Tecnológico de Monterrey

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Inteligencia artificial avanzada para la ciencia de datos I

TC3006C - Gpo 101

Actividad #3: Proyecto Regresión logística

Integrantes:

Jorge Ignacio Reyes Pérez - A00573981

Fecha de entrega:
14 de Septiembre del 2025

Regresión lineal from scratch vs framework

En este documento se exponen cómo se puede construir y entrenar un modelo de regresión lineal de dos formas distintas: implementando desde cero (from scratch) y utilizando librerías de machine learning. A lo largo del texto se explican las parte más importantes de los códigos y se comparan los enfoques, para mostrar tanto el aprendizaje que aporta programar todo manualmente como la eficiencia de emplear herramientas ya optimizadas

Desarrollo

Comenzamos con la generación del dataset. En este caso from scratch se definió una función para simular datos de calibración de un motor, donde la entrada es el porcentaje de PWM aplicado y la salida es la velocidad angular

```python
def make_pwm_speed_dataset(n_samples=120, noise_std=0.25, pwm_min=0, pwm_max=100, rng=rng):
    a_true = 0.12
    b_true = -1.0
    pwm = rng.uniform(pwm_min, pwm_max, size=(n_samples, 1))
    noise = rng.normal(0, noise_std, size=(n_samples,))
    omega = b_true + a_true * pwm[:, 0] + noise
    return pwm, omega, a_true, b_true
```

Esta misma función se utilizó también en los modelos con librerías, lo que garantiza que los métodos comparados trabajen con los mismos datos y puedan compararse en igualdad de condiciones

El siguiente paso fue dividir los datos en entrenamiento y prueba. Tanto en el modelo from scratch como con librerías se programó manualmente con la función *train_test_split*, esta decisión de usar la misma función fue para mantener una consistencia en todos los modelos.

```python
def train_test_split(X, y, test_size=0.25, rng=rng):
    n = X.shape[0]
    idx = np.arange(n)
    rng.shuffle(idx)
    n_test = int(round(n * test_size))
    test_idx = idx[:n_test]
```

```
    train_idx = idx[n_test:]
    return X[train_idx], X[test_idx], y[train_idx], y[test_idx]
```

En dado caso que se quiera evitar programar a mano la función, se puede sustituir usando sklearn.model_selection.train_test_split

En cuanto al preprocesamiento, el modelo from scratch necesito una funcion para poder estandarizar las entradas y así evitar problemas

```python
def standardize(X, mean=None, std=None, eps=1e-8):
    if mean is None:
        mean = X.mean(axis=0)
    if std is None:
        std = X.std(axis=0)
    std = np.where(std < eps, 1.0, std)
    return (X - mean) / std, mean, std
```

En cambio, al usar librerías se simplificó con **StandardScaler** de sklearn

```python
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.transform(X_test)
```

Con esto podremos apreciar una gran diferencia entre las dos maneras de programar los modelos, con ayuda de las librerías se reduce la cantidad de código escrito, pero principalmente ganamos robustez, ya que **StandardScaler** ya está optimizado para esta tarea.

Para el aprendizaje en el modelo from scratch la implementación fue en base a un descenso de gradiente. Se definió una clase *LinearRegressionGD* que actualiza los pesos en base a un número de iteraciones

```python
class LinearRegressionGD:
    def __init__(self, lr=0.05, max_iter=50_000, tol=1e-10):
        self.lr = lr
        self.max_iter = max_iter
        self.tol = tol
        self.w = None

    def fit(self, X, y):
        n, d = X.shape
        self.w = rng.uniform(-0.1, 0.1, size=(d,))
        for it in range(self.max_iter):
            y_hat = X @ self.w
            grad = (2.0 / n) * (X.T @ (y_hat - y))
            self.w -= self.lr * grad
```

