



**Tecnológico
de Monterrey**

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Querétaro

TC3006C Inteligencia artificial avanzada para la ciencia de datos I

Módulo 2 Análisis y Reporte sobre el desempeño del modelo.

Profesores:

Benjamín Valdés Aguirre

Carlos Alberto Dorantes Dosamantes

José Antonio Cantoral Ceballos

Denisse Lizbeth Maldonado Flores

Alejandro Fernández Vilchis

Presenta:

José Emiliano Riosmena Castañón – A01704245

Fecha:

Sábado, 7 de septiembre del 2024

Índice

Introducción	4
Descripción del Dataset	5
Selección de columnas (features).....	5
Uso de ETL (Extract – Transform – Load)	7
Implementación sin el uso de framework	10
Función de escalamiento (Scaling).....	10
Hipótesis	11
Mean Square Error (MSE)	12
Gradiente Descendiente	13
Coeficiente R^2	14
Resultados (Implementación sin framework)	16
Resultados MSE	16
Resultados R^2	16
Visualización de los resultados (gráficas).....	17
Conclusiones y áreas de oportunidad	18
Implementación con el uso de framework	19
Función de escalamiento.....	19
Gradiente Descendiente	20
Resultados (Implementación con framework)	21
Resultados MSE	21
Resultados R^2	21
Resultados Bias	22
Visualización de los resultados (gráficas).....	22
Conclusiones y áreas de oportunidad	23
Implementación ajustada	24
Variables globales	24
Regularización L2 (Ridge)	24
Resultados (Implementación ajustada).....	26
Resultados MSE	26
Resultados R^2	26
Resultados Bias	27

Visualización de los resultados (gráficas).....	27
Comparación de resultados	28
Conclusión (Comparativa de resultados)	29
Conclusiones generales.....	30
Referencias	31

Introducción

En el presente ensayo, se explorará el proceso integral para construir un modelo de regresión lineal, abarcando desde la elección del conjunto de datos adecuado hasta la interpretación detallada de los resultados obtenidos. El análisis incluirá una discusión sobre los usos de las principales técnicas y algoritmos estudiados en clase, los cuales han sido fundamentales para la creación y ajuste del modelo.

Se profundizará en los desafíos encontrados durante la implementación, así como en las soluciones adoptadas para superarlos, resaltando la importancia de realizar ajustes dinámicos en cada etapa. Este proceso de refinamiento continuo asegura no solo un análisis robusto de los datos, sino también su conversión efectiva en información valiosa para la toma de decisiones.



Descripción del Dataset

Antes de analizar las técnicas empleadas, es importante familiarizarnos con el dataset utilizado en este proyecto. Comprender el dataset nos ayudará a entender cómo funcionan las técnicas aplicadas a este último.

Para este proyecto, utilicé un dataset obtenido de Kaggle que incluye especificaciones de automóviles de 1985. Entre las características del dataset se encuentran la marca, el tipo de combustible, la tracción de las ruedas, la ubicación del motor, el tipo de vehículo, el cilindraje, el tamaño, entre otras. Este dataset resulta adecuado debido a su variedad de columnas numéricas y categóricas, así como a su amplia cantidad de instancias, lo que facilita su manipulación y permite obtener resultados precisos.

Con esta base, ahora podemos explorar las técnicas utilizadas en el modelo. Empezaremos con ETL, la primera técnica aplicada para extraer, transformar y cargar nuestro dataset.

symboling	normalized_to	make	fuel_type	aspiration	num_doors	body_style	drive_wheels	engine_locatic	wheel_base	length
3 ?		alfa-romero	gas	std	two	convertible	rwd	front	88.6	168.8
3 ?		alfa-romero	gas	std	two	convertible	rwd	front	88.6	168.8
1 ?		alfa-romero	gas	std	two	hatchback	rwd	front	94.5	171.2
2	164	audi	gas	std	four	sedan	fwd	front	99.8	176.6
2	164	audi	gas	std	four	sedan	4wd	front	99.4	176.6
2 ?		audi	gas	std	two	sedan	fwd	front	99.8	177.3
1	158	audi	gas	std	four	sedan	fwd	front	105.8	192.7
1 ?		audi	gas	std	four	wagon	fwd	front	105.8	192.7
1	158	audi	gas	turbo	four	sedan	fwd	front	105.8	192.7
0 ?		audi	gas	turbo	two	hatchback	4wd	front	99.5	178.2
2	192	bmw	gas	std	two	sedan	rwd	front	101.2	176.8
0	192	bmw	gas	std	four	sedan	rwd	front	101.2	176.8
0	188	bmw	gas	std	two	sedan	rwd	front	101.2	176.8
0	188	bmw	gas	std	four	sedan	rwd	front	101.2	176.8
1 ?		bmw	gas	std	four	sedan	rwd	front	103.5	189
0 ?		bmw	gas	std	four	sedan	rwd	front	103.5	189
0 ?		bmw	gas	std	two	sedan	rwd	front	103.5	193.8
0 ?		bmw	gas	std	four	sedan	rwd	front	110	197
2	121	chevrolet	gas	std	two	hatchback	fwd	front	88.4	141.1
1	98	chevrolet	gas	std	two	hatchback	fwd	front	94.5	155.9
0	81	chevrolet	gas	std	four	sedan	fwd	front	94.5	158.8
1	118	dodge	gas	std	two	hatchback	fwd	front	93.7	157.3

Figura 1.0. Ejemplo de los datos originales del dataset.

Selección de columnas (features)

Como mencionaba anteriormente, el objetivo del modelo era predecir el precio de un vehículo basado en sus diferentes características. Por lo que el siguiente paso es identificar qué características nos servirían para lograr esta predicción. Para este

caso, consideré que las columnas más relevantes para poder hacer esta predicción son las siguientes:

- Distancia entre ejes (*wheel_base*): Influye en el tamaño y la estabilidad del vehículo, mientras más larga sea esta distancia, los vehículos suelen ser más espaciosos y costosos.
- Peso en vacío (*curb_weight*): Los vehículos más pesados suelen tener precios más altos debido a los costos asociados con materiales y componentes adicionales.
- Tamaño del motor (*engine_size*): El tamaño de un motor es un indicador directo de la potencia y el rendimiento del vehículo. Generalmente, los motores más grandes suelen ser más caros debido a la capacidad y rendimiento mejorado.
- Potencia (*horsepower*): La potencia del motor nos ayuda a medir el rendimiento del vehículo. Los vehículos con mayor potencia suelen ser más caros ya que ofrecen un rendimiento superior y normalmente están asociados con modelos deportivos o de lujo.
- Millas por galón en ciudad (*city_mpg*): La eficiencia de combustible en la ciudad puede influir en el precio. Los vehículos que ofrecen mejor rendimiento de combustible en la ciudad pueden tener un precio más alto debido a la demanda por vehículos económicos.
- Millas por galón en carretera (*highway_mpg*): Similar al caso anterior, la eficiencia del combustible en la carretera puede influir en el precio. Los vehículos que son eficientes en carretera suelen tener un precio mayor debido a su menor costo de operación.

