regression-i-esp

November 8, 2024

1 Análisis de Regresión (I)

En este cuaderno exploraremos cómo el análisis de regresión puede ayudarnos a **entender el comportamiento de los datos**, **predecir valores** (ya sean continuos o dicotómicos) y **identificar los predictores más importantes** en un modelo disperso. Presentaremos distintos tipos de modelos de regresión: regresión lineal simple, regresión lineal múltiple y regresión polinómica. Evaluaremos los resultados tanto cualitativamente, a través de herramientas de visualización de Seaborn, como cuantitativamente, utilizando la biblioteca Scikit-learn y otras herramientas especializadas.

Utilizaremos diferentes conjuntos de datos reales para ilustrar los modelos: * Predicción del precio en un nuevo mercado inmobiliario * Extensión del hielo marino y cambio climático * Conjunto de datos de diabetes de Scikit-learn * Conjunto de datos macroeconómicos de EE. UU. de Longley * Conjunto de datos sobre publicidad

1.0.1 Contenidos del cuaderno:

- Regresión
  - Regresión Lineal Simple
  - Regresión Lineal Múltiple
  - Regresión Polinómica
- OLS (Mínimos Cuadrados Ordinarios)
- Evaluación del ajuste (MSE, R^2)
- Predicción (Scikit-learn)
- Visualización (Seaborn lmplot)

1.1 Cómo hacer predicciones sobre cantidades del mundo real

Algunas preguntas del mundo real que podemos abordar con regresión incluyen: + ¿Cómo cambia el volumen de ventas con los cambios en los precios? ¿Cómo se ve afectado por el clima? + ¿Cómo varía la cantidad de un medicamento absorbido con el peso corporal de un paciente? ¿Depende de la presión arterial? + ¿Cuántos clientes podemos esperar hoy? + ¿A qué hora debo salir para evitar el atasco? + ¿Cuál es la probabilidad de lluvia para los próximos dos lunes? ¿Cuál será la temperatura?

1.1.1 Ejemplo:

Puedes encontrar otro ejemplo interesante aquí.

Antes de comenzar, definimos algunas configuraciones básicas para el cuaderno:

2 Configuración para visualizaciones

[1]:
```
# Importamos la biblioteca para gráficos
import matplotlib.pylab as plt

# Configuramos para que los gráficos se muestren dentro del cuaderno
%matplotlib inline

# Establecemos el tamaño de la fuente en los gráficos
plt.rc('font', size=12)

# Ajustamos el tamaño de las figuras
plt.rc('figure', figsize=(12, 5))

# Importamos Seaborn para crear gráficos atractivos
import seaborn as sns

# Configuramos el estilo de los gráficos
sns.set_style("whitegrid")

# Ajustamos el contexto visual
sns.set_context("notebook", font_scale=1, rc={"lines.linewidth": 2, 'font.
 ↪family': [u'times']})

# Importamos NumPy para realizar cálculos numéricos
import numpy as np

# Importamos pandas para la manipulación de datos
import pandas as pd

# Definimos una semilla para la reproducibilidad de los resultados
seed = 42
```

2.0.1 Notación

x_i es un elemento de un vector, **x** es un vector columna, **x'** es su transpuesta (un vector fila), y X es una matriz.

2.0.2 De los Datos a los Modelos

Todas estas preguntas siguen una estructura común: tratamos de entender cómo una variable de respuesta **y** puede depender de una o más variables independientes $\mathbf{x}_i$ (también conocidas como *covariables*, *predictores* o *regresores*).

El propósito de la regresión es construir un modelo (una fórmula matemática) que nos permita predecir la variable respuesta a partir de las covariables.

3 Modelo de Regresión Lineal

El modelo más simple que podemos considerar es el **modelo lineal**, donde la variable respuesta $\mathbf{y}$ depende linealmente de los predictores $\mathbf{x}_i$:

$$\mathbf{y} = a_1\mathbf{x}_1 + \cdots + a_m\mathbf{x}_m + \epsilon$$

Los a_i son los *parámetros* del modelo o *coeficientes*, y ϵ es el *término de error*, también conocido como *término de perturbación* o *ruido* (en contraste con la "señal" proporcionada por los predictores). Este término captura los factores no incluidos en el modelo que afectan a la variable dependiente $\mathbf{y}$.

Esta ecuación puede expresarse de manera más compacta (en forma matricial) como:

$$\mathbf{y} = X\mathbf{w} + \epsilon$$

donde:

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, X = \begin{pmatrix} x_{11} \ldots x_{1m} \\ x_{21} \ldots x_{2m} \\ \vdots \\ x_{n1} \ldots x_{nm} \end{pmatrix}, \mathbf{w} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}, \epsilon = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_m \end{pmatrix}$$

La regresión lineal es la técnica utilizada para construir estos modelos lineales.

3.0.1 Regresión Lineal Simple

En la **regresión lineal simple**, trabajamos con un solo predictor y describimos la relación entre este predictor y la respuesta mediante una línea recta.

El modelo es:

$$\mathbf{y} = a_0 + a_1\mathbf{x}_1 + \epsilon$$

El parámetro a_0 es conocido como *término constante* o *intercepto*.

En la forma matricial, se agrega un término constante utilizando la matriz $(\mathbf{1}, X)$.

Ejemplo: ¿Depende el precio del seguro de la experiencia al conducir?

Dado el conjunto de datos con los precios mensuales de seguros de automóviles ($\mathbf{y}$) y los años de experiencia al conducir ($\mathbf{x}_1$) de un grupo de n=8 personas, podemos construir un modelo lineal para responder a esta pregunta.

También podemos predecir el precio mensual del seguro para un conductor con 20 años de experiencia al volante.

3.0.2 Interpolación vs. Extrapolación

Al hacer predicciones, es importante tener en cuenta la diferencia entre **interpolación** y **extrapolación**. Predecir un valor de y para un valor de x dentro del rango de datos observados (es decir, en el intervalo de valores utilizados para ajustar el modelo) se denomina **interpolación**. Por otro lado, predecir un valor de y para un valor de x fuera de este rango se llama **extrapolación**. Si bien la interpolación es relativamente segura, la extrapolación debe manejarse con cautela, ya que puede llevar a predicciones imprecisas o poco realistas.

3.0.3 Regresión Múltiple

La regresión lineal simple se puede extender a casos con múltiples variables predictoras. En este caso, ajustamos un hiperplano m-dimensional a los m predictores.

$$\mathbf{y} = a_1\mathbf{x}_1 + \cdots + a_m\mathbf{x}_m = X\mathbf{w}$$

3.0.4 Regresión Polinómica

A pesar de su nombre, la regresión lineal también puede usarse para ajustar relaciones no lineales. Esto se debe a que un modelo de regresión lineal es lineal en los parámetros del modelo, pero no necesariamente en los predictores.

Si agregamos transformaciones no lineales de los predictores, como términos cuadráticos o cúbicos, el modelo se vuelve no lineal en los predictores.

$$\mathbf{y} = a_1\phi(\mathbf{x}_1) + \cdots + a_m\phi(\mathbf{x}_m)$$

Un ejemplo común de regresión no lineal es la **regresión polinómica**, que modela la relación entre la variable dependiente y los predictores como un polinomio de orden n. A medida que aumentamos el orden del polinomio, el modelo puede ajustarse a más complejidades en los datos.

Por ejemplo, un modelo cúbico sería:

$$y_i \approx a_0 + a_1x_i + a_2x_i^2 + a_3x_i^3$$

Sin embargo, usar polinomios de orden superior tiene un costo: **mayor complejidad computacional** y **riesgo de sobreajuste**. El sobreajuste ocurre cuando el modelo se ajusta demasiado a los detalles específicos de los datos de entrenamiento, perdiendo capacidad para generalizar a nuevos datos no observados.

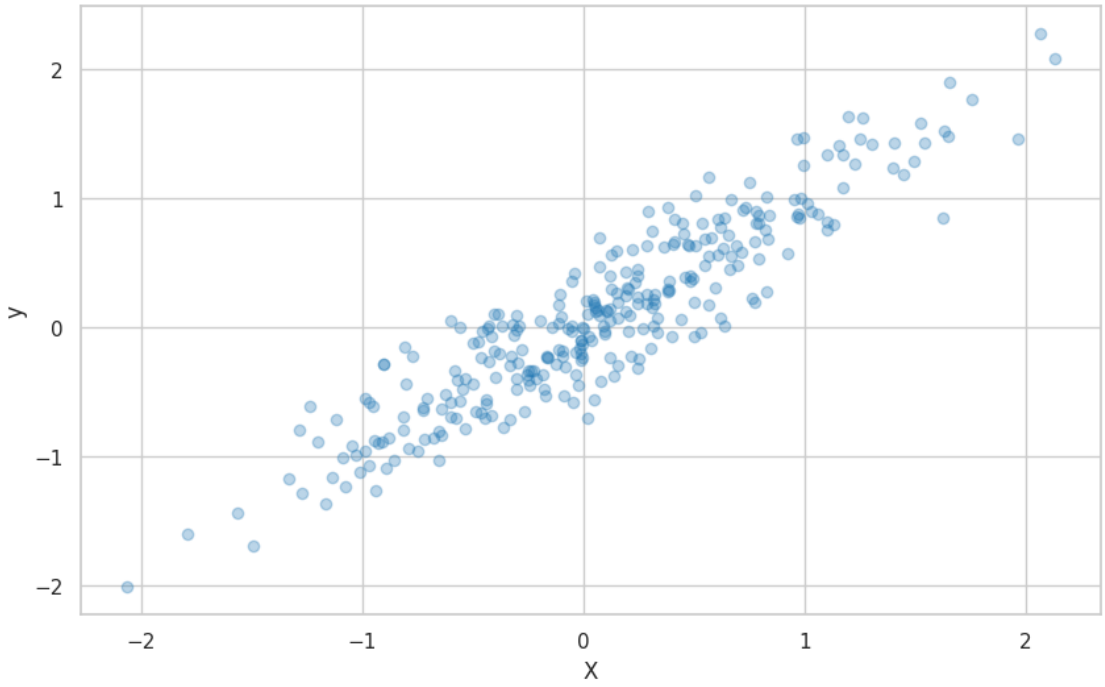
Ejemplo de sobreajuste:

3.1 Estimadores

Generemos un conjunto de datos para ilustrar la regresión lineal simple.

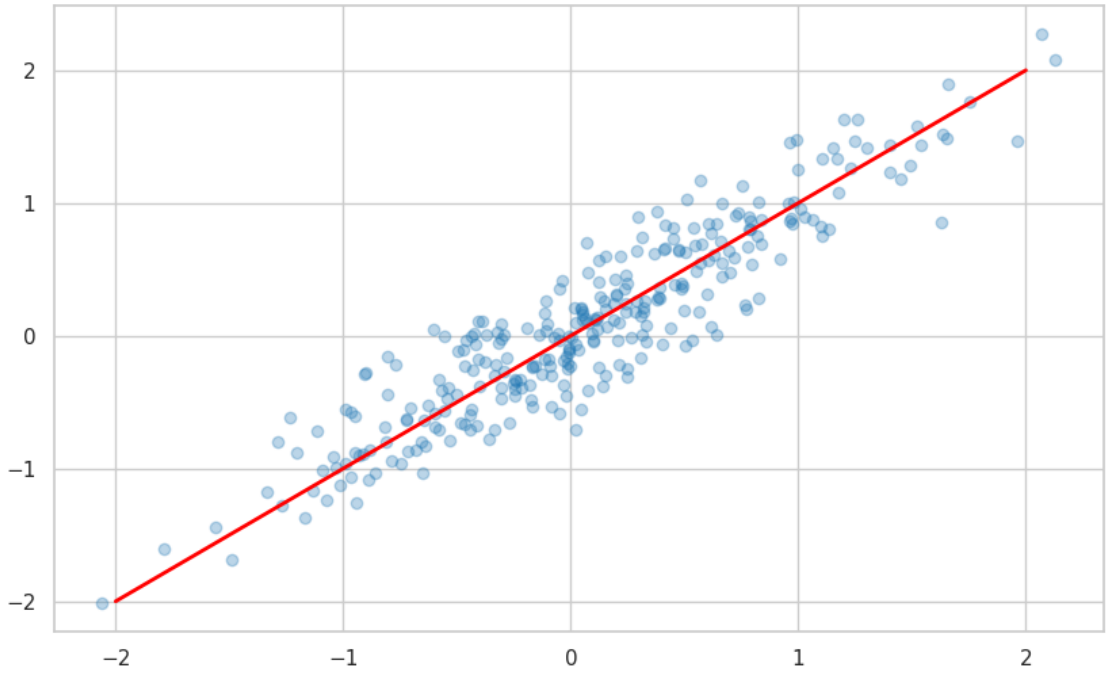
[2]:
```
# Generamos números aleatorios extraídos de una distribución normal (Gaussiana)
X1 = np.random.randn(300, 2)
```

```
# Definimos una matriz de transformación para crear una relación lineal entre␣
 ↪las variables
A = np.array([[0.6, 0.4], [0.4, 0.6]])

# Aplicamos la transformación de la matriz sobre los datos aleatorios para␣
 ↪obtener nuevos datos X2
X2 = np.dot(X1, A)

# Extraemos las dos primeras columnas de X2, donde la primera será la variable␣
 ↪independiente X y la segunda la dependiente y
X = X2[:, 0]  # Variable independiente X
y = X2[:, 1]  # Variable dependiente y

# Establecemos el tamaño de la figura (ancho=10, alto=6)
plt.figure(figsize=(10, 6))

# Graficamos los datos con una transparencia del 30% para visualizar la␣
 ↪dispersión de puntos
plt.plot(X, y, "o", alpha=0.3)

# Agregamos etiquetas a los ejes
plt.xlabel('X')
plt.ylabel('y')

# Mostramos el gráfico
plt.show()
```



A partir de estos datos, podemos construir un modelo lineal simple para intentar predecir los valores de `y` a partir de los valores de `X`.

