Actividad individual: Modelo de regresión lineal con redes neuronales

María Luisa Silva Mejías

Aprendizaje automático

Ingeniería informática

Índice

Introducción

El objetivo de esta actividad es modificar el código de la práctica de redes neuronales para realizar una comparación con la práctica de regresión.

Se debe de buscar el mejor ajuste en las distintas variantes (escalón, sigmoide y delta) para los datos de la práctica 1 y comparar los resultados con la gráfica del error.

Descripción del método y configuración

Lo primero que se ha hecho para todas las variantes es leer el archivo que contiene los datos (regresion_1.csv) y dividir el dataset en dos variables, una de entrada y otra de salida.

Todas las variantes siguen la misma estructura. Primero se crea el perceptrón con una entrada, ya que este dataset solo tiene una, después, se entrena este perceptrón con una asignación de 100 épocas, y, por último, se obtienen las gráficas de errores y de línea de regresión, calculando el error cuadrático medio para la predicción final en la Regla Delta y en sigmoide y el error absoluto total en escalón. Todo esto se hace para tres factores de aprendizaje diferentes.

```python
# Cargar el conjunto de datos regresion_1.csv con nombres de columnas
data = pd.read_csv('regresion_1.csv', header=None)

# Dividir el dataset en entradas (X) y etiquetas (y)
X = data.iloc[:, 0].values.reshape(-1, 1)   # Seleccionar la columna X como entrada
y = data.iloc[:, 1].values                  # Seleccionar la columna Y como salida

# Definir los valores de learning_rate que queremos probar
learning_rates = [0.0036, 0.012, 0.01235]

# Entrenar el perceptrón para cada learning_rate y almacenar los errores
for lr in learning_rates:
    perceptron = Perceptron(input_size=1, learning_rate=lr)
    errors = perceptron.train(X, y, epochs=100)

    # Graficar el error por época para cada learning_rate
    plt.figure(figsize=(10, 6))
    plt.plot(errors, marker='o', label=f'Error cuadrático medio por época')
    plt.title(f'Error durante el entrenamiento del Perceptrón\nfactor de aprendizaje = {lr}')
    plt.xlabel('Época')
    plt.ylabel('Error cuadrático medio')
    plt.grid()
    plt.legend()
    plt.show()

    # Comparación final con la recta de regresión para el último modelo entrenado (con la última tasa de aprendizaje)
    y_pred = []
    print("\nPrueba del perceptrón con datos reales:")
    for inputs in X:
        output = perceptron.predict(inputs)
        print(f"Entradas: {inputs.item()}, Salida predicha: {output.item()}")
        y_pred.append(output.item())

    # Calcular el Error Cuadrático Medio
    ecm = mean_squared_error(y,y_pred)
    print("Error Cuadrático Medio:", ecm)

# Calcular el Error Absoluto Total
total_error = np.sum(np.abs(y - y_pred))
print("Error Absoluto Total:", total_error)
```

En el entrenamiento del perceptrón es donde está la diferencia entre la Regla Delta, el sigmoide y el escalón:

Regla Delta

Teoría

Es una variante del método de descenso por el gradiente. En lugar de tratar de minimizar el error cuadrático cometido sobre todos los ejemplos del dataset, procede incrementalmente tratando de descender el error cuadrático cometido sobre el ejemplo que se esté tratando en cada momento.

Para el entrenamiento hace falta un dataset, un factor de aprendizaje y una función de activación lineal. Los pesos se van actualizando con esta función: $w_i \leftarrow w_i + \eta(y - o)x_i$, donde w es el peso, η es el factor de aprendizaje, y es la columna de salida del dataset, o es la predicción y x es la columna de entrada del dataset.

Implementación y descripción de código

```python
# Clase del perceptrón
class Perceptron:
    def __init__(self, input_size, learning_rate=0.01235):
        # Inicializar pesos y bias (con valores aleatorios pequeños)
        self.weights = np.random.rand(input_size) * 0.01
        self.bias = np.random.rand() * 0.01
        self.learning_rate = learning_rate

    # Predicción usando la función de activación lineal
    def predict(self, inputs):
        # Suma ponderada de las entradas más el sesgo
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        return weighted_sum

    # Entrenamiento del perceptrón utilizando la regla delta
    def train(self, training_inputs, labels, epochs=10):
        # Lista para almacenar el error en cada época
        errors = []
        for epoch in range(epochs):
            total_error = 0  # Variable para acumular el error total de la época
            for inputs, label in zip(training_inputs, labels):
                # Calcular la salida predicha
                prediction = self.predict(inputs)

                # Calcular el error (diferencia entre la salida real y la predicción)
                error = label - prediction

                # Actualización de pesos y bias usando la regla delta
                self.weights += self.learning_rate * error * inputs
                self.bias += self.learning_rate * error

                # Acumular el error
                total_error += error**2

            # Almacenar el error total para esta época
            errors.append(total_error/(2*len(training_inputs)))

        return errors
```

Se crea la clase Perceptron donde tenemos una función para inicializar las variables de peso, bias y learning_rate (factor de aprendizaje). Esta última se ha probado con diferentes valores (0.0036, 0.012 y 0.01235). Con cada factor obtenemos un error diferente tanto en el entrenamiento con en la predicción final.

También está la función predict, donde se hace la predicción. Se devuelve directamente la suma ponderada de las entradas más el sesgo, ya que la función de activación es lineal (g'(in) = C).