En contraste, con librerías se puede lograr lo mismo con muchas menos líneas de código. Usando **SGDRegressor** se simula el mismo proceso de descenso de gradiente

```python
from sklearn.linear_model import SGDRegressor
sgd = SGDRegressor(loss="squared_error", penalty=None,
                   learning_rate="constant", eta0=0.05,
                   max_iter=1, tol=None, random_state=42)
sgd.partial_fit(X_train_std, y_train)
```

Además de **SGDRegressor** y para tener una comparativa extra, se implementó de igual manera con **LinearRegression** donde se obtiene directamente la solución con mínimos cuadrados ordinarios

```python
from sklearn.linear_model import LinearRegression
linreg = LinearRegression()
linreg.fit(X_train_std, y_train)
```

Con estas diferencias podemos apreciar que con el modelo from scratch se tiene un control total sobre el algoritmo, mientras que con las librerías se sacrifica una parte del control pero se gana simplicidad y eficiencia.

En cuanto a la evaluación, el modelo from scratch también incluye funciones para calcular métricas como MSE, MAE y R^2

```python
def mse(y_true, y_pred):
    return float(np.mean((y_true - y_pred) ** 2))

def mae(y_true, y_pred):
    return float(np.mean(np.abs(y_true - y_pred)))

def r2_score(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 float(1.0 - ss_res / (ss_tot + 1e-12))
```

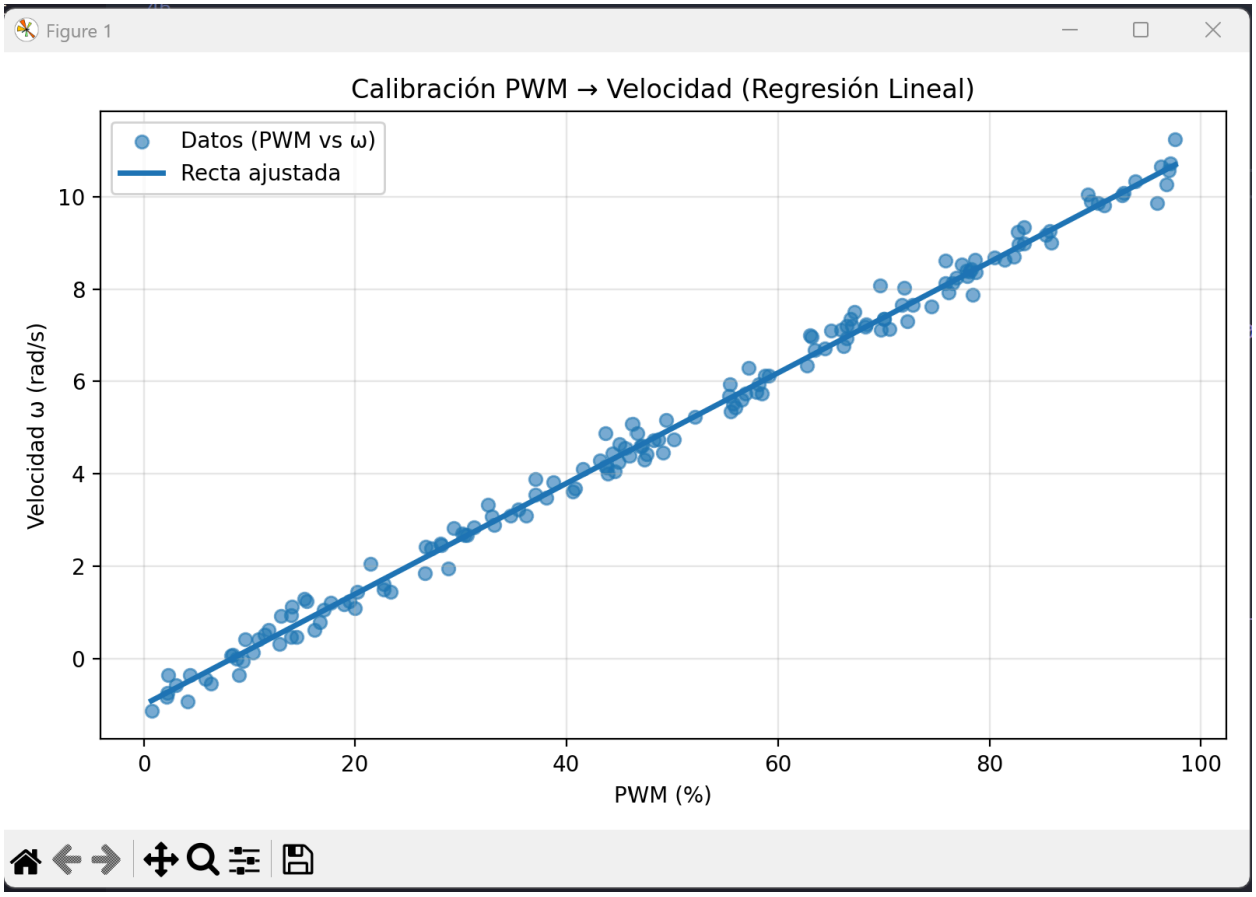
Con **sklearn**, estas métricas ya están implementadas

```python
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
print(mean_squared_error(y_test, y_pred))
```

