4WeekTutorial_M2003B_contestado

September 28, 2025

1 Actividad 4: M2003B

Author: A. Ramirez-Morales (andres.ramirez@tec.mx)

1.1 Instrucciones:

- Active el kernel proveniente de `Anaconda`
- Complete las funciones donde vea líneas de código inconclusas
- Use comentarios para documentar de manera integral sus funciones
- Pruebe sus funciones con distintos parámetros
- Aumente las explicaciones en el Markdown y en el código
- Procure NO usar chatGPT ú otra tecnología similar, usted tiene la capacidad intelectual suficiente para resolverlo por usted mismo
- Use la documentación oficial de las librerías que se utilizan
- Se entrega un archivo PDF CANVAS como lo indique el profesor

[1]:
```
# cargar librerías básicas
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm, chi2, t, kstest, shapiro
```

1.2

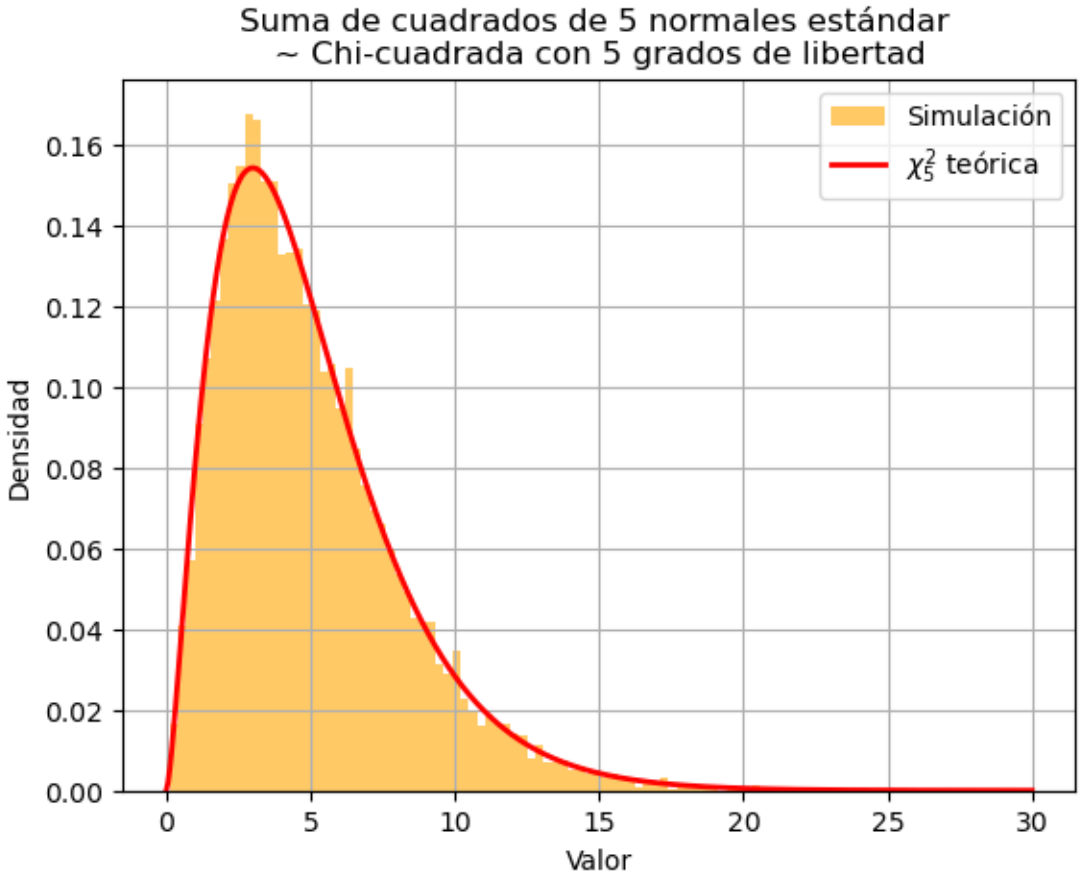
1.3 1. Preliminares estadísticas

1.3.1 1.0 Distribución del estimador de $\hat{\sigma}^2$

Ejercicio: Demuestre numéricamente que la suma de los cuadrados de variables normales independientes sigue una chi2. Complete el código siguiente:

[3]:
```
n_samples = 10000
k = 5 # número de normales estándar

# generar k variables normales estándar por simulación
normals = np.random.randn(n_samples, k)

# calcular suma de cuadrados de cada fila
sum_of_squares = np.sum(normals**2, axis=1)

# comparar con la distribución chi-cuadrada teórica
```

```
x = np.linspace(0, 30, 500)
plt.hist(sum_of_squares, bins=100, density=True, alpha=0.6, color='orange',␣
 ↪label='Simulación')
plt.plot(x, chi2.pdf(x, df=k), 'r-', lw=2, label=r'$\chi^2_5$ teórica')
plt.title(f'Suma de cuadrados de {k} normales estándar \n~ Chi-cuadrada con {k}␣
 ↪grados de libertad')
plt.xlabel('Valor')
plt.ylabel('Densidad')
plt.legend()
plt.grid(True)
plt.show()
```



1.3.2 1.1 Distribuciones Student-t

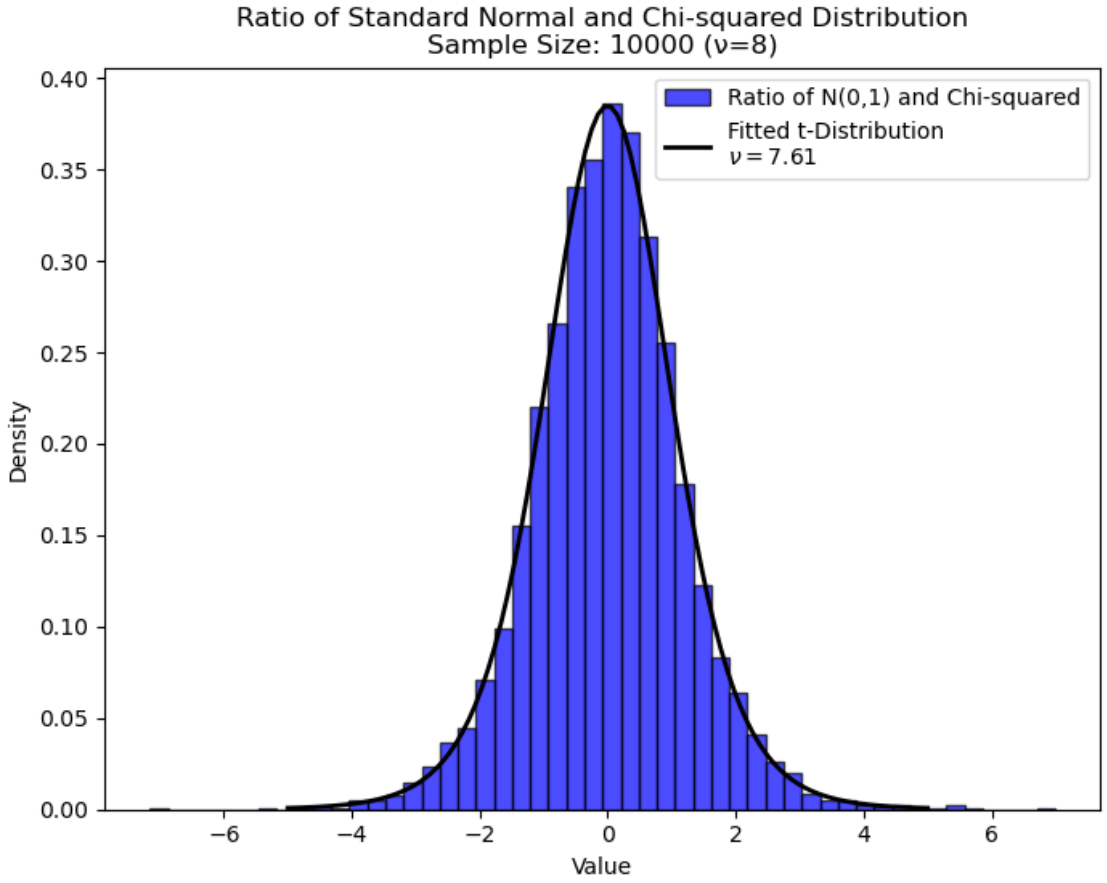
Ejercicio: Demuestre numéricamente que la razón de una distribución Gaussiana y una distribución χ^2 da como resultado una distribución Student-t. Asegúrese que la distribución que esta construyendo cumple con lo anterior usanda la prueba estadística de Kolmogorov-Smirnov.

