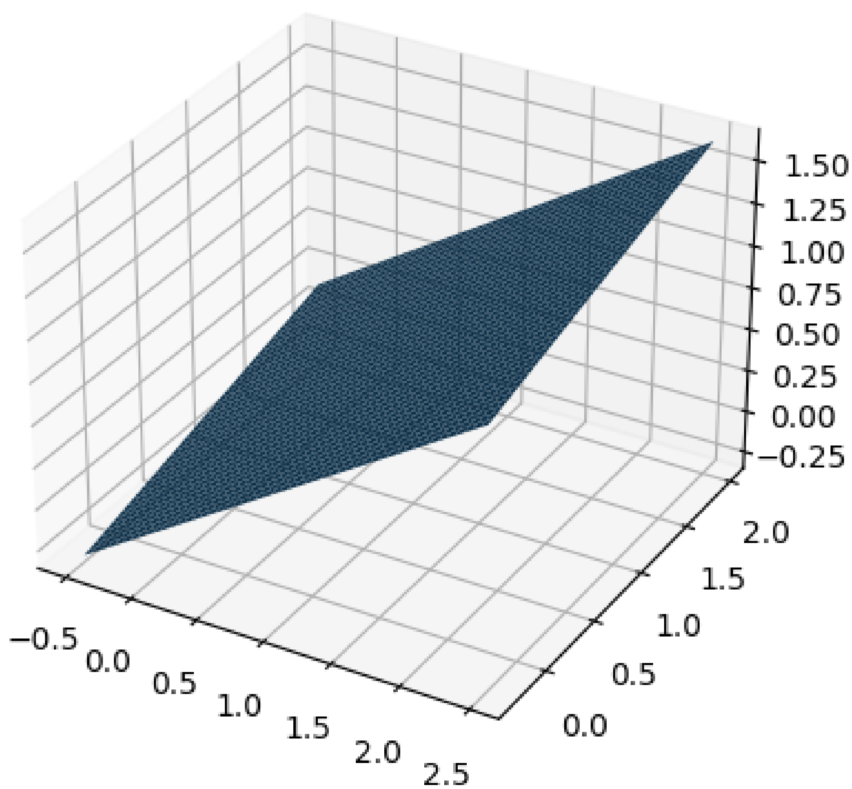


Figura 2.1: Grafica del plano 1

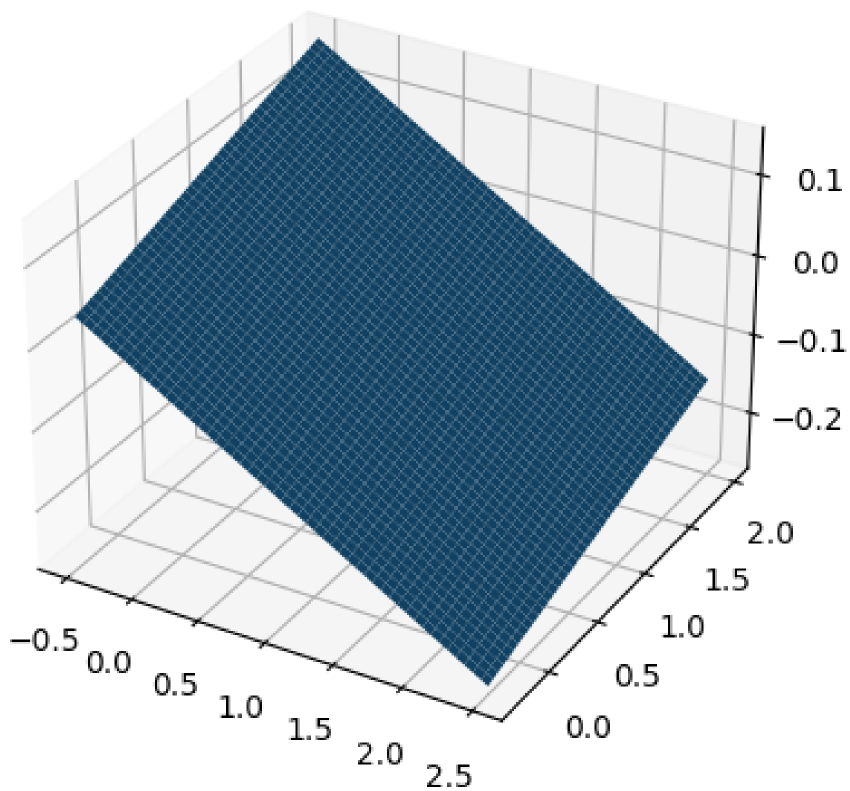


Figura 2.2: Gráfica del plano 2

2.1.2. Subproblema *b*

Para cada plano, se calcula el vector normal en el punto $P = (1, 1)$ y se grafica una curva de nivel perpendicular a dicho vector normal.

Curva de Nivel y Vector Normal para $f_1(\vec{x})$

En el punto $P = (1, 1)$, el vector normal es $\vec{w}_1$. La curva de nivel es graficada en el plano x_1x_2 de forma perpendicular a este vector:

$$\vec{w}_1 = \begin{bmatrix} 0{,}5 \\ 0{,}2 \end{bmatrix}$$

y su proyección sobre el plano es representada en la gráfica de las curvas de nivel.

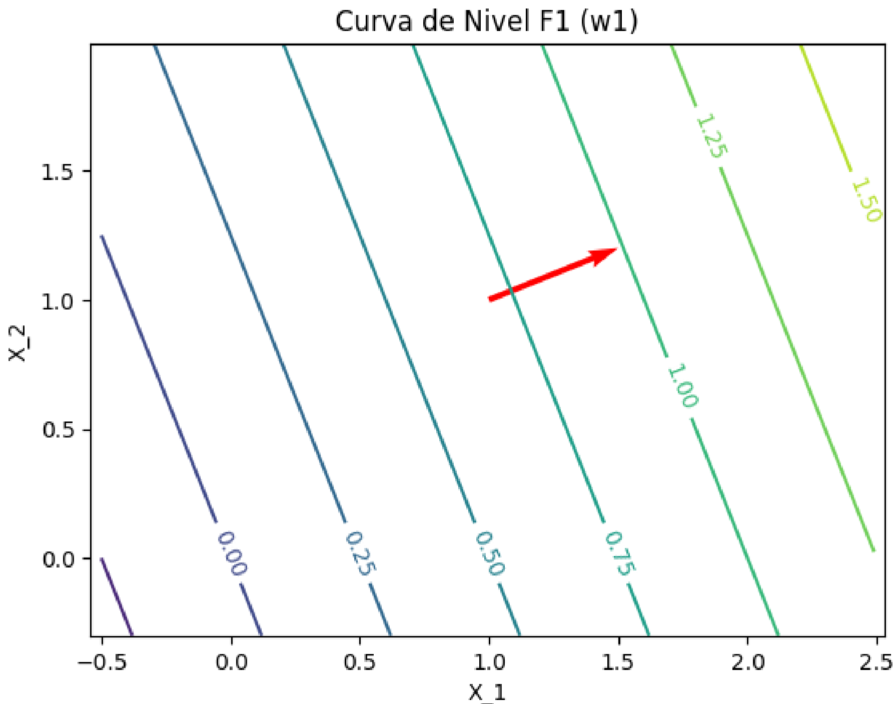


Figura 2.3: Curva de Nivel $f_1(\vec{x})$

Curva de Nivel y Vector Normal para $f_2(\vec{x})$

En el punto $P = (1, 1)$, el vector normal es $\vec{w}_2$. De forma similar, la curva de nivel se grafica perpendicularmente a este vector:

$$\vec{w}_2 = \begin{bmatrix} -0{,}1 \\ 0{,}05 \end{bmatrix}$$

y se proyecta en el gráfico de curvas de nivel.