[3]:
```
# Definimos un modelo lineal simple para ilustrar la relación entre X e y
model = [0 + 1 * x for x in np.arange(-2, 3)]

# Establecemos el tamaño de la figura (ancho=10, alto=6)
plt.figure(figsize=(10, 6))

# Graficamos los puntos de los datos y la línea roja del modelo
plt.plot(X, y, "o", alpha=0.3);
plt.plot(np.arange(-2, 3), model, 'r');
plt.show()
# La línea roja es la predicción del modelo lineal simple sobre los datos
```

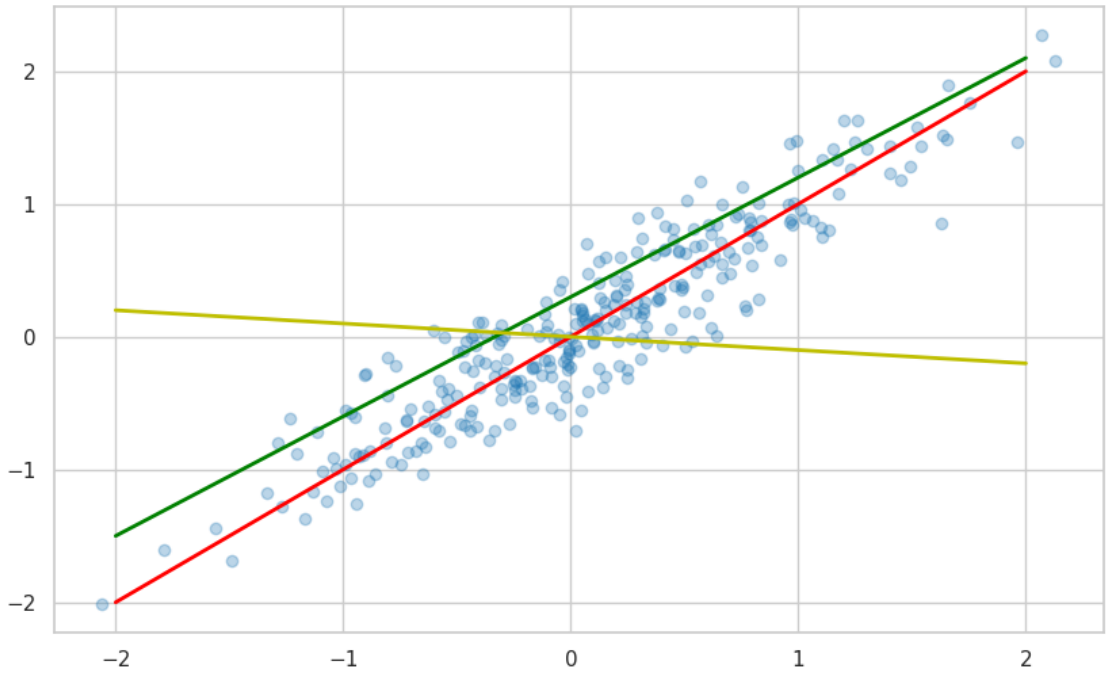


Sin embargo, en la práctica, pueden existir otros modelos lineales que ajusten los datos de distintas maneras.

[4]:
```
# Establecemos el tamaño de la figura (ancho=10, alto=6)
plt.figure(figsize=(10, 6))

# Graficamos varios modelos lineales para comparar
plt.plot(X, y, "o", alpha=0.3);
```

```
# Definimos varios modelos con diferentes coeficientes
model1 = [0 + 1 * x for x in np.arange(-2, 3)]
model2 = [0.3 + 0.9 * x for x in np.arange(-2, 3)]
model3 = [0 - 0.1 * x for x in np.arange(-2, 3)]

# Graficamos las tres líneas de regresión en diferentes colores para compararlas
plt.plot(np.arange(-2, 3), model1, 'r')
plt.plot(np.arange(-2, 3), model2, 'g')
plt.plot(np.arange(-2, 3), model3, 'y')
```

[4]: [<matplotlib.lines.Line2D at 0x7f24131d0cd0>]



¿Cuál es el mejor modelo para un conjunto de muestras? En este punto, nos enfrentamos a la pregunta clave: ¿cómo determinar cuál es el mejor modelo para describir los datos? Existen diferentes métodos para ajustar estos modelos y evaluar cuál tiene el mejor desempeño. Uno de los más comunes es el método de **Mínimos Cuadrados Ordinarios (OLS)**.

3.1.1 MCO (Mínimos Cuadrados Ordinarios)

El modelo de Mínimos Cuadrados Ordinarios (OLS) es una técnica fundamental en la regresión lineal. En términos simples, el OLS busca encontrar una línea que minimice la suma de los errores al cuadrado entre los valores predichos y los valores reales. Este proceso de minimización se lleva a cabo de forma que la suma de las distancias verticales entre los puntos de los datos y la línea de regresión sea la menor posible.

El modelo de regresión lineal se expresa como:

$$\mathbf{y} = a_0 + a_1\mathbf{x}$$

Donde: - $ a_0 $ es la intersección (el valor de $ y $ cuando $ x = 0 $), - $a_1 $ es la pendiente de la línea, que representa cómo cambia $ y $ por cada unidad de cambio en $ x $.

El objetivo del MCO es encontrar los valores de $ a_0 $ y $ a_1 $ que minimicen la **suma de los errores cuadrados** (SSE):

$$||a_0 + a_1\mathbf{x} - \mathbf{y}||_2^2 = \sum_{j=1}^{n}(a_0 + a_1 x_j - y_j)^2$$

Este enfoque busca ajustar una recta de la forma más precisa posible minimizando el error cuadrático.

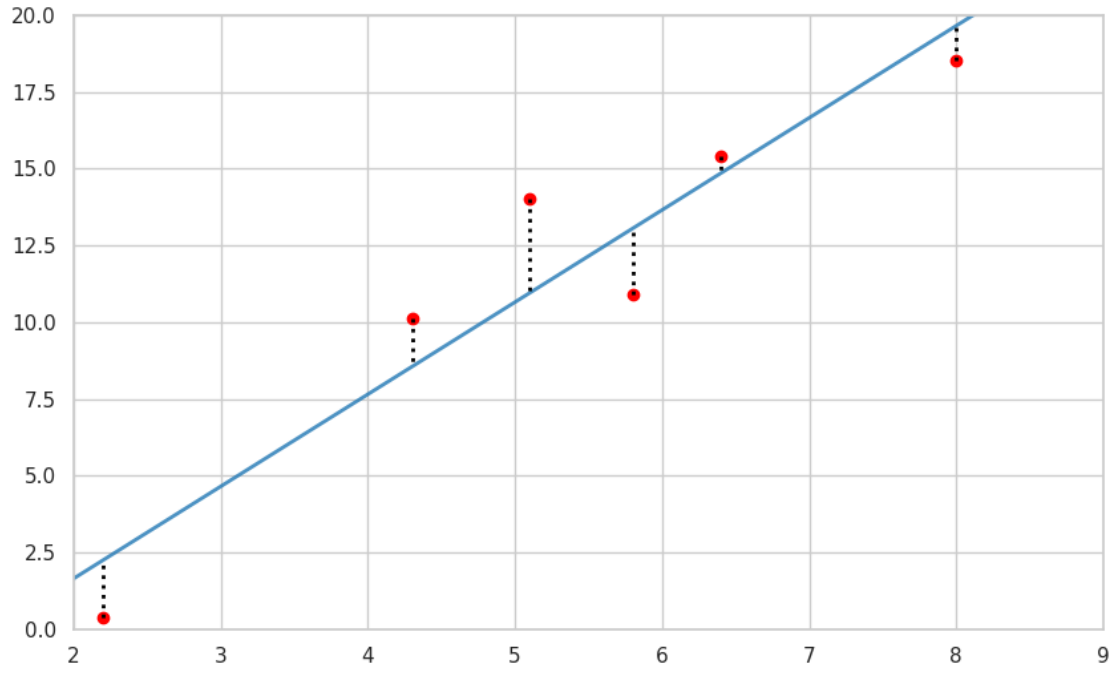
Cómo calcular el MCO: Scipy.optimize Para calcular los parámetros $ a_0 $ y $ a_1 $ utilizando el método de MCO, podemos utilizar la librería `scipy.optimize` que nos permite minimizar la función de error cuadrático.

[5]:
```
# Importamos la función fmin de scipy para la optimización
from scipy.optimize import fmin

# Datos de ejemplo para ajustar el modelo
x = np.array([2.2, 4.3, 5.1, 5.8, 6.4, 8.0])
y = np.array([0.4, 10.1, 14.0, 10.9, 15.4, 18.5])

# Definimos la función lambda para calcular la suma de errores cuadrados
sse = lambda a, x, y: np.sum((a[0] + a[1]*x - y) ** 2)
# Lambda es una función anónima pequeña que toma los coeficientes y los datos
# y calcula la suma de los cuadrados de las diferencias

# Utilizamos la función fmin para minimizar la suma de errores cuadrados
a0, a1 = fmin(sse, [0, 1], args=(x, y));

# Establecemos el tamaño de la figura (ancho=10, alto=6)
plt.figure(figsize=(10, 6))

# Graficamos los puntos de datos originales y la línea de regresión
plt.plot(x, y, 'ro')
plt.plot([0, 10], [a0, a0 + a1 * 10], alpha=0.8) # Línea de regresión en azul
for xi, yi in zip(x, y):
    plt.plot([xi]*2, [yi, a0 + a1 * xi], "k:") # Líneas negras punteadas para␣
 ↪ilustrar los errores
plt.xlim(2, 9); plt.ylim(0, 20) # Restringimos el dominio
```

```
Optimization terminated successfully.
         Current function value: 21.375000
```

```
        Iterations: 79
        Function evaluations: 153
```

[5]: (0.0, 20.0)



En la gráfica, la línea negra punteada muestra los **errores verticales** entre los puntos de datos y la línea de regresión, los cuales se están minimizando. El objetivo del MCO es que estos errores sean lo más pequeños posible.

Nota: Existen alternativas a los modelos de regresión con errores en las variables, como los **mínimos cuadrados totales (TLS)**, que tienen en cuenta errores en ambas variables (tanto en x como en y).

3.1.2 Otros estimadores

Existen otras técnicas para ajustar modelos de regresión. Por ejemplo, podemos minimizar la **suma de los errores absolutos** (SAE), en lugar de la suma de los errores cuadrados. Esto puede ser útil cuando los datos contienen **outliers** o valores atípicos, ya que el SAE no penaliza tan fuertemente a los valores que se encuentran lejos de la línea de ajuste.

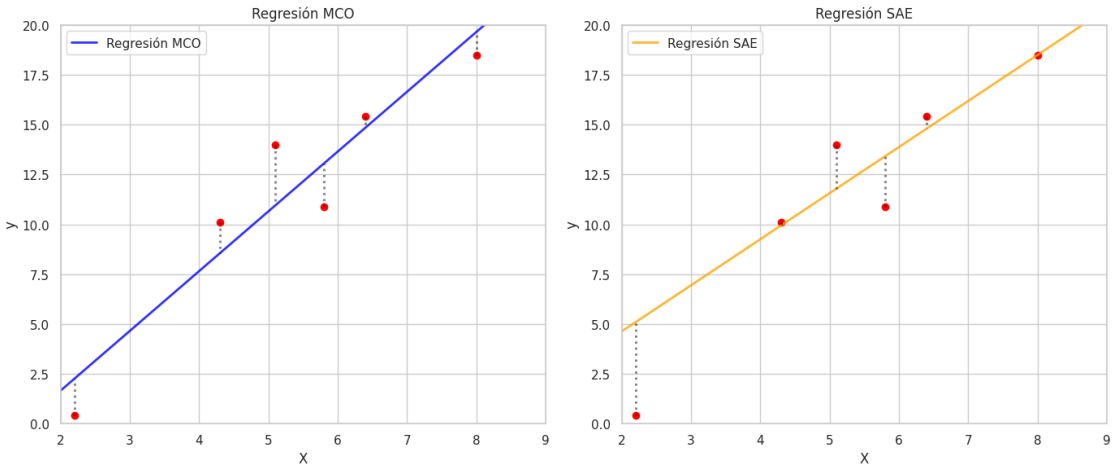
$$\sum_{j=1}^{n} |a_0 + a_1 x_j - y_j|$$

Pregunta Intenta ajustar un modelo lineal utilizando la técnica de SAE y compara los resultados con el modelo obtenido mediante MCO. Observa cómo afectan los valores atípicos a ambos enfoques.

[6]:
```
from scipy.optimize import fmin
import numpy as np
import matplotlib.pyplot as plt

# Datos para ajustar el modelo
x = np.array([2.2, 4.3, 5.1, 5.8, 6.4, 8.0])
y = np.array([0.4, 10.1, 14.0, 10.9, 15.4, 18.5])

# Definimos la función lambda para la suma de errores cuadrados (MCO)
sse = lambda a, x, y: np.sum((a[0] + a[1]*x - y) ** 2)

# Y la función lambda para la suma de errores absolutos (SAE)
sae = lambda a, x, y: np.sum(abs(a[0] + a[1]*x - y))

# Minimizar la suma de errores cuadrados (MCO)
a0, a1 = fmin(sse, [0, 1], args=(x, y))

# Minimizar la suma de errores absolutos (SAE)
b0, b1 = fmin(sae, [0, 1], args=(x, y))

# Crear la figura con 2 subgráficas
fig, axs = plt.subplots(1, 2, figsize=(14, 6))  # 1 fila, 2 columnas

# --- Primer gráfico: MCO ---
# Graficamos los puntos de los datos en el primer gráfico
axs[0].plot(x, y, 'ro')

# Graficamos la línea de regresión de MCO (en azul)
axs[0].plot([0, 10], [a0, a0 + a1 * 10], alpha=0.8, label='Regresión MCO',␣
 ↪color='blue')

# Agregamos las líneas de error (en negro) para la regresión MCO
for xi, yi in zip(x, y):
    axs[0].plot([xi]*2, [yi, a0 + a1 * xi], "k:", alpha=0.5)  # Líneas de error␣
 ↪para MCO

# Añadimos etiquetas y leyenda en el primer gráfico
axs[0].set_title('Regresión MCO')
axs[0].set_xlim(2, 9)
axs[0].set_ylim(0, 20)
axs[0].set_xlabel('X')
axs[0].set_ylabel('y')
axs[0].legend()

# --- Segundo gráfico: SAE ---
# Graficamos los puntos de los datos en el segundo gráfico
axs[1].plot(x, y, 'ro')
```

```
# Graficamos la línea de regresión de SAE (en naranja)
axs[1].plot([0, 10], [b0, b0 + b1 * 10], alpha=0.8, label='Regresión SAE',␣
 ↪color='orange')

# Agregamos las líneas de error (en negro) para la regresión SAE
for xi, yi in zip(x, y):
    axs[1].plot([xi]*2, [yi, b0 + b1 * xi], "k:", alpha=0.5)  # Líneas de error␣
 ↪para SAE

# Añadimos etiquetas y leyenda en el segundo gráfico
axs[1].set_title('Regresión SAE')
axs[1].set_xlim(2, 9)
axs[1].set_ylim(0, 20)
axs[1].set_xlabel('X')
axs[1].set_ylabel('y')
axs[1].legend()

# Mostrar la figura con ambas subgráficas
plt.tight_layout()  # Ajusta el espacio entre subgráficas
plt.show()
```

```
Optimization terminated successfully.
         Current function value: 21.375000
         Iterations: 79
         Function evaluations: 153
Optimization terminated successfully.
         Current function value: 10.162463
         Iterations: 39
         Function evaluations: 77
```



3.1.3 ¿Por qué OLS es tan popular?

OLS es uno de los enfoques más utilizados para la regresión lineal debido a varias razones:

1. **Simplicidad computacional**: Es fácil de calcular y muy eficiente desde el punto de vista computacional.
2. **Interpretabilidad**: Los coeficientes a_0 y a_1 son fáciles de interpretar. En muchas situaciones, se busca entender cómo cambia la variable dependiente (y) a medida que cambia la independiente (x).
3. **Robustez**: OLS es bastante robusto frente a modelos simples, y tiene un buen desempeño incluso cuando hay algo de ruido en los datos.
4. **Fundamentación estadística**: Ofrece buenas propiedades estadísticas, como la estimación insesgada de los parámetros, cuando se cumplen ciertos supuestos (como la homocedasticidad y la independencia de los errores).