[1]:
```
def student_t_distribution_check(num_samples=10000, n=10):
    """
    Documentar
    """
    import numpy as np
    import matplotlib.pyplot as plt
    from scipy.stats import t, kstest

    # calcular el # grado de libertad n para la t-distribution
    nu = n - 2

    # generar una muestra para una distribucion normal estandar
    X = np.random.normal(0, 1, num_samples)

    # generar muestras para una distribucion chi2 con (n-2) grados de libertad
    # [Sugerencia: np.random.chisquare()]
    Y = np.random.chisquare(nu, num_samples)

    # calcular la cantidad pivotal t student
    # [Sugerencia: use definicion vista en clase]
    T = X / np.sqrt(Y / nu)

    # ajustar una distribucion a t-distribution a la cantidad T
    # [Sugerencia: t.fit()]
    t_shape, t_loc, t_scale = t.fit(T)

    # realizar la prueba estadistica KS
    # [Sugerencia: kstest()]
    D, p_value = kstest(T, 't', args=(t_shape, t_loc, t_scale))

    # graficar el histograma
    plt.figure(figsize=(8, 6))
    count, bins, _ = plt.hist(T, bins=50, density=True, alpha=0.7,␣
↪color='blue', edgecolor='black', label='Ratio of N(0,1) and Chi-squared')

    # graficar la funcion ajustada
    x = np.linspace(-5, 5, 100)
    p = t.pdf(x, t_shape, loc=t_loc, scale=t_scale)  # PDF of the fitted␣
↪t-distribution
    plt.plot(x, p, 'k', linewidth=2, label=f'Fitted␣
↪t-Distribution\n$\u03bd={t_shape:.2f}$')

    # nombres
    plt.title(f'Ratio of Standard Normal and Chi-squared Distribution\nSample␣
↪Size: {num_samples} (\u03bd={nu})')
    plt.xlabel('Value')
    plt.ylabel('Density')
```

```
    plt.legend()
    plt.show()

    # imrprimir numero degrados de libertad y el p-value
    print(f"Estimated degrees of freedom: {t_shape:.2f}")
    print(f"KS test p-value: {p_value:.5f}")

    # inteprete con mensajes el p-value
    if p_value > 0.05:
        print("No se rechaza H0: la muestra sigue una distribución t con buen␣
↪ajuste.")
    else:
        print("Se rechaza H0: la muestra no parece provenir de una distribución␣
↪t.")
```

1.3.3 Pruebe su función

[3]:
```
# escriba aquí su código
student_t_distribution_check()
```



```
Estimated degrees of freedom: 7.61
KS test p-value: 0.88433
No se rechaza H0: la muestra sigue una distribución t con buen ajuste.
```

1.3.4 1.2 Valores críticos de la estadística-t

Ejercicio: Complete la siguiente función que obtiene el valor critico de la distribución Student-t para n-2 grados de libertad.

[4]:
```python
def critical_t_values(alpha, n):
    """
    Calcula el valor crítico t para una prueba de dos colas
    con nivel de significancia alpha y muestra de tamaño n.
    """
    from scipy import stats

    # calcular el numero de grados de libertad
    df = n - 1

    # obtener los valores t criticos para una prueba de dos colas
    # [Sugerencia: stats.t.ppf()]
    t_critical = stats.t.ppf(1 - alpha/2, df)

    return t_critical
```

1.3.5 Pruebe su función

[5]:
```python
alpha = 0.05
n = 10  # tamaño de la muestra
critical_value = critical_t_values(alpha, n)
print(f"Critical t-values for alpha={alpha} and n={n}: {critical_value}")
```

```
Critical t-values for alpha=0.05 and n=10: 2.2621571628540993
```

1.3.6 1.3 Cantidad pivotal

Una cantidad pivotal es una función que depende de los parámetros de otra distribución dada. Además, la distribución de esta cantidad está libre de parámetros desconocidos. Por ejemplo tenemos la función

$$z = \frac{x - \mu}{\sigma},$$

cuya distribución es una normal con $\mu = 0, \sigma = 1$.

