Regresión Lineal

Walter Andrés Fontecha

Lilibeth Ramirez Esquivel

Fundación Universitaria uninpahu

Ingeniería de datos

Bogotá, 2025

Introducción

En la actualidad, el análisis de datos se ha vuelto fundamental en diversos campos como la economía, la ciencia, la salud y la tecnología. Por eso, cada vez es más importante usar métodos que nos ayuden a entender la información y predecir lo que podría pasar. Uno de los métodos más comunes con este propósito es la regresión, una técnica estadística que permite identificar relaciones entre variables y construir modelos para hacer predicciones.

La regresión lineal es un método estadístico utilizado para el análisis predictivo. Es decir, es una herramienta que usamos para entender cómo cambia una cosa en función de otra. Es muy usada para hacer predicciones, ya que permite identificar patrones y tendencias en la información. Esto ayuda a anticipar lo que podría pasar en el futuro o a completar datos que faltan sin tener que hacerlo manualmente.

Una de las cosas más interesantes de la regresión lineal es que busca una conexión directa entre lo que queremos predecir (la variable dependiente) y los factores que creemos que afectan a eso (las variables independientes). Si usamos solo una variable para hacer la predicción, la relación entre las dos cosas se puede representar con una línea recta, y eso es lo que se llama "regresión lineal".
Esta relación se puede escribir con una fórmula matemática que también se puede dibujar en una gráfica.
La forma más simple de esta fórmula es:
Y = mX + b
Donde:

- **Y** es lo que queremos saber o predecir.
- **X** es el dato que ya tenemos.
- **m** muestra cuánto cambia Y cuando X cambia.
- **b** es el punto donde la línea cruza el eje vertical.

Dependiendo de cómo sea la relación entre las variables, tenemos tres formas de usar la regresión:
Regresión lineal simple: Este es el modelo más básico, donde solo se usa una variable para predecir algo.
Regresión lineal múltiple: Este modelo es como el anterior, pero con más de una variable que puede influir en lo que estamos tratando de predecir.
Regresión no lineal: Este modelo es un poco más complicado porque no sigue una línea recta, sino que tiene una forma más curvada.
Este trabajo tiene como objetivo principal abordar en detalle y con mayor profundidad los diferentes tipos de regresión analizando sus principales características, usos, beneficios y posibles desventajas. Se incluirán ejemplos prácticos para ilustrar cada modelo y se explicarán los escenarios en los que resulta más conveniente aplicarlos.

OBJETIVOS

1. Comprender las relaciones entre las variables
2. Identificar relaciones lineales en grandes volúmenes
3. Evaluar el impacto en cada uno de los problemas propuestos
4. 4. Utilizar regresión lineal para predecir los resultados finales

1. Regresión Lineal

La regresión lineal es una pieza fundamental del análisis de datos permitiendo la integridad entre las variables. Siendo este modelo eficiente para interpretar datos y procesarlos en poco tiempo.

En Big Data, la regresión lineal los modelos de regresión lineal son relativamente simples y proporcionan una fórmula matemática fácil de interpretar para generar predicciones. La regresión lineal es una técnica estadística establecida y se aplica fácilmente al software y a la computación. Las empresas lo utilizan para convertir datos sin procesar de manera confiable y predecible en inteligencia empresarial y conocimiento práctico. Los científicos de muchos campos, incluidas la biología y las ciencias del comportamiento, ambientales y sociales, utilizan la regresión lineal para realizar análisis de datos preliminares y predecir tendencias futuras.

A continuación se presenta un ejemplo de regresión lineal con una sola variable

1.2 Descripción del problema

En esta parte del ejercicio, se implementará regresión lineal con una variable para predecir la nota obtenida de un estudiante según el tiempo de estudio invertido en Bogotá.

El archivo ex1data1.txt contiene el conjunto de datos para nuestro problema de regresión lineal.

	Horas_estudio	Notal_final
0	14	4.5
1	2	1.9
2	8	3.8
3	19	5.0
4	7	3.0

La primera columna representa las horas de estudio de un estudiante en Bogotá y la segunda columna representa la nota final obtenida en la ciudad de Bogotá, esta descripción se realizará a través de las diferentes codificaciones

1.3 Trazar los datos o códigos

Importación de las principales librerías usadas en análisis, visualización y manipulación de datos

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

cargar y visualizar las primeras filas de un archivo CSV (valores separados por comas) que contiene datos sobre horas de estudio y notas finales. Vamos a ver qué hace línea por línea

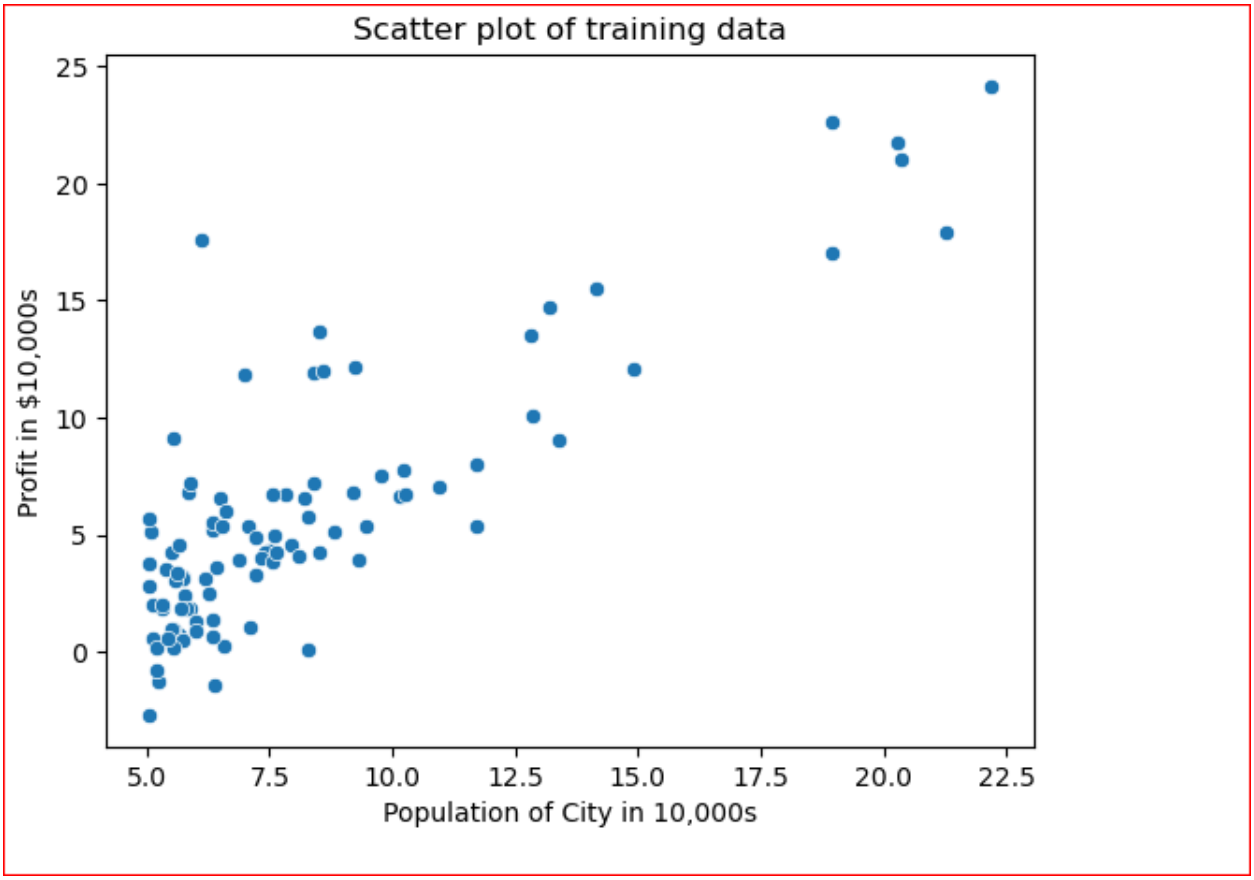
```
url = 'https://raw.githubusercontent.com/damasoer/MACHINE-LEARNING-I/main/data/ex1data1.txt'
df = pd.read_csv(url, sep=",", header=None)
df.columns = ['population', 'profit']
df.head()
```

1.3.1 Resultado

	population	profit
0	6.1101	17.5920
1	5.5277	9.1302
2	8.5186	13.6620
3	7.0032	11.8540
4	5.8598	6.8233

1.3.2 : Gráfico de dispersión (scatter plot) para visualizar la relación entre las horas de estudio y la nota final usando la biblioteca Seaborn.

```
ax = sns.scatterplot(x='population', y='profit', data=df)
ax.set(xlabel='Population of City in 10,000s', ylabel='Profit in $10,000s', title='Scatter plot of training data');
# Mostrar el gráfico
plt.show()
```



La trama muestra que tienen una relación lineal.

1.4 Descenso por gradientes:

Ajustar los parámetros de la regresión linéa a los datos utilizando el descenso por gradientes

1.4.1 Ecuaciones

En la regresión lineal, las hipótesis principales son sobre la relación entre las variables dependiente e independiente. Se busca establecer si existe una relación lineal significativa y, en caso afirmativo, cómo de fuerte es esa relación. Se formulan dos hipótesis principales: la hipótesis nula (H0) y la hipótesis alternativa (H1).

1.4.2 Implementación

En programación, se crea el código para el descenso del gradiente

```
# Número de muestras (filas) en el DataFrame
m = df.shape[0]  # Construcción de la matriz de características X:

# Se agrega una columna de unos (para el término independiente) y se concatena con la variable 'Horas_estudio'
X = np.hstack((np.ones((m,1)), df.Horas_estudio.values.reshape(-1,1)))

# Vector de salida/etiqueta y: se convierte la columna 'Notal_final' en un vector columna
y = np.array(df.Notal_final.values).reshape(-1,1)

# Inicialización del vector de parámetros theta con ceros (dimensión: número de características + 1)
theta = np.zeros(shape=(X.shape[1],1))
# Número de iteraciones para el algoritmo de descenso por gradiente
iterations = 1500

# Tasa de aprendizaje (learning rate)
alpha = 0.01

###$$$$$ ### POr favor imprimir cada cambio
```

1.4.3 Calculando la Función de Costo:

Esa función compute_cost_one_variable calcula el costo (o error promedio cuadrático) de un modelo de regresión lineal con una sola variable (aunque puede generalizarse a varias).

```
def compute_cost_one_variable(X, y, theta):
    m = y.shape[0]
    h = X.dot(theta)
    J = (1/(2*m)) * (np.sum((h - y)**2))
    return J
```

```
J = compute_cost_one_variable(X, y, theta)
print('With theta = [0 ; 0]\nCost computed =', J)
print('Expected cost value (approx) 32.07')
#- La función devuelve el valor del costo ( J ), que indica cuán bien o mal está funcionando
# nuestro modelo con los valores actuales de theta.
#El costo ( J ) indica qué tan bien ajusta el modelo a los datos. Para saber si es alto o bajo, considera estos puntos:
# - Costo muy alto (( J > 10 )) → El modelo está lejos de predecir correctamente.
# Costo moderado (( 1 < J \leq 10 )) → El modelo empieza a mejorar, pero aún tiene errores.
# Costo bajo (( J \approx 0 )) → La predicción es muy precisa.
```

```
With theta = [0 ; 0]
Cost computed = 7.1616
Expected cost value (approx) 32.07
```

```
J = compute_cost_one_variable(X, y, [[-1],[2]])
print('With theta = [-1 ; 2]\nCost computed =', J)
print('Expected cost value (approx) 54.24')
#- La función devuelve el valor del costo ( J ), que indica cuán bien o mal está funcionando
# nuestro modelo con los valores actuales de theta.
#El costo ( J ) indica qué tan bien ajusta el modelo a los datos. Para saber si es alto o bajo, considera estos puntos:
# - Costo muy alto (( J > 10 )) → El modelo está lejos de predecir correctamente.
# Costo moderado (( 1 < J \leq 10 )) → El modelo empieza a mejorar, pero aún tiene errores.
# Costo bajo (( J \approx 0 )) → La predicción es muy precisa.
```

```
With theta = [-1 ; 2]
Cost computed = 189.01760000000002
Expected cost value (approx) 54.24
```

1.4.4: Gradiente Descendiente

es un algoritmo de optimización genérico que mide el gradiente local de la función de costo con respecto al parámetro y avanza en la dirección del gradiente descendente.