De nuevo, el contraste es claro: programar todo desde cero ayuda a comprender la matemática detrás, mientras que las librerías facilitan el trabajo en proyectos más grandes y complejos.

Resultados obtenidos

From scratch



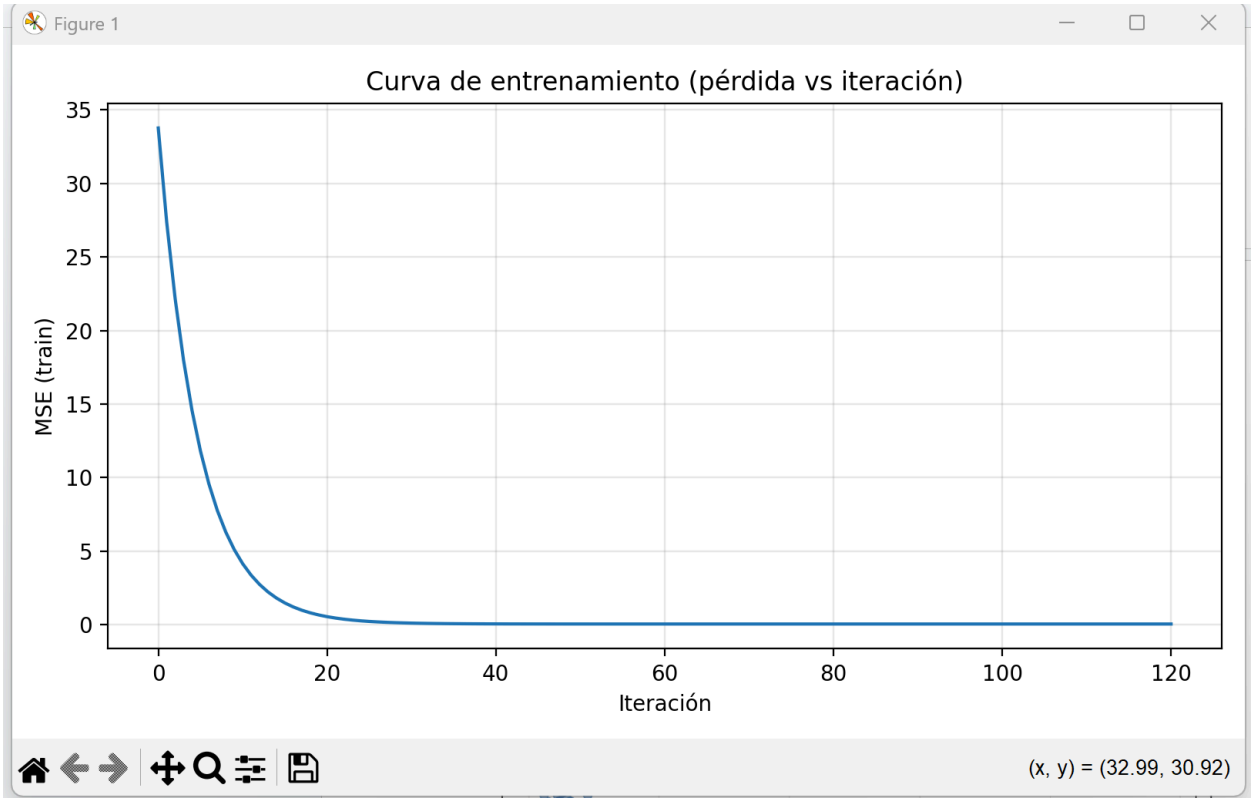
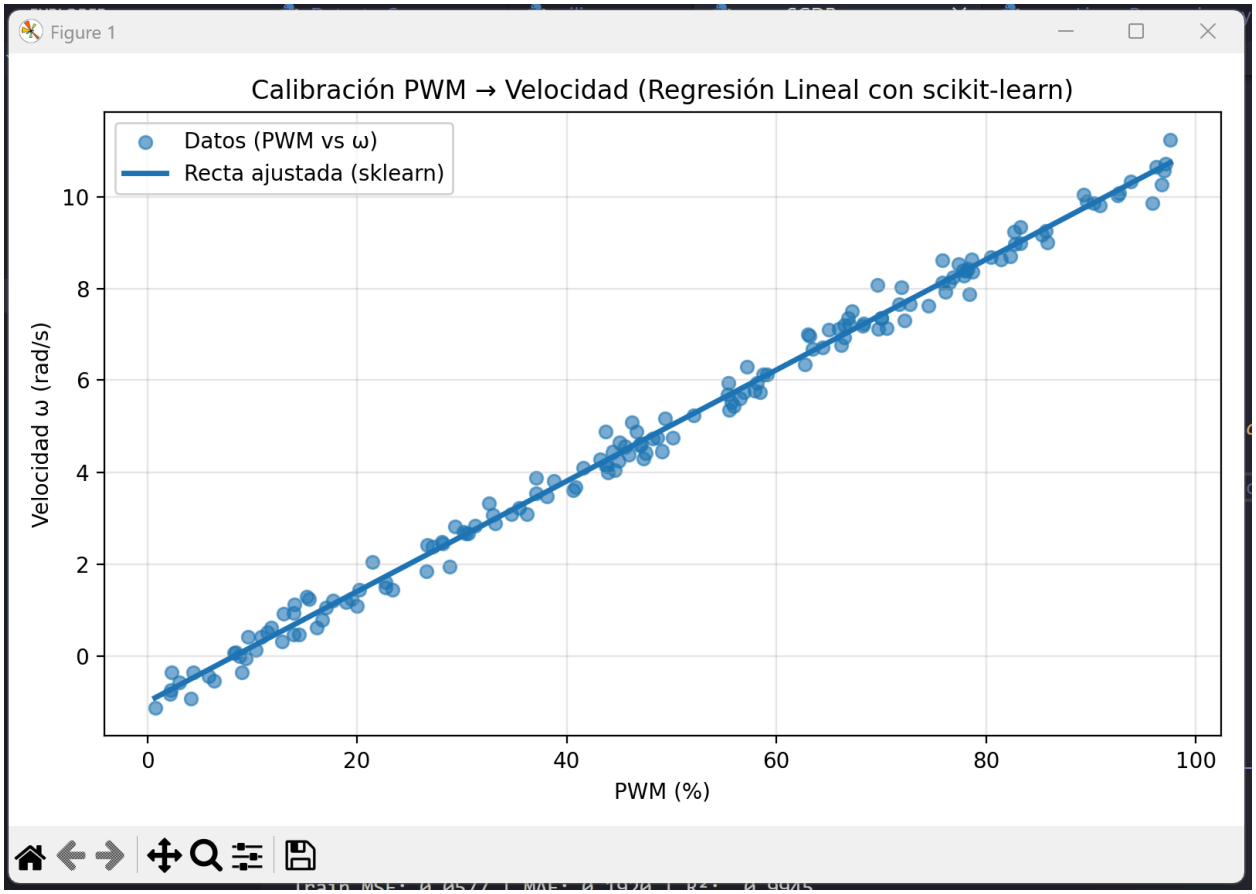


