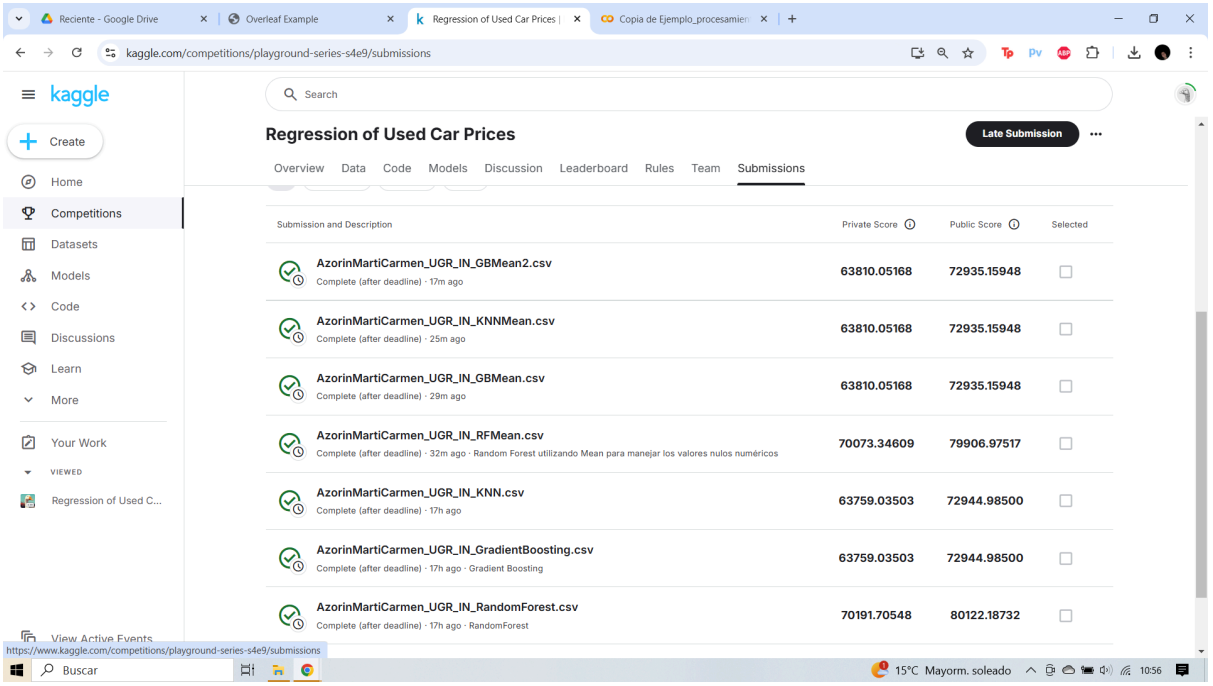
UNIVERSIDAD
DE GRANADA

PRÁCTICA 3
Competición en Kaggle
Inteligencia de Negocio
DGIIM - 2024/2025
Carmen Azorín Martí
G1
carmenazorin@correo.ugr.es



Índice

Introducción

En este proyecto, el objetivo es predecir el precio de coches usados a partir de sus características (como marca, modelo, año del modelo, kilometraje, tipo de combustible, etc.). El conjunto de datos contiene 188533 instancias y 13 atributos, principalmente categóricos, y la métrica de evaluación es el RMSE (Root Mean Squared Error).

Preprocesado de datos

El preprocesado es un paso crucial para garantizar que los datos sean adecuados para los modelos de regresión. Ha habido un preprocesado general para todos los algoritmos aunque, para ciertos algoritmos, se han añadido más pasos.

En primer lugar hemos eliminado la columna de 'id', ya que es irrelevante para la predicción del precio de los coches, tanto en en el conjunto de datos de entrenamiento como en el de test.

```
train.drop('id', axis=1, inplace=True)

test = test.drop('id', axis=1)
```

Para obtener un vistazo de los valores de cada atributo, hemos imprimido los valores de los cinco primeros registros de coches y vemos que en la variable 'engine' se incluyen una serie de características que podrían ser muy relevantes en la regresión:

172.0HP 1.6L 4 Cylinder Engine Gasoline Fuel
252.0HP 3.9L 8 Cylinder Engine Gasoline Fuel
320.0HP 5.3L 8 Cylinder Engine Flex Fuel Capab...
420.0HP 5.0L 8 Cylinder Engine Gasoline Fuel
208.0HP 2.0L 4 Cylinder Engine Gasoline Fuel

Observamos que en la descripción textual del motor aparecen características útiles como:

- Potencia (Horsepower) medida en caballos de fuerza (HP)
- Cilindrada (Displacement) que se mide en litros (L)
- Tipo de motor (Engine Type) como "V4", "V6", "Flat 6"...
- Número de cilindros (Cylinder count), por ejemplo "4 Cylinder"
- Tipo de combustible (Fuel Type) como "Gasoline", "Diesel"...