Estas características nos proporcionan una buena representación de aspectos importantes del vehículo, y nos podrían servir para que nuestro modelo pudiera predecir el precio de un vehículo con este conjunto de características.

Uso de ETL (Extract – Transform – Load)

Para el dataset utilizado, lo primero que hicimos fue definir qué información del conjunto de datos nos resultaba útil. En este caso, empleé el dataset para intentar predecir el precio de un vehículo basándome en algunas de sus características. Para ello, usé la librería pandas para extraer los datos del archivo imports-85.data e imports-85.names para identificar las columnas. Luego, combinamos ambos archivos en un dataframe que contenía toda la información, aunque inicialmente de forma desordenada.

```
def load_dataset():  
    """  
    This function loads the dataset from a CSV file and returns it as a DataFrame.  
    :return: pandas DataFrame containing the dataset  
    """  
    columns = ['symboling', 'normalized_losses', 'make', 'fuel_type',  
               'aspiration', 'num_doors', 'body_style', 'drive_wheels',  
               'engine_location', 'wheel_base', 'length', 'width',  
               'height', 'curb_weight', 'engine_type', 'num_cylinders',  
               'engine_size', 'fuel_system', 'bore', 'stroke',  
               'compression_ratio', 'horsepower', 'peak_rpm',  
               'city_mpg', 'highway_mpg', 'price']  
  
    path = 'data/imports-85.data'  
  
    data = pd.read_csv(path, names=columns)  
  
    return data
```

Figura 2.0. Función para cargar el dataset en un DataFrame

Con los datos cargados en el dataframe, el siguiente paso fue limpiarlos e identificar qué información sería relevante para construir el modelo. Primero, eliminé los valores vacíos en la columna de precios (price). En este dataset, los valores faltantes estaban representados con un signo de interrogación (?), que pandas interpreta como una cadena de texto (string). Esto podría causar problemas al combinar esos caracteres con los valores numéricos de la columna. Para resolverlo, reemplacé los signos de interrogación por NaN y luego eliminé los valores vacíos.

Después de eso, convertimos las columnas con datos numéricos para evitar conflictos con las cadenas de texto. En nuestro caso, las columnas que presentaban este inconveniente eran la de precios y la de potencia del motor (horsepower).

Luego de seleccionar las columnas que usaríamos como características (features), realizamos la imputación de los valores faltantes en el resto de las columnas. Para ello, calculamos los promedios de cada columna y reemplazamos los valores vacíos por sus respectivos promedios. Con esto, completamos la fase de transformación de los datos.

```
# Calculate the mean of the columns
price_mean = np.mean(price_vals)
wheel_base_mean = np.mean(wheel_base_vals)
curb_weight_mean = np.mean(curb_weight_vals)
engine_size_mean = np.mean(engine_size_vals)
horsepower_mean = np.mean(horsepower_vals)
city_mpg_mean = np.mean(city_mpg_vals)
highway_mpg_mean = np.mean(highway_mpg_vals)
```

Figura 2.1. Obtención del promedio de las columnas

```
if row['wheel_base'] == 'nan':
    wheel_base_val = wheel_base_mean

if row['curb_weight'] == 'nan':
    curb_weight_val = curb_weight_mean

if row['engine_size'] == 'nan':
    engine_size_val = engine_size_mean

if row['horsepower'] == 'nan':
    horsepower_val = horsepower_mean

if row['city_mpg'] == 'nan':
    city_mpg_val = city_mpg_mean

if row['highway_mpg'] == 'nan':
    highway_mpg_val = highway_mpg_mean
```

Figura 2.2. Sustitución de valores vacíos por los promedios

En cuanto a la fase de carga, el uso fue limitado. Básicamente, una vez transformados los datos, los empleamos en diferentes algoritmos para las funcionalidades del proyecto, generando análisis e interpretaciones. No obstante, creo que una buena práctica habría sido guardar los datos transformados en un archivo CSV al finalizar el proceso. Esto habría evitado tener que repetir la

transformación cada vez que se ejecuta el código, reduciendo la complejidad y el tamaño del archivo final.

Ya hemos hablado sobre los primeros pasos para realizar nuestro análisis. Ahora veremos cómo usamos estos datos para obtener nuestro modelo, qué métodos y algoritmos hemos aplicado para obtener nuestros resultados.

Implementación sin el uso de framework

El primer reto de esta actividad fue implementar uno de los algoritmos vistos en clase sin el uso de un framework como sci-kit learn, etc. En clase, el profesor nos compartió una serie de varios de los algoritmos que veíamos en las clases. Yo decidí utilizar el del gradiente descendiente, pues ese fue el que más llamó mi atención y creí que podría obtener buenos resultados con él. La solución que he construido fue basada de la implementación compartida por nuestro profesor.