Por último, tenemos la función de entrenamiento (train) que devuelve el error, en este caso el cuadrático medio sí es apropiado para la Regla Delta. En cada época recorremos todas las filas del dataset, y en cada fila calculamos la predicción. Seguidamente, se

calcula el error restándole a la y la predicción obtenida. Después se actualizan el peso y el bias usando las funciones correspondientes a la Regla Delta. Por último, se eleva el error, obtenido con anterioridad, al cuadrado y se añade al sumatorio que ya se tiene de las demás filas anteriores, para cuando se termine de recorrer todas las filas, se calcule el error cuadrático medio final dividiéndolo entre dos por el número de datos de entrenamientos.

Sigmoide

Teoría

Esta función de activación se calcula con esta ecuación:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Aquí, los pesos se van actualizando de esta manera: $w_i \leftarrow w_i + \eta(y - o) \cdot f'(o) \cdot x_i$, donde w es el peso, η es el factor de aprendizaje, y es la columna de salida del dataset, o es la predicción, f'(o) es la derivada de la función de activación sigmoide para o ($f'(o) = o(1 - o)$), y x es la columna de entrada del dataset.

Implementación y descripción de código

```python
# Definir la función de activación sigmoide
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Definir la derivada de la función sigmoide
def sigmoid_derivative(x):
    return x * (1 - x)

# Clase del perceptrón
class Perceptron:
    def __init__(self, input_size, learning_rate=0.1):
        # Inicializar pesos y bias (con valores aleatorios pequeños)
        self.weights = np.random.rand(input_size) * 0.01
        self.bias = np.random.rand() * 0.01
        self.learning_rate = learning_rate
        self.errors = []  # Lista para almacenar el error por época

    # Predicción usando la función de activación sigmoide
    def predict(self, inputs):
        # Suma ponderada de las entradas más el sesgo
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        # Aplicar la función de activación (sigmoide)
        return sigmoid(weighted_sum)  # Retornar la salida continua de la sigmoide

    # Entrenamiento del perceptrón utilizando la regla delta
    def train(self, training_inputs, labels, epochs=10):
        for epoch in range(epochs):
            epoch_error = 0  # Inicializar el error de la época
            for inputs, label in zip(training_inputs, labels):
                # Obtener la predicción usando sigmoide
                prediction = self.predict(inputs)

                # Cálculo del error (diferencia entre la salida real y la predicción sigmoide)
                error = label - prediction
                epoch_error += error**2

                # Derivada de la función de activación sigmoide
                adjustment = error * sigmoid_derivative(prediction)
                # Actualización de pesos y bias
                self.weights += self.learning_rate * adjustment * inputs
                self.bias += self.learning_rate * adjustment

            # Guardar el error promedio para esta época
            self.errors.append(epoch_error/(2*len(training_inputs)))
```

Se ha hecho lo mismo que en la Regla Delta, pero cambiando las ecuaciones correspondientes. Primero se han definido las funciones de sigmoid (sigmoide) y sigmoid_derivative (derivada de sigmoide). La primera se usa en la predicción, ya que ahora no tenemos una función de activación lineal. Y la segunda se usa a la hora de calcular la actualización de los pesos y de bias.

En este caso el factor de aprendizaje da igual que valor se le ponga ya que para todos da unos errores parecidos debido a que el sigmoide no es una buena técnica para problemas de regresión.

Escalón

Teoría

Es una técnica para problemas de clasificación, por lo que se considera un valor de salida como “SI” (1) y otro valor como “NO” (0). Por lo tanto, la función de activación escalonada será booleana. Esta función sólo podrá ser correctamente representada por un perceptrón escalón si existe un hiperplano que separa los elementos con valor 1 de los elementos con valor 0 (linealmente separable).

La función de actualización de pesos es la misma que la de la Regla Delta.

Implementación y descripción de código

```python
# Definir la función de activación escalonada para la salida booleana
def step_function(x):
    return 1 if x >= 0.5 else 0

# Clase del perceptrón
class Perceptron:
    def __init__(self, input_size, learning_rate=0.1):
        # Inicializar pesos y bias (con valores aleatorios pequeños)
        self.weights = np.random.rand(input_size) * 0.01
        self.bias = np.random.rand() * 0.01
        self.learning_rate = learning_rate

    # Predicción usando la función escalonada
    def predict(self, inputs):
        # Suma ponderada de las entradas más el sesgo
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        # Aplicar la función de activación (escalonada)
        return step_function(weighted_sum)

    # Entrenamiento del perceptrón utilizando un enfoque simplificado
    def train(self, training_inputs, labels, epochs=100):  # Se establecen 100 épocas aquí
        # Lista para almacenar el error en cada época
        errors = []
        for epoch in range(epochs):
            total_error = 0  # Variable para acumular el error total de la época
            for inputs, label in zip(training_inputs, labels):
                # Calcular la salida predicha
                prediction = self.predict(inputs)

                # Calcular el error (diferencia entre la salida real y la predicción)
                error = label - prediction

                # Actualización de pesos y bias usando un promedio de errores
                self.weights += self.learning_rate * error * inputs
                self.bias += self.learning_rate * error

                # Acumular el error total
                total_error += abs(error)

            # Almacenar el error total para esta época
            errors.append(total_error)


        # Devolver la lista de errores para graficar
        return errors
```

Se define la función de activación escalonada y la clase Perceptrón que muy similar a la de la Regla Delta, lo único que cambia es que en la función predict, a la hora de hacer la predicción, se cambia la función de activación lineal por la escalonada. Además, el error que se calcula es el total, ya que para el entrenamiento de un perceptrón con función de activación escalonada no es adecuado calcular el error cuadrático medio debido a que esta función se usa para problemas de clasificación.