Vamos a separar estas características en nuevas columnas descriptivas de cada coche. Para ello hemos creado esta función:

```
import re
```

```
def extract_engine_feature(df):

    # extract horsepower

    df['Horsepower'] = df['engine'].apply(lambda x:
float(re.search(r'(\d+(\.\d+)?)HP', x).group(1)) if
re.search(r'(\d+(\.\d+)?)HP', x) else None)



    # extract displacement

    df['Displacement'] = df['engine'].apply(lambda x:
float(re.search(r'(\d+\.\d+)L|(\d+\.\d+) Liter', x).group(1) or
re.search(r'(\d+\.\d+)L|(\d+\.\d+) Liter', x).group(2)) if
re.search(r'(\d+\.\d+)L|(\d+\.\d+) Liter', x) else None)



    # extract engine type

    df['Engine Type'] = df['engine'].apply(lambda x:
re.search(r'(V\d+|I\d+|Flat \d+|Straight \d+)', x).group(1) if
re.search(r'(V\d+|I\d+|Flat \d+|Straight \d+)', x) else None)



    # extract cylinder count

    df['Cylinder Count'] = df['engine'].apply(lambda x:
int(re.search(r'(\d+) Cylinder', x).group(1)) if re.search(r'(\d+)
Cylinder', x) else None)



    # extract fuel type

    fuel_types = ['Gasoline', 'Diesel', 'Electric', 'Hybrid', 'Flex
Fuel']

    df['Fuel Type'] = df['engine'].apply(lambda x: next((fuel for fuel
in fuel_types if fuel in x), None))



    return df
```

Mediante expresiones regulares, obtenemos las características del motor guardadas como atributos diferentes. Si imprimimos las columnas en este punto, obtenemos:

['brand', 'model', 'model_year', 'milage', 'fuel_type', 'engine', 'transmission', 'ext_col', 'int_col', 'accident', 'clean_title', 'price', 'Horsepower', 'Displacement', 'Engine Type', 'Cylinder Count', 'Fuel Type']

Vemos que el tipo de combustible está repetido dos veces y que el atributo 'engine', que describe el motor del coche, ya no es necesario. Por tanto, eliminamos dichas columnas.

```
train = train.drop('Fuel Type', axis=1)
test = test.drop('Fuel Type', axis=1)
train = train.drop('engine', axis=1)
test = test.drop('engine', axis=1)
```

Para procesar los valores perdidos (missing values), primero debemos imprimir cuántos de ellos hay, por ejemplo en train:

```
train.isnull().sum()
```

Obtenemos una tabla que indica que la mayoría de atributos no tienen valores nulos, pero hay algunos que sí, sobretodo aquellos extraídos de las características del motor:

- fuel_type: 5083
- accident: 2452
- clean_title: 21419
- Horsepower: 33259
- Displacement: 6770
- Engine Type: 99563
- Cylinder Count: 37855

En test hay valores perdidos de las mismas variables. Lo que haremos será reemplazar los valores categóricos por el más frecuente. Para ello, podemos usar la clase SimpleImputer de la siguiente forma:

```
imputer_cat = SimpleImputer(strategy="most_frequent")
imputer_cat.fit(input_all[col_cat])
train[col_cat] = imputer_cat.transform(train[col_cat])
test[col_cat] = imputer_cat.transform(test[col_cat])
```

La tabla 'input_all' contiene todos los valores tanto de train como de test, que son todos los valores posibles de todos los atributos. Además, *col_cat* es una lista de todas las columnas categóricas: ['brand', 'model', 'fuel_type', 'transmission', 'ext_col', 'int_col', 'accident', 'clean_title', 'Engine Type'].