Algoritmo:

- Tasa de aprendizaje demasiado pequeña: descenso de gradiente lento
- Tasa de aprendizaje demasiado grande: el descenso del gradiente puede sobrepasar el mínimo y puede no converger

```
def gradient_descent(X, y, theta, alpha, num_iters):
    m = y.shape[0]
    J_history = np.zeros(shape=(num_iters, 1))

#m: Obtiene la cantidad de datos en y.
#J_history: Un array para guardar el costo ( J ) en cada iteración, lo que nos permite visualizar cómo disminuye con el


    for i in range(0, num_iters):
        h = X.dot(theta) #Calcula las predicciones del modelo utilizando ( h = X \cdot \theta ).
        diff_hy = h - y # diff_hy almacena la diferencia entre la predicción h y los valores reales y.



        delta = (1/m) * (diff_hy.T.dot(X))
        theta = theta - (alpha * delta.T)
        J_history[i] = compute_cost_one_variable(X, y, theta)
# delta calcula el gradiente (la dirección en que se debe ajustar theta).
# theta = theta - (alpha * delta.T): Actualiza los parámetros theta en cada iteración, moviéndolos en la dirección del
#j history Calcula y almacena el costo ( J ) en cada iteración, para ver cómo disminuye.

    return theta, J_history
# Devuelve theta (los valores óptimos para la regresión) y J_history (la evolución del costo).
```

#ejecuta el algoritmo de descenso por gradiente para encontrar los valores óptimos de theta en una regresión lineal.

theta, _ = gradient_descent(X, y, theta, alpha, iterations)
print('Theta found by gradient descent:\n', theta)

 # Muestra los valores finales de theta, los cuales representan la mejor línea de ajuste para los datos.
print('Expected theta values (approx)\n -3.6303\n 1.1664')
 # Presenta los valores aproximados esperados de theta, permitiendo verificar si la optimización fue exitosa

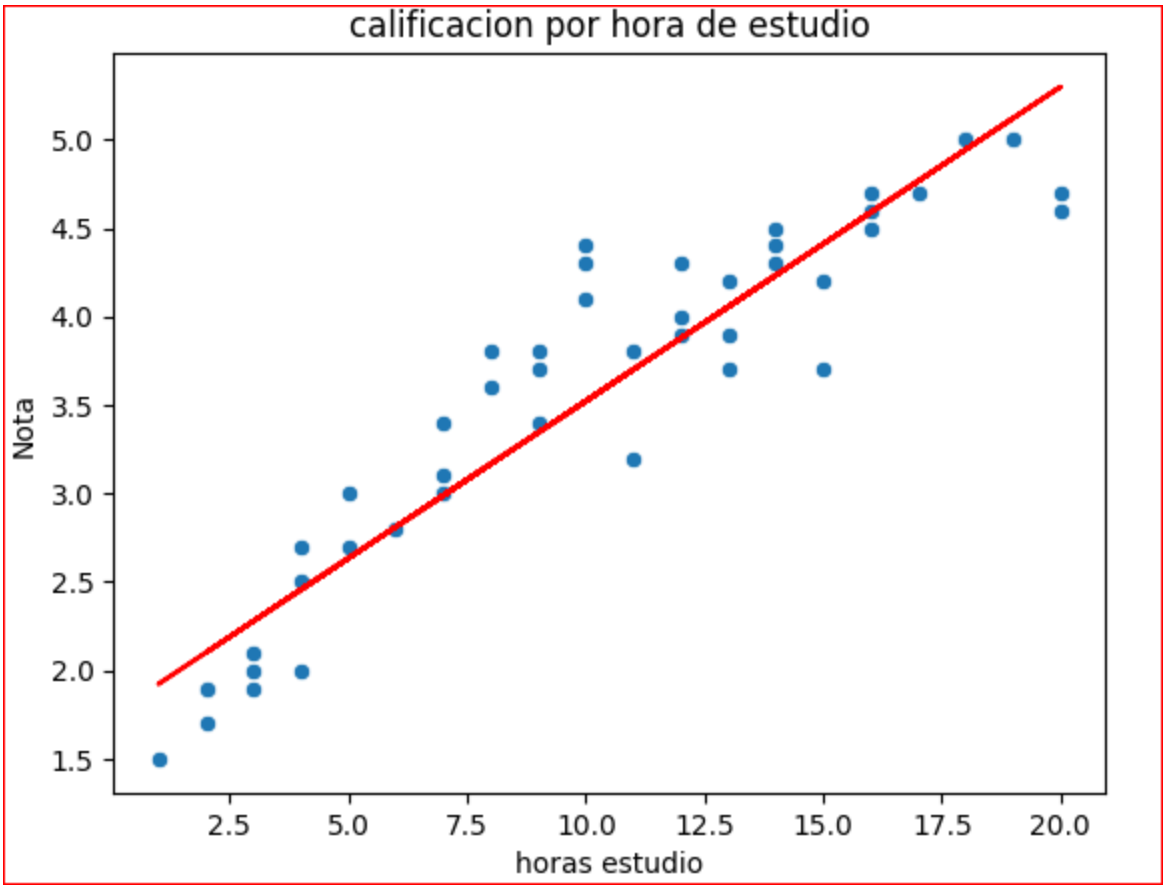
Theta found by gradient descent:
 [[1.74620969]
 [0.1776025]]
Expected theta values (approx)
 -3.6303
  1.1664

 # Resultado de Theta Found
#- Intercepto (theta[0] = 1.7462) → Representa el valor predicho cuando Horas_estudio = 0. Es el punto de inicio de la recta de regresión.
 # Pendiente (theta[1] = 0.1776) → Indica cuánto cambia la Nota_final por cada hora adicional de estudio.
 # En este caso, cada hora de estudio incrementa la nota en aproximadamente 0.1776.

1.4.5 Grafica el ajuste lineal

Visualización de los datos y la línea de regresión lineal

```
ax = sns.scatterplot(x='Horas_estudio', y='Notal_final', data=df)
plt.plot(X[:,1], X.dot(theta), color='r')
ax.set(xlabel='horas estudio', ylabel='Nota', title='calificacion por hora de estudio');
```




```
# Definir  cantidad de horas de estudio para hacer la predicción
horas_estudio_nueva = 8  # se puede cambiar segun se quiera

# Crear el vector de entrada
y_pred = np.array([1, horas_estudio_nueva]).dot(theta)

# Mostrar la predicción
print(f'Para {horas_estudio_nueva} horas de estudio, se predice una nota de {y_pred[0]:.2f}')
```

```
Para 8 horas de estudio, se predice una nota de 3.17
```

```
# Definir  cantidad de horas de estudio para hacer la predicción
horas_estudio_nueva = 10  # se puede cambiar segun se quiera

# Crear el vector de entrada
y_pred = np.array([1, horas_estudio_nueva]).dot(theta)

# Mostrar la predicción
print(f'Para {horas_estudio_nueva} horas de estudio, se predice una nota de {y_pred[0]:.2f}')
```

```
Para 10 horas de estudio, se predice una nota de 3.52
```

1.4.6 Visualizante J(0)

theta0_vals = np.linspace(-10, 10, 100) *# Representa posibles valores para theta0, el término independiente en una regresión lineal.*

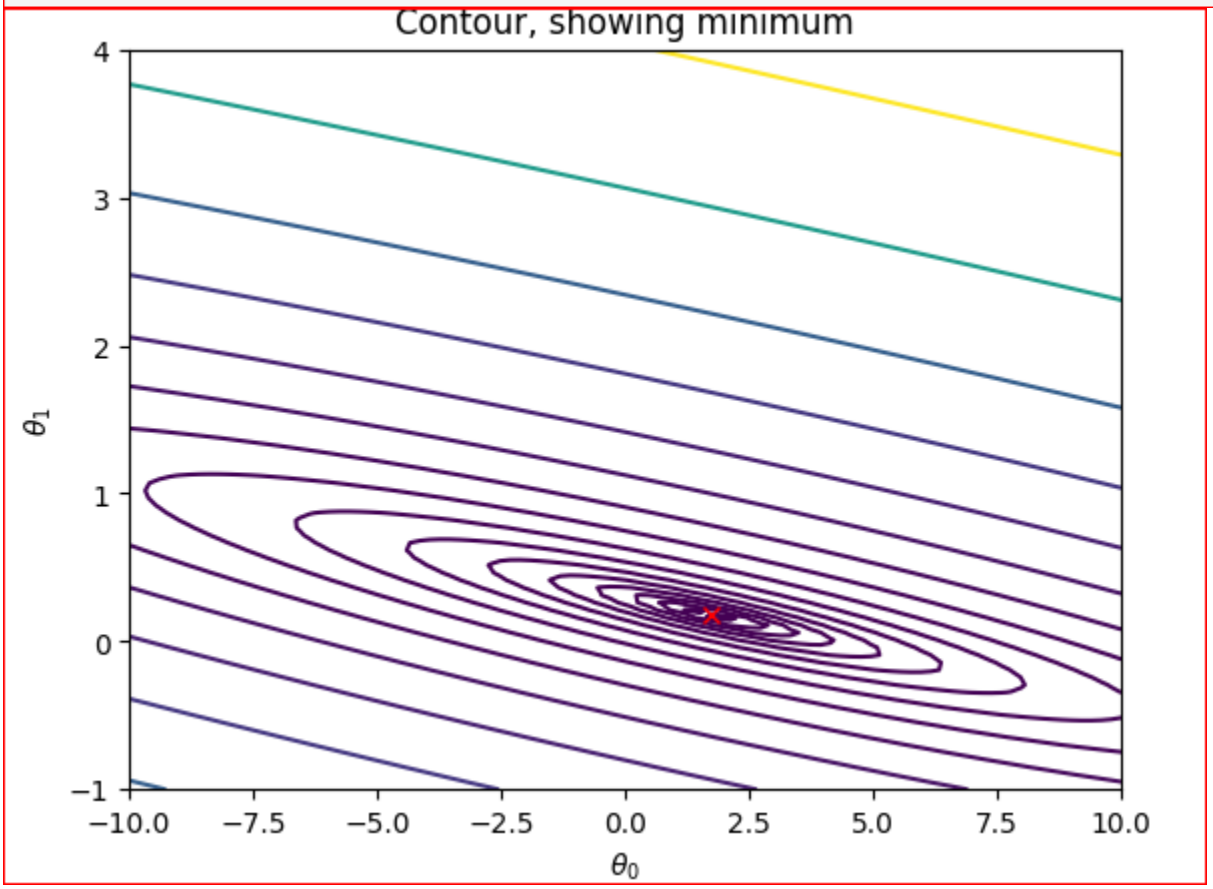
theta1_vals = np.linspace(-1, 4, 100) *# - Representa posibles valores para theta1, el coeficiente de la variable predictora*

```
J_vals = np.zeros(shape=(len(theta0_vals), len(theta1_vals)))
# len(theta0_vals): Número de valores de theta0 (100 en este caso).
# len(theta1_vals): Número de valores de theta1 (100 en este caso).
# Esto significa que la matriz J_vals tendrá 100 filas y 100 columnas
```

for i **in** range(0, len(theta0_vals)): *# Recorre todos los valores de theta0*
  for j **in** range(0, len(theta1_vals)): *# Recorre todos los valores de theta1*
    J_vals[i,j] = compute_cost_one_variable(X, y, [[theta0_vals[i]], [theta1_vals[j]]]) *# Calcula el costo para cada combinación de theta0 y theta1*

#grafica de contorno que muestra cómo varía la función de costo (J(\theta)) en función de los valores de (\theta_0) y (\theta_1)

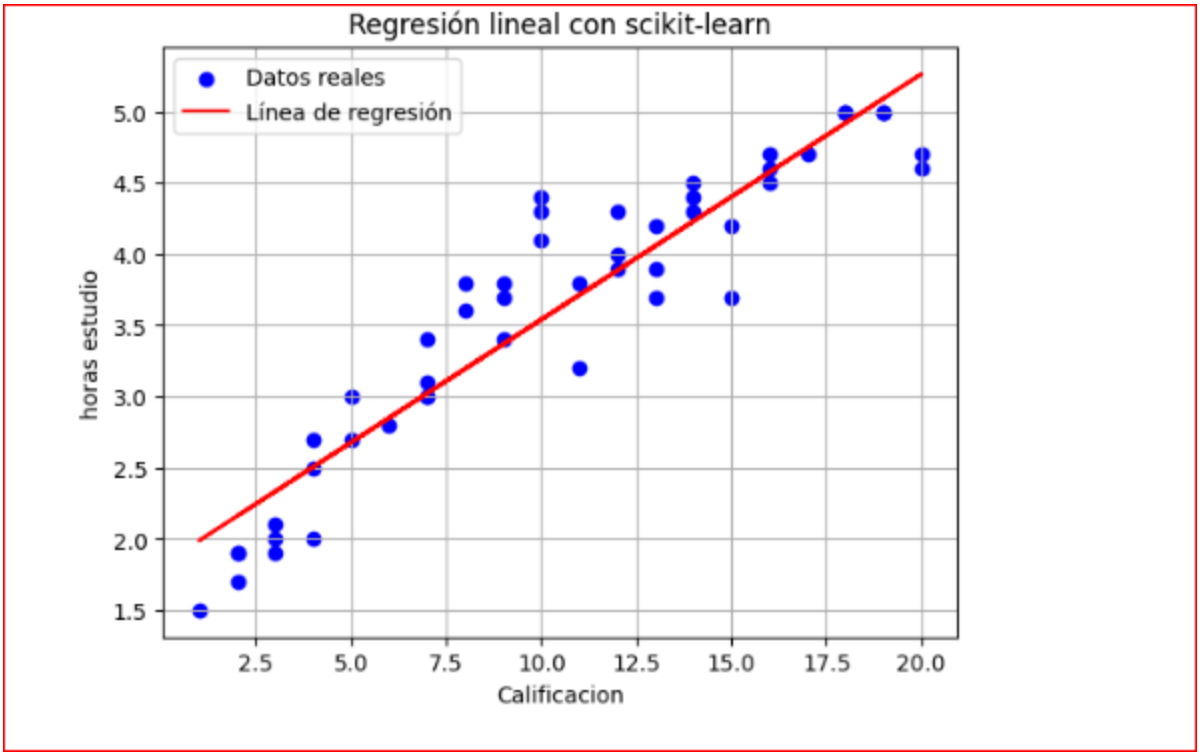
```
ax = plt.contour(theta0_vals, theta1_vals, np.transpose(J_vals), levels=np.logspace(-2,3,20))
plt.plot(theta[0,0], theta[1,0], marker='x', color='r');
plt.xlabel(r'$\theta_0$');
plt.ylabel(r'$\theta_1$');
plt.title('Contour, showing minimum');
plt.show()
```



