



**Tecnológico
de Monterrey**

**Inteligencia artificial avanzada para la ciencia de
datos I (Gpo 101)**

Momento de Retroalimentación: Módulo 2

**Implementación de una técnica de aprendizaje
máquina sin el uso de un framework. (Portafolio
Implementación)**

**Uso de framework o biblioteca de aprendizaje
máquina para la implementación de una solución.
(Portafolio Implementación)**

Maxime Vilcocq Parra

A01710550

Contenido

Implementación de una técnica de aprendizaje máquina sin el uso de un framework.	3
Gradient descent:	3
Elaboración de código sin framework:	5
Train, validate y test:	12
Cómo evaluamos el modelo:	12
En código:	13
Estandarización	16
Resultados:.....	18
Resultados y comparación del Modelo:	19
Mejora:	20
¿Qué son los hiperparámetros?.....	20
Resultados de mejora:.....	21
Uso de framework o biblioteca de aprendizaje máquina para la implementación de una solución.	22
Pytorch:	22

Implementación de una técnica de aprendizaje máquina sin el uso de un framework.

Implementación de una técnica de aprendizaje máquina sin el uso de un framework:

Para esta entrega estaré programando el algoritmo de gradient descent para generar un modelo predictivo (calculando coeficientes de distintas variables) reduciendo el MSE (mean squared error).

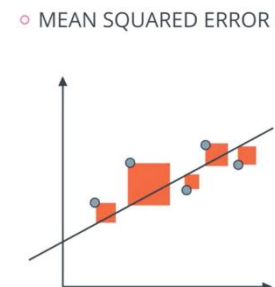
Para mi programa estaré utilizando el dataset de [abalone](#), el cual es un set de datos para predecir la edad de este tipo de árbol a partir de medidas físicas (en lugar del tedioso método tradicional). Sin embargo, el código debe de funcionar con otros sets de datos siempre y cuando se introduzcan de forma correcta.

Gradient descent:

El algoritmo de gradient descent nos permite crear un modelo predictivo al reducir el error (MSE) de un modelo predictivo inicial (propuesto por nosotros) hasta que ese error sea el más bajo posible.

Nota: en ocasiones no es posible llegar a un MSE de 0, esto se puede deber a tener datos insuficientes (ya sea que falten considerar variables, o hacer más mediciones). Cuando esto sucede es probable que no conozcamos uno o más factores que impactan a nuestra variable dependiente.

La siguiente imagen del blog de Krystian Safjan ("[Comprehensive Guide to Interpreting \$R^2\$, MSE, and RMSE for Regression Models.](#)"). Muestra visualmente como se ve este MSE. La línea recta es un modelo de predicción y mientras más cerca estén los puntos (valores reales de la variable independiente), menor será el área naranja (MSE).



De forma matemática el mean squared error se calcula de la siguiente manera (imagenes sacada del video [Machine Learning Tutorial Python - 4:](#)

Gradient Descent and Cost Function):

Mean Squared Error

$$mse = \frac{1}{n} \sum_{i=1}^n (y_i - y_{predicted})^2$$
$$mse = \frac{1}{n} \sum_{i=1}^n (y_i - (mx_i + b))^2$$

Cost Function

Esto es de forma literal, restarle cada Y que predice nuestro modelo a cada Y real del set de datos, elevar cada resta al cuadrado (evita que los valores negativos cancelen a los positivos), sumar los resultados y promediar el total de dicha suma.

Lo que queremos es ver cómo cada una de nuestros coeficientes “m” y nuestro bias “b” afectan este mean squared error. Para ello derivamos MSE con respecto a cada “m” y al “b”.

$$\partial/\partial m = \frac{2}{n} \sum_{i=1}^n -x_i (y_i - (mx_i + b))$$

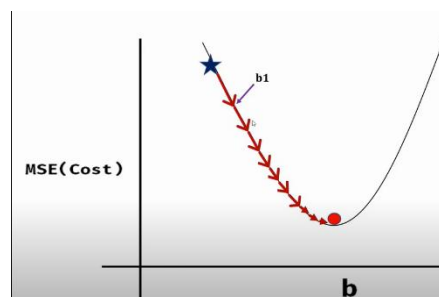
$$\partial/\partial b = \frac{2}{n} \sum_{i=1}^n -(y_i - (mx_i + b))$$

Imagen sacada del video [Machine Learning Tutorial Python - 4: Gradient Descent and Cost Function](#).

Esta derivada nos dice la dirección a la que cada bias y coeficiente está moviendo nuestro MSE, la idea es irnos moviendo hacia el MSE más pequeño posible. Esto se consigue utilizando un step (que es lo mismo que el learning rate), que nos dice qué tanto movernos hacia la dirección que hace que el MSE disminuya.

$$m = m - \text{learning rate} * \partial/\partial m$$

$$b = b - \text{learning rate} * \partial/\partial b$$



Imágenes tomadas del video [Machine Learning Tutorial Python - 4: Gradient Descent and Cost Function](#).

En la imagen anterior podemos ver como el valor de b se va actualizando en cada iteración obteniendo un MSE cada vez más pequeño.

El algoritmo de gradient descent suele requerir de varias épocas o iteraciones para llegar a un buen resultado.