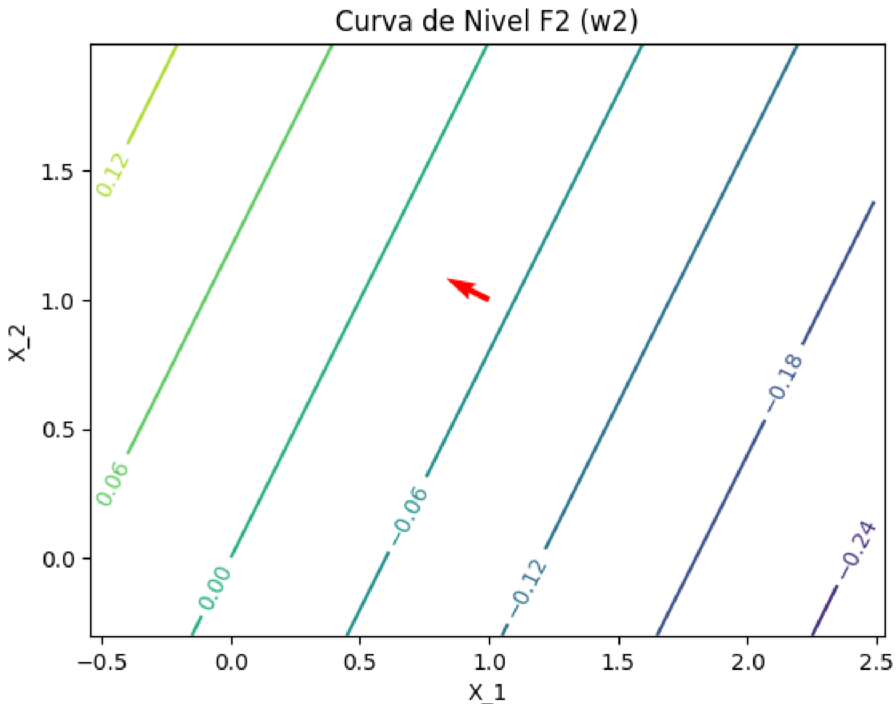


Figura 2.4: Curva de Nivel $f_2(\vec{x})$

Código Utilizado

El código Python implementado, utilizando PyTorch y Matplotlib, gráfica las superficies de los planos y las curvas de nivel con los vectores normales correspondientes en cada uno de los puntos de evaluación.

```
import torch as torch
import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
#Vectores normales w1 y w2
w1 = torch.tensor([0.5, 0.2])
w2 = torch.tensor([-0.1, 0.05])
#crear 1D tensores
step = 0.01
x_1 = torch.arange(-0.5, 2.5, step)
x_2 = torch.arange(-0.3, 2.0, step)
#Crear 2D tensors con la variacion en los ejes
```

```
X_1, X_2 = torch.meshgrid(x_1, x_2)

# funciones f1 y f2
f1 = w1[0] * X_1 + w1[1] * X_2
f2 = w2[0] * X_1 + w2[1] * X_2

# Graficar plano f1
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X_1.numpy(), X_2.numpy(), f1.numpy())
plt.show()

# Graficar plano f2
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X_1.numpy(), X_2.numpy(), f2.numpy())
plt.show()


# Curva de nivel f1
plt.title('Curva de Nivel F1 (w1)')
plt.xlabel('X_1')
plt.ylabel('X_2')
contours = plt.contour(X_1.numpy(), X_2.numpy(), f1.numpy())
#Vector en plano f1
plt.quiver(1, 1, w1[0], w1[1], angles='xy', scale_units='xy',
↪  color=['r','b','g'], scale=1)
plt.axis('equal')
plt.clabel(contours, inline=1, fontsize=10)
plt.show()

# Curva de nivel f2
plt.title('Curva de Nivel F2 (w2)')
plt.xlabel('X_1')
plt.ylabel('X_2')
contours = plt.contour(X_1.numpy(), X_2.numpy(), f2.numpy())
#Vector en plano f2
plt.quiver(1, 1, w2[0], w2[1], angles='xy', scale_units='xy',
↪  color=['r','b','g'], scale=0.6)
plt.axis('equal')
plt.clabel(contours, inline=1, fontsize=10)
plt.show()
```

2.2. El Vector Gradiente

(30 puntos) El vector gradiente: Para cada una de las siguientes funciones multivariable: (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) y (3) Calcule la magnitud de tal vector gradiente en cada punto (4) Calcule lo que se conoce como la matriz Hessiana.

2.2.1. Función $f(x, y) = x^3y^2 + 1$

El gradiente será evaluado en los puntos $P_0 = (0, 0)$ y $P_1 = (7{,}4, -6{,}3)$.

Gráfico de la superficie con dominio entre -10, 10

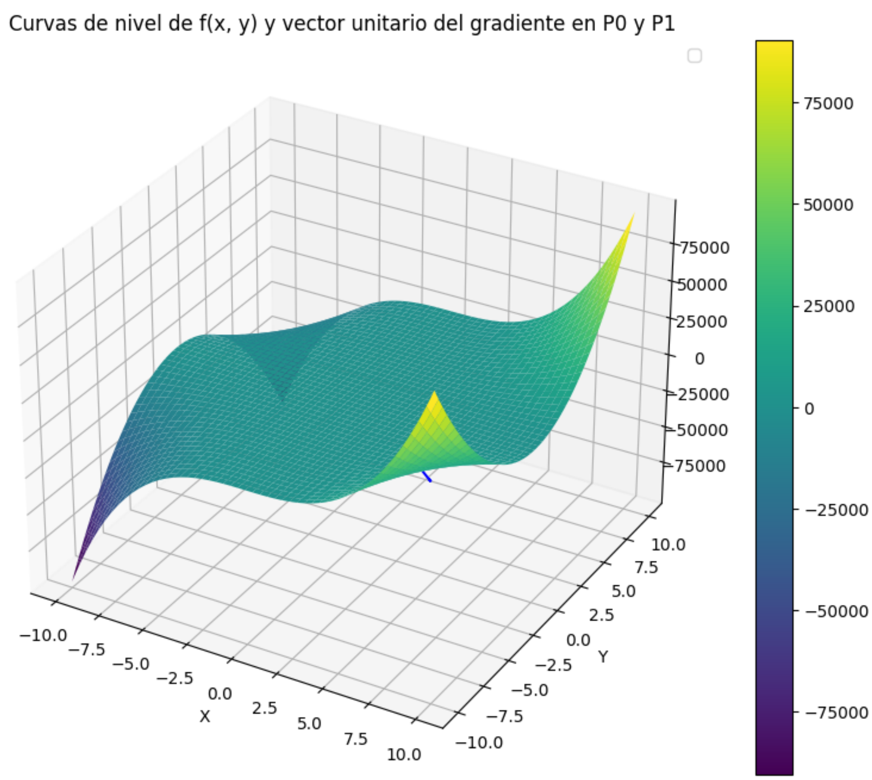


Figura 2.5: Gráfica de la superficie de $f(x, y) = x^3y^2 + 1$

Calculo del vector gradiente

Dada la función $f(x,y) = x^3y^2 + 1$, calculamos las derivadas parciales para obtener el gradiente:

$$\frac{df}{dx} = 3x^2y^2, \quad \frac{df}{dy} = 2x^3y$$

El gradiente es:

$$\nabla f(x,y) = (3x^2y^2, 2x^3y)$$

Evaluación en los puntos:

- Para el punto $P_0 = (0,0)$:

$$\nabla f(0,0) = (3(0)^2(0)^2, 2(0)^3(0)) = (0,0)$$

- Para el punto $P_1 = (7{,}4, -6{,}3)$:

$$\nabla f(7{,}4, -6{,}3) = (3(7{,}4)^2(-6{,}3)^2, 2(7{,}4)^3(-6{,}3)) = (6520{,}27, -5105{,}82)$$

Gráfica del vector unitario para ambos puntos

El vector unitario en la dirección del gradiente se obtiene dividiendo cada componente del gradiente por la magnitud.