1.4.7 Usando sklearn

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Cargar los datos
df = pd.read_csv("C:\\Users\\LILI\\Downloads\\horas_estudio_vs_nota_final_bogota_.csv")
df.columns = ['Horas_estudio', 'Notal_final']

# Preparar X e y
X = df[['Horas_estudio']]  # variable independiente (caracteristica)
y = df['Notal_final']        # variable dependiente (objetivo)

# Crear y ajustar el modelo de regresión lineal
model = LinearRegression()
model.fit(X, y)

# Mostrar parámetros del modelo
print(f"Intercepto (theta_0): {model.intercept_}")
print(f"Coeficiente (theta_1): {model.coef_[0]}")

# Predicción de valores ajustados
y_pred = model.predict(X)

# Graficar los datos y la línea de regresión
plt.scatter(X, y, color='blue', label='Datos reales')
plt.plot(X, y_pred, color='red', label='Línea de regresión')
plt.xlabel('Calificacion')
plt.ylabel('horas estudio')
plt.title('Regresión lineal con scikit-learn')
plt.legend()
plt.grid(True)
plt.show()
```

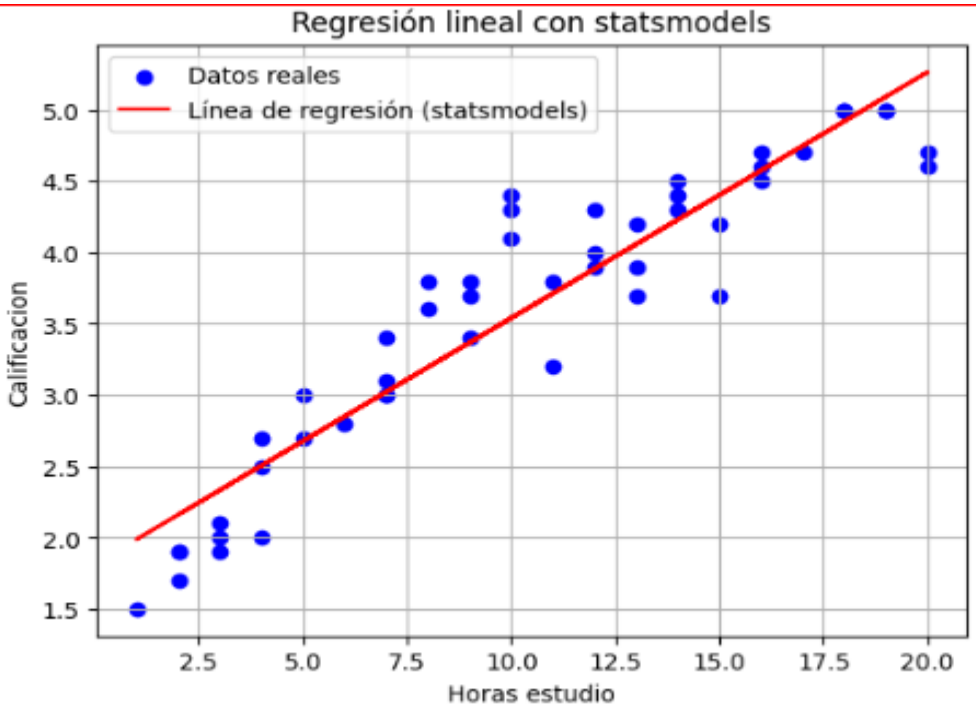
```
Intercepto (theta_0): 1.816761904761904
Coeficiente (theta_1): 0.17238095238095244
```



1.4.8. Usando statsmodels

```
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt

# Cargar los datos
df = pd.read_csv("C:\\Users\\LILI\\Downloads\\horas_estudio_vs_nota_final_bogota_.csv")
df.columns = ['Horas_estudio', 'Notal_final']

# Variable independiente (con constante añadida) y dependiente
X = sm.add_constant(df['Horas_estudio'])  # Agrega una columna de 1s para el intercepto (theta_0)
y = df['Notal_final']

# Ajustar el modelo de regresión lineal
model = sm.OLS(y, X).fit()

# Mostrar resumen del modelo
print(model.summary())

# Predicciones
df['y_pred'] = model.predict(X)

# Graficar los datos y la línea de regresión
plt.scatter(df['Horas_estudio'], df['Notal_final'], color='blue', label='Datos reales')
plt.plot(df['Horas_estudio'], df['y_pred'], color='red', label='Línea de regresión (statsmodels)')
plt.xlabel('Horas estudio')
plt.ylabel('Calificacion')
plt.title('Regresión lineal con statsmodels')
plt.legend()
plt.grid(True)
plt.show()
```

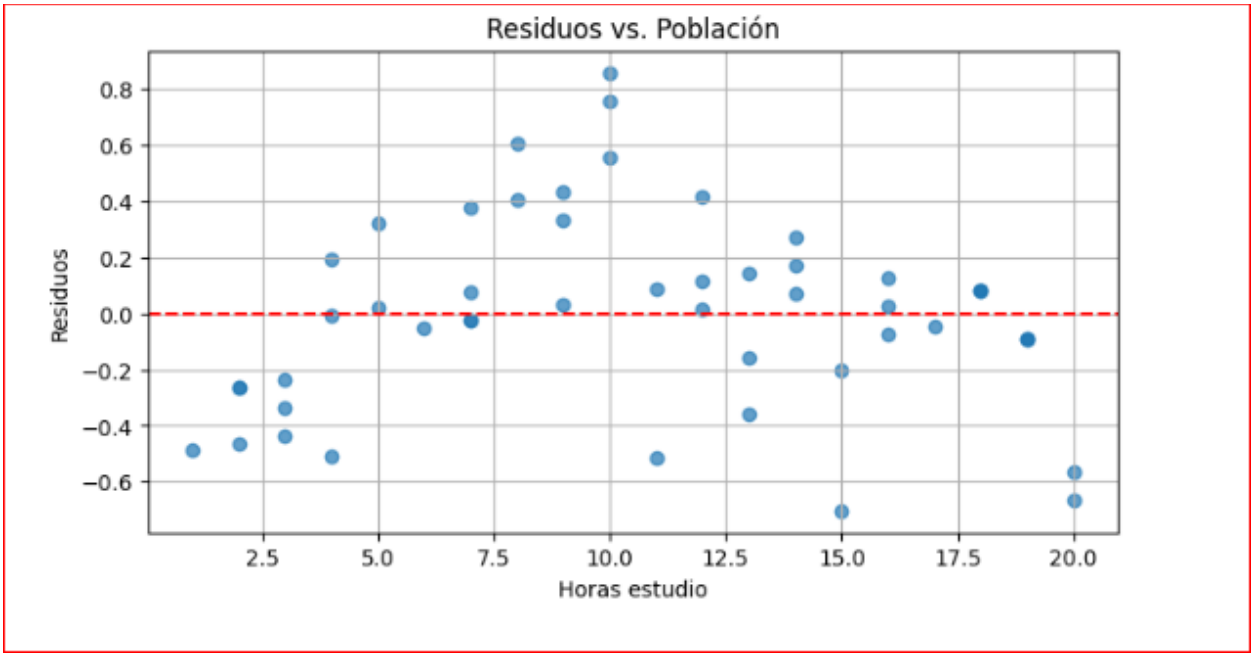
```
                            OLS Regression Results
==============================================================================
Dep. Variable:             Notal_final   R-squared:                       0.884
Model:                             OLS   Adj. R-squared:                  0.882
Method:                  Least Squares   F-statistic:                     366.6
Date:                 Mon, 09 Jun 2025   Prob (F-statistic):           4.08e-24
Time:                         21:53:08   Log-Likelihood:                -18.130
No. Observations:                   50   AIC:                             40.26
Df Residuals:                       48   BIC:                             44.08
Df Model:                            1
Covariance Type:             nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          1.8168      0.108     16.849      0.000       1.600       2.034
Horas_estudio  0.1724      0.009     19.147      0.000       0.154       0.190
==============================================================================
Omnibus:                        0.291   Durbin-Watson:                   2.183
Prob(Omnibus):                  0.865   Jarque-Bera (JB):                0.224
Skew:                           0.153   Prob(JB):                        0.894
Kurtosis:                       2.884   Cond. No.                         25.9
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```




```
## Validacion de supuertos de Modelos
```

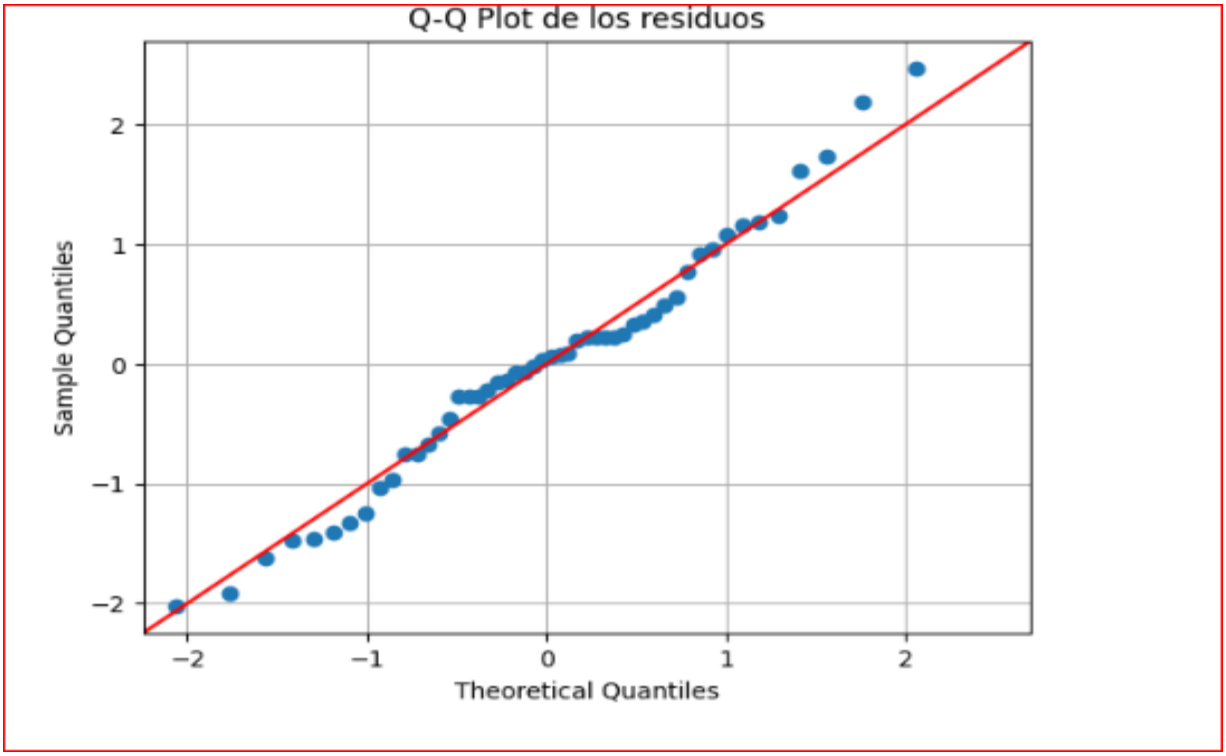
```python
from scipy.stats import shapiro
from statsmodels.stats.diagnostic import het_breuschpagan
# Obtener residuos
residuals = model.resid

# --- 1. Normalidad de los residuos ---
shapiro_test = shapiro(residuals)
print("\nShapiro-Wilk test (normalidad):")
print(f"  Estadístico: {shapiro_test.statistic:.4f}, p-valor: {shapiro_test.pvalue:.4f}")
if shapiro_test.pvalue > 0.05:
    print("  ✔ Los residuos parecen normales (no se rechaza H0).")
else:
    print("  ✖ Los residuos no parecen normales (se rechaza H0).")

# --- 2. Homocedasticidad (varianza constante) ---
bp_test = het_breuschpagan(residuals, X)
bp_labels = ['LM Statistic', 'LM p-value', 'F-statistic', 'F p-value']
print("\nBreusch-Pagan test (homocedasticidad):")
for label, value in zip(bp_labels, bp_test):
    print(f"  {label}: {value:.4f}")
if bp_test[1] > 0.05:
    print("  ✔ No hay evidencia fuerte de heterocedasticidad.")
else:
    print("  ✖ Posible heterocedasticidad (varianza no constante).")

# --- 3. Autocorrelación ---
print(f"\nDurbin-Watson (ya en el resumen): {model.summary().tables[1].data[2][1]}")

# --- (Opcional) Graficar residuos ---
plt.figure(figsize=(8, 4))
plt.scatter(df['Horas_estudio'], residuals, alpha=0.7)
plt.axhline(0, color='red', linestyle='--')
plt.title("Residuos vs. Población")
plt.xlabel("Horas estudio")
plt.ylabel("Residuos")
plt.grid(True)
plt.show()
```

```
Shapiro-Wilk test (normalidad):
  Estadístico: 0.9816, p-valor: 0.6226
  ✔ Los residuos parecen normales (no se rechaza H0).

Breusch-Pagan test (homocedasticidad):
  LM Statistic: 0.1785
  LM p-value: 0.6727
  F-statistic: 0.1720
  F p-value: 0.6802
  ✔ No hay evidencia fuerte de heterocedasticidad.

Durbin-Watson (ya en el resumen):     0.1724
```