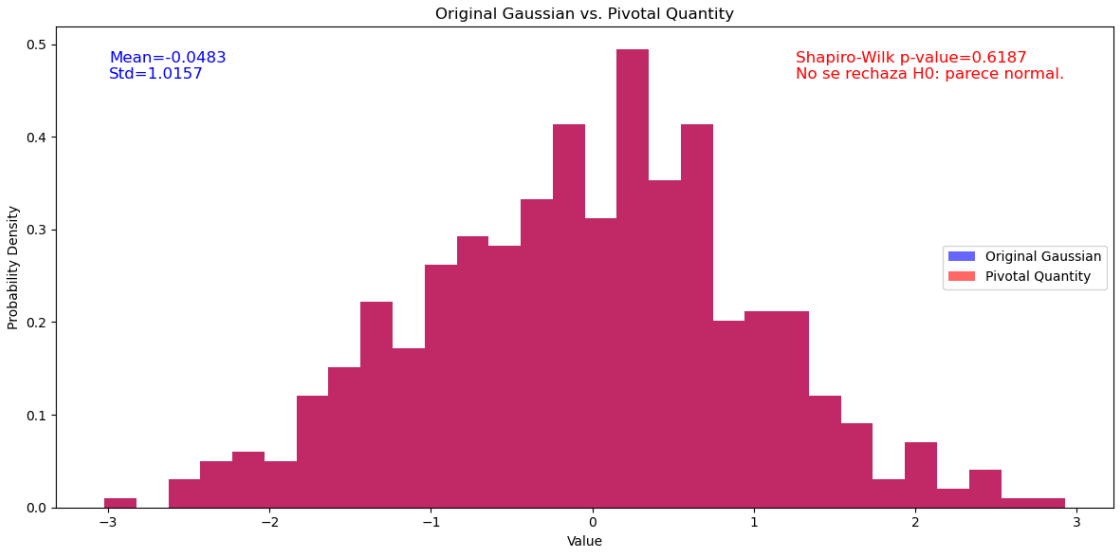
Ejercicio: Utilizando la cantidad pivotal anterior, demuestre que su distribución es Gaussiana con $\mu = 0, \sigma = 1$. Use la prueba de Shapiro-Wilk para completar su demostración.

[6]:
```python
def gaussian_pivotal(mean, std_dev, num_samples):
    """
    Genera muestras gaussianas, calcula la cantidad pivotal estandarizada
    y prueba si sigue una distribución normal usando Shapiro-Wilk.
```

```
    """
    import numpy as np
    import matplotlib.pyplot as plt
    from scipy.stats import shapiro

    # distribuciones gaussianas
    samples = np.random.normal(mean, std_dev, num_samples)

    # calcular media y desviación estándar muestral
    sample_mean = np.mean(samples)
    sample_std_dev = np.std(samples, ddof=1)

    # encontrar la cantidad pivotal (distribucion standard normal)
    # (X - ) /    N(0,1)
    pivotal_quantity = (samples - mean) / std_dev

    # realizar la prueba Shapiro-Wilk para normalidad
    # [Sugerencia: shapiro() ]
    stat, p_value = shapiro(pivotal_quantity)

    # interpretar
    alpha = 0.05  # nivel de significancia
    if p_value > alpha:
        interpretation = "No se rechaza H0: parece normal."
    else:
        interpretation = "Se rechaza H0: no parece normal."

    # graficar histos
    plt.figure(figsize=(12, 6))

    # graficar la distro gaussiana original
    plt.hist(samples, bins=30, density=True, alpha=0.6, color='b',␣
↪label='Original Gaussian')
    plt.text(0.05, 0.95, f'Mean={sample_mean:.4f}\nStd={sample_std_dev:.4f}',
             transform=plt.gca().transAxes, fontsize=12,␣
↪verticalalignment='top', color='blue')

    # graficar la cantidad pivotal
    plt.hist(pivotal_quantity, bins=30, density=True, alpha=0.6, color='r',␣
↪label='Pivotal Quantity')
    plt.text(0.7, 0.95, f'Shapiro-Wilk p-value={p_value:.4f}\n{interpretation}',
             transform=plt.gca().transAxes, fontsize=12,␣
↪verticalalignment='top', color='red')

    plt.title('Original Gaussian vs. Pivotal Quantity')
    plt.xlabel('Value')
    plt.ylabel('Probability Density')
```

