Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus: Querétaro

“Momento de Retroalimentación: Módulo 2 Implementación de una técnica de aprendizaje máquina sin el uso de un framework. (Portafolio Implementación)”

Materia:

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

Alumno:

Juan Pablo Cabrera Quiroga - A01661090

Fecha de entrega:
24 de Agosto del 2024

Leyenda:

Como alumnos del Tec y “Apegándose al código de Ética de los estudiantes del Tecnológico de Monterrey, nos comprometemos a que nuestra actuación en esta actividad de evaluación esté regida por la integridad académica. En congruencia con el mismo realizaremos esta actividad de forma honesta y personal, para reflejar a través de ella nuestros conocimientos y competencias”

ÍNDICE

Introducción.

En este ensayo detallaré el proceso completo que he llevado a cabo para desarrollar un modelo de regresión lineal, el cuál me permite realizar predicciones. Explicaré desde la extracción y transformación de datos (ETL) hasta la implementación y evaluación del modelo. A través de este documento, presentaré tanto los aspectos teóricos como prácticos, destacando las técnicas y herramientas vistas en clase y que son utilizadas en cada etapa.

Es importante destacar que el proceso que he llevado a cabo no es un proceso lineal. Como nos fuimos dando cuenta en las clases, durante la construcción de un modelo, es muy común que sea necesario retroceder a etapas previas de la construcción, ajustar parámetros, o incluso avanzar a etapas siguientes para poder validar decisiones tomadas anteriormente, todo esto con el fin de **analizar e interpretar los datos**.

ETL.

¿Qué es ETL y de qué nos sirve?

ETL es un proceso fundamental en la gestión y análisis de datos. Significa **Extraer**, **Transformar** y **Cargar** (en inglés es Extract, Transform y Load, por eso ETL).

Extraer implica tomar datos de diversas fuentes, como bases de datos, archivos, o APIs.

En la fase de **Transformación**, estos datos se limpian, se ajustan , y se formatean para que sean consistentes y útiles para el análisis. Esto incluye tareas como corregir errores, llenar valores faltantes, o convertir datos categóricos en numéricos. Esto es algo que se verá de forma más detallada en las siguientes secciones.

Finalmente, en la etapa de **Carga**, los datos transformados se guardan y luego se suben o transfieren al modelo donde serán utilizados. Se pueden mandar como JSON, CSV, guardar en un servidor, etc.

El proceso ETL nos sirve porque permite que los datos que inicialmente pueden estar desorganizados, incompletos o que traigan información que no nos parezca relevante, se conviertan en información precisa y bien estructurada. Esto es crucial para poder realizar análisis confiables, construir nuestro modelo y tomar decisiones informadas basadas en datos. Es muy importante entender que sin un proceso ETL adecuado, es muy probable que los análisis resulten inexactos o que se pierda información valiosa en el camino.

Descripción del Dataset utilizado.

El dataset que elegí para este proyecto fue extraído de la página de Kaggle. El dataset que elegí se centra en las especificaciones de diferentes automóviles de 1985, incluyendo características como el tipo de combustible, marca, el número de cilindros, la potencia del motor y el precio del vehículo. Después de haber corroborado con mi profesor, concordamos en que este conjunto de datos me sirve para trabajar en el proyecto, ya que incluye varios atributos (columnas) categóricos como numéricos. Además de tener una cantidad de instancias (filas) suficiente como para poder manipularlo y realizar análisis.

Extracción de datos.

Previamente mencioné que el proceso de ETL comienza con la extracción de datos. En mi caso, utilicé un archivo .data (que contiene el dataset) que descargué de la página de Kaggle. Como ya expliqué antes, este dataset contenía las especificaciones de automóviles de 1985. Durante este proceso de extracción, me centré principalmente en escoger un dataset que fuera relevante para mi y lo que quería lograr con el modelo, que es tener predicciones de datos.

De la página de Kaggle, además de poder descargar el set de datos, también pude descargar un archivo .names. Este es un archivo esencial para entender el significado de cada atributo y tratar de asegurar un correcto análisis de los datos. La información contenida en el archivo .names me permitió identificar y etiquetar adecuadamente las columnas al momento de cargar los datos, identificar también para que se había usado el anteriormente el dataset y sobre todo una parte muy importante al final de este archivo que me ayudaba a identificar los valores faltantes en el dataset.

Ahora bien, centrándome en lo que está en mi código, utilicé la biblioteca de **Pandas** para cargar el dataset en un DataFrame, con los nombres de los features. Esto me permitió que en los siguientes pasos pudiera trabajar de manera eficiente con los datos.

Transformación de datos.

Esta etapa para mí es la más importante de todo el proceso de ETL. Ya que en esta etapa es en la que se realiza la limpieza, estructuración y preparación de los datos para su posterior análisis. En mi proyecto, implementé varias técnicas de preprocesamiento para asegurar que los datos estuvieran en el mejor estado posible para el modelado predictivo. Las presento a continuación.

Imputación de valores faltantes.

La primera técnica que apliqué tiene que ver con el cálculo de valores faltantes. En el dataset original, los valores faltantes estaban representados por el símbolo "?", esto nos lo menciona el archivo .name que mencioné anteriormente. Después pude sacar la cantidad de datos faltantes

gracias a una función que hice con la librería de Pandas. Ahora bien, para facilitar el procesamiento posterior, reemplacé estos símbolos por NaN utilizando la función replace de pandas:

```
Python
df.fillna(df.mean(), inplace=True)
```

Imputación de datos por media y moda.

La segunda técnica que apliqué tiene que ver con cómo manejo esos valores faltantes. A esto se le conoce como “Imputación de datos”. En este proyecto, apliqué dos métodos diferentes de imputación:

1. Imputación por moda:

   Para todas las columnas excepto 'Price', utilicé la imputación por moda. La moda es el valor más frecuente en cada columna, y decidí usar este método porque es muy útil cuando se trabaja con datos categóricos o cuando hay un valor que se repite con mayor frecuencia en una columna. En mi caso, muchas de las columnas contenían datos categóricos, como el número de puertas o el tipo de combustible, por esta razón me pareció adecuado utilizar la moda. Este es el código que utilicé para lograrlo:

```
Python
mode_imputer = other_columns.mode().iloc[0] other_columns_imputed =
other_columns.fillna(mode_imputer)
```

2. Imputación por media:

   Por otro lado, para 'Price', utilicé la imputación por media. Esto significa que reemplacé los valores faltantes con el promedio de los valores existentes en esa columna. Decidí hacer esto porque 'Price' es un dato numérico, y la media es una buena forma de representar el valor típico en este tipo de datos.

De esta forma es como lo realicé en el código:

```
Python
price_imputed = price_column.fillna(price_column.mean())
```

Este paso de realizar la imputación de los datos, ya se me complicó bastante. Lo que sucedió es que en mis columnas con datos faltantes, tenía yo datos que no eran numéricos (strings). Entonces realmente no podía obtener valores. Esto me lleva a mi tercera técnica de preprocesamiento.

Conversión de datos categóricos a numéricos.

En mi caso, por el dataset que elegí, este contenía muchos valores que no eran numéricos. Como lo vimos en clase, la mejor forma para solucionar esto es hacer diccionarios de datos.

La forma en como yo lo hice fue que asigné un valor numérico a cada categoría dentro de las columnas correspondientes. Por ejemplo, en la columna "Num-of-doors", los valores "two" y "four" fueron transformados en 2 y 4, respectivamente.

Esta técnica de conversión no sólo me ayudó a poder resolver el problema de los datos faltantes en columnas categóricas (y poder aplicar la imputación de datos), sino que también permitió que todos los datos estuvieran en el mismo formato (numérico). Este paso fue fundamental para asegurar que mi modelo pudiera procesar y analizar correctamente todas las variables involucradas.

Ahora bien, después de aplicar las técnicas de imputación y conversión de datos, me di cuenta de que los datos que guardé en mi data frame eran demasiado variables. Por ejemplo, en una columna tenía valores de peso que no bajaban de los 1,500 o RPM que llegaba hasta los 5,000 . En contraste, otra columna, como la del número de puertas, solo contenía valores de 2 o 4.

Esta gran diferencia en los rangos de valores entre las columnas me empezó a generar problemas en mi modelo de regresión lineal, ya que algunas características podrían influir más en el modelo solo porque sus números son mayores. Como lo vimos en clase, los números más pequeños tienden a ser más irrelevantes, a desaparecer.

Escalamiento de datos.