1.4.9 Crear el Q-Q plot

```
# Crear el Q-Q plot
sm.qqplot(residuals, line='45', fit=True)

plt.title("Q-Q Plot de los residuos")
plt.grid(True)
plt.show()
```



1.5 Regresión lineal con múltiples variables

En esta parte, implementará una regresión lineal con múltiples variables para predecir los precios de las viviendas. Suponga que está vendiendo su casa y desea saber cuál sería un buen precio de mercado. Una forma de hacerlo es recopilar información sobre las casas vendidas recientemente y crear un modelo de precios de vivienda. El archivo ex1data2.txt contiene un conjunto de entrenamiento de precios de vivienda en Portland, Oregón. La primera columna representa el tamaño de la casa (en pies cuadrados), la segunda el número de habitaciones y la tercera el precio de la vivienda.

1.5.1 Normalización de funciones

```
url = 'https://raw.githubusercontent.com/damasoer/MACHINE-LEARNING-I/main/data/ex1data2.txt'
df2 = pd.read_csv(url, sep=",", header=None)
df2.columns = ['house_size', 'bedrooms', 'house_price']
df2.describe().T
```

	count	mean	std	min	25%	50%	75%	max
house_size	47.0	2000.680851	794.702354	852.0	1432.0	1888.0	2269.0	4478.0
bedrooms	47.0	3.170213	0.760982	1.0	3.0	3.0	4.0	5.0
house_price	47.0	340412.659574	125039.899586	169900.0	249900.0	299900.0	384450.0	699900.0

Al observar los valores, note que los tamaños de las casas son aproximadamente 1000 veces mayores que el número de habitaciones. Cuando las características difieren por órdenes de magnitud, se realiza

escalado d(escalado de características) previamente puede hacer que el descenso por gradiente converja mucho más rápido

Podemos acelerar el descenso por gradiente si hacemos que cada uno de nuestros valores de entrada esté, idealmente, en un rango similar $-1 \leq x_i \leq 1$ or $-0.5 \leq x_i \leq 0.5$.

- Escala de características: implica dividir los valores de entrada por el rango (es decir, el valor máximo menos el valor mínimo) de la variable de entrada.
- Normalización de media: implica restar el valor promedio de una variable de entrada de los valores de esa variable de entrada

$x_i := \frac{x_i - \mu_i}{s_i}$ donde μ_i es el promedio de todos los valores de las características (i) y s_i es el rango de valores (máximo-mín.), la desviación estándar.

```
def feature_normalize(X, mean=np.zeros(1), std=np.zeros(1)):
    X = np.array(X)
    if len(mean.shape) == 1 or len(std.shape) == 1:
        mean = np.mean(X, axis=0)
        std = np.std(X, axis=0, ddof=1)

    X = (X - mean)/std
    return X, mean, std
```

```
X_norm, mu, sigma = feature_normalize(df2[['house_size', 'bedrooms']])
```

```
df2['house_size_normalized'] = X_norm[:,0]
df2['bedrooms_normalized'] = X_norm[:,1]
df2[['house_size_normalized', 'bedrooms_normalized']].describe().T
```

1.5.2 Descenso de gradiente

La única diferencia con el problema de regresión univariada es que ahora hay una característica más en la matriz X. La función de hipótesis y la regla de actualización del descenso por gradiente por lotes permanecen sin cambios.

Nota: En el caso multivariable, la función de costo también puede escribirse en la siguiente forma vectorizada:

$$J(\theta) = \frac{1}{2m}(X\theta - y)^T(X\theta - y)$$

```
def compute_cost(X, y, theta):
    m = y.shape[0]
    h = X.dot(theta)
    J = (1/(2*m)) * ((h-y).T.dot(h-y))
    return J
```

```
def gradient_descent(X, y, theta, alpha, num_iters):
    m = y.shape[0]
    J_history = np.zeros(shape=(num_iters, 1))

    for i in range(0, num_iters):
        h = X.dot(theta)
        diff_hy = h - y

        delta = (1/m) * (diff_hy.T.dot(X))
        theta = theta - (alpha * delta.T)
        J_history[i] = compute_cost(X, y, theta)

    return theta, J_history
```

1.5.3 Seleccionando Tasa de aprendizaje

```
m = df2.shape[0]
X2 = np.hstack((np.ones((m,1)),X_norm))
y2 = np.array(df2.house_price.values).reshape(-1,1)
theta2 = np.zeros(shape=(X2.shape[1],1))
```

```
alpha = [0.3, 0.1, 0.03, 0.01]
colors = ['b','r','g','c']
num_iters = 50
```

```
for i in range(0, len(alpha)):
    theta = np.zeros(shape=(X.shape[1],1))
    theta, J_history = gradient_descent(X2, y2, theta2, alpha[i], num_iters)
    plt.plot(range(len(J_history)), J_history, colors[i], label='Alpha {}'.format(alpha[i]))
plt.xlabel('Number of iterations');
plt.ylabel('Cost J');
plt.title('Selecting learning rates');
plt.legend()
plt.show()
```

```
iterations = 250
alpha = 0.1
theta2, J_history = gradient_descent(X2, y2, theta2, alpha, iterations)

print('Theta found by gradient descent:')
print(theta)
```

Estima el precio de una casa de 1650 pies cuadrados con 3 habitaciones.

```
sqft = (1650 - mu[0])/sigma[0]
bedrooms = (3 - mu[1])/sigma[1]
y_pred = theta2[0] + theta2[1]*sqft + theta2[2]*bedrooms
f'precio de una casa de 1650 pies cuadrados con 3 habitaciones: {y_pred[0]}$'
```

1.5.4 Ecuaciones normales

Una solución de forma cerrada para encontrar sin iteración.

$$\theta = (X^T X)^{-1} X^T y$$

```
def normal_eqn(X, y):
    inv = np.linalg.pinv(X.T.dot(X))
    theta = inv.dot(X.T).dot(y)
    return theta
```

```
Xe = np.hstack((np.ones((m,1)),df2[['house_size', 'bedrooms']].values))
theta_e = normal_eqn(Xe, y2)
theta_e
```

```
y_pred = theta_e[0] + theta_e[1]*1650 + theta_e[2]*3
f'Price of a house with 1650 square feet and 3 bedrooms: {y_pred[0]}$'
```

1.5.5 Código equivalente usando Scikit-Learn

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt


# 2. División de datos
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 3. Normalización (si no tienes X_norm)
scaler = StandardScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)

# 4. Entrenar el modelo
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# 5. Predicciones
y_pred = lin_reg.predict(X_test)

# 6. Evaluación del modelo
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print(f"MAE  (Error Absoluto Medio): {mae:.2f}")
print(f"MSE  (Error Cuadrático Medio): {mse:.2f}")
print(f"RMSE (Raiz del Error Cuadrático Medio): {rmse:.2f}")
print(f"R²   (Coeficiente de determinación): {r2:.2f}")

# 7. Visualización
# 7. Visualización de predicciones vs valores reales
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, edgecolors=(0, 0, 0))
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
plt.xlabel('Valores Reales')
plt.ylabel('Predicciones')
plt.title('Regresión Lineal: Predicciones vs Reales')
plt.grid(True)
plt.show()
```

```
### 2.5 Equivalent Code using statsmodels
```

```python
import numpy as np
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt

# Supongamos que ya tienes X y y definidos como DataFrames o arrays
# X: variables independientes, y: variable dependiente

# 1. División de datos (manual o con numpy)
from sklearn.model_selection import train_test_split  # solo esta parte de sklearn para dividir
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 2. Normalización manual (opcional, si hay variables en diferente escala)
X_train_mean = X_train.mean()
X_train_std = X_train.std()
X_train_norm = (X_train - X_train_mean) / X_train_std
X_test_norm = (X_test - X_train_mean) / X_train_std  # usar media y std del train

# 3. Agregar constante (intercepto)
X_train_sm = sm.add_constant(X_train_norm)
X_test_sm = sm.add_constant(X_test_norm)

# 4. Entrenar modelo con OLS (Ordinary Least Squares)
model = sm.OLS(y_train, X_train_sm)
results = model.fit()

# 5. Imprimir resumen estadístico
print(results.summary())

# 6. Predicciones
y_pred = results.predict(X_test_sm)

# 7. Métricas de evaluación
mae = np.mean(np.abs(y_test - y_pred))
mse = np.mean((y_test - y_pred) ** 2)
rmse = np.sqrt(mse)
ss_total = np.sum((y_test - np.mean(y_test)) ** 2)
ss_res = np.sum((y_test - y_pred) ** 2)
r2 = 1 - (ss_res / ss_total)

print("\nEvaluación del modelo:")
print(f"MAE  (Error Absoluto Medio): {mae:.2f}")
print(f"MSE  (Error Cuadrático Medio): {mse:.2f}")
print(f"RMSE (Raíz del Error Cuadrático Medio): {rmse:.2f}")
print(f"R²   (Coeficiente de determinación): {r2:.2f}")

# 8. Visualización: valores reales vs predichos
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, edgecolors=(0, 0, 0))
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
plt.xlabel('Valores Reales')
plt.ylabel('Predicciones')
plt.title('Predicciones vs Reales (Statsmodels)')
plt.grid(True)
plt.show()
```

2. Programming Exercise 2: Regresión Logística

2.1 Descripción del problema

En esta parte del ejercicio, construirás un modelo de regresión logística para predecir los créditos que se pueden otorgar de acuerdo a los ingresos de cada cliente.

Para determinar la probabilidad de aprobación de un crédito en función de sus ingresos.\n", "Cuentas con datos del ingreso de los clientes que pueden utilizar como conjunto de entrenamiento para la regresión logística. \n", "Tu tarea es construir un modelo de clasificación que estime la probabilidad de aprobación o negación de un crédito obteniendo a partir de sus ingresos mensuales.

2.2 Visualizando los datos

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

```
df = pd.read_csv("C:\\Users\\WALTER\\Downloads\\datos_credito_bogota_millones.csv")
df.columns = ['Ingresos_Mensuales', 'Monto_deuda_actual','label']
df.head()
```

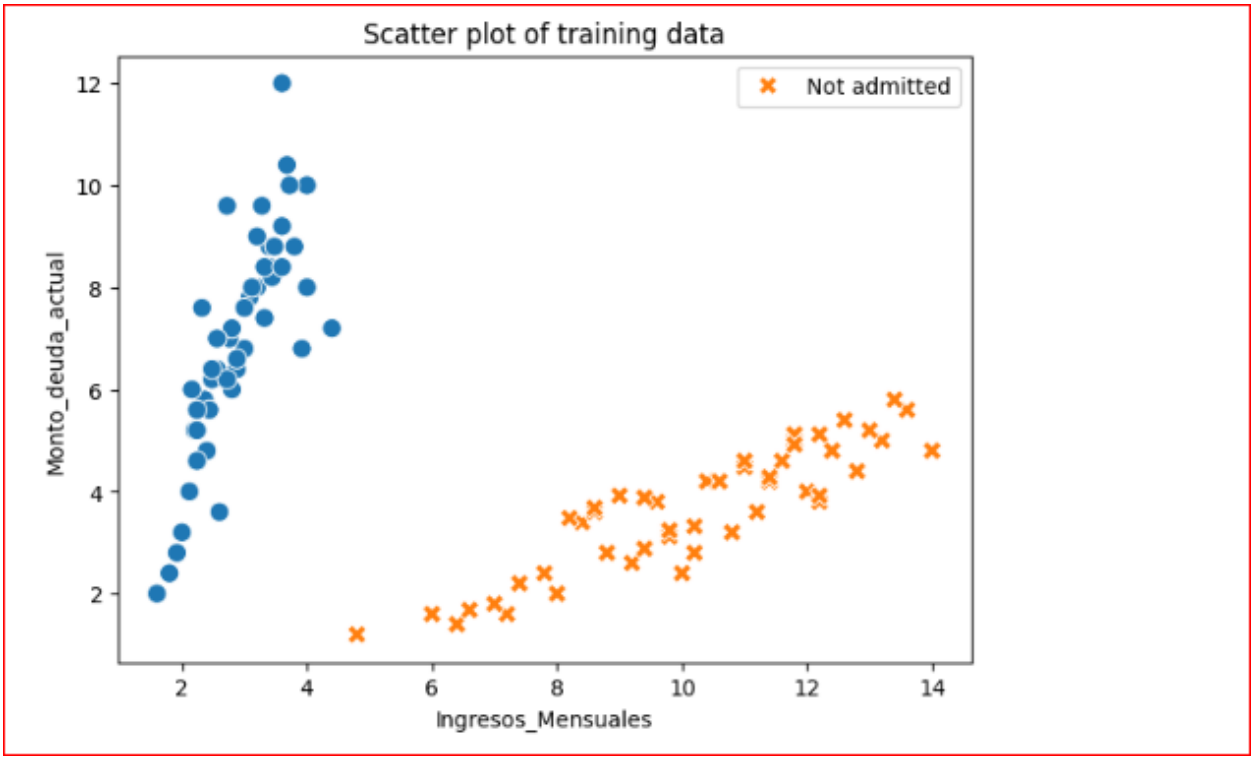
	Ingresos_Mensuales	Monto_deuda_actual	label
0	4.8	1.2	1
1	1.6	2.0	0
2	14.0	4.8	1
3	3.2	8.0	0
4	10.0	2.4	1