Figure 1
Curva de entrenamiento (pérdida vs iteración)
MSE (train)
Iteración
0
5
10
15
20
25
30
35
0
20
40
60
80
100
120
(x, y) = (32.99, 30.92)

SGDRegressor



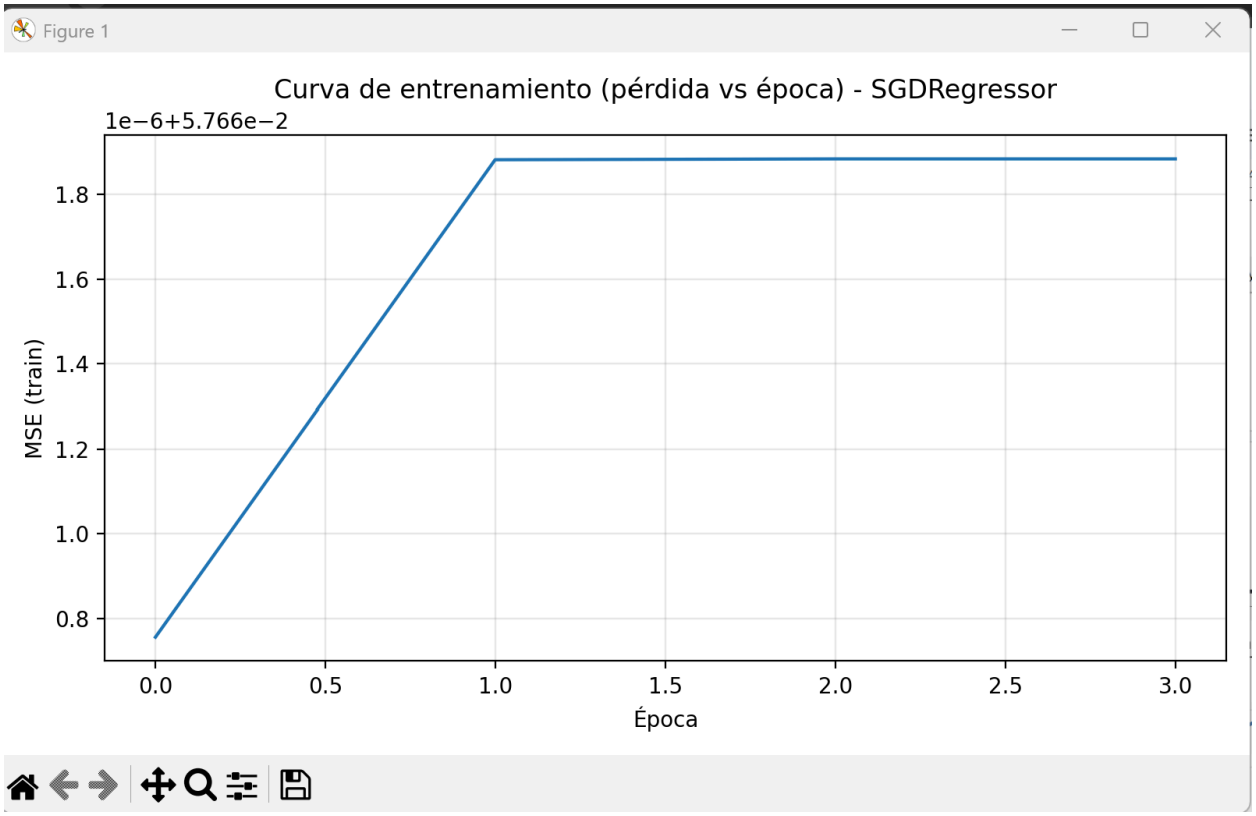
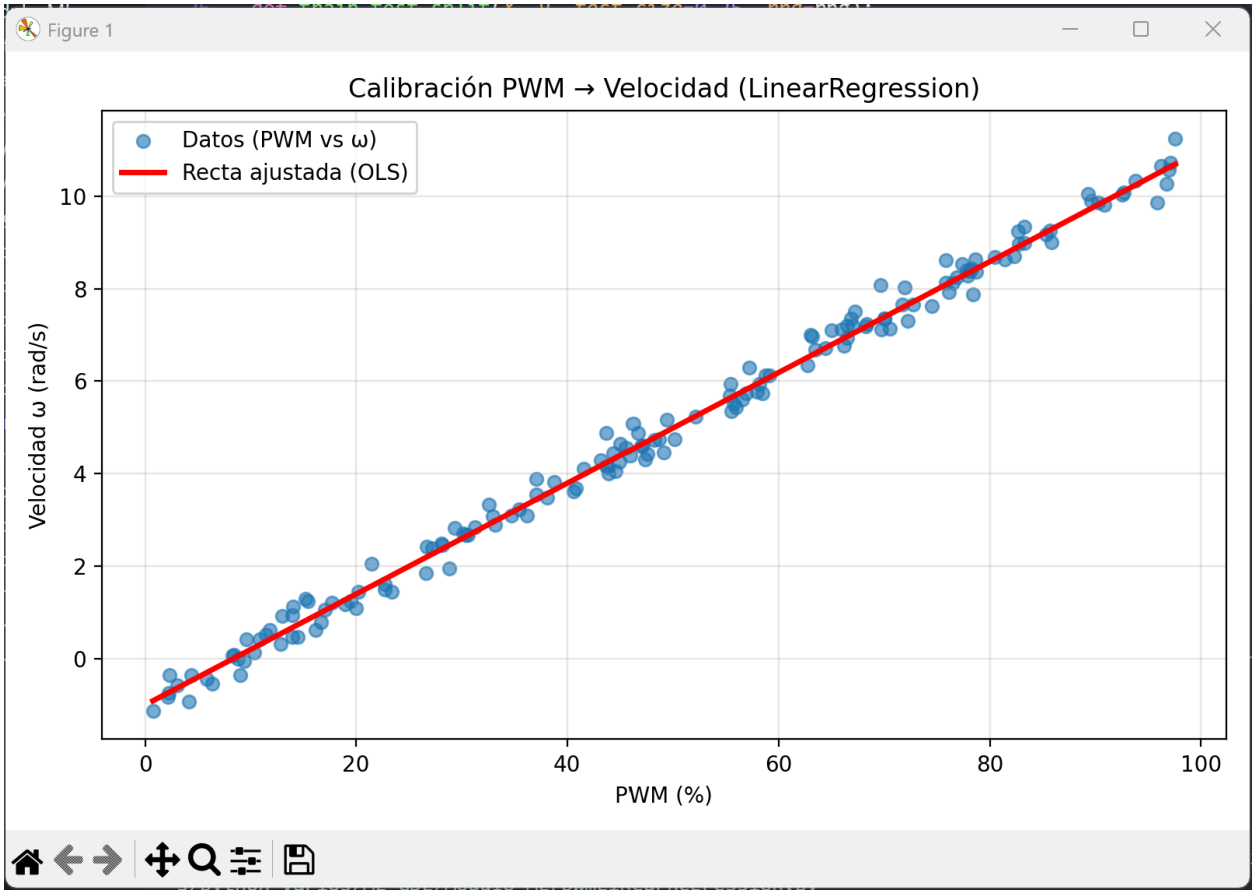