Para sustituir los valores nulos de las variables numéricas usamos la media de cada columna. Cabe destacar que el precio no tiene ningún valor nulo en entrenamiento, pero si lo tuviese, habría que eliminar el registro de ese coche.

```
imputer_num = SimpleImputer(strategy="mean")
imputer_num.fit(input_all[col_num])
train[col_num] = imputer_num.transform(train[col_num])
test[col_num] = imputer_num.transform(test[col_num])
```

Ahora ya no existen valores nulos en los atributos. Podemos proceder al etiquetado de las variables categóricas. Existen dos variables categóricas cuyos valores podrían interpretarse como numéricos: la variable *accident* y *clean_title*. Para convertirlos a valores numéricos hemos creado la siguiente función:

```
def mapping_columns(df):
    # replace values in the 'accident' column
    df['accident'] = df['accident'].replace({
        'At least 1 accident or damage reported': 1,
        'None reported': 0
    })

    # replace values in the 'clean_title' column
    df['clean_title'] = df['clean_title'].replace({
        'Yes': 1,
        'No': 0
    })

    return df
```

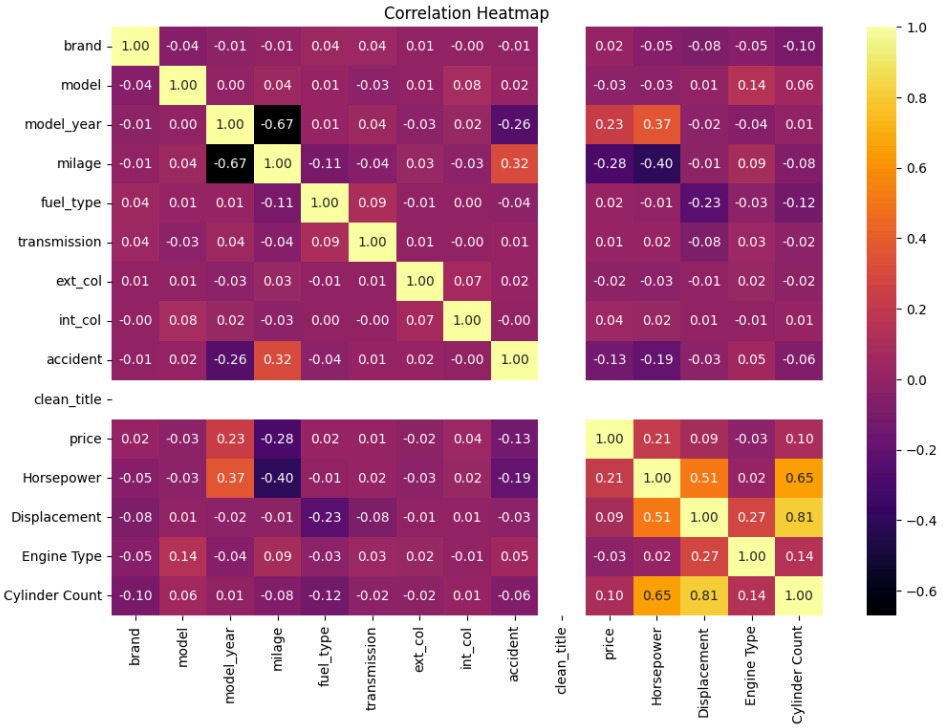
Si no ha tenido accidentes, entonces se asocia con el cero; y si ha tenido algún accidente, con el 1. La variable *clean_title* indica si el título del anuncio está limpio o no. Un título se considera no limpio cuando tiene caracteres especiales, letras en mayúsculas, palabras irrelevantes, etc.

Una vez aplicada la función *mapping_columns* a nuestros conjuntos de entrenamiento y test, ya podemos hacer el etiquetado del resto de variables categóricas usando la clase *LabelEncoder*.

```
from sklearn.preprocessing import LabelEncoder
labelers = {}
test_l = test.copy()
train_l = train.copy()

for col in col_cat:
    labelers[col] = LabelEncoder().fit(input_all[col])
    test_l[col] = labelers[col].transform(test[col])
    train_l[col] = labelers[col].transform(train[col])
```