Elaboración de código sin framework:

Estoy empleando numpy simplemente para manejar los datos en matrices y facilitar el manejo de estos para hacer producto punto entre estas. En otras palabras, para poder multiplicar cada coeficiente de nuestras variables independientes por su respectivo valor y con eso obtener una predicción. Si el código de Python no funciona al ejecutarse, es probable que falte hacer pip install de numpy.

1. El primer paso es identificar y separar nuestra variable dependiente (Y) de las variables independientes (nuestras X) y cargarlas.

En el caso del dataset de abalone, el valor que queremos conseguir es el número de anillos que tiene el árbol (con eso se determina la edad). Además el dataset incluye la columna de Sex que es una variable no cuantitativa por lo que se requiere hacer un hot encoding para poder procesar el dato.

Carga el archivo:

```
def load_data(filename="abalone.data"):
    data = []
    with open(filename, "r") as f:
        for line in f:
            #separar valores por comas
            values = line.strip().split(",")
            data.append(values)
    #convertir las líneas de datos en un array
    return np.array(data)
```

Convierte de cualitativo a cuantitativo:

```
# Hot encoding para "Sex" (Male, Female, Infant)
def encode_sex(data):
    encoded = []
    for row in data:
        sex = row[0] #sex es el primer valor de la fila
        if sex == "M":
            encoded.append([1, 0, 0]) # M
        elif sex == "F":
            encoded.append([0, 1, 0]) # F
        else:
            encoded.append([0, 0, 1]) # I

    return np.array(encoded)
```

Separa la variable dependiente y junta todas las independientes en una sola matriz:

```
# Separar variables independientes (X) y dependiente (y)
def preprocess_data(raw_data):
    # Separar columna sex (cualitativa) y el resto
    sex_encoded = encode_sex(raw_data)

    # Convertir todo a float para calculos
    numeric_data = raw_data[:, 1:].astype(float)

    # Nuestra x son las independientes sin el numero de rings (y)
    #hstack permite juntar las 3 columnas del sex_encoded con el resto
    X = np.hstack((sex_encoded, numeric_data[:, :-1]))

    # Variable dependiente = rings
    y = numeric_data[:, -1]
    return X, y
```

2. Darles un valor inicial a nuestros pesos (los coeficientes de cada una de nuestras variables independientes) y a nuestro bias.

Para esto estoy creando un vector de coeficientes, un coeficiente por cada variable independiente, todos inicializados en 0.

```
n_samples, n_features = X.shape # filas, columnas

# Inicializar coeficientes (w) y bias (b)
w = np.zeros(n_features)
b = 0
```

n_samples son las filas y n_features, las columnas.

3. Saco el cálculo de la predicción con el producto punto.

```
# Predicción del modelo
y_pred = np.dot(X, w) + b #  $X \cdot w + b$  ( $mx_i + b$ )
```

En esta línea se está multiplicando cada fila de X (cada fila tiene n columnas, una por cada variable independiente) por su respectivo coeficiente dentro del vector de pesos w y luego le suma b.

Lo que obtiene la variable y_pred es un arreglo de predicciones del mismo tamaño que las filas de nuestro set de datos. Es decir, si tenemos 15 filas en el dataset, habrá 15 predicciones dentro de y_pred.

4. Sacamos el MSE con la predicción recién calculada:

```
# Calcular costo (MSE)
cost = (1/n_samples) * np.sum((y - y_pred) ** 2)
```

$$mse = \frac{1}{n} \sum_{i=1}^n (y_i - y_{predicted})^2$$

A cada valor real de y se le resta el valor de la predicción y se eleva al cuadrado (esto nos deja un arreglo de resultados). Con np.sum sumamos cada valor del arreglo para tener un solo valor numérico el cual se divide entre el número de filas para saber el promedio (MSE).

5. Sacamos un vector de derivadas para saber cuánto y cómo (+/-) influye cada variable independiente y el bias a nuestra variable dependiente.

```
# Derivadas parciales
dw = -(2/n_samples) * np.dot(X.T, (y - y_pred)) # vector de derivadas
db = -(2/n_samples) * np.sum(y - y_pred)
```

$$\partial/\partial m = \frac{2}{n} \sum_{i=1}^n -x_i (y_i - (mx_i + b))$$

$$\partial/\partial b = \frac{2}{n} \sum_{i=1}^n -(y_i - (mx_i + b))$$

dw nos muestra un vector (usa np.dot) con la magnitud y signo de la derivada respecto a cada variable independiente. Db (usa.sum) nos da únicamente un valor el cual representa el impacto del bias en el MSE.

