Week 3 LR Stats

October 7, 2024

1 Ejercicios de regresión lineal: M2003B

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

By: Jorge Eduardo Guijarro Márquez | A01563113 Claudio José González Arriaga | A00232276 Cristobal Estrada Salinas | A01174432 Frado García Palacios | A01352112 ## Instrucciones: - Complete las funciones donde vea líneas de código inconclusas - Use comentarios para documentar de manera integral sus funciones - Pruebe sus funciones con distintos parámetros

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

1.1 1. Teorema del límite central

1.1.1 1.1 Distribuciones Gaussianas

Ejercicio: Muestre numéricamente que la combinación lineal de distribuciones Gaussianas es una distribución Gaussiana. Asegúrese de que esta nueva distribución en efecto sea Gaussiana con una prueba de Shapiro-Wilk, es decir encuentre un p-valor a aceptar que es Gaussiana.

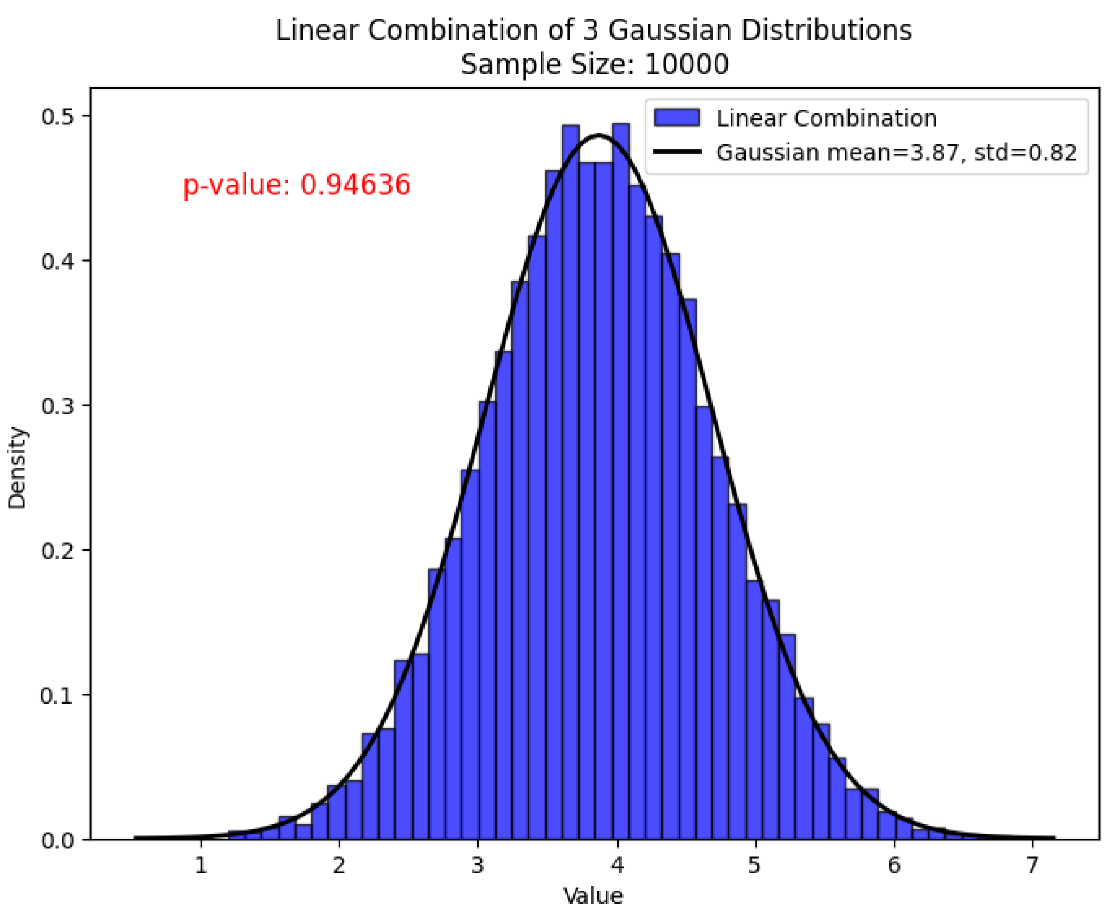
[2]:
```python
def sum_gaussians_tlc(num_samples=10000, n_gaussians=3):
    """



    """
    # Generate independent Gaussian distributions
    means = np.random.uniform(-10, 10, n_gaussians)
    std_devs = np.random.uniform(1, 3, n_gaussians)
    coefficients = np.random.uniform(0, 1, n_gaussians)

    # Initialize an empty list to store the samples
    gaussians = []

    # Loop through each Gaussian distribution
    for i in range(n_gaussians):
        # Generate samples for the current Gaussian distribution
        samples = np.random.normal(means[i], std_devs[i], int(num_samples))
```

```
        # Add the generated samples to the list
        gaussians.append(samples)

    # Linear combination of these Gaussians
    linear_combination = np.zeros(num_samples) # np.concatenate(gaussians)
    for i in range(n_gaussians):
        linear_combination = linear_combination + coefficients[i] *␣
↪gaussians[i] #np.random.normal(means[i], std_devs[i],␣
↪len(linear_combination))

    # Fit a Gaussian (normal) distribution to the linear combination
    mu = np.mean(linear_combination)
    std = np.std(linear_combination)

    # Plot the histogram of the linear combination
    plt.figure(figsize=(8, 6))
    count, bins, _ = plt.hist(linear_combination, bins=50, density=True,␣
↪alpha=0.7, color='blue', edgecolor='black', label='Linear Combination')

    # Plot the Gaussian fit
    xmin, xmax = plt.xlim()  # Get the limits of the x-axis for plotting the␣
↪fit over the same range
    x = np.linspace(xmin, xmax, 100)
    p = norm.pdf(x, mu, std)

    # Scale the fit by the max height of the histogram to make it align
    plt.plot(x, p, 'k', linewidth=2, label=f'Gaussian mean={mu:.2f}, std={std:.
↪2f}')

    # Perform the Shapiro-Wilk test for normality
    stat = shapiro(linear_combination)
    p_value = stat[1]

    # Display the p-value on the plot
    plt.text(xmin + (xmax - xmin) * 0.05, max(count) * 0.9, f'p-value: {p_value:
↪.5f}', fontsize=12, color='red')

    # Add labels, title, and legend
    plt.title(f'Linear Combination of {n_gaussians} Gaussian␣
↪Distributions\nSample Size: {num_samples}')
    plt.xlabel('Value')
    plt.ylabel('Density')
    plt.legend()

    # Show the plot
    plt.show()
```

```python
    # Print the p-value
    if p_value > 0.05:
        print(f" el p-value es mayor al nivel de significancia, por lo tanto,␣
↪no se puede rechazar la hipótesis nula de que los datos provienen de una␣
↪distribución normal.")
    else:
        print(f" el p-value es menor al nivel de significancia, por lo tanto,␣
↪se rechaza la hipótesis nula de que los datos provienen de una distribución␣
↪normal.")
```

[3]:
```python
# Run the demonstration
sum_gaussians_tlc(num_samples=10000, n_gaussians=3)
```

```
c:\Users\jorge\AppData\Local\Programs\Python\Python312\Lib\site-
packages\scipy\stats\_axis_nan_policy.py:531: UserWarning: scipy.stats.shapiro:
For N > 5000, computed p-value may not be accurate. Current N is 10000.
  res = hypotest_fun_out(*samples, **kwds)
```



```
 el p-value es mayor al nivel de significancia, por lo tanto, no se puede
rechazar la hipótesis nula de que los datos provienen de una distribución
normal.
```