Es por esta razón que utilicé otra técnica que se conoce como escalamiento de datos. Específicamente, implementé el “StandardScaler” de la biblioteca sklearn. Esta técnica transforma los datos de manera que cada característica tenga una media de 0 y una desviación estándar de 1. Esta es la única ocasión en la que hago uso de una biblioteca en toda mi implementación y esto lo hago porque sklearn está diseñado para manejar este tipo de transformaciones con precisión y sin riesgo de pérdida de datos o errores de cálculo.

La verdad es que al realizar el escalamiento manualmente, existe la posibilidad de introducir errores que podrían alterar la integridad de los datos o afectar negativamente el rendimiento

del modelo. Es por eso que en un paso tan importante decidí hacerlo con un framework que me asegurara que el proceso no iba a salir mal.

Después de aplicar el escalamiento, todas mis características ahora están en una escala comparable, lo que debería mejorar significativamente el rendimiento y la estabilidad de mi modelo de regresión lineal. Llevándome al último paso que realicé de este proceso.

Carga de datos.

Una vez completado todo el proceso de limpieza, transformación y escalamiento de los datos, el último paso de mi ETL fue cargar estos datos procesados en un formato que pudiera ser fácilmente utilizado para el modelado posterior.

Para esto, utilicé la función to_csv de pandas para guardar mis datos procesados en un nuevo archivo CSV. Elegí el formato CSV porque a mi gusto es el más fácil de manipular con las diferentes herramientas y lenguajes de programación. Además, al guardar los datos procesados, puedo evitar tener que repetir todo el proceso de ETL cada vez que quiera trabajar con estos datos.

Siguientes pasos.

Es importante decir que, aunque hemos completado los pasos principales de ETL, los datos aún experimentarán una transformación adicional **muy importante** antes de ser utilizados en el modelo. Con esto me refiero a que se realizará una división entre datos de entrenamiento y prueba.

Esta división es clave para poder evaluar que tan bien funciona el modelo con datos que no ha visto durante el entrenamiento. Esto nos ayuda a entender si el modelo puede hacer buenas predicciones en datos nuevos. Los detalles de esta división y cómo se implementa se explicarán en la sección del modelo de regresión lineal.

Modelo de regresión lineal.

Introducción.

La regresión lineal es un método estadístico que nos permite entender y predecir la relación entre una variable dependiente (la que queremos predecir) y una o más variables independientes (las que usamos para hacer la predicción).

Un ejemplo de la vida real para entender mejor cuándo usar regresión lineal, podría ser cuando se usa para predecir precios de casas basándose en características como el tamaño, la ubicación, y el número de habitaciones. Otro ejemplo es el uso en negocios para prever ventas futuras a partir de datos históricos.

Existen dos tipos principales de regresión lineal: la **regresión lineal simple**, que usa una sola variable independiente, y la **regresión lineal múltiple**, que usa dos o más variables independientes para hacer predicciones. En mi proyecto yo utilicé la regresión lineal múltiple, porque estoy trabajando con varias variables independientes (como el tipo de combustible, el número de cilindros, la potencia del motor, etc.) para predecir una variable dependiente, que es el precio del vehículo.

El tipo de modelo que estoy construyendo es de tipo "Aprendizaje supervisado". Con esto me refiero a que vamos a comparar lo que el modelo predice con los valores reales que ya conocemos. Durante el proceso de entrenamiento, el modelo aprende ajustando sus parámetros para que sus predicciones se acerquen lo más posible a estos valores conocidos. Este enfoque es fundamental porque nos permite medir y mejorar la precisión del modelo antes de aplicarlo a nuevos datos.

Establecer función de hipótesis.

La hipótesis en la regresión lineal es una fórmula matemática que describe cómo las variables independientes se relacionan con la variable dependiente que queremos predecir . La fórmula general para la hipótesis en una regresión lineal múltiple es:

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

Esta fórmula parece muy rebuscada, pero realmente es algo muy sencillo. Esta es la explicación:

- $h(x)$ representa la predicción que queremos lograr con el modelo. En mi caso ejemplo, esta representa el precio del carro.

- θ_0 es el intercepto, es decir, el valor de $h(x)$ cuando todas las variables independientes son cero. En la clase nosotros lo estamos manejando como “Bias”.
- Por último, θ_1, θ_2, ..., θ_n son los coeficientes que multiplican a cada una de las variables independientes $x1, x2, ..., xn$. Estos coeficientes indican cuánto influye cada variable independiente en la predicción.

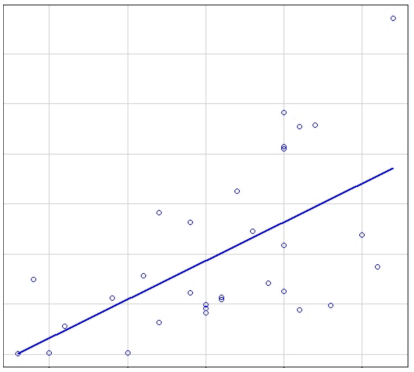
Para entender esto de una mejor manera, voy a poner un ejemplo con mi proyecto. Es importante que previo a esto, hayan leído la parte de ETL. Sobre todo para entender las columnas (atributos) y filas (features).

Si estoy usando atributos como “el tipo de combustible”,” el número de cilindros”, y “la potencia del motor” para predecir el precio del vehículo, la hipótesis podría verse así:

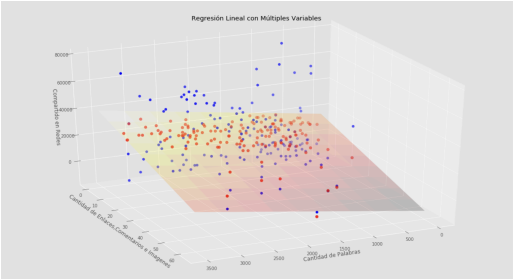
$$h(precio) = \theta_0 + \theta_1(tipo\ de\ combustible) + \theta_2(número\ de\ cilindros) + \theta_3(potencia\ del\ motor) + \ldots$$

En este caso, θ_1, θ_2, θ_3 son los parámetros que el modelo ajusta durante el entrenamiento para que la predicción de $h(x)$ se acerque lo más posible al precio real del vehículo.

Visualmente, si fuera una regresión lineal simple con solo una variable independiente, la hipótesis se representaría como una línea recta en un gráfico:



En la regresión lineal múltiple, estamos ajustando un plano o un hiperplano en un espacio de varias dimensiones, dependiendo de cuántas variables independientes estemos utilizando. Me gusta clarificar esto porque es una pregunta que he realizado múltiples veces en clase y el profesor Benjamin me ha ayudado a entender.



Ahora bien, en relación a cómo manejo esto en mi código: Definí una función de hipótesis que realiza este cálculo de una manera simple y directa. La función hypothesis que creé toma como entradas los parámetros (params), los valores de las variables independientes (sample), y el Bias (bias). Luego, calcula la predicción usando el producto punto entre los parámetros y las variables, sumándole el Bias al final. Este enfoque me pareció más claro que incluir el Bias dentro de la matriz de datos desde el principio.

```python
def hypothesis(params, sample, bias):
  return np.dot(params, sample) + bias
```

Como podemos observar, hago la misma fórmula que explique al inicio pero representado como una multiplicación de vectores en NumPy.

Función de costo o pérdida.

La función de costo es lo siguiente que implementé en mi modelo. Esta función mide qué tan bien el modelo está realizando sus predicciones en comparación con los valores reales. (Recordemos que estamos trabajando con un modelo de aprendizaje supervisado).

Existen varios tipos de funciones de costo, cada una con sus propias características y aplicaciones específicas. Algunas de las más comunes incluyen:

- Error Cuadrático Medio (Mean Squared Error - MSE)
- Error Absoluto Medio (Mean Absolute Error - MAE)
- Error Cuadrático Logarítmico (Mean Squared Logarithmic Error - MSLE)
- Entropía Cruzada (Cross-Entropy)

Para el modelo de regresión lineal, he optado por utilizar el Error Cuadrático Medio (MSE) como función de costo. Esto se debe a que es la función que más hemos visto en clase, porque es la que más he practicado y me he dado cuenta de que es particularmente útil con modelos de regresión lineal.