```
df.describe().T  # Descripcion de las variables
```


```
df.describe().T  # Descripcion de las variables
```

	count	mean	std	min	25%	50%	75%	max
Ingresos_Mensuales	100.0	6.5532	3.998836	1.6	2.86	4.60	10.25	14.0
Monto_deuda_actual	100.0	5.3016	2.389373	1.2	3.57	4.96	7.00	12.0
label	100.0	0.5000	0.502519	0.0	0.00	0.50	1.00	1.0

```
plt.figure(figsize=(7,5))
ax = sns.scatterplot(x='Ingresos_Mensuales', y='Monto_deuda_actual', hue='label', data=df, style='label', s=80)
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles[1:], ['Not admitted', 'Admitted'])
plt.title('Scatter plot of training data')
plt.show(ax)
#Este gráfico ayuda a visualizar cómo los ingresos y la deuda afectan la probabilidad de aprobación de crédito
```



2.3 Implementacion

2.3.1 Función sigmoidea

Logistic regression hypothesis:

$$h_\theta(x) = g(\theta^T x)$$

$$g(z) = \frac{1}{1+e^{-z}}$$

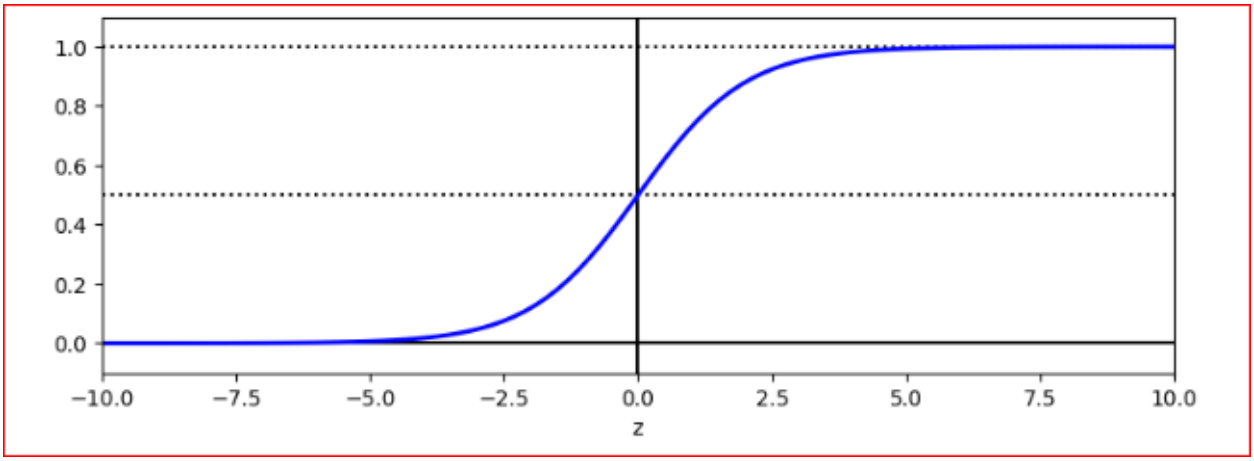
2.3.2 Gráfico de la función sigmoidea:

```python
def sigmoid(z):
    z = np.array(z)
    return 1 / (1+np.exp(-z))
#Este código define la función sigmoide, que es fundamental en modelos de regresión logística
#ayudará a calcular probabilidades en tu modelo
```

Plot of sigmoid function:

```python
import matplotlib.pyplot as plt
%matplotlib inline
z = np.linspace(-10, 10, 100)
sig = sigmoid(z)
plt.figure(figsize=(9, 3))
plt.plot([-10, 10], [0, 0], "k-")
plt.plot([-10, 10], [0.5, 0.5], "k:")
plt.plot([-10, 10], [1, 1], "k:")
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(z, sig, "b-", linewidth=2)
plt.xlabel("z")
plt.axis([-10, 10, -0.1, 1.1])
plt.show()
#- Aplica la función sigmoide a cada valor de z para calcular su transformación.
#El gráfico mostrará la clásica curva sigmoide:
#Se mantiene cerca de 0 para valores negativos de z.
#Se aproxima a 1 para valores positivos de z.
#En z=0, el valor de la sigmoide es 0.5.
```


```python
import matplotlib.pyplot as plt
%matplotlib inline
z = np.linspace(-10, 10, 100)
sig = sigmoid(z)
plt.figure(figsize=(9, 3))
plt.plot([-10, 10], [0, 0], "k-")
plt.plot([-10, 10], [0.5, 0.5], "k:")
plt.plot([-10, 10], [1, 1], "k:")
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(z, sig, "b-", linewidth=2)
plt.xlabel("z")
plt.axis([-10, 10, -0.1, 1.1])
plt.show()
#- Aplica la función sigmoide a cada valor de z para calcular su transformación.
#El gráfico mostrará la clásica curva sigmoide:
#Se mantiene cerca de 0 para valores negativos de z.
#Se aproxima a 1 para valores positivos de z.
#En z=0, el valor de la sigmoide es 0.5.
```

2.4 Función de costo y gradiente

```
def cost_function(theta, X, y):
    m = y.shape[0]
    theta = theta[:, np.newaxis] #truco para hacer que Numpy minimice el trabajo
    h = sigmoid(X.dot(theta))
    J = (1/m) * (-y.T.dot(np.log(h)) - (1-y).T.dot(np.log(1-h)))

    diff_hy = h - y
    grad = (1/m) * diff_hy.T.dot(X)

    return J, grad
#Este código implementa una parte clave del entrenamiento de un modelo de regresión logística: la función de costo y el gra
```

```
m = df.shape[0]
X = np.hstack((np.ones((m,1)),df[['Ingresos_Mensuales', 'Monto_deuda_actual']].values))
y = np.array(df.label.values).reshape(-1,1)
initial_theta = np.zeros(shape=(X.shape[1]))
#- m: número de muestras
# X: matriz de características con columna de unos para el sesgo (bias)
# y: vector columna con las etiquetas
# initial_theta: vector de parámetros inicializado en ceros
```

```
cost, grad = cost_function(initial_theta, X, y)
print('Cost at initial theta (zeros):', cost)
print('Expected cost (approx): 0.693')
print('Gradient at initial theta (zeros):')
print(grad.T)
print('Expected gradients (approx):\n -0.1000\n -12.0092\n -11.2628')
```

```
Cost at initial theta (zeros): [[0.69314718]]
Expected cost (approx): 0.693
Gradient at initial theta (zeros):
[[ 0.    ]
 [-1.8234]
 [ 0.8212]]
Expected gradients (approx):
 -0.1000
 -12.0092
 -11.2628
```

```
#Tu modelo indica que:
# Más ingresos mensuales → mayor probabilidad de aprobación (peso más positivo).
# Más deuda actual → menor probabilidad de aprobación (peso más negativo).
```

```
test_theta = np.array([-24, 0.2, 0.2])
[cost, grad] = cost_function(test_theta, X, y)

print('Cost at test theta:', cost)
print('Expected cost (approx): 0.218')
print('Gradient at test theta:')
print(grad.T)
print('Expected gradients (approx):\n 0.043\n 2.566\n 2.647')
#se esta comparando los resultados reales con valores esperados conocidos para asegurar
#  que la implementación numérica de la función cost_function()
```

```
Cost at test theta: [[10.61408]]
Expected cost (approx): 0.218
Gradient at test theta:
[[-0.5   ]
 [-5.1   ]
 [-1.8296]]
Expected gradients (approx):
 0.043
 2.566
 2.647
```

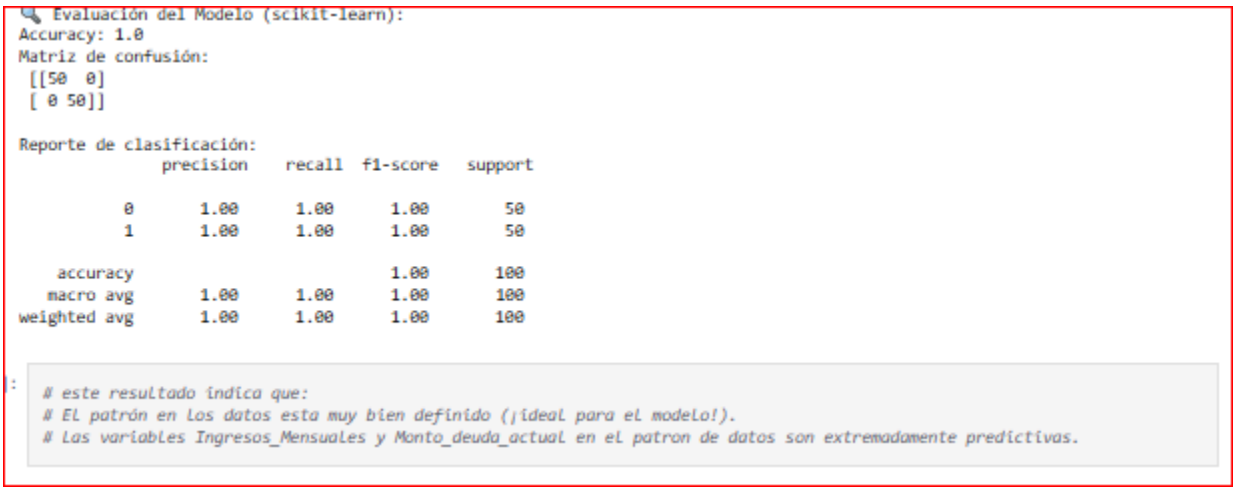
2.4 Codigo Equivalente usando Scikit-Learn:

```
# 1. Importar librerías necesarias
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# 2. Cargar los datos
df = pd.read_csv("C:\\Users\\WALTER\\Downloads\\datos_credito_bogota_millones.csv")
df.columns = ['Ingresos_Mensuales', 'Monto_deuda_actual','label']
df.head()

# 3. Definir X e y
X = df[["Ingresos_Mensuales", "Monto_deuda_actual"]]  # NOTA: aquí no se necesita add_constant
y = df["label"]

# 4. Dividir en entrenamiento y prueba (opcional)
# En este caso se entrena con todos los datos (como con statsmodels)
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)



# 5. Crear y ajustar el modelo de regresión logística
modelo = LogisticRegression(solver='liblinear')

# Solvers disponibles:
# - 'liblinear'  : Bueno para datasets pequeños. Soporta L1 y L2. (por defecto en regresión binaria)
# - 'lbfgs'      : Rápido, robusto y multiclase (one-vs-rest). Ideal para muchos datos. Soporta solo L2.
# - 'newton-cg'  : Similar a lbfgs, pero usa el método de Newton. Solo L2.
# - 'sag'        : Estocástico (bueno para datasets grandes). Solo L2. Requiere que X esté estandarizado.
# - 'saga'       : Como 'sag' pero soporta L1, L2 y elastic net. Funciona bien con datos dispersos (sparse).

modelo.fit(X, y)

# 6. Obtener predicciones de clase y de probabilidad
df["probabilidad"] = modelo.predict_proba(X)[:, 1]
df["prediccion"] = modelo.predict(X)