Evaluación en los puntos:

- Para $P_0 = (0,0)$, dado que la magnitud es cero, el vector unitario también es cero:

$$\vec{u_0} = (0,0)$$

- Para $P_1 = (7{,}4, -6{,}3)$:

$$u_x = \frac{6520{,}27}{8281{,}51} = 0{,}787, \quad u_y = \frac{-5105{,}82}{8281{,}51} = -0{,}616$$

Por lo tanto, el vector unitario es:

$$\vec{u_1} = (0{,}787, -0{,}616)$$

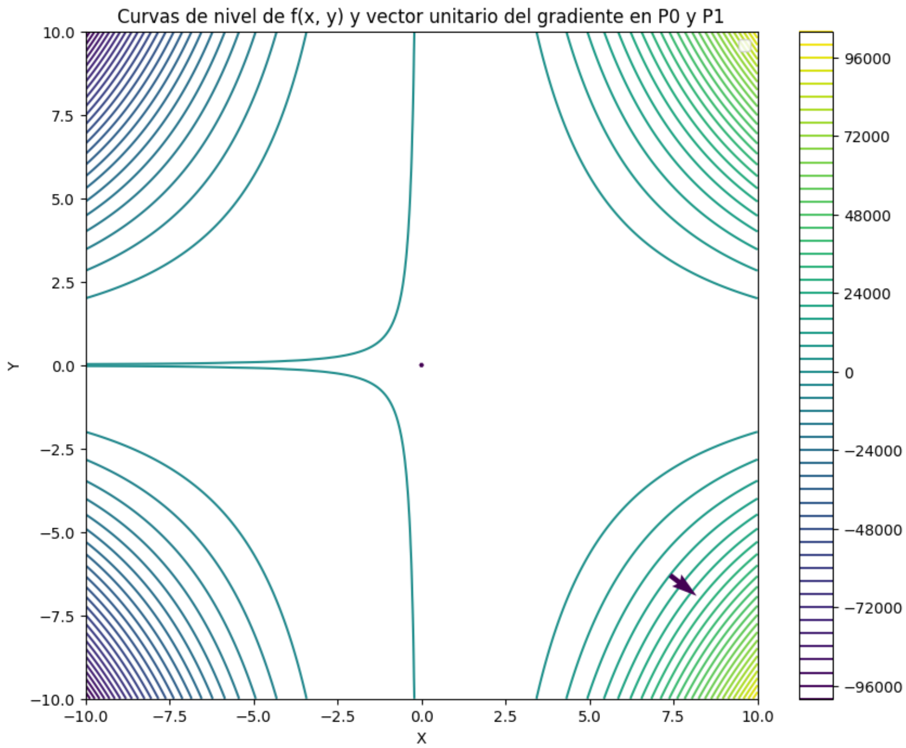


Figura 2.6: Gráfica con el vector unitario de $f(x, y) = x^3y^2 + 1$

Magnitud del vector gradiente en cada punto

La magnitud del vector gradiente se calcula usando la fórmula:

$$|\nabla f(x, y)| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

Evaluación en los puntos:

- Para $P_0 = (0, 0)$:
$$|\nabla f(0, 0)| = \sqrt{(0)^2 + (0)^2} = 0$$

- Para $P_1 = (7{,}4, -6{,}3)$:

$$|\nabla f(7{,}4, -6{,}3)| = \sqrt{(6520{,}27)^2 + (-5105{,}82)^2} = 8281{,}51$$

Calculo de la Matriz Hessiana

La matriz Hessiana se compone de las segundas derivadas parciales de la función:

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

Las segundas derivadas parciales son:

$$\frac{\partial^2 f}{\partial x^2} = 6xy^2, \quad \frac{\partial^2 f}{\partial y^2} = 2x^3$$

La derivada mixta es:

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial^2 f}{\partial y \partial x} = 6x^2 y$$

Por lo tanto, la matriz Hessiana es:

$$H(f) = \begin{pmatrix} 6xy^2 & 6x^2y \\ 6x^2y & 2x^3 \end{pmatrix}$$

2.2.2. Función $f(x, y) = \sin(x^2) + x\cos(y^3)$

El gradiente será evaluado en los puntos $P_0 = (1{,}5, -5{,}5)$ y $P_1 = (-10, -10)$.

Gráfico de la superficie con dominio entre -10, 10

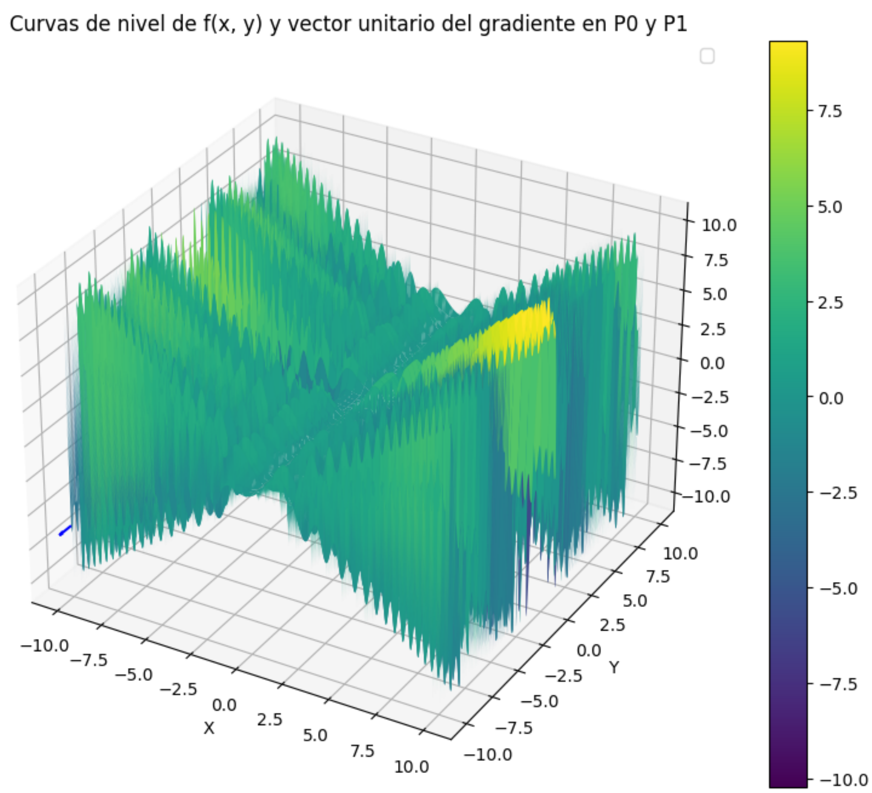


Figura 2.7: Gráfica de la superficie de $f(x, y) = \sin(x^2) + x\cos(y^3)$

Cálculo del vector gradiente

Dada la función $f(x, y) = \sin(x^2) + x\cos(y^3)$, calculamos las derivadas parciales para obtener el gradiente:

$$\frac{\partial f}{\partial x} = 2x\cos(x^2) + \cos(y^3)$$

$$\frac{\partial f}{\partial y} = -3xy^2\sin(y^3)$$

Por lo tanto, el gradiente es:

$$\nabla f(x,y) = \big(2x\cos(x^2) + \cos(y^3), -3xy^2\sin(y^3)\big)$$

Evaluación en los puntos:

- Para el punto $P_0 = (1{,}5, -5{,}5)$:

$$\nabla f(1{,}5,-5{,}5) = \big(2(1{,}5)\cos((1{,}5)^2) + \cos((-5{,}5)^3), -3(1{,}5)(-5{,}5)^2\sin((-5{,}5)^3)\big)$$
$$= (-2{,}8761, 17{,}5668)$$

- Para el punto $P_1 = (-10, -10)$:

$$\nabla f(-10,-10) = \big(2(-10)\cos((-10)^2) + \cos((-10)^3), -3(-10)(-10)^2\sin((-10)^3)\big)$$
$$= (-16{,}6839, -2480{,}6947)$$

Gráfico del vector unitario para ambos puntos

El vector unitario en la dirección del gradiente se calcula dividiendo cada componente del gradiente entre la magnitud correspondiente.

Evaluación en los puntos:

- En el Punto $P_0 = (1{,}5, -5{,}5)$:

$$u_x = \frac{-2{,}8761}{17{,}8006}, \quad u_y = \frac{17{,}5668}{17{,}8006}$$
$$u_x = -0{,}1615, \quad u_y = 0{,}9872$$

  Por lo tanto, el vector unitario es:

$$\vec{u_0} = (-0{,}1615, 0{,}9872)$$

- En el Punto $P_1 = (-10, -10)$:

$$u_x = \frac{-16{,}6839}{2480{,}6947}, \quad u_y = \frac{-2480{,}6947}{2480{,}6947}$$
$$u_x = -0{,}0067, \quad u_y = -1{,}0000$$