En resumen, el modelo de Mínimos Cuadrados Ordinarios es un enfoque poderoso, ampliamente utilizado y fácil de implementar para la regresión lineal.

El modelo ajustado a los datos se representa como:

$$\hat{\mathbf{y}} = \hat{a}_0 + \hat{a}_1 \mathbf{x}$$

Aquí, los sombreros sobre las variables indican que estos son los valores estimados a partir de los datos disponibles.

4 Implementación en Python con Scikit-learn

4.0.1 División de Entrenamiento y Prueba

Uno de los aspectos clave del aprendizaje automático supervisado es la evaluación y validación del modelo. Cuando evaluamos el rendimiento predictivo de un modelo, es esencial que el proceso sea imparcial. No puedes evaluar el rendimiento predictivo de un modelo con los mismos datos que usaste para entrenar. Necesitas evaluar el modelo con datos nuevos que no hayan sido vistos por el modelo antes. Para ello, es necesario dividir el conjunto de datos en dos subconjuntos: uno para entrenamiento y otro para prueba. Este proceso ayuda a obtener una evaluación más precisa de la capacidad del modelo para generalizar a nuevos datos.

Usando la función `train_test_split()` de la biblioteca **Scikit-learn**, puedes dividir tu conjunto de datos en subconjuntos de entrenamiento y prueba de forma eficiente. Esto reduce el potencial de sesgo en el proceso de evaluación y validación del modelo.

Implementación de la división de entrenamiento y prueba Primero, generamos un conjunto de datos de ejemplo usando distribuciones aleatorias:

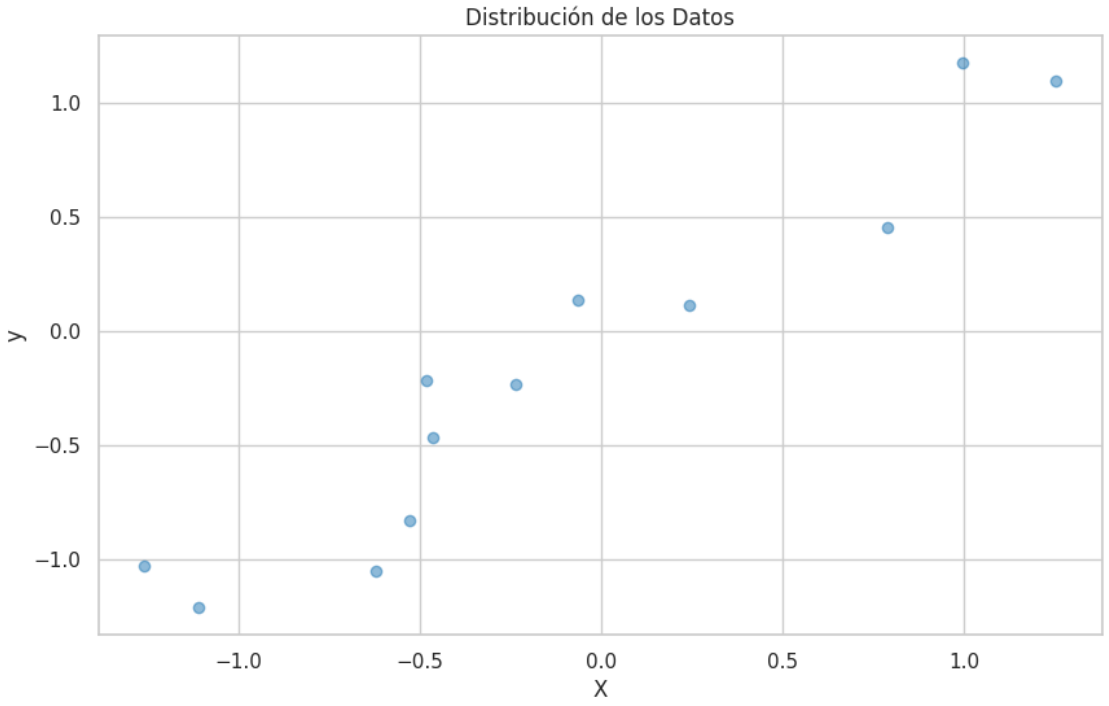
[7]:
```
# Establecemos una semilla para garantizar la reproducibilidad de los resultados
np.random.seed(42)

# Generamos datos aleatorios para simular una relación lineal
X1 = np.random.randn(12, 2)  # Generamos números aleatorios con una␣
 ↪distribución "normal" (Gaussiana)
```

```
# Matriz de transformación para crear una relación lineal entre las variables
A = np.array([[0.6, .4], [.4, 0.6]])

# Aplicamos la transformación para obtener nuevos datos X2
X2 = np.dot(X1, A)  # Producto punto de X1 y la matriz A

# Extraemos las columnas de X2 para asignarlas a X (entradas) y y (salidas)
X = X2[:, 0].reshape(-1, 1)  # X debe ser un arreglo 2D para el modelo
y = X2[:, 1]  # La variable dependiente y

# Mostramos las dimensiones de X y y
print('Tamaño de X e y:', X.shape, y.shape)
```

```
Tamaño de X e y: (12, 1) (12,)
```

Ahora, visualizamos nuestros datos para ver su distribución:

[8]:
```
# Establecemos el tamaño de la figura (ancho=10, alto=6)
plt.figure(figsize=(10, 6))

# Graficamos nuestros datos
plt.plot(X, y, "o", alpha=0.5)  # alpha ajusta la transparencia de los puntos
plt.title("Distribución de los Datos")
plt.xlabel("X")
plt.ylabel("y")
plt.show()
```



Para completar: Ahora dividimos los datos en un conjunto de entrenamiento y un conjunto de prueba, y verificamos los tamaños de los subconjuntos:

[9]:
```
from sklearn.model_selection import train_test_split

# Dividimos los datos en conjunto de entrenamiento y conjunto de prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Verificamos los tamaños de los subconjuntos de entrenamiento y prueba
print("Tamaño de X:", X.shape, "Tamaño de X_train:", X_train.shape, "Tamaño de␣
 ↪X_test:", X_test.shape)
print("Tamaño de y:", y.shape, "Tamaño de y_train:", y_train.shape, "Tamaño de␣
 ↪y_test:", y_test.shape)
```

```
Tamaño de X: (12, 1) Tamaño de X_train: (9, 1) Tamaño de X_test: (3, 1)
Tamaño de y: (12,) Tamaño de y_train: (9,) Tamaño de y_test: (3,)
```

4.1 Predicción

Ahora pasemos a la predicción utilizando **Scikit-learn**. Esta biblioteca proporciona un enfoque simple y eficiente para construir modelos de aprendizaje automático. **Scikit-learn** incluye una variedad de técnicas de aprendizaje supervisado y no supervisado, y cada modelo se implementa a través de una interfaz orientada a objetos que utiliza el concepto de un "estimador".

El método `fit()` de un estimador ajusta el modelo a los datos de entrenamiento. Este método recibe un conjunto de datos de características (X_train) y un conjunto de etiquetas o respuestas (y_train). Los estimadores que pueden generar predicciones proporcionan un método `predict()` para realizar estas predicciones.

Durante el proceso de ajuste, el estado del estimador se almacena en atributos de la instancia con un guion bajo al final (por ejemplo, `coef_`). Para una regresión lineal, los coeficientes del modelo se almacenan en el atributo `coef_`.

A continuación, crearemos un estimador de regresión lineal, lo ajustaremos a los datos de entrenamiento y luego realizaremos predicciones en los datos de prueba.

Para completar: Creamos el estimador de regresión lineal y realizamos el ajuste del modelo:

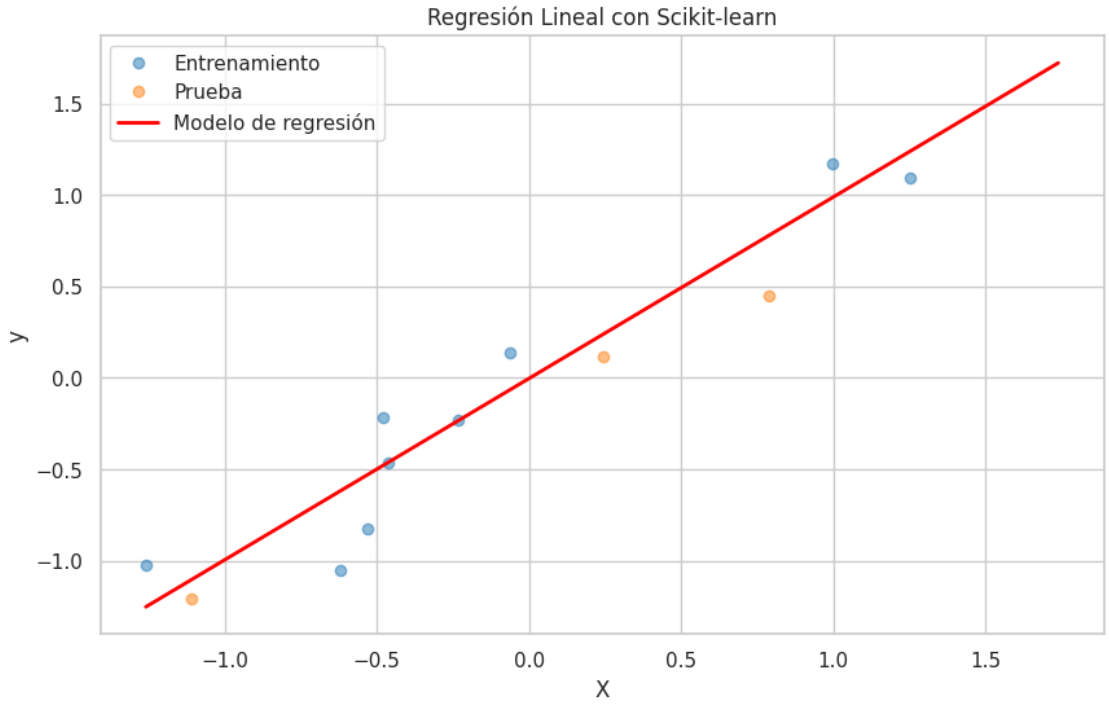
[10]:
```
from sklearn.linear_model import LinearRegression

# Creamos el estimador de regresión lineal
lm = LinearRegression()

# Ajustamos el modelo a los datos de entrenamiento
lm.fit(X_train, y_train)
```

```
# Mostramos el valor del intercepto y los coeficientes de la regresión
print("Intercepto:", lm.intercept_)
print("Coeficientes:", lm.coef_)
```

```
Intercepto: -0.0030997253091730737
Coeficientes: [0.99215814]
```

Ahora, realizamos predicciones en el conjunto de prueba y visualizamos el modelo ajustado:

[11]:
```
# Definimos los valores mínimos y máximos de X para el modelo
xmin, xmax = X.min(), X.max()

# Creamos un rango de valores de X para visualizar la línea de regresión
x_model = np.arange(xmin, xmax+1)
y_model = [lm.intercept_ + lm.coef_ * x for x in x_model]  # Calculamos los␣
 ↪valores predichos para y

# Establecemos el tamaño de la figura (ancho=10, alto=6)
plt.figure(figsize=(10, 6))

# Graficamos los puntos de entrenamiento, prueba y la línea de regresión
plt.plot(X_train, y_train, "o", alpha=0.5, label="Entrenamiento")  # Puntos de␣
 ↪entrenamiento
plt.plot(X_test, y_test, "o", alpha=0.5, label="Prueba")  # Puntos de prueba
plt.plot(x_model, y_model, 'r', label="Modelo de regresión")  # Línea de␣
 ↪regresión

# Añadimos leyenda y etiquetas
plt.legend()
plt.title("Regresión Lineal con Scikit-learn")
plt.xlabel("X")
plt.ylabel("y")
plt.show()
```



4.1.1 Conclusiones

En este ejemplo, hemos implementado un modelo de regresión lineal utilizando **Scikit-learn**, dividiendo los datos en un conjunto de entrenamiento y un conjunto de prueba para evitar sobreajuste (overfitting) y evaluar correctamente el rendimiento del modelo. Además, hemos utilizado el método `fit()` para entrenar el modelo y `predict()` para realizar predicciones.

La visualización de los datos y la línea de regresión obtenida nos permite observar cómo el modelo ajusta la relación lineal entre las variables, proporcionando una herramienta útil para la predicción y el análisis de datos.

Recuerda que este enfoque puede adaptarse a una variedad de otros modelos y problemas, como clasificación o regresión múltiple, y la biblioteca **Scikit-learn** ofrece muchas más opciones para mejorar y optimizar el rendimiento de los modelos.

4.2 Ejemplo 1: Datos de Vivienda en Boston

El conjunto de datos de vivienda de Boston es un conjunto clásico en el aprendizaje automático y contiene registros de mediciones de 13 atributos que describen los mercados de vivienda en distintas áreas de la ciudad de Boston. Además de los atributos, el conjunto incluye la variable objetivo: el precio medio de las viviendas en cada área.

El objetivo en este caso es predecir el **precio de una vivienda** dada una serie de atributos, como la proporción de la población de bajo estatus social, la cantidad de criminalidad, el nivel de contaminación, entre otros.

Vamos a realizar una regresión lineal utilizando uno de los atributos del conjunto de datos para predecir el precio de la vivienda. Para este ejemplo, utilizaremos el atributo **LSTAT**, que representa el porcentaje de la población con estatus bajo.

4.2.1 Cargar el conjunto de datos

Para obtener el conjunto de datos de Boston, usaremos la función `fetch_openml` de Scikit-learn, que descarga el conjunto desde la plataforma OpenML. El conjunto contiene tanto los atributos de las viviendas como el precio medio de las viviendas.

[12]:
```
from sklearn.datasets import fetch_openml

# Cargar el conjunto de datos de Boston desde fetch_openml
boston = fetch_openml(data_id=531)

# Obtener los datos X (características de las viviendas) y el target y (precio␣
 ↪de las viviendas)
X_boston = boston.data
y_boston = boston.target.astype(np.float64)  # Convertimos los precios a tipo␣
 ↪float64 para mayor precisión

# Imprimir la forma de los datos (número de filas y columnas)
print('Shape of data: {} {}'.format(X_boston.shape, y_boston.shape))
```

```
Shape of data: (506, 13) (506,)
```

El conjunto de datos de Boston consta de **506 instancias** (datos) y **13 atributos**. Cada fila representa un barrio o zona de Boston, mientras que cada columna corresponde a un atributo como la tasa de criminalidad, el porcentaje de casas con baja renta, el índice de accesibilidad al transporte, entre otros.