- Utilizando un learning rate actualizamos el valor de nuestro bias y nuestros coeficientes.

```
def gradient_descent(X, y, learning_rate=0.001, iterations=25000):
```

Aquí uso un learning rate default de .001.

```
# Actualizar pesos
w -= learning_rate * dw
b -= learning_rate * db
```

$$m = m - \text{learning rate} * \partial/\partial m$$

$$b = b - \text{learning rate} * \partial/\partial b$$

- Finalmente repetimos este proceso las veces que queramos actualizando cada vez los valores del bias y los pesos.

```
def gradient_descent(X, y, learning_rate=0.001, iterations=25000):
```

El default está colocado en 25000.

```
for i in range(iterations):
    # Prediccion del modelo
    y_pred = np.dot(X, w) + b # X*w + b

    # Calcular costo (MSE)
    cost = (1/n_samples) * np.sum((y - y_pred) ** 2)

    # Derivadas parciales
    dw = -(2/n_samples) * np.dot(X.T, (y - y_pred)) # vector de derivadas
    db = -(2/n_samples) * np.sum(y - y_pred)

    # Actualizar pesos
    w -= learning_rate * dw
    b -= learning_rate * db

    # Mostrar progreso cada cierto número de iteraciones
    if i % 500 == 0:
        print(f"Iteración {i}: w={w}, b={b:.4f}, cost={cost:.4f}")
```


8. Una vez concluidas las épocas regresamos el vector de pesos y el bias final.

```
return w, b
```

```
w, b = gradient_descent(X, y, learning_rate=0.001, iterations=20000)
print("Pesos finales:", w)
print("Bias final:", b)
```

En la foto anterior uso “X” y “y” pero en la aplicación real uso el “X” y “y” de mi dataset de entrenamiento.

Ya que tenemos los valores podemos comenzar a hacer predicciones.

```
# Predecir con los coeficientes obtenidos
def predict(X, w, b):
    return np.dot(X, w) + b
```

9. Crear un modelo funcional.

Para ver que tanto se ajusta a nuevos valores, separamos nuestro dataset en dos, uno para entrenar y otro para probar.

```
# Separar en entrenamiento (80%) y test (20%)
def train_test_split(X, y, test_size=0.2, seed=42):
    np.random.seed(seed)
    n_samples = X.shape[0]
    indices = np.arange(n_samples)
    np.random.shuffle(indices)

    test_size = int(n_samples * test_size)
    test_idx = indices[:test_size]
    train_idx = indices[test_size:]

    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]

    return X_train, X_test, y_train, y_test
```

El código funciona igual solo que en lugar de enviar toda el dataset, enviamos solamente la X y Y de entrenamiento para obtener coeficientes. Después con los coeficientes obtenidos y los valores de X y Y de test podemos comparar los valores que predecimos con valores reales.

10. Predecir y comparar

Con los coeficientes obtenidos y el segmento de prueba comparamos los valores reales con los que pronosticamos.

Estoy utilizando R squared,

$$R^2 = 1 - \frac{\sum_{i=1}^n (\hat{y}_i - y_i)^2}{\sum_{i=1}^n (y_i - \bar{y}_i)^2}$$

Imagen de [R-squared in Linear Regression Models: Concepts, Examples.](#)

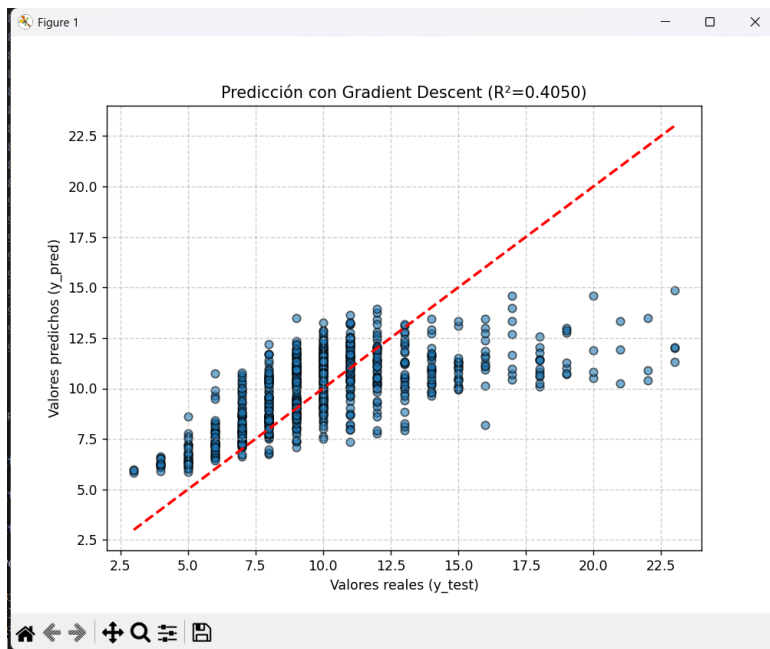
```
# Calcular R squared
def r_squared(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    return 1 - (ss_res / ss_tot)
```

Este valor nos sirve para explicar qué tanto nuestro modelo explica la variación de nuestra variable dependiente. Va de 0 a 1, donde un 1 significa que el modelo es perfecto.

11. Graficar los resultados.

Utilicé matplotlib para mostrar un visual que nos permite entender más fácilmente nuestra predicción contra los valores reales.