Las principales funciones que destacar sobre esta implementación son las siguientes:

- Escalamiento
- Hipótesis
- MSE
- Gradiente Descendiente

Función de escalamiento (Scaling)

El escalamiento es la transformación de variables predictores para que tengan una escala común. Nos ayuda a mejorar el rendimiento del modelo. Pues en un algoritmo como el del gradiente descendiente, donde las características suelen tener rangos muy diferentes, las actualizaciones de los parámetros podrían ser desiguales y afectaría a la convergencia del modelo. Las técnicas más comunes de escalamiento son:

- Normalización: Ajustamos los valores para que queden dentro de un rango específico.
- Estandarización: Transformamos la variable para que tenga un promedio de 0 y una desviación estándar de 1.

```
def scaling(samples):
    """
    This function scales the features to avoid overflow during gradient descent.

    :param samples(list): list of sample features
    :return: scaled features
    """
    samples = np.array(samples).T
    for i in range(1, len(samples)):
        average = np.mean(samples[i])
        max_value = np.max(samples[i])
        samples[i] = (samples[i] - average) / max_value
    return samples.T.tolist()
```

Figura 3.0. Función de escalamiento

En la implementación, empleamos una combinación de técnicas que nos permite procesar las características de manera más eficiente. Inicialmente, tomamos la lista de características y la convertimos en arreglos transpuestos, lo que nos facilita trabajar con cada una de ellas de forma individual. A continuación, calculamos tanto el promedio como el valor máximo de cada característica. Estos valores nos sirven de base para normalizar los datos. Realizamos nuestro escalamiento, a cada muestra le restamos el promedio y lo dividimos entre el valor máximo. De esta manera, ajustamos los datos para que estén dentro de un rango más uniforme, facilitando el análisis posterior y mejorando el rendimiento de los modelos predictivos que dependen de estos datos.

Hipótesis

Esta se refiere a la suposición sobre la relación entre las variables. En regresión línea, esta establece que la variable dependiente es una función lineal de las variables independientes. Esta se expresa de la siguiente forma:

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

```
def hypothesis(params, samples):
    """
    This function calculates the hypothesis (prediction) for linear regression.

    :param params(list): list of parameters (theta)
    :param samples(list): list of sample features
    :return: predicted value (hypothesis)
    """
    return sum(p * s for p, s in zip(params, samples))
```

Figura 3.1. Función para calcular la hipótesis

En la implementación, la función es bastante sencilla. Lo que hacemos es multiplicar cada elemento en params por su característica en samples. Y finalmente hacemos la sumatoria de estos productos para obtener nuestra hipótesis.

Mean Square Error (MSE)

Es una métrica que se usa para medir el rendimiento de un modelo de regresión. Evalúa qué tan bien se ajustan las predicciones del modelo a valores reales, calculando el promedio de los cuadrados de los errores. Su fórmula es la siguiente:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^i - \hat{y}^i)^2$$

Cuando tenemos errores muy grandes, estos tienen un fuerte impacto en el MSE, ya que los errores se elevan al cuadrado y los convierte en potenciales outliers. Cuando tenemos un MSE bajo, nos dice que las predicciones del modelo están cercanas a los valores reales. Por otro lado, un MSE alto nos dice que las predicciones no son precisas.

```

def mean_square_error(params, samples, y):
    """
    This function calculates the mean squared error

    :param params(list): list of parameters (theta)
    :param samples(list): list of sample features
    :param y(list): list of actual results
    :return: mean squared error
    """
    acum = 0
    for i in range(len(samples)):
        hyp = hypothesis(params, samples[i])
        error = hyp - y[i]
        acum += error ** 2
    return acum / len(samples)

```

Figura 3.2. Función MSE

En nuestra función, utilizamos la función de la hipótesis para calcular nuestros errores. Al resultado de la hipótesis le restamos su valor real de Y, y con eso obtenemos el valor de nuestro error, estos los elevamos al cuadrado y los vamos acumulando para finalmente dividirlos entre el número de muestras de nuestro modelo.

Gradiente Descendiente

El gradiente descendiente es un algoritmo de optimización utilizado para minimizar funciones de costo en problemas de aprendizaje automático en la regresión. Su objetivo es encontrar los parámetros del modelo que minimizan el error entre las predicciones del modelo y los valores reales.

La fórmula para actualizar los parámetros del modelo es el siguiente:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

El gradiente es el vector de derivadas parciales de la función de costo con respecto a cada parámetro, se mide cómo cambia la función de costo si se modifica un parámetro específico. Los parámetros del modelo se actualizan en la dirección opuesta al gradiente para reducir la función de costo. Repetimos este proceso por

un número definido de iteraciones o que la función de costo converja a un valor lo suficientemente bajo.

```
def descending_gradient(params, samples, y, alpha):  
    """  
    This function performs the gradient descent optimization.  
    :param params(list): list of parameters (theta)  
    :param samples(list): list of sample features  
    :param y(list): list of actual results  
    :param alpha(float): learning rate  
    :return: updated parameters after one step of gradient descent  
    """  
    temp = params.copy()  
    for j in range(len(params)):  
        acum = sum((hypothesis(params, sample) - y[i]) * sample[j] for i, sample in enumerate(samples))  
        temp[j] = params[j] - alpha * (1 / len(samples)) * acum  
    return temp
```

Figura 3.3. Algoritmo del gradiente descendiente

En la implementación, utilizamos una variable temporal para almacenar los nuevos valores después de realizar una actualización, de ese modo evitamos modificar la lista original durante la iteración. Recorremos cada parámetro y calculamos el gradiente usando la suma de los errores multiplicados por el valor de la característica correspondiente. Los parámetros en temp se actualizan después de restar el producto del gradiente y la tasa de aprendizaje, y finalmente regresamos esos parámetros actualizados.

Coeficiente R²

El coeficiente de determinación es una medida estadística que indica qué tan bien se ajusta el modelo a los datos. Nos muestra la proporción de la variabilidad de la variable dependiente que puede ser explicada por las variables independientes del modelo. Su fórmula es:

$$R^2 = 1 - \frac{\text{Suma de los errores cuadráticos}}{\text{Suma total de los cuadrados}}$$

Un valor de R² de alto nos dice que el modelo puede explicar la mayor parte de la variabilidad de los datos, lo que indica un buen ajuste. Por otro lado, un valor de R² bajo nos dice que el modelo no puede explicar con precisión la mayor parte de

la variabilidad. Lo que nos puede indicar que nuestro modelo no es útil, o incluso decirnos que es peor usar el modelo que usar la media como predicción.

```
def r_squared(predicted_y, real_y):  
    """  
    This function calculates the R-squared value to evaluate the model.  
    :param predicted_y(list): predicted values from the model  
    :param real_y(list): actual values from the data  
    :return: R-squared value  
    """  
    mean_y = np.mean(real_y)  
    ss_total = sum((real_y - mean_y) ** 2)  
    ss_res = sum((real_y - predicted_y) ** 2)  
    return 1 - (ss_res / ss_total)
```

Figura 3.4. Función para determinar el R^2 del modelo

La implementación de nuestra función resume lo que hemos explicado anteriormente. Calculamos el coeficiente de determinación al realizar la suma de los valores reales menos el promedio, elevados al cuadrado, y la suma de los valores reales menos los valores predichos, elevados al cuadrado, y hacemos la resta de uno menos el cociente de la suma total de errores cuadráticos sobre la suma total de los cuadrados.

