Actividad 2: M2003B

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

Instrucciones:

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

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

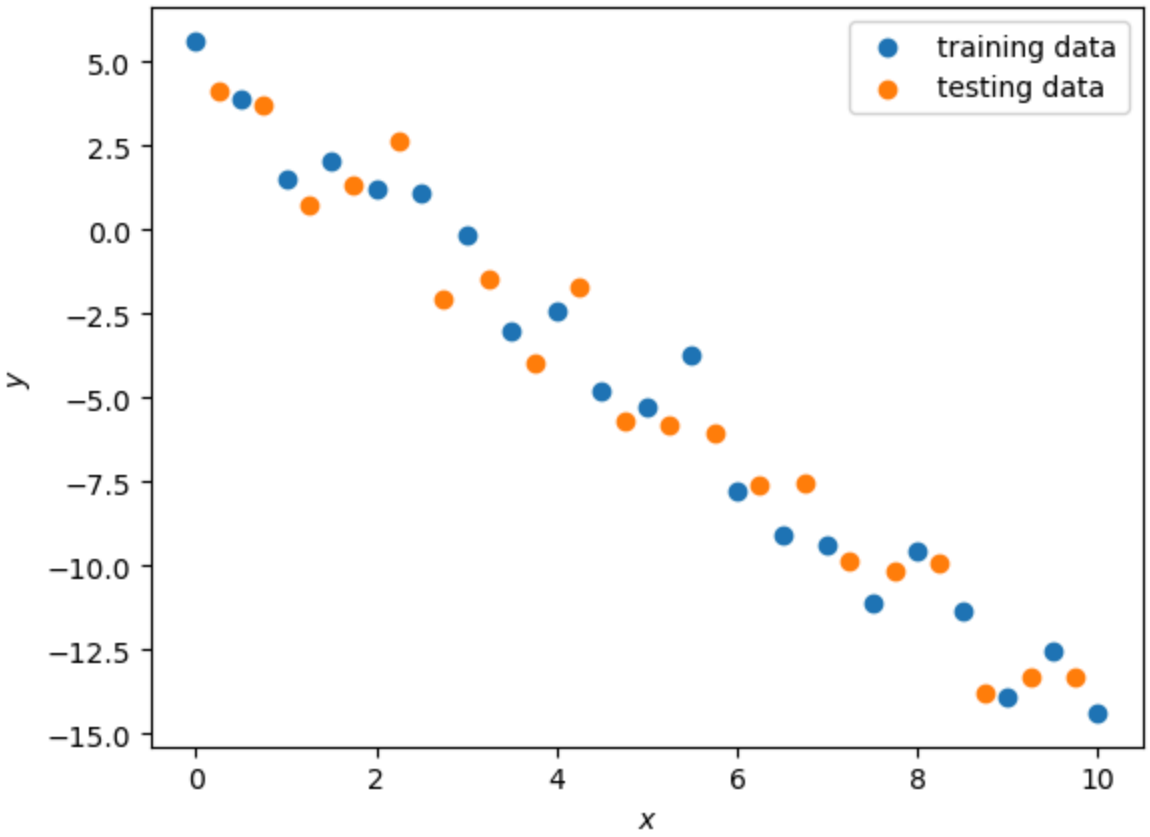
1. Regresión lineal - decenso del gradiente

1.1 Cargar y visualizar los datos

Buscamos los archivos:

- `.../data/tutorial_1_data/linear_training.npy`
- `.../data/tutorial_1_data/linear_testing.npy`.

In [2]:
```
xy_training = np.load('C:/Users/diana/Documents/tutorial_1_data/linear_training.npy
xy_testing = np.load('C:/Users/diana/Documents/tutorial_1_data/linear_testing.npy')
x = xy_training[:, 0]
y = xy_training[:, 1]
x_test = xy_testing[:, 0]
y_test = xy_testing[:, 1]

# graficamos
plt.scatter(x, y, label='training data')
plt.scatter(x_test, y_test, label='testing data')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend()
plt.show()
```



Esta son los datos a los cuales intentaremos ajustar una recta. Queremos ver qué tan cerca podemos llegar de los valores verdaderos de la pendiente y la intersección con el eje y que introdujimos manualmente.