Vamos a obtener más detalles sobre los atributos del conjunto de datos:

[13]:
```
# Ver los atributos clave y la descripción del conjunto de datos
print('keys: {}'.format(boston.keys()))  # Imprime las claves principales del␣
 ↪conjunto de datos
print('feature names: {}'.format(boston.feature_names))  # Imprime los nombres␣
 ↪de los atributos
print(boston.DESCR)  # Imprime una descripción detallada del conjunto de datos
```

```
keys: dict_keys(['data', 'target', 'frame', 'categories', 'feature_names',
'target_names', 'DESCR', 'details', 'url'])
feature names: ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
'TAX', 'PTRATIO', 'B', 'LSTAT']
**Author**:
**Source**: Unknown - Date unknown
**Please cite**:

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
```

```
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
…', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.
Variables in order:
CRIM     per capita crime rate by town
ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
INDUS    proportion of non-retail business acres per town
CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
NOX      nitric oxides concentration (parts per 10 million)
RM       average number of rooms per dwelling
AGE      proportion of owner-occupied units built prior to 1940
DIS      weighted distances to five Boston employment centres
RAD      index of accessibility to radial highways
TAX      full-value property-tax rate per $10,000
PTRATIO  pupil-teacher ratio by town
B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
LSTAT    % lower status of the population
MEDV     Median value of owner-occupied homes in $1000's


Information about the dataset
CLASSTYPE: numeric
CLASSINDEX: last

Downloaded from openml.org.
```

A continuación, vamos a visualizar los datos en un **DataFrame** de pandas para facilitar el análisis y la manipulación de los datos.

[14]:
```
# Crear un DataFrame a partir de los datos de Boston para facilitar su␣
 ↪visualización y manipulación
df_boston = pd.DataFrame(boston.data, columns=boston.feature_names)
df_boston['PRICE'] = boston.target  # Añadir la variable objetivo 'PRICE' al␣
 ↪DataFrame

# Ver las primeras filas del DataFrame
df_boston.head()  # Muestra las primeras filas del conjunto de datos
```

[14]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	\
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	

	B	LSTAT	PRICE
0	396.90	4.98	24.0

```
1  396.90   9.14   21.6
2  392.83   4.03   34.7
3  394.63   2.94   33.4
4  396.90   5.33   36.2
```

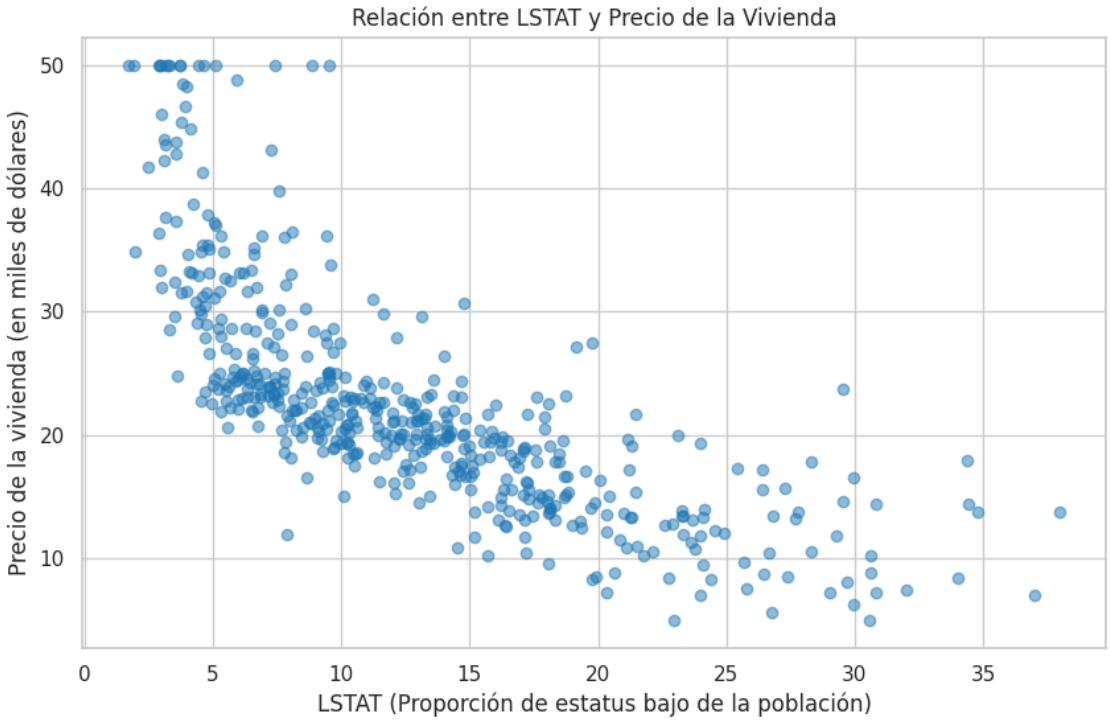
4.2.2 Predicción del Precio de las Viviendas

En este ejemplo, nuestro objetivo es predecir el **precio medio de las viviendas** en el área de Boston utilizando el atributo **LSTAT** como predictor. **LSTAT** representa el porcentaje de la población con un estatus socioeconómico bajo.

Para completar este ejercicio: - Visualizaremos la relación entre **LSTAT** y el precio de las viviendas. - Dividiremos los datos en dos conjuntos: entrenamiento y prueba. - Ajustaremos un modelo de **regresión lineal simple** para predecir el precio en función de **LSTAT**. - Evaluaremos la calidad de nuestro modelo.

Primero, vamos a visualizar los datos y entender la relación entre el atributo **LSTAT** y el precio de las viviendas.

[15]:
```
# Seleccionar la variable LSTAT como predictor (X) y el Precio como target (y)
X = df_boston.LSTAT.values.reshape(-1, 1)  # Convertimos LSTAT a un arreglo 2D␣
 ↪para que sea compatible con el modelo
y = df_boston.PRICE.values  # El precio de las viviendas es la variable objetivo

# Establecemos el tamaño de la figura (ancho=10, alto=6)
plt.figure(figsize=(10, 6))

# Graficar los datos para ver la relación entre LSTAT y el Precio
plt.plot(X, y, "o", alpha=.5)  # Graficamos con puntos, ajustando la␣
 ↪transparencia con alpha
plt.xlabel('LSTAT (Proporción de estatus bajo de la población)')
plt.ylabel('Precio de la vivienda (en miles de dólares)')
plt.title('Relación entre LSTAT y Precio de la Vivienda')
plt.show()
```



La gráfica debe mostrar una relación negativa entre **LSTAT** y el **precio**, lo que indica que a mayor porcentaje de población de bajo estatus socioeconómico, más bajo es el precio de las viviendas en esa área.

4.2.3 Dividir los Datos en Conjuntos de Entrenamiento y Prueba

Ahora, vamos a dividir los datos en dos conjuntos: - **Entrenamiento** (80% de los datos) para ajustar el modelo. - **Prueba** (20% de los datos) para evaluar el rendimiento del modelo.

Utilizamos la función `train_test_split` de Scikit-learn para hacer esta división de manera aleatoria pero reproducible (usando una semilla).

[16]:
```
from sklearn.model_selection import train_test_split

# Dividir los datos en conjunto de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2,␣
 ↪random_state=42)  # 80% entrenamiento, 20% prueba

# Verificar las dimensiones de los conjuntos de entrenamiento y prueba
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(404, 1) (102, 1) (404,) (102,)
```

Esto nos dará los tamaños de los conjuntos de entrenamiento y prueba, lo cual es esencial para asegurarnos de que la evaluación del modelo no esté sesgada.

4.2.4 Ajuste de un Modelo de Regresión Lineal Simple

Ahora, ajustaremos un modelo de regresión lineal simple para predecir el precio de las viviendas usando **LSTAT** como predictor. Esto se puede hacer utilizando la clase `LinearRegression` de Scikit-learn.

[17]:
```
from sklearn.linear_model import LinearRegression

# Crear el estimador de regresión lineal
lm = LinearRegression()

# Ajustar el modelo a los datos de entrenamiento
lm.fit(X_train, y_train)

# Imprimir el intercepto y el coeficiente de la regresión
print('Intercepto:', lm.intercept_)
print('Coeficiente:', lm.coef_)
```

```
Intercepto: 34.83694982031851
Coeficiente: [-0.9665309]
```

El **intercepto** es el valor de la predicción cuando **LSTAT** es igual a cero, y el **coeficiente** muestra cuánto cambia el precio de la vivienda por cada unidad de aumento en **LSTAT**. Esto nos da una idea de la relación entre el atributo predictor y la variable objetivo.

4.2.5 Evaluación del Modelo

Para evaluar el rendimiento del modelo, utilizamos dos métricas clave: - **Coeficiente de determinación (R^2)**: Mide la proporción de la variabilidad de la variable objetivo que se explica por el modelo. Un valor de R^2 cercano a 1 indica un buen ajuste, mientras que un valor cercano a 0 indica que el modelo no explica bien la variabilidad. - **Error cuadrático medio (MSE)**: Mide el promedio de los errores cuadrados entre las predicciones y los valores reales. Un valor más bajo indica un mejor ajuste.

Cálculo del R^2 y el MSE:

[18]:
```
# Calcular el puntaje R^2 para los conjuntos de entrenamiento y prueba
print('Puntaje R^2 en el conjunto de entrenamiento:', lm.score(X_train,␣
 ↪y_train))
print('Puntaje R^2 en el conjunto de prueba:', lm.score(X_test, y_test))
```

```
Puntaje R^2 en el conjunto de entrenamiento: 0.5423180734793516
Puntaje R^2 en el conjunto de prueba: 0.5429180422970384
```

[19]:
```
# Calcular el MSE para los conjuntos de entrenamiento y prueba
y_train_pred = lm.predict(X_train)
y_test_pred = lm.predict(X_test)

mse_train = np.mean((y_train_pred - y_train)**2)
mse_test = np.mean((y_test_pred - y_test)**2)
```

```
print('MSE en el conjunto de entrenamiento:', mse_train)
print('MSE en el conjunto de prueba:', mse_test)
```

```
MSE en el conjunto de entrenamiento: 39.76038682967429
MSE en el conjunto de prueba: 33.51954917268489
```

Estas métricas nos ayudarán a comprender la calidad del ajuste y la capacidad predictiva del modelo.

4.2.6 Evaluación de Todas las Características

Ahora, realizaremos una evaluación más completa utilizando todas las características disponibles en el conjunto de datos. Esto nos permitirá observar cómo cada característica contribuye al rendimiento del modelo de regresión lineal. Para esto, vamos a iterar sobre todas las características y ajustar un modelo de regresión para cada una de ellas por separado.

[20]:
```
from sklearn.linear_model import LinearRegression
import numpy as np

X_boston, y_boston = boston.data, boston.target  # Crear la matriz X y el␣
 ↪vector y del conjunto de datos.
X_train, X_test, y_train, y_test = train_test_split(X_boston, y_boston,␣
 ↪test_size=.2)

# Inicializar el modelo de regresión lineal
# En este caso, utilizamos una regresión lineal simple, que ajusta una línea␣
 ↪recta a los datos.
regr_feat1 = LinearRegression()

# Obtener el número de características (atributos) del conjunto de datos de␣
 ↪Boston
# El conjunto de datos de Boston tiene 13 características, las cuales se␣
 ↪almacenan en boston.data.
n_features = boston.data.shape[1]  # Número de características

# Lista donde guardaremos los resultados de las métricas para cada␣
 ↪característica
scores = []

# Iterar sobre todas las características disponibles en el conjunto de datos de␣
 ↪Boston
for i in range(n_features):
    # Seleccionar la característica i (usamos su nombre para acceder a los␣
 ↪datos)
    feat_name = boston.feature_names[i]

    # Seleccionar los datos de entrenamiento para la característica i (en␣
 ↪formato 2D, ya que el modelo espera este formato)
```

```
    # Reshape(-1, 1) asegura que los datos sean un vector columna, necesario␣
 ↪para la entrada del modelo.
    feat1_train = X_train[[feat_name]].values.reshape(-1, 1)

    # Seleccionar los datos de prueba para la característica i (igual que en el␣
 ↪entrenamiento, se convierte a 2D)
    feat1_test = X_test[[feat_name]].values.reshape(-1, 1)

    # Entrenar el modelo de regresión lineal utilizando solo la característica␣
 ↪seleccionada
    regr_feat1.fit(feat1_train, y_train)

    # Predecir los valores de y (precio de las viviendas) para el conjunto de␣
 ↪entrenamiento y el conjunto de prueba
    y_train_pred = regr_feat1.predict(feat1_train)
    y_test_pred = regr_feat1.predict(feat1_test)

    # Evaluar el rendimiento del modelo en el conjunto de entrenamiento
    # score() devuelve el coeficiente de determinación R^2, que indica qué tan␣
 ↪bien se ajusta el modelo a los datos
    train_score = regr_feat1.score(feat1_train, y_train)

    # Evaluar el rendimiento del modelo en el conjunto de prueba
    test_score = regr_feat1.score(feat1_test, y_test)

    # Calcular el error cuadrático medio (MSE) para el conjunto de entrenamiento
    # El MSE es una medida de cuán cerca están las predicciones de los valores␣
 ↪reales (el valor más bajo es mejor)
    mse_train = np.mean((y_train_pred - y_train)**2)

    # Calcular el MSE para el conjunto de prueba
    mse_test = np.mean((y_test_pred - y_test)**2)

    # Guardar los resultados de las métricas para esta característica
    # Esto nos permite comparar cómo cada característica contribuye al modelo
    scores.append([train_score, test_score, mse_train, mse_test])

# Convertir los resultados a un DataFrame para facilitar la visualización
# Cada fila representará una característica y las métricas asociadas a ella␣
 ↪(train_score, test_score, mse_train, mse_test)
df_scores = pd.DataFrame(scores, columns=["train_score", "test_score",␣
 ↪"train_mse", "test_mse"], index=boston.feature_names)

# Ordenar las características por el puntaje de prueba (test_score) de mayor a␣
 ↪menor para ver cuál tiene el mejor rendimiento
df_scores.sort_values(by="test_score", ascending=False, inplace=True)
```

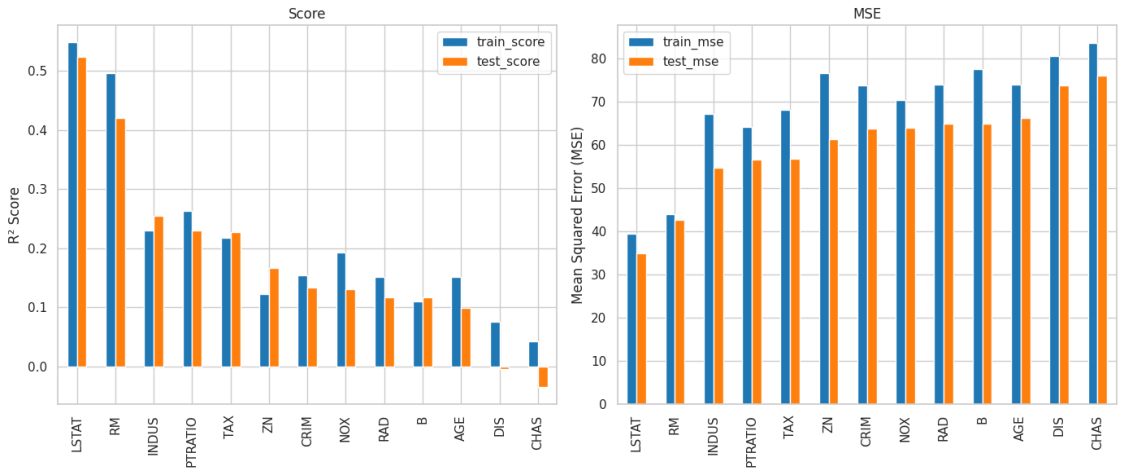
```
# Mostrar el DataFrame con los resultados
df_scores
```

[20]:

	train_score	test_score	train_mse	test_mse
LSTAT	0.548013	0.524136	39.391425	34.971573
RM	0.496501	0.420571	43.880783	42.582636
INDUS	0.229277	0.254556	67.169835	54.783132
PTRATIO	0.263340	0.230517	64.201190	56.549788
TAX	0.217544	0.227134	68.192376	56.798391
ZN	0.121836	0.165841	76.533511	61.302912
CRIM	0.153522	0.133279	73.772043	63.695882
NOX	0.192806	0.130428	70.348377	63.905429
RAD	0.151127	0.117342	73.980788	64.867097
B	0.109640	0.116635	77.596407	64.919031
AGE	0.150680	0.098429	74.019722	66.257008
DIS	0.075126	-0.004088	80.604389	73.791060
CHAS	0.041821	-0.034841	83.507026	76.051141

En este análisis, podemos ver cómo cada característica individual contribuye a la capacidad predictiva del modelo. Además, graficamos los resultados para comparar el desempeño del modelo en los conjuntos de entrenamiento y prueba.