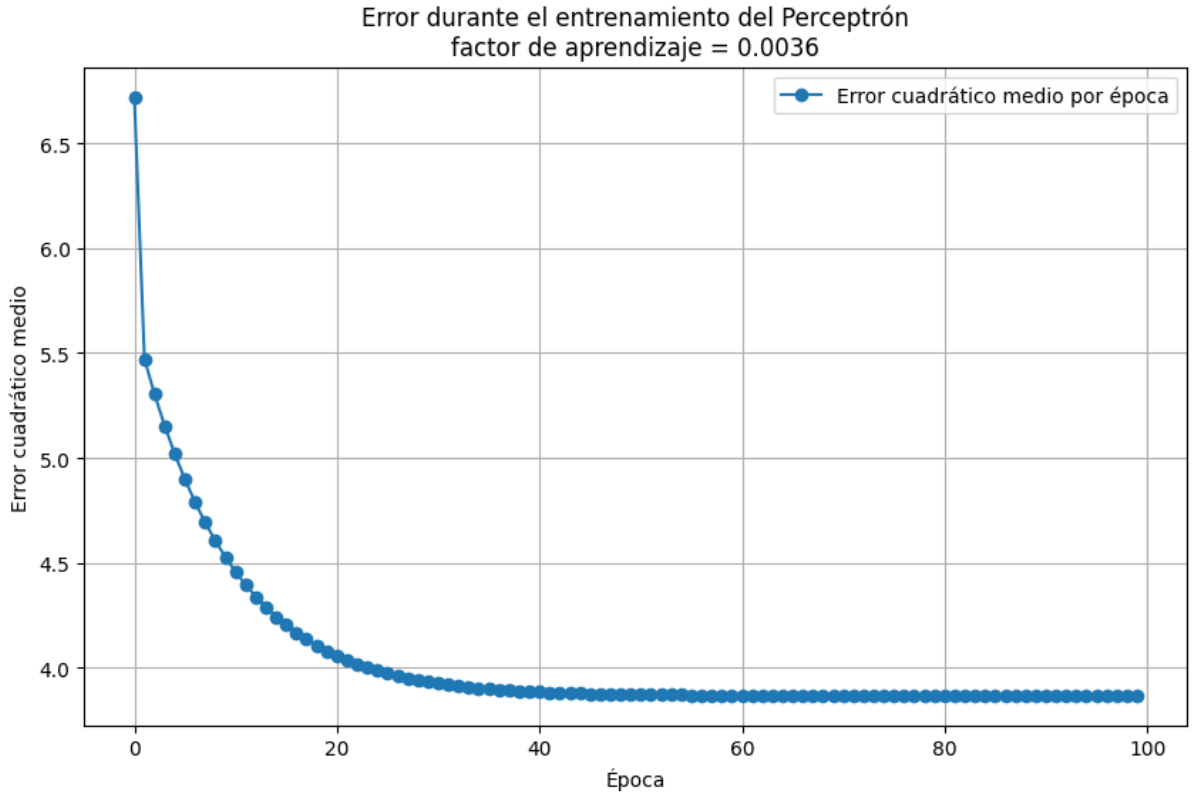
Aquí, al igual que en la variante sigmoide, el valor del factor de aprendizaje es algo que nos da igual ya que el error es igual de alto para todos los valores.

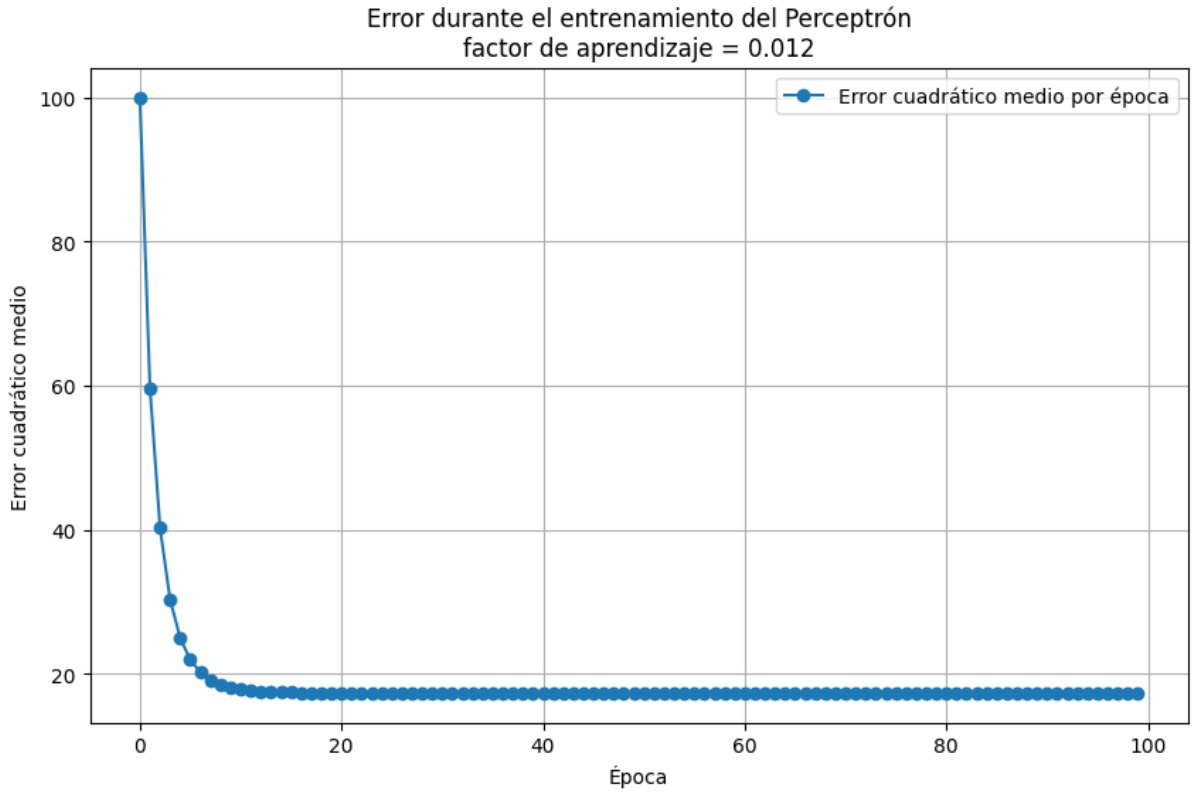
Resultados y análisis de gráficas de error