Resultados (Implementación sin framework)

A lo largo de este ensayo, hemos hablado sobre el dataset utilizado, que características tiene, y cómo lo usaríamos. También hemos hablado de cómo hemos construido nuestras funciones para tratar de predecir el precio de un automóvil basado en las características mencionadas anteriormente.

Ahora vamos a ver los resultados que hemos obtenido en esta primera ejecución, veremos cuales son los valores de R^2 para train y validation; el valor de nuestro MSE y una visualización de la gráfica de train y validation. Hemos establecido un total de Epochs de 1000 con una tasa de aprendizaje de 0.9.

Resultados MSE

Epoch: 992	Train Error: 16160290.702818686	Validation Error: 22046250.958446644
Epoch: 993	Train Error: 16160030.951961523	Validation Error: 22046393.196611874
Epoch: 994	Train Error: 16159771.723808032	Validation Error: 22046536.514486253
Epoch: 995	Train Error: 16159513.016855815	Validation Error: 22046680.90737343
Epoch: 996	Train Error: 16159254.829608174	Validation Error: 22046826.370598372
Epoch: 997	Train Error: 16158997.160574008	Validation Error: 22046972.899507172
Epoch: 998	Train Error: 16158740.008267801	Validation Error: 22047120.48946707
Epoch: 999	Train Error: 16158483.371209666	Validation Error: 22047269.13586623
Epoch: 1000	Train Error: 16158227.247925203	Validation Error: 22047418.8341137

Figura 4.0. Resultados de MSE

En los últimos epochs del modelo, observamos que el error en Train disminuye, lo que indica que el modelo está aprendiendo mejor los patrones en los datos de entrenamiento. Sin embargo, también notamos que el error en Validation aumenta, lo cual sugiere que el modelo está perdiendo su capacidad de generalizar a datos nuevos. Esto ocurre porque, en lugar de continuar aprendiendo los patrones subyacentes, el modelo comienza a memorizar o ajustarse al ruido del conjunto de entrenamiento, lo que reduce su rendimiento en los datos de validación. Este comportamiento indica la presencia de **overfitting**.

Resultados R^2

```
Train R^2: 0.7568627075082426
Validation R^2: 0.46353076986757813
```

Figura 4.1. Resultados de los coeficientes R^2

Estos son los resultados de nuestros coeficientes R^2 en Train y Validation. La diferencia entre estos dos nos revela puntos importantes sobre el desempeño del modelo:

- Train $\rightarrow 0.7568$: Significa que el modelo puede explicar aproximadamente el 75.68% de la variabilidad en los datos de entrenamiento. Es un ajuste relativamente bueno, pues sugiere que el modelo está capturando los patrones presentes en esos datos.
- Validation $\rightarrow 0.4635$: Significa que el modelo solo explica el 46.35% de la variabilidad en los datos nuevos. Este valor es más bajo que el de entrenamiento, nos dice que el modelo no está generalizando bien a datos nuevos.

Esta diferencia nos dice que, aunque el modelo se ajusta relativamente bien a los datos que le hemos dado, pierde precisión cuando se enfrenta a datos nuevos, indicando una capacidad limitada de generalización. Como mencionábamos anteriormente, el modelo está ajustándose a detalles específicos de los datos de entrenamiento, pero estos detalles no son representativos de los datos de validación. Por ende, el rendimiento en validación es peor de lo que se esperaba.

Visualización de los resultados (gráficas)

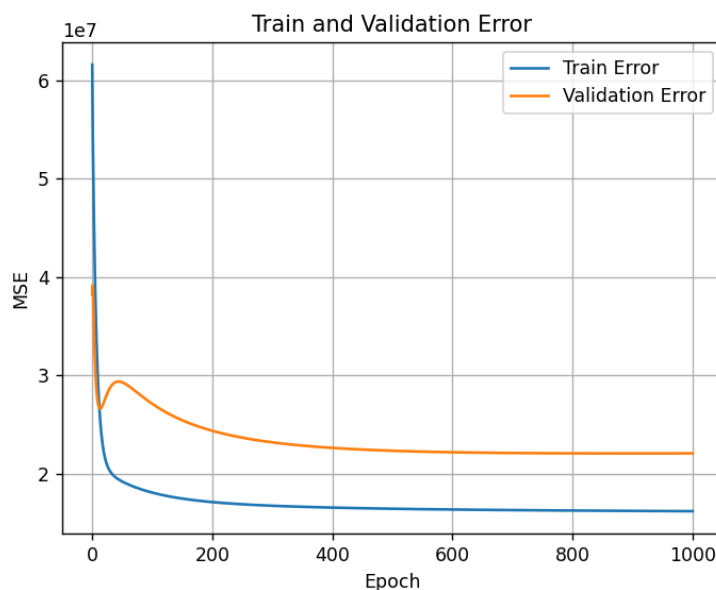


Figura 4.2. Gráficas de los errores en train y validation

La gráfica muestra el comportamiento del error en Train y en Validation a lo largo de los epochs del modelo.

Por un lado, en Train el error de entrenamiento es bastante alto al principio, sugiriendo que el modelo tiene un bajo rendimiento en los primeros pasos del entrenamiento. Sin embargo, conforme el entrenamiento avanza, el error disminuye de forma rápida y se estabiliza alrededor de los 300 epochs. Como hemos mencionado, esta reducción sugiere que el modelo mejora su capacidad para ajustarse a los datos de entrenamiento.

Por otro lado, en Validation al principio, este también disminuye, lo que indica que el modelo mejoraba su rendimiento en datos no vistos. Sin embargo, poco antes de los 100 epochs, el modelo deja de disminuir y aumenta en algunos puntos. Como mencionamos, nuestro modelo está sobre ajustado a los datos de entrenamiento. Por lo que, aunque el modelo mejore en su rendimiento con los datos de entrenamiento, pierde su capacidad para generalizar datos nuevos.