1.2 Definir la función de pérdida

Para lograr un buen ajuste, primero debemos definir qué significa un buen ajuste. En una situación donde estamos tratando de predecir un número, es común usar el **error cuadrático medio** (*Mean Squared Error*, MSE). Este se define como:

$$L^{\text{MSE}} = \frac{1}{N}\sum_{i}^{N}\left(y_i - f(x_i)\right)^2$$

donde y_i es el valor *verdadero* en los datos y $f(x_i)$ es el valor predicho.

Ejercicio:

Complete la siguiente función:

In [3]:
```
def MSE(y_true, y_pred):
    '''
    Calcula el Error Cuadrático Medio (Mean Squared Error, MSE).
    Fórmula: MSE = (1/N) * sum((y_true - y_pred)^2)
    Requiere que y_true y y_pred tengan la misma longitud.

    '''
```

```
    N = y_true.shape[0]

    # calcular la MSE
    # no use una loop
    ### empiece su codigo aqui ### (alrededor de 1 linea de code)
    mse = np.sum((y_true - y_pred) ** 2) / N
    ### aqui acaba el codigo ###

    return mse
```

In [4]:
```
# pruebe su función
y_true = np.array([1, 2, 3])
y_pred = np.array([1.1, 1.9, 3])
print('MSE = ' + str(MSE(y_true, y_pred)))
```

```
MSE = 0.006666666666666678
```

Salida esperada:

MSE	0.006666666666666678

1.3 Encontrar el gradiente

Para encontrar el mínimo de la función de pérdida (minimizar el error), necesitamos calcular el gradiente de la pérdida con respecto a los parámetros del modelo. Como nuestros datos parecen lineales, usemos una forma simple de:

$$f(x_i) = mx_i + b$$

Escribimos esto aplicando la regla de la cadena, como preparación para el caso de redes neuronales profundas.

Derivada parcial con respecto a m

$$\frac{\partial}{\partial m} L^{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} -2 \left(y_i - f(x_i)\right) \cdot \frac{\partial f(x_i)}{\partial m}$$

Como $\frac{\partial f(x_i)}{\partial m} = x_i$, esto se convierte en:

$$\frac{\partial}{\partial m} L^{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} -2 \left(y_i - f(x_i)\right) \cdot x_i$$

Derivada parcial con respecto a b

$$\frac{\partial}{\partial b} L^{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} -2 \left(y_i - f(x_i)\right) \cdot \frac{\partial f(x_i)}{\partial b}$$

Como $\frac{\partial f(x_i)}{\partial b} = 1$, esto se convierte en:

$$\frac{\partial}{\partial b} L^{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} -2 \left(y_i - f(x_i) \right)$$

1.4 Calcular la pérdida y el gradiente juntos

Ejercicio Este parte del ejercicio será llamado **"propagación hacia adelante"** (*forward propagation*) cuando lleguemos a las redes neuronales. Queremos una función que reciba:

- los parámetros del modelo m y b,
- los datos de entrada,
- los valores verdaderos de y ,

y que calcule y devuelva la **pérdida** y el **gradiente** (tanto para m como para b.

Complete la siguiente función de propagación hacia adelante:

In [5]:
```
def propagate(m, b, X, Y):
    '''
    Implementa la función de pérdida y su gradiente

    Argumentos:
    m -- escalar, pendiente de la recta
    b -- escalar, sesgo (intersección)
    X -- np.array, datos de entrada
    Y -- valores verdaderos

    Regresa:
    loss -- Error cuadrático medio
    dm -- gradiente de la pérdida con respecto a m
    db -- gradiente de la pérdida con respecto a b
    '''
    ### Implemente su code aqui ###
    y_pred = m * X + b
    loss =  np.mean((y_pred - Y) ** 2)
    ## fin de code ###

    ### encontrar gradiente
    ### Implemente su code aqui ###
    dm = 2 * np.mean((y_pred - Y) * X)
    db = 2 * np.mean(y_pred - Y)
    ## fin code ###

    # chequeos de sanidad
    assert(dm.dtype == float)
    assert(db.dtype == float)

    # este nos crea una tabla "look-up", facilita
    grads = {'dm': dm,
             'db': db
            }

    return grads, loss
```