  Por lo tanto, el vector unitario es:

$$\vec{u_1} = (-0{,}0067, -1{,}0000)$$

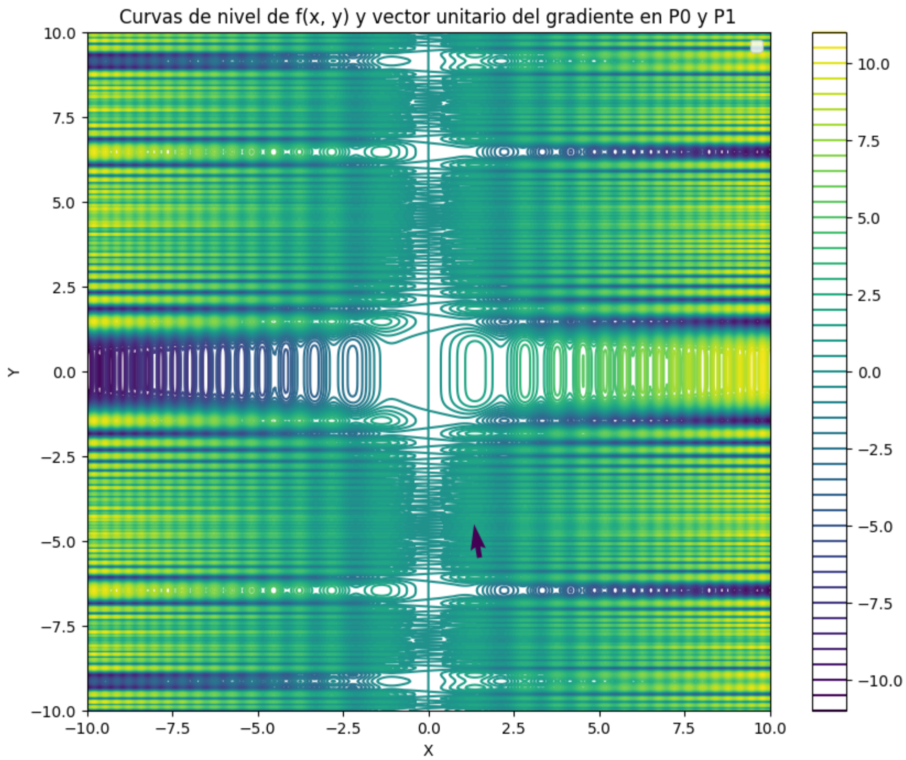


Figura 2.8: Gráfica con el vector unitario de $f(x, y) = \sin(x^2) + x\cos(y^3)$

Magnitud del vector gradiente en cada punto

La magnitud del gradiente en el Punto $P_0 = (1{,}5, -5{,}5)$:

$$|\nabla f(1{,}5, -5{,}5)| = \sqrt{(-2{,}8761)^2 + (17{,}5668)^2} = 17{,}8006$$

Y la magnitud en el Punto $P_1 = (-10, -10)$:

$$|\nabla f(-10, -10)| = \sqrt{(-16{,}6839)^2 + (-2480{,}6947)^2} = 2480{,}6947$$

Cabe recalcar que por el dominio y la dirección que se muestra en la magnitud del vector unitario no es posible que se visualice en la gráfica.

Cálculo de la Matriz Hessiana

La matriz Hessiana está compuesta por las segundas derivadas parciales. Para la función $f(x, y) = \sin(x^2) + x\cos(y^3)$, tenemos:

$$\frac{\partial^2 f}{\partial x^2} = 2\cos(x^2) - 4x^2\sin(x^2)$$

$$\frac{\partial^2 f}{\partial y^2} = -9xy^4\cos(y^3) - 6xy\sin(y^3)$$

La derivada mixta es:

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial}{\partial y}\left(2x\cos(x^2) + \cos(y^3)\right) = -3x^2\sin(y^3)$$

$$\frac{\partial^2 f}{\partial y \partial x} = \frac{\partial}{\partial x}\left(-3xy^2\sin(y^3)\right) = -3y^2\sin(y^3)$$

Por lo tanto, la Matriz Hessiana es:

$$H(f) = \begin{pmatrix} 2\cos(x^2) - 4x^2\sin(x^2) & -3x^2\sin(y^3) \\ -3y^2\sin(y^3) & -9xy^4\cos(y^3) - 6xy\sin(y^3) \end{pmatrix}$$

2.2.3. Función $f(x, y) = 3^{2x} + 5^{4y} + 2x + y^4$

El gradiente será evaluado en los puntos $P_0 = (-4, -2)$ y $P_1 = (-2, 9)$.

Gráfico de la superficie con dominio entre -10, 10

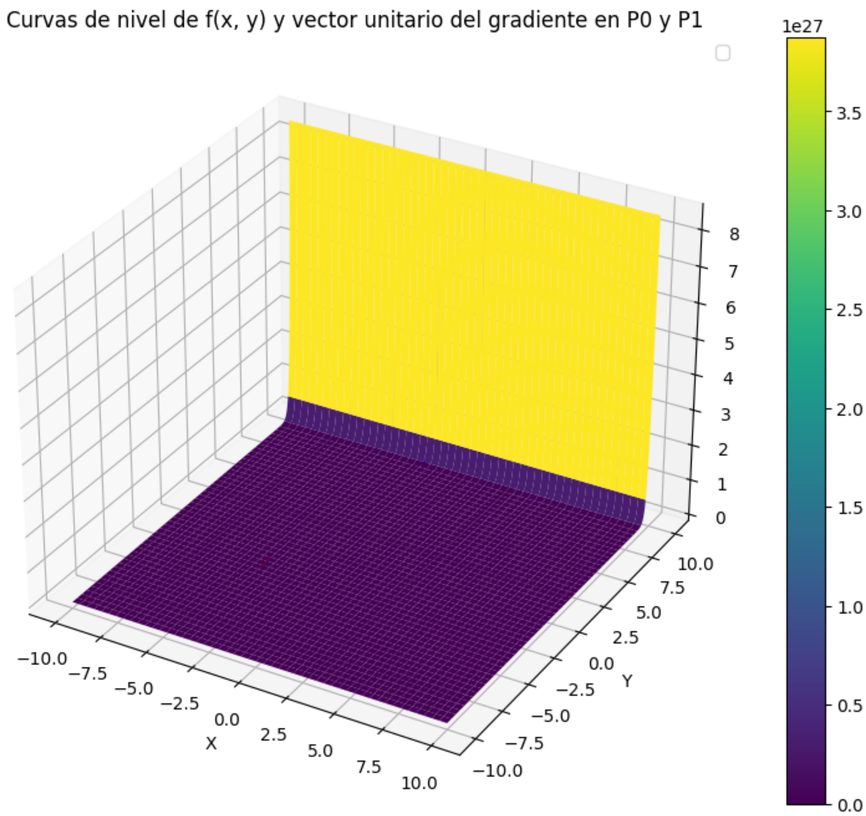


Figura 2.9: Gráfica de la superficie de $f(x, y) = 3^{2x} + 5^{4y} + 2x + y^4$

Cálculo del vector gradiente

Dada la función $f(x, y) = 3^{2x} + 5^{4y} + 2x + y^4$, calculamos las derivadas parciales para obtener el gradiente:

$$\frac{\partial f}{\partial x} = 2 \cdot 3^{2x} \ln(3) + 2$$

$$\frac{\partial f}{\partial y} = 4 \cdot 5^{4y} \ln(5) + 4y^3$$

Por lo tanto, el gradiente es:

$$\nabla f(x,y) = \left(2 \cdot 3^{2x}\ln(3) + 2, 4 \cdot 5^{4y}\ln(5) + 4y^3\right)$$

Evaluación en los puntos:

- Para el punto $P_0 = (-4,-2)$:

$$\nabla f(-4,-2) = \left(2 \cdot 3^{2(-4)}\ln(3) + 2, 4 \cdot 5^{4(-2)}\ln(5) + 4(-2)^3\right)$$
$$= (2{,}0603, -31{,}99998)$$

- Para el punto $P_1 = (-2,9)$:

$$\nabla f(-2,9) = \left(2 \cdot 3^{2(-2)}\ln(3) + 2, 4 \cdot 5^{4(9)}\ln(5) + 4(9)^3\right)$$
$$= (2{,}027126, 9{,}368161 \times 10^{25})$$

Gráfico del vector unitario para ambos puntos

El vector unitario en la dirección del gradiente se calcula dividiendo cada componente del gradiente entre la magnitud correspondiente.