Figure 1
Curva de entrenamiento (pérdida vs época) - SGDRegressor
1e−6+5.766e−2
1.8
1.6
1.4
1.2
1.0
0.8
MSE (train)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Época

LinearRegressor



Pasando a un análisis más profundo del modelo usando librerias, para este se modifico el codigo para poder pasar del enfoque comparativo anterior a un enfoque en el propio rendimiento del modelo y su posible mejora usando hiperparamteros

En la implementación con **SGDRegressor** se trabajó con tres conjuntos: entrenamiento, validación y prueba, lo que permitió evaluar el modelo de manera más rigurosa.

```python
# Dataset sintético
X, y, a_true, b_true = make_pwm_speed_dataset(n_samples=160, noise_std=0.25)

# Split 60/20/20
X_tr, y_tr, X_val, y_val, X_te, y_te = split_train_val_test(
    X, y, test_size=0.20, val_size=0.20, rng=rng
)
```

El procedimiento incluyó early stopping con base en la pérdida de validación, evitando sobreentrenar.

```python
best_model, scaler, (tr_hist, val_hist), best_epoch, best_val = train_sgd_with_early_stopping(
    X_tr, y_tr, X_val, y_val,
    lr=0.05, penalty="l2", alpha=1e-4,
    max_epochs=30000, tol=1e-12, patience=4000, log_every=5000
)
```

Baseline:

```
=== BASELINE (penalty=L2, alpha=1e-4, lr=0.05) ===
[    1] train MSE=0.067013 | val MSE=0.065202
[    2] train MSE=0.067028 | val MSE=0.065199
[    3] train MSE=0.067028 | val MSE=0.065199
[    5] train MSE=0.067028 | val MSE=0.065199
[INFO] Early stopping en época 6 (mejor en 5)

--- MÉTRICAS Baseline ---
 Train → MSE: 0.0670 | MAE: 0.2123 | R²: 0.9932
  Val  → MSE: 0.0652 | MAE: 0.1921 | R²: 0.9955
 Test  → MSE: 0.0744 | MAE: 0.2077 | R²: 0.9929

Diagnóstico:
- Bias: bajo
- Varianza: baja
- Ajuste global: buen ajuste
```

Train → MSE = 0.0670, MAE = 0.2123, R^2 = 0.9932

Validation → MSE = 0.0652, MAE = 0.1921, R^2 = 0.9955

Test → MSE = 0.0744, MAE = 0.2077, R^2 = 0.9929

Estos valores muestran un rendimiento sobresaliente en los tres conjuntos, con diferencias mínimas entre entrenamiento y validación.

Diagnóstico de bias y varianza

```python
def diagnose_bias_variance(r2_tr, r2_val, r2_te):
    if r2_tr < 0.8: bias = "alto"
    elif r2_tr < 0.92: bias = "medio"
    else: bias = "bajo"
    gap_tv = abs(r2_tr - r2_val)
    if gap_tv > 0.10: variance = "alta"
    elif gap_tv > 0.05: variance = "media"
    else: variance = "baja"
    if r2_tr < 0.8 and r2_val < 0.8 and r2_te < 0.8:
        fit = "underfit"
    elif (r2_tr - r2_val > 0.10) or (r2_tr - r2_te > 0.10):
        fit = "overfit"
    else:
        fit = "buen ajuste"
    return bias, variance, fit

bias, variance, fit = diagnose_bias_variance(r2_tr, r2_va, r2_te)
print("Bias:", bias)
```

El bias se puede considerar bajo, dado que el modelo logra un R^2 superior al 0.99 en entrenamiento.
La varianza también es baja, pues la diferencia entre R^2 de entrenamiento y validación es prácticamente despreciable (<0.01).