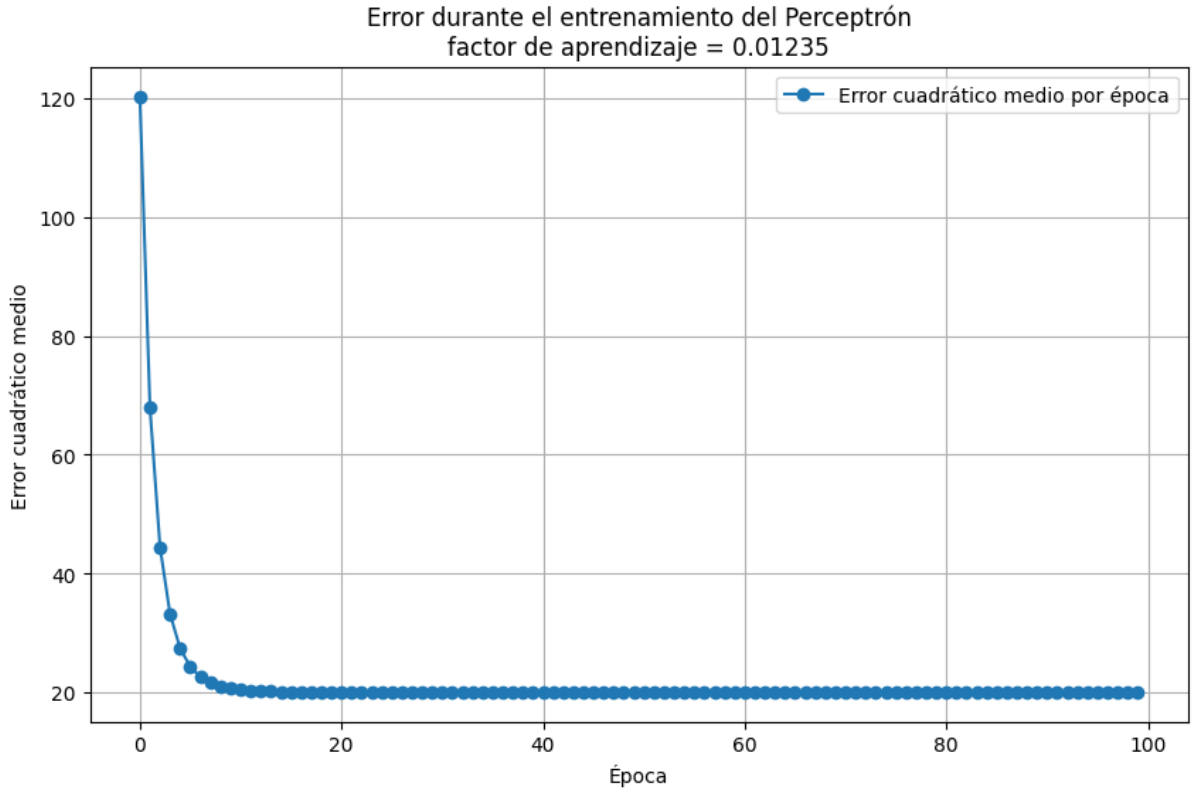
Comparación del error de las tres variantes