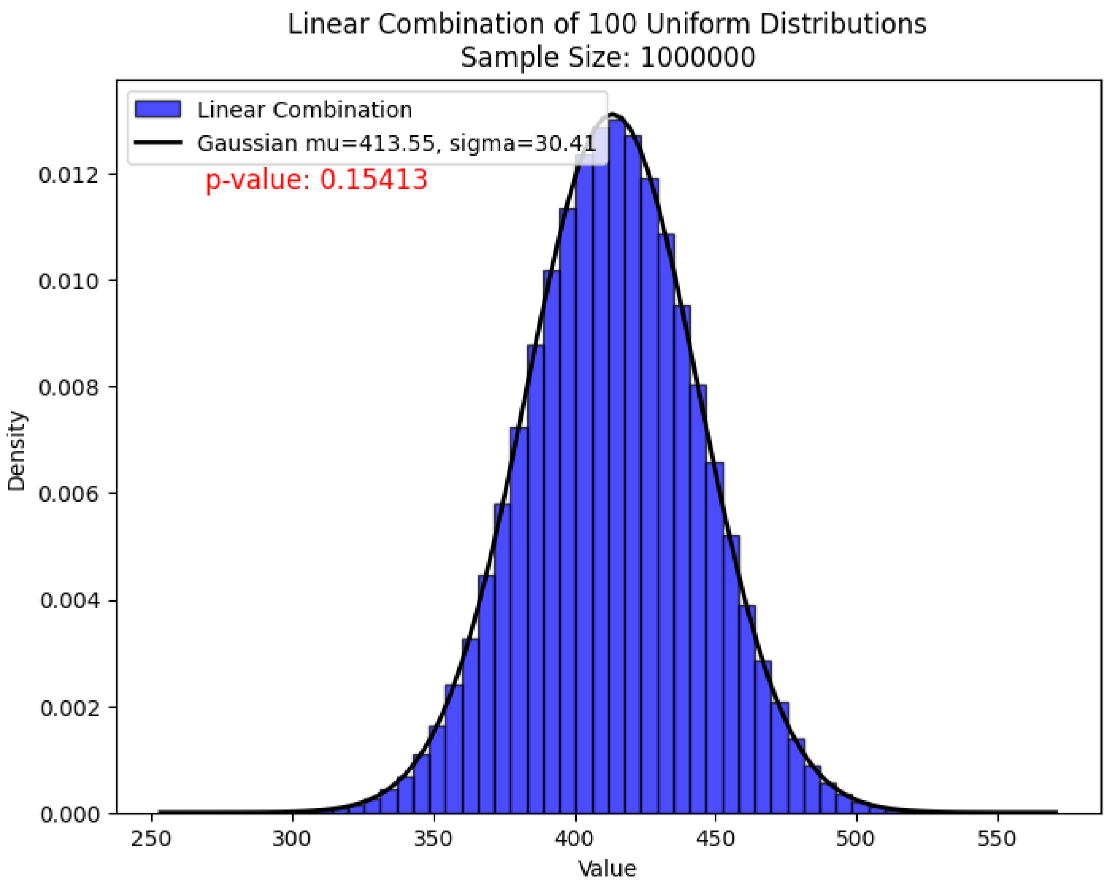
1.1.2 1.2 Distribuciones uniformes

Ejercicio: Repita el ejercicio anterior con distribuciones uniformes finitas. Es decir, el teorema establece que la distribución de la combinación lineal de un gran número de variables aleatorias independientes e idénticamente distribuidas se aproximará a una distribución normal. Sume cuando menos 50 distribuciones.

[4]:
```
def sum_uniforms_tlc(num_samples=1000000, n_uniforms=100):
    """



    """
    # Generate independent uniform distributions
    lower_bounds = np.random.uniform(-10, 10, n_uniforms)
    upper_bounds = np.random.uniform(10, 20, n_uniforms)
    coefficients = np.random.uniform(0, 1, n_uniforms)

    uniforms = []

    # Loop through each uniform distribution
    for i in range(n_uniforms):
        # Generate random samples for the current uniform distribution
        current_samples = np.random.uniform(lower_bounds[i], upper_bounds[i],␣
 ↪int(num_samples))
        # Append the generated samples to the list
        uniforms.append(current_samples)

    # Linear combination of these uniforms
    linear_combination = np.zeros(num_samples)
    for i in range(n_uniforms):
        linear_combination = linear_combination + coefficients[i] * uniforms[i]

    # Fit a Gaussian (normal) distribution to the linear combination
    mu, std = np.mean(linear_combination), np.std(linear_combination)

    # Plot the histogram of the linear combination
    plt.figure(figsize=(8, 6))
    count, bins, _ = plt.hist(linear_combination, bins=50, density=True,␣
 ↪alpha=0.7, color='blue', edgecolor='black', label='Linear Combination')

    # Plot the Gaussian fit
    xmin, xmax = plt.xlim()  # Get the limits of the x-axis for plotting the␣
 ↪fit over the same range
    x = np.linspace(xmin, xmax, 100)
```

```
    p = norm.pdf(x, mu, std)

    # Scale the fit by the max height of the histogram to make it align
    plt.plot(x, p, 'k', linewidth=2, label=f'Gaussian mu={mu:.2f}, sigma={std:.
↪2f}')

    # Perform the Shapiro-Wilk test for normality
    stat = shapiro(linear_combination)
    p_value = stat[1]

    # Display the p-value on the plot
    plt.text(xmin + (xmax - xmin) * 0.05, max(count) * 0.9, f'p-value: {p_value:
↪.5f}', fontsize=12, color='red')

    # Add labels, title, and legend
    plt.title(f'Linear Combination of {n_uniforms} Uniform␣
↪Distributions\nSample Size: {num_samples}')
    plt.xlabel('Value')
    plt.ylabel('Density')
    plt.legend()

    # Show the plot
    plt.show()

    # Print the p-value
    if p_value > 0.05:
        print(f" el p-value es mayor al nivel de significancia, por lo tanto,␣
↪no se puede rechazar la hipótesis nula de que los datos provienen de una␣
↪distribución uniforme.")
    else:
        print(f" el p-value es menor al nivel de significancia, por lo tanto,␣
↪se rechaza la hipótesis nula de que los datos provienen de una distribución␣
↪uniforme.")
```

[6]:
```
# Run the demonstration
sum_uniforms_tlc(num_samples=1000000, n_uniforms=100)
```




```
 el p-value es mayor al nivel de significancia, por lo tanto, no se puede
rechazar la hipótesis nula de que los datos provienen de una distribución
uniforme.
```