Conclusiones y áreas de oportunidad

En esta primera implementación, lo que podemos concluir, es lo que hemos mencionado a lo largo de estos resultados. A medida que los epochs avanzan, el modelo está aprendiendo demasiado bien los datos de entrenamiento, incluyendo los patrones específicos y el ruido, pero pierde su capacidad para generalizar a nuevos datos. Lo que nos dice que nuestro modelo tiene overfitting. Unas posibles formas de mejorar el desempeño del modelo es aplicar técnicas de regularización como Ridge o Lasso para evitar que el modelo se ajuste demasiado a los datos de entrenamiento. Otra solución podría ser detener el entrenamiento cuando el error de validación comience a aumentar, antes de que el modelo se sobreajuste.

Implementación con el uso de framework

Para esta segunda implementación. Hemos vuelto a aplicar el algoritmo del gradiente descendiente pero ahora usando un framework de desarrollo, Sci-kit learn. Esta librería nos permite usar la función del gradiente descendiente con el uso de una sola línea en lugar de generar toda una función completa, además de las otras funciones usadas en la implementación pasada.

Como en la implementación sin framework, hemos vuelto a aplicar la técnica de ETL para la extracción de datos, siendo esa la única función que tenemos presente en nuestro código y donde se realiza lo mismo que hicimos anteriormente.

```
def load_dataset():  
    """  
    This function loads the dataset from a CSV file and returns it as a DataFrame.  
    :return: pandas DataFrame containing the dataset  
    """  
    columns = ['symboling', 'normalized_losses', 'make', 'fuel_type',  
               'aspiration', 'num_doors', 'body_style', 'drive_wheels',  
               'engine_location', 'wheel_base', 'length', 'width',  
               'height', 'curb_weight', 'engine_type', 'num_cylinders',  
               'engine_size', 'fuel_system', 'bore', 'stroke',  
               'compression_ratio', 'horsepower', 'peak_rpm',  
               'city_mpg', 'highway_mpg', 'price']  
  
    data = pd.read_csv('data/imports-85.data', names=columns)  
    return data
```

Figura 5.0. Función para cargar el dataset en un DataFrame

A continuación, veremos las funciones destacadas de nuestra implementación. En este caso, no veremos lo que es cada una, sino nos iremos directamente a su funcionamiento, pues ya hemos explicado lo que son en la implementación sin framework.

Función de escalamiento

```
scaler = StandardScaler()  
train_x = scaler.fit_transform(train_x)  
val_x = scaler.transform(val_x)  
test_x = scaler.transform(test_x)
```

Figura 5.1. Uso de la función de escalamiento de Sci-kit learn

En esta implementación, creamos un objeto tipo StandardScaler, este estandariza los datos ajustándolos para que tengan una media de 0 y una desviación estándar de 1. Después el escalador aprende la media y la desviación estándar de los datos de entrenamiento, prueba y validación y luego aplica la transformación para estandarizar cada uno de estos datos. Asegurando que los conjuntos de datos estén en la misma escala.

Gradiente Descendiente

```
model = SGDRegressor(max_iter=epochs, eta0=learning_rate, tol=0.0001)
model.fit(train_x, train_y)
```

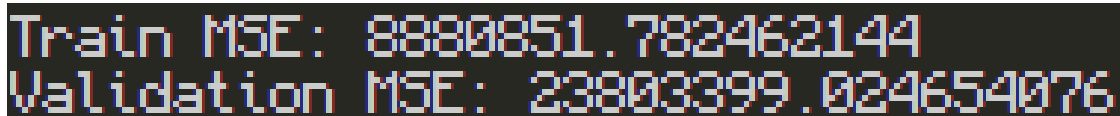
Figura 5.2. Uso del gradiente descendiente de Sci-kit learn

Para nuestro modelo, la función de SGDRegressor implementa una regresión lineal usando el gradiente descendiente. En este caso, los parámetros utilizados son el número de epochs para el algoritmo, establecemos como eta0 nuestra tasa de aprendizaje. Y el parámetro tol=0.0001 especifica el criterio de tolerancia para detener el proceso de entrenamiento si la mejora en el error del modelo es menor que este valor en dos iteraciones consecutivas. Posterior a eso, entrenamos nuestro modelo usando los valores de entrenamiento, ajustando sus parámetros para minimizar el error en los datos de entrenamiento.

Resultados (Implementación con framework)

Ya hemos hablado sobre las funciones implementadas en esta implementación. Al igual que la implementación anterior, intentaremos predecir el precio de un auto basado en las características mencionadas al principio. Veamos las diferencias entre los resultados de la implementación sin framework con la implementación con framework.

Resultados MSE

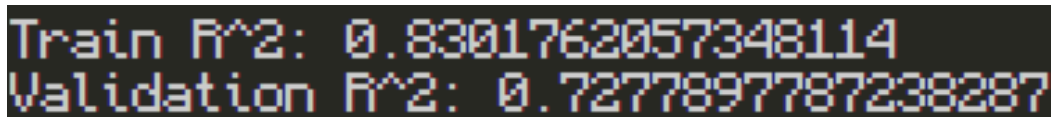


```
Train MSE: 8880851.782462144
Validation MSE: 23803399.024654076
```

Figura 6.0. Resultados de los MSE para Train y Validation

Similar a la implementación anterior, podemos ver una diferencia enorme en los MSE de los datos de entrenamiento y validación. Al igual que antes, estas diferencias nos sugieren que el modelo tiene **overfitting**, pues nuestro modelo esta aprendiendo demasiado bien los detalles y el ruido del conjunto de entrenamiento, pero no generaliza bien a nuevos datos. Además, ambos valores son altos, lo que nos dice que el modelo puede no estar capturando bien la relación entre las variables.

Resultados R^2



```
Train R^2: 0.8301762057348114
Validation R^2: 0.7277897787238287
```

Figura 6.1. Resultados de R^2 para Train y Validation

En estos resultados a diferencia de la implementación, podemos ver una diferencia significativa con los valores de nuestros coeficientes.

- Train \rightarrow 0.8301: Nos dice que el modelo explica aproximadamente el 83.01% de la variabilidad en los datos de entrenamiento. Este valor sugiere que el modelo captura una gran parte de la relación entre las variables en el conjunto de entrenamiento.

- Validation $\rightarrow 0.7277$: Nos dice que el modelo explica aproximadamente el 72.77% de la variabilidad en el conjunto de validación. Sigue siendo más bajo que el R^2 de entrenamiento, sin embargo, la diferencia entre estos es mucho menor a como era en la primera implementación. Y este valor aún entra en el indicador de que el modelo generaliza relativamente bien a datos nuevos.