Regla Delta

Fase de entrenamiento







Predicción final

Error cuadrático medio para factor de aprendizaje = 0.0036: 13.4715

Error cuadrático medio para factor de aprendizaje = 0.012: 9.0056

Error cuadrático medio para factor de aprendizaje = 0.01235: 8.9658

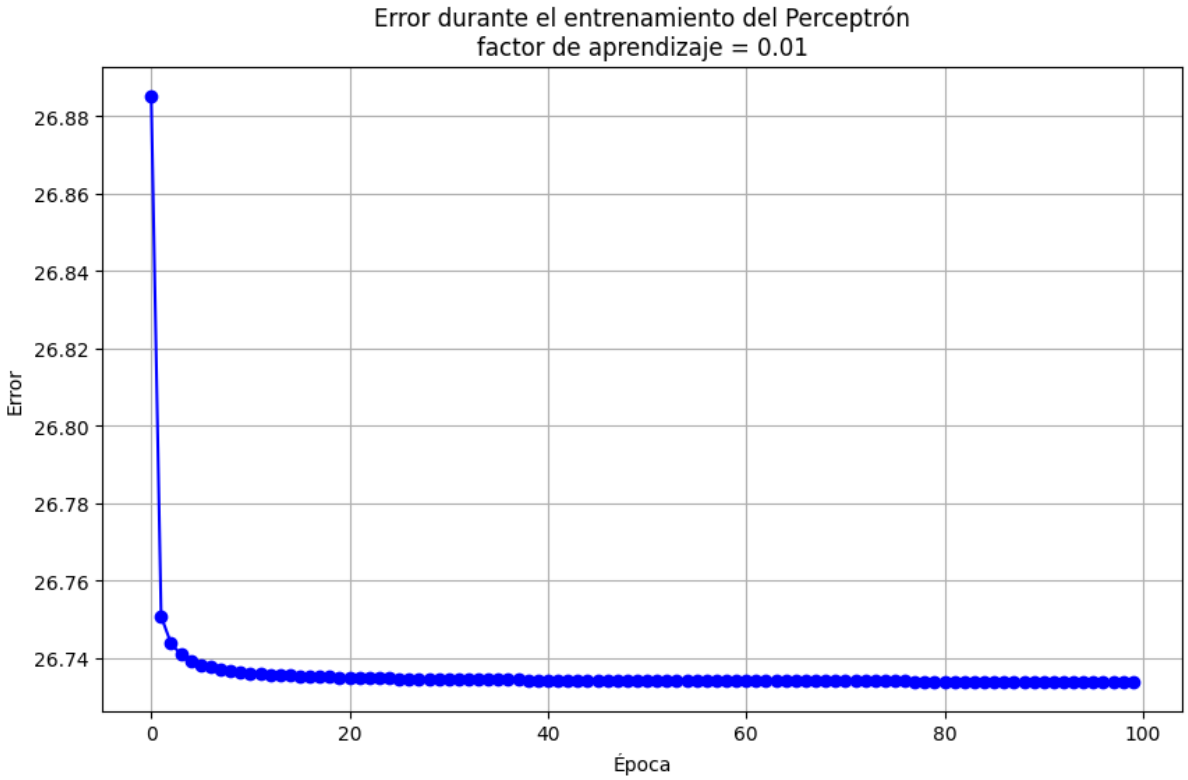
Conclusión

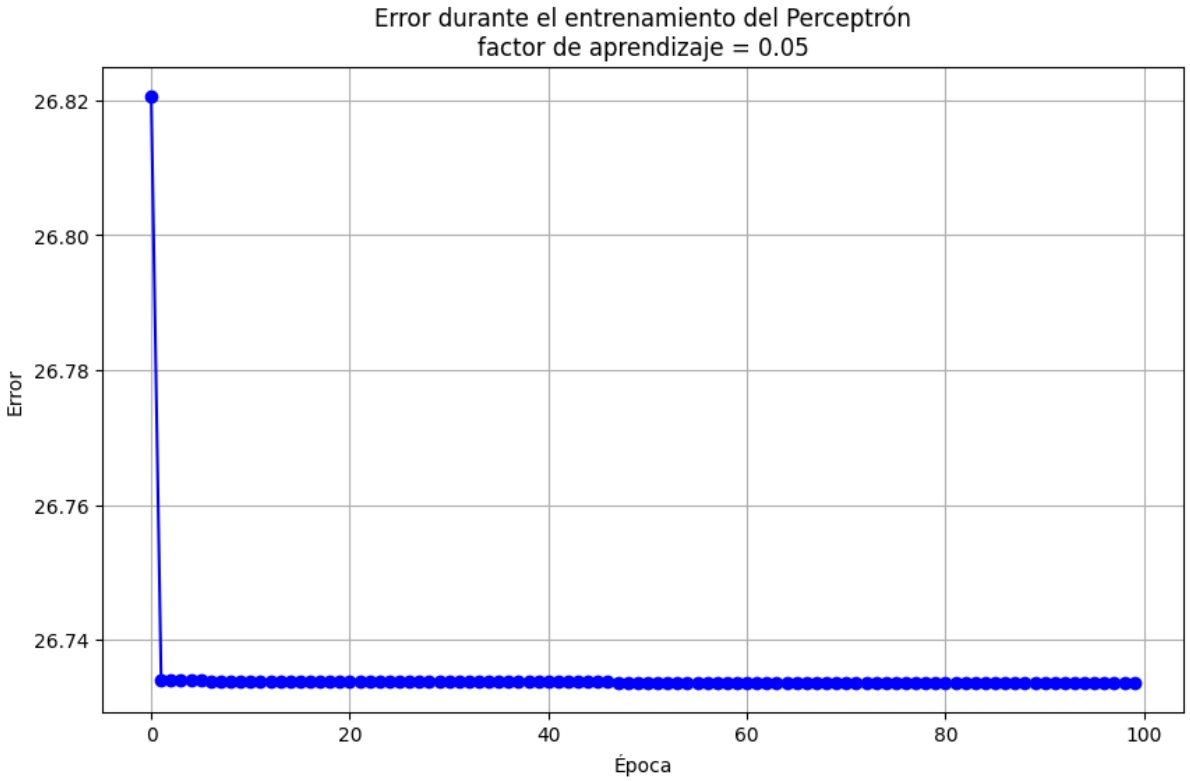
Como podemos observar, la gráfica con menor error para el entrenamiento es la primera, la que tiene el factor de aprendizaje a 0.0036. Pero, en la predicción final, el error mínimo lo da el factor de aprendizaje de 0.01235.

