INTELIGENCIA DE NEGOCIO

Práctica 3. Competición en Kaggle

Alejandro Coman Venceslá

Índice

1. Introducción

En esta práctica hemos abordado una competición en *Kaggle*, en concreto *Regression of Used Car Prices*. El objetivo de esta competición es predecir el precio de coches usados. La métrica de evaluación usada es la raíz del error cuadrático medio (*root-mean-square error,* RMSE) entre los valores predichos y los valores observados.

1.1. Entorno / software utilizado.

Esta práctica ha sido realizada con el lenguaje de programación ***Python,*** específicamente con ***Jupyter Notebooks***, con el apoyo de librerías como *numpy, pandas, scikit-learn, etc…,* en un entorno de ***Anaconda*** y programando en ***Visual Studio Code***.

1.2. Estrategia seguida

Mi primer instinto, al darme cuenta de que la competición se trata de una ya finalizada, fue dirigirme a los equipos que obtuvieron las primeras posiciones y estudiar brevemente sus enfoques e intentar entender por qué fueron los mejores. Tras haber hecho esto, me dí cuenta de que los modelos de ensamble han proporcionado los mejores resultados de la competición. Es por ello que he decidido entrenar distintos algoritmos por separado y luego promediar todas las predicciones de los mismos.

1.3. Scripts utilizados

En esta práctica me he apoyado en el uso de varios Jupyter Notebooks, ya que he realizado distintas pruebas con varios algoritmos. A continuación voy a enumerar los distintos scripts:

- ***practica3_base.ipynb:*** este ha sido el primer script que he usado para las primeras pruebas de la práctica. He aplicado un preprocesado simple en el cual he imputado valores perdidos de las variables.

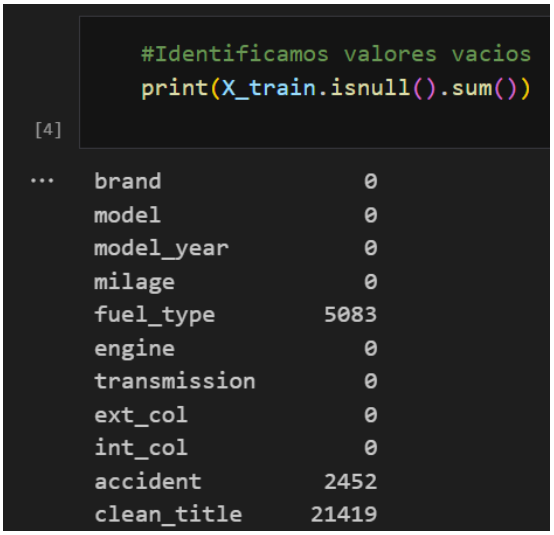
- ***practica3_logprice.ipynb:*** segundo script en el cual he añadido una capa más de preprocesamiento a la variable “price”, pues tiene valores bastante extremos y, en un intento de mitigar este efecto en la predicción, convertí la variable objetivo a una escala logarítmica. Una vez hechas las predicciones sobre el conjunto de datos “*test*” se convierten mediante un exponente.

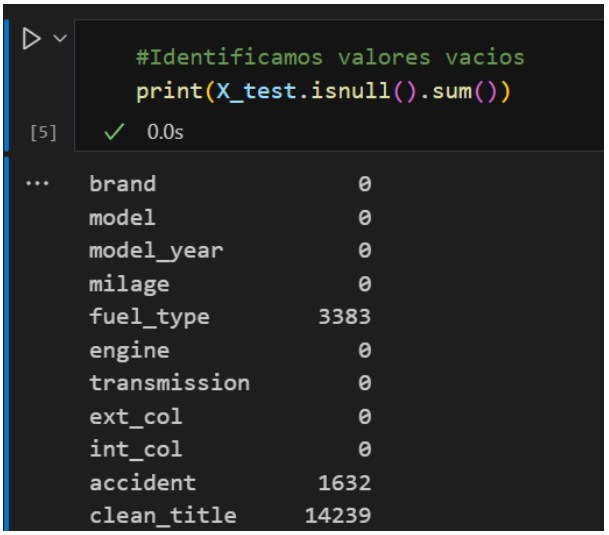
- ***practica3_h2oml.ipynb*:** tercer script en el que he utilizado H2O AutoML para la regresión.. Se trata de una herramienta de aprendizaje automático automatizado que facilita el entrenamiento, evaluación y selección de modelos para tareas como clasificación, regresión y series temporales. Automatiza procesos como la preparación de datos, selección de características y optimización de modelos, generando opciones como GBM, Random Forest y Stacked Ensembles.

1.4. Preprocesamiento utilizado

Aunque no he aplicado un preprocesamiento muy profundo, voy a enumerar con algo más de detalle los preprocesados que he usado en las distintas pruebas.

- *Imputación de valores perdidos:* Necesaria pues varias variables (todas categóricas en este caso) tenían valores perdidos. He barajado dos opciones para resolver este problema: reemplazar el valor vacío por la categoría más frecuente o crear una nueva categoría "unknown". Finalmente, me decanté por la segunda opción, pues pienso que es la opción más representativa al no estar "engañando" a los algoritmos con valores que realmente son incorrectos para determinados datos.





Figuras 1 y 2. Comprobación de valores perdidos en "train.csv" y "test.csv" respectivamente.