[21]:
```
# Importar la librería de matplotlib para crear las visualizaciones
import matplotlib.pyplot as plt

# Crear la figura con 2 subgráficas (1 fila, 2 columnas)
fig, axs = plt.subplots(1, 2, figsize=(14, 6))  # 1 fila, 2 columnas, tamaño de la figura

# Visualizar los puntajes de entrenamiento y prueba en la primera subgráfica
df_scores[["train_score", "test_score"]].plot(kind="bar", ax=axs[0], title='Score')
axs[0].set_ylabel('R² Score')  # Etiqueta para el eje Y de la primera subgráfica

# Visualizar los MSE de entrenamiento y prueba en la segunda subgráfica
df_scores[["train_mse", "test_mse"]].plot(kind="bar", ax=axs[1], title='MSE')
axs[1].set_ylabel('Mean Squared Error (MSE)')  # Etiqueta para el eje Y de la segunda subgráfica

# Mostrar la figura con ambas subgráficas
plt.tight_layout()  # Ajusta el espaciado entre las subgráficas para que no se sobrepongan
plt.show()  # Muestra la visualización
```



4.2.7 Resumen y Conclusiones

A través de este análisis, hemos: 1. Cargado el conjunto de datos de Boston. 2. Realizado una regresión lineal simple utilizando **LSTAT** como predictor del precio de las viviendas. 3. Evaluado el modelo utilizando las métricas R^2 y **MSE** para asegurarnos de su calidad. 4. Realizado un análisis de todas las características del conjunto de datos para observar el impacto de cada una en el rendimiento del modelo.

Este enfoque puede extenderse fácilmente a regresión múltiple utilizando más de un predictor, lo cual podría mejorar el rendimiento del modelo al capturar más complejidades en los datos.

4.3 Ejemplo 2: Cambio Climático y Extensión del Hielo Marino

Queremos responder a la pregunta: ¿Ha habido una disminución en la cantidad de hielo en los últimos años?

Para ello utilizaremos las mediciones de la extensión del hielo marino del Centro Nacional de Datos de Nieve y Hielo.

Realizaremos los siguientes pasos de procesamiento:

- Leer y limpiar los datos.
- Calcular la tendencia en un intervalo de tiempo dado (mes), normalizando los datos.
- Graficar estos valores para toda la serie temporal o para meses específicos.
- Calcular la tendencia mediante una regresión lineal simple (OLS) y evaluarla cuantitativamente.
- Estimar la extensión del hielo para el año 2025.

4.3.1 1. Leer y limpiar los datos

Cargamos los datos del archivo `SeaIce.txt` usando pandas, mostramos su forma y las primeras filas para inspeccionar. Esto nos permite verificar la estructura del conjunto de datos.

[22]:
```
# Cargar los datos y mostrar la información y el contenido:
import pandas as pd
ice = pd.read_csv('files/ch06/SeaIce.txt', sep='\s+')
print('shape: {}'.format(ice.shape))
ice.head()
```

```
shape: (424, 6)
```

[22]:

	year	mo	data_type	region	extent	area
0	1979	1	Goddard	N	15.54	12.33
1	1980	1	Goddard	N	14.96	11.85
2	1981	1	Goddard	N	15.03	11.82
3	1982	1	Goddard	N	15.26	12.11
4	1983	1	Goddard	N	15.10	11.92

Verificamos los tipos de datos para asegurarnos de que las columnas tienen el formato correcto.

[23]:
```
ice.dtypes
```

[23]:
```
year           int64
mo             int64
data_type     object
region        object
extent       float64
area         float64
dtype: object
```

Descripción de los campos del DataFrame

1. **year**: Año en el que se registró la medición.
2. **mo**: Mes en el que se registró la medición (1 para enero, 2 para febrero, etc.).
3. **data_type**: Tipo de datos registrados, puede indicar la fuente o el método de medición.
4. **region**: Región geográfica donde se realizó la medición del hielo marino.
5. **extent**: Extensión del hielo marino medida en millones de kilómetros cuadrados.
6. **area**: Área del hielo marino medida en millones de kilómetros cuadrados.

Para calcular la anomalía en un intervalo de tiempo dado, primero necesitamos calcular la media para ese intervalo (usando el período de 1981 a 2010 para la extensión media), antes de limpiar los datos.

[24]:
```
# Verificar valores nulos en los datos
ice.isnull().sum()  # Verifica cuántos valores nulos hay en cada columna
```

[24]:
```
year         0
mo           0
data_type    0
region       0
extent       0
area         0
```

```
dtype: int64
```

Limpiamos los datos eliminando las filas que contienen valores nulos.

[25]:
```
# Limpiar los datos eliminando filas con valores nulos
ice_clean = ice.dropna()
```

Seleccionamos solo las columnas numéricas que serán útiles para nuestro modelo. Esto excluye columnas no numéricas como `year`, `mo`, `data_type`, y `region`.

[26]:
```
# Seleccionar solo las columnas numéricas
num_features = ice_clean.select_dtypes(include=[np.number])
num_features.head()
```

[26]:

	year	mo	extent	area
0	1979	1	15.54	12.33
1	1980	1	14.96	11.85
2	1981	1	15.03	11.82
3	1982	1	15.26	12.11
4	1983	1	15.10	11.92

Ahora, dividimos nuestros datos en las características (X) y el objetivo (y). Asumimos que `extent` o `area` es el objetivo, eligiendo uno de ellos.

[27]:
```
# Separar las características y el objetivo
X = num_features.drop(columns=['extent'])  # Si 'extent' es la columna objetivo
y = num_features['extent']
```

[28]:
```
# Dividimos los datos en conjuntos de entrenamiento y prueba. Usamos un 20% de␣
 ↪los datos para prueba.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Inicializar el modelo de regresión lineal
regr_feat1 = LinearRegression()

# Obtener el número de características
n_features = X_train.shape[1]

# Creamos una lista para almacenar los resultados del modelo.

scores = []

# Iteramos sobre todas las características para entrenar y evaluar el modelo␣
 ↪con cada una de ellas.

for i in range(n_features):
    # Seleccionar la característica i
```

```
    feat_name = X_train.columns[i]
    feat1_train = X_train[[feat_name]].values.reshape(-1, 1)  # Asegurar que␣
↪sea una matriz 2D
    feat1_test = X_test[[feat_name]].values.reshape(-1, 1)    # Asegurar que␣
↪sea una matriz 2D

    # Entrenar el modelo
    regr_feat1.fit(feat1_train, y_train)

    # Predecir los valores en los conjuntos de entrenamiento y prueba
    y_train_pred = regr_feat1.predict(feat1_train)
    y_test_pred = regr_feat1.predict(feat1_test)

    # Evaluar el modelo calculando el R² (coeficiente de determinación) y el␣
↪error cuadrático medio (MSE)
    train_score = regr_feat1.score(feat1_train, y_train)  # R² en entrenamiento
    test_score = regr_feat1.score(feat1_test, y_test)    # R² en prueba
    mse_train = np.mean((y_train_pred - y_train)**2)  # MSE en entrenamiento
    mse_test = np.mean((y_test_pred - y_test)**2)    # MSE en prueba

    # Guardar los resultados para cada característica
    scores.append([train_score, test_score, mse_train, mse_test])

# Creamos un DataFrame con los resultados obtenidos para cada característica.


df_scores = pd.DataFrame(scores, columns=["train_score", "test_score",␣
↪"train_mse", "test_mse"], index=X_train.columns)
df_scores.sort_values(by="test_score", ascending=False, inplace=True)
df_scores
```

[28]:

	train_score	test_score	train_mse	test_mse
area	0.999999	1.000000	0.301029	2.858522e-01
year	0.002089	-0.003233	294139.005032	1.168981e+06
mo	0.008054	-0.024666	292380.732226	1.193954e+06

4.3.2 Interpretación de Resultados

La interpretación de los resultados se realiza observando los valores de las columnas `train_score`, `test_score`, `train_mse` y `test_mse` para cada característica. A continuación, se proporciona una interpretación detallada:

Columnas del DataFrame

- **train_score**: Coeficiente de determinación $ R^2 $ en el conjunto de entrenamiento. Indica la proporción de la variabilidad en la variable dependiente que es explicada por la característica en el modelo de entrenamiento.

- **test_score**: Coeficiente de determinación $ R^2 $ en el conjunto de prueba. Indica la proporción de la variabilidad en la variable dependiente que es explicada por la característica en el modelo de prueba.
- **train_mse**: Error cuadrático medio (MSE) en el conjunto de entrenamiento. Representa la media de los errores al cuadrado entre las predicciones del modelo y los valores reales en el conjunto de entrenamiento.
- **test_mse**: Error cuadrático medio (MSE) en el conjunto de prueba. Representa la media de los errores al cuadrado entre las predicciones del modelo y los valores reales en el conjunto de prueba.

Interpretación de las características

1. Área (area)

- **train_score: 0.999999**: El modelo explica prácticamente toda la variabilidad en `extent` cuando se usa `area` en el conjunto de entrenamiento.
- **test_score: 1.000000**: El modelo explica toda la variabilidad en `extent` cuando se usa `area` en el conjunto de prueba. Esto sugiere un ajuste perfecto.
- **train_mse: 0.301029**: El error cuadrático medio en el conjunto de entrenamiento es extremadamente bajo, lo que indica predicciones muy precisas.
- **test_mse: 0.285852**: El error cuadrático medio en el conjunto de prueba también es muy bajo, lo que confirma la precisión del modelo en datos no vistos.

2. Año (year)

- **train_score: 0.002089**: El modelo explica solo una pequeña fracción de la variabilidad en `extent` usando `year` en el conjunto de entrenamiento.
- **test_score: -0.003233**: El valor de $ R^2 $ en el conjunto de prueba es negativo, lo que sugiere que el modelo es incluso peor que una simple línea horizontal promedio.
- **train_mse: 294139.005032**: El error cuadrático medio en el conjunto de entrenamiento es muy alto, lo que indica predicciones imprecisas.
- **test_mse: 1.168981e+06**: El error cuadrático medio en el conjunto de prueba es aún mayor, lo que sugiere que el modelo no generaliza bien.

3. Mes (mo)

- **train_score: 0.008054**: El modelo explica solo una fracción mínima de la variabilidad en `extent` usando `mo` en el conjunto de entrenamiento.
- **test_score: -0.024666**: El valor de $ R^2 $ en el conjunto de prueba es negativo, lo que sugiere que el modelo no tiene capacidad predictiva.
- **train_mse: 292380.732226**: El error cuadrático medio en el conjunto de entrenamiento es alto, indicando que las predicciones son imprecisas.
- **test_mse: 1.193954e+06**: El error cuadrático medio en el conjunto de prueba también es muy alto, lo que indica que el modelo no generaliza bien.

4.3.3 Conclusiones

1. **Area** es la característica más relevante y efectiva para predecir `extent`, ya que tanto el coeficiente de determinación $R^2 $ como los errores cuadráticos medios (MSE) son excelentes

en ambos conjuntos de datos (entrenamiento y prueba).

2. **Year** y **mo** no son características útiles para predecir `extent`. Ambos tienen valores de $ R^2 $ muy bajos (incluso negativos en el conjunto de prueba), lo que sugiere que no explican bien la variabilidad de `extent`. Además, los errores cuadráticos medios son muy altos, lo que implica que las predicciones son muy inexactas.

En futuros modelos predictivos, se debe poner más atención en la característica `area` y menos en `year` y `mo`, o considerar la combinación de varias características para mejorar el rendimiento del modelo.

4.3.4 2. Calcular la media de la extensión del hielo para el periodo 1981-2010

Queremos calcular la media mensual de la extensión del hielo marino para el periodo de referencia de 1981 a 2010, lo que nos permitirá observar el comportamiento promedio y normalizar los datos para análisis adicionales.