Evaluación en los puntos:

- En el Punto $P_0 = (-4,-2)$:

$$u_x = \frac{2{,}0603}{31{,}0064}, \quad u_y = \frac{-31{,}99998}{31{,}0064}$$
$$u_x = 0{,}0665, \quad u_y = -0{,}9993$$

Por lo tanto, el vector unitario es:

$$\vec{u_0} = (0{,}0665, -0{,}9993)$$

- En el Punto $P_1 = (-2,9)$:

$$u_x = \frac{2{,}027126}{9{,}368161 \times 10^{25}}, \quad u_y = \frac{9{,}368161 \times 10^{25}}{9{,}368161 \times 10^{25}}$$
$$u_x \approx 2{,}163 \times 10^{-26}, \quad u_y = 1$$

Por lo tanto, el vector unitario es:

$$\vec{u_1} = (2{,}163 \times 10^{-26}, 1)$$

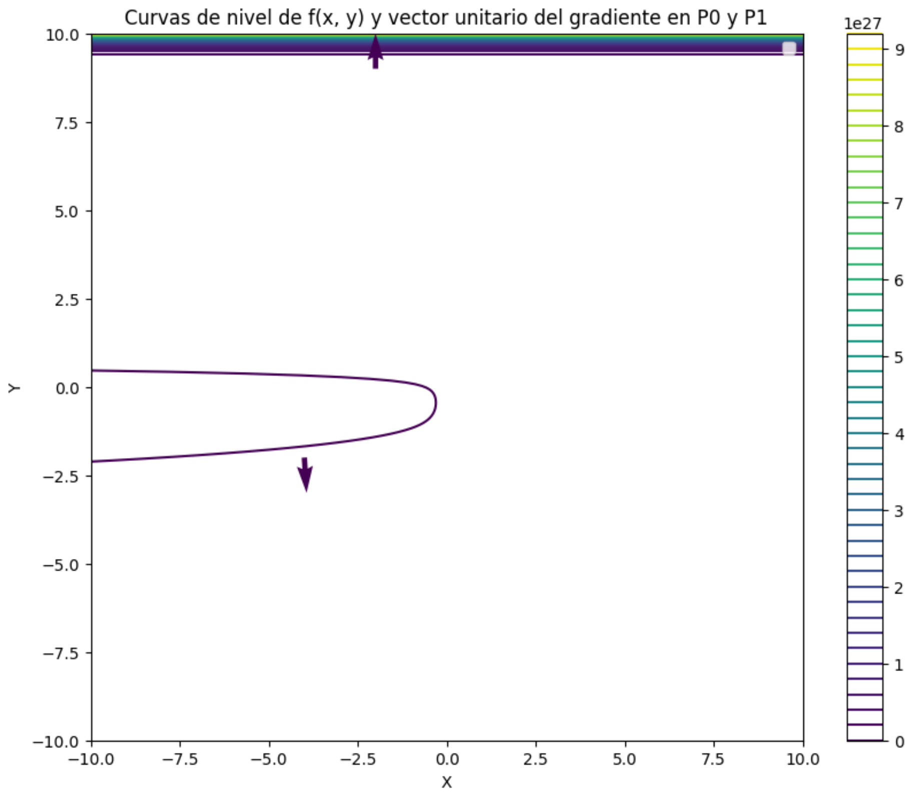


Figura 2.10: Grafica con el vector unitario de $f(x, y) = 3^{2x} + 5^{4y} + 2x + y^4$

Magnitud del vector gradiente en cada punto

La magnitud del gradiente en el Punto $P_0 = (-4, -2)$:

$$|\nabla f(-4, -2)| = \sqrt{(2{,}0603)^2 + (-31{,}99998)^2} = 31{,}0064$$

Y la magnitud en el Punto $P_1 = (-2, 9)$:

$$|\nabla f(-2, 9)| = \sqrt{(2{,}027126)^2 + (9{,}368161 \times 10^{25})^2} \approx 9{,}368161 \times 10^{25}$$

Cálculo de la Matriz Hessiana

La matriz Hessiana está compuesta por las segundas derivadas parciales. Para la función $f(x, y) = 3^{2x} + 5^{4y} + 2x + y^4$, tenemos:

$$\frac{\partial^2 f}{\partial x^2} = 4 \cdot 3^{2x} \ln^2(3)$$

$$\frac{\partial^2 f}{\partial y^2} = 16 \cdot 5^{4y} \ln^2(5) + 12y^2$$

La derivada mixta es:

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial^2 f}{\partial y \partial x} = 0$$

Por lo tanto, la Matriz Hessiana es:

$$H(f) = \begin{pmatrix} 4 \cdot 3^{2x} \ln^2(3) & 0 \\ 0 & 16 \cdot 5^{4y} \ln^2(5) + 12y^2 \end{pmatrix}$$

Código para gráfica de superficies

En esta parte se implementaron dos versiones de codigo, la primera muestra como se despliega la superficie en 3D y la segunda parte es para graficar en 2D las funciones correspondientes.

Código para despliegue en 3D

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import interp2d
import math
import torch
# Crear una malla de puntos entre -10 y 10
dominio_1 = torch.arange(-10, 10, 0.01)
dominio_2 = torch.arange(-10, 10, 0.01)
X_1, X_2 = torch.meshgrid(dominio_1, dominio_2)
# Definir las funciones
# Function 1: (x**3) * (y**2) + 1
def function_1(x, y):
    return (x**3) * (y**2) + 1
# Function 2: np.sin(x**2) + (x * np.cos(y**3))
def function_2(x, y):
    return np.sin(x**2) + (x * np.cos(y**3))
# Function 3: 3**(2*x) + 5**(4*y) + 2*x + y**4
def function_3(x, y):
    return 3**(2*x) + 5**(4*y) + 2*x + y**4
# Evaluar las funciones en la malla
function1 = function_1(X_1, X_2)
function2 = function_2(X_1, X_2)
function3 = function_3(X_1, X_2)
#Calculo de los vectores gradientes de cada funcion con sus
↪  derivadas
def function_1_df_dx(x, y):
    df_dx = 3 * x**2 * y**2
    df_dy = 2 * x**3 * y
    return df_dx, df_dy
```

```python
def function_2_df_dx(x, y):
    df_dx = 2 * x * np.cos(x**2) + np.cos(y**3)
    df_dy = (-3*x*(y**2))*np.sin(y**3)
    return df_dx, df_dy
def function_3_df_dx(x,y):
    df_dx = 2*(3**(2*x))*math.log(3,np.e)+2
    df_dy = 4*(5**(2*y))*math.log(5,np.e)+4*(y**3)
    return df_dx, df_dy

#Funcion para calcular gradiente dependiendo del numero de funcion ingresado
def calculate_gradient(func_number, x, y):
    switcher = {
        1: function_1_df_dx,
        2: function_2_df_dx,
        3: function_3_df_dx
    }
    # Obtiene la función correspondiente
    func = switcher.get(func_number)
    if func is None:
        return None
    # Calcula y devuelve el gradiente
    return func(x, y)
#Funcion para seleccionar la funcion a evaluar
def select_function(func_number, x, y):
    switcher = {
        1: function_1,
        2: function_2,
        3: function_3
    }
    # Obtiene la función correspondiente
    func = switcher.get(func_number)
    if func is None:
        return "Error: Función no válida"
    # Ejecuta la función seleccionada con los parámetros x e y
    return func(x, y)
#Calcular Magnitud del vector gradiente
def magnitude(grad_x, grad_y):
    return np.sqrt(grad_x**2 + grad_y**2)
# Calcular el vector unitario
def unit_vector(grad_x, grad_y):
    mag = magnitude(grad_x, grad_y)
    if mag == 0:
        return (0, 0)
    return (grad_x / mag, grad_y / mag)
#Funcion para desplegar las superficies de las funciones y los vectores unitario de
#de los vectores gradientes
def plot_function(p_x0,p_y0, p_x1,p_y1, num_funct):
```

```python
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    contour = ax.plot_surface(X_1.numpy(), X_2.numpy(), select_function(num_funct,X_1,X_2).numpy(), cmap='viridis')
    # Calcular el gradiente en P0 y P1
    P0_x, P0_y = calculate_gradient(num_funct, p_x0, p_y0)
    P1_x, P1_y = calculate_gradient(num_funct, p_x1, p_y1)
    # Calcular el vector unitario en P0 y P1
    unit_grad_P0 = unit_vector(P0_x, P0_y)
    unit_grad_P1 = unit_vector(P1_x, P1_y)
    # Calcular la magnitud del vector gradiente en P0 y P1
    mag_P0 = magnitude(P0_x, P0_y)
    mag_P1 = magnitude(P1_x, P1_y)
    # Graficar los vectores unitarios en el plano
    ax.quiver3D(p_x0, p_y0, select_function(num_funct,p_x0,p_y0), unit_grad_P0[0], unit_grad_P0[1], 0, color='red', length=1, normalize=True,zorder=5)
    ax.quiver3D(p_x1, p_y1, select_function(num_funct,p_x1,p_y1), unit_grad_P1[0], unit_grad_P1[1], 0, color='blue', length=1, normalize=True,zorder=5)
     # Etiquetas
    ax.set_title('Curvas de nivel de f(x, y) y vector unitario del gradiente en P0 y P1')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    # Mostrar graficas
    plt.colorbar(contour)
    plt.legend()
    plt.show()
    #Imprimir las magnitudes de los vectores
    print(f"Magnitud del gradiente en P0: {mag_P0}")
    print(f"Magnitud del gradiente en P1: {mag_P1}")
"""
En esta parte las funciones se invocan para graficar las funciones con sus respectivos
vectores unitarios calculados en los vecteres gradientes. Los primeros 4 parametros
son para el punto P0 y el punto P1 respectivamente como puntos cartesianos(x,y) y el
quinto parametro es la funcion a graficar entre los 3 problemas respectivamente.
"""
plot_function(0,0,7.4,-6.3,1)
plot_function(1.5,-5.5,-10,-10,2)
plot_function(-4,-2,-2,9,3)
```

Código para despliegue en 2D