- *Transformación de la variable objetivo a una escala logarítmica:* tuve en cuenta la posibilidad de que el gran rango de valores de la variable precio, causado por valores bastante extremos (coches muy caros) pudiera dificultar el aprendizaje de los modelos. Es por ello que decidí convertir la escala de esta variable objetivo a una logarítmica, reduciendo significativamente el rango de valores.

  No obstante, esto implica que, al final de la ejecución de los script en los que aplique este preprocesamiento, debo revertir esta conversión de escala, pues las predicciones también estarán en escala logarítmica. Esto es bastante sencillo pues con un exponente queda solucionado.

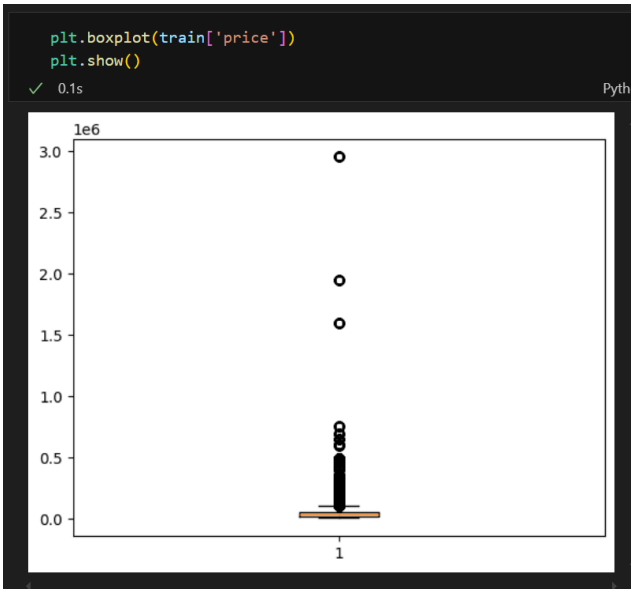


Figura 3. Boxplot que muestra el rango de valores de la variable "price".

```
# Transformar la variable de preci
Y_train = train['price']
Y_train.describe()
```
[9] ✓ 0.0s

```
count    1.885330e+05
mean     4.387802e+04
std      7.881952e+04
min      2.000000e+03
25%      1.700000e+04
50%      3.082500e+04
75%      4.990000e+04
max      2.954083e+06
Name: price, dtype: float64
```

```
# Transformar la variable de precio
Y_train = np.log1p(train['price'])
Y_train.describe()
```
[8] ✓ 0.0s

```
count    188533.000000
mean         10.291787
std           0.844173
min           7.601402
25%           9.741027
50%          10.336114
75%          10.817796
max          14.898699
Name: price, dtype: float64
```

Figuras 4 y 5. Variable "price" antes y después de convertirla a una escala logarítmica.

- *Label encoding:* algunos algoritmos empleados para esta práctica necesitan que todas las variables sean numéricas. Es por ello que he aplicado "Label encoding" que ha convertido todas las categorías de cada variable a números, empezando por el 0 e incrementando de uno en uno hasta el número de categorías de cada variable categórica.

- *Normalización:* tras haber aplicado Label Encoding para algunos algoritmos, he decidido normalizar las variables para un mejor aprendizaje, especialmente en el caso de la Red Neuronal que he usado.

1.5. Algoritmos utilizados:

- CatBoost: excelente para problemas de regresión con variables principalmente categóricas, como lo es este caso, porque implementa un manejo nativo y eficiente de estas variables, sin necesidad de transformarlas mediante técnicas como one-hot encoding o label encoding, lo que reduce la dimensionalidad y evita sobreajuste. Utiliza un enfoque basado en estadísticas acumuladas que convierte las categorías en valores numéricos considerando su distribución en los datos, preservando así patrones relevantes. Además tiene una gran resistencia ante el sobreajuste.

- LightGBM: muy eficaz en regresiones con variables categóricas porque utiliza un enfoque basado en histogramas que reduce el tiempo de entrenamiento y mejora la eficiencia computacional. Puede manejar variables categóricas directamente mediante su técnica de codificación basada en enteros, preservando patrones importantes sin necesidad de realizar un preprocesamiento intensivo como el one-hot encoding. Su diseño optimizado para grandes conjuntos de datos y su capacidad para dividir datos de manera inteligente en hojas con mayor ganancia hacen que sea ideal para capturar relaciones complejas en problemas de regresión.

- RandomForest: construye múltiples árboles de decisión sobre subconjuntos de datos y promedia los resultados, captura relaciones no lineales y reduce el riesgo de sobreajuste. Además, su capacidad para seleccionar automáticamente las características más importantes permite aprovechar las variables categóricas relevantes, incluso en problemas con datos mixtos y ruido, proporcionando predicciones robustas y precisas.

- Red Neuronal: eficaces en regresiones con variables categóricas porque pueden aprender representaciones complejas y no lineales de los datos. Aunque requieren que las variables categóricas se preprocesen para cambiarlas a variables numéricas, este enfoque permite capturar relaciones intrínsecas entre categorías. Su arquitectura es flexible, y permite modelar patrones ocultos y generar predicciones precisas incluso en problemas con alta dimensionalidad o características combinadas.

