

Práctica 3: competición

Abelardo Fernández Campo



**UNIVERSIDAD
DE GRANADA**

Inteligencia de Negocio

4º SI

Email: abefdez@correo.ugr.es/e.abefdez@go.ugr.es

MEJOR RESULTADO



submission_ensemble (15).csv

Complete (after deadline) · 1d ago · prueba

63453.45901

72380.73679

ÍNDICE

ÍNDICE.....	3
1.INTRODUCCIÓN.....	4
2.ELABORACIÓN DE PROPUESTAS.....	6
1.Preprocesamiento.....	6
2.Aplicación de los algoritmos.....	10
2.1Regresión lineal y SGDRegressor:.....	10
2.2 Random Forest Regressor y LightGBM Regressor:.....	11
2.3 XGBoost Regressor:.....	13
Versión 1: XGBoost Regressor Simple.....	13
Versión 2: XGBoost Regressor con Mejora (Ajuste de Hiperparámetros).....	14
Versión 2: XGBoost Regressor con Mejora (Ajuste de Hiperparámetros y mejora preprocesamiento).....	14
2.4 LightGBM con ingeniería de características:.....	16
2.5 Ensemble learning:.....	17
3.Tabla resumen.....	20
Referencias bibliográficas.....	22

1.INTRODUCCIÓN

En esta práctica nos centraremos en el desarrollo y la aplicación de algoritmos de regresión para predecir el precio de coches usados basándose en un conjunto de características específicas de cada vehículo.

Para llevar a cabo esto, utilizaremos Kaggle, una plataforma basada competencias de Ciencia de Datos que permite a la gente competir para resolver desafíos reales mediante el análisis de datos. Kaggle proporciona un entorno ideal para practicar y mejorar nuestras habilidades en la manipulación de datos, el diseño de modelos predictivos y la optimización de su rendimiento.

En esta práctica, se nos suministran tres archivos .csv:

1. `sample_submission.csv`: Un archivo de ejemplo que ilustra el formato esperado para las soluciones propuestas, compuesto por dos columnas: `id` y `price`.
2. `train.csv`: El conjunto de datos de entrenamiento que contiene múltiples instancias de coches usados, cada una con una serie de características descriptivas y el precio correspondiente.
3. `test.csv`: El conjunto de datos de prueba que incluye las mismas características que el conjunto de entrenamiento, pero sin los precios, los cuales serán nuestro objetivo a predecir.

El objetivo principal es desarrollar scripts en Python que apliquen diferentes algoritmos de regresión para generar predicciones de precios lo más precisas posible. Posteriormente, estas predicciones se enviarán a Kaggle para evaluar su rendimiento en comparación con las soluciones de otros participantes, utilizando la métrica de Root Mean Square Error (RMSE), que mide la diferencia entre los valores predichos y los observados.

Los pasos que llevaré a cabo en cada uno de los distintos scripts que vamos a evaluar son:

1. Carga y Exploración de Datos: Importar los conjuntos de datos proporcionados, explorar sus características y comprender la naturaleza de las variables involucradas.
2. Análisis de Correlación: Utilizar visualizaciones, como un heatmap, para identificar relaciones significativas entre las variables independientes y la variable objetivo (`price`), lo que facilitará la selección de características relevantes.
3. Preprocesamiento de Datos: Realizar tareas esenciales como la gestión de valores faltantes, la codificación de variables categóricas y la normalización de variables numéricas para preparar los datos para el modelado.
4. Selección y Entrenamiento de Modelos: Implementar y entrenar diferentes algoritmos de regresión, inicialmente comenzando con modelos sencillos como la Regresión Lineal y explorando otros modelos eficientes como la Regresión con Descenso de Gradiente Estocástico (SGDRegressor).
5. Evaluación de Modelos: Evaluar el rendimiento de los modelos utilizando métricas apropiadas y seleccionar el modelo que ofrezca el mejor equilibrio entre precisión y eficiencia.
6. Optimización y Mejora de Modelos: Aplicar técnicas de optimización de hiperparámetros y considerar métodos avanzados para mejorar la capacidad predictiva de los modelos seleccionados.

7. Generación de Predicciones y Subida a Kaggle: Generar las predicciones finales sobre el conjunto de prueba y preparar el archivo de envío conforme a las especificaciones de Kaggle.

Este enfoque me permite lo primero analizar el conjunto de datos, analizando la relación entre las distintas variables para saber cuales debo seleccionar, luego exploraré dichas variables en búsqueda de valores que necesiten normalización, tratamiento de nulos y ruido, etc. Después de esto aplicar distintos algoritmos de regresión, de forma que obtengamos distintos resultados en búsqueda del más cercano al óptimo.

2. ELABORACIÓN DE PROPUESTAS

Lo primero de todo en estos casos, es decidir qué bibliotecas vamos a utilizar. Como en la anterior práctica vamos a necesitar las siguientes:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Modelos de regresión
from sklearn.linear_model import Ridge, LinearRegression, SGDRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
```

Como es obvio ya necesitamos pandas para la carga y tratamiento de los datos, que en nuestro caso están en formato csv, numpy lo usamos para cuestiones de cálculos numéricos, seaborn y matplotlib.pyplot nos permitirán realizar visualizaciones (gráficas) para la selección de variables y análisis de cada algoritmo. El resto proviene todo de sklearn que permite el uso de los algoritmos y su evaluación.

1. Preprocesamiento

Una vez determinado esto, pasamos a ver qué atributos del conjunto de datos presenta valores nulos. Observando un poco los datos, destaca mucho que la columna 'fuel_type' es de lejos la que más valores nulos presenta, otras columnas como 'engine' o 'int_col' también los presentan, por lo que haremos un tratamiento para todas las variables en el que, si encontramos un valor nulo lo sustituiremos por el valor más frecuente (moda). El código de dicha imputación de valores perdidos es:

```
# 5. Manejo de Valores Faltantes
# Rellenar valores faltantes en variables categóricas con el valor más frecuente (mode)
for col in categorical_features:
    mode_value = X[col].mode()[0]
    X[col] = X[col].fillna(mode_value)
    X_test[col] = X_test[col].fillna(mode_value)
    print(f"Valores nulos en '{col}' han sido reemplazados por '{mode_value}'.")
```

Como se puede ver, un fragmento de código muy simple que recorre todas los atributos categóricos, calcula su moda y substituye los nulos por dicho valor.

En caso de haber valores nulos en variables numéricas, se establece la mediana, de la siguiente forma:

```
# Rellenar valores faltantes en variables numéricas con la mediana
imputer_num = SimpleImputer(strategy='median')
X[numerical_features] = imputer_num.fit_transform(X[numerical_features])
X_test[numerical_features] = imputer_num.transform(X_test[numerical_features])
```

Para continuar con un correcto preprocesamiento, lo siguiente es la captación de outliers y eliminación de ruido, para así tener un conjunto de datos más limpio sin posibles valores muy desviados. El código que realiza ese trabajo es el siguiente:

```
# 6. Eliminación de Ruido: Detección y Capping de Outliers en Características Numéricas
def cap_outliers(df, numerical_cols):
    """
    Copea los outliers en las columnas numéricas utilizando el método IQR.
    """
    for col in numerical_cols:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        before = df[col].shape[0]
        df[col] = np.where(df[col] < lower_bound, lower_bound, df[col])
        df[col] = np.where(df[col] > upper_bound, upper_bound, df[col])
        after = df[col].shape[0]
        print(f"Outliers en '{col}' han sido capados: {before} -> {after}")
    return df

# Aplicar el capping de outliers
X = cap_outliers(X, numerical_features)
X_test = cap_outliers(X_test, numerical_features)
```

Como vemos, se establecen los percentiles, y aquellos valores que están muy desviados del resto se eliminan. Obviamente, este tratamiento solo se hace sobre variables numéricas, como en este caso la mayoría de atributos son categóricos, la mejora de rendimiento va a ser mínima, pero algo de mejora existe.

Como solo existe una clase (precio), no es necesario hacer un balance de clases ni nada por el estilo, luego solo nos falta una fase en el preprocesamiento: selección de características.

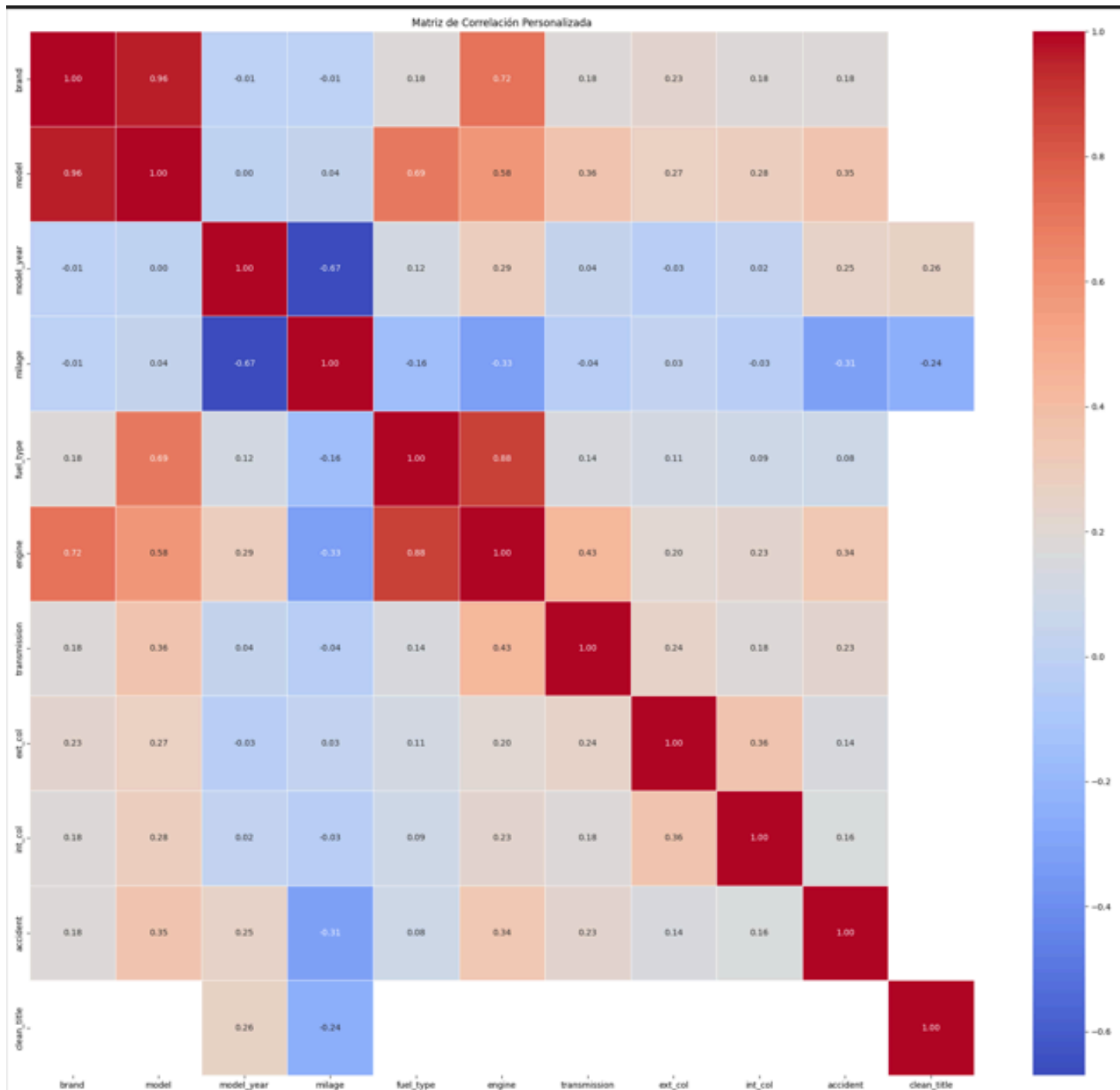
Para la selección de características he decidido crear un heatmap, pero esto se complica a la hora de tener muchas variables categóricas. Para hacerlo, he calculado la correlación de Pearson junto con el cálculo de correlación Cramér's V. Usamos un punto biserial después de codificar las variables categóricas con LabelEncoder, lo que permite crear una matriz de correlación 'personalizada', iterando por cada par de atributos calculando la correlación correspondiente dependiendo del tipo. Almacenamos dicha matriz de correlación y junto con seaborn se hace el heatmap, La implementación de todo lo mencionado es la siguiente:

```
def calculate_custom_correlation_matrix(df, target_col, exclude_cols=[]):
    for var1 in variables:
        for var2 in variables:
            if var1 == var2:
                corr_matrix_custom.loc[var1, var2] = 1.0
            else:
                # Determinar el tipo de variable
                if df[var1].dtype in ['int64', 'float64'] and df[var2].dtype in ['int64', 'float64']:
                    # Correlación de Pearson
                    corr, _ = pearsonr(df[var1], df[var2])
                    corr_matrix_custom.loc[var1, var2] = round(corr, 2)
                elif (df[var1].dtype in ['int64', 'float64'] and df[var2].dtype == 'object') or \
                    (df[var1].dtype == 'object' and df[var2].dtype in ['int64', 'float64']):
                    # Correlación Punto-Biserial
                    try:
                        if df[var1].dtype == 'object':
                            x = LabelEncoder().fit_transform(df[var1])
                            corr, _ = pointbiserialr(x, df[var2])
                        else:
                            x = LabelEncoder().fit_transform(df[var2])
                            corr, _ = pointbiserialr(x, df[var1])
                        corr_matrix_custom.loc[var1, var2] = round(corr, 2)
                    except Exception as e:
                        corr_matrix_custom.loc[var1, var2] = np.nan
                elif df[var1].dtype == 'object' and df[var2].dtype == 'object':
                    # Cramér's V
                    corr = cramers_v(df[var1], df[var2])
                    corr_matrix_custom.loc[var1, var2] = round(corr, 2)
                else:
                    corr_matrix_custom.loc[var1, var2] = np.nan
    return corr_matrix_custom
```

El código calcula la matriz de correlación personalizada para un DataFrame df. Evalúa el tipo de datos de cada par de variables y aplica diferentes métodos de correlación: Pearson para variables numéricas, Punto-Biserial para combinaciones de variables numéricas y categóricas, y Cramér's V para variables categóricas. Los resultados se almacenan en corr_matrix_custom y se asigna np.nan si no se puede calcular la correlación.

Una vez hecho esto, podemos visualizar la correlación entre todas las variables del conjunto de datos, pudiendo analizar si tenemos algunas muy similares, por lo que nos podamos quedar con la más adecuada de las dos, simplificando así los cálculos y mejorando los resultados (siempre dependiendo del algoritmo, ya que habrá algoritmos que funcionen mejor con todas las variables aunque den información muy parecida).

El heatmap conseguido es el siguiente:



Como podemos observar, tenemos unos claros vencedores en cuanto a similaridad entre distintos atributos, 'brand' y 'model'. En este caso, como model en general tiene un valor mayor de correlación con todos los atributos, he optado por eliminar dicho atributo del conjunto de datos, y los resultados mejoran en general, sobre todo para el algoritmo de regresión lineal (profundizaremos más adelante). Otro que resalta es la correlación entre 'fuel_type' y 'engine'. En este caso he optado por eliminar 'fuel_type' del conjunto de datos, ya que tiene un abanico de valores mucho menor, por lo que creo que caracteriza menos a cada tipo de dato (aunque lo eliminemos los resultados por lo general no mejoran, pero simplificamos la selección de características).

El resto de características creo que son indispensables para el estudio, por lo que nos quedaremos con todas menos 'model' y 'fuel_type'.

2. Aplicación de los algoritmos

En este apartado debemos decidir los algoritmos que iremos probando, paso a paso, en cada una de las implementaciones que pruebe probaré dos algoritmos, los comprobaré mediante el uso de métricas, y utilizaré el mejor de los dos algoritmos comparados para realizar la predicción. Una vez hecha la predicción, la subiré a Kaggle, obteniendo la puntuación correspondiente y añadiendo el resultado a la tabla de resultados obtenidos en Kaggle.

Para ser más eficientes y buscar la solución más óptima lo más rápido posible, analizaré algoritmos de dos en dos, evaluando cuál de esos dos es mejor y aplicando dicho mejor para realizar las predicciones que subiré a Kaggle.

Para evaluar cuál de los dos algoritmos tiene mejores resultados utilizo la métrica RMSE, que mide la diferencia promedio entre los valores predichos y los reales (esto para el conjunto de entrenamiento), el que tenga menor RMSE es considerado mejor, porque indica menos discrepancia entre los valores predichos y los valores reales.

Este enfoque me permite evaluar los algoritmos de dos en dos, por lo que es más rápido encontrar una solución más óptima.

Esto todo se ve reflejado en la siguiente implementación:

```
# 9. Definir los modelos de regresión
# Modelo 1: Regresión Lineal
linear_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])

# Modelo 2: Regresión con Descenso de Gradiente Estocástico (SGDRegressor)
sgd_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', SGDRegressor(max_iter=1000, tol=1e-3, random_state=42))
])