[29]:
```
# Filtrar los datos para el periodo de referencia 1981-2010
ice_filtered = ice_clean[(ice_clean['year'] >= 1981) & (ice_clean['year'] <=␣
 ↪2010)]  # Filtramos solo el intervalo 1981-2010

# Seleccionar solo las columnas numéricas para evitar posibles errores de tipo␣
 ↪de datos
num_features = ice_filtered.select_dtypes(include=[np.number])  # Seleccionamos␣
 ↪solo columnas numéricas

# Separar las características de la variable objetivo
# Suponemos que 'extent' es la variable que queremos predecir, y 'area' también␣
 ↪podría ser una característica relevante.
X = num_features.drop(columns=['extent'])  # Eliminamos la columna 'extent' del␣
 ↪conjunto de características
y = num_features['extent']  # Asignamos 'extent' como la variable objetivo

# Dividir los datos en conjuntos de entrenamiento y prueba para evaluar el␣
 ↪modelo
from sklearn.model_selection import train_test_split  # Importamos la función␣
 ↪para dividir los datos
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)  # 80% entrenamiento, 20% prueba

# Inicializar el modelo de regresión lineal
from sklearn.linear_model import LinearRegression
regr_feat1 = LinearRegression()  # Creamos una instancia del modelo de␣
 ↪regresión lineal

# Obtener el número de características disponibles
n_features = X_train.shape[1]  # Determinamos el número de columnas en el␣
 ↪conjunto de entrenamiento
```

```
scores = []  # Inicializamos una lista para almacenar los resultados de cada␣
 ↪característica

# Iterar sobre todas las características en el conjunto de entrenamiento
for i in range(n_features):
    # Seleccionar la característica individual en cada iteración
    feat_name = X_train.columns[i]  # Obtenemos el nombre de la característica␣
 ↪actual
    feat1_train = X_train[[feat_name]].values.reshape(-1, 1)  # Convertimos los␣
 ↪datos a una matriz de 2D
    feat1_test = X_test[[feat_name]].values.reshape(-1, 1)    # Convertimos los␣
 ↪datos de prueba a una matriz de 2D

    # Entrenar el modelo de regresión con la característica seleccionada
    regr_feat1.fit(feat1_train, y_train)  # Ajustamos el modelo con los datos␣
 ↪de entrenamiento

    # Realizar predicciones en los conjuntos de entrenamiento y prueba
    y_train_pred = regr_feat1.predict(feat1_train)  # Predicciones en el␣
 ↪conjunto de entrenamiento
    y_test_pred = regr_feat1.predict(feat1_test)    # Predicciones en el␣
 ↪conjunto de prueba

    # Evaluar el modelo en términos de coeficiente de determinación y error␣
 ↪cuadrático medio
    train_score = regr_feat1.score(feat1_train, y_train)  # Coeficiente de␣
 ↪determinación en entrenamiento
    test_score = regr_feat1.score(feat1_test, y_test)    # Coeficiente de␣
 ↪determinación en prueba
    mse_train = np.mean((y_train_pred - y_train)**2)  # Error cuadrático medio␣
 ↪en entrenamiento
    mse_test = np.mean((y_test_pred - y_test)**2)    # Error cuadrático medio␣
 ↪en prueba

    # Agregar los resultados de la característica actual a la lista de puntajes
    scores.append([train_score, test_score, mse_train, mse_test])  # Guardamos␣
 ↪las métricas de desempeño

# Crear un DataFrame con los resultados de la evaluación de cada característica
df_scores = pd.DataFrame(scores, columns=["train_score", "test_score",␣
 ↪"train_mse", "test_mse"], index=X_train.columns)
df_scores.sort_values(by="test_score", ascending=False, inplace=True)  #␣
 ↪Ordenamos de mejor a peor rendimiento
df_scores  # Mostramos el DataFrame con los puntajes
```

[29]:

	train_score	test_score	train_mse	test_mse
area	9.999996e-01	0.968961	0.291947	0.276685
mo	8.767695e-07	-551.855666	691102.277330	4928.250624
year	5.968969e-03	-1054.444870	686977.711807	9408.417341

4.3.5 Interpretación de Resultados para el Periodo 1981-2010

La interpretación de los resultados obtenidos puede realizarse analizando `train_score`, `test_score`, `train_mse` y `test_mse` para cada característica, donde:

Descripción de las métricas:

- **train_score**: Coeficiente de determinación R^2 en el conjunto de entrenamiento. Muestra qué proporción de la variabilidad de la variable dependiente es explicada por el modelo.
- **test_score**: Coeficiente de determinación R^2 en el conjunto de prueba. Muestra qué proporción de la variabilidad de la variable dependiente es explicada por el modelo en los datos de prueba.
- **train_mse**: Error cuadrático medio (MSE) en el conjunto de entrenamiento. Indica la media de los errores al cuadrado entre las predicciones y los valores reales en entrenamiento.
- **test_mse**: Error cuadrático medio (MSE) en el conjunto de prueba. Representa la media de los errores al cuadrado entre las predicciones y los valores reales en el conjunto de prueba.

Interpretación de las Características

1. **Área (area)**
   - **train_score: 0.9999996**: El modelo con `area` explica casi toda la variabilidad en `extent` en el entrenamiento.
   - **test_score: 0.968961**: El modelo con `area` explica el 96.9% de la variabilidad en `extent` en el conjunto de prueba.
   - **train_mse y test_mse**: Los errores son muy bajos, lo que indica una alta precisión.

2. **Mes (mo)**
   - **train_score**: Valor extremadamente bajo, no explica la variabilidad en `extent`.
   - **test_score**: Valor negativo, indica que el modelo con `mo` es peor que la media.
   - **train_mse y test_mse**: Los errores son altos, indicando predicciones imprecisas.

3. **Año (year)**
   - Similar a **Mes (mo)**, con bajos `train_score` y `test_score` y altos errores, indicando predicciones inexactas.

4. **Anomalía (anomaly)**
   - **train_score**: Explica bastante la variabilidad en entrenamiento.
   - **test_score**: Valor negativo, indicando sobreajuste.
   - **train_mse y test_mse**: Bajo en entrenamiento, alto en prueba, indicando sobreajuste.

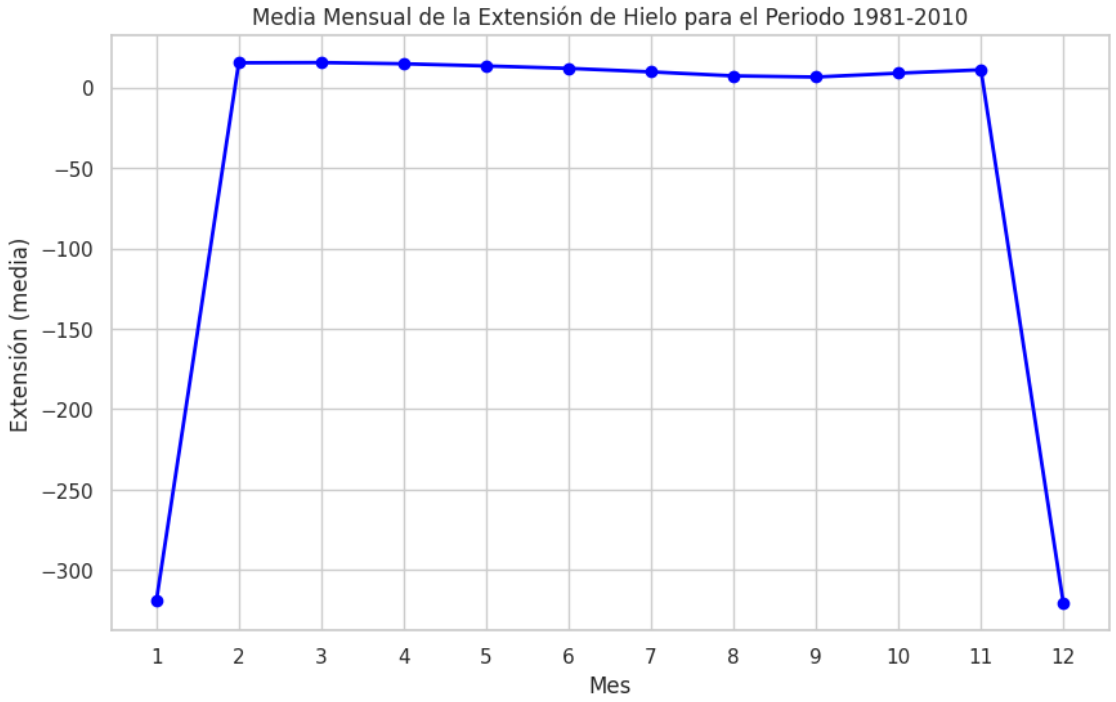
4.3.6 Conclusiones

- **Area** es la característica más efectiva para predecir `extent`.
- **Anomaly** sugiere sobreajuste.
- **Year** y **mo** no predicen bien `extent`.

[30]:
```
# Filtrar el periodo de referencia de 1981-2010
reference_period = ice_clean[(ice_clean['year'] >= 1981) & (ice_clean['year']␣
 ↪<= 2010)]

# Calcular la media mensual para el periodo de referencia
monthly_mean = reference_period.groupby('mo')['extent'].mean()  # Calculamos la␣
 ↪media de 'extent' por mes
print(monthly_mean)  # Mostramos los valores medios por mes
```

```
mo
1    -319.252333
2      15.345667
3      15.492333
4      14.753667
5      13.391000
6      11.890333
7       9.696333
8       7.224667
9       6.517000
10      8.905333
11     10.990333
12   -320.676000
Name: extent, dtype: float64
```

[31]:
```
# Graficar la media mensual de la extensión de hielo para el periodo de␣
 ↪referencia
import matplotlib.pyplot as plt  # Importamos la librería para gráficos
plt.figure(figsize=(10, 6))  # Configuramos el tamaño de la figura
plt.plot(monthly_mean.index, monthly_mean.values, marker='o', linestyle='-',␣
 ↪color='b')  # Configuramos la línea de la gráfica
plt.xlabel('Mes')  # Etiqueta del eje X
plt.ylabel('Extensión (media)')  # Etiqueta del eje Y
plt.title('Media Mensual de la Extensión de Hielo para el Periodo 1981-2010') ␣
 ↪# Título del gráfico
plt.grid(True)  # Activamos la cuadrícula
plt.xticks(range(1, 13))  # Configuramos los valores en el eje X para cada mes
plt.show()  # Mostramos el gráfico
```



4.3.7 3. Normalizar los datos

Restamos la media mensual de cada mes para calcular la anomalía, la cual representa las desviaciones en la extensión del hielo marino respecto al promedio mensual del periodo de referencia (1981-2010).

[32]:
```
# Añadir columna de anomalía calculando la diferencia respecto a la media␣
 ↪mensual
ice_clean['anomaly'] = ice_clean.apply(lambda row: row['extent'] -␣
 ↪monthly_mean[row['mo']], axis=1)
ice_clean.head()  # Visualizar las primeras filas para verificar la nueva␣
 ↪columna 'anomaly'
```

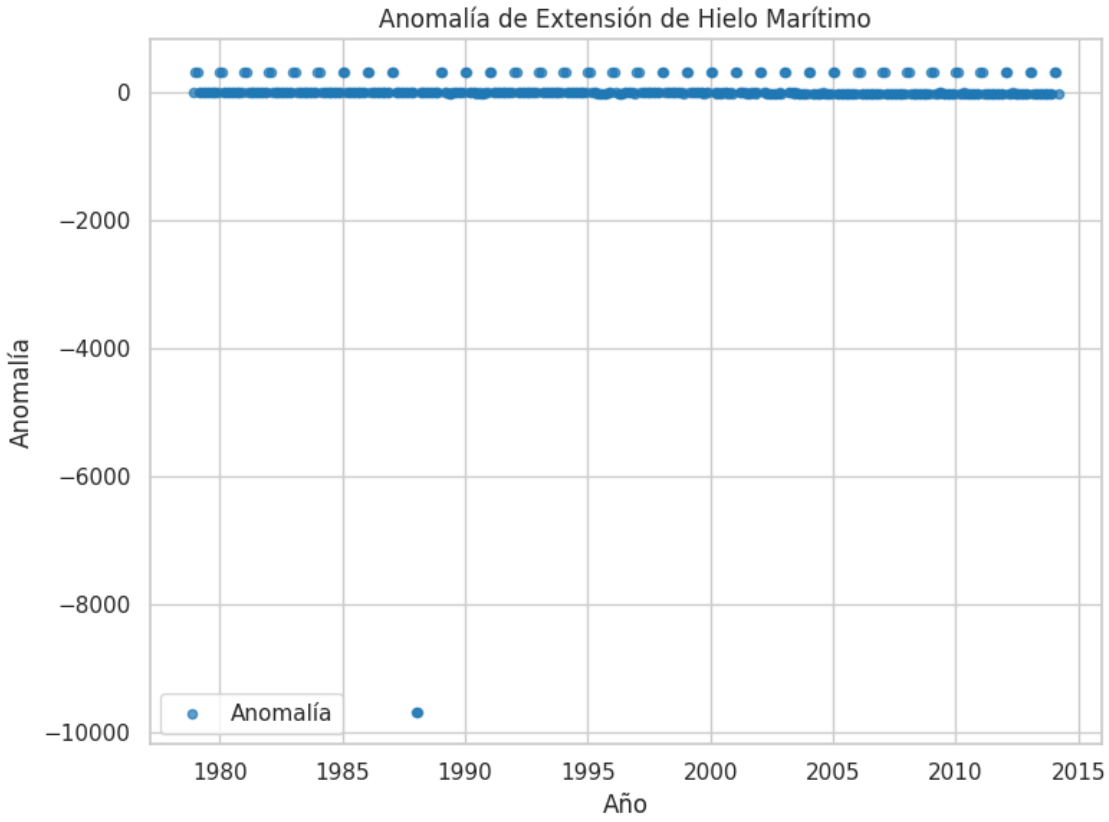
[32]:

	year	mo	data_type	region	extent	area	anomaly
0	1979	1	Goddard	N	15.54	12.33	334.792333
1	1980	1	Goddard	N	14.96	11.85	334.212333
2	1981	1	Goddard	N	15.03	11.82	334.282333
3	1982	1	Goddard	N	15.26	12.11	334.512333
4	1983	1	Goddard	N	15.10	11.92	334.352333

4.3.8 4. Graficar los datos

A continuación, graficamos la anomalía a lo largo del tiempo para analizar las tendencias mensuales y observar cambios en la extensión del hielo.

[33]:
```
# Graficar la anomalía en función del tiempo (año y mes)
plt.figure(figsize=(8, 6))
plt.scatter(ice_clean['year'] + ice_clean['mo'] / 12, ice_clean['anomaly'],␣
 ↪marker='o', s=20, alpha=0.7, label='Anomalía')
plt.xlabel('Año')
plt.ylabel('Anomalía')
plt.title('Anomalía de Extensión de Hielo Marítimo')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

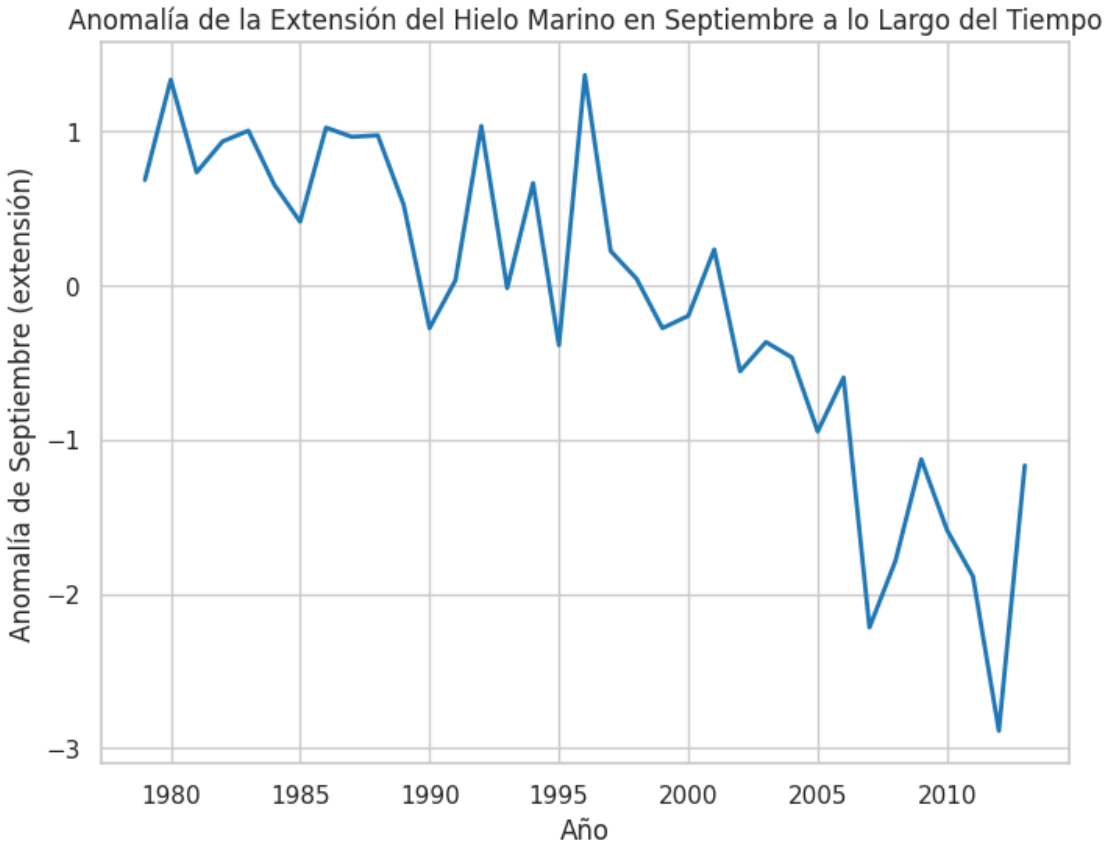


4.3.9 Graficar las anomalías de un mes específico

Para observar variaciones anuales específicas en un mes concreto, como septiembre, graficamos la anomalía de extensión del hielo marino en dicho mes.

[34]:
```
# Graficar las anomalías para un mes específico (ejemplo, septiembre)
september_anomalies = ice_clean[ice_clean['mo'] == 9]
plt.figure(figsize=(8, 6))
plt.plot(september_anomalies['year'], september_anomalies['anomaly'])
```