El MSE se calcula de la siguiente manera:

$$MSE = (1/n) * \Sigma(y_i - \hat{y}_i)^2$$

Donde:

- n es el número total de muestras
- y_i es el valor real
- $\hat{y}_i$ es el valor predicho por el modelo

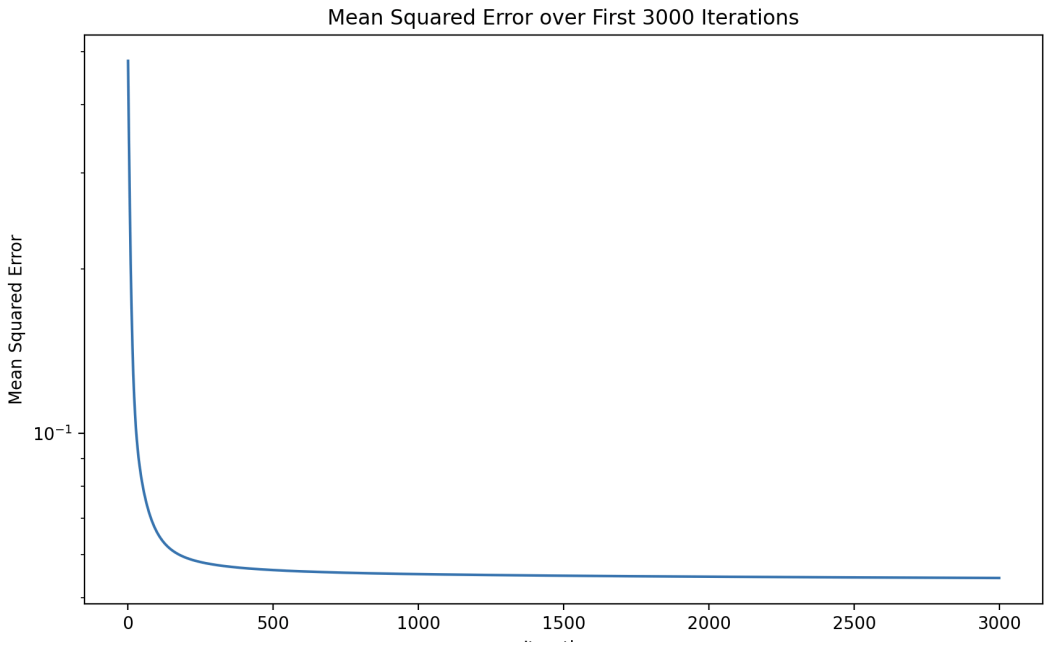
En otras palabras, el MSE calcula el promedio de las diferencias al cuadrado entre los valores que predice el modelo y los valores reales.

En mi implementación, codifiqué la función de costo MSE de la siguiente manera:

```python
def cost_function(params, bias, samples, y):
    y_prima = np.dot(samples, params) + bias
    mse = np.mean(np.square(y_prima - y)) / 2
    return mse
```

Esta función toma los parámetros del modelo (params), el sesgo (bias), las muestras de entrada (samples) y los valores reales (y). Calcula las predicciones (y_prima) utilizando los parámetros actuales y luego calcula el MSE.

Después de realizar este código, hice la gráfica para representar cómo los valores van cambiando y ajustando el error a 0.



Gradiente descendiente.

Una vez que hemos definido nuestra función de costo, el siguiente paso es ajustar los parámetros del modelo para que haga las mejores predicciones posibles. Para hacer esto, necesitamos encontrar los valores de los parámetros que minimicen la función de costo, es decir, que hagan que el error entre las predicciones del modelo y los valores reales sea lo más pequeño posible. Aquí es donde entra el Gradiente Descendente.

El Gradiente Descendente es un algoritmo que nos ayuda a encontrar esos valores óptimos de los parámetros. Básicamente, lo que hace es empezar con valores iniciales para los parámetros y luego, en cada paso, ajustarlos un poco en la dirección que reduce el error (lo hace a través de las derivadas). Este proceso se repite muchas veces hasta que el modelo llega a un punto donde los cambios en los parámetros no mejoran mucho más el modelo. De esta forma, logramos que el modelo sea lo más preciso posible en sus predicciones.

La fórmula que estoy utilizando es la siguiente:

$$\theta_j = \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} [(h_\Theta(x_i) - y_i)x_i]$$

Donde:

- Θ_j: Es el parámetro que se está ajustando en cada paso del Gradiente Descendente.
- α :Es la tasa de aprendizaje, que controla el tamaño de los pasos que damos en la dirección del mínimo. Si es muy grande, podemos saltarnos el mínimo; si es muy pequeño, el algoritmo puede ser muy lento.
- m: Es el número total de muestras en el dataset. Este valor lo utilizamos para calcular el promedio de los gradientes.
- $h_\Theta(x_i)$: Es la predicción del modelo para la i-ésima muestra, calculada usando la hipótesis.
- y_i: Es el valor real para la i-ésima muestra, que estamos tratando de predecir.
- x_i: Es el valor de la variable independiente para la i-ésima muestra.

Ahora pasemos esto a mi código:

```python
def gradient_descent(params, bias, samples, y, alpha, iterations):
  m = len(y)
```

```
    for _ in range(iterations):
        predictions = np.dot(samples.T, params) + bias
        errors = predictions - y

        # Calcular los gradientes
        gradient_params = (1/m) * np.dot(samples, errors)
        gradient_bias = (1/m) * np.sum(errors)

        # Actualizar los parámetros y el bias
        params -= alpha * gradient_params
        bias -= alpha * gradient_bias

    return params, bias
```

Este código es muy lineal y fácil de entender. Básicamente, lo que hacemos es aplicar paso a paso el algoritmo de Gradiente Descendente que mostré antes, integrándolo ahora con las funciones de MSE e Hipótesis que ya definimos:

1. Inicializamos los parámetros (incluyendo el bias) con ceros. (esto lo hago más abajo donde defino los parámetros del modelo).
2. Utilizando los parámetros actuales, calculamos las predicciones del modelo. Esto se hace con el producto punto entre las variables independientes y los parámetros, sumando el bias al final. (El bias en la primera iteración es 0 pero la magia viene que después ese bias va a ser modificado con cada iteración).
3. Luego, calculamos la diferencia entre las predicciones del modelo y los valores reales (es decir, el error).
4. Con base en el error, calculamos los gradientes de la función de costo con respecto a cada parámetro. Estos gradientes nos indican en qué dirección debemos ajustar los parámetros para reducir el error.
5. Después actualizamos los parámetros. Ajustamos los parámetros (incluyendo el bias) en la dirección opuesta al gradiente, es decir, hacia donde la función de costo disminuye. Esto nos permite acercarnos cada vez más a la solución óptima.

Ahora, de conceptos que son importantes para poder entender este código:

- α es la tasa de aprendizaje, que controla el tamaño de los pasos que damos en la dirección del gradiente.
- “iterations" es el número máximo de iteraciones que realizaremos. (En clase lo hemos visto como EPOCHS).

Los pasos 2-5 los repetimos hasta que el cambio en la función de costo sea muy pequeño o alcancemos un número máximo de iteraciones (EPOCHS, en este momento lo tengo en 10000).

Separación de datos en entrenamiento, validación y prueba.

Como mencioné anteriormente, en la última etapa de ETL, todavía me faltaba un paso más en la manipulación de datos: la separación entre datos de entrenamiento, validación y prueba. Esta división es crucial para evaluar el rendimiento del modelo en datos que no ha visto durante el entrenamiento. En mi caso, dividí el dataset en tres partes:

- Entrenamiento (60%): Usado para ajustar los parámetros del modelo.
- Validación (20%): Empleado para ajustar los hiperparámetros y evitar el overfitting.
- Prueba (20%): Usado para evaluar el rendimiento final del modelo en datos no vistos.