# 10. Entrenar los modelos
print("\nEntrenando el modelo de Regresión Lineal...")
linear_pipeline.fit(X_train, y_train)
print("Entrenando el modelo de SGD Regressor...")
sgd_pipeline.fit(X_train, y_train)

# 11. Evaluar los modelos
print("\nEvaluación de Modelos:")

# Predicciones y evaluación para Regresión Lineal
linear_pred = linear_pipeline.predict(X_val)
linear_rmse = np.sqrt(mean_squared_error(y_val, linear_pred))
print(f"Regresión Lineal RMSE: {linear_rmse:.2f}")

# Predicciones y evaluación para SGD Regressor
sgd_pred = sgd_pipeline.predict(X_val)
sgd_rmse = np.sqrt(mean_squared_error(y_val, sgd_pred))
print(f"SGD Regressor RMSE: {sgd_rmse:.2f}")

# 12. Seleccionar el mejor modelo (el que tenga menor RMSE)
if linear_rmse < sgd_rmse:
    best_model = linear_pipeline
    print("\nSeleccionando Regresión Lineal como el mejor modelo.")
else:
    best_model = sgd_pipeline
    print("\nSeleccionando SGD Regressor como el mejor modelo.")
```

Tras determinar cuál de los dos modelos es mejor, entrenamos con el conjunto de entrenamiento de nuevo, hacemos las predicciones sobre el conjunto de prueba, y dichas predicciones las almacenamos en un 'predicciones.csv' el cual tendrá el id de cada coche seguido de su precio, este csv será el que se suba a Kaggle para obtener la puntuación.

Esta estructura la seguirán todos los scripts que vamos a utilizar, cambiando los algoritmos y sus parámetros para obtener la solución más cercana a la óptima.

2.1 Regresión lineal y SGDRegressor:

El primer algoritmo que voy a utilizar, un poco obvio, es el algoritmo de Regresión Lineal, un algoritmo básico y eficiente, pero a la vez eficaz, que nos permitirá tener un enfoque básico de predicción para nuestro problema. Junto a este algoritmo vamos a comparar el SGDRegressor que es un modelo lineal que ajusta minimizando una pérdida empírica regularizada con SGD (Descenso de Gradiente Estocástico). Este último algoritmo lo he sacado de la documentación de scikitlearn, y necesita pasarle tres parámetros a la hora de llamar la función:

- Número máximo de iteraciones (epochs) que el algoritmo de optimización (Descenso de Gradiente Estocástico) realizará sobre el conjunto de datos durante el entrenamiento. Controla cuánto tiempo se entrena el modelo. (establecido en 1000)
- Criterio de tolerancia para la convergencia. Si la mejora en la función de pérdida entre iteraciones consecutivas es menor que este valor, el entrenamiento se detiene. (establecido en $1e-3$)
- Semilla utilizada por el generador de números aleatorios para asegurar la reproducibilidad de los resultados. (establecido en 42)

Una vez establecidos estos parámetros de función, tenemos que probar los siguientes parámetros, que son más importantes:

- alpha: Controla la fuerza de la regularización para prevenir el sobreajuste.
- penalty: Define el tipo de regularización a aplicar ('l2', 'l1', 'elasticnet').
- learning_rate: Determina la estrategia de actualización de la tasa de aprendizaje ('constant', 'optimal', 'invscaling').
- eta0: Establece la tasa de aprendizaje inicial para las actualizaciones del modelo.


El resto de parámetros son los que tiene el algoritmo por defecto.

Los resultados obtenidos al ejecutar el script son los siguientes:

```
Evaluación de Modelos:
Regresión Lineal RMSE: 68879.01
SGD Regressor RMSE: 68492.75

Seleccionando SGD Regressor como el mejor modelo.
```

Como vemos, aunque el SGDRegressor tiene mejor rendimiento, es muy similar al de Regresión Lineal. En este caso, subiremos las predicciones a Kaggle para obtener la puntuación que en este caso nos dá:

	predicciones.csv	63994.07630	72927.00446
	Complete (after deadline) · 12d ago · SGDRegressor con preprocesamie...		

Como vemos es un resultado bastante decente para ser un script simple que se ejecuta muy rápido. (detalles de la subida a Kaggle en la tabla final).

2.2 Random Forest Regressor y LightGBM Regressor:

En la comparación de estos dos algoritmos el resultado fue el siguiente:

```
Evaluación de Modelos:
Random Forest Regressor RMSE: 77792.65
LightGBM Regressor RMSE: 68286.54