Ejercicio:

Implemente su función

In [6]:
```
m, b, X, Y = 1.0, 2.5, np.array([1, 2, 3, 4]), np.array([1.2, -2.3, 3.0, 4.5])
grads, loss = propagate(m, b, X, Y)
print('dm = ' + str(grads['dm']))
print('db = ' + str(grads['db']))
print('MSE = ' + str(loss))
```

```
dm = 15.7
db = 6.8
MSE = 15.444999999999999
```

Salida esperada:

dm	15.7
db	6.8
MSE	15.444999999999999

1.5 Realizar la optimización

Ejercicio: Escriba la función de optimización. El objetivo es aprender los valores de m y b minimizando la función de pérdida.

Para un parámetro θ, la regla de actualización es:

$$\theta = \theta - \alpha \cdot d\theta$$

donde α es la **tasa de aprendizaje** (*learning rate*) y $d\theta$ es la derivada de la pérdida con respecto a $theta$.

In [7]:
```
def optimize(m, b, X, Y, num_iterations, learning_rate, print_loss=False):
    '''
    La función encuentra los valores óptimos de m y b ejecutando un algoritmo de de

    Argumentos:
    m -- escalar, pendiente de la recta
    b -- escalar, sesgo (intersección)
    X -- np.array, datos de entrada
    Y -- valores verdaderos
    num_iterations -- número de iteraciones del ciclo de optimización
    learning_rate -- tasa de aprendizaje de la regla de actualización del descenso
    print_loss -- True para imprimir la pérdida cada 100 pasos

    Regresa:
    params -- diccionario que contiene la pendiente m y el sesgo b
    grads -- diccionario que contiene los gradientes de la pendiente y el sesgo con
    losss -- lista de todas las pérdidas calculadas durante la optimización, se usa
    '''
```

```
    losss = []
    slopes = []
    biases = []

    for i in range(num_iterations):
        slopes.append(m)
        biases.append(b)

        # loss and caculate gradient
        ### empiece code aqui ### (cerca 1 lineas de code)
        grads, loss = propagate(m, b, X, Y)
        ### termine codigo aqui ###

        # obtenga las derivadas
        ### empiece code aqui ### (cerca 2 lineas de code)
        dm = grads['dm']
        db = grads['db']
        ### termine ###

        # update rule
        ### empezar code aqui ### (cerca 2 lineasa de code)
        m = m = m - learning_rate * dm
        b = b - learning_rate * db
        ### terminar codigo aqui ###

        # guardar la loss
        losss.append(loss)

        # imprimir loss cada 100 training iterations
        if print_loss and i % 100 == 0:
            print('loss after iteration %i: %f' %(i, loss))

    params = {'m': m,
              'b': b
             }
    grads = {'dm': dm,
             'db': db
            }

    return params, grads, losss, slopes, biases
```

Ejercicio: Ahora usa la función `optimize` para encontrar la mejor recta que se ajuste a los datos, usando como suposición inicial $m = 0$ y $b = 0$. Por ahora, utilize una **tasa de aprendizaje** de 0.01 y realiza al menos **1000 iteraciones** de entrenamiento.

In [8]:
```
# complete codigo
params, grads, costs, slopes, biases = optimize(m=0.0, b=0.0, X=x, Y=y, num_iterati
```

```
loss after iteration 0: 60.550528
loss after iteration 100: 3.389798
loss after iteration 200: 1.821635
loss after iteration 300: 1.274666
loss after iteration 400: 1.083885
loss after iteration 500: 1.017341
loss after iteration 600: 0.994131
loss after iteration 700: 0.986036
loss after iteration 800: 0.983212
loss after iteration 900: 0.982227
loss after iteration 1000: 0.981883
loss after iteration 1100: 0.981764
loss after iteration 1200: 0.981722
loss after iteration 1300: 0.981707
loss after iteration 1400: 0.981702
loss after iteration 1500: 0.981700
loss after iteration 1600: 0.981700
loss after iteration 1700: 0.981700
loss after iteration 1800: 0.981699
loss after iteration 1900: 0.981699
```

Vamos a graficar los resultados para ver si el entrenamiento funcionó.