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

    # interpretar el valor p en consola
    if p_value > alpha:
        print("La cantidad pivotal sigue aproximadamente una distribución␣
↪normal (no se rechaza H0).")
    else:
        print("La cantidad pivotal no sigue una distribución normal (se rechaza␣
↪H0).")
```

[7]:
```
#uso
gaussian_pivotal(mean=0, std_dev=1, num_samples=500)
```




```
La cantidad pivotal sigue aproximadamente una distribución normal (no se rechaza
H0).
```

1.4 2. Inferencia en el modelo RL

1.4.1 2.1 Intervalos de confianza

Con las distribuciones de nuestros estimadores tenemos la posibilidad de construir intervalos de confianza. Esto se hace habitualmente con cantidades pivotales, en particular con la estadística T.

Ejercicio: Demuestre numéricamente, que las estadísticas T para los estimadores $\hat{\beta}_0, \hat{\beta}_1$ siguen distribuciones t_{n-2}.

Sugerencia: use la(s) funciones de la primera parte de este tutorial y de los tutoriales anteriores

Ejercicio: Encuentre los intervalos de confianza de β_0, β_1

[ ]:
```python
from math import sqrt
from random import random, gauss, seed
from scipy.stats import t as student_t

# ==========================
# Generar datos simulados
# ==========================
seed(123)
n = 100
beta0_real = 2.0
beta1_real = 1.5
sigma = 1.0

x = []
y = []
for i in range(n):
    xi = random()  # uniforme(0,1)
    ei = gauss(0.0, sigma)
    yi = beta0_real + beta1_real*xi + ei
    x.append(xi)
    y.append(yi)

# ==========================
# Calcular estimadores
# ==========================
suma_x = sum(x)
suma_y = sum(y)
xbar = suma_x / n
ybar = suma_y / n

Sxx = 0.0
Sxy = 0.0
for i in range(n):
    dx = x[i] - xbar
    dy = y[i] - ybar
    Sxx += dx*dx
    Sxy += dx*dy

beta1_hat = Sxy / Sxx
beta0_hat = ybar - beta1_hat * xbar

# ==========================
# Calcular residuos y RSS
# ==========================
RSS = 0.0
for i in range(n):
    ei = y[i] - (beta0_hat + beta1_hat*x[i])
```

```
    RSS += ei*ei

df = n - 2
sigma2_hat = RSS / df

# ==========================
# Errores estándar
# ==========================
se_beta1 = sqrt(sigma2_hat / Sxx)
se_beta0 = sqrt(sigma2_hat * (1.0/n + (xbar**2)/Sxx))

# ==========================
# Intervalos de confianza 95%
# ==========================
alpha = 0.05
tcrit = student_t.ppf(1 - alpha/2, df)

ic_beta0 = (beta0_hat - tcrit*se_beta0, beta0_hat + tcrit*se_beta0)
ic_beta1 = (beta1_hat - tcrit*se_beta1, beta1_hat + tcrit*se_beta1)

print("Estimador beta0:", round(beta0_hat,4))
print("IC 95% para beta0:", ic_beta0)
print()
print("Estimador beta1:", round(beta1_hat,4))
print("IC 95% para beta1:", ic_beta1)
```

```
Estimador beta0: 2.0897
IC 95% para beta0: (np.float64(1.7350932453637387),
np.float64(2.444353567171173))

Estimador beta1: 1.443
IC 95% para beta1: (np.float64(0.8355047666141374),
np.float64(2.0505038630995243))
```

1.4.2

1.4.3 2.2 Pruebas de hipótesis

Ejercicios

1. Escriba funciones que hagan pruebas de hipótesis para los betas.
2. Use sus funciones para hacer pruebas de hipótesis para cuando se asume que los betas son igual a cero.
3. Proponga un valor razonable para los betas y haga pruebas de hipótesis
4. Escriba un programa que genere datos lineales de tal manera que tanto el intercepto y la pendiente sean cero.
5. Con los datos del punto anterior, haga pruebas de hipótesis asumiendo que los betas sean cero y distintos de cero.
6. Escriba en markdown toda la teoría vista en clase antes y/o después de cada función que

escriba

[11]:
```
# INSERTE SU CODIGO AQUI
# === Pruebas de hipotesis para beta0 y beta1 en RL simple===
from math import sqrt, inf
from random import random, gauss, seed
from scipy.stats import t as student_t