- XGBoost: efectivo en regresiones con variables categóricas porque puede manejar estas variables mediante técnicas de codificación eficientes, como el uso de la codificación ordinal o la conversión directa durante el entrenamiento. Su enfoque de boosting, que combina varios modelos débiles para mejorar el rendimiento, permite capturar relaciones no lineales y complejas en los datos. Además, XGBoost es altamente eficiente y flexible, optimizando tanto el uso de recursos como la precisión del modelo, lo que lo hace especialmente adecuado para trabajar con datos mixtos, incluidos aquellos con variables categóricas. Otro plus de este modelo es que permite aprovechar la potencia de una GPU dedicada para acelerar su ejecución.

- H2O AutoML: eficaz en regresiones como las de este problema ya que aparte de automatizar el preprocesamiento, selecciona el mejor modelo para el conjunto de datos, incluyendo el manejo de variables categóricas. Optimiza el rendimiento sin que sea necesaria una intervención manual. Su capacidad para probar múltiples enfoques, incluidos algoritmos de boosting y redes neuronales, y combinar modelos mediante el stacking (o super learning), le permite capturar patrones complejos en los datos, incluyendo aquellos relacionados con variables categóricas, mejorando así la precisión y eficiencia del modelo.

2. Descripción de las pruebas realizadas.

Antes de comenzar, cabe mencionar que en todos los algoritmos he utilizado el 80% de los datos contenidos en "train.csv" para entrenar los distintos modelos, reservándome el otro 20% de los datos para validar los datos que vamos entrenando.

Además vamos a utilizar algunas siglas para las métricas de evaluación:

- RMSE entrenamiento: **RE**
- RMSE Kaggle (Privado): **RPr**
- RMSE Kaggle (Público): **RPu**

Realmente los dos que se nos piden en la práctica serían **RE y RPr**, ya que RPu se calculaba únicamente con el 20% de los datos de test de la competición y, una vez acabó, se calculaba RPr con el otro 80% de los datos de test, siendo esta la puntuación definitiva.

2.1. CatBoost con imputación de valores perdidos

He comenzado utilizando mi estrategia básica de preprocesado en la que he imputado los valores perdidos de las variables categóricas. Los he reemplazado por una clase nueva que se llama "unknown".

La configuración básica que he empleado para el algoritmo CatBoost ha sido con los siguientes parámetros:

Parámetro	Valor
iterations	500
learning_rate	0.1
depth	6

Tabla 1. Configuración básica de CatBoost

Scores (RE, RPr, RPu) = 73067.06, 64143.62, 73104.77

2.2. LightGBM con imputación de valores perdidos

La configuración básica que he empleado para el algoritmo CatBoost ha sido con los siguientes parámetros:

Parámetro	Valor
num_leaves	31
max_depth	-1
learning_rate	0.1
n_estimators	500

Tabla 2. Configuración básica de LightGBM

Scores (RE, RPr, RPu) = 52614.92, 69962.21, 79023.3

Aunque el RMSE del entrenamiento haya sido menor, podemos ver que las puntuaciones de Kaggle son significativamente mayores que las del algoritmo anterior, esto se puede deber a que nos hemos sobreajustado a los datos de entrenamiento. Vamos a comprobar a continuación si podemos mejorar la predicción con este algoritmo.

2.3. LightGBM con retoque de parámetros e imputación de valores perdidos

Hemos modificado ligeramente los parámetros, haciendo que el modelo sea algo menos complejo y, por lo tanto, que le sea más difícil aprender los datos de entrenamiento, evitando así el sobreajuste. Hemos bajado el ratio de aprendizaje a la mitad y el número de estimadores de 500 a 200.

Parámetro	Valor
num_leaves	31
max_depth	-1
learning_rate	0.1 -> **0.05**
n_estimators	50 -> **200**

Tabla 3. Parámetros mejorados de LightGBM

Scores (RE, RPr, RPu) = 63738.24, 64986.97, 74073.79

Vemos que hemos mejorado significativamente las puntuaciones de Kaggle. Vamos a dejarlo así de momento y seguir probando otros algoritmos.

2.4. RandomForest con imputación de valores perdidos y codificación de etiquetas

Este algoritmo, además de utilizar datos con valores perdidos imputados, ha requerido convertir las variables categóricas a numéricas. Esto lo he hecho mediante *"Label Encoding"* y se ha realizado de la siguiente manera:

```
# Crear un diccionario para almacenar los LabelEncoders para cada columna categórica
label_encoders = {}

# Copiar X_train y X_val para no modificar los originales
X_train_encoded = X_train.copy()
X_val_encoded = X_val.copy()

# Aplicar LabelEncoder a cada columna categórica
for col in categorical_columns:
    le = LabelEncoder()
    X_train_encoded[col] = le.fit_transform(X_train_encoded[col])

    # Imputar valores no vistos en el conjunto de validación con 'unknown'
    X_val_encoded[col] = X_val_encoded[col].map(lambda s: 'unknown' if s not in le.classes_ else s)
    # Añadir 'unknown' a las clases del LabelEncoder
    le.classes_ = np.append(le.classes_, 'unknown')

    X_val_encoded[col] = le.transform(X_val_encoded[col])
    label_encoders[col] = le
```

Figura 6. Label Encoding aplicado a los datos de entrenamiento y test.