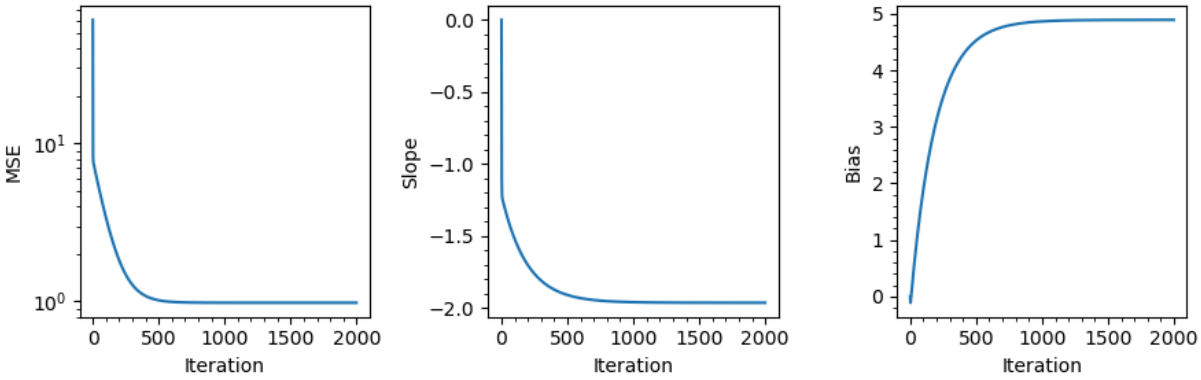
- El panel izquierdo debe mostrar la pérdida MSE, la cual debería disminuir en cada paso de entrenamiento.
- Los siguientes dos paneles muestran el valor de la pendiente y la intersección con el eje y en cada paso de entrenamiento.

Estos valores deberían acercarse asintóticamente a aproximadamente -2 y 5, respectivamente.

Si esto no ocurre, debe depurar el código de `optimize` anterior.

In [9]:
```
plt.figure(figsize=(9, 3))

plt.subplot(1, 3, 1)
plt.plot(costs)
plt.xlabel('Iteration')
plt.ylabel('MSE')
plt.yscale('log')
plt.minorticks_on()

plt.subplot(1, 3, 2)
plt.plot(slopes)
plt.xlabel('Iteration')
plt.ylabel('Slope')
plt.minorticks_on()

plt.subplot(1, 3, 3)
plt.plot(biases)
plt.xlabel('Iteration')
plt.ylabel('Bias')
plt.minorticks_on()
```