Seleccionando LightGBM Regressor como el mejor modelo.
```

Así que en un principio, comencé por escoger el LightGBM como mejor algoritmo.

LightGBM (Light Gradient Boosting Machine) es un algoritmo de boosting basado en árboles que se destaca por su eficiencia y rendimiento. Está diseñado para manejar grandes volúmenes de datos y puede capturar relaciones complejas entre características. LightGBM utiliza técnicas como el crecimiento de árbol basado en hojas, lo que permite una mayor precisión y velocidad en comparación con otros algoritmos (mucho más rápido que el Random Forest).

Parámetros Clave:

Al configurar el LGBMRegressor, es crucial ajustar ciertos parámetros para maximizar su rendimiento:

- **n_estimators**: Número de árboles en el modelo. Un valor mayor puede mejorar la precisión pero aumenta el tiempo de entrenamiento.
- **learning_rate**: Tasa de aprendizaje que controla la contribución de cada árbol al modelo final. Valores más bajos pueden mejorar la generalización.
- **num_leaves**: Número de hojas por árbol. Controla la complejidad del modelo. Valores más altos permiten capturar más relaciones complejas.
- **max_depth**: Profundidad máxima de los árboles. Limita la complejidad y previene el sobreajuste.
- **random_state**: Semilla para asegurar la reproducibilidad.
- **boosting_type**: Tipo de boosting a utilizar. Por defecto, LightGBM usa 'gbdt' (Gradient Boosting Decision Tree).

Configuración y Entrenamiento

Configuramos el LGBMRegressor de la siguiente manera:

- **n_estimators=100**: Configuramos 100 árboles para un buen balance entre rendimiento y tiempo de entrenamiento.
- **learning_rate=0.1**: Una tasa de aprendizaje estándar que proporciona una buena convergencia.
- **num_leaves=31**: Permite capturar interacciones no lineales sin sobreajustar.
- **random_state=42**: Garantiza la reproducibilidad de los resultados.


Evaluación del Modelo

Tras entrenar el modelo, evaluamos su rendimiento en el conjunto de validación utilizando el RMSE.

LightGBM Regressor RMSE: 68286.54

El LightGBM Regressor alcanzó un RMSE de 68286.54, superando notablemente al Random Forest Regressor en términos de precisión, y sobre todo de eficiencia, ya que el Random Forest tardó mucho en ejecutarse. Esta mejora sugiere que LightGBM es capaz de capturar de manera más eficiente las relaciones complejas en los datos, ofreciendo predicciones más precisas.

Tras determinar que LightGBM Regressor ofrece un rendimiento ligeramente superior, procedimos a realizar las predicciones finales sobre el conjunto de prueba y subimos los resultados a Kaggle. Las puntuaciones obtenidas fueron:

	predicciones.csv Complete (after deadline) · 14d ago · prueba lightGBM Regressor	63891.92838	72939.64336
---	--	--------------------	--------------------

Como podemos observar, un rendimiento muy parecido al anterior caso, ya que todavía hemos dejado el mismo procesamiento de los datos.

El Random Forest Regressor intenté de varias maneras probar su rendimiento, ya que tengo entendido que es un gran algoritmo aunque un poco complejo. Probé tanto a mejorar sus parámetros como a establecer mejoras en el preprocesamiento, centrándose en el tratamiento de outliers y en aplicar ingeniería de características, pero el trabajo fue en vano para este algoritmo ya que los resultados no fueron buenos, además de las largas esperas para la ejecución de este.

2.3 XGBoost Regressor:

Después de comprobar el rendimiento de algoritmos como la Regresión Lineal, el SGDRegressor, el Random Forest y LightGBM, ante el buen rendimiento sobre todo de LightGBM, decidí probar también XGBoost debido a la comparación que hicimos en teoría sobre estos dos algoritmos.


Versión 1: XGBoost Regressor Simple

En la primera aproximación, empleé un XGBRegressor con una configuración relativamente básica:

- **n_estimators**: Establecí un número medio de árboles para equilibrar el tiempo de entrenamiento y la capacidad de generalización.
- **learning_rate**: Mantuve un valor estándar (0.1) que ofreciera un equilibrio entre convergencia y estabilidad.
- **max_depth**: Seleccioné una profundidad moderada, previniendo un sobreajuste excesivo.
- **random_state**: A 42 para asegurar reproducibilidad.

En esta versión he usado el mismo preprocesamiento que anteriormente, utilizando un pipeline para establecerlo.

Tras construir el pipeline de preprocesamiento (con imputación de valores nulos, transformación de outliers y codificación de variables categóricas), se entrenó el XGBRegressor simple y evaluamos usando la métrica de RMSE en el conjunto de validación. Aunque los resultados fueron satisfactorios, se obtuvo un resultado algo peor que en los anteriores casos, el cual fue el siguiente:

	predicciones_xgb.csv Complete (after deadline) · 6m ago · xgboost	65415.51222	74239.46644
---	---	--------------------	--------------------

Como se puede ver todavía hay margen de mejora. Esto se debe a que es un algoritmo más complejo, por lo que debemos perfeccionar más el preprocesamiento.

Versión 2: XGBoost Regressor con Mejora (Ajuste de Hiperparámetros)

Tras la primera prueba, decidí optimizar ciertos hiperparámetros de XGBoost para exprimir al máximo su capacidad predictiva. Para ello, se usó un enfoque basado en técnicas como RandomizedSearchCV o GridSearchCV, ajustando parámetros cruciales como:

- `n_estimators` (por ejemplo, probando rangos como 200, 300 o 500),
- `learning_rate` (0.01, 0.05, 0.1),
- `max_depth` (3, 5, 7),
- Otras variables como `subsample`, `colsample_bytree` o `reg_alpha`, `reg_lambda` en el caso de un RandomizedSearch más extenso.

Este proceso de ajuste (o *tuning*) me permitió encontrar una configuración más apropiada, mejorando así el RMSE en validación. Con los parámetros óptimos, se reentrenó todo el conjunto de datos de entrenamiento y, finalmente, se obtuvieron las predicciones sobre el conjunto de prueba para subirlas a Kaggle.

Realizando esta búsqueda de mejores parámetros (muy sencilla pero eficiente) mejoramos notablemente el resultado obtenido, el cual fue el siguiente:



predicciones_xgb_mejorado.csv

Complete (after deadline) · 19s ago · xgb mejorado

64209.25228

73282.59892

Como podemos observar mejoró notablemente el resultado, pero sigue por debajo del resultado de los anteriores algoritmos, como mencioné anteriormente, añadiremos un mejor preprocesamiento para la siguiente versión.

Versión 2: XGBoost Regressor con Mejora (Ajuste de Hiperparámetros y mejora preprocesamiento)

En este caso, voy a aplicar ingeniería de características, que no será más que aplicar distintos cambios a las características del dataset en sí para dividir la información y crear categorías más concretas que tengan más relevancia y puedan mejorar las predicciones.

En mi caso, he implementado la siguiente función:

```
# =====
# 2. Ingeniería de Características
# =====
def feature_engineering(df):
    # 1) Crear la edad del coche
    if 'model_year' in df.columns:
        df['car_age'] = 2025 - df['model_year']
    else:
        df['car_age'] = np.nan # Por si no existe, no romper

    # 2) Extraer información del motor con regex
    if 'engine' in df.columns:
        df['engine'] = df['engine'].astype(str)
        import re
        engine_regex = (
            r'(?P<engine_hp>\d+\.?\d*)HP.*?'
            r'(?P<engine_displacement>\d+\.?\d*)L.*?'
            r'(?P<engine_cylinders>\d+) Cylinder'
        )
        extracted = df['engine'].str.extract(engine_regex, expand=True)
        df['engine_hp'] = pd.to_numeric(extracted['engine_hp'], errors='coerce')
        df['engine_displacement'] = pd.to_numeric(extracted['engine_displacement'], errors='coerce')
        df['engine_cylinders'] = pd.to_numeric(extracted['engine_cylinders'], errors='coerce')
        df.drop(columns=['engine'], inplace=True, errors='ignore')
    else:
        df['engine_hp'] = np.nan
        df['engine_displacement'] = np.nan
        df['engine_cylinders'] = np.nan

    # 3) is_automatic (1 si A/T o Automatic, 0 si no)
    if 'transmission' in df.columns:
        df['is_automatic'] = df['transmission'].str.contains('A/T|Automatic', na=False).astype(int)
    else:
        df['is_automatic'] = np.nan

    # 4) had_accidents (1 si 'At least 1 accident' o 'damage reported')
    if 'accident' in df.columns:
        df['had_accidents'] = df['accident'].str.contains('At least 1 accident|damage reported', na=False).astype(int)
        df.drop(columns=['accident'], inplace=True, errors='ignore')
    else:
        df['had_accidents'] = np.nan

    return df
```

La función `feature_engineering(df)` es un componente crucial del preprocesamiento de datos que realiza varias transformaciones importantes.

Lo primero que hace es calcular la edad de un vehículo (`car_age`), que no es más que la diferencia que el año actual (2025 para tenerlo recién actualizado) y el año del modelo del vehículo. Me parece interesante realizar esto ya que es un valor mucho más directo de la antigüedad que un año de modelo.

Lo siguiente que se realiza es, mediante una expresión regular, extraer las distintas partes de información que hay en la característica 'engine', de la cual extraemos las siguientes subcaracterísticas:

- `engine_hp`: Caballos que tiene el motor
- `engine_displacement`: Desplazamiento del motor (litros)
- `engine_cylinders`: Número de cilindros

De esta forma el algoritmo tendrá información más específica, de forma que podrá diferenciar mejor las distintas instancias.

La función también detecta la transmisión automática mediante la nueva subcategoría 'is_automatic', busca los términos 'A/T' o 'Automatic', si encuentra dichos términos es 1, si no 0.


Por último, vi conveniente eliminar la columna 'accidents' sustituyendola por una booleana de forma que sea 1 si tuvo accidentes o 0 si no, la cual se llama 'had_accidents'.

En resumen, esta función es fundamental para el preprocesamiento ya que:

1. Crea características más informativas a partir de datos crudos
2. Estandariza la información en formatos numéricos utilizables
3. Extrae información valiosa de campos de texto
4. Maneja de forma robusta los casos donde faltan columnas o datos

La transformación de características realizada por esta función es crucial para mejorar la calidad de los datos.

Tras añadir esto al algoritmo XGBoost, se mejoraron sus predicciones un poco, siendo ahora su resultado en Kaggle el siguiente:

	predicciones_xgb_mejorado (2).csv	63985.99225	72996.20212
	Complete (after deadline) · 14s ago · xgboost con ingeniería de caracter...		


Como vemos, hemos alcanzado los resultados obtenidos por LightGMB o SGDRegressor, ya que ahora la entrada al algoritmo está mucho más depurada. En mi opinión, este algoritmo funciona peor por su complejidad, tiene un gran número de parámetros a configurar lo cual se hace imposible conseguir la combinación óptima, si bien he usado el Randomized y el Grid para probar una gran cantidad de parámetros, siempre nos dejamos muchas posibilidades, y probarlos todos es imposible computacionalmente (por lo menos para mi ordenador o notebooks de Google Colab).

La cuestión en este punto es, si el XGBoost mejoró notablemente con la ingeniería de características, por qué no lo pruebo en LightGMB?, pues esto será lo que probaremos ahora.

2.4 LightGMB con ingeniería de características:

En este caso, utilizaré el mismo código utilizado para el apartado 2.2, lo único que haré será eliminar el entrenamiento del Random Forest y añadiré la ingeniería de características vista en el anterior apartado.

Haciendo simplemente eso, obtenemos el siguiente resultado:

	predicciones_lightgbm.csv	63831.98014	72783.20402
	Complete (after deadline) · 15s ago · lightgmb con ingeniería de caracte...		

Hasta ahora el mejor resultado que hemos obtenido, esto sin tocar los parámetros establecidos en el apartado 2.2, por lo que ahora, implementamos el RandomizedSearchCV, al igual que con el anterior algoritmo, para ver si todavía podemos afinar más los resultados de este algoritmo.

La configuración es la siguiente:

```
# Definimos un rango de parámetros para LightGBM
param_dist = {
    'regressor__n_estimators': [500, 1000, 2000],
    'regressor__learning_rate': [0.01, 0.05, 0.1],
    'regressor__max_depth': [5, 7, 10],
    'regressor__num_leaves': [31, 63, 127],
    'regressor__subsample': [0.8, 1.0],
    'regressor__colsample_bytree': [0.8, 1.0]
}
```


son parámetros estándar (bajos, medios y altos), para ver si podemos mejorar la predicción ajustando los parámetros, resultado el cuál fue el siguiente:



predicciones_lightgbm (1).csv

Complete (after deadline) · 18s ago · lightgmb ingenieria de caracteristic...

63710.91443

72813.16783

Como podemos observar, hemos mejorado notablemente el resultado, solo asignando unos parámetros un poco más ajustados. Al igual que en el caso anterior, encontrar la combinación óptima es muy complejo computacionalmente.

Llegado a este punto, hice muchas pruebas, probar algoritmos distintos, escalas logarítmicas en el preprocesamiento, pero estaban los resultados muy estancados y no conseguía ninguna manera de optimizar el resultado obtenido. Esto fue así hasta que conocí el ensemble learning que básicamente es una técnica que en vez de utilizar las predicciones de un solo algoritmo, combina las predicciones de varios algoritmos, permitiendo obtener resultados más equilibrados y óptimos. Eso es lo que haremos en el siguiente apartado.

2.5 Ensemble learning:

En este apartado, usaremos el ensemble learning con el objetivo de obtener unos resultados mejores. Veo conveniente en este apartado explicar un algoritmo que voy a utilizar, que no ha sido mencionado anteriormente, llamado CatBoostRegressor.

El CatBoostRegressor es un algoritmo de gradient boosting, diseñado para trabajar eficazmente con variables categóricas sin necesidad de un extenso preprocesamiento. Se basa en árboles de decisión y utiliza técnicas de ordenamiento y procesamiento especial de variables categóricas para reducir el sesgo y mejorar la generalización. Entre sus parámetros clave se incluyen:

- **iterations:** Número de iteraciones o árboles que el modelo construirá. En el código se establecen 5000 iteraciones para permitir un aprendizaje profundo del conjunto de datos.
- **learning_rate:** La tasa de aprendizaje controla cuánto contribuye cada árbol nuevo a la predicción final. Un valor de 0.02 equilibra la velocidad de aprendizaje con la estabilidad del proceso.
- **depth:** La profundidad máxima de cada árbol, configurada a 11 en este caso, que permite capturar interacciones complejas entre variables.
- **l2_leaf_reg:** Parámetro de regularización L2 aplicado a los valores en las hojas del árbol para prevenir el sobreajuste.
- **random_seed:** Semilla para asegurar la reproducibilidad de los resultados.
- **task_type:** Especifica el uso de la GPU para acelerar el entrenamiento, si está disponible.

He decidido añadir el uso de este algoritmo porque mirando los foros de Kaggle, vi que la gente hablaba bastante bien de su rendimiento para este caso.

Después de definir los parámetros para LightGBM, CatBoostRegressor y XGBoostRegressor, instancio cada uno de los modelos y los entreno utilizando validación cruzada (CV) con early

stopping. Durante el entrenamiento, la función `cross_validate_and_predict`, la cual es la siguiente:

```
def cross_validate_and_predict(model, X, y, X_test, n_splits=5,
                              early_stop_rounds=100, random_state=42,
                              use_cat_features=False):
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=random_state)
    test_preds_folds = np.zeros((len(X_test), n_splits), dtype=np.float32)

    for fold, (trn_idx, val_idx) in enumerate(kf.split(X, y)):
        print(f"\n=== Fold {fold+1} de {n_splits} ===")
        X_trn, X_val = X.iloc[trn_idx], X.iloc[val_idx]
        y_trn, y_val = y.iloc[trn_idx], y.iloc[val_idx]

        # Entrenamiento del modelo
        if isinstance(model, lgb.LGBMRegressor):
            # lightGBM con callbacks para early stopping
            callbacks = [
                early_stopping(stopping_rounds=early_stop_rounds),
                log_evaluation(period=100)
            ]
            model.fit(
                X_trn, y_trn,
                eval_set=[(X_val, y_val)],
                callbacks=callbacks
            )

        elif isinstance(model, CatBoostRegressor):
            # CatBoost con early stopping
            cat_params = {
                'silent': True,
                'early_stopping_rounds': early_stop_rounds,
            }
            if use_cat_features:
                # Extraemos las columnas categóricas
                cat_features_idx = [
                    X_trn.columns.get_loc(c)
                    for c in X_trn.select_dtypes(include='category').columns
                ]
                model.fit(
                    X_trn, y_trn,
                    eval_set=(X_val, y_val),
                    cat_features=cat_features_idx,
                    **cat_params
                )
            else:
                model.fit(
                    X_trn, y_trn,
                    eval_set=(X_val, y_val),
                    **cat_params
                )

        elif isinstance(model, xgb.XGBRegressor):
            # XGBoost con early stopping
            model.fit(
                X_trn, y_trn,
                eval_set=[(X_val, y_val)],
                verbose=100 # Mostrar log cada 100 iteraciones
            )
        else:
            # Para otros (ej. SGDRegressor), no hay early stopping
            model.fit(X_trn, y_trn)

        # Predicción en val
        val_pred = model.predict(X_val)
        fold_rmse_score = rmse(y_val, val_pred)
        print(f"Fold {fold+1} RMSE: {fold_rmse_score:.2f}")

        # Predicción en X_test
        test_preds_folds[:, fold] = model.predict(X_test)

    # Promediamos las predicciones
    test_preds_mean = np.mean(test_preds_folds, axis=1)
    return test_preds_mean
```

Ejecuta un KFold para generar múltiples predicciones de validación y predice sobre el conjunto de prueba.

Para ensamblar las predicciones de los distintos algoritmos, primero obtengo tres vectores de predicciones: uno para LightGBM (`lgb_preds`), otro para CatBoostRegressor (`cat_preds`) y otro para XGBoostRegressor (`xgb_preds`). Luego, en la etapa de blending final, combino estas predicciones mediante un promedio simple, asignando un peso igual a cada modelo. Esto se realiza con la expresión:

```
# =====  
# 7. Blending final (triple ensemble)  
# =====  
final_preds = (lgb_preds + cat_preds + xgb_preds) / 3.0  
final_preds = np.clip(final_preds, 0, None)
```


Este enfoque asume que cada modelo aporta valor predictivo, y al promediar sus salidas se mitigan los errores específicos de cada uno, logrando una predicción más robusta. Finalmente, se aplica `np.clip` para asegurar que no existan precios negativos y se genera el archivo de envío para Kaggle. De este modo, el ensemble learning combina la eficiencia de LightGBM, la capacidad de manejo de variables categóricas de CatBoost y la potencia de XGBoost, con la esperanza de mejorar la precisión del modelo final.

Por temas de rendimiento, no he podido aplicar la función `cross_validate_and_predict` al XGBoost, ya que no fui capaz de conseguir que terminase. En su defecto, utilicé la misma configuración que en el anterior apartado para combinar los resultados para el ensemble. Los resultados de las predicciones fueron los siguientes:

	submission_ensemble (7).csv Complete (after deadline) · 14s ago · prueba	63521.86079	72492.72797
---	--	--------------------	--------------------

Como vemos, hemos mejorado notablemente los resultados gracias al ensemble learning.

Tras varias pruebas e impresiones de los resultados del RMSE de cada algoritmo, me di cuenta que el que mayor RMSE tenía era el XGBoost, por lo que hice una prueba sobre el mismo código, pero utilizando solo el LightGBM y el CatBoost, y los resultados fueron los siguientes:

	submission_ensemble (15).csv Complete (after deadline) · 1d ago · prueba	63453.45901	72380.73679
---	--	--------------------	--------------------

Como vemos, el resultado todavía ha mejorados más, por lo que a partir de aquí, para ahorrar tiempo y intentar mejorar los resultados, todos los intentos son sobre esos dos algoritmos.

3. Tabla resumen

Fecha/Hora	Score en Entrenamiento	Score en Kaggle	Preprocesamiento	Algoritmo(s) y Parámetros
25/12/2024 11:00	RMSE = 68540.99	63994 (Privada) 72927 (Pública)	- Eliminación de outliers (percentiles 1 y 99) - Imputación de nulos en categóricas con la moda - Escalado en numéricas con StandardScaler	Regresión Lineal + SGDRegressor SGDRegressor(max_iter=1000, tol=1e-3, alpha=0.0001, penalty='l2', random_state=42) Se compara con LinearRegression(); gana SGD por ~RMSE=74, se sube la predicción de SGD.
23/12/2024 13:45	RMSE = 68286.54	63891 (Privada) 72939 (Pública)	- Eliminación de outliers (cap IQR 1.5) - Imputación de nulos en numéricas con la mediana - Reducción de variables correlacionadas (sin model, sin fuel_type)	RandomForestRegressor(n_estimators=100) vs. LightGBM LightGBM con n_estimators=100, learning_rate=0.1, num_leaves=31, random_state=42 Gana LightGBM, se sube su predicción.
5/01/2025 9:30	RMSE = 70,155	65415(Privada) 74239(Pública)	- Igual preprocesamiento anterior	XGBoost Regressor (versión 1, simple): n_estimators=300, learning_rate=0.1, max_depth=7, random_state=42 Sin una búsqueda exhaustiva de hiperparámetros.
5/01/2025 16:10	RMSE = 69500	64209(Privada) 73282 (Pública)	- Mismo preprocesamiento - Búsqueda de hiperparámetros con RandomizedSearchCV en XGBoost - Ajuste final con best_params	XGBoost Regressor (versión 2, con tuning): RandomizedSearchCV: - n_estimators en [200,300,500] - learning_rate en [0.01,0.05,0.1] - max_depth en [3,5,7] etc. Obtenemos best_params y reentrenamos en todo el train.
5/01/2025 20:10	RMSE = 68800	63985(Privada) 72996 (Pública)	- Mismo preprocesamiento - Búsqueda de hiperparámetros con RandomizedSearchCV en XGBoost -Ingeniería de características (car_age, engine_hp, etc.) - Ajuste final con best_params	XGBoost Regressor (versión 3, con ingeniería de características): RandomizedSearchCV: - n_estimators en [200,300,500] - learning_rate en [0.01,0.05,0.1] - max_depth en [3,5,7] etc. Obtenemos best_params y reentrenamos en todo el train.

6/01/2025 22:20	LGB: RMSE=67970 Cat: RMSE=68160 XGB=68800	63521(Priv ada) 72492 (Pública)	- Mismo preproc + feature_engineering + outlier capping - Mínimo tuning en LightGBM y CatBoost - XGBoost entrenado aparte (sin CV) - Ensemble final = (lgb_preds + cat_preds + xgb_preds)/3	LightGBM(n_estimators=6000, lr=0.0015, num_leaves=80, ...), CatBoost(iterations=5000, lr=0.02, depth=11, ...), XGBoost(n_estimators=300, lr=0.05, max_depth=7, ...). Blending final con pesos iguales.
7/01/2025 14:20	LGB: RMSE=67970 Cat: RMSE=68160	63453(Priv ada) 72380(Públ ica)	- Mismo preproc + feature_engineering + outlier capping - Mínimo tuning en LightGBM y CatBoost - Ensemble final = (lgb_preds + cat_preds + xgb_preds)/3	LightGBM(n_estimators=6000, lr=0.0015, num_leaves=80, ...), CatBoost(iterations=5000, lr=0.02, depth=11, ...), Blending final con pesos iguales.

Realmente realicé muchas más submission, pero la mayoría de ellas fueron cambios pequeños, con ajustes de parámetros o pruebas rápidas, las realmente relevantes para el desarrollo y desenlace del resultado son estas.