Esto lo realicé de la siguiente manera:

```python
def train_val_test_split(X, y, test_size=0.2, val_size=0.1, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    n_samples = len(X)
    n_test = int(n_samples * test_size)
    n_val = int(n_samples * val_size)
    n_train = n_samples - n_test - n_val

    indices = np.random.permutation(n_samples)

    X_train = X[indices[:n_train]]
    y_train = y[indices[:n_train]]

    X_val = X[indices[n_train:n_train + n_val]]
    y_val = y[indices[n_train:n_train + n_val]]

    X_test = X[indices[n_train + n_val:]]
    y_test = y[indices[n_train + n_val:]]

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

Si me gustaría decir que este paso es crucial. Contar con una adecuada separación de datos entre entrenamiento, validación y prueba mejora significativamente la capacidad de mi modelo para generalizar a datos nuevos que no ha visto antes. Al utilizar aparte un conjunto de validación, podemos ajustar el modelo durante el entrenamiento y seleccionar el mejor conjunto de parámetros. Esto me permite hacer predicciones más precisas cuando el modelo se enfrenta a datos reales, mejorando su rendimiento y reduciendo el riesgo de overfitting.

Resultados y evaluación del modelo.

Es momento de poner en práctica todos los conceptos que mencioné anteriormente. En mi caso, como lo mencioné anteriormente, voy a estar tratando de predecir el costo del carro basándose en sus características (diferentes atributos del dataset).

Estos fueron los resultados que obtuve al correr mi modelo de regresión lineal:

Función de MSE.

- **Entrenamiento:** El error final en el conjunto de entrenamiento (recordemos que esto abarca el 80% de mi dataset) fue de **0.0533**. Este valor representa que tan bien se ajusta la función a los datos de entrenamiento; entre menor sea, quiere decir que se ajusta de mejor manera.

- **Prueba:** El error final en el conjunto de prueba fue de **0.1379**. Este valor muestra el MSE del modelo al predecir sobre datos no vistos durante el entrenamiento (me refiero al 20% restante de los datos del dataset).

Al comparar los resultados de los dos, puedo inferir que mi modelo no está prediciendo con la misma precisión en los datos de prueba que en los de entrenamiento, lo que sugiere un posible overfitting. Pero no me gustaría decir nada hasta no ver el Bias y en ese momento dar una buena conclusión.

Coeficiente de Determinación (R^2).

En mi caso, la R^2 es una métrica que utilizo para evaluar qué tan bien mi modelo de regresión lineal predice el valor de la variable dependiente en función (precio) de las variables independientes (otras features).

Los valores que me dieron para mi R^2 fueron los siguientes:

- **Entrenamiento:** En el conjunto de entrenamiento fue de **0.8890**. Este valor es un valor que yo considero como bueno ya que nos indica que aproximadamente el 88.90% de la variabilidad en el costo del carro está explicada por el modelo en el conjunto de entrenamiento.
- **Prueba:** Por otro lado, en el conjunto de prueba el R^2 fue de **0.7587**. Esto significa que el modelo explica alrededor del 75.87% de la variación en el costo del carro cuando se utiliza en datos nuevos, no vistos durante el entrenamiento. Aunque sigue siendo un valor relativamente alto, es menor que en el conjunto de entrenamiento, lo que sugiere que el modelo podría estar perdiendo algo de precisión al generalizar a nuevos datos.

Evaluación del Bias.

Para el Bias, obtuve el siguiente resultado:

- Con un valor final de 0.0170, considero que mi valor de Bias es bajo. Recordemos que un Bias bajo indica que el modelo es capaz de analizar o captar bien los patrones en los datos de entrenamiento (algo que hemos visto en la función de MSE y R^2).

Con esta información, me gustaría todavía aplicar un proceso más a mi modelo, el cúal, en teoría, me va a ayudar a mejorarlo.

Regularización de mi modelo.

Primero, me gustaría decir que la regularización es una técnica que se utiliza para mejorar la capacidad de generalización de un modelo, es decir, su rendimiento en datos no vistos durante el entrenamiento. En mi caso, después de observar que mi modelo presentaba signos de overfitting, decidí implementar Regularización L2 (también conocida como Ridge).

Pero… ¿Para qué es la Regularización?

La regularización L2 añade un término adicional a la función de costo (en mi caso en el código cree una función nueva de MSE) que penaliza los valores grandes de los coeficientes del modelo. Al hacerlo, evito que el modelo se ajuste demasiado a los datos de entrenamiento, capturando ruidos o patrones específicos que no se generalizan bien a nuevos datos.

En mi código lo hago de la siguiente manera (cree funciones diferentes para no afectar a las que ya tenía):

```python
def cost_function_with_regularization(params, bias, samples, y, lambda_):
    y_prima = np.dot(samples, params) + bias
    mse = np.mean(np.square(y_prima - y)) / 2
    reg_term = (lambda_ / 2) * np.sum(np.square(params))
    return mse + reg_term

def gradient_descent_with_regularization(params, bias, samples, y, alpha,
iterations, lambda_):
    m = len(y)
    mse_history = []

    for _ in range(iterations):
        predictions = np.dot(samples.T, params) + bias
```

```
        errors = predictions - y
        mse = np.mean(np.square(errors)) / 2
        reg_term = (lambda_ / 2) * np.sum(np.square(params))
        mse_history.append(mse + reg_term)

        gradient_params = (1/m) * np.dot(samples, errors) + (lambda_ / m) *
params
        gradient_bias = (1/m) * np.sum(errors)