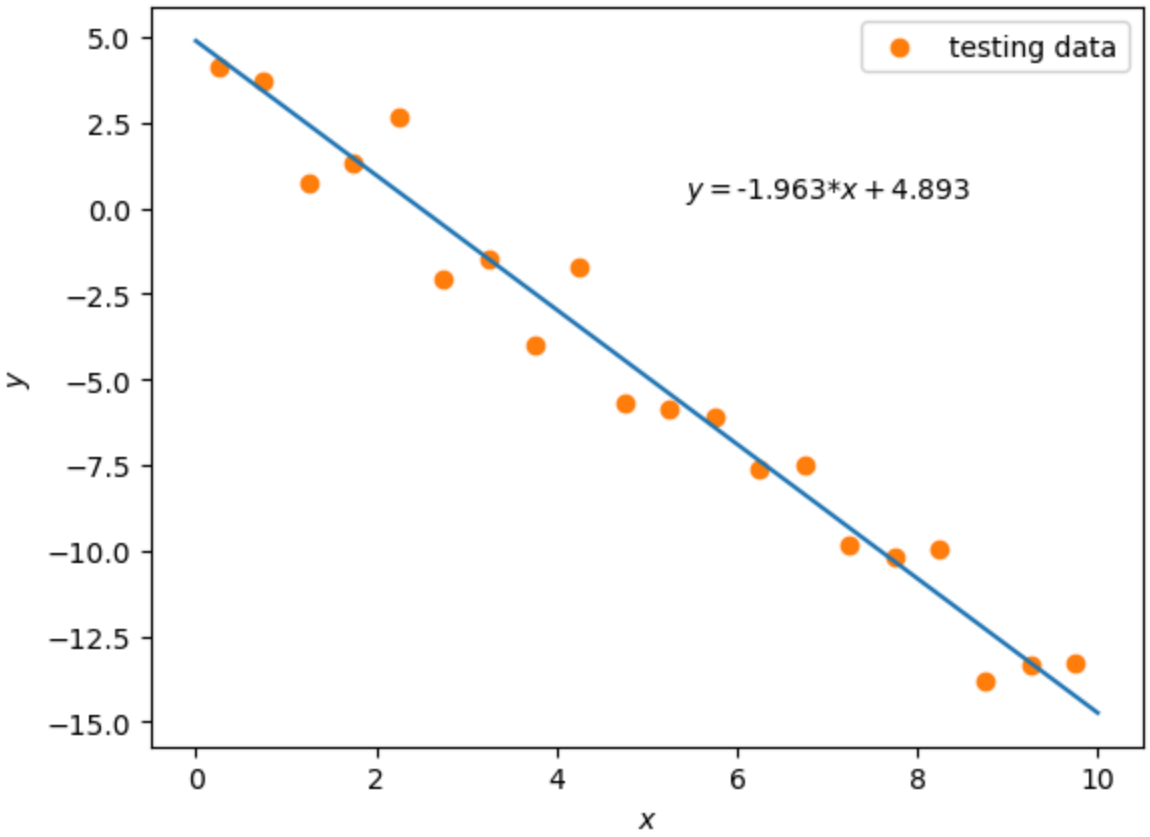
```
plt.tight_layout()
```



1.6 Resultados

Ahora grafique los resultados de las predicciones comparados con los valores de prueba (test)

In [10]:
```
xrange = np.linspace(0, 10)
y_pred = xrange * params['m'] + params['b']
plt.plot(xrange, y_pred)
plt.scatter(x_test, y_test, color='C1', label='testing data')
plt.text(7., 1, r'$y=$' + '{0:.3f}'.format(params['m']) + r'$*x + $' + '{0:.3f}'.fo
         ha='center', va='top')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend()
plt.show()
```



Observa que la pérdida es menor para el conjunto de entrenamiento. Esto se debe a que nuestro modelo se entrena para minimizar la pérdida en esos datos. Sin embargo, las pérdidas siguen siendo similares, por lo que no parece que hayamos sobreajustado los datos.

1.6 Ejercicios de Tarea

1. ¿Cuál es el efecto de la tasa de aprendizaje? Intenta variar la tasa de aprendizaje entre 10^{-3} y 1. ¿Qué sucede cuando la tasa de aprendizaje es grande? ¿Por qué? La tasa de aprendizaje tiene un efecto crucial en el comportamiento del descenso por gradiente. Si es muy pequeña (por ejemplo, 10^{-3}, el algoritmo converge de forma extremadamente lenta y requiere muchas iteraciones para acercarse al mínimo. Con un valor moderado (como 10^{-2}), la convergencia es más rápida y estable. Sin embargo, si la tasa de aprendizaje es grande (del orden de 1 o más), el algoritmo puede oscilar alrededor del mínimo o incluso divergir, haciendo que la pérdida aumente en lugar de disminuir. Esto ocurre porque en cada actualización se modifican los parámetros m y b en la dirección negativa del gradiente, y un valor demasiado grande provoca saltos que sobrepasan el mínimo, mientras que un valor demasiado pequeño da pasos microscópicos y retrasa el aprendizaje. Por ello, lo ideal es elegir un punto intermedio, lo cual normalmente se determina probando varios valores y observando la evolución de la curva de pérdida.