1.2

1.3 2. Preliminares estadísticas

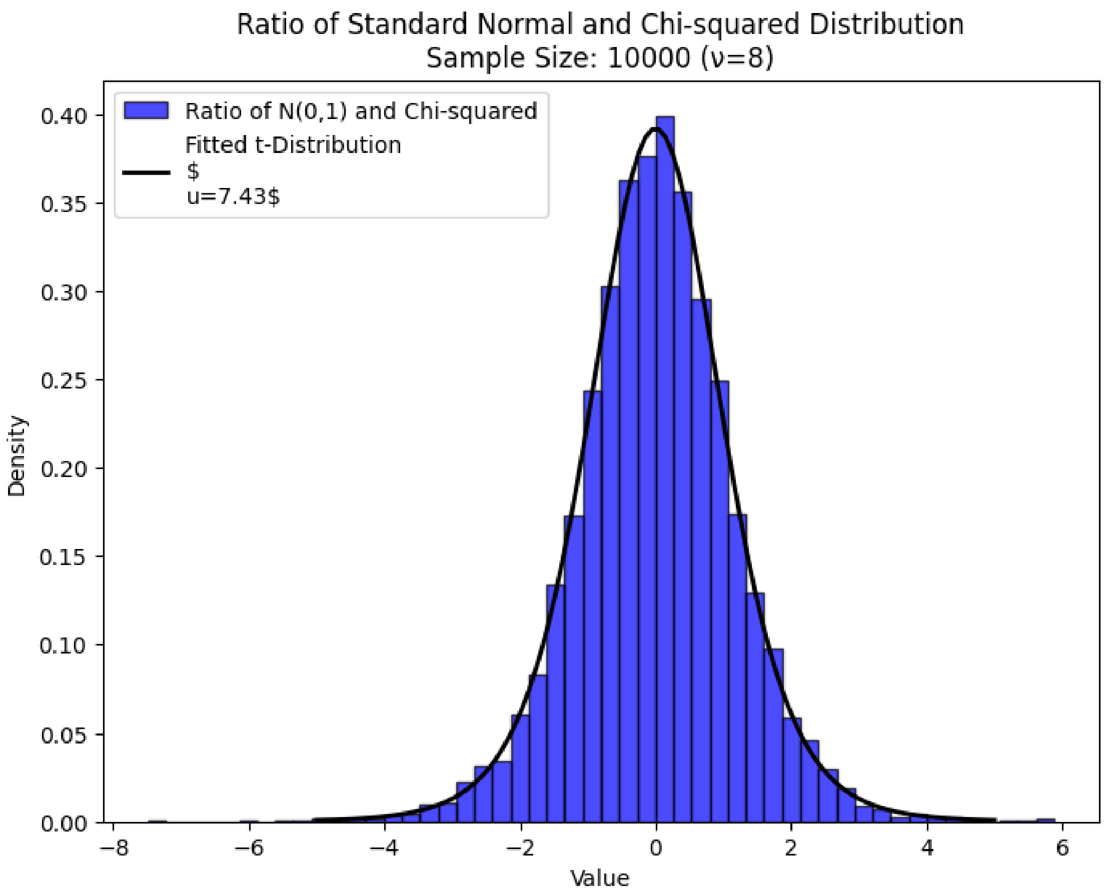
1.3.1 2.1 Distribuciones Student-t

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

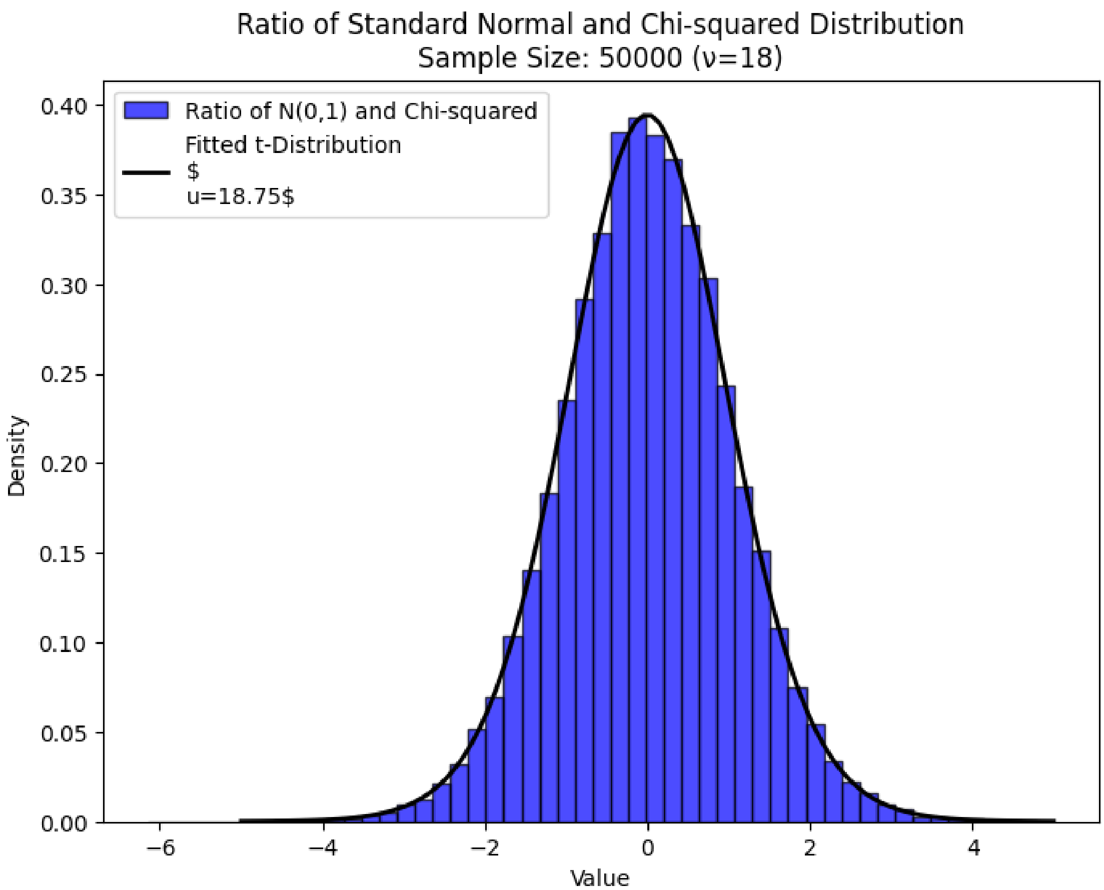
[2]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm, chi2, t, kstest

def student_t_distribution_check(num_samples=10000, n=10):
    """
```

```python
    Verifica numéricamente que la razón de una distribución Gaussiana y una distribución  2
    da como resultado una distribución Student-t.

    Args:
    num_samples (int): Número de muestras a generar.
    n (int): Parámetro para los grados de libertad (n-2) de la distribución  2.

    Returns:
    None: La función muestra un gráfico y imprime los resultados del test.
    """
    # Calculate degrees of freedom for the t-distribution
    nu = n - 2

    # Generate samples from a standard normal distribution
    X = np.random.standard_normal(num_samples)

    # Generate samples from a chi-squared distribution with (n-2) degrees of freedom
    Y = np.random.chisquare(df=nu, size=num_samples)

    # Calculate the Student's t variable
    T = X / np.sqrt(Y / nu)

    # Fit a t-distribution to the sample T
    t_shape, t_loc, t_scale = t.fit(T)

    # Perform the KS test to compare T with a t-distribution
    D, p_value = kstest(T, 't', args=(t_shape, t_loc, t_scale))

    # Plot the histogram of T
    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')

    # Plot the fitted Student's t distribution
    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$\nu={t_shape:.2f}$')

    # Add labels, title, and legend
    plt.title(f'Ratio of Standard Normal and Chi-squared Distribution\nSample Size: {num_samples} ( ={nu})')
    plt.xlabel('Value')
```

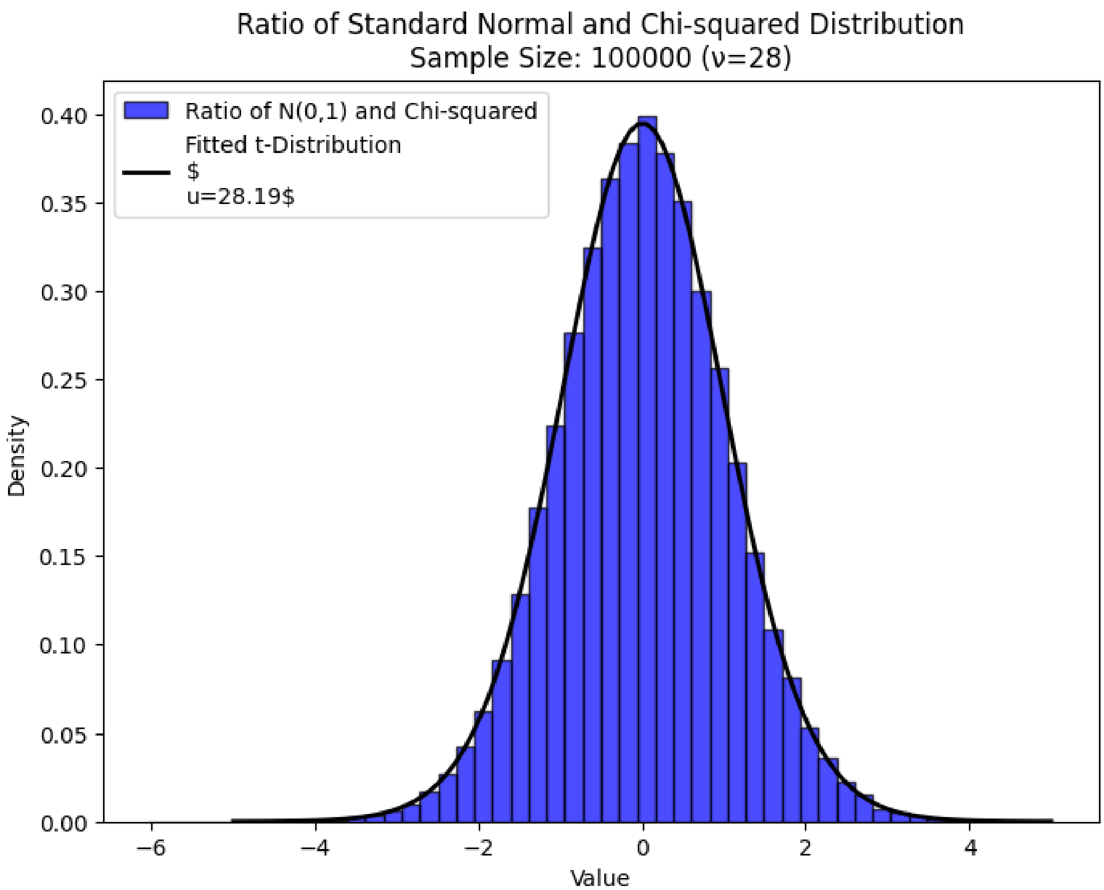
```
    plt.ylabel('Density')
    plt.legend()

    # Show the plot
    plt.show()

    # Print the estimated degrees of freedom from the fitted t-distribution and␣
 ↪p-value
    print(f"Estimated degrees of freedom: {t_shape:.2f}")
    print(f"KS test p-value: {p_value:.5f}")

    # Interpret the p-value
    if p_value > 0.05:
        print("No se puede rechazar la hipótesis nula. La distribución se␣
 ↪ajusta bien a una t de Student.")
    else:
        print("Se rechaza la hipótesis nula. La distribución no se ajusta bien␣
 ↪a una t de Student.")