        params -= alpha * gradient_params
        bias -= alpha * gradient_bias

    return params, bias, mse_history
```

Estas son funciones que ya había utilizado previamente solo agregando los parámetros que platiqué anteriormente para hacer la función de regularización.

Ahora sí, habiendo terminado todos los procesos, haré el diagnóstico de mi modelo.

Diagnóstico del modelo.

Después de aplicar la regularización L2, evalúe nuevamente mi modelo utilizando los conjuntos de datos de entrenamiento, validación y prueba. A continuación, presento los resultados obtenidos:

- Bias final: 0.0076
- Error final en el conjunto de entrenamiento: 0.0480
- Error en validación: 0.1381
- Error final en el conjunto de prueba: 0.1731
- R^2 en conjunto de entrenamiento: 0.8891
- R^2 en conjunto de validación: 0.8276
- R^2 en conjunto de prueba: 0.6993

Mi modelo sigue mostrando signos de overfitting, a pesar de haber aplicado la regularización L2. Esto lo puedo ver en la diferencia entre el R^2 en el conjunto de entrenamiento (0.8891) y en el conjunto de prueba (0.6993), lo que me indica que el modelo sigue ajustándose demasiado bien a los datos de entrenamiento en comparación con su rendimiento en datos no vistos.

Análisis de los Resultados

La regularización L2 ha reducido el Bias del modelo a 0.0076, lo cual es un indicador positivo.

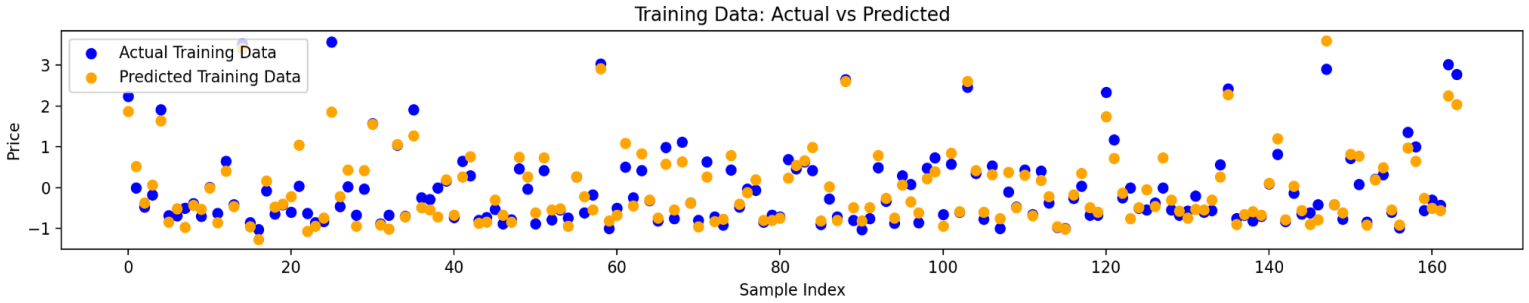
Sin embargo, la diferencia entre los resultados de entrenamiento y prueba me dice que aún persisten los problemas de overfitting. Esto podría indicar que el parámetro de regularización “lambda” no es lo suficientemente grande o que el modelo aún necesita ajustes adicionales, como la selección de características más relevantes o el uso de un modelo más complejo.

Visualización de Resultados

Para entender mejor cómo el modelo predice el costo de los automóviles, comparé los valores reales con los valores predichos en los conjuntos de entrenamiento, validación, y prueba.

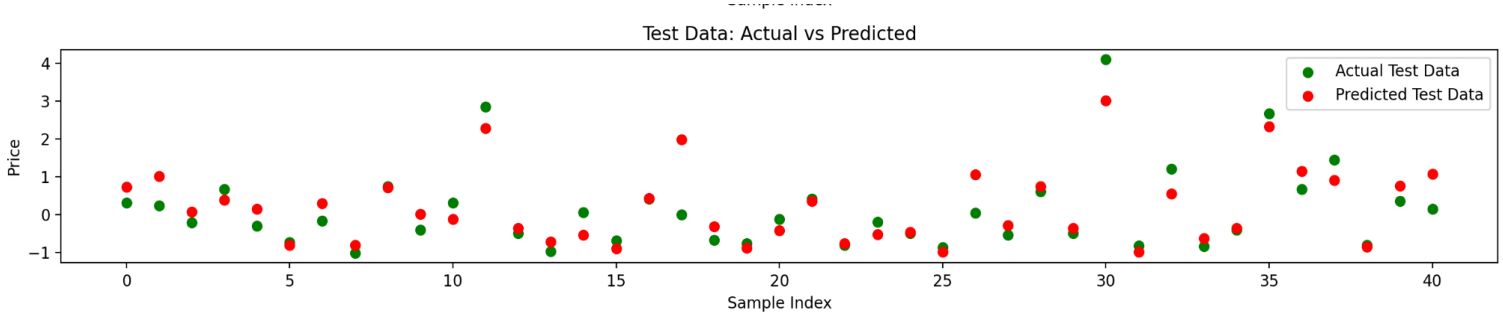
Esta comparación visual me permite identificar cómo se comporta el modelo en datos que ya conoce (entrenamiento), en datos que utiliza para ajustar hiperparámetros (validación), y en datos nuevos (prueba), lo cual es fundamental para entender su capacidad de generalización. También es importante que mencione que estos son los resultados son ya habiendo pasado mis datos por el proceso de regularización L2. A continuación, se presentan los gráficos que ilustran estas comparaciones:

Datos de Entrenamiento: Reales vs. Predicción



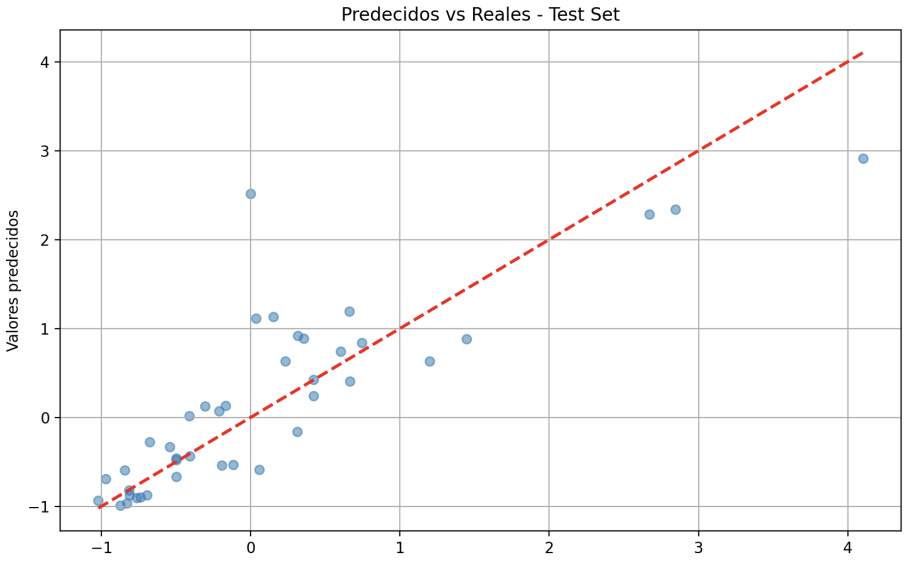
Este gráfico muestra cómo las predicciones del modelo se alinean con los valores reales en el conjunto de entrenamiento.

Datos de Prueba: Reales vs. Predicción



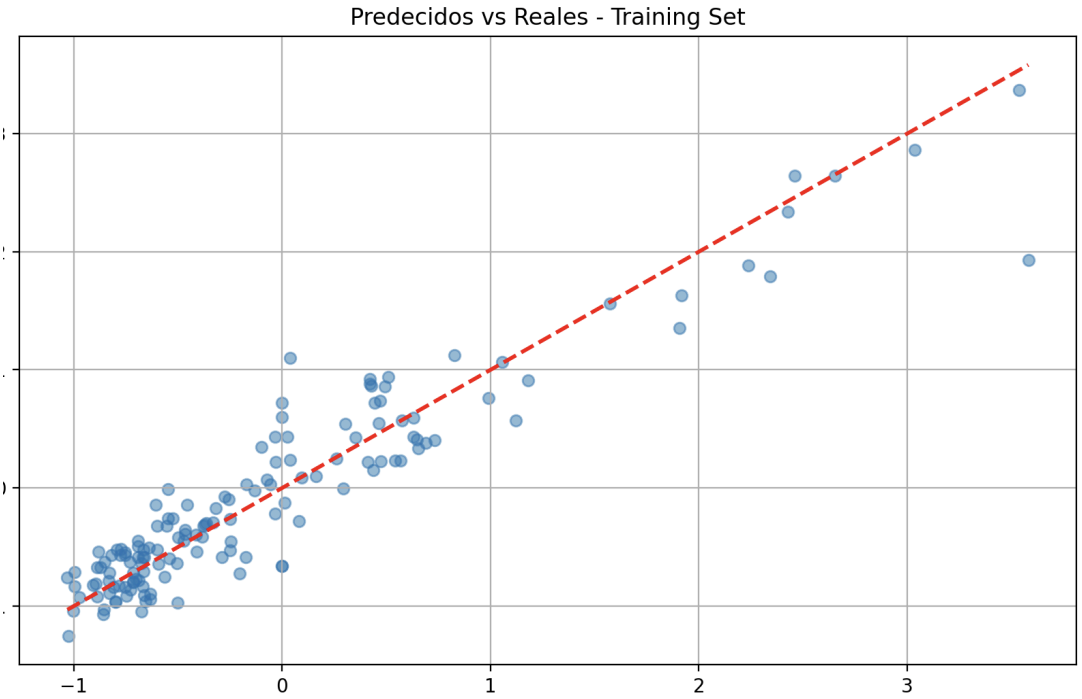
Este gráfico muestra la capacidad del modelo para predecir con datos no vistos.

Datos de Prueba: Reales vs. Predicción



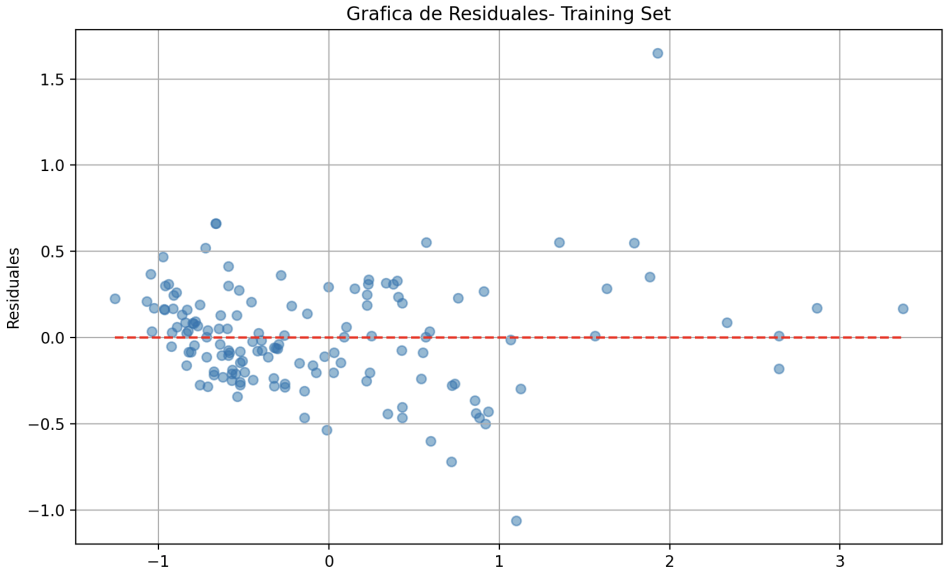
Este gráfico lo generé para ver cómo es que se ve de forma lineal los datos que son predecidos de prueba entre los reales y los de predicción.

Datos de Entrenamiento: Reales vs. Predicción

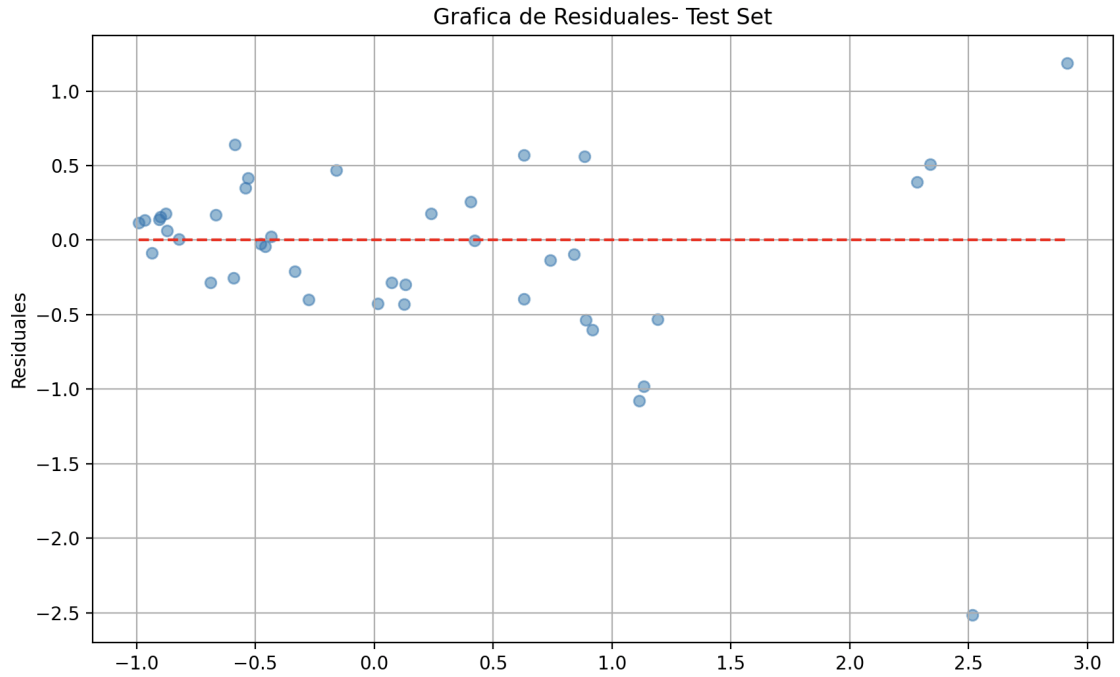


Este gráfico lo generé para ver cómo es que se ve de forma lineal los datos que son predecidos de entrenamiento entre los reales y los de predicción.

Datos de Entrenamiento: Reales vs. Predicción

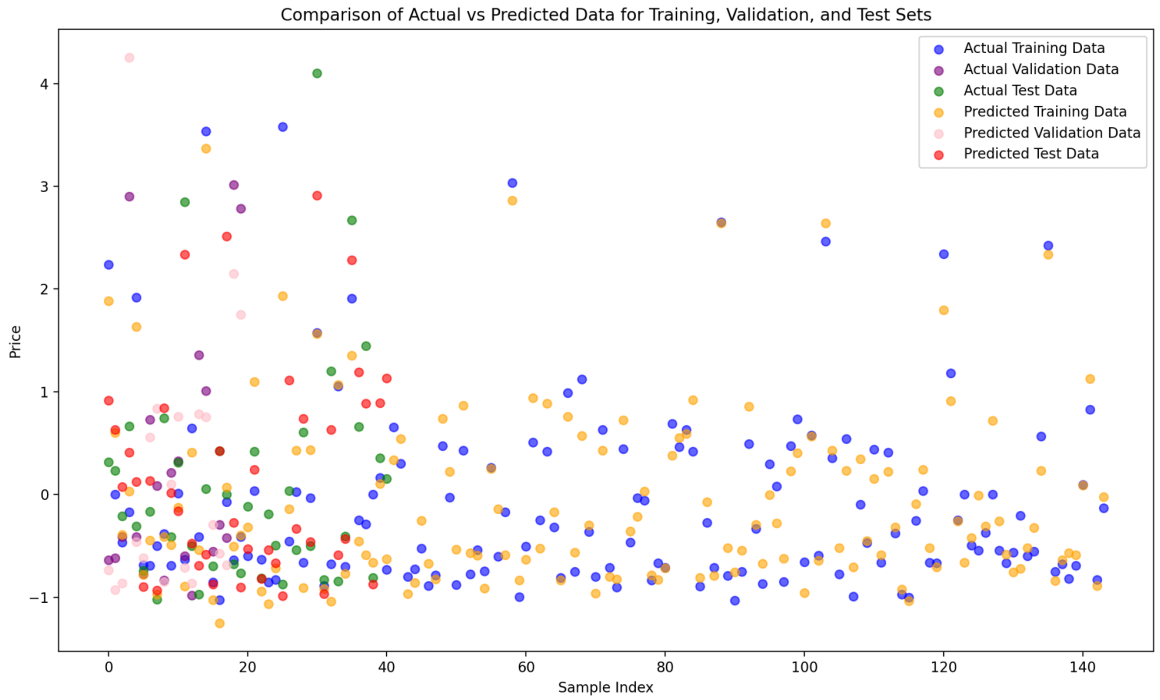


Este gráfico me ayuda a ver las diferencias entre son las diferencias entre los valores observados (reales) y los valores predichos por el modelo para los datos de entrenamiento.



Este gráfico me ayuda a ver las diferencias entre son las diferencias entre los valores observados (reales) y los valores predichos por el modelo para los datos de prueba.

Datos Reales vs. Predicción vs. Validation



Por último, este gráfico muestra tanto el comportamiento de los datos de predicción de train, test y validación contra los valores reales.

Conclusión.

A lo largo de todo este ensayo, me gustaría pensar que demostré todos los conocimientos que he obtenido en este corto periodo de tiempo respecto a los modelos predictivos y el proceso de ETL. Este ensayo es algo de lo que me enorgullezco porque pude plasmar paso por paso como fue que fui desarrollando mi modelo desde 0, entendiendo las bases y el porqué de las cosas.

Desde la selección y preparación de los datos, hasta la implementación y evaluación del modelo de regresión lineal, cada etapa representó un desafío y una oportunidad para aplicar lo aprendido en clase. Siendo esta la segunda entrega me hace darme cuenta de lo mucho que estoy disfrutando la concentración porque realmente estoy aprendiendo y comprendiendo lo que estoy haciendo.

Uno de los aspectos más valiosos de este proyecto fue la implementación de técnicas avanzadas como la regularización L2 (Ridge). Estas técnicas me permitieron no solo mejorar la capacidad predictiva del modelo, sino también mitigar problemas comunes como el overfitting. Lamentablemente, aunque logré implementar todo, no pude reducir el overfitting.

Las gráficas de residuales y la comparación entre valores predichos y valores reales también fueron herramientas fundamentales para entender la calidad del modelo. Estas visualizaciones me permitieron detectar cualquier patrón o anomalía en el comportamiento del modelo, confirmando así que las mejoras implementadas eran efectivas.

En la siguiente parte de mi reporte, ahora hablaré de mi implementación de un modelo de random forest con la librería de scikit-learn.

Random Forest Model

En esta sección, voy a explicar cómo implementé un modelo de Random Forest utilizando la biblioteca Scikit-learn. Este modelo es un algoritmo de aprendizaje supervisado que se emplea tanto para tareas de clasificación como de regresión. En mi caso, lo utilicé para predecir el precio de un automóvil basándome en diversas características extraídas de otras columnas del dataset, similar a lo que hice en mi modelo de regresión lineal.

Carga y Preparación de Datos