2. ¿Cómo cambiaría el procedimiento si los datos tuvieran forma curva? Carga los archivos `../data/tutorial_1_data/linear_regression_curved_training.npy` y

`../data/tutorial_1_data/linear_regression_curved_test.npy` y encuentra una función que ajuste la curva.

En ese caso hay que modificar el modelo:

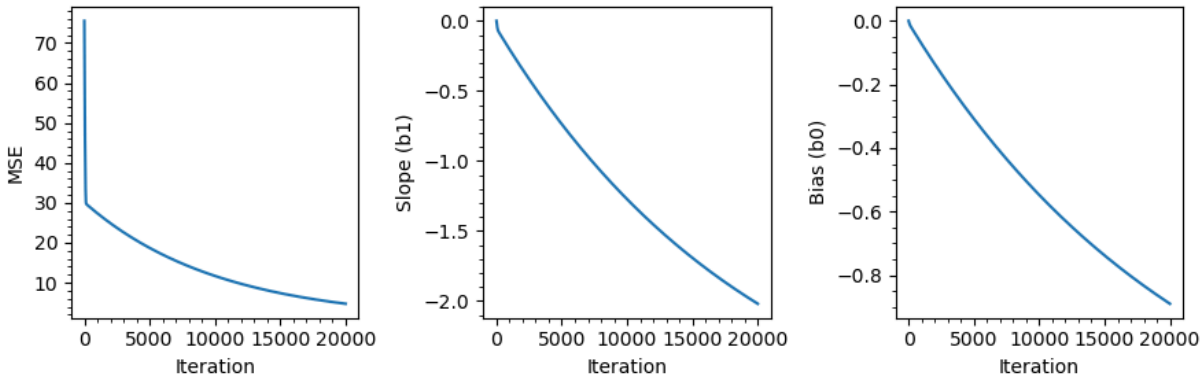
Una opción básica: regresión polinómica, es decir, añadir potencias de x como nuevas variables.

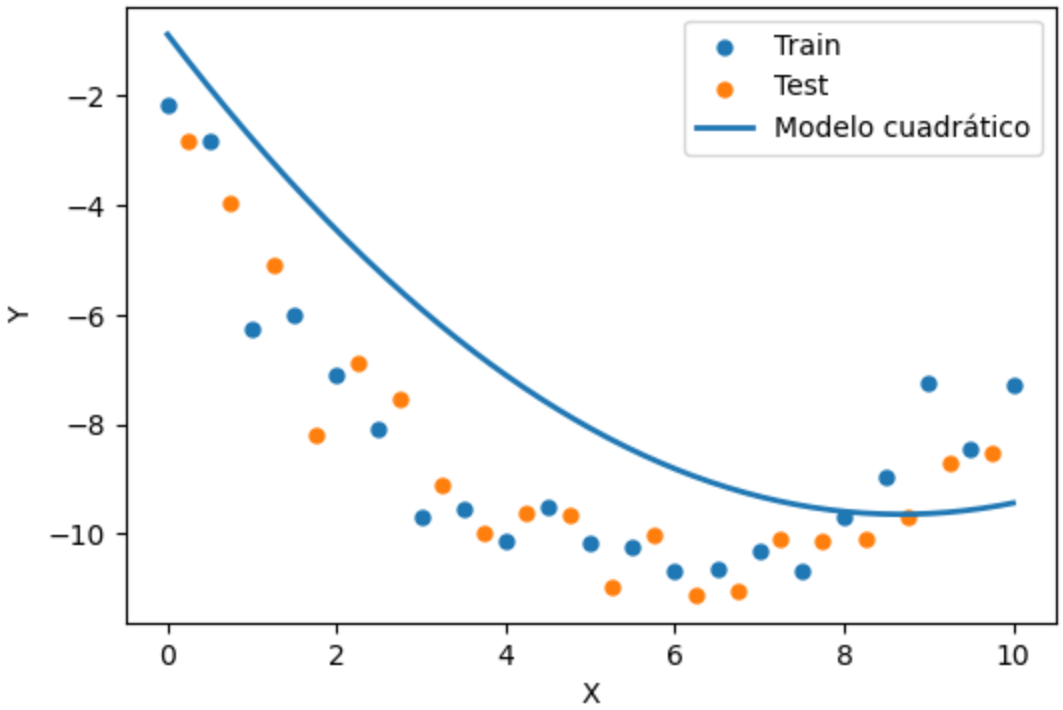
Para probarlo con los datos Como experimento mental adicional, este ejemplo y el ejemplo curvo solo tienen una pieza de información de entrada (es decir, x). ¿Cómo generalizaríamos el esquema a más de 1 variable de entrada? Ahora, cada 'evento' x_i es en realidad un vector con j dimensiones. Cambiaré la notación y escribiré todo el conjunto de datos como X^i_j, donde los componentes inferiores representan el vector de información, y el componente superior representa los diferentes 'eventos' o puntos de datos. Desarrollaremos esto más en la siguiente sección, cuando exploremos la regresión logística.