def generar_datos_lineales(n=100, beta0=0.0, beta1=0.0, sigma=1.0, semilla=123):
    """Genera x ~ U(0,1) y =  0 +  1 x + N(0, ²)."""
    seed(semilla)
    x, y = [], []
    for _ in range(n):
        xi = random()           # uniforme(0,1)
        ei = gauss(0.0, sigma) # normal(0, )
        yi = beta0 + beta1*xi + ei
        x.append(xi); y.append(yi)
    return x, y

def ajustar_rl_simple(x, y):
    """
    Ajusta y =  0 +  1 x + e por MCO y devuelve estimadores, errores estándar y␣
 ↪df.
    Todo se calcula "a mano" con sumatorias.
    """
    n = len(x)

    # medias "a mano"
    sum_x = sum(x); sum_y = sum(y)
    xbar = sum_x / n
    ybar = sum_y / n

    # Sxx y Sxy
    Sxx = 0.0; Sxy = 0.0
    for i in range(n):
        dx = x[i] - xbar
        dy = y[i] - ybar
        Sxx += dx*dx
        Sxy += dx*dy

    # estimadores
    beta1_hat = Sxy / Sxx
    beta0_hat = ybar - beta1_hat * xbar

    # residuos y RSS
    RSS = 0.0
    residuos = []
```

```
    for i in range(n):
        ei = y[i] - (beta0_hat + beta1_hat*x[i])
        residuos.append(ei)
        RSS += ei*ei

    # varianza residual y errores estándar
    df = n - 2
    sigma2_hat = RSS / df
    se_beta1 = sqrt(sigma2_hat / Sxx)
    se_beta0 = sqrt(sigma2_hat * (1.0/n + (xbar*xbar)/Sxx))

    return {
        "beta0_hat": beta0_hat,
        "beta1_hat": beta1_hat,
        "se_beta0": se_beta0,
        "se_beta1": se_beta1,
        "sigma2_hat": sigma2_hat,
        "df": df,
        "xbar": xbar,
        "Sxx": Sxx,
        "residuos": residuos
    }

def prueba_t(beta_hat, beta_null, se, df, alpha=0.05, alternativa="two-sided"):
    """
    Estadístico t y p-valor para H0:  = _null.
    alternativa: 'two-sided' | 'greater' | 'less'
    """
    t_stat = (beta_hat - beta_null) / se

    if alternativa == "two-sided":
        p = 2.0 * (1.0 - student_t.cdf(abs(t_stat), df))
        tcrit = student_t.ppf(1.0 - alpha/2.0, df)
        ci = (beta_hat - tcrit*se, beta_hat + tcrit*se)
    elif alternativa == "greater":
        p = 1.0 - student_t.cdf(t_stat, df)
        tcrit = student_t.ppf(1.0 - alpha, df)
        ci = (beta_hat - tcrit*se, inf)
    else: # 'less'
        p = student_t.cdf(t_stat, df)
        tcrit = student_t.ppf(1.0 - alpha, df)
        ci = (-inf, beta_hat + tcrit*se)

    return {"t": t_stat, "p": p, "ci": ci, "df": df}

def pruebas_para_betas(x, y, alpha=0.05):
    """
```

```
    Ejecuta H0: 0=0 y H0: 1=0 (dos colas) y regresa todo lo necesario.
    """
    fit = ajustar_rl_simple(x, y)
    t0 = prueba_t(fit["beta0_hat"], 0.0, fit["se_beta0"], fit["df"], alpha,␣
 ↪"two-sided")
    t1 = prueba_t(fit["beta1_hat"], 0.0, fit["se_beta1"], fit["df"], alpha,␣
 ↪"two-sided")
    return fit, t0, t1

# ================== Ejemplo mínimo de uso (puntos 2, 3, 4 y 5)␣
 ↪==================

# Caso nulo: 0 = 0, 1 = 0
x0, y0 = generar_datos_lineales(n=120, beta0=0.0, beta1=0.0, sigma=1.0,␣
 ↪semilla=7)
fit0, t0_b0, t0_b1 = pruebas_para_betas(x0, y0, alpha=0.05)

print("Caso nulo ( 0=0, 1=0)")
print(f"  beta0_hat={fit0['beta0_hat']:.4f}, SE={fit0['se_beta0']:.4f},␣
 ↪t={t0_b0['t']:.3f}, p={t0_b0['p']:.3f}, CI95%={t0_b0['ci']}")
print(f"  beta1_hat={fit0['beta1_hat']:.4f}, SE={fit0['se_beta1']:.4f},␣
 ↪t={t0_b1['t']:.3f}, p={t0_b1['p']:.3f}, CI95%={t0_b1['ci']}")
print()