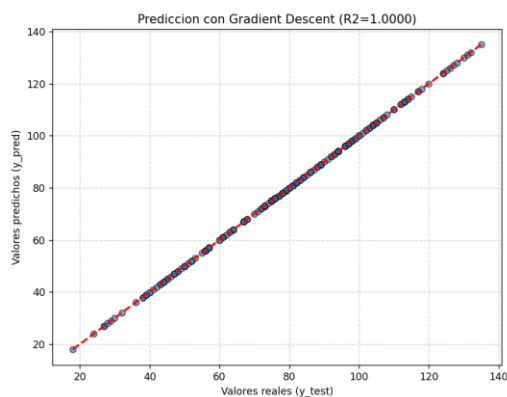
```
plt.figure(figsize=(8,6))
plt.scatter(y_test, y_pred_test, alpha=0.6, edgecolor="k")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', linewidth=2)
plt.xlabel("Valores reales (y_test)")
plt.ylabel("Valores predichos (y_pred)")
plt.title(f"Prediccion con Gradient Descent (R2={r2:.4f})")
plt.grid(True, linestyle="--", alpha=0.6)
plt.show()
```



12. Comprobar los datos y el funcionamiento del código:

Tras ejecutar mi código con el set de datos de abalone, el resultado obtenido es un R^2 de .4 lo que significa que mi modelo predice un alrededor de un 40% de la variabilidad. Este valor no es muy alto, esto probablemente se debe a que estamos tratando de hacer una regresión lineal y que otro modelo de regresión (por ejemplo, un exponencial) podría ajustarse mejor al dataset.

Sin embargo, esto no significa que el código no funciona, para demostrar esto creé un dataset artificial con valores estrictamente lineares. Podemos verificar si el código funciona si vemos que los coeficientes son iguales a los que utilicé. Por ser un set de datos artificial, se puede alcanzar un MSE de casi 0, y una R^2 de casi 1.



```
Iteración 19500: w=[2.00388771 3.00397674 5.00416822 7.00420067], b=3.9201, cost=0.0006
Pesos finales: [2.00354048 3.00362156 5.00379594 7.00382549]
Bias final: 3.927264157969956
R2 en test set: 1.0000
```

```
np.random.seed(42)
X = np.random.randint(0, 10, (1000, 4))
y = 2*X[:,0] + 3*X[:,1] + 5*X[:,2] + 7*X[:,3] + 4
```

Train, validate y test:

Actualmente, el código separa el dataset en dos partes. Una para entrenar y otra para probar el modelo entrenado. El punto de esto es que los valores con los que se prueba no afecten el entrenamiento (overfitting). Sin embargo, esto no es suficiente para confiar en que nuestro modelo funcionará correctamente cuando se enfrente a nuevos datos. Por ello, agregamos una capa extra llamada validation.

Validation no es más que un test adicional. A nuestro set de entrenamiento le quitamos un fragmento para realizar una prueba adicional. Esto nos deja con un set de entrenamiento ligeramente más pequeño, pero nos da la posibilidad de probar con un set de datos adicional. Tras comparar los valores de R^2 que se obtienen de probar dentro de training, de validate y de test podemos ver si nuestro modelo realmente funciona o si se está sobre ajustando. Además, podemos comparar como evoluciona el MSE de los sets de validate y de train en cada iteración para identificar si hay un underfit o un overfit. Underfit si ambas líneas se encuentran muy altas y overfit si ambas gráficas son muy distintas.

Cómo evaluamos el modelo:

Bias: Qué tanto aprende el modelo (que tanta variabilidad es capaz de modelar). Si el R^2 es alto $>.7$, entonces el bias es bajo. Es decir, el modelo aprende bien los patrones presentes en el data set. Si el $R^2 < .3$ entonces el bias es alto y por ello no aprende bien los patrones. Un punto entre $.3$ y $.7$ es un aprendizaje limitado.

Varianza: Qué tan sensible es el modelo a los datos de entrenamiento. Si la varianza es muy alta, significa que el modelo se está “memorizando” los datos de entrenamiento. En este problema estoy utilizando una diferencia de R^2 entre un set de pruebas y otro para calcularla. Si la diferencia de R^2 es grande, es porque se está aprendiendo los valores de entrenamiento y no es capaz de hacer una buena predicción con nuevos valores.

Ajuste: Overfitting, underfitting y fit. Si el bias es alto es underfit (no captura el patrón de comportamiento), si el bias es bajo, pero la varianza alta es overfit (se memoriza datos de entrenamiento y falla con datos externos). Finalmente un modelo fit, tendría un bias de mediano a bajo (es mejor si es bajo) y una varianza baja (es bueno para generalizar).

En código:

```
# Separar en entrenamiento (60%) y test (20%) y validacion 20%
def train_val_test_split(X, y, val_size=0.2, test_size=0.2, seed=42):
    np.random.seed(seed)
    n_samples = X.shape[0]
    indices = np.arange(n_samples)
    np.random.shuffle(indices)

    test_split = int(n_samples * test_size)
    val_split = int(n_samples * (test_size + val_size))

    test_idx = indices[:test_split]
    val_idx = indices[test_split:val_split]
    train_idx = indices[val_split:]

    X_train, X_val, X_test = X[train_idx], X[val_idx], X[test_idx]
    y_train, y_val, y_test = y[train_idx], y[val_idx], y[test_idx]

    return X_train, X_val, X_test, y_train, y_val, y_test
```