# función con diferentes parámetros
student_t_distribution_check()
student_t_distribution_check(num_samples=50000, n=20)
student_t_distribution_check(num_samples=100000, n=30)
```




```
Estimated degrees of freedom: 7.43
KS test p-value: 0.99813
No se puede rechazar la hipótesis nula. La distribución se ajusta bien a una t
de Student.
```




```
Estimated degrees of freedom: 18.75
KS test p-value: 0.79109
No se puede rechazar la hipótesis nula. La distribución se ajusta bien a una t
de Student.
```




```
Estimated degrees of freedom: 28.19
KS test p-value: 0.98769
No se puede rechazar la hipótesis nula. La distribución se ajusta bien a una t
de Student.
```

1.3.2

2.1.1 Valores críticos de la estadística-t **Ejercicio**: Escribir una función que obtenga el valor critico de la distribución Student-t para n-2 grados de libertad.

[5]:
```
import scipy.stats as stats

def critical_t_values(alpha, n):
    """
    Obtiene los valores críticos de la distribución t de Student para n - 2␣
 ↪grados de libertad.

    Parámetros:
```

```python
    alpha (float): Nivel de significancia (e.g., 0.05 para un intervalo de␣
↪confianza del 95%).
    n (int): Tamaño de la muestra.

    Retorna:
    tuple: Valores críticos para las colas inferior y superior.
    """
    # Calcula los grados de libertad
    df = n - 2

    # Obtiene los valores críticos de t para una prueba de dos colas
    t_critical = stats.t.ppf((1 - alpha/2, alpha/2), df)

    return t_critical

# Ejemplo de uso de la función
alpha = 0.05
n = 20
critical_values = critical_t_values(alpha, n)
print(f"Para alpha = {alpha} y n = {n}:")
print(f"Valor crítico inferior: {critical_values[0]:.4f}")
print(f"Valor crítico superior: {critical_values[1]:.4f}")

# Probamos con diferentes valores
alphas = [ 0.05, 0.1]
ns = [5,10, 30]

for alpha in alphas:
    for n in ns:
        critical_values = critical_t_values(alpha, n)
        print(f"\nPara alpha = {alpha} y n = {n}:")
        print(f"Valor crítico inferior: {critical_values[0]:.4f}")
        print(f"Valor crítico superior: {critical_values[1]:.4f}")
```

```
Para alpha = 0.05 y n = 20:
Valor crítico inferior: 2.1009
Valor crítico superior: -2.1009

Para alpha = 0.05 y n = 5:
Valor crítico inferior: 3.1824
Valor crítico superior: -3.1824

Para alpha = 0.05 y n = 10:
Valor crítico inferior: 2.3060
Valor crítico superior: -2.3060

Para alpha = 0.05 y n = 30:
Valor crítico inferior: 2.0484
```

```
Valor crítico superior: -2.0484

Para alpha = 0.1 y n = 5:
Valor crítico inferior: 2.3534
Valor crítico superior: -2.3534

Para alpha = 0.1 y n = 10:
Valor crítico inferior: 1.8595
Valor crítico superior: -1.8595

Para alpha = 0.1 y n = 30:
Valor crítico inferior: 1.7011
Valor crítico superior: -1.7011
```

1.3.3 Pruebe su función

[129]:
```python
alpha = 0.05
n = 10  # Sample size
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: -2.2281388519649385
```

1.3.4 2.2 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

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

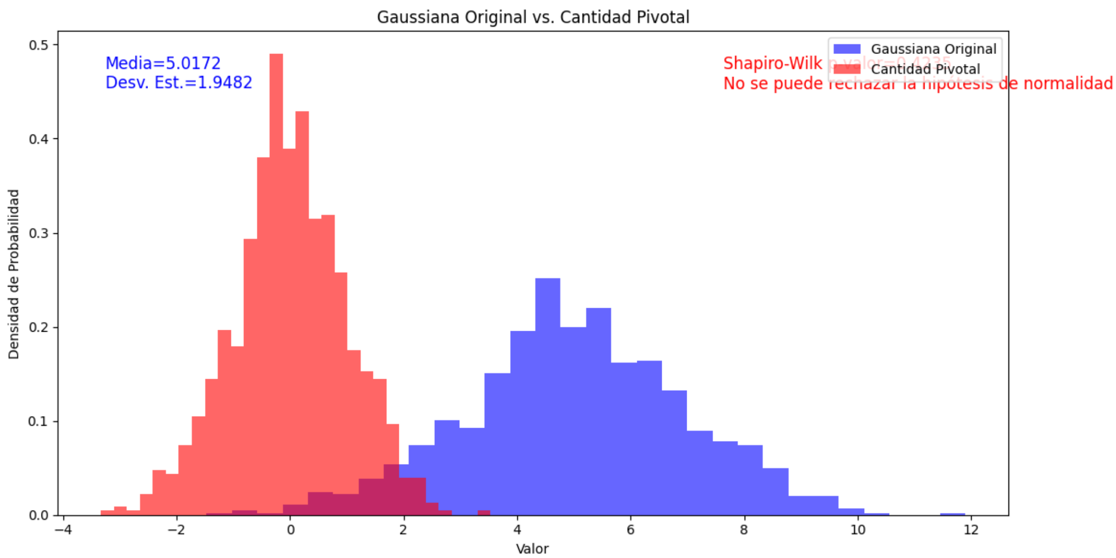
cuya distribución es una normal con $\mu = 0, \sigma = 1$.

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.

[13]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import shapiro

def gaussian_pivotal(mean, std_dev, num_samples):
    """
    Genera muestras de una distribución Gaussiana, las transforma en una␣
↪cantidad pivotal,
    y prueba si la distribución resultante sigue una distribución normal␣
↪estándar.

    Parámetros:
    mean (float): Media de la distribución Gaussiana original.
    std_dev (float): Desviación estándar de la distribución Gaussiana original.
```