Estos valores sugieren que tenemos un buen ajuste en el modelo, pues este está capturando una cantidad significativa de la variabilidad tanto en Train como en Validation. No obstante, la diferencia entre el R^2 de Train y Validation indica que el modelo se ajusta mejor a los datos de entrenamiento que los de validación, aunque no es una diferencia tan grande como en la primera implementación, aún sugiere que puede existir overfitting.

Resultados Bias

Bias: 12.759241583333415

Figura 6.2. Resultados del bias

Al considerar el rango de nuestros datos, el valor del bias indica que este es relativamente bajo. Esto sugiere que el modelo está capturando adecuadamente los patrones en los datos de entrenamiento. El hecho de que el bias sea positivo nos revela que el modelo tiende a sobreestimar las predicciones en comparación con los valores reales, lo que significa que, en promedio, las predicciones son ligeramente más altas de lo esperado.

Visualización de los resultados (gráficas)

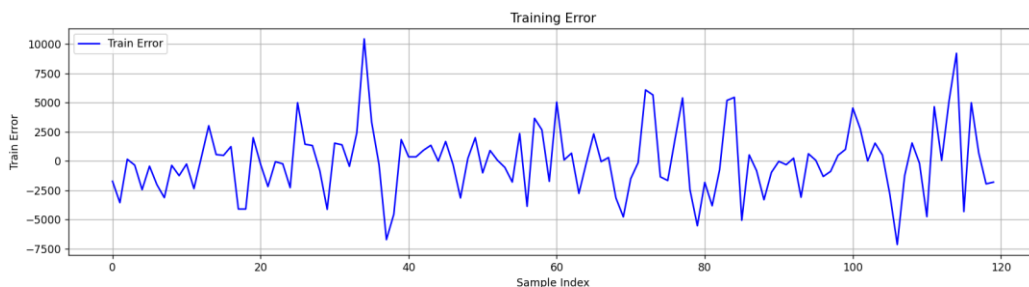


Figura 6.3. Gráfica de Training Error

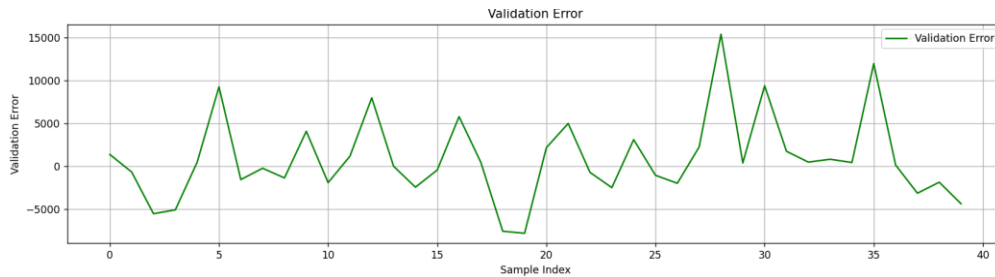


Figura 6.4. Gráfica de Validation Error

A diferencia de la implementación anterior, hemos separado nuestras gráficas debido a las diferentes longitudes de los conjuntos de datos.

- Training error: El error varía considerablemente a lo largo de las muestras. La amplitud de los errores oscila entre -7500 y 1000, esto nos dice que algunas predicciones están subestimadas y otras están sobreestimadas. Por otro lado, el error promedio se mantiene alrededor de cero, lo que sugiere que el modelo está equilibrado en cuanto a las predicciones subestimadas y sobreestimadas.
- Validation error: El error varía también, con errores más pronunciados que en Training, alcanzando los 15000 y -5000 en algunos puntos. Los picos más altos nos dicen que el modelo falla en ciertos puntos de validación, lo que sugiere un problema de overfitting, pues el modelo se comporta de peor manera en los datos de validación. Hay unos momentos en el que el error es cercano a cero, pero hay mayor variabilidad y desviación en comparación con Training.

Conclusiones y áreas de oportunidad

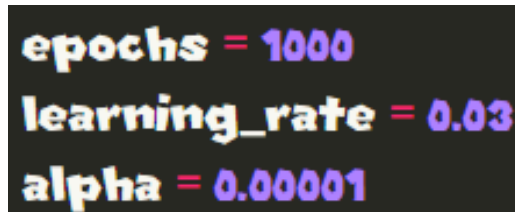
En esta segunda implementación, similar al caso anterior, el modelo se ajusta bien a los datos de entrenamiento, con una variabilidad y errores en los datos de validación. Lo que nos dice que nuestro modelo tiene overfitting, pues el modelo está aprendiendo demasiado bien los datos de entrenamiento, pero no generaliza bien a datos no vistos. Una posible forma de mejorar el rendimiento sería aplicar una técnica de regularización como Ridge para evitar el sobreajuste.

Implementación ajustada

En este ensayo, hemos analizado y diagnosticado los dos modelos construidos con el gradiente descendiente, siendo el del framework el que mejores resultados dio. Considero que debido a la implementación sencilla que fue el modelo con framework, ese debía ser al que debíamos aplicar técnicas de regularización para mejorar el modelo. Para ello construí un nuevo archivo para conservar el original y poder comparar, y le hemos hecho ligeros ajustes a la implementación para implementar la técnica de Regularización de L2. Dado a que esta tercera implementación, son modificaciones del archivo original, vamos a destacar los siguientes puntos.

Variables globales

En esta tercera implementación, hemos añadido una nueva variable para definir la fuerza de la regularización.



```
epochs = 1000
learning_rate = 0.03
alpha = 0.00001
```

Figura 7.0. Variable Alpha para definir la fuerza de regularización

Un valor mayor de Alpha aumentará la regularización y podría mejorar el rendimiento del modelo en datos no vistos. Sin embargo, puede también reducir la capacidad del modelo para ajustarse a los datos de entrenamiento.