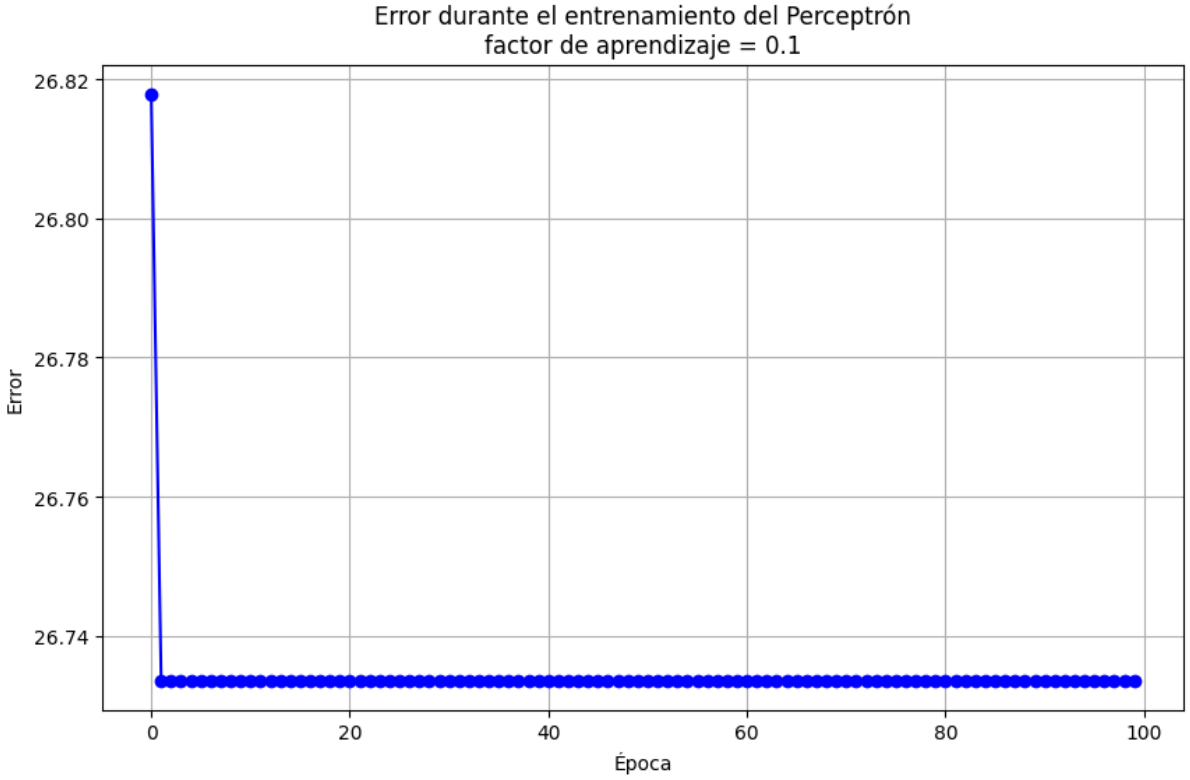
En mi opinión, el mejor ajuste del factor de aprendizaje para la Regla Delta en este problema es el de 0.01235, ya que al final del día, el rendimiento del modelo en los datos de predicción es lo más importante. El objetivo principal es que se realicen predicciones precisas.

Sigmoide

Fase de entrenamiento







Predicción final

Error cuadrático medio para factor de aprendizaje = 0.01: 53.4678

Error cuadrático medio para factor de aprendizaje = 0.05: 53.4673

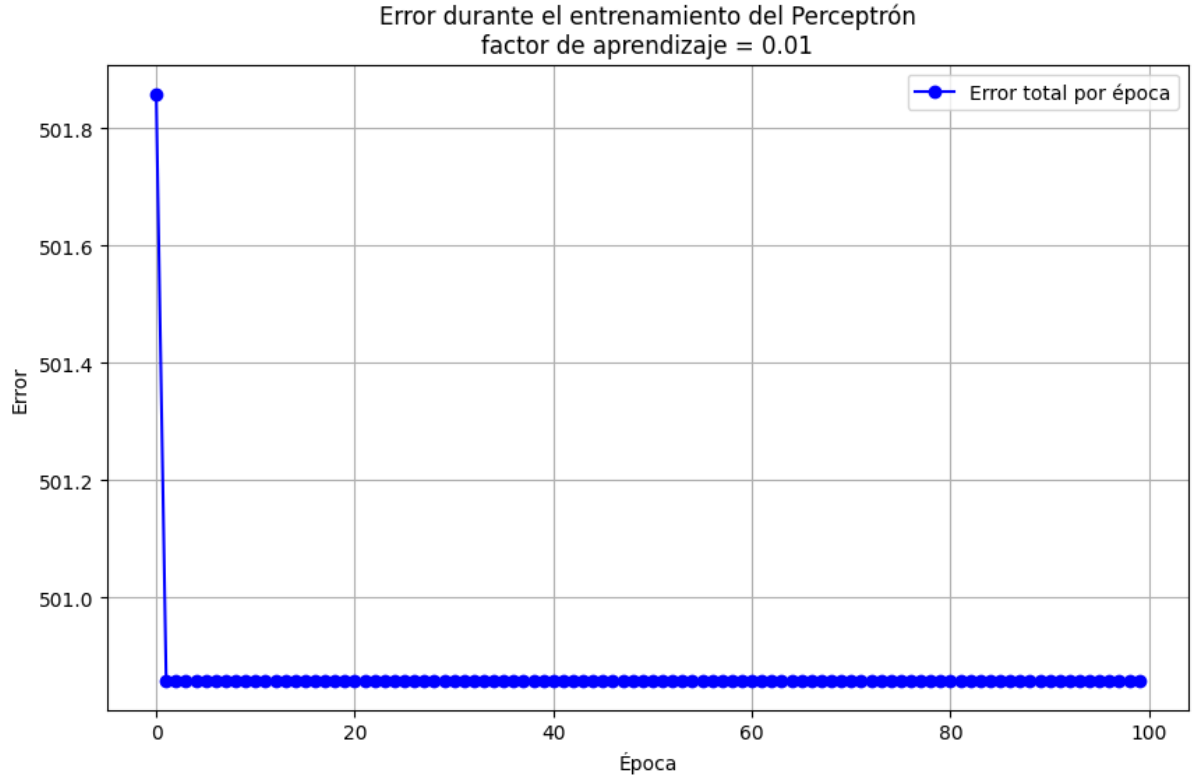
Error cuadrático medio para factor de aprendizaje = 0.1: 53.4672