En este algoritmo tuve unos pequeños problemas iniciales ya que tardaba demasiado en ejecutar. Mis parámetros iniciales eran simplemente ajustar el número de estimadores a 500. No obstante, reduje posteriormente la complejidad del modelo para que aprendiera los datos más rápido. Sus parámetros quedaron de la siguiente manera:

Parámetro	Valor
min_samples_split	10
min_samples_leaf	5
max_depth	10
n_estimators	500

Tabla 4. Configuración básica de Random Forest.

Scores (RE, RPr, RPu) = 68773.66, 66169.23, 74769.86

Hemos vuelto a obtener unos resultados un poco malos, así que también voy a retocar ligeramente este modelo para intentar obtener mejores métricas.

2.5. RandomForest con retoque de parámetros, imputación de valores perdidos y codificación de etiquetas

Tras haber detectado de nuevo indicios de sobreajuste he optado por un modelo menos complejo con esta configuración:

Parámetro	Valor
min_samples_split	10
min_samples_leaf	5
max_depth	10 -> **6**
n_estimators	500 -> **200**

Tabla 5. Parámetros mejorados de Random Forest.

Scores (RE, RPr, RPu) = 73295.13, 64256.66, 73299.98

Podemos ver, de nuevo, como hemos mejorado las métricas de Kaggle para esta nueva parametrización del algoritmo. Vamos a dejarlo así de momento.

2.6. Red Neuronal con imputación de valores perdidos, codificación de etiquetas y normalización.

En adición a la normalización que le habiamos aplicado a Random Forest para pasar todas las variables a variables numéricas. He considerado importante normalizar todas las variables:

```
# Escalar variables numéricas
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_encoded)
X_val_scaled = scaler.transform(X_val_encoded)
X_test_scaled = scaler.transform(X_test_encoded)
```

Figura 7. Normalización aplicada a los datos.

Parámetro	Valor
Neuronas por capa	11 (entrada) -> 64 -> 32 -> 1 (salida)
Funciones de activación	Rectified Linear Units (ReLU)
Optimizador	Adaptive Moment Estimation (Adam)
Epochs	50
num_batches	20

Tabla 6. Configuración básica de la Red Neuronal.

La estructura de la NN y los parámetros utilizados se ven reflejados en la Tabla 6. Se trata de un MLP bastante sencillo con apenas dos capas ocultas. El rendimiento que nos ha proporcionado ha sido el siguiente:

Scores (RE, RPr, RPu) = 74799.44, 64762.56, 73773.30

Se trata de un rendimiento bastante similar al de anteriores algoritmos en cuanto a las puntuaciones de Kaggle. No obstante, podemos ver como el error de entrenamiento es bastante alto, lo cual nos puede llevar a intuir que esta red neuronal no es lo suficientemente potente para aprender bien los datos. Es por ello que vamos a usar una NN algo más potente más adelante.

2.7. Red Neuronal más compleja con imputación de valores perdidos, codificación de etiquetas y normalización.

Vamos a dejar el resto de parámetros igual. La única diferencia es que vamos a añadir una capa oculta más a la red neuronal con 128 neuronas de entrada, permitiéndole aprender con más robustez los datos de entrenamiento. Su estructura quedaría tal que así:

Parámetro	Valor
Neuronas por capa	11 (entrada) -> **128** -> 64 -> 32 -> 1 (salida)
Funciones de activación	Rectified Linear Units (ReLU)
Optimizador	Adaptive Moment Estimation (Adam)
Epochs	50
num_batches	20

Tabla 7. Red Neuronal mejorada.

Scores (RE, RPr, RPu) = 74527.41, 64351.14, 73381.18

Hemos mejorado ligeramente las métricas, por lo que nos quedaremos con este otro modelo superior. Se podrían mejorar más con estructuras más grandes aún, pero las Redes Neuronales son bastante costosas de entrenar y encontrar los parámetros óptimos podría requerir mucho tiempo de prueba y error.

2.8. CatBoost + LightGBM + RandomForest + RedNeuronal. Parámetros básicos.

En esta prueba he decidido hacer un ensamble de lo que llevo hasta ahora y combinar los resultados de los 4 modelos anteriores. Además, haré una comparativa de las métricas proporcionadas por este ensamble usando los modelos con sus parámetros originales y otro ensamble con los mejorados.

```python
# Definir los pesos para cada modelo
weights = {
    'catboost': 0.25,
    'lightgbm': 0.25,
    'randomforest': 0.25,
    'nn': 0.25
}

final_predictions_test = (
    weights['catboost'] * cat_predictions_test +
    weights['lightgbm'] * lgb_predictions_test +
    weights['randomforest'] * rf_predictions_test +
    weights['nn'] * nn_predictions_test
)

# Crear un DataFrame con las predicciones finales
submission = pd.DataFrame({
    'id': test_ids,
    'price': final_predictions_test
})

# Guardar las predicciones en un archivo CSV
submission.to_csv('submission_all2.csv', index=False)

final_predictions_train = (
    weights['catboost'] * cat_predictions_train +
    weights['lightgbm'] * lgb_predictions_train +
    weights['randomforest'] * rf_predictions_train +
    weights['nn'] * nn_predictions_train
)

# Calcular el error cuadrático medio en el conjunto de entrenamiento
rmse_train = np.sqrt(mean_squared_error(Y_train, final_predictions_train))
print(f'RMSE en el conjunto de entrenamiento: {rmse_train}')
```

Figura 8. Ensamble aplicado a los cuatro modelos anteriores.