# 7. Evaluar desempeño del modelo
print("\n🔍 Evaluación del Modelo (scikit-learn):")
print("Accuracy:", round(accuracy_score(y, df["prediccion"]), 4))
print("Matriz de confusión:\n", confusion_matrix(y, df["prediccion"]))
print("\nReporte de clasificación:\n", classification_report(y, df["prediccion"]))
```

NOTAS IMPORTANTES

#LogisticRegression() usa regularización L2 por defecto. Puedes cambiarla con penalty='l1' y usar solver='liblinear' para compatibilidad.
#Puedes ajustar la fuerza de regularización con el parámetro C. Por ejemplo: LogisticRegression(C=0.1) (a menor C, mayor regularización).
#predict_proba() devuelve las probabilidades de pertenecer a cada clase (útil para umbrales distintos de 0.5).

2.5.1 Evaluación del Modelo (scikit-learn):



```
🔍 Evaluación del Modelo (scikit-learn):
Accuracy: 1.0
Matriz de confusión:
 [[50  0]
 [ 0 50]]

Reporte de clasificación:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      1.00      1.00        50

    accuracy                           1.00       100
   macro avg       1.00      1.00      1.00       100
weighted avg       1.00      1.00      1.00       100
```

```
# este resultado indica que:
# El patrón en los datos esta muy bien definido (¡ideal para el modelo!).
# Las variables Ingresos_Mensuales y Monto_deuda_actual en el patron de datos son extremadamente predictivas.
```

2.6 Regresión Logística (Logit clásico) con statsmodels

```python
# 1. Importar librerías necesarias
import numpy as np
import pandas as pd
import statsmodels.api as sm
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix

df = pd.read_csv("C:\\Users\\WALTER\\Downloads\\datos_credito_bogota_millones.csv")
df.columns = ['Ingresos_Mensuales', 'Monto_deuda_actual','label']
df.head()


# 2 Definir variables X e y
X = sm.add_constant(df[["Ingresos_Mensuales", "Monto_deuda_actual"]])  # Agrega columna 'const' para intercepto
y = df["label"]

# 4. Dividir en entrenamiento y prueba (opcional
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# 5. Crear y ajustar el modelo Logit
logit_model = sm.Logit(y, X)
result = logit_model.fit()

# 6. Mostrar resumen estadístico del modelo
print(result.summary())

# 7. Obtener predicciones (probabilidades)
df["probabilidad"] = result.predict(X)

# 8. Clasificación binaria con umbral 0.5
df["prediccion"] = (df["probabilidad"] >= 0.5).astype(int)

# 9. Medir desempeño del modelo
print("\n🔍 Evaluación del Modelo:")
print("Accuracy:", round(accuracy_score(y, df["prediccion"]), 4))
print("Matriz de confusión:\n", confusion_matrix(y, df["prediccion"]))
```

#¿Qué ventajas ofrece statsmodels?
#Permite analizar la significancia estadística de cada variable (p-value).
#Muestra intervalos de confianza y errores estándar.
#Es ideal para interpretación, informes técnicos y papers

```
Warning: Maximum number of iterations has been exceeded.
         Current function value: 0.000000
         Iterations: 35
                           Logit Regression Results
==============================================================================
Dep. Variable:                  label   No. Observations:                  100
Model:                          Logit   Df Residuals:                       97
Method:                           MLE   Df Model:                            2
Date:                Sun, 15 Jun 2025   Pseudo R-squ.:                   1.000
Time:                        11:17:29   Log-Likelihood:            -3.5503e-05
converged:                      False   LL-Null:                       -69.315
Covariance Type:            nonrobust   LLR p-value:                 7.889e-31
======================================================================================
                        coef    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------------
const               -12.5746   3065.055     -0.004      0.997   -6019.971    5994.822
Ingresos_Mensuales    6.6396    947.119      0.007      0.994   -1849.679    1862.958
Monto_deuda_actual   -7.5275   1578.448     -0.005      0.996   -3101.229    3086.174
======================================================================================

Complete Separation: The results show that there iscomplete separation or perfect prediction.
In this case the Maximum Likelihood Estimator does not exist and the parameters
are not identified.
```

2.5.1 Evaluación del Modelo:

```
Accuracy: 1.0
Matriz de confusión:
 [[50  0]
 [ 0 50]]
c:\Users\WALTER\AppData\Local\Programs\Python\Python313\Lib\site-packages\statsmodels\discrete\discrete_model.py:227: Perfect
SeparationWarning: Perfect separation or prediction detected, parameter may not be identified
  warnings.warn(msg, category=PerfectSeparationWarning)
c:\Users\WALTER\AppData\Local\Programs\Python\Python313\Lib\site-packages\statsmodels\discrete\discrete_model.py:227: Perfect
SeparationWarning: Perfect separation or prediction detected, parameter may not be identified
  warnings.warn(msg, category=PerfectSeparationWarning)
c:\Users\WALTER\AppData\Local\Programs\Python\Python313\Lib\site-packages\statsmodels\discrete\discrete_model.py:227: Perfect
SeparationWarning: Perfect separation or prediction detected, parameter may not be identified
  warnings.warn(msg, category=PerfectSeparationWarning)
c:\Users\WALTER\AppData\Local\Programs\Python\Python313\Lib\site-packages\statsmodels\base\model.py:607: ConvergenceWarning:
Maximum Likelihood optimization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

 # Accuracy: 1.0 y matriz de confusión perfecta
 # El modelo clasificó todos los ejemplos correctamente.

2.7 Interpretación de los Coeficientes en Regresión Logística

2.7.1 ¿Qué son los odds?

La razón de probabilidades (odds) se calcula así:

$$\text{odds} = \frac{p}{1-p}$$

Donde p es la probabilidad de que ocurra el evento (por ejemplo, ser admitido).

Probabilidad p	Odds $\frac{p}{1-p}$
0.50 (50%)	1
0.75 (75%)	3
0.20 (20%)	0.25
0.90 (90%)	9

2.8 Modelo Logístico

La regresión logística modela el **logit** de la probabilidad como una combinación lineal de variables:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

- El lado izquierdo es el **logit** (logaritmo de los odds).
- Cada coeficiente βi representa el cambio en el log-odds.
- Usamos eβi para interpretar directamente sobre los odds.

2.8.1 Interpretación de (Razón de odds)

Tipo de variable	Interpretación de e^{β}
Continua	Por cada unidad adicional, los **odds** del evento se multiplican por e^{β}.
Categórica binaria	Al pasar de la categoría base (0) a la comparada (1), los odds se multiplican por e^{β}.
Categórica (dummies)	Comparado con la categoría de referencia, los odds se multiplican por e^{β}.

Ejemplo 1: Variable Continua

Supón que:

β=0.2⇒e0.2≈1.22

Interpretación:
Por cada unidad adicional en la variable (ej. puntaje de examen), los **odds del evento aumentan un 22%**.
La probabilidad también aumenta, aunque no de forma lineal.

Ejemplo 2: Variable Categórica Binaria

Supón que:

$\beta = -0.4 \Rightarrow e^{-0.4} \approx 0.67$

Interpretación:
Pasar de la categoría base a la comparada **reduce los odds del evento en 33%**.
Esto implica que la **probabilidad del evento disminuye**.

Notas clave

- β afecta el **logaritmo de los odds**.
- e^{β} afecta directamente los **odds** (razón de probabilidades).
- La **probabilidad** no cambia de forma lineal respecto a β.

Resumen

“Cada coeficiente

nos dice cómo cambia la razón de que ocurra frente a que no ocurra el evento.

Si tomamos

, eso nos dice cuántas veces más o menos probable es que ocurra el evento cuando cambia esa variable.”

Calculo de la Curva ROC

2.9 ¿Cómo surge la curva ROC?

La curva ROC (Receiver Operating Characteristic) muestra gráficamente la capacidad de un modelo de clasificación binaria para distinguir entre clases a medida que varía el umbral de decisión.

Se construye comparando los valores predichos como probabilidades con los valores reales (0 o 1).

2.9.1 ¿Cómo se construye paso a paso?

1. El modelo predice probabilidades

Por ejemplo:

Observación	Probabilidad predicha
1	0.90
2	0.75
3	0.60
4	0.45
5	0.20

2.9.2 Se elige un umbral

Un modelo de clasificación convierte las probabilidades en clases con un **umbral**.

- Umbral típico: **0.5**
- Si cambiamos el umbral, cambian las predicciones y la matriz de confusión.

. Para cada umbral se calcula:

- **TPR** (True Positive Rate o Sensibilidad):

$$TPR = \frac{TP}{TP + FN}$$

- **FPR** (False Positive Rate o 1 - especificidad):

$$FPR = \frac{FP}{FP + TN}$$

2.9.3 Se repite para muchos umbrales

Por ejemplo: desde 1.0 hasta 0.0, en pasos.

Para cada umbral:

- Se generan predicciones binarias
- Se calcula la matriz de confusión
- Se obtienen TPR y FPR
- Se guarda el punto ((FPR, TPR))

2.9.4 . Se grafica la curva

- Eje **X**: FPR (Tasa de Falsos Positivos)
- Eje **Y**: TPR (Tasa de Verdaderos Positivos)

Curva ROC={(FPR,TPR) para todos los umbrales}

- Un modelo **perfecto** alcanza el punto (0, 1)
- Un modelo **aleatorio** genera una línea diagonal de (0, 0) a (1, 1)

2.9.5 ¿Qué representa la curva ROC?

Muestra el **compromiso entre sensibilidad y especificidad**.

Cuanto más arriba y a la izquierda esté la curva, **mejor es el modelo**.

Se puede resumir con el **AUC (Área bajo la curva ROC)**.

AUC y su interpretación

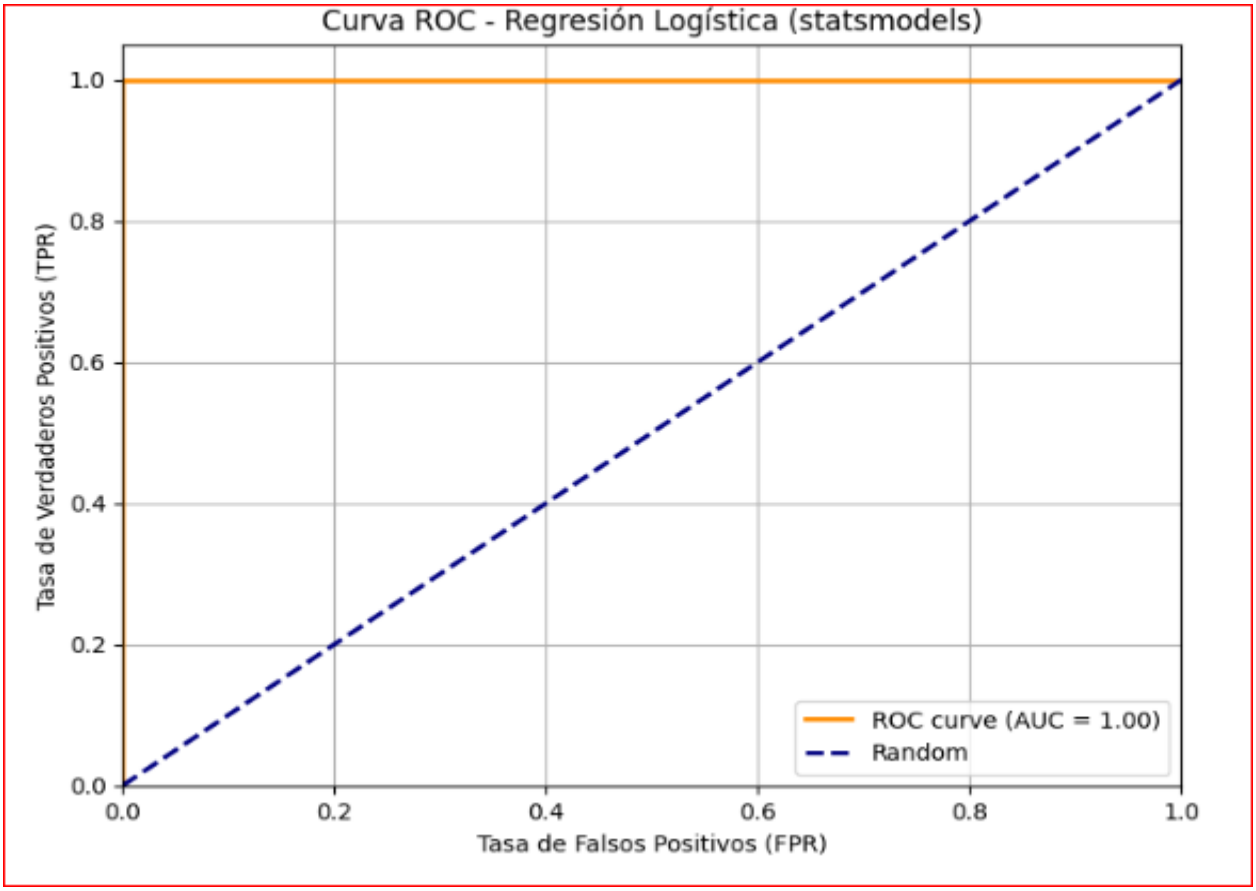
AUC	Interpretación
0.5	Modelo aleatorio
0.6 – 0.7	Discriminación pobre
0.7 – 0.8	Aceptable
0.8 – 0.9	Buena
0.9 – 1.0	Excelente
1.0	Perfecto (sospechoso en la práctica)

2.9.6 Resumen

1. El modelo predice probabilidades.
2. Se prueban muchos umbrales.
3. Para cada uno, se calcula TPR y FPR.
4. Se grafican los puntos ((FPR, TPR)).
5. Se analiza el **AUC** para medir la calidad general.