```python
    import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import math
import torch
# Crear una malla de puntos entre -10 y 10
x = np.linspace(-10, 10, 400)
y = np.linspace(-10, 10, 400)
X_1, X_2 = np.meshgrid(x, y)

# Definir las funciones
# Function 1: (x**3) * (y**2) + 1
def function_1(x, y):
    return (x**3) * (y**2) + 1
# Function 2: np.sin(x**2) + (x * np.cos(y**3))
def function_2(x, y):
    return np.sin(x**2) + (x * np.cos(y**3))
# Function 3: 3**(2*x) + 5**(4*y) + 2*x + y**4
def function_3(x, y):
    return 3**(2*x) + 5**(4*y) + 2*x + y**4
# Evaluar las funciones en la malla
function1 = function_1(X_1, X_2)
function2 = function_2(X_1, X_2)
function3 = function_3(X_1, X_2)
#Calculo de los vectores gradientes de cada funcion
def function_1_df_dx(x, y):
    df_dx = 3 * x**2 * y**2
    df_dy = 2 * x**3 * y
    return df_dx, df_dy
def function_2_df_dx(x, y):
    df_dx = 2 * x * np.cos(x**2) + np.cos(y**3)
    df_dy = (-3*x*(y**2))*np.sin(y**3)
    return df_dx, df_dy
def function_3_df_dx(x,y):
    df_dx = 2*(3**(2*x))*math.log(3,np.e)+2
    df_dy = 4*(5**(2*y))*math.log(5,np.e)+4*(y**3)
    return df_dx, df_dy
#Funcion para calcular gradiente dependiendo del numero de funcion
↪ ingresado
def calculate_gradient(func_number, x, y):
    switcher = {
        1: function_1_df_dx,
        2: function_2_df_dx,
        3: function_3_df_dx
    }
    # Obtiene la función correspondiente
    func = switcher.get(func_number)
```

```python
    if func is None:
        return None
    # Calcula y devuelve el gradiente
    return func(x, y)
#Funcion para seleccionar la funcion a evaluar
def select_function(func_number, x, y):
    switcher = {
        1: function_1,
        2: function_2,
        3: function_3
    }
    # Obtiene la función correspondiente
    func = switcher.get(func_number)

    if func is None:
        return "Error: Función no válida"
    # Ejecuta la función seleccionada con los parámetros x e y
    return func(x, y)
#Calcular Magnitud del vector gradiente
def magnitude(grad_x, grad_y):
    return np.sqrt(grad_x**2 + grad_y**2)

# Calcular el vector unitario
def unit_vector(grad_x, grad_y):
    mag = magnitude(grad_x, grad_y)
    if mag == 0:
        return (0, 0)
    return (grad_x / mag, grad_y / mag)
#Funcion para desplegar las superficies de las funciones y los vectores unitario de
#de los vectores gradientes
def plot_function(p_x0,p_y0, p_x1,p_y1, num_funct):
  fig, ax = plt.subplots(figsize=(10, 8))
  contour = ax.contour(X_1, X_2, select_function(num_funct,X_1,X_2),levels=50, cmap='viridis')
  # Calcular el gradiente en P0 y P1
  P0_x, P0_y = calculate_gradient(num_funct, p_x0, p_y0)
  P1_x, P1_y = calculate_gradient(num_funct, p_x1, p_y1)
  # Calcular el vector unitario en P0 y P1
  unit_grad_P0 = unit_vector(P0_x, P0_y)
  unit_grad_P1 = unit_vector(P1_x, P1_y)
  # Calcular la magnitud del vector gradiente en P0 y P1
  mag_P0 = magnitude(P0_x, P0_y)
  mag_P1 = magnitude(P1_x, P1_y)
  # Marcar los puntos P0 y P1 en la gráfica
  #ax.scatter([P0_x, P1_x], [P0_y, P1_y], color='black', label='Puntos evaluados')
  # Graficar los vectores unitarios en el plano
```

```python
    ax.quiver(p_x0, p_y0, unit_grad_P0[0], unit_grad_P0[1], 0, color='red', angles='xy', scale_units='xy', scale=1,zorder=5)
    ax.quiver(p_x1, p_y1, unit_grad_P1[0], unit_grad_P1[1], 0, color='red', angles='xy', scale_units='xy', scale=1,zorder=5)
     # Etiquetas
    ax.set_title('Curvas de nivel de f(x, y) y vector unitario del gradiente en P0 y P1')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    # Mostrar graficas
    plt.colorbar(contour)
    plt.legend()
    plt.show()
    print(f"Magnitud del gradiente en P0: {mag_P0}")
    print(f"Magnitud del gradiente en P1: {mag_P1}")
"""
En esta parte las funciones se invocan para graficar las funciones con sus respectivos
vectores unitarios calculados en los vecteres gradientes. Los primeros 4 parametros
son para el punto P0 y el punto P1 respectivamente como puntos cartesianos(x,y) y el
quinto parametro es la funcion a graficar entre los 3 problemas respectivamente.
"""
plot_function(0,0,7.4,-6.3,1)
plot_function(1.5,-5.5,-10,-10,2)
plot_function(-4,-2,-2,9,3)
```

2.3. 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 $d(\vec{x}_i - \vec{x}_j)$.

2.3.1. 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 argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np;
import pandas as pandas;
from scipy import ndimage
```

```python
from torchvision import datasets, transforms
from torch.distributions import normal
from torch.distributions import multivariate_normal
import matplotlib.pyplot as plt
import time
import sklearn.model_selection as model_selection
from sklearn.model_selection import train_test_split

""" 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



"""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).
```

```python
    :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.

2.3.2. 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
    """
```

```python
    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
```

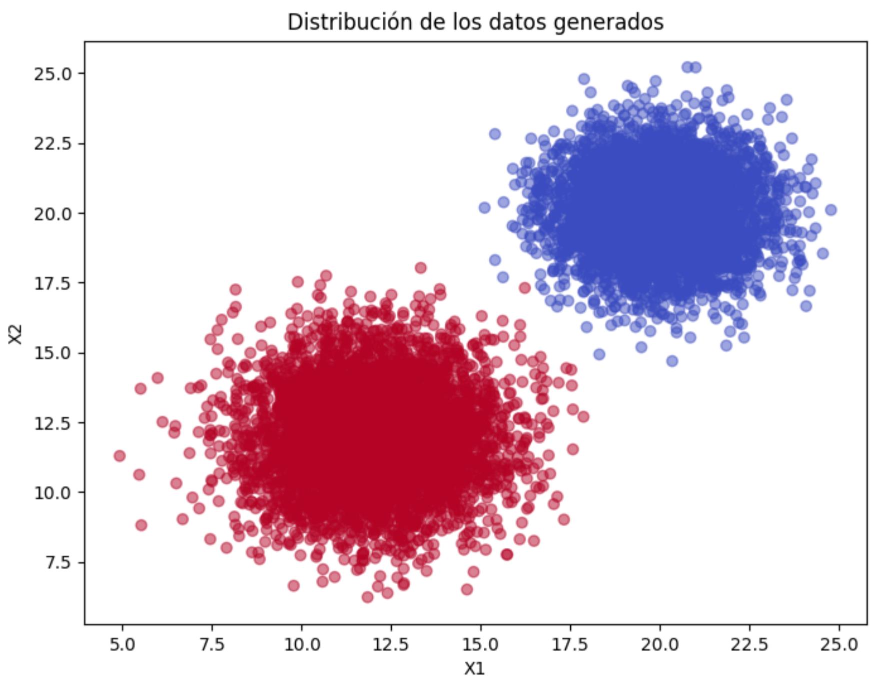
A continuación calculamos la distancia entre una observación de prueba y todas las muestras en data_training usando norma Lp.

```python
        """ 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 2.11: 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}.

2.3.3. 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.