Hacemos una separación adicional para validación.

```

# Diagnostico del modelo basado en R2 de train y val
def diagnostico_modelo(y_train, y_train_pred, y_val, y_val_pred):
    r2_train = r_squared(y_train, y_train_pred)
    r2_val = r_squared(y_val, y_val_pred)

    # Bias
    if r2_train < 0.3:
        bias = "Alto (modelo no aprende bien)"
    elif r2_train < 0.7:
        bias = "Medio (modelo mejora pero sigue limitado)"
    else:
        bias = "Bajo (modelo aprende bien)"

    # Varianza
    diff = abs(r2_train - r2_val)

    if diff > 0.2:
        varianza = "Alta (cambia mucho entre train y val)"
    elif diff > 0.1:
        varianza = "Media (algo de diferencia)"
    else:
        varianza = "Baja (modelo consistente)"

    # Ajuste
    if r2_train < 0.3:
        ajuste = "Underfit (el modelo no captura los patrones)"
    elif r2_train >= 0.7 and diff > 0.2:
        ajuste = "Overfit (memoriza train pero falla en val)"
    else:
        ajuste = "Fit (buen balance entre bias y varianza)"

    return {
        "R2_train": r2_train,
        "R2_val": r2_val,
        "Bias": bias,
        "Varianza": varianza,
        "Ajuste": ajuste
    }

```

Calculamos la R2 de nuestra predicción en validación y en test y con eso calculamos el Bias, la varianza y el ajuste (fit).

```

def run_Abalone_Validation():
    raw_data = load_data("abalone.data")
    X, y = preprocess_data(raw_data)

    # usar el split con validación
    X_train, X_val, X_test, y_train, y_val, y_test = train_val_test_split(X, y)

    # entrenar el modelo
    w, b = gradient_descent(X_train, y_train, learning_rate=0.001, iterations=20000)

    # predicciones
    y_train_pred = predict(X_train, w, b)
    y_val_pred = predict(X_val, w, b)
    y_test_pred = predict(X_test, w, b)

    # diagnóstico
    diag = diagnostico_modelo(y_train, y_train_pred, y_val, y_val_pred)
    print("\n--- Diagnóstico del modelo ---")
    for k, v in diag.items():
        print(f"{k}: {v}")

    # desempeño en test final
    r2_test = r_squared(y_test, y_test_pred)
    print(f"\nr2 en test set (generalización): {r2_test:.4f}")

    return w, b, (y_train, y_train_pred), (y_val, y_val_pred), (y_test, y_test_pred)

```

Separamos los datos, hacemos predicciones para train, validate y test y corremos el diagnóstico para saber como se comporta el modelo dentro de train a comparación de validate.

Estandarización

Para mejorar la R2 de mi modelo estoy haciendo una estandarización z (z-Standardization).

A cada valor de x le resto la media de ese valor y lo divido entre la desviación estándar del mismo.

Este tipo de estandarización es útil para mi data set ya que permite hacer que variables medidas de distintas formas sean comparables (por ejemplo milímetros contra gramos [situación real del data set de Abalone]).

En pocas palabras lo que hace esta normalización es cambiar la magnitud de los valores por una medida de qué tan diferente se comportó una variable a su promedio. Esto hace que variables con magnitudes grandes y pequeñas tengan la misma importancia para el modelo.

Esto no cambia la relación que existe entre las variables dependientes e independientes, simplemente cambia como el modelo ve la magnitud de cada variable (una magnitud grande puede afectar el step del gradiente, haciendo que una variable sea “priorizada” sobre otra).

En código:

```
# Función de estandarización
def standardize(X_train, X_val=None, X_test=None):
    mean = X_train.mean(axis=0)
    std = X_train.std(axis=0)
    X_train_scaled = (X_train - mean) / std

    results = [X_train_scaled]
    if X_val is not None:
        X_val_scaled = (X_val - mean) / std
        results.append(X_val_scaled)
    if X_test is not None:
        X_test_scaled = (X_test - mean) / std
        results.append(X_test_scaled)
    return tuple(results)
```

Hacemos una función para estandarizar los valores de X.


```

def run_Abalone_Validation_Standardized():
    raw_data = load_data("abalone.data")
    X, y = preprocess_data(raw_data)

    # Split train/val/test
    X_train, X_val, X_test, y_train, y_val, y_test = train_val_test_split(X, y)

    # Estandarizar los datos
    X_train, X_val, X_test = standardize(X_train, X_val, X_test)

    # Entrenar modelo sin regularización
    w, b = gradient_descent(
        X_train, y_train,
        learning_rate=0.001,
        iterations=20000
    )

    # Predicciones
    y_train_pred = predict(X_train, w, b)
    y_val_pred = predict(X_val, w, b)
    y_test_pred = predict(X_test, w, b)

    # Diagnóstico
    diag = diagnostico_modelo(y_train, y_train_pred, y_val, y_val_pred)
    print("\n--- Diagnóstico del modelo ESTANDARIZADO ---")
    for k, v in diag.items():
        print(f"{k}: {v}")

    # R² en test
    r2_test = r_squared(y_test, y_test_pred)
    print(f"\nR2 en test set (generalización): {r2_test:.4f}")

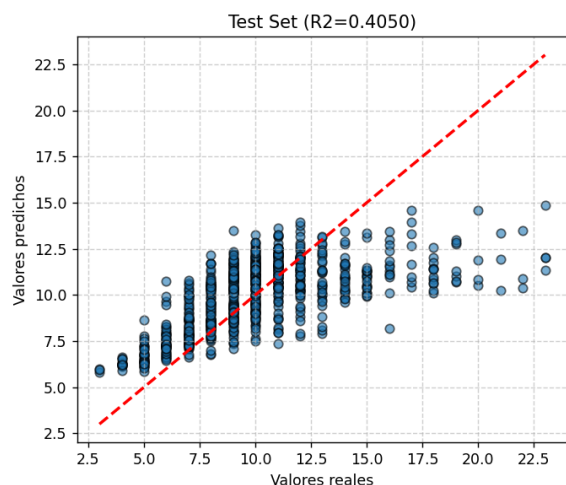
    return w, b, (y_train, y_train_pred), (y_val, y_val_pred), (y_test, y_test_pred)

```

Le aplicamos la estandarización a las X (variables independientes).