# Valores propuestos "razonables": 0 = 2, 1 = 1.5
x1, y1 = generar_datos_lineales(n=120, beta0=2.0, beta1=1.5, sigma=1.0,␣
 ↪semilla=11)
fit1, t1_b0, t1_b1 = pruebas_para_betas(x1, y1, alpha=0.05)

print("Valores propuestos ( 0=2, 1=1.5)")
print(f"  beta0_hat={fit1['beta0_hat']:.4f}, SE={fit1['se_beta0']:.4f},␣
 ↪t={t1_b0['t']:.3f}, p={t1_b0['p']:.3e}, CI95%={t1_b0['ci']}")
print(f"  beta1_hat={fit1['beta1_hat']:.4f}, SE={fit1['se_beta1']:.4f},␣
 ↪t={t1_b1['t']:.3f}, p={t1_b1['p']:.3e}, CI95%={t1_b1['ci']}")
```

```
Caso nulo ( 0=0, 1=0)
  beta0_hat=-0.0169, SE=0.1790, t=-0.094, p=0.925,
CI95%=(np.float64(-0.3713139295266415), np.float64(0.3375323971468152))
  beta1_hat=0.0296, SE=0.3194, t=0.093, p=0.926,
CI95%=(np.float64(-0.6028126050073404), np.float64(0.662020767244109))

Valores propuestos ( 0=2, 1=1.5)
  beta0_hat=1.7400, SE=0.1729, t=10.064, p=0.000e+00,
CI95%=(np.float64(1.3976550175800075), np.float64(2.0824175322146243))
  beta1_hat=1.6152, SE=0.3016, t=5.355, p=4.279e-07,
CI95%=(np.float64(1.0179123578989497), np.float64(2.212480908476855))
```

1.5 Teoría de pruebas de hipótesis en regresión lineal simple

En el modelo de regresión lineal simple $y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$, con errores independientes de media cero, varianza constante σ^2 y normales, los estimadores de mínimos cuadrados $\hat{\beta}_0$ y $\hat{\beta}_1$ se obtienen con sumas básicas. Primero se calculan las medias $\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$ y $\bar{y} = \frac{\sum_{i=1}^n y_i}{n}$. Luego se forman los términos de dispersión $S_{xx} = \sum_{i=1}^n (x_i - \bar{x})^2$ y $S_{xy} = \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$. Con esto, la pendiente estimada es $\hat{\beta}_1 = \frac{S_{xy}}{S_{xx}}$ y la ordenada al origen $\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$.

La varianza residual se estima como

$$\hat{\sigma}^2 = \frac{\mathrm{RSS}}{n-2}, \qquad \mathrm{RSS} = \sum_{i=1}^n \left(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i\right)^2.$$

A partir de ahí, los errores estándar son

$$\mathrm{se}(\hat{\beta}_1) = \sqrt{\frac{\hat{\sigma}^2}{S_{xx}}}, \qquad \mathrm{se}(\hat{\beta}_0) = \sqrt{\hat{\sigma}^2 \left(\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}}\right)}.$$

Para contrastar $H_0 : \beta_j = \beta_{j,0}$ se usa la cantidad pivotal

$$t = \frac{\hat{\beta}_j - \beta_{j,0}}{\mathrm{se}(\hat{\beta}_j)} \sim t_{n-2}.$$

La decisión se toma vía p-valor contra un nivel α o construyendo intervalos de confianza

$$\hat{\beta}_j \pm t_{1-\alpha/2,\, n-2}\, \mathrm{se}(\hat{\beta}_j).$$

Cuando los parámetros verdaderos son nulos ($\beta_0 = 0,\ \beta_1 = 0$), las estimaciones fluctúan alrededor de cero y los p-valores suelen ser grandes, por lo que normalmente no se rechaza H_0. Si los parámetros son distintos de cero, los estadísticos t aumentan en magnitud y los p-valores se vuelven pequeños, aportando evidencia para rechazar la hipótesis nula y permitiendo, además, intervalos de confianza interpretables para los coeficientes del modelo.