La curva ROC no depende de un solo umbral y es muy útil cuando las clases están desbalanceadas.

```python
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt

# 10. Calcular valores para la curva ROC
fpr, tpr, thresholds = roc_curve(y, df["probabilidad"])
roc_auc = roc_auc_score(y, df["probabilidad"])

# 11. Graficar la curva ROC
plt.figure(figsize=(8,6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Tasa de Falsos Positivos (FPR)')
plt.ylabel('Tasa de Verdaderos Positivos (TPR)')
plt.title('Curva ROC - Regresión Logistica (statsmodels)')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
```



 # - FPR = 0 y TPR = 1: señal de que el modelo no comete errores al clasificar positivos y negativos.
 # La línea punteada azul representa un clasificador aleatorio para comparación.
 # El área bajo la curva (AUC) igual a 1 respalda lo que ya veías en la matriz de confusión: precisión total.

2.10 Mas Metricas de la matriz de confusion

```python
from sklearn.metrics import confusion_matrix

def metricas_matriz_confusion(y_verdadero, y_predicho):
    """
    Calcula e imprime los principales indicadores derivados de la matriz de confusión
    para un problema de clasificación binaria.

    Parámetros:
    - y_verdadero: etiquetas reales (lista, array o Series)
    - y_predicho: etiquetas predichas (lista, array o Series)
    """
    # Extraer verdaderos y falsos positivos/negativos
    tn, fp, fn, tp = confusion_matrix(y_verdadero, y_predicho).ravel()

    # Cálculo de métricas
    exactitud     = (tp + tn) / (tp + tn + fp + fn)
    precision     = tp / (tp + fp) if (tp + fp) > 0 else 0
    sensibilidad  = tp / (tp + fn) if (tp + fn) > 0 else 0  # también llamada Recall
    especificidad = tn / (tn + fp) if (tn + fp) > 0 else 0
    valor_f1      = 2 * (precision * sensibilidad) / (precision + sensibilidad) if (precision + sensibilidad) > 0 else 0
    tasa_falsos_positivos = fp / (fp + tn) if (fp + tn) > 0 else 0

    # Imprimir resultados
    print("📊 Métricas de la Matriz de Confusión")
    print(f" Verdaderos Positivos (VP / TP): {tp}")
    print(f" Falsos Positivos (FP): {fp}")
    print(f" Falsos Negativos (FN): {fn}")
    print(f" Verdaderos Negativos (VN / TN): {tn}")
    print(f"\n Exactitud:     {exactitud:.4f}")
    print(f" Precisión:     {precision:.4f}")
    print(f" Sensibilidad:  {sensibilidad:.4f}")
    print(f" Especificidad: {especificidad:.4f}")
    print(f" Valor F1:      {valor_f1:.4f}")
    print(f" Tasa FP:       {tasa_falsos_positivos:.4f}")
```

```python
metricas_matriz_confusion(y, df["prediccion"])
```

Métricas de la Matriz de Confusión
Verdaderos Positivos (VP / TP): 50
Falsos Positivos (FP): 0
Falsos Negativos (FN): 0
Verdaderos Negativos (VN / TN): 50

Exactitud: 1.0000
Precisión: 1.0000
Sensibilidad: 1.0000
Especificidad: 1.0000
Valor F1: 1.0000
Tasa FP: 0.0000

3. Comparación de Modelos: Regresión Logística vs Árbol de Decisión (Python)

En esta clase realizaremos una comparación entre dos modelos de clasificación binaria: **Regresión Logística (Logit)** y **Árbol de Decisión**, usando el dataset de créditos

3.1 Importar Librerías

```python
import numpy as np  # Biblioteca para cálculos numéricos y manejo de arrays multidimensionales
import pandas as pd  # Libreria para análisis y manipulación de datos en estructuras tipo tabla (DataFrame)
import matplotlib.pyplot as plt  # Herramienta para generar gráficos en 2D como líneas, barras y dispersión
import seaborn as sns  # Biblioteca basada en matplotlib para visualizaciones estadísticas más atractivas

from sklearn.datasets import load_breast_cancer  # Carga el dataset de cáncer de mama de sklearn
from sklearn.model_selection import train_test_split  # Divide el dataset en conjuntos de entrenamiento y prueba
from sklearn.linear_model import LogisticRegression  # Importa el modelo de regresión logística (Logit)
from sklearn.tree import DecisionTreeClassifier  # Importa el modelo de árbol de decisión

from sklearn.metrics import (  # Métricas para evaluar el rendimiento de los modelos
    confusion_matrix,         # Crea la matriz de confusión: TP, FP, FN, TN
    classification_report,    # Genera métricas como precisión, recall y F1-score
    roc_curve,                # Calcula la curva ROC (tasa de verdaderos positivos vs falsos positivos)
    auc,                      # Calcula el área bajo la curva ROC (AUC)
    RocCurveDisplay           # Herramienta para graficar la curva ROC fácilmente
)
```

```
       Ingresos_Mensuales  Monto_deuda_actual
count          100.000000         100.000000
mean             6.553200           5.301600
std              3.998836           2.389373
min              1.600000           1.200000
25%              2.860000           3.570000
50%              4.600000           4.960000
75%             10.250000           7.000000
max             14.000000          12.000000
label
0          50
1          50
Name: count, dtype: int64
```

3.2 Dataset de Evaluación Crediticia

Este conjunto de datos simula evaluaciones de crédito en base a ingresos y deudas mensuales. Es ideal para practicar modelos de clasificación binaria

Información general
Número de muestras: 100
Número de características (variables): 2
Variable objetivo (target): Aprobación del crédito
0: Crédito no aprobado
1: Crédito aprobado

3.3 Variable Objetivo: target

Valor	Diagnóstico crediticio
0	Crédito denegado (no aprobado)
1	Crédito aprobado

3.4 Descripción de las Variables Predictoras

Cada variable se calcula a partir de una imagen digitalizada de una muestra de tejido.
Para 10 atributos básicos, se generan 3 tipos de métricas:

- **Media (mean)**
- **Error estándar (error)**
- **Peor valor (worst)**

3.4.1 Atributos Básicos

Atributo	Descripción
Ingresos_Mensuales	Ingreso mensual declarado por el solicitante (en millones de COP)
Monto_deuda_actual	Total de deuda mensual actual (en millones de COP)

3.4.2 Lista completa de variables

Variables tipo mean (promedios)

- mean radius
- mean texture
- mean perimeter
- mean area
- mean smoothness
- mean compactness
- mean concavity
- mean concave points
- mean symmetry
- mean fractal dimension

2.4.2 Variables tipo error (error estándar)

- radius error
- texture error
- perimeter error
- area error
- smoothness error
- compactness error
- concavity error
- concave points error
- symmetry error
- fractal dimension error

3.4.3 Variables tipo worst (valores más extremos)

- worst radius
- worst texture
- worst perimeter
- worst area
- worst smoothness
- worst compactness
- worst concavity
- worst concave points
- worst symmetry
- worst fractal dimension

3.5 Aplicaciones del dataset

Este dataset es ideal para:

- Modelos de clasificación binaria
- Comparar algoritmos como:
- Regresión Logística
- Árboles de Decisión

```
X.head()
```

	Ingresos_Mensuales	Monto_deuda_actual
0	4.8	1.2
1	1.6	2.0
2	14.0	4.8
3	3.2	8.0
4	10.0	2.4

3.6 Particionar los Datos

La partición de los datos en conjuntos de entrenamiento y prueba es fundamental en machine learning porque permite evaluar de forma objetiva la capacidad del modelo para generalizar a datos nuevos. Si un modelo se entrena y evalúa sobre el mismo conjunto, corre el riesgo de memorizar los datos (sobreajuste) en lugar de aprender patrones reales. Al separar un subconjunto exclusivo para prueba, se simula cómo se comportaría el modelo en un entorno real, garantizando que las métricas obtenidas reflejen su verdadero desempeño. Además, usar una semilla aleatoria asegura que esta división sea reproducible, lo que es esencial para comparar modelos de manera justa y transparente.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
```

3.7 Entrenar los Modelos

Por qué es importante usar semillas
Asegura que la partición sea reproducible. Si tú o un estudiante corre el código varias veces, obtendrán el mismo resulatdo Ideal para clases, investigación o pruebas controladas.

 # 🔷 Modelo de Regresión Logística (Logit)
log_model = LogisticRegression(max_iter=10000)
 # Creamos el modelo logístico. Se establece max_iter=10000 para asegurar que el algoritmo converge,
 # especialmente cuando hay muchas variables o los datos requieren más iteraciones.

log_model.fit(X_train, y_train) *# Entrenamos el modelo usando los datos de entrenamiento. Aquí el modelo "aprende" la relación entre las variables X e y.*
y_pred_log = log_model.predict(X_test) *# Generamos predicciones de clase (0 o 1) sobre el conjunto de prueba. Esto es lo que el modelo cree que es la clase verdadera.*

y_proba_log = log_model.predict_proba(X_test)[:, 1] *# Obtenemos las probabilidades predichas de que la clase sea "1" (benigno).*
 # Esto se usa para calcular curvas ROC, métricas de umbral, etc.

 # 🌲 Modelo de Árbol de Decisión
tree_model = DecisionTreeClassifier(random_state=42) *# Creamos el árbol de decisión. Usamos random_state para que los resultados sean reproducibles.*
tree_model.fit(X_train, y_train) *# Entrenamos el árbol con los datos de entrenamiento. El árbol genera reglas basadas en divisiones de los datos.*

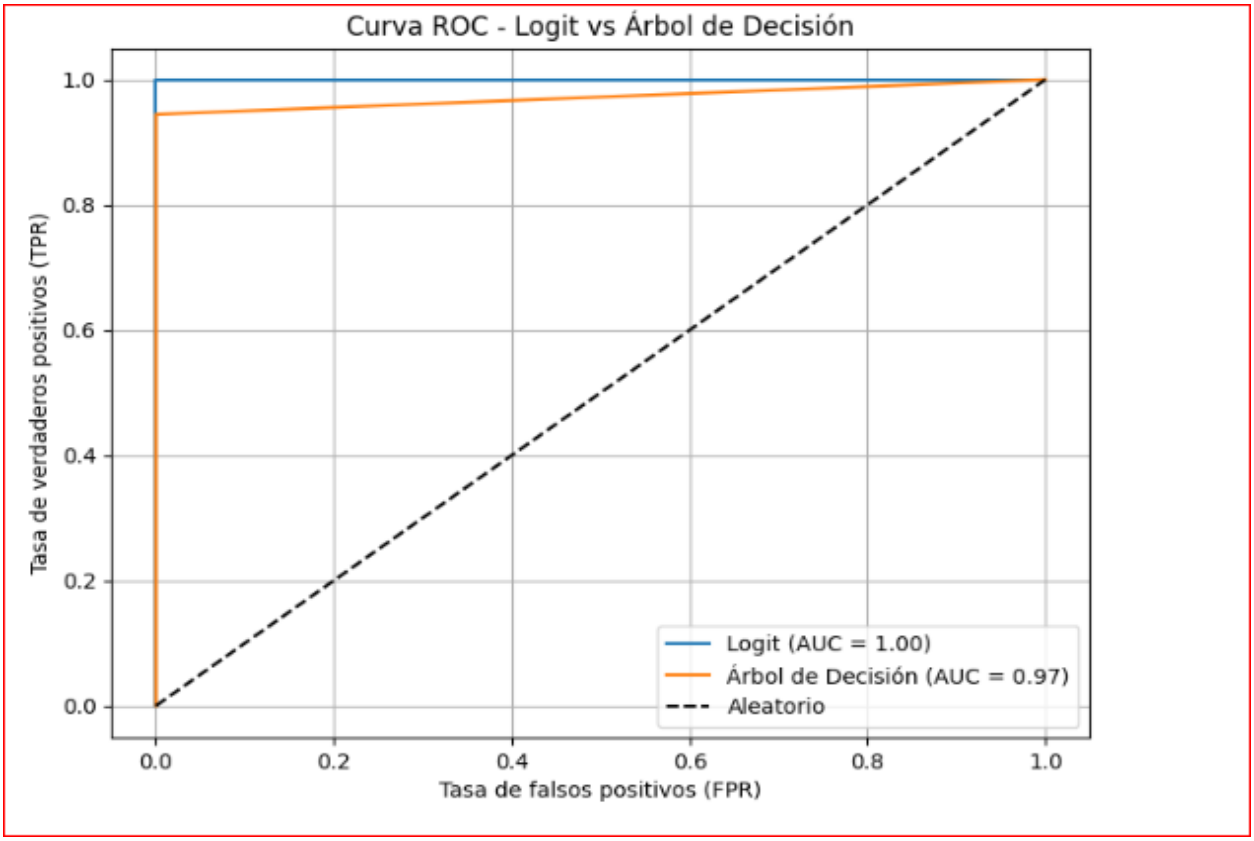
y_pred_tree = tree_model.predict(X_test)
 # Realizamos predicciones de clase (0 o 1) sobre los datos de prueba usando el árbol entrenado.

y_proba_tree = tree_model.predict_proba(X_test)[:, 1]
 # Extraemos la probabilidad predicha de que la clase sea "1".
 # Igual que en el logit, esto sirve para métricas como AUC o análisis de umbrales de decisión.

c:\Users\WALTER\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\utils\validation.py:1406: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using ravel().
  y = column_or_1d(y, warn=True)

3.4 Curvas ROC y AUC