Resultados:

Test y train solamente:



Si corremos el código únicamente con un set de entrenamiento y otro de prueba, obtenemos una R^2 de .405. Sin embargo, como solo hay un set para probar, no sabemos realmente si estamos cayendo en un overfitting, underfitting o si está bien nuestro fit. Por ello, agregamos un nuevo segmento llamado validate para hacer pruebas.

MSE de validate vs train	MSE de validate vs train (data set estandarizado).
Como podemos ver en la gráfica el error disminuye significativamente y ambas líneas son muy parecidas por lo que no hay un underfitting ni un overfitting significativo.	Como podemos ver en la gráfica el error disminuye significativamente y ambas líneas son muy parecidas por lo que no hay un underfitting ni un overfitting significativo. Además, podemos ver que la línea de train es ligeramente más cerca a la de validation.

Resultados y comparación del Modelo:

	Train	Validate	Test
Gráfica			
R2	.3755	.3729	.4085
MSE	.6838	5.6165	6.4050
Bias	Medio (modelo limitado)	Medio (modelo limitado)	Medio (modelo limitado)
Varianza	La varianza entre los 3 modelos es muy baja (la R2 y el MSE son prácticamente iguales), esto significa que no hay una memorización del set de entrenamiento.		
Fit	El fit está entre underfit y fit. La R2 es algo baja por lo que el patrón que está aprendiendo el modelo no es capaz de representar la mayoría de la variabilidad de los datos reales. Sin embargo la varianza es muy buena ya que casi no cambia entre un set y otro.		

	Train (Estandarizado)	Val (Estandarizado)	Test (Estandarizado)
Gráfica			
R2	.5320	.4978	.5331
MSE	5.0085	4.4980	5.0556
Bias	Medio (modelo limitado)	Medio (modelo limitado)	Medio (modelo limitado)
Varianza	La varianza entre los 3 modelos es muy baja (la R2 y el MSE son prácticamente iguales), esto significa que no hay una memorización del set de entrenamiento.		
Fit	El fit está entre underfit y fit. La R2 es algo baja por lo que el patrón que está aprendiendo el modelo no es capaz de representar gran parte de la variabilidad de los datos reales. Sin embargo, estos valores son mucho mejores que el set sin estandarización. La varianza sigue siendo muy buena ya que casi no cambia entre un set y otro.		

Mejora:

Para obtener buenos resultados con el algoritmo de gradient descent, lo que se puede hacer (además de estandarizar los datos) es mover los hiperparámetros.

¿Qué son los hiperparámetros?

Son literalmente valores arbitrarios que nosotros elegimos.

- Número de epochs (iteraciones).
- Learning rate (tamaño del step).
- Pesos iniciales. (Puede hacer que converja más rápido, pero no tiene tanto impacto en una regresión lineal).
- Batch size. (Mientras más grande es más lento, pero generaliza mejor).

Para nuestro modelo no moveremos los pesos iniciales ni el batch size ya que el impacto de estos en nuestro algoritmo y con nuestro set de datos no representaría una mejora tan significativa.

		Learning rate			
		.00001	.0001	.001	.01
Epochs	10000	--- Mean Squared Error (MSE) --- Train MSE: 74.5901 Validation MSE: 69.6070 Test MSE: 73.2328 (R2=-5.7632)	--- Mean Squared Error (MSE) --- Train MSE: 8.1166 Validation MSE: 6.5489 Test MSE: 7.7539 (R2=0.2839)	--- Mean Squared Error (MSE) --- Train MSE: 5.0645 Validation MSE: 4.4932 Test MSE: 5.0876 (R2=0.5301)	--- Mean Squared Error (MSE) --- Train MSE: 4.8907 Validation MSE: 4.5057 Test MSE: 4.9199 (R2=0.5456)
	15000	--- Mean Squared Error (MSE) --- Train MSE: 62.1640 Validation MSE: 57.7751 Test MSE: 61.0048 (R2=-4.6339)	--- Mean Squared Error (MSE) --- Train MSE: 6.2523 Validation MSE: 5.1013 Test MSE: 6.0071 (R2=0.4452)	--- Mean Squared Error (MSE) --- Train MSE: 5.0283 Validation MSE: 4.4968 Test MSE: 5.0710 (R2=0.5317)	--- Mean Squared Error (MSE) --- Train MSE: 4.8767 Validation MSE: 4.5231 Test MSE: 4.9103 (R2=0.5465)
	20000	--- Mean Squared Error (MSE) --- Train MSE: 52.0767 Validation MSE: 48.1055 Test MSE: 51.0461 (R2=-3.7142)	--- Mean Squared Error (MSE) --- Train MSE: 5.8303 Validation MSE: 4.8661 Test MSE: 5.6440 (R2=0.4788)	--- Mean Squared Error (MSE) --- Train MSE: 5.0085 Validation MSE: 4.4980 Test MSE: 5.0556 (R2=0.5331)	--- Mean Squared Error (MSE) --- Train MSE: 4.8724 Validation MSE: 4.5358 Test MSE: 4.9108 (R2=0.5465)
	25000	--- Mean Squared Error (MSE) --- Train MSE: 52.0767 Validation MSE: 48.1055 Test MSE: 51.0461 (R2=-3.7142)	--- Mean Squared Error (MSE) --- Train MSE: 5.6430 Validation MSE: 4.7753 Test MSE: 5.4933 (R2=0.4927)	--- Mean Squared Error (MSE) --- Train MSE: 4.9924 Validation MSE: 4.4970 Test MSE: 5.0390 (R2=0.5346)	--- Mean Squared Error (MSE) --- Train MSE: 4.8712 Validation MSE: 4.5436 Test MSE: 4.9128 (R2=0.5463)