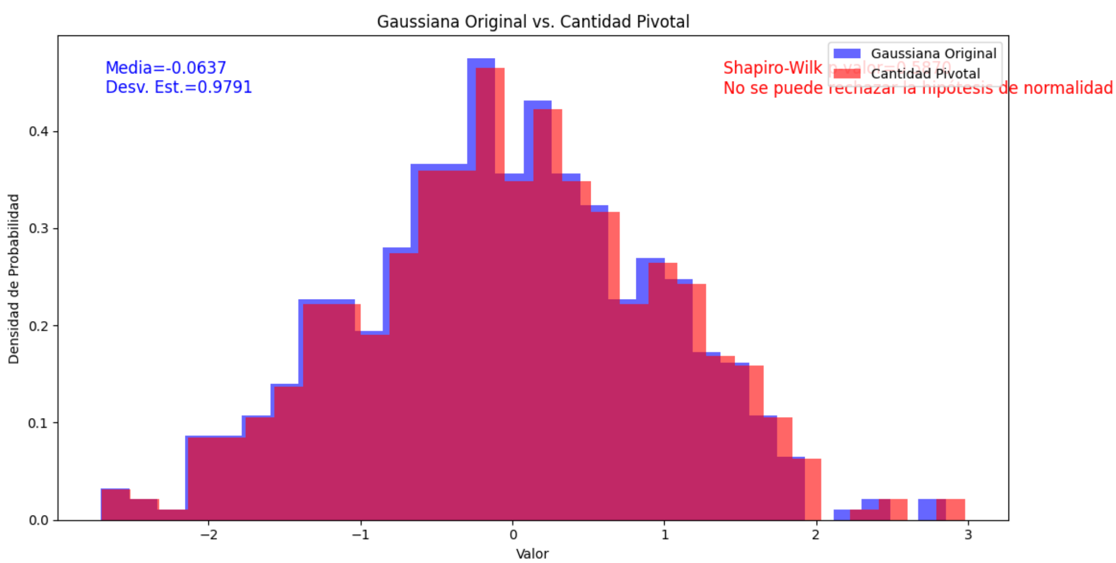
```
    num_samples (int): Número de muestras a generar.

    Retorna:
    None
    """
    # Genera muestras de la distribución Gaussiana
    samples = np.random.normal(mean, std_dev, num_samples)

    # Calcula la media y la desviación estándar de la muestra
    sample_mean = np.mean(samples)
    sample_std_dev = np.std(samples, ddof=1)  # ddof=1 para usar n-1 en el␣
↪denominador

    # Transforma en una cantidad pivotal (distribución normal estándar)
    pivotal_quantity = (samples - sample_mean) / sample_std_dev

    # Realiza la prueba de Shapiro-Wilk para normalidad
    stat, p_value = shapiro(pivotal_quantity)

    # Interpreta el valor p
    alpha = 0.05  # Nivel de significancia
    if p_value > alpha:
        interpretation = "No se puede rechazar la hipótesis de normalidad"
    else:
        interpretation = "Se rechaza la hipótesis de normalidad"

    # Grafica ambos histogramas en el mismo lienzo
    plt.figure(figsize=(12, 6))

    # Grafica la distribución Gaussiana original
    plt.hist(samples, bins=30, density=True, alpha=0.6, color='b',␣
↪label='Gaussiana Original')
    plt.text(0.05, 0.95, f'Media={sample_mean:.4f}\nDesv. Est.={sample_std_dev:.
↪4f}',
             transform=plt.gca().transAxes, fontsize=12,␣
↪verticalalignment='top', color='blue')

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

    # Personaliza el gráfico
    plt.title('Gaussiana Original vs. Cantidad Pivotal')
```

```
    plt.xlabel('Valor')
    plt.ylabel('Densidad de Probabilidad')
    plt.legend()
    plt.tight_layout()
    plt.show()

    # Interpreta el valor p
    if p_value > alpha:
        print(f"El valor p ({p_value:.4f}) es mayor que el nivel de␣
↪significancia ({alpha}).")
        print("No se puede rechazar la hipótesis nula.")
        print("Hay evidencia para afirmar que la cantidad pivotal sigue una␣
↪distribución normal estándar.")
    else:
        print(f"El valor p ({p_value:.4f}) es menor que el nivel de␣
↪significancia ({alpha}).")
        print("Se rechaza la hipótesis nula.")
        print("No hay suficiente evidencia para afirmar que la cantidad pivotal␣
↪sigue una distribución normal estándar.")

# Ejemplo de uso de la función
gaussian_pivotal(mean=5, std_dev=2, num_samples=1000)

# Probamos con diferentes parámetros
params = [
    (0, 1, 500),
    (10, 5, 1000),
    (-3, 0.5, 2000)
]

for mean, std_dev, num_samples in params:
    print(f"\nPrueba con media={mean}, desv. est.={std_dev}, n={num_samples}")
    gaussian_pivotal(mean, std_dev, num_samples)
```



El valor p (0.4235) es mayor que el nivel de significancia (0.05).
No se puede rechazar la hipótesis nula.
Hay evidencia para afirmar que la cantidad pivotal sigue una distribución normal estándar.

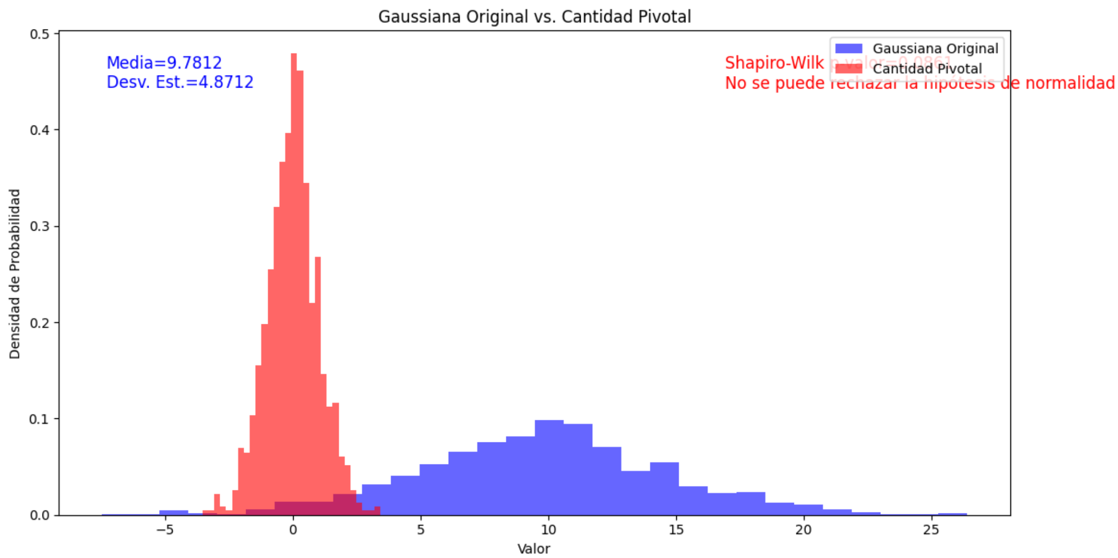
Prueba con media=0, desv. est.=1, n=500



El valor p (0.5870) es mayor que el nivel de significancia (0.05).
No se puede rechazar la hipótesis nula.
Hay evidencia para afirmar que la cantidad pivotal sigue una distribución normal

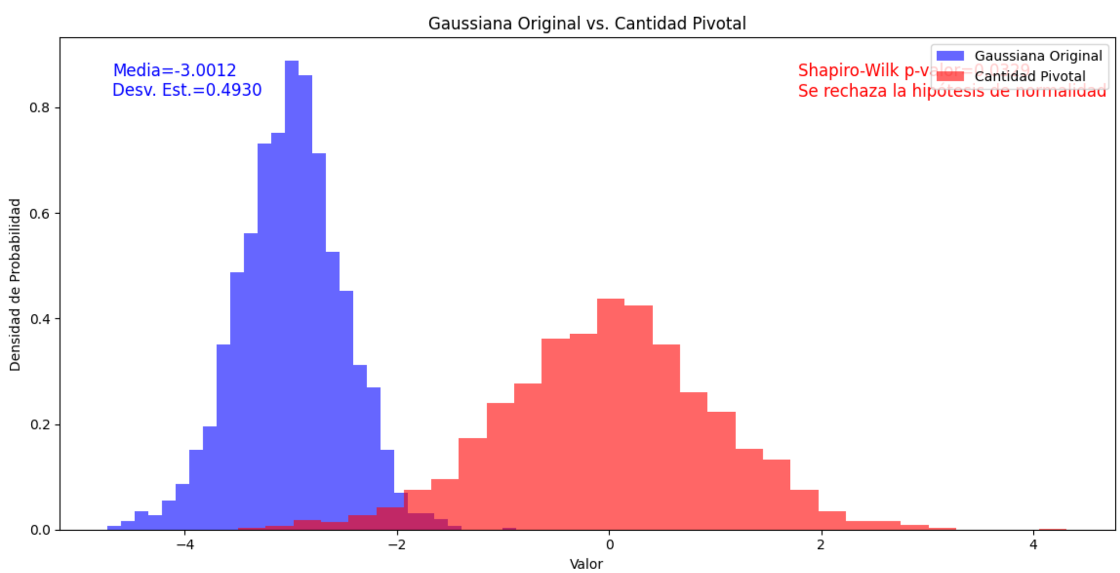
estándar.

Prueba con media=10, desv. est.=5, n=1000



El valor p (0.0861) es mayor que el nivel de significancia (0.05).
No se puede rechazar la hipótesis nula.
Hay evidencia para afirmar que la cantidad pivotal sigue una distribución normal estándar.

Prueba con media=-3, desv. est.=0.5, n=2000



```
El valor p (0.0329) es menor que el nivel de significancia (0.05).
Se rechaza la hipótesis nula.
No hay suficiente evidencia para afirmar que la cantidad pivotal sigue una
distribución normal estándar.
```

1.4

1.5 3. Regresión lineal

1.5.1 3.1 Cargar los datos.