In [11]:
```
X_train = np.load("C:/Users/diana/Documents/tutorial_1_data/linear_regression_curve
Y_train = X_train[:, 1].astype(float)
X_train = X_train[:, 0].astype(float)

X_test = np.load("C:/Users/diana/Documents/tutorial_1_data/linear_regression_curved
Y_test = X_test[:, 1].astype(float)
X_test = X_test[:, 0].astype(float)

def propagate(b0, b1, b2, X, Y):
    y_pred = b0 + b1*X + b2*(X**2)
    loss = np.mean((y_pred - Y)**2)
    db0 = 2 * np.mean(y_pred - Y)
    db1 = 2 * np.mean((y_pred - Y) * X)
    db2 = 2 * np.mean((y_pred - Y) * (X**2))
    grads = {'db0': float(db0), 'db1': float(db1), 'db2': float(db2)}
    return grads, float(loss)

def optimize(b0, b1, b2, X, Y, num_iterations, learning_rate, print_loss=False, cli
    losss = []
    slopes = []
    biases = []
    for i in range(num_iterations):
        slopes.append(b1)
        biases.append(b0)
        grads, loss = propagate(b0, b1, b2, X, Y)
        if clip is not None:
            for k in grads:
                if grads[k] > clip: grads[k] = clip
                elif grads[k] < -clip: grads[k] = -clip
        db0 = grads['db0']; db1 = grads['db1']; db2 = grads['db2']
        b1 = b1 - learning_rate * db1
        b0 = b0 - learning_rate * db0
        b2 = b2 - learning_rate * db2
        losss.append(loss)
        if print_loss and i % 500 == 0:
            print('loss after iteration %i: %f' %(i, loss))
```

```
    params = {'b0': b0, 'b1': b1, 'b2': b2}
    grads = {'db0': db0, 'db1': db1, 'db2': db2}
    return params, grads, losss, slopes, biases

params, grads, costs, slopes, biases = optimize(
    b0=0.0, b1=0.0, b2=0.0,
    X=X_train, Y=Y_train,
    num_iterations=20000,
    learning_rate=1e-5,
    print_loss=True,
    clip=100.0
)

plt.figure(figsize=(9, 3))
plt.subplot(1, 3, 1)
plt.plot(costs)
plt.xlabel('Iteration')
plt.ylabel('MSE')
plt.minorticks_on()

plt.subplot(1, 3, 2)
plt.plot(slopes)
plt.xlabel('Iteration')
plt.ylabel('Slope (b1)')
plt.minorticks_on()

plt.subplot(1, 3, 3)
plt.plot(biases)
plt.xlabel('Iteration')
plt.ylabel('Bias (b0)')
plt.minorticks_on()
plt.tight_layout()
plt.show()

x_plot = np.linspace(X_train.min(), X_train.max(), 300)
y_plot = params['b0'] + params['b1']*x_plot + params['b2']*(x_plot**2)

plt.figure(figsize=(6,4))
plt.scatter(X_train, Y_train, s=25, label='Train')
plt.scatter(X_test, Y_test, s=25, label='Test')
plt.plot(x_plot, y_plot, linewidth=2, label='Modelo cuadrático')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```