Aquí podemos ver cómo va evolucionando los resultados con distintos parámetros. Con un learning rate bajo, se requieren de muchísimas iteraciones por lo que no llegamos a un valor de R2 útil.

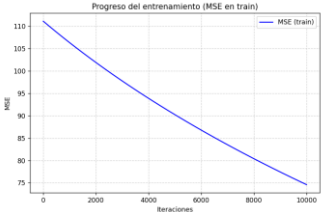
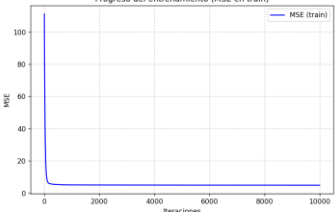
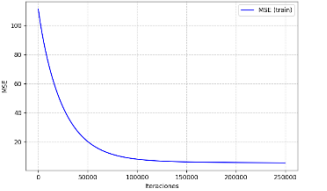
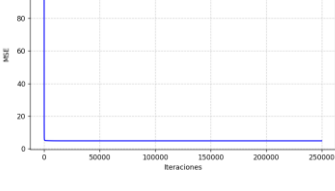
Con un learning rate más alto, el modelo puede convergir más rápido, por lo que podemos llegar en menos iteraciones a un buen resultado.

Con 15000 epochs y un learning rate (lr) de .01, podemos llegar a una R2 de .5465, en lugar de .5331 con el learning rate de .001 y 20000 iteraciones.

Por pura experimentación intenté correr 150000 épocas, el resultado dejo de cambiar en la iteración 120000 y se quedó en una R2 de .5459 (prácticamente la misma, pero tomando 8 veces más tiempo para entrenar).

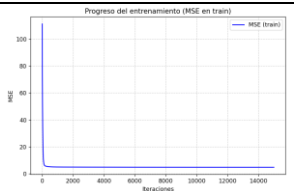
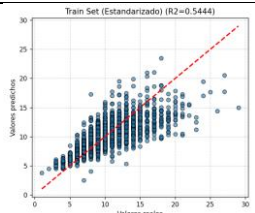
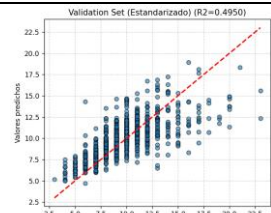
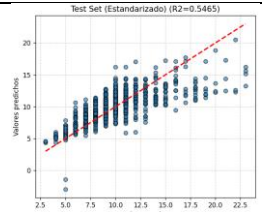
En mi caso, elegí quedarme con 15000 iteraciones y learning rate de .01, ya que el valor de R2 es muy cercano a obtenido con 150000 épocas y el mismo coeficiente de aprendizaje, y el valor de R2 es superior al obtenido con el resto de las combinaciones de epochs y lr.

Comportamiento de MSE:

	Learning rate bajo (.00001)	Learning rate alto (.01)
Pocas epochs (10000)		
Muchas epochs (250000)		

Con un lr alto, la diferencia es mínima cuando se comparan épocas más grandes (incluso con 2000 iteraciones el resultado es bastante similar al obtenido con 15000). Como mi data set no es demasiado grande estas 15000 iteraciones no toman tanto tiempo, pero en data sets más grandes sí podría haber una diferencia significativa entre usar más o menos epochs.

Resultados de mejora:

MSE in training:	Train:	Validate:	Test:
			
Convergencia rápida.	R2=.5444	R2=.4950	R2=.5465
A pesar de que el bias sigue siendo medio, es un poco más bajo que antes ya que la R2 es más grande. Además, la varianza sigue igual. El modelo sigue estando entre fit y underfit.			

Uso de framework o biblioteca de aprendizaje máquina para la implementación de una solución.