```python
    """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
```

```python
    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)**.

2.3.4. 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.

```python
    """ 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
```

```python
    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.

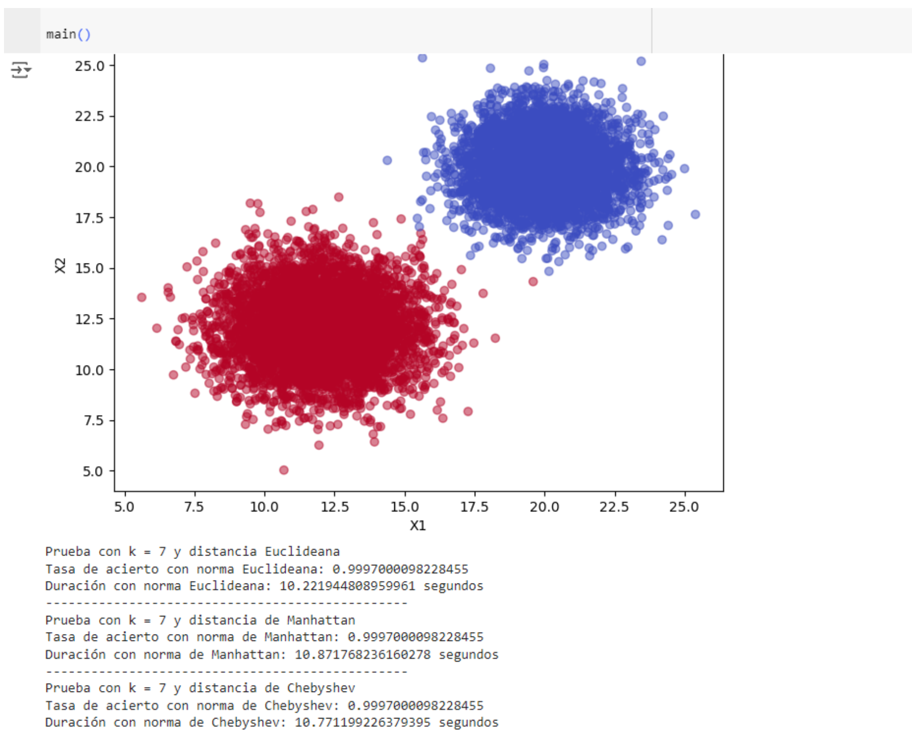


Figura 2.12: Resultado de ejecución con la distancias, euclidiana, manhattan y chebyshev

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.

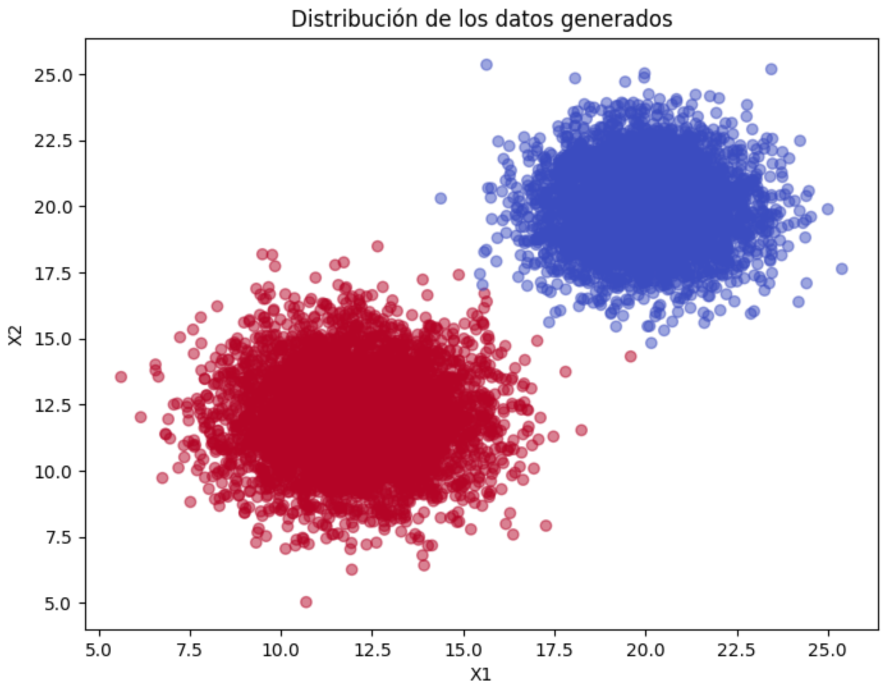


Figura 2.13: Gráfico de los datos generados

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.

2.3.5. 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:

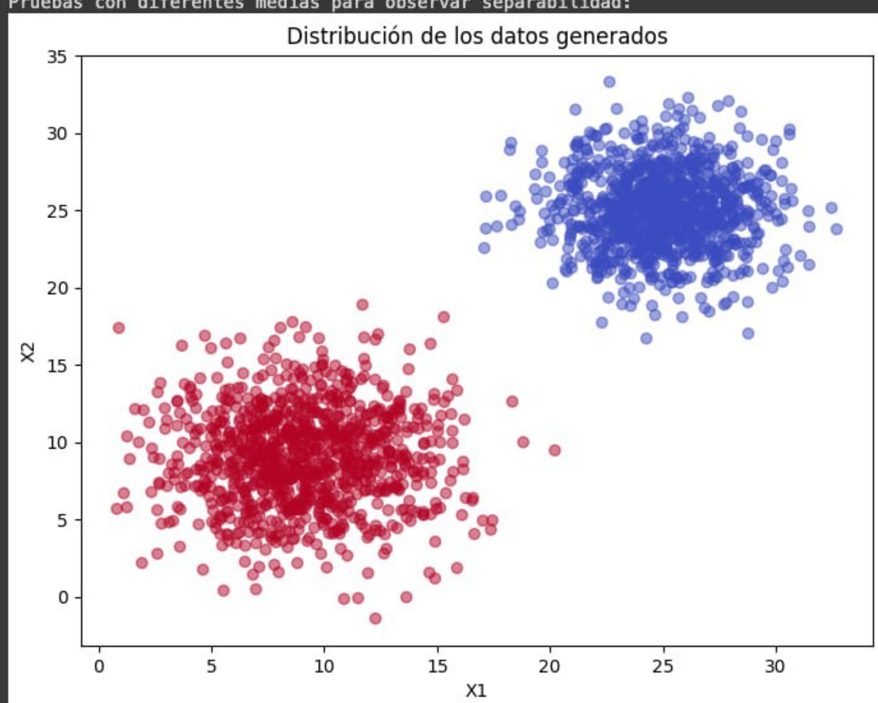
- **Primera Prueba:**

```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=[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 2.14: Prueba 1

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

```
Pruebas con diferentes medias para observar separabilidad:
```




```
Pruebas con diferentes medias y distancia euclidiana
Tasa de acierto Euclideana con diferentes medias: 1.0
Duración con norma Euclideana: 0.35398316383361816 segundos
/n/n/n ---------------------------------------------
Evaluando una observación individual:
La categoría predicha para el punto [[12.5, 14.5]] es: 1.0
```