[14]:
```
import pandas as pd
data = pd.read_csv('Week 3 Linear Data.csv')
X = data['X']
Y = data['Y']
```

1.5.2 3.2 Modelo lineal:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

Ejercicio: Escriba una función (o copie y pegue las que ya tiene) para obtener los estimadores de β_0, σ y β_1.

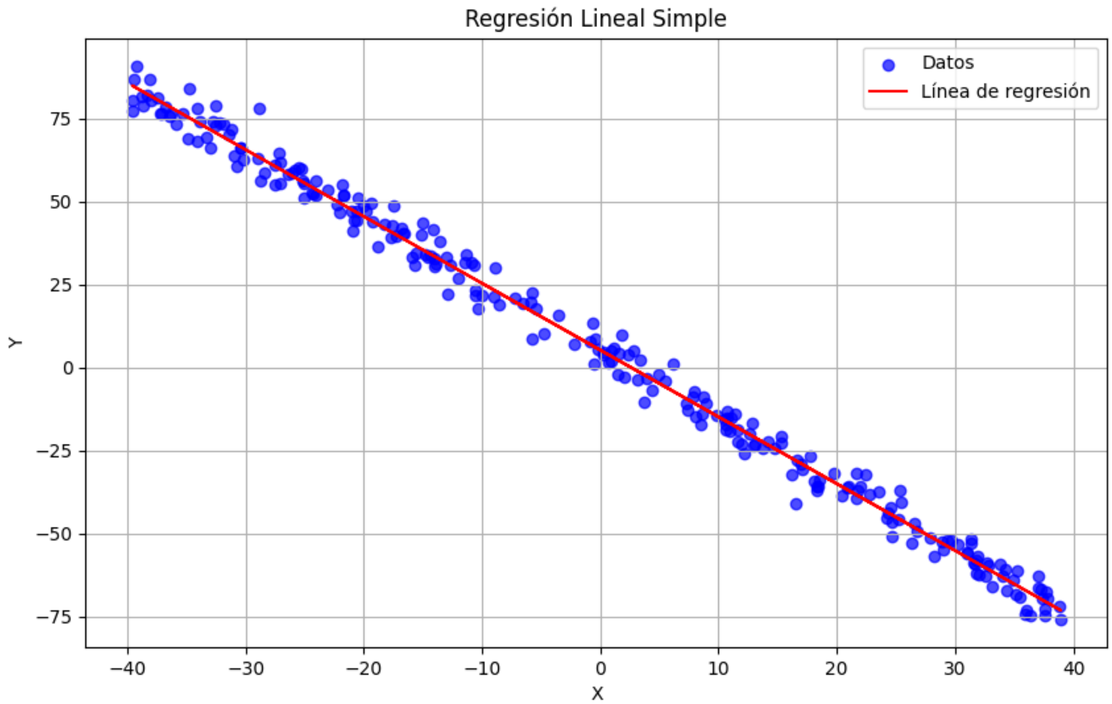
[16]:
```
import numpy as np
import pandas as pd

def simple_linear_regression(x, y):
    """
    Realiza una regresión lineal simple y calcula los estimadores de  ,   y  .

    Parámetros:
    x (array-like): Variable independiente.
    y (array-like): Variable dependiente.

    Retorna:
    tuple: Estimadores de   (intercepto),   (pendiente) y   (error estándar).
    """
    # Convert input lists to numpy arrays for easier computation
    x = np.array(x)
    y = np.array(y)

    # Calculate means of x and y
    x_mean = np.mean(x)
    y_mean = np.mean(y)


    # Calculate the numerator and denominator for beta_1
    numerator = np.sum((x - x_mean) * (y - y_mean))
    denominator = np.sum((x - x_mean)**2)
```

```
    # Calculate beta_1 (slope)
    beta_1_estimador = numerator / denominator

    # Calculate beta_0 (intercept)
    beta_0_estimador = y_mean - beta_1_estimador * x_mean

    # Calculate predicted values
    y_hat = beta_0_estimador + beta_1_estimador * x

    # Calculate residuals
    residuals = y - y_hat

    # Estimate sigma (standard deviation of errors)
    n = len(y)  # Number of observations
    sigma_estimador = np.sqrt(np.sum(residuals**2) / (n - 2))

    return beta_0_estimador, beta_1_estimador, sigma_estimador
```

Pruebe su función

[17]:
```
beta_0_estimador, beta_1_estimador, sigma_estimador =␣
 ↪simple_linear_regression(X, Y)
print(f"Estimated beta_0 (intercept): {beta_0_estimador:.4f}")
print(f"Estimated beta_1 (slope): {beta_1_estimador:.4f}")
print(f"Estimated sigma (error): {sigma_estimador:.4f}")

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.scatter(X, Y, color='blue', alpha=0.7, label='Datos')
plt.plot(X, beta_0_estimador + beta_1_estimador * X, color='red', label='Línea␣
 ↪de regresión')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Regresión Lineal Simple')
plt.legend()
plt.grid(True)
plt.show()
```

```
Estimated beta_0 (intercept): 5.2606
Estimated beta_1 (slope): -2.0116
Estimated sigma (error): 3.9776
```



1.5.3

1.5.4 3.3 Errores para los estimadores

Ejercicio: Escriba una función que calcule los errores de β_0, β_1

[19]:
```
def calculate_standard_errors(x, sigma_hat):
    """
    Calculate the standard errors for   and   in simple linear regression.

    Parameters:
    x (array-like): Independent variable values.
    sigma_hat (float): Estimated standard deviation of the errors.

    Returns:
    tuple: Standard errors for   (intercept) and   (slope).
    """

    # Convert input to numpy array
    x = np.array(x)

    # Calculate mean of x
    x_mean = np.mean(x)

    # Number of observations
```

```
    n = len(x)

    # Calculate sum of squared deviations from the mean
    sum_squared_dev = np.sum((x - x_mean)**2)

    # Calculate standard error for beta_1 (slope)
    se_beta_1 = sigma_hat / np.sqrt(sum_squared_dev)

    # Calculate standard error for beta_0 (intercept)
    se_beta_0 = sigma_hat * np.sqrt(1/n + x_mean**2 / sum_squared_dev)

    return se_beta_0, se_beta_1
```

[21]:
```
# Test the function
se_beta_0, se_beta_1 = calculate_standard_errors(X, sigma_estimador)
print(f"Standard error for beta_0 (intercept): {se_beta_0:.4f}")
print(f"Standard error for beta_1 (slope): {se_beta_1:.4f}")

# Calculate confidence intervals (95%)
import scipy.stats as stats

# Degrees of freedom
df = len(X) - 2

# t-value for 95% confidence interval
t_value = stats.t.ppf(0.975, df)

# Confidence intervals
ci_beta_0 = (beta_0_estimador - t_value * se_beta_0, beta_0_estimador + t_value␣
 ↪* se_beta_0)
ci_beta_1 = (beta_1_estimador - t_value * se_beta_1, beta_1_estimador + t_value␣
 ↪* se_beta_1)

print(f"\n95% Confidence Interval for beta_0: ({ci_beta_0[0]:.5f},␣
 ↪{ci_beta_0[1]:.5f})")
print(f"95% Confidence Interval for beta_1: ({ci_beta_1[0]:.5f}, {ci_beta_1[1]:.
 ↪5f})")
```

```
Standard error for beta_0 (intercept): 0.2517
Standard error for beta_1 (slope): 0.0106

95% Confidence Interval for beta_0: (4.76485, 5.75634)
95% Confidence Interval for beta_1: (-2.03244, -1.99071)
```

1.5.5

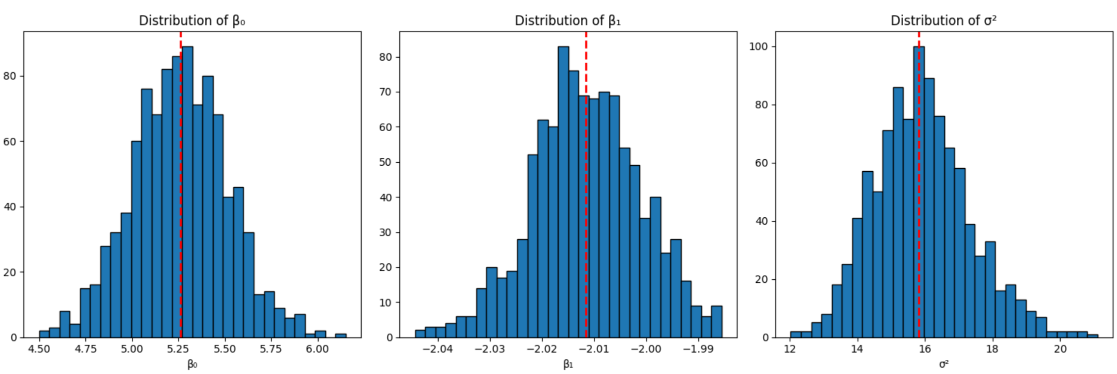
1.5.6 3.4 Distribuciones para los estimadores

Dado que los estimadores son no-cesgados y además por el TLC tienen distribuciones Gaussianas, es posible construir distribuciones para ellos.