Pytorch:

Para esta entrega estaré usando pytorch, un framework de meta para machine learning.

```
def gradient_descent_torch(X, y, learning_rate=0.001, iterations=20000):
    # Convertir numpy a tensores de PyTorch
    X = torch.tensor(X, dtype=torch.float32)
    y = torch.tensor(y, dtype=torch.float32).view(-1, 1) # columna

    n_samples, n_features = X.shape

    # Definir modelo lineal:  $y = Xw + b$ 
    model = nn.Linear(n_features, 1, bias=True)

    # Definir función de pérdida (MSE)
    criterion = nn.MSELoss()

    # Definir optimizador (SGD en este caso)
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    # Entrenar
    for i in range(iterations):
        # Forward pass
        y_pred = model(X)

        # Calcular pérdida
        loss = criterion(y_pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()

        # Update
        optimizer.step()

        if i % 5000 == 0:
            print(f"Iter {i}: Loss={loss.item():.4f}")

    # Extraer pesos y bias entrenados
    w = model.weight.detach().numpy().flatten()
    b = model.bias.item()
    return w, b, model
```

En el código, lo que estamos haciendo es exactamente lo mismo que antes, pero con funciones ya incluidas en la biblioteca de PyTorch.

Qué hace PyTorch: Hace el cálculo de derivadas y la actualización de parámetros.

PyTorch maneja tensores en lugar de arrays, por lo que convertimos nuestros datos a este formato.

```
# Convertir numpy a tensores de PyTorch
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).view(-1, 1) # columna
```

Definimos el modelo que vamos a usar: linear y le decimos que vamos a tener un bias (termino independiente b). El n_features es la cantidad de variables independientes.

```
# Definir modelo lineal:  $y = Xw + b$ 
model = nn.Linear(n_features, 1, bias=True)
```

Definimos el criterio que vamos a usar, para gradient descent se usa la perdida de MSE.

```
# Definir función de pérdida (MSE)
criterion = nn.MSELoss()
```

Definimos como vamos a optimizar el modelo: SGD=Stochastic Gradient Descent, con nuestro learning rate.

```
# Definir optimizador (SGD en este caso)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

Entrenamos el modelo por el número de iteraciones:

```
# Entrenar
for i in range(iterations):
    # Forward pass
    y_pred = model(X)

    # Calcular pérdida
    loss = criterion(y_pred, y)

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()

    # Update
    optimizer.step()
```

Lo siguiente corresponde al forward pass:

```
# Entrenar
for i in range(iterations):
    # Forward pass
    y_pred = model(X)
```

Se calcula la perdida (MSE).

```
# Calcular pérdida
loss = criterion(y_pred, y)
```

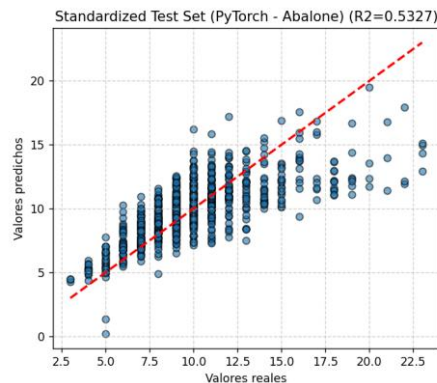
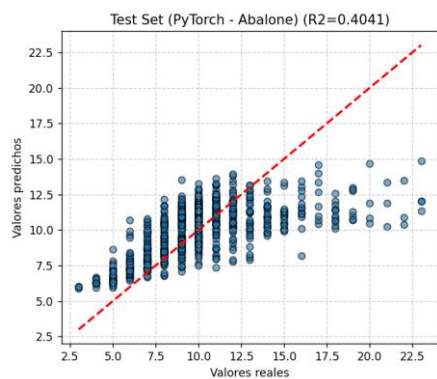
loss.backward se encarga de derivar para saber el cambio de la pérdida respecto a nuestras w y nuestras b.

```
# Backpropagation
optimizer.zero_grad()
loss.backward()
```

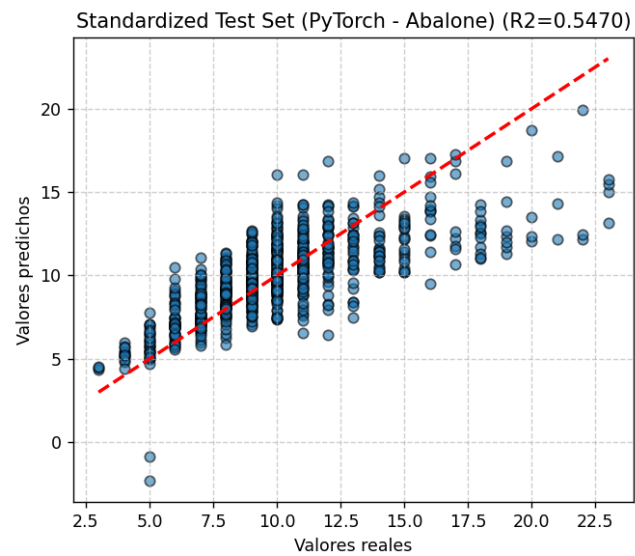
Finalmente, extraemos los datos para regresar cada coeficiente.

```
# Extraer pesos y bias entrenados
w = model.weight.detach().numpy().flatten()
b = model.bias.item()
return w, b, model
```

Al graficar los resultados, obtenemos los mismos resultados (resultados muy parecidos) que para nuestra implementación manual. Se muestran una versión usando torch con valores no estandarizados y estandarizados.



Tras usar los hiperparámetros de la mejora ($lr=.01$, iteraciones=15000):



Los resultados son los mismos que los obtenidos antes con los mismos hiperparámetros.