Figura 2.15: Resultado 1

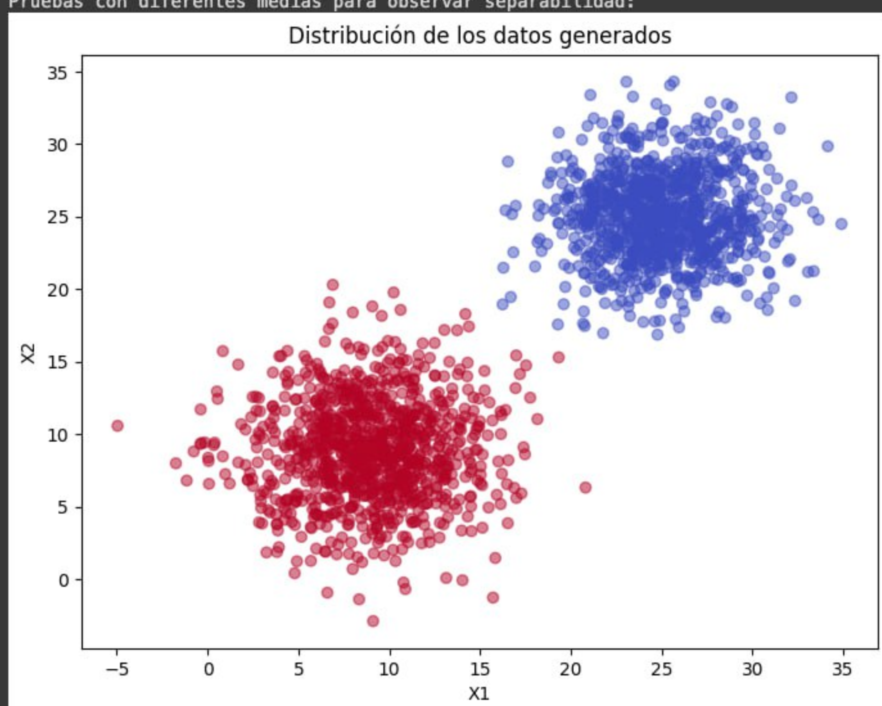
- **Segunda Prueba:**

```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=[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 2.16: Prueba 2

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

```
Pruebas con diferentes medias para observar separabilidad:
```




```
Pruebas con diferentes medias y distancia euclidiana
Tasa de acierto Euclideana con diferentes medias: 1.0
Duración con norma Euclideana: 0.3712003231048584 segundos
/n/n/n ---------------------------------------------
Evaluando una observación individual:
La categoría predicha para el punto [[12.5, 14.5]] es: 1.0
```

Figura 2.17: Resultado 2

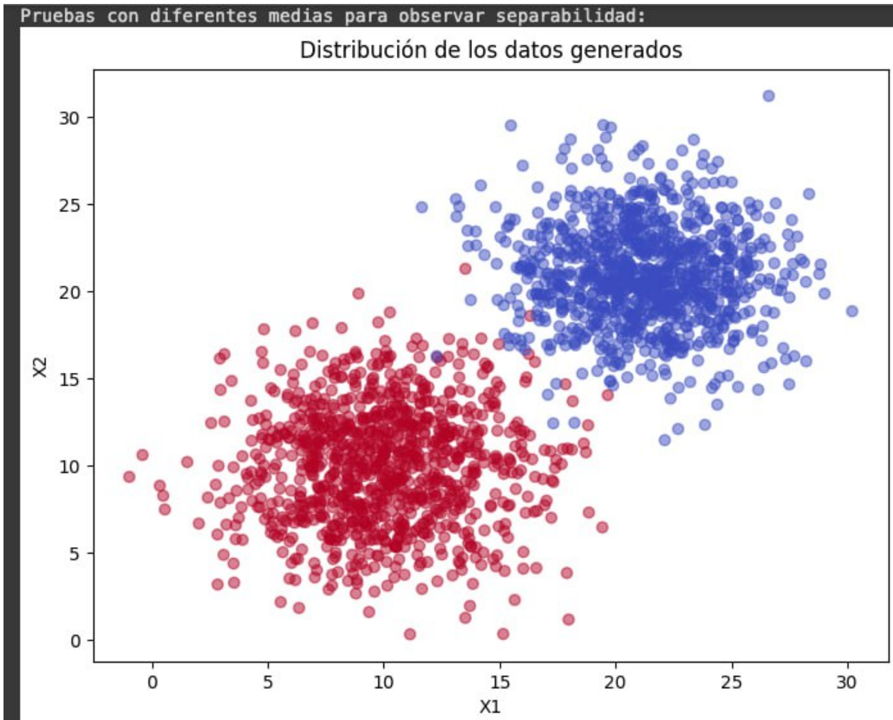
- **Tercera Prueba:**

```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 2.18: Prueba 3

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

```
Pruebas con diferentes medias para observar separabilidad:
```




```
Pruebas con diferentes medias y distancia euclidiana
Tasa de acierto Euclideana con diferentes medias: 0.9934999942779541
Duración con norma Euclideana: 0.368915319442749 segundos
/n/n/n ---------------------------------------------
Evaluando una observación individual:
La categoría predicha para el punto [[12.5, 14.5]] es: 1.0
```

Figura 2.19: Resultado 3

- **Cuarta 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=[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 2.20: Prueba 4

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

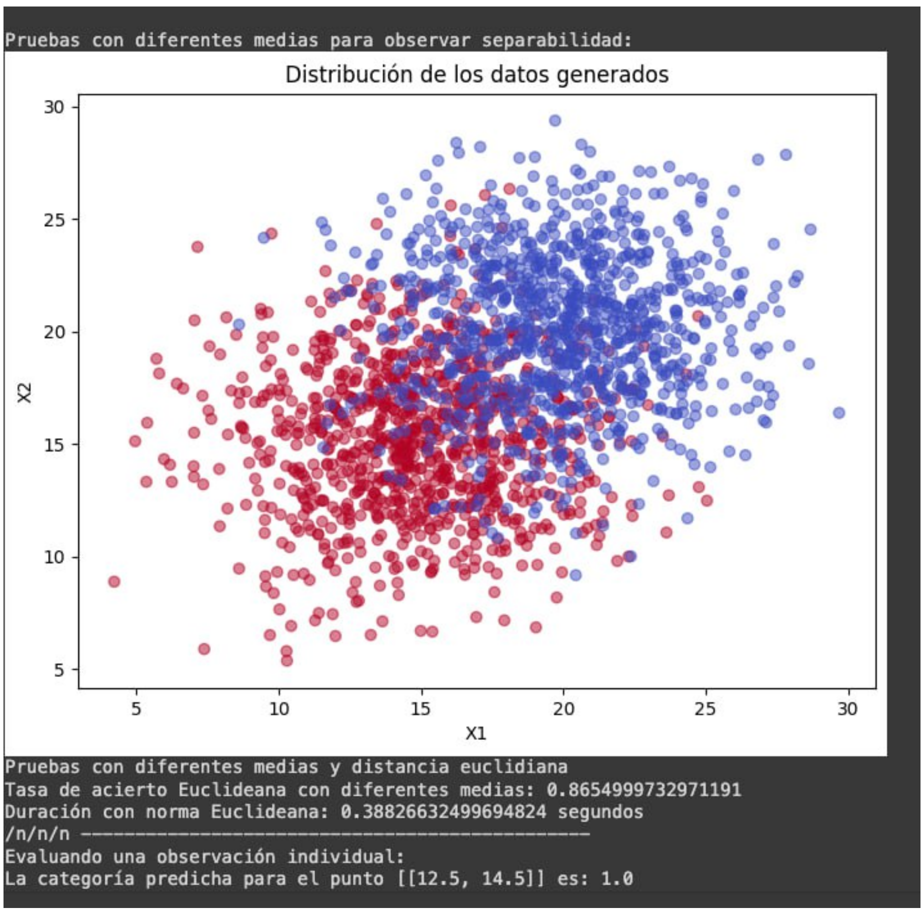


Figura 2.21: Resultado 4

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 abarcó dos áreas principales: funciones multivariables y la implementación del algoritmo K-vecinos más cercanos (KNN). En la primera parte, se estudiaron funciones lineales y no lineales, calculando gradientes, magnitudes y matrices Hessianas en diversos puntos. El uso de herramientas como PyTorch facilitó la visualización de las superficies y curvas de nivel, permitiendo una comprensión más clara de cómo los vectores normales y las curvas de nivel interactúan en el espacio tridimensional. El análisis de funciones multivariables mostró la importancia de los gradientes para identificar la dirección de máximo crecimiento, y la matriz Hessiana reveló la curvatura de las superficies en puntos clave.

En la segunda parte, se implementó el algoritmo KNN en su versión matricial, optimizando el cálculo de distancias y la clasificación de puntos usando Pytorch. Los resultados demostraron que KNN alcanza altas tasas de acierto en datos bien separados, pero su rendimiento disminuye en conjuntos de datos con menor separabilidad. Además, la implementación sin ciclos for y el uso de funciones nativas de Pytorch como ‘cdist‘ y ‘mode‘ mejoraron la eficiencia del algoritmo, especialmente en cuanto a tiempo de ejecución.

En resumen, el análisis de funciones multivariables proporcionó una sólida base geométrica, mientras que la implementación de KNN demostró la importancia de optimizar algoritmos en escenarios de clasificación. Ambos enfoques resaltaron la relevancia de la optimización matemática y computacional en la resolución de problemas complejos.