Scores (RE, RPr, RPu) = 65837.68, 63944.42, 73032.4

Podemos ver que, incluso con los modelos con parámetros básicos, las métricas mejoran significativamente. Vamos a comprobar ahora qué rendimiento obtenemos con un ensamble de los modelos con parámetros mejorados.

2.9. CatBoost + LightGBM + RandomForest + RedNeuronal. Parámetros optimizados.

Vamos a comprobar ahora si hay algún cambio en las métricas de rendimiento de Kaggle cuando utilizamos los modelos con los parámetros retocados. La intuición nos dice que si los modelos individuales mejoraron su rendimiento, el de ensamble también lo hará:

Scores (RE, RPr, RPu) = 70391.35, 63509.34, 72634.00 ***(Por ahora la mejor métrica)***

2.10. XGBoost con imputación de valores perdidos

He decidido también incluir el algoritmo XGBoost, ya que el modelo anterior de ensamble ha proporcionado muy buenos resultados. No obstante, estudiemos su comportamiento individual primero.

Parámetro	Valor
max_depth	6
eta	0.1
subsample	0.8
colsample_bytree	0.8
num_boost_round	100
early_stopping_rounds	10

Tabla 8. Configuración básica XGBoost

Scores (RE, RPr, RPu) = 62170.89, 65110.02, 74064.78

Vemos que el rendimiento obtenido no es malo del todo pero mejorable. Vamos a retocar un poco estos parámetros para ver si conseguimos mejorarlo.

2.11. XGBoost con retoque de parámetros e imputación de valores perdidos

Parámetro	Valor
max_depth	6 -> **8**
eta	0.1 -> **0.05**
subsample	0.8 -> **0.9**
colsample_bytree	0.8 -> **0.9**
lambda (L2 reg)	**2**
alpha (L1 reg)	**1**
num_boost_round	100 -> **300**
early_stopping_rounds	10

Tabla 9. Configuración optimizada XGBoost.

Scores (RE, RPr, RPu) = 54838.49, 64976.33, 74122.52

Vemos que ha mejorado ligeramente pero no demasiado.

2.12. CatBoost + LightGBM + RandomForest + RedNeuronal + XGBoost con imputación de valores perdidos

Una vez ejecutamos XGBoost y comprobamos que no tiene un mal rendimiento del todo, vamos a añadirlo al ensamble anterior. También vamos a comparar el ensamble de los cinco modelos con sus parámetros básicos y sus parámetros optimizados.

```python
# Definir los pesos para cada modelo
weights = {
    'catboost': 0.2,
    'lightgbm': 0.2,
    'randomforest': 0.2,
    'xgboost': 0.2,
    'nn': 0.2
}

final_predictions_test = (
    weights['catboost'] * cat_predictions_test +
    weights['lightgbm'] * lgb_predictions_test +
    weights['randomforest'] * rf_predictions_test +
    weights['xgboost'] * xgb_predictions_test +
    weights['nn'] * nn_predictions_test
)

# Crear un DataFrame con las predicciones finales
submission = pd.DataFrame({
    'id': test_ids,
    'price': final_predictions_test
})

# Guardar las predicciones en un archivo CSV
submission.to_csv('submission_all_xgboost.csv', index=False)

final_predictions_train = (
    weights['catboost'] * cat_predictions_train +
    weights['lightgbm'] * lgb_predictions_train +
    weights['randomforest'] * rf_predictions_train +
    weights['xgboost'] * xgb_predictions_train +
    weights['nn'] * nn_predictions_train
)

# Calcular el error cuadrático medio en el conjunto de entrenamiento
rmse_train = np.sqrt(mean_squared_error(Y_train, final_predictions_train))
print(f'RMSE en el conjunto de entrenamiento: {rmse_train}')
```

Figura 9. Ensamblado de los 5 modelos estudiados hasta ahora.

Scores (RE, RPr, RPu) = 64777.78, 63569.58, 72693.20

2.13. CatBoost + LightGBM + RandomForest + RedNeuronal + XGBoost con imputación de valores perdidos, parámetros optimizados.

Scores (RE, RPr, RPu) = 66742.04, 63532.94, 72672.47

Tras estudiar qué rendimiento presentan los cinco modelos estudiados hasta ahora, con sus respectivos preprocesados y con los parámetros optimizados que hemos incluido en nuestras anteriores pruebas, apreciamos que los resultados mejoran algo pero no significativamente. Sin embargo, hablando en términos absolutos, es mejor que lo que teníamos antes y, por tanto, la mejor métrica hasta el momento.

2.14. CatBoost con valores perdidos y precio logarítmico

Además de haber estudiado estos casos anteriores y haber ido probando nuevos algoritmos he decidido añadir alguna capa más de preprocesamiento a los datos. Tras observar el rango de valores que la variable objetivo "price" estaba tomando, como hemos observado anteriormente en la *Figura 3.* donde mostramos esto mismo, decidí ajustar los valores de dicha variable y entrenar los modelos para que predigan los precios en escala logarítmica, resultando esto en valores menos extremos.