Regularización L2 (Ridge)

Esta técnica permite prevenir el sobre ajuste del modelo y mejorar su capacidad de generalización. Esta añade una penalización al tamaño de los coeficientes del modelo durante el proceso de entrenamiento. La función de pérdida con regularización L2 es la siguiente:

$$Pérdida = Error + \lambda \sum_{i=1}^n \beta_i^2$$

La regularización penaliza los coeficientes grandes, y ayuda a reducir el sobreajuste. Además, esta tiende a hacer que los coeficientes sean más pequeños y distribuidos, esto puede mejorar la estabilidad del modelo y capacidad para generalizar.

```
model = SGDRegressor(max_iter=epochs, eta0=learning_rate, tol=0.0001, penalty='l2', alpha=alpha)
model.fit(train_x, train_y)
```

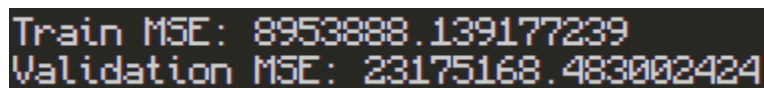
Figura 7.1. Modificación del modelo para realizar la regularización L2

Para la implementación, modificamos los parámetros de la llamada a la función SGDRegressor, añadiendo el parámetro penalty y Alpha. En ellos establecemos el tipo de regularización a utilizar, en este caso L2, y establecemos la variable Alpha que habíamos definido anteriormente para la fuerza de la regularización.

Resultados (Implementación ajustada)

Ya hemos revisado las modificaciones realizadas a nuestra implementación. Con estos ajustes en mente, reconstruimos nuestro modelo con las nuevas configuraciones y parámetros. Analizaremos los resultados obtenidos para evaluar el impacto de las modificaciones y determinar si hemos logrado mejorar la capacidad de generalización y la precisión del modelo en comparación con la versión anterior.

Resultados MSE

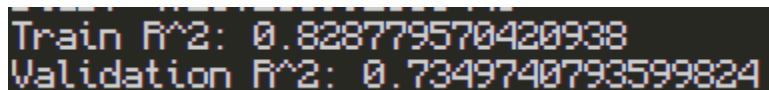


```
Train MSE: 8953888.139177239
Validation MSE: 23175168.483002424
```

Figura 8.0. Resultados MSE para Train y Validation

En este caso, los resultados para el MSE indican que aún tenemos **overfitting**, pues los datos de validación siguen siendo mucho mayor que los datos de entrenamiento. Lo que nos dice que el modelo ha aprendido demasiado bien los patrones en Training y no esta generalizando bien a datos no vistos.

Resultados R²



```
Train R^2: 0.828779570420938
Validation R^2: 0.7349740793599824
```

Figura 8.1. Resultados R² para Train y Validation

En este caso, los coeficientes de determinación son muy similares a los anteriores, con muy pocos cambios.

- Train → 0.8287: Nos indica que aproximadamente 82.87% de la variabilidad en los datos es explicada por el modelo. Es un valor decente, y nos dice que modelo se ajusta bien a los datos de entrenamiento.
- Validation → 0.7349: Nos dice que el 73.49% de la variabilidad en los datos de validación se puede explicar con el modelo. Aunque es más bajo que el R² de Train, sigue siendo un valor relativamente decente, y nos dice que el modelo puede generalizar de buena forma a datos no vistos.

Ambos valores son lo suficientemente buenos para decir que el modelo captura una buena proporción de la variabilidad de los datos. No obstante, la disminución notable en el R^2 de Validation nos dice que el modelo sigue con overfitting.

Resultados Bias

Bias: 4.137150992006445

Figura 8.2. Resultados del Bias para el modelo

El bias obtenido nos dice que, en promedio, las predicciones del modelo están por encima de los valores reales. Dado a que es positivo, nos dice que el modelo tiende a sobreestimar los valores del modelo. Considero que es un bias bajo, en relación con el rango de los datos, pues en comparación con el bias anterior, este es mucho menor, lo que nos dice que las predicciones del modelo están más cerca de los valores reales en promedio.

Visualización de los resultados (gráficas)

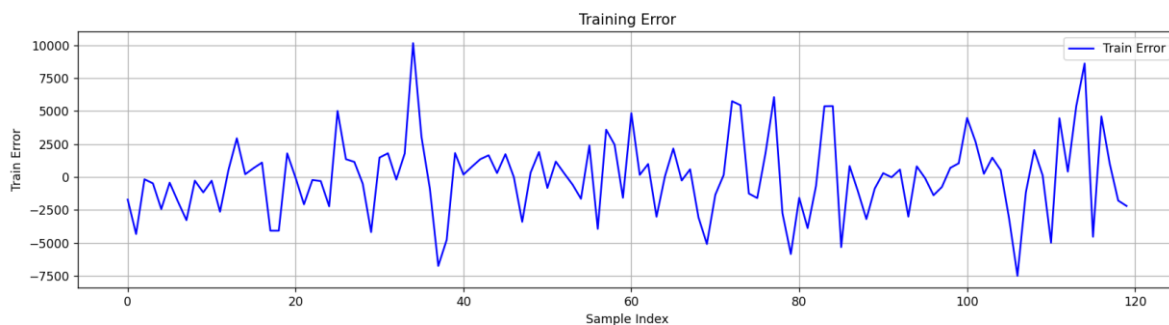


Figura 8.3. Gráfico de Training

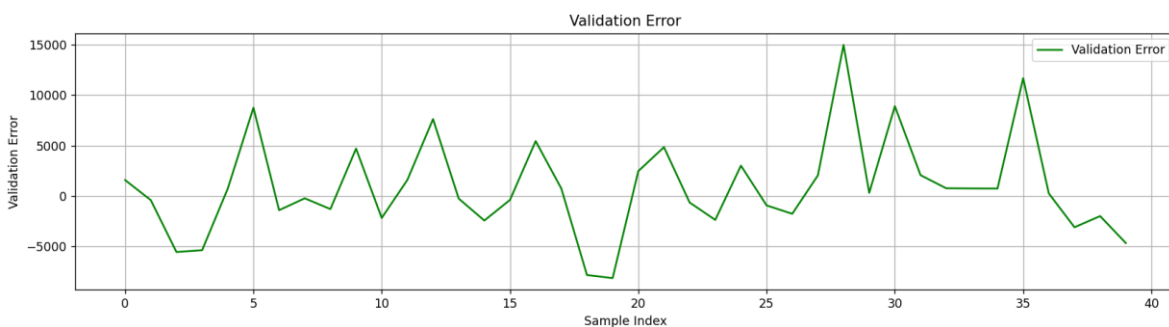


Figura 8.4. Gráfico de Validation

Al igual que nuestra implementación anterior, hemos separado ambas gráficas debido a la diferencia de tamaños en los Train y Validation.