```
# Calcular los puntos para la curva ROC del modelo Logit
fpr_log, tpr_log, _ = roc_curve(y_test, y_proba_log)  # Falsos positivos y verdaderos positivos para cada umbral
roc_auc_log = auc(fpr_log, tpr_log)  # Área bajo la curva ROC para el modelo Logit

# Calcular los puntos para la curva ROC del modelo Árbol de Decisión
fpr_tree, tpr_tree, _ = roc_curve(y_test, y_proba_tree)
roc_auc_tree = auc(fpr_tree, tpr_tree)

# Crear una sola figura para mostrar ambas curvas ROC
plt.figure(figsize=(8, 6))

# Dibujar curva ROC para el modelo Logit
plt.plot(fpr_log, tpr_log, label=f"Logit (AUC = {roc_auc_log:.2f})")

# Dibujar curva ROC para el modelo Árbol
plt.plot(fpr_tree, tpr_tree, label=f"Árbol de Decisión (AUC = {roc_auc_tree:.2f})")

# Agregar línea diagonal de referencia (modelo aleatorio)
plt.plot([0, 1], [0, 1], 'k--', label="Aleatorio")

# Configurar la gráfica
plt.title("Curva ROC - Logit vs Árbol de Decisión")
plt.xlabel("Tasa de falsos positivos (FPR)")
plt.ylabel("Tasa de verdaderos positivos (TPR)")
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
# Este código compara visualmente el desempeño de dos modelos de clasificación binaria:
#  uno de regresión logística (Logit) y otro de árbol de decisión,
# usando la Curva ROC como herramienta de evaluación
```

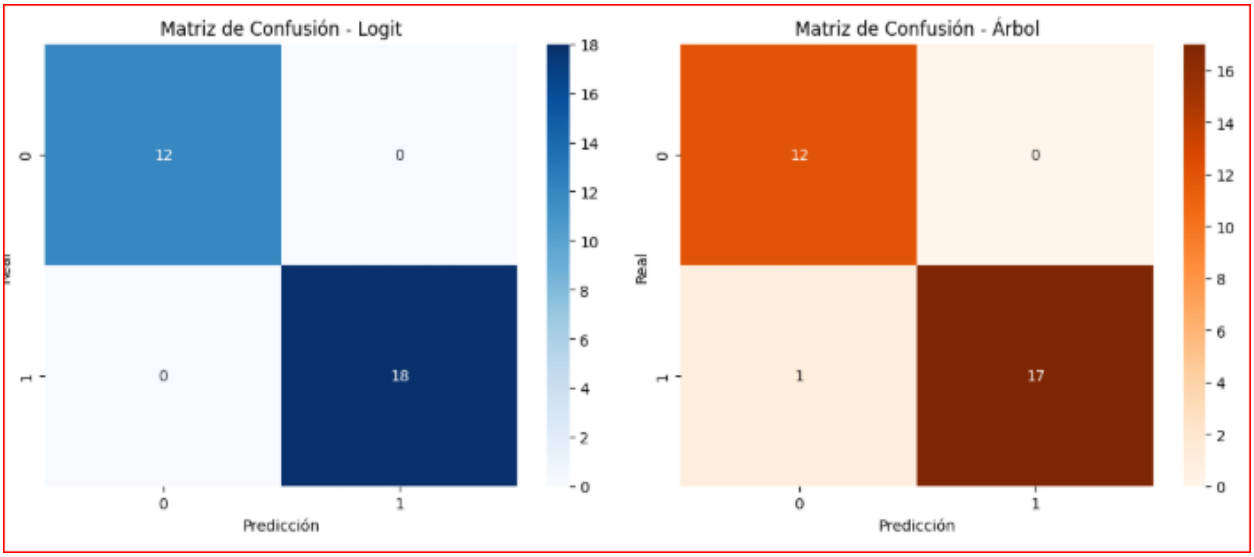


 # La curva ROC muestra claramente que ambos modelos —el logit y el árbol de decisión— tienen un desempeño sobresaliente:

 # Logit (AUC = 1.00): separación perfecta entre clases. El modelo es capaz de distinguir sin errores entre créditos aprobados y no aprobados, sin importar el umbral.
 # Árbol de Decisión (AUC = 0.97): también excelente. Tiene una ligera curva más alejada del ángulo superior izquierdo en comparación al logit, pero sigue siendo un modelo altamente preciso

3.5 Matriz de Confusión y Métricas

```
conf_log = confusion_matrix(y_test, y_pred_log)
conf_tree = confusion_matrix(y_test, y_pred_tree)

report_log = classification_report(y_test, y_pred_log, output_dict=True)
report_tree = classification_report(y_test, y_pred_tree, output_dict=True)

fig, ax = plt.subplots(1, 2, figsize=(12, 5))
sns.heatmap(conf_log, annot=True, fmt="d", cmap="Blues", ax=ax[0])
ax[0].set_title("Matriz de Confusión - Logit")
ax[0].set_xlabel("Predicción")
ax[0].set_ylabel("Real")

sns.heatmap(conf_tree, annot=True, fmt="d", cmap="Oranges", ax=ax[1])
ax[1].set_title("Matriz de Confusión - Árbol")
ax[1].set_xlabel("Predicción")
ax[1].set_ylabel("Real")

plt.tight_layout()
plt.show()
```



 # Matriz de Confusión – Logit
#TP = 18, TN = 12, FP = 0, FN = 0

 # El modelo de regresión logística clasificó todos los casos correctamente. No cometió errores, ni rechazó a clientes aprobables ni aprobó a clientes no elegibles.

 # Esto confirma lo que ya habías visto con el AUC de 1.00: separación perfecta entre clases en el conjunto de prueba.

 # Matriz de Confusión – Árbol de Decisión
 # TP = 17, TN = 12, FP = 0, FN = 1
 # El árbol cometió un único error: rechazó por error a un cliente que sí debía ser aprobado.

 # Aunque su rendimiento sigue siendo muy alto, esto se ve reflejado en el AUC ligeramente menor (0.97).

3.6 Comparación de Métricas en Tabla

```
df_comp = pd.DataFrame({
    "Métrica": ["Accuracy", "Precision clase 1", "Recall clase 1", "F1-score clase 1", "F1-score macro"],
    "Logit": [
        report_log["accuracy"],
        report_log["1"]["precision"],
        report_log["1"]["recall"],
        report_log["1"]["f1-score"],
        report_log["macro avg"]["f1-score"]
    ],
    "Árbol de Decisión": [
        report_tree["accuracy"],
        report_tree["1"]["precision"],
        report_tree["1"]["recall"],
        report_tree["1"]["f1-score"],
        report_tree["macro avg"]["f1-score"]
    ]
})

print(df_comp)
```

```
             Métrica  Logit  Árbol de Decisión
0           Accuracy    1.0          0.966667
1  Precision clase 1    1.0          1.000000
2     Recall clase 1    1.0          0.944444
3   F1-score clase 1    1.0          0.971429
4     F1-score macro    1.0          0.965714
```

```
#El modelo de regresión logística mostró un rendimiento impecable, con un 100 % de exactitud,
#  sensibilidad y precisión en la detección de solicitudes de crédito aprobadas. El árbol de decisión,
# si bien también tuvo un rendimiento sobresaliente, presentó una leve caída en sensibilidad y F1-score,
# producto de un falso negativo en las predicciones
```

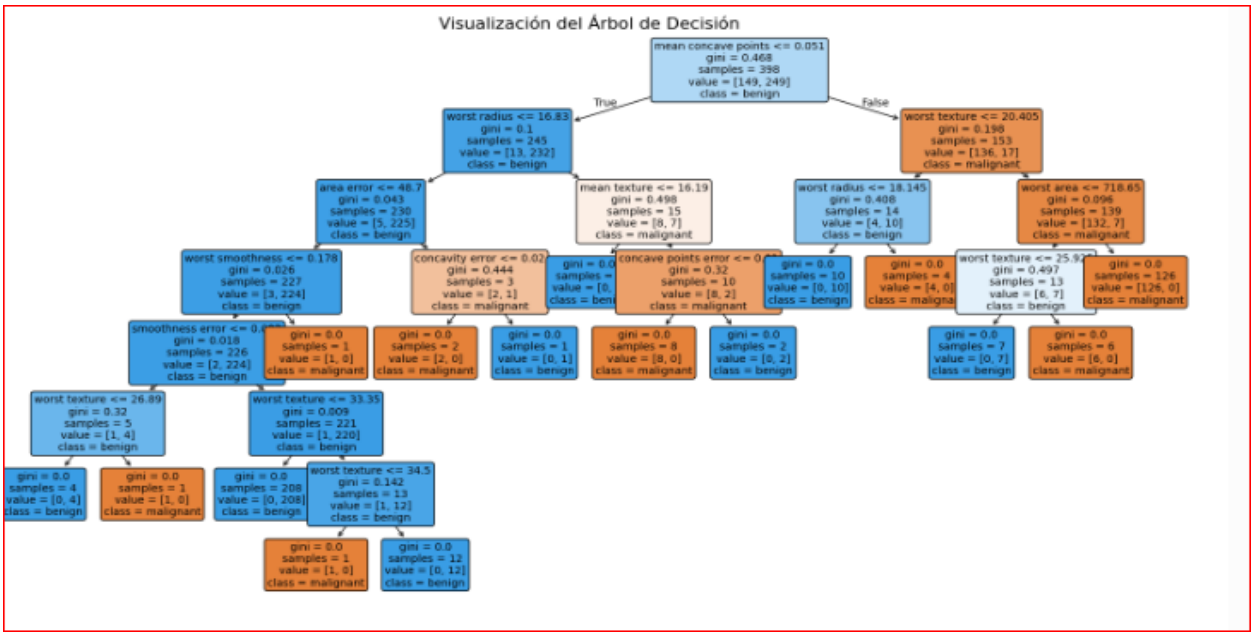
#El modelo de regresión logística mostró un rendimiento impecable, con un 100 % de exactitud
 # sensibilidad y precisión en la detección de solicitudes de crédito aprobadas. El árbol de decisión,
 # si bien también tuvo un rendimiento sobresaliente, presentó una leve caída en sensibilidad y F1-score,
 # producto de un falso negativo en las predicciones

3.7 Conclusión

El modelo logit es más estable y con menor tasa de falsos negativos.
El árbol de decisión puede capturar relaciones no lineales pero es más propenso al sobreajuste.

Ambos modelos son útiles dependiendo del contexto.

```
# Importar librerías necesarias
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
import pandas as pd

# Cargar el dataset de cáncer de mama
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target, name="target")

# Partir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Crear y entrenar el modelo de árbol de decisión
tree_model = DecisionTreeClassifier(random_state=42)
tree_model.fit(X_train, y_train)

# Visualizar el árbol
plt.figure(figsize=(20, 10))  # Ajustar el tamaño de la figura
plot_tree(
    tree_model,
    feature_names=data.feature_names,  # Nombres de las variables
    class_names=data.target_names,     # Nombres de las clases (benigno, maligno)
    filled=True,                       # Colorear los nodos según la clase
    rounded=True,                      # Bordes redondeados
    fontsize=10                        # Tamaño de la fuente en los nodos
)
plt.title("Visualización del Árbol de Decisión", fontsize=16)
plt.show()
```



Un árbol de decisión trabaja dividiendo los datos en ramas basadas en preguntas binarias sobre las variables predictoras, con el objetivo de separar las clases de la forma más pura posible en cada nodo. Comienza en un nodo raíz y, en cada paso, elige la variable y el umbral que mejor separan las clases según una métrica de impureza (como Gini o entropía). A medida que se crean divisiones, se forman nodos internos que representan decisiones y hojas finales que contienen la predicción de clase. Este proceso continúa hasta que se cumplen ciertos criterios de parada

Conclusiones

- El modelo de regresión logística clasificó todos los casos correctamente. No cometió errores, ni rechazó a clientes aprobables ni aprobó a clientes no elegibles.
- El modelo es capaz de distinguir sin errores entre créditos aprobados y no aprobados, sin importar el umbral.
- Al realizar el análisis de los dos modelos de regresión logística con una variable y con dos variables, podemos concluir que es más fácil de predecir los resultados cuando se tienen dos variables, sin embargo el modelo con una sola variables es mucho más fácil de interpretar
- La curva ROC muestra claramente que ambos modelos —el logit y el árbol de decisión— tienen un desempeño sobresaliente