Para los algoritmos estudiados voy a comprobar el rendimiento aplicando este preprocesamiento y comparando los modelos con los parámetros básicos con los optimizados. Comenzamos con CatBoost y sus parámetros básicos, mostrados en la "*Tabla 1. Configuración básica de CatBoost".*

Scores (RE, RPr, RPu) = 44240.91, 64110.96, 73141.22

2.15. CatBoost con retoque de parámetros, valores perdidos y precio logarítmico

En el caso de CatBoost, no optimizamos los parámetros anteriormente así que vamos a retocarlos ahora. Bajaremos el número de iteraciones, así como el ratio de aprendizaje y la profundidad máxima, para tener un modelo más simple y que le cueste más aprender los datos. También añadiremos regularización y early stopping:

Parámetro	Valor
iterations	500 -> **100**
learning_rate	0.1 -> **0.01**
depth	6 -> **4**
l2_leaf_reg	10
early_stopping_rounds	10

Tabla 10. Parámetros optimizados de CatBoost.

Vamos a comprobar qué métricas nos proporciona este modelo y si mejora algo.

Scores (RE, RPr, RPu) = 33509.03, 68022.49, 76858.85.

Vemos que no solo no hemos mejorado los resultados, sino que los hemos empeorado bastante. Es por ello que vamos a seguir con los parámetros básicos en CatBoost

2.16. LightGBM con valores perdidos y precio logarítmico

Vamos a comprobar ahora cómo se comporta este algoritmo con dichos parámetros básicos y los optimizados más adelante. Primero con los básicos, referenciados en la "*Tabla 2. Configuración básica de LightGBM".*

Scores (RE, RPr, RPu) = *46151.69, 63909.89, 73094.28*

No están nada mal, ya que hemos bajado de 64000 en el RPr. No obstante, sigue sin superar a la prueba de ensamble anterior utilizando los 5 modelos. Aun así, ha mejorado bastante con respecto a la anterior prueba en la que utilizábamos LightGBM individualmente sin aplicar la escala logarítmica al precio.

2.17. LightGBM con retoque de parámetros, valores perdidos y precio logarítmico

Vamos a comprobar los resultados para LightGBM pero utilizando ahora los parámetros utilizados anteriormente, referenciados en la "*Tabla 3. Parámetros mejorados de LightGBM".*

Scores (RE, RPr, RPu) = *44520.31, 63948.08, 73073.09*

Vemos que las métricas han empeorado algo al hacer el modelo más simple. Vamos a comprobar qué pasaría si lo hiciéramos con un modelo más potente.

2.18. LightGBM más potente, valores perdidos y precio logarítmico

Parámetro	Valor
num_leaves	31
max_depth	-1
learning_rate	0.1 -> **0.2**
n_estimators	50 -> **1000**

Tabla 11. Modelo más complejo de LightGBM

Scores (RE, RPr, RPu) = *52991.35, 64267.43, 73193.85*

Vemos que, definitivamente, los parámetros básicos de LightGBM proporcionan un mejor rendimiento cuando aplicamos la escala logarítmica a la variable objetivo.

2.19. RandomForest con valores perdidos, codificación de etiquetas y precio logarítmico

Para simplificar un poco el estudio, he concluido que cuando estoy aplicando la escala logarítmica, utilizar los parámetros básicos nos ha ido proporcionando los mejores resultados. Es por ello que no voy a emplear los parámetros óptimos para este preprocesado. Los resultados obtenidos por Random Forest utilizando los parámetros referenciados por la "*Tabla 4. Configuración básica de Random Forest." son:*

Scores (RE, RPr, RPu) = *42664.26, 64705.81, 73803.64*

2.20. Red Neuronal con valores perdidos, codificación de etiquetas, normalización y precio logarítmico

Los resultados para la Red Neuronal con la configuración referenciada por la "*Tabla 6. Configuración básica de la Red Neuronal.*" son los siguientes:

Scores (RE, RPr, RPu) = *41893.69, 64973.48, 74028.74*

2.21. XGBoost con valores perdidos y precio logarítmico

Finalmente ejecutamos XGBoost con el preprocesado del precio logarítmico y comprobamos su rendimiento. Los parámetros usados están referenciados por la "*Tabla 8. Configuración básica XGBoost*"

Scores (RE, RPr, RPu) = *41893.69, 64973.48, 74028.74*

2.22. CatBoost + LightGBM + RandomForest + RedNeuronal + XGBoost con valores perdidos y precio logarítmico

Ya que anteriormente hice un modelo de ensamble con esos primeros 5 algoritmos, haré lo mismo pero aplicando este preprocesado. El ensamble se ha hecho de la misma manera, ponderando con los mismos pesos todos los modelos (es decir, un 20% para cada uno). Los resultados son los siguientes:

Scores (RE, RPr, RPu) = *45741.79, 64983.68, 74124.38*

2.23. H2O autoML (5 min)

A continuación, voy a probar una herramienta de automatización de aprendizaje automático, en la que vamos a construir un "metamodelo" que consiste en hacer "stacking" de distintos modelos. Esta herramienta entrena distintos modelos y va ajustando los parámetros de los mismos, además de aplicar los respectivos preprocesados necesarios para cada modelo.