Ejercicio: Escriba una función que obtenga dichas distribuciones usando los valores calculados en los ejercicios anteriores.

[22]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

def generate_distributions(beta_0, se_beta_0, beta_1, se_beta_1, sigma_hat, n, num_samples=1000):
    """
    Generate distributions for  ,   estimators and  ² in linear regression.

    Parameters:
    beta_0 (float): Estimated intercept.
    se_beta_0 (float): Standard error of  .
    beta_1 (float): Estimated slope.
    se_beta_1 (float): Standard error of  .
    sigma_hat (float): Estimated standard deviation of errors.
    n (int): Number of observations.
    num_samples (int): Number of samples to generate for each distribution.

    Returns:
    tuple: Samples from distributions for  ,  , and  ².
    """
    # Generate samples for   and   from normal distributions
    beta_0_dist = np.random.normal(beta_0, se_beta_0, num_samples)
    beta_1_dist = np.random.normal(beta_1, se_beta_1, num_samples)

    # Generate samples for  ² from chi-squared distribution
    nu = n - 2  # Degrees of freedom
    chi2_dist = (nu * sigma_hat**2) / stats.chi2.rvs(nu, size=num_samples)

    return beta_0_dist, beta_1_dist, chi2_dist

# Use the function with previously calculated values
beta_0_dist, beta_1_dist, sigma2_dist = generate_distributions(
    beta_0_estimador, se_beta_0, beta_1_estimador, se_beta_1, sigma_estimador, len(X)
)

# Plot the distributions
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))
```

```
ax1.hist(beta_0_dist, bins=30, edgecolor='black')
ax1.set_title('Distribution of  ')
ax1.set_xlabel(' ')
ax1.axvline(beta_0_estimador, color='r', linestyle='dashed', linewidth=2)

ax2.hist(beta_1_dist, bins=30, edgecolor='black')
ax2.set_title('Distribution of  ')
ax2.set_xlabel(' ')
ax2.axvline(beta_1_estimador, color='r', linestyle='dashed', linewidth=2)

ax3.hist(sigma2_dist, bins=30, edgecolor='black')
ax3.set_title('Distribution of  ²')
ax3.set_xlabel(' ²')
ax3.axvline(sigma_estimador**2, color='r', linestyle='dashed', linewidth=2)

plt.tight_layout()
plt.show()

# Print summary statistics
print(f" : Mean = {np.mean(beta_0_dist):.4f}, Std = {np.std(beta_0_dist):.4f}")
```




```
 : Mean = 5.2568, Std = 0.2561
```

1.5.7

1.5.8 3.5 Intetvalos 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 β_0, β_1 siguen distribuciones t_{n-2}. *Sugerencia:* use la(s) funciones de la primera parte de este tutorial.

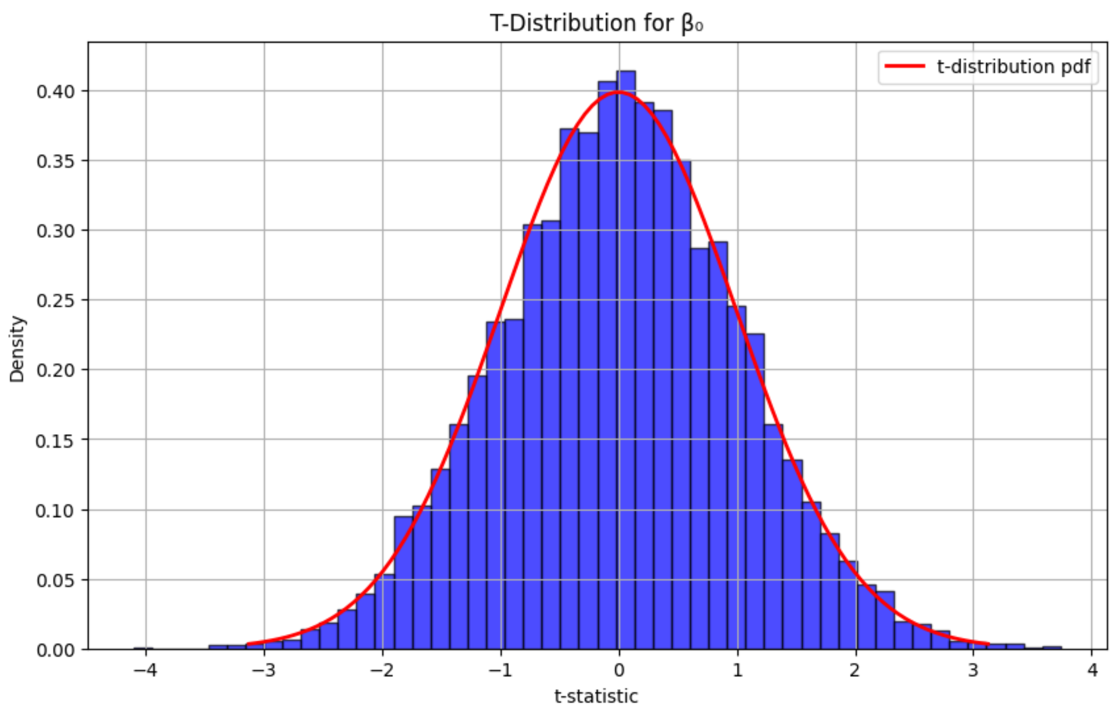
Ejercicio: Encuentre los intervalos de confianza de β_0, β_1

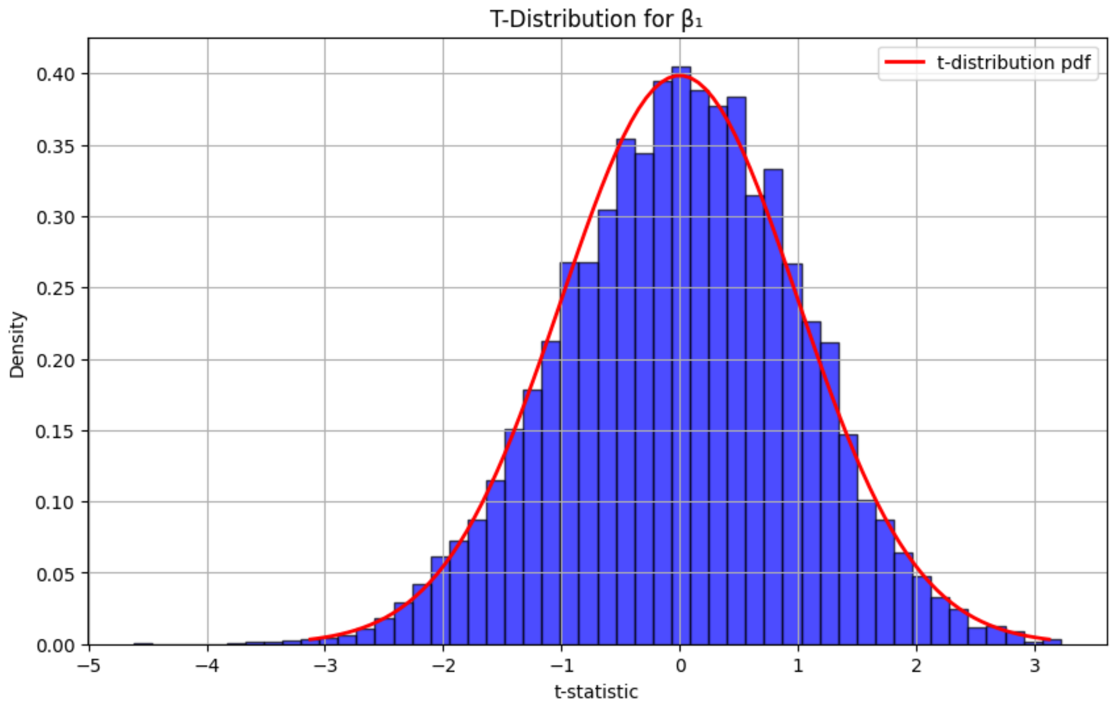
[23]:

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

def calculate_t_statistics(beta, beta_hat, se_beta, n):
    """
    Calculate t-statistic for a regression coefficient.

    Parameters:
    beta (float): True parameter value (used for simulation).
    beta_hat (float): Estimated parameter value.
    se_beta (float): Standard error of the estimate.
    n (int): Sample size.

    Returns:
    float: t-statistic
    """
    return (beta_hat - beta) / se_beta

def simulate_t_statistics(beta, se_beta, n, num_simulations=10000):
    """
    Simulate t-statistics for a regression coefficient.

    Parameters:
    beta (float): True parameter value.
    se_beta (float): Standard error of the estimate.
    n (int): Sample size.
    num_simulations (int): Number of simulations to run.

    Returns:
    numpy.array: Array of simulated t-statistics.
    """
    beta_hats = np.random.normal(beta, se_beta, num_simulations)
    return calculate_t_statistics(beta, beta_hats, se_beta, n)

def plot_t_distribution(t_stats, df, title):
    """
    Plot histogram of t-statistics against theoretical t-distribution.

    Parameters:
    t_stats (numpy.array): Simulated t-statistics.
    df (int): Degrees of freedom for t-distribution.
    title (str): Title for the plot.
    """
    plt.figure(figsize=(10, 6))
    plt.hist(t_stats, bins=50, density=True, alpha=0.7, color='blue',␣
 ↪edgecolor='black')
```

```python
    x = np.linspace(stats.t.ppf(0.001, df), stats.t.ppf(0.999, df), 100)
    plt.plot(x, stats.t.pdf(x, df), 'r-', lw=2, label='t-distribution pdf')

    plt.title(title)
    plt.xlabel('t-statistic')
    plt.ylabel('Density')
    plt.legend()
    plt.grid(True)
    plt.show()

def calculate_confidence_interval(beta_hat, se_beta, n, confidence_level=0.95):
    """
    Calculate confidence interval for a regression coefficient.

    Parameters:
    beta_hat (float): Estimated parameter value.
    se_beta (float): Standard error of the estimate.
    n (int): Sample size.
    confidence_level (float): Confidence level (default: 0.95 for 95% CI).

    Returns:
    tuple: Lower and upper bounds of the confidence interval.
    """
    df = n - 2
    t_value = stats.t.ppf((1 + confidence_level) / 2, df)
    margin_of_error = t_value * se_beta
    return beta_hat - margin_of_error, beta_hat + margin_of_error

# Assuming we have these values from previous exercises
beta_0_hat = beta_0_estimador
beta_1_hat = beta_1_estimador
se_beta_0 = se_beta_0
se_beta_1 = se_beta_1
n = len(X)

# Simulate t-statistics
t_stats_beta_0 = simulate_t_statistics(beta_0_hat, se_beta_0, n)
t_stats_beta_1 = simulate_t_statistics(beta_1_hat, se_beta_1, n)

# Plot t-distributions
plot_t_distribution(t_stats_beta_0, n-2, "T-Distribution for  ")
plot_t_distribution(t_stats_beta_1, n-2, "T-Distribution for  ")

# Calculate confidence intervals
ci_beta_0 = calculate_confidence_interval(beta_0_hat, se_beta_0, n)
ci_beta_1 = calculate_confidence_interval(beta_1_hat, se_beta_1, n)
```

```python
print(f"95% Confidence Interval for  : ({ci_beta_0[0]:.4f}, {ci_beta_0[1]:.4f})")
print(f"95% Confidence Interval for  : ({ci_beta_1[0]:.4f}, {ci_beta_1[1]:.4f})")

# Perform Kolmogorov-Smirnov test
ks_statistic_beta_0, p_value_beta_0 = stats.kstest(t_stats_beta_0, 't', args=(n-2,))
ks_statistic_beta_1, p_value_beta_1 = stats.kstest(t_stats_beta_1, 't', args=(n-2,))

print(f"\nKolmogorov-Smirnov test for  : statistic = {ks_statistic_beta_0:.4f}, p-value = {p_value_beta_0:.4f}")
print(f"Kolmogorov-Smirnov test for  : statistic = {ks_statistic_beta_1:.4f}, p-value = {p_value_beta_1:.4f}")
```






```
95% Confidence Interval for  : (4.7649, 5.7563)
95% Confidence Interval for  : (-2.0324, -1.9907)

Kolmogorov-Smirnov test for  : statistic = 0.0103, p-value = 0.2401
Kolmogorov-Smirnov test for  : statistic = 0.0156, p-value = 0.0156
```