```
loss after iteration 0: 75.559046
loss after iteration 500: 28.714136
loss after iteration 1000: 27.373347
loss after iteration 1500: 26.096314
loss after iteration 2000: 24.880006
loss after iteration 2500: 23.721532
loss after iteration 3000: 22.618144
loss after iteration 3500: 21.567220
loss after iteration 4000: 20.566265
loss after iteration 4500: 19.612902
loss after iteration 5000: 18.704867
loss after iteration 5500: 17.840005
loss after iteration 6000: 17.016262
loss after iteration 6500: 16.231682
loss after iteration 7000: 15.484401
loss after iteration 7500: 14.772647
loss after iteration 8000: 14.094729
loss after iteration 8500: 13.449038
loss after iteration 9000: 12.834041
loss after iteration 9500: 12.248278
loss after iteration 10000: 11.690358
loss after iteration 10500: 11.158957
loss after iteration 11000: 10.652814
loss after iteration 11500: 10.170727
loss after iteration 12000: 9.711552
loss after iteration 12500: 9.274200
loss after iteration 13000: 8.857632
loss after iteration 13500: 8.460859
loss after iteration 14000: 8.082941
loss after iteration 14500: 7.722980
loss after iteration 15000: 7.380122
loss after iteration 15500: 7.053554
loss after iteration 16000: 6.742501
loss after iteration 16500: 6.446225
loss after iteration 17000: 6.164023
loss after iteration 17500: 5.895226
loss after iteration 18000: 5.639196
loss after iteration 18500: 5.395326
loss after iteration 19000: 5.163038
loss after iteration 19500: 4.941780
```





2. Regresión lineal - método analítico

2.1 Cargar y visualizar los datos

In [12]:
```
import pandas as pd
data = pd.read_csv('C:/Users/diana/Documents/linear_data.csv')
X = data['X']
Y = data['Y']
```

2.2 Modelo lineal:

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

Ejercicio: Demuestre que:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\,\bar{y}}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2}$$

Ejercicio: Complete la siguiente función para obtener los estimadores de β_0, σ y β_1.

In [13]:
```
def simple_linear_regression(x, y):
    """
    Estima los parámetros de una regresión lineal simple (y = β0 + β1*x + error).

    Parámetros:
    x -- array-like, variable independiente
    y -- array-like, variable dependiente
```

```
    Retorna:
    beta_0_estimador -- intersección (ordenada al origen)
    beta_1_estimador -- pendiente (coeficiente de regresión)
    sigma_estimador  -- estimación de la desviación estándar de los errores

    """
    # convertir listas de entrada a arrays de numpy
    x = np.array(x)
    y = np.array(y)

    # calcular promedios de x, y
    x_mean = np.mean(x)
    y_mean = np.mean(y)

    # calcular numerador y denominador para beta_1
    numerator = np.sum((x - x_mean) * (y - y_mean))
    denominator = np.sum((x - x_mean) ** 2)

    # Calcular beta_1 (pendiente)
    beta_1_estimador = numerator / denominator

    # Calcular beta_0 (intersección)
    beta_0_estimador = y_mean - beta_1_estimador * x_mean

    # Calcular valores predichos
    y_hat = beta_0_estimador + beta_1_estimador * x

    # Calcular residuos
    residuals = y - y_hat

    # Estimar sigma (desviación estándar de los errores)
    n = len(x)
    sigma_estimador = np.sqrt(np.sum(residuals ** 2) / (n - 2))

    return beta_0_estimador, beta_1_estimador, sigma_estimador
```

Pruebe su función

In [14]:
```
beta_0_estimador, beta_1_estimador, sigma_estimador = simple_linear_regression(X, Y
print(f"Estimated beta_0 (intercept): {beta_0_estimador}")
print(f"Estimated beta_1 (slope): {beta_1_estimador}")
print(f"Estimated sigma (error): {sigma_estimador}")
```

```
Estimated beta_0 (intercept): 5.260594723979204
Estimated beta_1 (slope): -2.011575418787928
Estimated sigma (error): 3.977648904150111
```