```
plt.xlabel('Año')
plt.ylabel('Anomalía de Septiembre (extensión)')
plt.title('Anomalía de la Extensión del Hielo Marino en Septiembre a lo Largo␣
  ↪del Tiempo')
plt.show()
```



4.3.10 5. Calcular la tendencia como una regresión lineal simple (OLS)

Utilizamos una regresión lineal simple para evaluar la tendencia y cuantificar la posible disminución en la extensión del hielo marino a lo largo de los años.

[35]:
```
# Preparar los datos para la regresión lineal
X = ice_clean['year'].values.reshape(-1, 1)
y = ice_clean['anomaly'].values

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

# Obtener los coeficientes de la regresión
```

```
slope = model.coef_[0]
intercept = model.intercept_

print(f'Pendiente: {slope}')
print(f'Intercepto: {intercept}')
```

```
Pendiente: 3.8035048033351457
Intercepto: -7582.394666073303
```

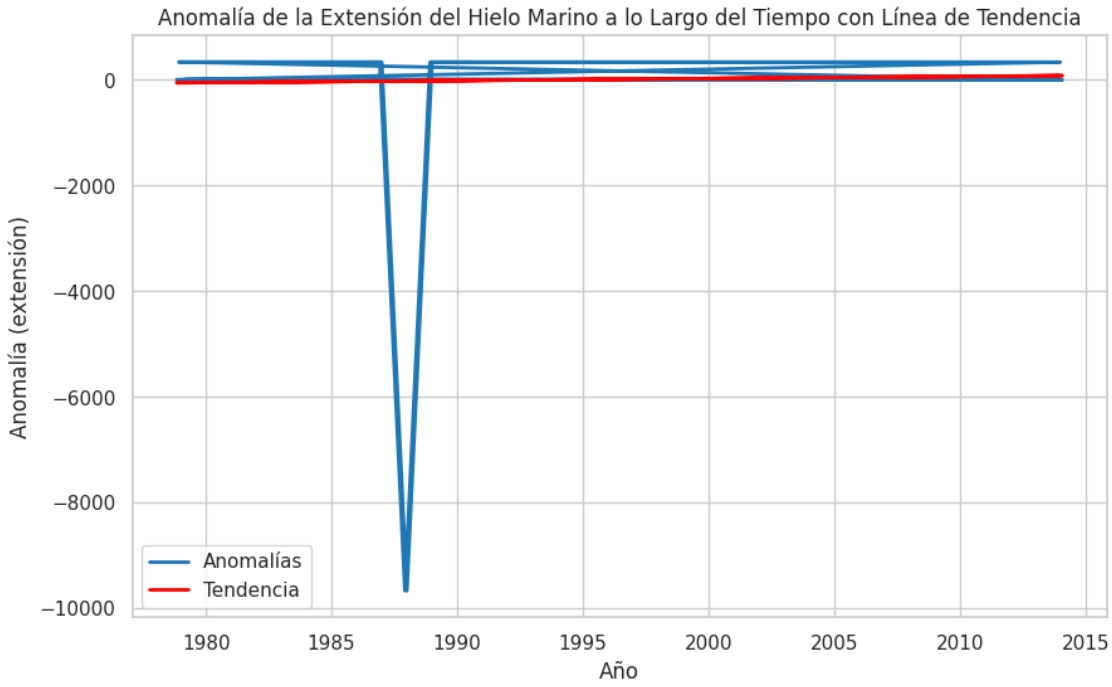
4.3.11 Interpretación de los coeficientes de regresión

- **Pendiente**: Indica el cambio promedio en la anomalía de la extensión de hielo marino cada año. Un valor negativo implica una tendencia de disminución en la extensión del hielo.
- **Intercepto**: Valor inicial de la anomalía cuando el año es 0 (en este caso, es más útil como referencia matemática que práctica).

4.3.12 Graficar la tendencia

A continuación, visualizamos la línea de tendencia de la anomalía para observar la disminución general en la extensión del hielo marino.

[36]:
```
# Graficar la tendencia junto con las anomalías
plt.figure(figsize=(10, 6))
plt.plot(ice_clean['year'] + (ice_clean['mo'] - 1) / 12, ice_clean['anomaly'],␣
 ↪label='Anomalías')
plt.plot(ice_clean['year'] + (ice_clean['mo'] - 1) / 12, model.predict(X),␣
 ↪color='red', label='Tendencia')
plt.xlabel('Año')
plt.ylabel('Anomalía (extensión)')
plt.title('Anomalía de la Extensión del Hielo Marino a lo Largo del Tiempo con␣
 ↪Línea de Tendencia')
plt.legend()
plt.show()
```



4.3.13 6. Estimar el valor de la extensión para 2025

Usamos el modelo de regresión lineal para predecir la anomalía estimada de extensión de hielo marino en el año 2025, con enfoque en el mes de septiembre.

[37]:
```
# Predecir la anomalía para el año 2025
year_2025 = np.array([[2025]])
anomaly_2025 = model.predict(year_2025)
print(f'Anomalía estimada para 2025: {anomaly_2025[0]}')
```

```
Anomalía estimada para 2025: 119.70256068036724
```

4.3.14 Calcular la extensión estimada para septiembre de 2025

Sumamos la anomalía estimada a la media histórica de septiembre para obtener la extensión proyectada de hielo marino en ese mes.

[38]:
```
# Calcular la extensión estimada para septiembre de 2025
september_mean_extent = monthly_mean[9]
estimated_extent_2025 = september_mean_extent + anomaly_2025[0]
print(f'Extensión estimada del hielo marino para septiembre de 2025:␣
 ↪{estimated_extent_2025}')
```

```
Extensión estimada del hielo marino para septiembre de 2025: 126.21956068036724
```

4.3.15 Interpretación de los Valores

Los valores proyectados para 2025 proporcionan una visión sobre la posible extensión de hielo marino y su desviación del promedio histórico en septiembre:

- **Anomalía estimada para 2025**: Este valor indica la desviación proyectada en la extensión del hielo para 2025 con respecto al promedio. Una anomalía positiva sugiere que el hielo podría extenderse más de lo usual para este mes, mientras que una anomalía negativa indicaría lo contrario.
- **Extensión estimada del hielo marino para septiembre de 2025**: Este valor, obtenido sumando la anomalía a la media histórica de septiembre, representa la extensión proyectada en unidades de superficie. Proporciona una referencia cuantitativa sobre la cobertura de hielo esperada en el futuro.

4.4 Visualización

4.4.1 La importancia de graficar

El cuarteto de Anscombe es una colección de cuatro conjuntos de datos que fueron creados por el estadístico Francis Anscombe en 1973. Estos conjuntos fueron diseñados para mostrar una lección clave en el análisis de datos: **la importancia de graficar los datos antes de analizarlos**. Aunque estos conjuntos de datos tienen estadísticas descriptivas simples casi idénticas, sus distribuciones son muy diferentes y se ven significativamente distintas cuando se grafican.

Los cuatro conjuntos de datos en el cuarteto de Anscombe tienen características estadísticas similares, como la media, la varianza, la correlación y la pendiente de la regresión lineal, pero cuando se grafican, muestran patrones muy distintos. Esto pone de manifiesto que las estadísticas descriptivas y los cálculos como la media o la desviación estándar no siempre cuentan toda la historia de los datos.

Cada conjunto de datos contiene **once puntos de datos** (x, y), lo que facilita el análisis visual y permite observar cómo las visualizaciones pueden influir en la interpretación de los datos. Anscombe demostró que, aunque las estadísticas de estos conjuntos de datos sean similares, los patrones visuales revelan diferentes historias, especialmente en presencia de **valores atípicos** o **puntos de datos influyentes**.

El propósito de esta colección es **advertir sobre los riesgos de basarse únicamente en las estadísticas resumidas** sin antes realizar una exploración visual. A través de este cuarteto de datos, Anscombe resaltó que **no se deben tomar decisiones basadas solo en las estadísticas tradicionales** sin considerar los efectos que los valores atípicos y otros factores pueden tener en los resultados.

La visualización es una herramienta esencial en el análisis de datos, porque permite identificar relaciones no lineales, detectar valores atípicos, y proporciona una comprensión más profunda de los patrones que podrían pasar desapercibidos en los cálculos estadísticos convencionales.

Para ilustrar esto, cada conjunto del cuarteto tiene el mismo promedio, varianza y correlación, pero al graficarlos, se puede observar una diferencia sustancial en su forma. Los cuatro gráficos revelan diferentes patrones:

1. **Conjunto 1**: Una relación lineal perfecta, con todos los puntos de datos alineados en una recta.

2. **Conjunto 2**: Una relación cuadrática, donde los puntos de datos siguen una parábola.
3. **Conjunto 3**: Un conjunto con un valor atípico en el extremo, lo que afecta la forma de la regresión.
4. **Conjunto 4**: Un conjunto donde casi todos los puntos están alineados, pero un único valor atípico desvirtúa la tendencia.

La lección clave de Anscombe es que **si solo se observaran las estadísticas**, como la media y la desviación estándar, los cuatro conjuntos parecerían casi idénticos. Sin embargo, **cuando los datos se grafican**, se revela que los patrones subyacentes son muy diferentes, lo que puede cambiar completamente la interpretación de los datos.

Por lo tanto, el cuarteto de Anscombe es un recordatorio de la necesidad de una **exploración visual** de los datos antes de hacer análisis más complejos o conclusiones basadas solo en estadísticas agregadas. Sin un análisis visual, es fácil pasar por alto patrones importantes y tomar decisiones erróneas.

Fuente: Wikipedia.

4.5 Ejemplo 3: Datos de Vivienda

En este ejemplo, continuaremos con el conjunto de datos de vivienda de Boston para explorar las relaciones lineales entre las variables y realizar visualizaciones de regresión.

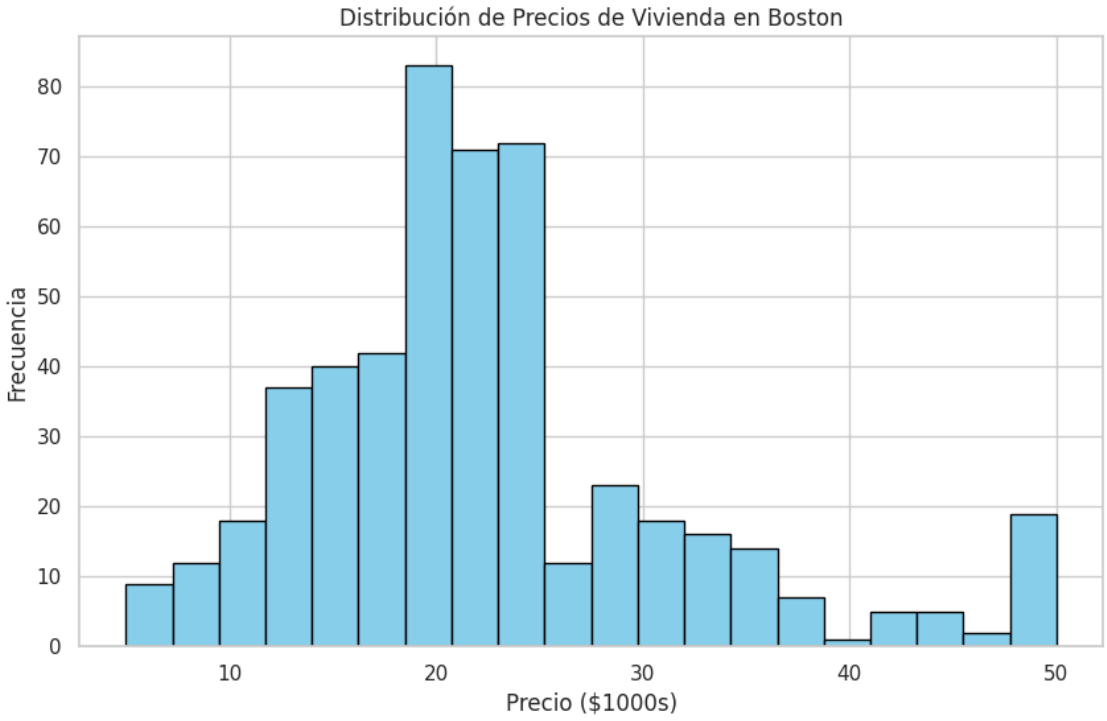
[39]:
```
# Cargar los datos de vivienda de Boston
from sklearn import datasets
#boston = datasets.load_boston() # Dictionary-like object that exposes its keys␣
 ↪as attributes.
df_boston = pd.DataFrame(boston.data, columns=boston.feature_names) # Crear un␣
 ↪DataFrame a partir del conjunto de datos de Boston
df_boston['PRICE'] = boston.target
print('Forma de los datos: {}'.format(df_boston.shape)) # Confirmar dimensiones␣
 ↪del DataFrame
```