En estas pruebas realizadas solamente he retocado un parámetro de este modelo que es el tiempo máximo de ejecución. He comenzado ejecutándolo 5 minutos:

Scores (RE, RPr, RPu) = *72655.33, 63256.00, 72418.05*

Esta herramienta, además de ser fácil de usar, parece ser bastante eficaz y conseguir buenas métricas. En concreto, para estas predicciones se han usado 11 modelos de "Gradient Boosting" y 5 Redes Neuronales, las cuales han sido "stackeadas".

El "stacking", al igual que el ensamble o boosting, combina distintos modelos para crear predicciones únicas a partir de lo aprendido por todos ellos. No obstante, se diferencia en que, además, se crea un "metamodelo" que aprende también sobre las predicciones generadas por los modelos individuales.

```python
import h2o
from h2o.automl import H2OAutoML
import pandas as pd

h2o.init()

train = h2o.import_file("train.csv")
test = h2o.import_file("test.csv")

y = "price"
x = train.columns
x.remove(y)

aml = H2OAutoML(max_runtime_secs=3000, seed=42)
aml.train(x=x, y=y, training_frame=train)

predictions = aml.leader.predict(test)

predictions_df = predictions.as_data_frame()
test_ids = test["id"].as_data_frame()

submission = pd.DataFrame({
    'id': test_ids['id'],
    'price': predictions_df['predict']
})

submission_path = 'submission_h2o.csv'
submission.to_csv(submission_path, index=False)
```

Figura 10. Código necesario para ejecutar H2O autoML.

Proporciono en la Figura 10. todo el código necesario para poder usar esta herramienta. Se puede apreciar que en apenas 20 líneas podemos tener configurado ya este algoritmo, lo que demuestra su sencillez, habiendo comprobado ya su eficacia, podemos concluir que esta es la herramienta más interesante que he probado en esta práctica.

2.24. H2O autoML (50 min)

Ya que en la prueba anterior solamente lo ejecuté 5 minutos, voy a probar a ejecutarlo durante un periodo más extenso de tiempo. Esta vez serán 50 minutos. Comprobremos pues si mejora el rendimiento.

Scores (RE, RPr, RPu) = *72593.07, 63210.22, 72369.60*

2.25. H2O autoML (5000 s)

Hemos visto anteriormente que ha vuelto a mejorar el rendimiento conforme ibamos aumentando el tiempo de ejecución. No obstante, la mejora no ha sido significativa, a pesar de haber entrenado el modelo durante un periodo de tiempo mayor. He realizado una última prueba con 5000 segundos en vez de 1000.

Scores (RE, RPr, RPu) = *72592.53, 63203.32, 72348.35*

2.26. CatBoost + LightGBM + RandomForest + RedNeuronal + XGBoost + H2O (5 min)

Aunque no haya optado por usar H2O durante más tiempo, si he considerado combinar estas predicciones con el resto de modelos de antes. Vamos a hacer otro ensamble de modelos con los estudiados anteriormente en esta práctica, con su configuración por defecto. En este caso, al saber que H2O nos ha proporcionado los mejores resultados, voy a hacer un ensamble con distintos pesos. En lugar de ponderar todos los modelos por igual, H2O irá 50 / 50 con el resto de modelos (los otros 5 tendrán un 10% de peso entonces).

Los resultados que obtenemos al juntar las predicciones de los cinco modelos anteriores con H2O son los siguientes:

Scores (RE, RPr, RPu) = 68716.39, 63246.27, 72405.55

Siguen siendo muy buenos resultados pero no mejoran a H2O de por sí solo. No obstante, hemos hecho el ensamble con el modelo de H2O ejecutado 5 minutos, que proporcionaba resultados ligeramente peores.

2.27. CatBoost + LightGBM + RandomForest + RedNeuronal + XGBoost + H2O (50min)

Probando ahora a ensamblar todos los modelos con el H2O ejecutado 50 minutos, también con un 10% de peso para los cinco algoritmos individuales y 50% para las predicciones generadas por H2O.

Scores (RE, RPr, RPu) = 68685.43, 63222.20, 72377.14

Vemos que logramos mejorar las métricas ya que el H2O entrenado 50 minutos tenía mejor resultado. No obstante, no es suficiente para igualar la puntuación de este mismo modelo de H2O.

En definitiva, tras haber hecho numerosas pruebas para este problema de regresión con coches usados, hemos podido ir comprobando la potencia que el ensamble de modelos alberga, especialmente esta técnica utilizada por H2O de “stacking”, en la que generamos predicciones de muchos modelos individuales y luego entrenamos sobre éstas para generar un “metamodelo”.

3. Esquema de las pruebas realizadas

A continuación, proporciono una tabla en la que resumo todas las pruebas realizadas en esta práctica, mostrando los preprocesados aplicados, algoritmos, parámetros de los algoritmos, fecha de subida a Kaggle y las puntuaciones RMSE de entrenamiento y las de test proporcionadas por Kaggle.