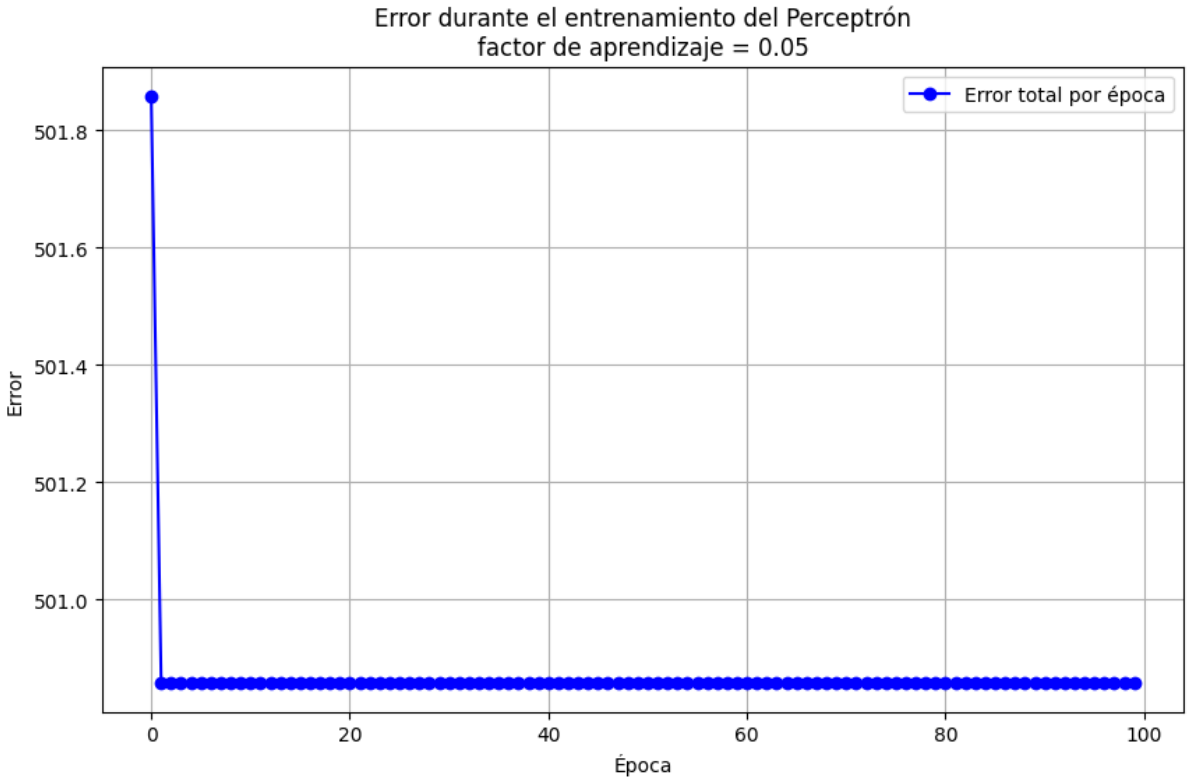
Conclusión

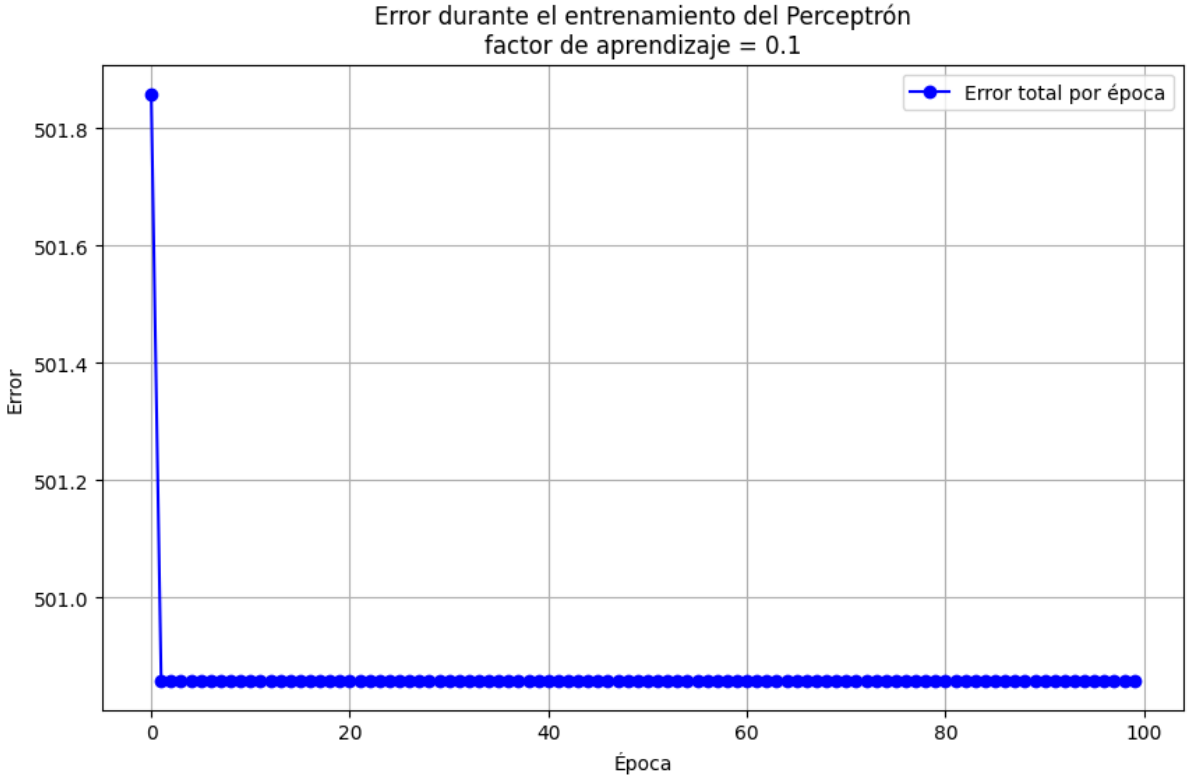
En este caso, vemos que la diferencia tanto en las gráficas como en los errores de la predicción final es insignificante. Da unos errores muy altos para los datos que tenemos. Esto es debido a que la función de activación sigmoide es apropiada para problemas de clasificación.