```python
import numpy as np
import pandas as pd

df_scaled = pd.read_csv('automobile_cleaned_scaled.csv')
X = df_scaled.drop('Price', axis=1).values
y = df_scaled['Price'].values
```

Lo primero que hice, al igual que en mi modelo de regresión lineal, fué cargar y preparar los datos utilizando la biblioteca de pandas. Ahora, eliminé la columna 'Price' del conjunto de características (X) y la separé como la variable objetivo (y).

División del Dataset en Conjuntos de Entrenamiento, Validación y Prueba

```python
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,
random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp,
test_size=0.5, random_state=42)
```

En esta sección, dividí el dataset en tres partes: un conjunto de entrenamiento (70% de los datos), un conjunto de validación (15%), y un conjunto de prueba (15%). Hago la separación de los datos en porcentajes diferentes a mi modelo de regresión lineal solamente para ver si esto ayuda con el overfitting que presentó mi modelo.

Creación e Implementación del Modelo de Random Forest

```python
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

Esta es la parte más importante de mi código. Lo que hago es crear un modelo de Random Forest utilizando la clase “RandomForestRegressor” de Scikit-learn.

El primer paso en la construcción del modelo fue decidir el número de árboles a utilizar en el bosque. Opté por 100 árboles (n_estimators=100), lo que significa que el modelo entrenará 100 árboles de decisión independientes en paralelo. Cada árbol de decisión actúa como un "experto" que intenta predecir el precio del automóvil basándose en diferentes subconjuntos de las características disponibles. Al final, el modelo toma un promedio de las predicciones de todos los árboles, lo que tiende a mejorar la precisión y reducir el riesgo de overfitting.