- Training error: El error actúa de manera desordenada, no hay una tendencia clara de la disminución del error. Esto sugiere que puede sugerir que el modelo está encontrando mucho ruido en los datos o que los hiper parámetros no están bien ajustados. Sin embargo, estos errores están alrededor de cero, eso nos dice que el modelo está suficientemente equilibrado.
- Validation error: El error también es variable, con picos altos y caídas bruscas. Esto podría sugerir problemas de overfitting, pues el modelo muestra dificultades para generalizar a nuevos datos.

Comparación de resultados

Celdas en verde → Mejor resultado

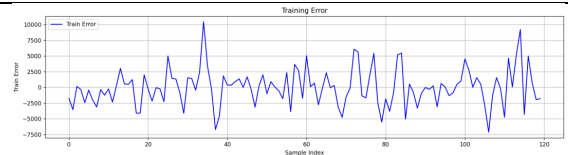
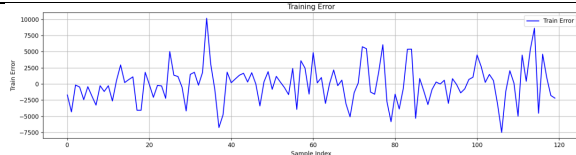


	Implementación 2	Implementación 3
MSE Train	8880851.78	8953888.13
MSE Validation	23803399.02	23175168.48
R² Train	0.8301	0.8287
R² Validation	0.7277	0.7349
Bias	12.75	4.13
Gráficos		
Implementación 2		Implementación 3
		
		

Tabla 1.0. Comparación de los resultados entre implementación 2 e implementación 3

En esta tabla, lo que más se puede distinguir es que los cambios son mínimos. En la implementación 3 logramos reducir el valor del MSE y R² para Validation, lo que

le permitió al modelo poder generalizar un poco más sobre datos no vistos. Por otro lado, la implementación de la regularización, afectó ligeramente al MSE y R^2 de Training, haciendo que el error de entrenamiento sea mayor y tengamos un coeficiente más bajo.

Por otra parte, en ambas implementaciones, podemos ver que nuestras gráficas han quedado prácticamente iguales, las diferencias son mínimas y muy difícil de distinguir.

Conclusión (Comparativa de resultados)

Esta comparativa nos permite concluir que la implementación 3, con el uso de regularización. Hemos logrado obtener mejores resultados. Pues la capacidad del modelo para aprender datos de entrenamiento, con su capacidad para generalizar a datos nuevos es más equilibrada. Y tenemos un bias más bajo lo que nos dice que los datos predichos por el modelo difieren menos en comparación a la implementación 2.

Sin embargo, estos cambios son mínimos, con apenas un aumento o disminución de una unidad en el MSE y R^2 tanto para Train y Validation. Esto sugiere que, para mejorar el rendimiento del modelo, es necesario realizar otros ajustes como Cross-Validation, o cambiar los features utilizados en el modelo.

Conclusiones generales

A lo largo de este proyecto hemos hablado sobre las diferentes implementaciones del algoritmo del gradiente descendiente, y hemos diagnosticado cada uno de los modelos construidos. También, hemos explicado las diferentes técnicas utilizadas para la construcción de las implementaciones. Cómo funcionan y cómo las hemos aplicado en nuestras soluciones.

En los tres modelos obtuvimos overfitting. Lo que nos dice que el modelo aprende muchísimo de los datos de entrenamiento. Pero no puede generalizar para datos nuevos.

En la implementación sin framework, obtuvimos el modelo más bajo, con una diferencia entre Train y Validation significativa, pues en Train teníamos un coeficiente relativamente bueno, pero en Validation, este era ineficiente. Por lo que el modelo no era lo suficientemente efectivo para el objetivo mencionado anteriormente.

En la implementación con framework, este resultado cambio, generando un modelo mucho más preciso que el anterior. Aún tenía diferencias entre Train y Validation, pero no tan significativas como el modelo anterior. Los coeficientes de estos modelos eran lo suficientemente buenos para decir que el modelo es efectivo para el objetivo principal.

El ajuste del modelo con framework, logro aumentar ligeramente la precisión del modelo, generando un resultado un poco más adecuado para la predicción, y reduciendo ligeramente el overfitting que tenía el modelo anterior. Sin embargo, el aumento fue mínimo, lo que sugiere que se tendrían que tomar enfoques diferentes para mejorar el rendimiento del modelo.

Esto nos permite concluir que el mejor modelo para predecir el precio de un automóvil basado en las características mencionadas al principio es el tercero, pues este se acerca más a un resultado preciso con un ajuste relativamente equilibrado entre los datos de entrenamiento con los de validación.

Referencias

Schlimmer J. (1987, 19 de mayo). *1985 Auto Imports Database*. Kaggle.

<https://www.kaggle.com/datasets/fazilbtopal/auto85>

Ortega C. (s.f.). *Análisis de regresión: Qué es, tipos y cómo realizarlo*.

QuestionPro. <https://www.questionpro.com/blog/es/analisis-de-regresion/>

Moral I. (2006, 3 de diciembre). *Modelos de regresión: lineal simple y regresión*

logística. RevistasEden. <https://www.revistaseden.org/files/14->

[CAP%2014.pdf](https://www.revistaseden.org/files/14-CAP%2014.pdf)

Universidad Complutense de Madrid (2013). *Modelo lineal general: hipótesis y*

estimación. <https://www.ucm.es/data/cont/docs/518-2013-10-25->

[Tema_3_1_EctrGrado.pdf](https://www.ucm.es/data/cont/docs/518-2013-10-25-Tema_3_1_EctrGrado.pdf)

Gupta A. (2024, 26 de junio). *Mean Squared Error: Overview, Examples, Concepts*

and More. Simplilearn. <https://www.simplilearn.com/tutorials/statistics->

[tutorial/mean-squared-error](https://www.simplilearn.com/tutorials/statistics-tutorial/mean-squared-error)

Sotaquirá M. (2018, 2 de julio). *¿Qué es el Gradiente Descendiente?*

Codificandobits. <https://www.codificandobits.com/blog/el-gradiente->

[descendente/](https://www.codificandobits.com/blog/el-gradiente-descendente/)

Arquez M. (2020, 29 de marzo). *Regularización*. RPubS.

<https://rpubs.com/arquez9512/591669>