Escalón

Fase de entrenamiento







Predicción final

Error absoluto total para factor de aprendizaje = 0.01: 500.8589

Error absoluto total para factor de aprendizaje = 0.05: 500.8589

Error absoluto total para factor de aprendizaje = 0.1: 500.8589

Conclusión

Las tres gráficas y los tres errores que se obtienen la predicción final son idénticos. Los errores son bastante altos para los datos que tenemos, debido a que la función de activación escalonada es adecuada para problemas de clasificación.

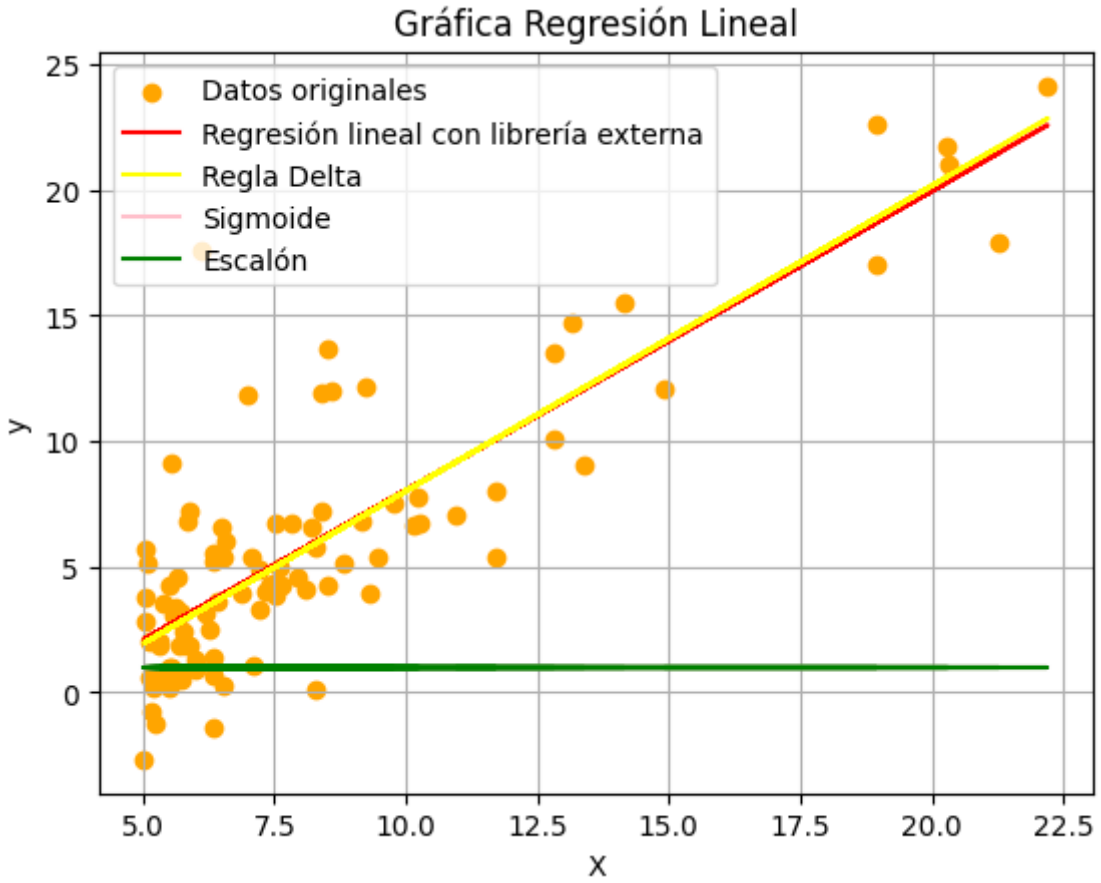
Conclusión

La mejor técnica es la Regla Delta que es con la que conseguimos un error menor. Esto es algo normal, ya que la función sigmoide y la escalonada son propias de problemas de clasificación.

Comparación con la regresión de la práctica 1

En la práctica 1 se hizo una predicción del mismo dataset con librería externa, ecuaciones normales, descenso por gradiente y descenso por gradiente estocástico.

Si comparamos la gráfica que se hizo de la regresión lineal con librería externa, por ejemplo, con la de la Regla Delta, sigmoide y escalón nos sale esto:



Aquí podemos observar que la mejor técnica es la Regla Delta ya que la recta que nos sale es muy similar a la que nos sale en la regresión lineal con librería externa. Para la Regla Delta se ha elegido el factor de aprendizaje con el que nos salía el menor error cuadrático medio para la predicción final (0.01235).

Además, se ha calculado el error cuadrático medio para la predicción con librería externa y da 8.9539, algo muy cercano al error que daba la predicción con Regla Delta (8.9658).

Conclusión

Con toda esta reflexión se puede inferir que la Regla Delta es la única adecuada para predecir modelos de regresión y que las funciones de activación sigmoide y escalonada son para abordar problemas de clasificación.

En comparación con la práctica 1, donde calculábamos las thetas para obtener la ecuación de la recta, obtenemos un error muy parecido a lo que obtenemos con la Regla Delta. Por tanto, podemos afirmar que esta técnica es bastante efectiva para problemas de regresión.