```
Forma de los datos: (506, 14)
```

4.5.1 Histogramas

Comencemos explorando la distribución de los precios mediante un histograma.

[40]:
```
# Histograma de precios:
plt.figure(figsize=(10, 6))
plt.hist(df_boston.PRICE, bins=20, color='skyblue', edgecolor='black')
plt.xlabel('Precio ($1000s)')
plt.ylabel('Frecuencia')
plt.title('Distribución de Precios de Vivienda en Boston')
plt.show()
```

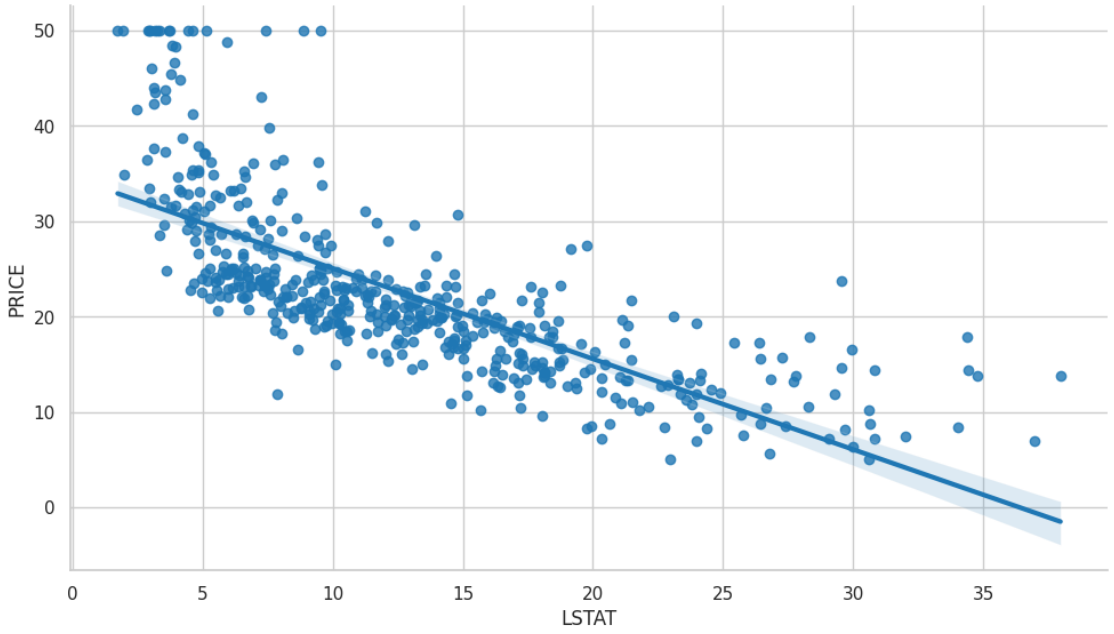


4.5.2 Visualización de Regresión Lineal con Seaborn (lmplot)

La función `lmplot()` del módulo Seaborn permite explorar relaciones lineales de diversas maneras en conjuntos de datos multidimensionales. Los datos deben estar en un `DataFrame` de Pandas. Para realizar una visualización de regresión, indicamos las variables predictoras y de respuesta junto con el conjunto de datos.

En este caso, usaremos el precio de las casas como nuestra variable de respuesta (`PRICE`) y, como predictor, el atributo `LSTAT` (proporción de estatus más bajo de la población) para observar cómo esta variable influye en los precios.

[41]:
```
# Visualización de la relación entre el precio y LSTAT
g = sns.lmplot(x="LSTAT", y="PRICE", data=df_boston, aspect=2);
g.fig.set_size_inches(10, 6)
```



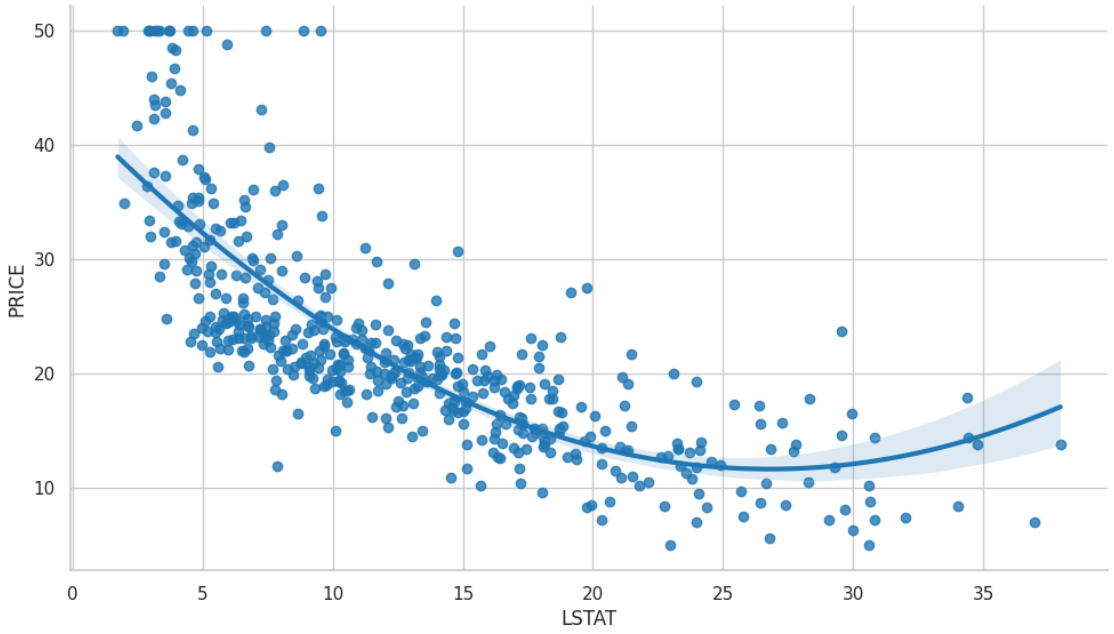
4.5.3 Componentes del lmplot

- **Diagrama de dispersión**: muestra los puntos de datos observados.
- **Línea de regresión**: muestra el modelo lineal estimado que relaciona las dos variables. Dado que la línea es solo una estimación, se incluye una banda de confianza del 95% (calculada mediante Bootstraping) para representar la certeza en el modelo.

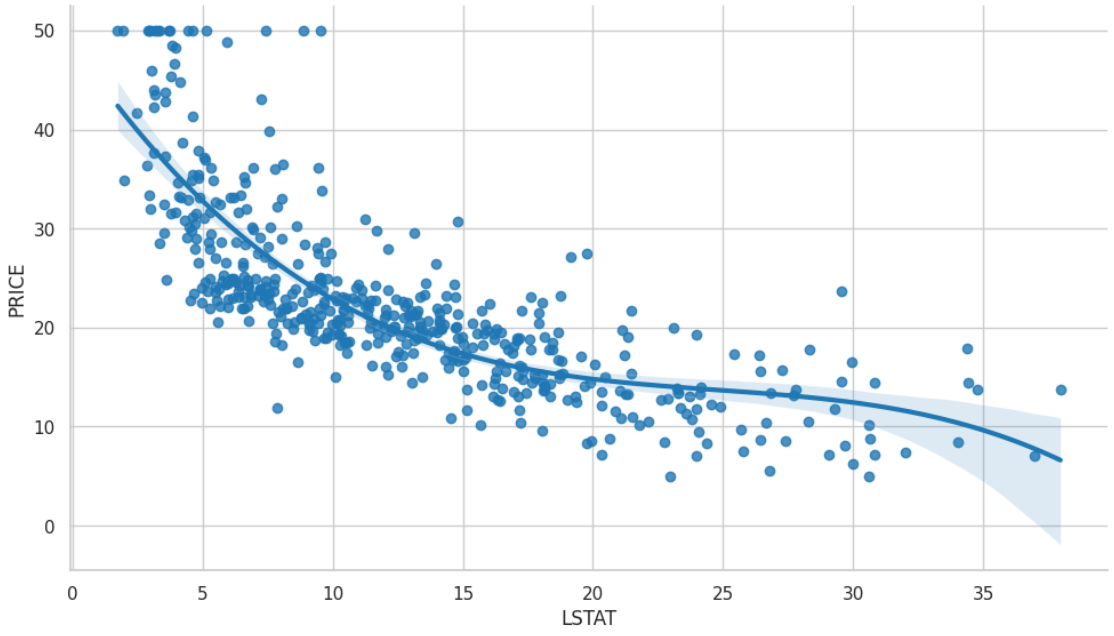
4.5.4 Análisis de la Relación entre Precio y LSTAT

Observemos si la relación entre el precio de la vivienda y `LSTAT` es no lineal o si una línea recta no es un buen ajuste. Podríamos obtener un mejor ajuste añadiendo términos polinomiales:

[42]:
```
# Estimar una regresión polinómica de orden 2
g = sns.lmplot(x="LSTAT", y="PRICE", data=df_boston, aspect=2, order=2);
g.fig.set_size_inches(10, 6)
```



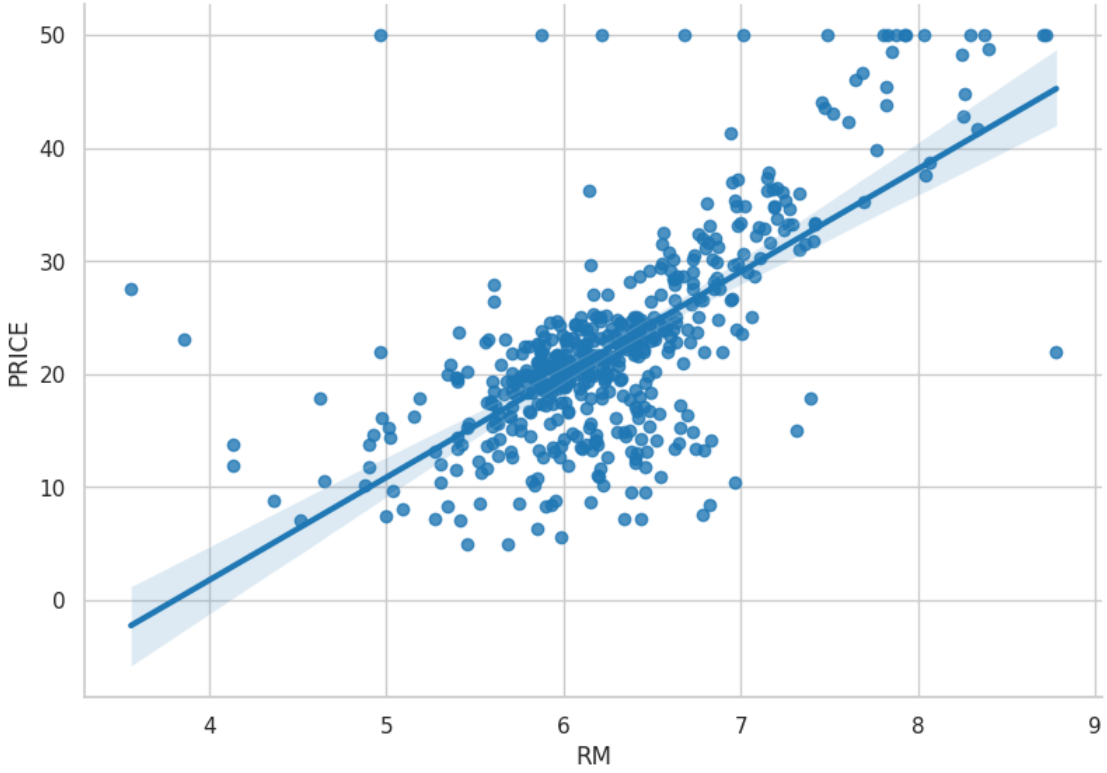
[43]:
```
# Estimar una regresión polinómica de orden 3
g = sns.lmplot(x="LSTAT", y="PRICE", data=df_boston, aspect=2, order=3);
g.fig.set_size_inches(10, 6)
```



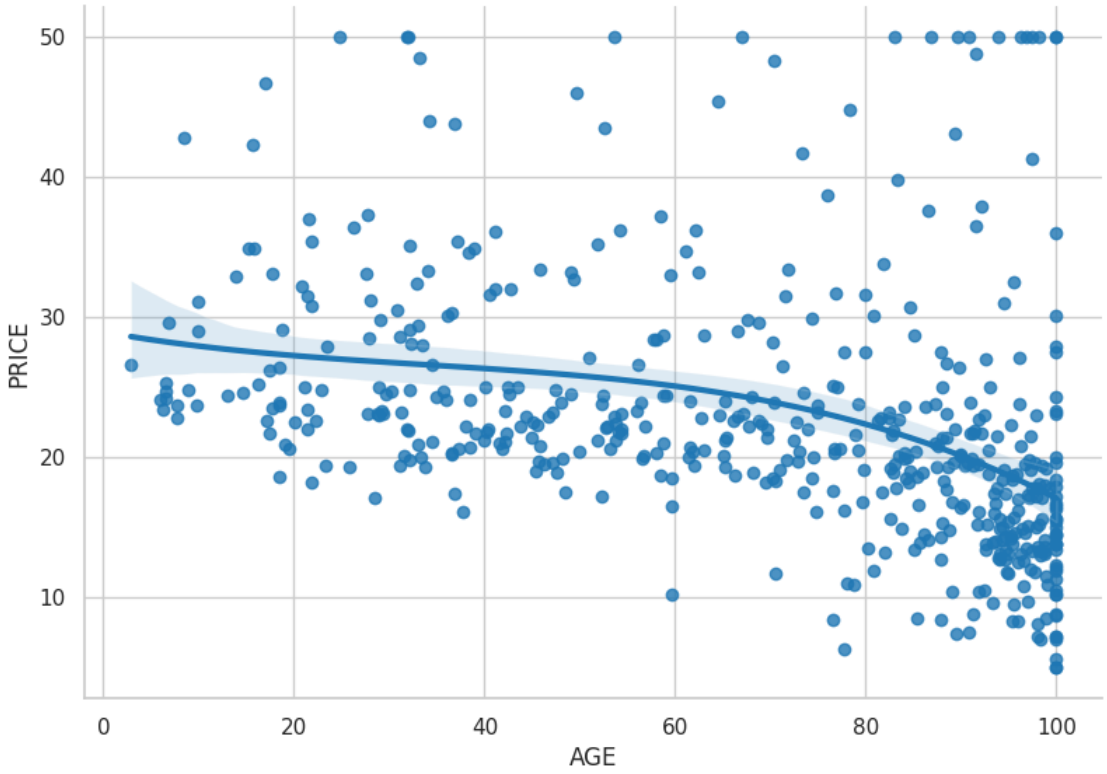
4.5.5 Exploración de Otras Variables Predictoras

Consideremos otras variables del conjunto de datos de vivienda, como `RM` (promedio de habitaciones por vivienda) y `AGE` (proporción de unidades ocupadas por sus propietarios construidas antes de 1940), para observar sus relaciones con los precios de las viviendas.

[44]:
```
# Visualización de la relación entre el precio y "promedio de habitaciones por␣
↪vivienda"
g = sns.lmplot(x="RM", y="PRICE", data=df_boston, aspect=2);
g.fig.set_size_inches(8, 6)
```



[45]:
```
# Visualización de la relación entre el precio y "proporción de unidades␣
↪ocupadas por sus propietarios construidas antes de 1940"
g = sns.lmplot(x="AGE", y="PRICE", data=df_boston, aspect=2, order=3);
g.fig.set_size_inches(8, 6)
```



4.5.6 Observaciones Finales

Estos gráficos nos permiten visualizar cómo distintas características (`LSTAT`, `RM`, `AGE`) se relacionan con el precio de las viviendas en Boston. La visualización de relaciones lineales y no lineales mediante regresión polinómica puede darnos una idea de qué tan bien estos atributos pueden predecir el precio de la vivienda y si es necesario utilizar modelos más complejos o transformaciones adicionales.