Una vez definido el número de árboles, entrené el modelo utilizando el conjunto de entrenamiento. Durante este proceso, cada árbol de decisión aprende las relaciones entre las características del automóvil (como el tipo de combustible, el número de cilindros, la potencia del motor, etc.) y su precio. Gracias al enfoque de Random Forest, el modelo puede captar interacciones complejas entre las características, lo que mejora su capacidad para realizar predicciones precisas incluso en casos donde algunas características son menos informativas o están correlacionadas.

Evaluación del Modelo con los Conjuntos de Validación y Prueba

```python
y_val_pred = model.predict(X_val)
y_test_pred = model.predict(X_test)

val_mse = mean_squared_error(y_val, y_val_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

val_r2 = r2_score(y_val, y_val_pred)
test_r2 = r2_score(y_test, y_test_pred)
```

Después de completar el entrenamiento del modelo, procedí a evaluar su rendimiento utilizando los conjuntos de validación y prueba. Para ello, calculé dos métricas clave: el Error Cuadrático Medio (MSE) y el coeficiente de determinación (R^2). El MSE me permitió medir la magnitud promedio de los errores en las predicciones del modelo, dándome una idea clara de cuán precisas eran esas predicciones. Por otro lado, el R^2 me mostró qué porcentaje de la

variabilidad en los precios de los automóviles podía ser explicado por el modelo en estos conjuntos de datos. Estas evaluaciones son fundamentales para entender cómo se desempeña el modelo con datos no vistos durante el entrenamiento y para detectar posibles problemas como el overfitting.

Validación Cruzada

```python
cross_val_mse = -cross_val_score(model, X_train, y_train, cv=5,
scoring='neg_mean_squared_error').mean()
```

Para asegurarme de que el modelo tuviera un rendimiento consistente y generalizable en diferentes subconjuntos de los datos, implementé validación cruzada.
Específicamente, utilicé una validación cruzada de 5 pliegues (K-Fold), en la cual dividí el conjunto de entrenamiento en 5 partes o pliegues. El modelo se entrenó y validó en diferentes combinaciones de estas partes, lo que me permitió obtener una estimación más precisa y robusta del error del modelo.

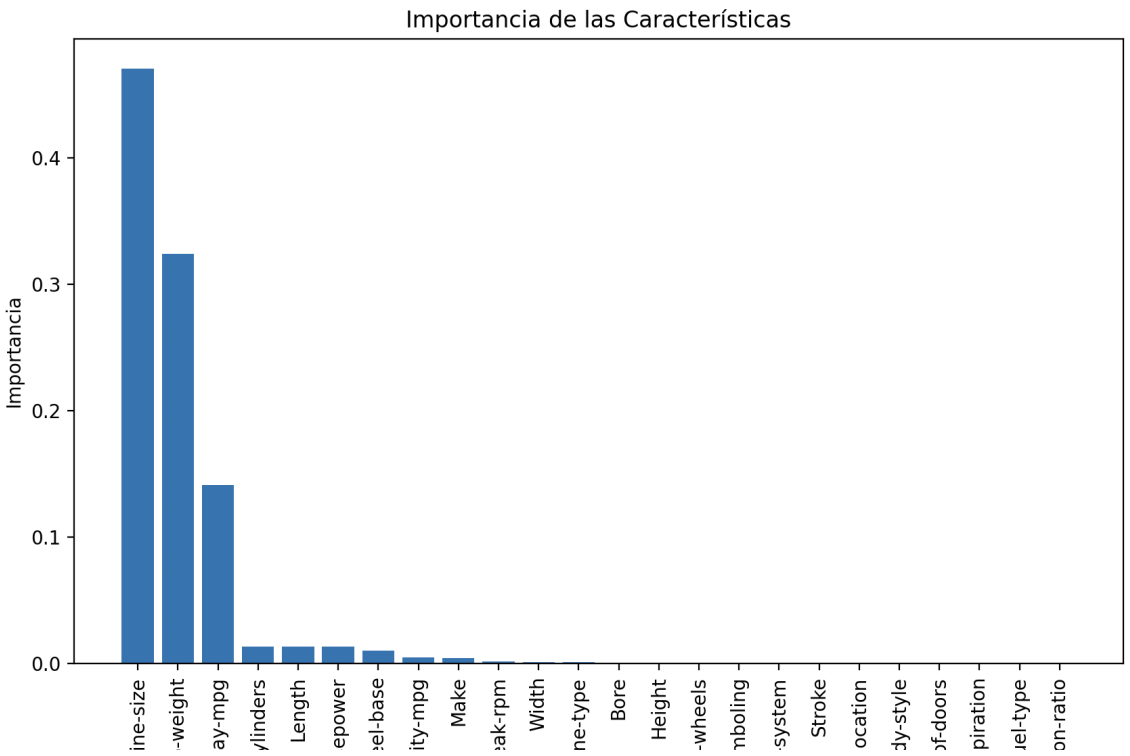
Al hacer esto, lo que logro es que en cada iteración, el modelo se entrene con una parte de los datos que no ha visto antes, lo que simula cómo se comportará con datos completamente nuevos.

Características más relevantes en mi modelo

```python
importances = model.feature_importances_
indices = np.argsort(importances)[::-1]

plt.figure(figsize=(10, 6))
plt.title('Importancia de las Características')
plt.bar(range(X.shape[1]), importances[indices], align='center')
plt.xticks(range(X.shape[1]), df_scaled.columns[indices], rotation=90)
plt.xlabel('Características')
plt.ylabel('Importancia')
plt.show()
```

Esto es algo que hice para ver que tal estaba mi modelo y por curiosidad. Una vez que el modelo estuvo entrenado, analicé la importancia de las diferentes características en la predicción del precio. Lo que hice después fue dropear las columnas que yo consideré no afectaba en mi modelo.



Comparación de resultados.

Lo que me doy cuenta al correr mi modelo es que presenta overfitting. Lo que voy a hacer es comparar los resultados de R^2 con diferentes hiper-parámetros.

Number of trees	Max tree Depth	Max Leaf Nodes	Min Samples Leaf	R^2 Train	R^2 Validation	R^2 Test
100	10	30	1	0.977262	0.914167	0.884546
100	20	50	2	0.962272	0.899816	0.891554
100	30	20	1	0.971847	0.907582	0.880196
10	10	10	1	0.943447	0.887746	0.889298
10	10	10	1	0.943447	0.887746	0.889298
10	20	50	2	0.954077	0.888037	0.908100
10	30	20	2	0.950949	0.887263	0.909361
50	10	10	5	0.910854	0.860161	0.891054
50	20	20	5	0.913090	0.865046	0.892099

50	5	5	5	0.884260	0.852790	0.879376

Como se puede observar, lo que fui haciendo para reducir el overfitting que mi modelo estaba presentando fué ajustar todos los hiper parámetros.

Al final, los mejores hiper parámetros fueron los siguientes:

- Number of trees: 50
- Max tree depth: 5
- Max Leaf Nodes: 5
- Min Samples Leaf: 5
- R^2train: 0.884
- R^2 validation: 0.852
- R^2 test: 0.879

Con estos resultados puedo decir con seguridad que teniendo una diferencia de 0.005 que mi resultado no presenta overfitting. Mi modelo se está ajustando de forma correcta a datos vistos y no vistos.

A continuación daré una mayor explicación de mi modelo

Diagnóstico del modelo.

Diagnóstico del Modelo

Error Cuadrático Medio (MSE)

- **Entrenamiento:** 0.1100
- **Validación:** 0.1126
- **Prueba:** 0.1745

El MSE que me da en el conjunto de entrenamiento es relativamente bajo (0.1100). Esto me indica que el modelo se ajusta bien a los datos de entrenamiento. En los conjuntos de validación y prueba, el MSE es ligeramente mayor (0.1126 y 0.1745, respectivamente), lo que sugiere que mi modelo mantiene un buen rendimiento al enfrentarse a datos no vistos. Aunque hay un aumento en el MSE en los conjuntos de validación y prueba, el incremento no es excesivo, lo que indica que el modelo tiene una capacidad de generalización razonable.

Coeficiente de Determinación (R^2)

- **Entrenamiento:** 0.8840
- **Validación:** 0.8520
- **Prueba:** 0.8790

Analizando el R^2 en el conjunto de entrenamiento, puedo decir que es alto (0.8840), lo que significa que el modelo explica el 88.40% de la variabilidad en los precios de los automóviles dentro del conjunto de entrenamiento.