Diagnóstico de ajuste del modelo

Con estos resultados, el modelo se clasifica como de buen ajuste. No hay señales de underfitting (ya que R^2 es alto en todos los conjuntos), ni de overfitting (pues no hay caída marcada de desempeño en validación o prueba).

Regularización y tuning de parámetros

Posteriormente, se aplicó una búsqueda de hiperparámetros, variando penalty, alpha y la tasa de aprendizaje.

```python
best_val_mse, best_pen, best_alpha, best_lr, tuned_model, tuned_scaler = tune_hyperparams(
```

```
    X_tr, y_tr, X_val, y_val
)
```

```
[MEJOR CONFIGURACIÓN] val MSE=0.057843 | penalty=l2 | alpha=0.001 | lr=0.01

--- MÉTRICAS TUNED ---
 Train → MSE: 0.0616 | MAE: 0.2011 | R²: 0.9938
  Val  → MSE: 0.0578 | MAE: 0.1783 | R²: 0.9960
 Test  → MSE: 0.0716 | MAE: 0.2014 | R²: 0.9931

=== RESUMEN DE MEJORA (Baseline → Tuned) ===
Val MSE: 0.065199 → 0.057843  (Δ=+0.007356)
Test MSE: 0.074399 → 0.071621 (Δ=+0.002778)
Test R² : 0.9929 → 0.9931   (Δ=+0.0003)
Mejor config: penalty=l2 | alpha=0.001 | lr=0.01
```

La mejor configuración encontrada fue:

penalty = l2, alpha = 0.001, lr = 0.01

Con esta configuración ajustada, los resultados mejoraron levemente:

Train → MSE = 0.0616, MAE = 0.2011, R^2 = 0.9938

Validation → MSE = 0.0578, MAE = 0.1783, R^2 = 0.9960

Test → MSE = 0.0716, MAE = 0.2014, R^2 = 0.9931

Comparación Baseline vs Tuned

Val MSE: 0.0652 → 0.0578 (mejora de 0.0074)

Test MSE: 0.0744 → 0.0716 (mejora de 0.0028)

Test R^2: 0.9929 → 0.9931 (mejora de 0.0003)

Aunque las diferencias no son dramáticas porque el problema es lineal y relativamente sencillo el tuning permitió reducir ligeramente el error en validación y prueba, mostrando que la regularización ayuda a estabilizar aún más el aprendizaje.

Conclusiones

La implementación manual permite comprender paso a paso cómo se calcula el gradiente, cómo se actualizan los parámetros y qué significa la convergencia. Es un ejercicio valioso para aprender los fundamentos del machine learning y entender de manera práctica el rol del descenso de gradiente.

Por otro lado, las librerías como scikit-learn ofrecen implementaciones optimizadas, más compactas y seguras, que permiten enfocarse en el problema y no en los detalles algorítmicos. SGDRegressor reproduce el descenso de gradiente de forma más automática, mientras que LinearRegression entrega la solución exacta en un solo paso mediante mínimos cuadrados ordinarios.

Al aplicar un análisis más profundo al modelo con SGDRegressor, se observó que el sesgo es bajo, la varianza es baja y el nivel de ajuste corresponde a un buen ajuste, lo que significa que el modelo generaliza correctamente sin caer en underfitting ni overfitting. Además, al incluir un proceso de regularización y ajuste de hiperparámetros (penalty=L2, alpha=0.001, lr=0.01), se logró una ligera mejora en los errores de validación y prueba, confirmando la estabilidad y robustez del modelo.

En todos los casos, los resultados fueron consistentes y el ajuste fue excelente ($R^2 > 0.99$). Esto demuestra que, para un problema con una relación lineal clara y con ruido controlado, tanto la implementación manual como el uso de librerías convergen al mismo modelo; la diferencia principal radica en el tiempo de desarrollo, el esfuerzo requerido y la facilidad para aplicar técnicas adicionales como regularización y tuning.