1.5.9

1.5.10 3.6 Pruebas de hipótesis

[24]:
```python
import numpy as np
from scipy import stats

def hypothesis_test(beta_hat, se_beta, n, null_hypothesis=0, alpha=0.05):
    """
    Perform a hypothesis test for a regression coefficient.

    Parameters:
    beta_hat (float): Estimated parameter value.
    se_beta (float): Standard error of the estimate.
    n (int): Sample size.
    null_hypothesis (float): Null hypothesis value (default: 0).
    alpha (float): Significance level (default: 0.05).

    Returns:
```

```
    dict: Dictionary containing test results.
    """
    # Calculate t-statistic
    t_stat = (beta_hat - null_hypothesis) / se_beta

    # Degrees of freedom
    df = n - 2

    # Calculate p-value (two-tailed test)
    p_value = 2 * (1 - stats.t.cdf(abs(t_stat), df))

    # Calculate critical t-value
    t_critical = stats.t.ppf(1 - alpha/2, df)

    # Make decision
    reject_null = abs(t_stat) > t_critical

    return {
        't_statistic': t_stat,
        'p_value': p_value,
        't_critical': t_critical,
        'reject_null': reject_null
    }

# Assuming we have these values from previous exercises
beta_0_hat = beta_0_estimador
beta_1_hat = beta_1_estimador
se_beta_0 = se_beta_0
se_beta_1 = se_beta_1
n = len(X)

# Perform hypothesis tests
test_results_beta_0 = hypothesis_test(beta_0_hat, se_beta_0, n)
test_results_beta_1 = hypothesis_test(beta_1_hat, se_beta_1, n)

# Print results
print("Hypothesis Test for  :")
print(f"t-statistic: {test_results_beta_0['t_statistic']:.4f}")
print(f"p-value: {test_results_beta_0['p_value']:.4f}")
print(f"Critical t-value: {test_results_beta_0['t_critical']:.4f}")
print(f"Reject null hypothesis: {test_results_beta_0['reject_null']}")

print("\nHypothesis Test for  :")
print(f"t-statistic: {test_results_beta_1['t_statistic']:.4f}")
print(f"p-value: {test_results_beta_1['p_value']:.4f}")
print(f"Critical t-value: {test_results_beta_1['t_critical']:.4f}")
print(f"Reject null hypothesis: {test_results_beta_1['reject_null']}")
```

```python
# Calculate R-squared
y_mean = np.mean(Y)
ss_tot = np.sum((Y - y_mean)**2)
y_pred = beta_0_hat + beta_1_hat * X
ss_res = np.sum((Y - y_pred)**2)
r_squared = 1 - (ss_res / ss_tot)

print(f"\nR-squared: {r_squared:.4f}")

# Calculate F-statistic
df_model = 1  # Number of predictors
df_residual = n - 2
ms_model = (ss_tot - ss_res) / df_model
ms_residual = ss_res / df_residual
f_statistic = ms_model / ms_residual

# Calculate p-value for F-test
p_value_f = 1 - stats.f.cdf(f_statistic, df_model, df_residual)

print(f"F-statistic: {f_statistic:.4f}")
print(f"p-value for F-test: {p_value_f:.4f}")
print("File end")
print("File")
```

```
Hypothesis Test for  :
t-statistic: 20.9003
p-value: 0.0000
Critical t-value: 1.9696
Reject null hypothesis: True

Hypothesis Test for  :
t-statistic: -189.8889
p-value: 0.0000
Critical t-value: 1.9696
Reject null hypothesis: True

R-squared: 0.9932
F-statistic: 36057.7861
p-value for F-test: 0.0000
```

[ ]:

```python

```