En los conjuntos de validación (0.8520) y prueba (0.8790), el R^2 sigue siendo elevado, lo que demuestra que el modelo también captura gran parte de la variabilidad en los datos no vistos, aunque con una ligera disminución en la precisión. Esta pequeña caída en el R^2 entre los conjuntos de entrenamiento, validación y prueba es normal y esperada.

Sobreajuste (Overfitting)

Con los parámetros actuales, mi modelo muestra un ajuste **¡¡equilibrado!!**. Aunque el MSE y el R^2 en los conjuntos de validación y prueba son ligeramente inferiores a los del conjunto de entrenamiento, la diferencia no es significativa. Esto me dice que mi modelo no está presentando overfitting con los datos de entrenamiento, sino que mantiene una buena capacidad de generalización. El hecho de que el R^2 en el conjunto de prueba sea muy cercano al del conjunto de entrenamiento indica que mi modelo no está capturando excesivamente el ruido en los datos de entrenamiento, sino que está aprendiendo patrones generales que se aplican bien a nuevos datos.

Validación Cruzada

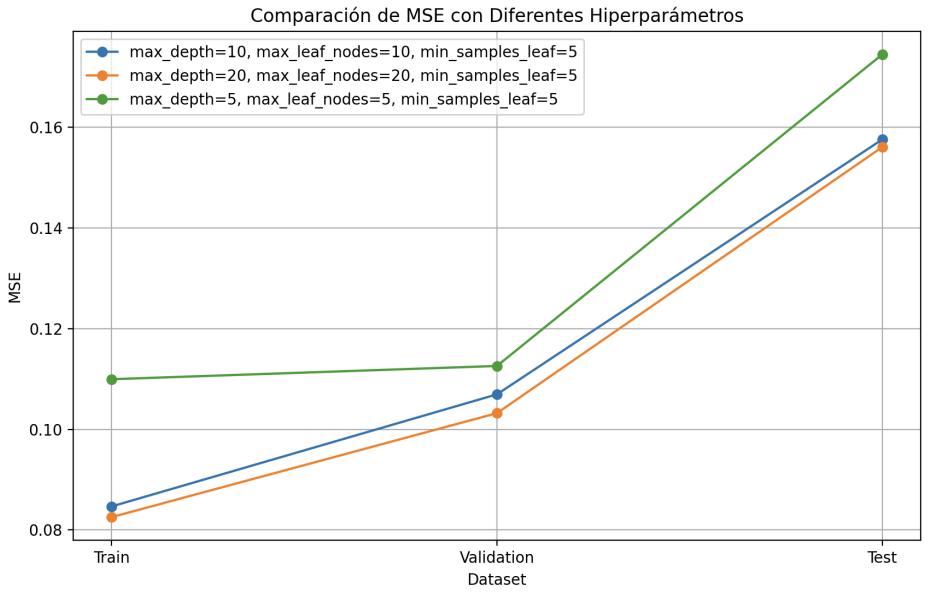
La validación cruzada proporciona una estimación más robusta del rendimiento de mi modelo. El MSE que obtuve a través de la validación cruzada es de 0.3111, lo cual es un poco mayor que el MSE en el conjunto de prueba (0.1745). Esto sugiere que, aunque el modelo tiene un rendimiento sólido, podría estar mostrando una ligera variabilidad cuando se enfrenta a diferentes subconjuntos de datos. Sin embargo, la diferencia no es tan significativa como para indicar un problema grave de generalización, pero sí sugiere que hay espacio para mejorar la consistencia del modelo a través de ajustes adicionales en los hiperparámetros.

Conclusión del Diagnóstico

Mi modelo de Random Forest muestra un rendimiento sólido y equilibrado en los conjuntos de entrenamiento, validación y prueba. Con valores de R^2 altos y MSE razonablemente bajos, mi modelo demuestra una capacidad de generalización eficaz. Aunque existen diferencias leves entre los errores en los diferentes conjuntos, estas no son significativas, lo que sugiere que el modelo no tiene overfitting y es fiable para predecir precios de automóviles.

Se podrían explorar ajustes adicionales en los hiperparámetros (como el número de árboles, la profundidad máxima o el número de nodos hoja) para afinar aún más el modelo, pero los resultados actuales indican que el modelo está bien optimizado para esta tarea.

Visualización de Resultados



En esta gráfica lo que quería ver es cómo se comporta el MSE trabajando con mis diferentes hiper parámetros. Lo que me gustó ver de esta gráfica es como se comportan los datos con los hiper parámetros que al final utilicé para mi modelo. (Línea verde)

Conclusión general.

A lo largo de este proyecto, implementé y evalué dos modelos de aprendizaje supervisado: un modelo de regresión lineal y un modelo de Random Forest. Ambos con el objetivo de predecir el precio de un automóvil basado en sus características. Cada uno de estos modelos tiene sus propias fortalezas y debilidades, y los resultados obtenidos reflejan esas diferencias.

Modelo de Regresión Lineal:

Mi modelo de regresión lineal mostró un buen rendimiento en términos de simplicidad y capacidad para capturar relaciones lineales entre las características y el precio del automóvil. Sin embargo, a pesar de obtener un buen R^2 en el conjunto de entrenamiento (0.8890), el modelo sufrió de overfitting, como lo indicaron las diferencias entre los resultados en los conjuntos de entrenamiento y prueba. Aunque intenté mejorar el modelo aplicando regularización L2, el overfitting persistió en cierta medida, lo que limitó su capacidad para generalizar a nuevos datos.

Modelo de Random Forest:

Por otro lado, mi modelo de Random Forest presentó un rendimiento significativamente mejor en términos de precisión y capacidad de generalización. Con un R^2 de 0.8520 en el conjunto de entrenamiento y 0.8790 en el conjunto de prueba. Lo que logré con este modelo fué reducir el overfitting y ajustar los R^2 para que mi modelo fuera lo suficientemente capaz de predecir el precio.

Comparación y Conclusión:

Comparando mis dos modelos, es evidente que el modelo de Random Forest superó al modelo de regresión lineal en casi todos los aspectos, especialmente en su capacidad para manejar la complejidad de los datos y evitar el overfitting. Mientras que la regresión lineal es una excelente opción para relaciones lineales simples y cuando se busca interpretar fácilmente los coeficientes de las variables, el Random Forest es mucho más adecuado para capturar relaciones no lineales y manejar grandes conjuntos de datos con múltiples características.

En conclusión, aunque ambos modelos ofrecen valiosos aprendizajes, el modelo de Random Forest es claramente superior en este contexto, proporcionando predicciones más precisas y confiables.

Referencias.

- *Find Open Datasets and Machine Learning Projects*. (n.d.). Kaggle. Retrieved August 15, 2024, from https://www.kaggle.com/datasets
- *DataFrame — pandas 2.2.2 documentation*. (n.d.). Pandas. Retrieved August 23, 2024, from https://pandas.pydata.org/docs/reference/frame.html
- *python - Change column type in pandas*. (2013, April 8). Stack Overflow. Retrieved August 19, 2024, from https://stackoverflow.com/questions/15891038/change-column-type-in-pandas
- *StandardScaler — scikit-learn 1.5.1 documentation*. (n.d.). Scikit-learn. Retrieved August 23, 2024, from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
- Bagnato, J. I. (2018, May 13). *Ejemplo Regresión Lineal Python*. Aprende Machine Learning. Retrieved August 23, 2024, from https://www.aprendemachinelearning.com/regresion-lineal-en-espanol-con-python/
- *python - Mean Squared Error in Numpy?* (2013, May 27). Stack Overflow. Retrieved August 19, 2024, from https://stackoverflow.com/questions/16774849/mean-squared-error-in-numpy
- Donges, N. (n.d.). What Is Gradient Descent? Built In. Retrieved August 15, 2024, from https://builtin.com/data-science/gradient-descent
- matplotlib.pyplot.axis — Matplotlib 3.9.2 documentation. (n.d.). Matplotlib. Retrieved August 23, 2024, from https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.axis.html
- Starmer, J. (2023, August 31). *Random Forest Algorithm Explained with Python and scikit-learn*. YouTube. Retrieved September 1, 2024, from https://www.youtube.com/watch?v=_QuGM_FW9eo
- RandomForestClassifier — scikit-learn 1.5.1 documentation. (n.d.). Scikit-learn. Retrieved September 1, 2024, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
- Random Forest Classification with Scikit-Learn. (n.d.). DataCamp. Retrieved September 1, 2024, from https://www.datacamp.com/tutorial/random-forests-classifier-python