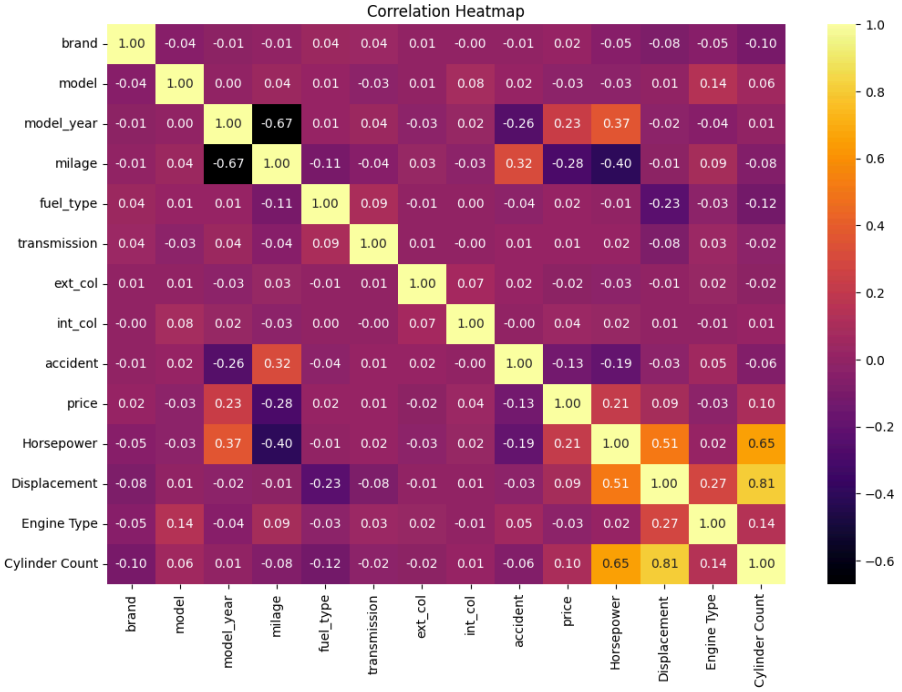
Cuando hemos visualizado un heatmap mostrando la correlación entre las variables obtenemos:



Resulta que la variable *clean_title* tiene el mismo valor en todos los registros de coches. Lo que implica que es irrelevante para la predicción del precio, así que podemos eliminar la columna:

```
train_l = train_l.drop('clean_title', axis=1)
test_l = test_l.drop('clean_title', axis=1)
input_all = input_all.drop('clean_title', axis=1)
```

Ya podemos imprimir el heatmap y obtener un resultado más interpretable:



Finalmente, ya podemos separar el conjunto de entrenamiento en atributos para la predicción y la etiqueta:

```
y_train = train_1.price
X_train = train_1.drop('price', axis=1)
```

El preprocesamiento general de los datos ha sido el detallado anteriormente, pero en el caso del algoritmo K-Nearest-Neighbors hace falta escalar los datos, ya que utiliza distancias para encontrar los vecinos más cercanos. Para hacerlo, hemos usado la clase *StandardScaler*.

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_split)
X_val_scaled = scaler.transform(X_val)
```

Selección y ajuste de modelos

Se probaron varios algoritmos de regresión para encontrar el que mejor se adaptara a los datos. Los modelos evaluados incluyen Random Forest, Gradient Boosting y KNN.

Para todos ellos hemos dividido el conjunto de datos de entrenamiento (*X_train* y *y_train*) en dos subconjuntos: uno para entrenar el modelo (*X_train_split*, *y_train_split*) y otro para validar su rendimiento (*X_val*, *y_val*). Esto se hace utilizando la función *train_test_split* de *sklearn*, reservando el 20% de los datos para validación (*test_size=0.2*) y asegurando que la partición sea reproducible al establecer una semilla aleatoria fija (*random_state=42*). Este paso permite evaluar cómo se desempeñará el modelo en datos no vistos antes de probarlo en el conjunto de prueba final.

Random Forest Regressor

Como primer algoritmo usamos Random Forest Regressor para predecir el precio de los coches porque es un algoritmo bastante versátil y funciona muy bien con problemas de regresión como este. Una de las ventajas de Random Forest es que combina varios árboles de decisión, lo que ayuda a reducir errores y evita que el modelo se ajuste demasiado a los datos de entrenamiento.

En este caso, Random Forest es ideal porque no necesita que los datos estén escalados ni requiere mucho preprocesamiento. Además, puede capturar relaciones complejas entre las variables, que es algo importante cuando hay tantos factores que afectan el precio de un coche, como la marca, el kilometraje o el tipo de motor.

```
# Modelo inicial

model = RandomForestRegressor(random_state=42, n_estimators=100)

model.fit(X_train_split, y_train_split)



# Evaluar en el conjunto de validación
```

```
y_val_pred = model.predict(X_val)

rmse = np.sqrt(mean_squared_error(y_val, y_val_pred))

print(f"RMSE en validación: {rmse}")
```

Gradient Boosted Regressor

Otro algoritmo usado ha sido Gradient Boosted Regressor porque es un modelo que suele dar muy buenos resultados en problemas de regresión, especialmente cuando las relaciones entre las variables son complejas. Este algoritmo construye árboles de decisión de forma secuencial, corrigiendo los errores de los árboles anteriores, lo que lo hace muy eficiente para capturar patrones en los datos y mejorar la precisión.

En el caso de predecir el precio de los coches, donde hay características tan variadas como la marca, el kilometraje y el tipo de combustible, el Gradient Boosted Regressor funciona muy bien porque es capaz de encontrar relaciones no lineales entre las variables. Aunque requiere un poco más de ajuste de hiperparámetros y puede tardar más en entrenarse, el esfuerzo vale la pena porque generalmente produce predicciones más precisas. Además, como en Random Forest, también podemos analizar la importancia de las características para entender qué factores influyen más en el precio.

```
# Definir el modelo

gb_model = GradientBoostingRegressor(n_estimators=100, random_state=42)



# Entrenar el modelo

gb_model.fit(X_train_split, y_train_split)



# Evaluar en validación

y_val_pred_gb = gb_model.predict(X_val)

rmse_gb = np.sqrt(mean_squared_error(y_val, y_val_pred_gb))

print(f"RMSE con Gradient Boosting: {rmse_gb}")
```

K Nearest Neighbors

El último algoritmo usado ha sido K-Nearest Neighbors (KNN) porque es un modelo intuitivo y fácil de implementar que se basa en encontrar patrones locales en los datos. Para este problema de predicción del precio de coches, utilicé un bucle para probar diferentes valores de vecinos (*n_neighbors*) desde 1 hasta 19 y evaluar cómo afecta al rendimiento del modelo. El modelo calcula el precio de un coche como un promedio ponderado de los precios de sus vecinos más cercanos, lo que puede ser útil para capturar similitudes locales entre coches con características similares.

Durante las pruebas, observamos que a medida que aumentaba el número de vecinos, el RMSE disminuía, lo que significa que el modelo mejoraba su capacidad de generalización. Con 19 vecinos, obtuve el RMSE más bajo de *70223.91*, lo que indica que este valor equilibra bien la precisión y la estabilidad del modelo. Aunque KNN puede ser sensible a la escala de los datos, solucioné esto escalando las variables previamente, lo que asegura que las distancias sean comparables. Aunque no es tan sofisticado como otros modelos como Random Forest o Gradient Boosting, KNN resultó ser una opción interesante para explorar el problema.

```
for k in range(1, 20):
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(X_train_scaled, y_train_split)
    y_val_pred = knn.predict(X_val_scaled)
    rmse = np.sqrt(mean_squared_error(y_val, y_val_pred))
    print(f"n_neighbors: {k}, RMSE: {rmse}")
```

```
# Configura el modelo KNN
knn = KNeighborsRegressor(n_neighbors=19)
knn.fit(X_train_scaled, y_train_split)

# Realiza predicciones
y_val_pred = knn.predict(X_val_scaled)
```

Contenido adicional

Además de ejecutar los algoritmos con el preprocesamiento inicial, también decidí probar una variante en el manejo de los valores faltantes numéricos. En lugar de rellenarlos con la media, los manejé utilizando la mediana, ya que esta es menos sensible a valores atípicos que podrían distorsionar los resultados. Con este enfoque, repetí exactamente los mismos algoritmos y configuraciones que antes (Random Forest, Gradient Boosting, y KNN) para comparar cómo afectaba este cambio al rendimiento del modelo. Esto me permitió evaluar si la estrategia de imputación de valores faltantes influía significativamente en las predicciones y, de ser así, cuál era la mejor opción para este problema en particular.

Evaluación de Modelos

La evaluación del rendimiento de los modelos se realizó utilizando el **RMSE** sobre el conjunto de validación. A continuación se presentan los resultados obtenidos para los distintos modelos:

Fecha y hora	Modelo	Score en entrenamiento	Score en Kaggle	Preprocesamiento en valores nulos numéricos
04/01/2025 10h	Random Forest *n_estimators= 100*	74974.0314	70073.34609	Media
04/01/2025 10h	Gradient Boosted *n_estimators= 100*	68270.4726	63810.05168	Media
04/01/2025 10h	KNN *n_neighbors=19*	70223.91421	63810.05168	Media
03/01/2025 17h	Random Forest *n_estimators= 100*	75014.23866	70191.70548	Mediana
03/01/2025 17h	Gradient Boosted *n_estimators= 100*	68366.27574	63759.03503	Mediana
03/01/2025 17h	KNN *n_neighbors=19*	70259.06997	63759.03503	Mediana