Preprocesado	Algoritmos	Parámetros	Fecha y hora de subida	Score train Kaggle (Private) Kaggle (Public)
Imputación de valores perdidos en variables categóricas a una nueva categoría "unknown"	CatBoost	iterations = 500 learning_rate = 0.1 depth = 6	28/12/2024 - 21:00	73067.06 64143.62 73104.77
	LightGBM	num_leaves = 31 max_depth = -1 learning_rate 0.1 n_estimators = 500	28/12/2024 - 21:00	52614.92 69962.21 79023.3
	LightGBM	num_leaves = 31 max_depth = -1 learning_rate 0.05 n_estimators = 200	30/12/2024 - 12:30	63738.24 64986.97 74073.79
Imputación de valores perdidos y codificación de etiquetas	RandomForest	min_samples_split = 10 min_samples_leaf = 5 max_depth = 10 n_estimators = 500	28/12/2024 - 21:00	68773.66 66169.23 74769.86
	RandomForest	min_samples_split = 10 min_samples_leaf = 5 max_depth = 6 n_estimators = 200	30/12/2024 - 12:45	73295.13 64256.66 73299.98
Imputación de valores perdidos, codificación de etiquetas y normalización.	Red Neuronal	11 - 64 - 32 - 1 ReLU epochs = 50 batch_size = 20	28/12/2024 - 21:00	74799.44 64762.56 73773.30
	Red Neuronal	11 - 128 - 64 - 32 - 1 ReLU epochs = 50 batch_size = 20	30/12/2024 - 13:00	74527.41 64351.14 73381.18
Preprocesados anteriores de de cada modelo	CatBoost + LightGBM + RandomForest + Red Neuronal	Parámetros básicos. Ensamble mediante media aritmética. (25% cada uno)	28/12/2024 - 21:00	65837.68 63944.42 73032.4
	CatBoost + LightGBM + RandomForest + Red Neuronal	Parámetros optimizados. Ensamble mediante media aritmética. (25% cada uno)	30/12/2024 - 13:30	70391.35 63509.34 72634.00
Imputación de valores perdidos	XGBoost	max_depth = 6 eta = 0.1 subsample = 0.8 colsample_bytree = 0.8 num_boost_round = 100	29/12/2024 - 16:00	62170.89 65110.02 74064.78

		early_stopping_rounds = 10		
	XGBoost	max_depth = 6 eta = 0.1 subsample = 0.8 colsample_bytree = 0.8 lambda = 2 alpha = 1 num_boost_round = 100 early_stopping_rounds = 10	30/12/2024 - 14:00	54838.49 64976.33 74122.52
Preprocesados anteriores de de cada modelo	CatBoost + LightGBM + RandomForest + Red Neuronal + XGBoost	Parámetros básicos. Ensamble mediante media aritmética. (20% cada uno)	29/12/2024 - 16:00	64777.78 63569.58 72693.20
	CatBoost + LightGBM + RandomForest + Red Neuronal + XGBoost	Parámetros optimizados. Ensamble mediante media aritmética. (20% cada uno)	30/12/2024 - 14:00	66742.04 63532.94 72672.47
Imputación de valores perdidos y variable "price" con escala logarítmica	CatBoost	iterations = 500 learning_rate = 0.1 depth = 6	29/12/2024 - 13:30	44240.91 64110.96 73141.22
	CatBoost	iterations = 100 learning_rate = 0.01 depth = 4 l2_leaf_reg = 10 early_stopping_rounds = 10	30/12/2024 - 18:00	33509.03 68022.49 76858.85
	LightGBM	num_leaves = 31 max_depth = -1 learning_rate 0.1 n_estimators = 500	29/12/2024 - 13:30	46151.69 63909.89 73094.28
	LightGBM	num_leaves = 31 max_depth = -1 learning_rate 0.05 n_estimators = 200	30/12/2024 - 18:10	44520.31 63948.08 73073.09
	LightGBM	num_leaves = 31 max_depth = -1 learning_rate 0.2 n_estimators = 1000	30/12/2024 - 18:20	52991.35 64267.43 73193.85
Imputación de valores perdidos, codificación de etiquetas y escala logarítmica.	RandomForest	min_samples_split = 10 min_samples_leaf = 5 max_depth = 10 n_estimators = 500	29/12/2024 - 13:30	42664.26 64705.81 73803.64
Imputación de valores perdidos, codificación de etiquetas, normalización y	Red Neuronal	64 - 32 - 1 ReLU epochs = 50 batch_size = 20	29/12/2024 - 13:30	41893.69 64973.48 74028.74

escala logarítmica.				
Imputación de valores perdidos, y escala logarítmica	XGBoost	max_depth = 6 eta = 0.1 subsample = 0.8 colsample_bytree = 0.8 num_boost_round = 100 early_stopping_rounds = 10	30/12/2024 - 18:30	45741.79 64983.68 74124.38
Preprocesados anteriores de cada modelo y escala logarítmica.	CatBoost + LightGBM + RandomForest + Red Neuronal + XGBoost	Parámetros básicos. Ensamble mediante media aritmética. (20% cada uno)	30/12/2024 - 18:40	44242.86 64199.50 73350.88
Sin preprocesamiento adicional. Sólo el de H2O	H2O AutoML stacked ensemble: 8 GBM models y 3 DeepLearning models. GLM metalearner. 5 minutos de ejecución	max_runtime_secs = 300	28/12/2024 - 23:00	72655.33 63256.00 72418.05
	H2O AutoML stacked ensemble. 11 GBM models y 5 DeepLearning models. GLM metalearner 50 minutos de ejecución	max_runtime_secs = 3000	29/12/2024 - 00:00	72593.07 63210.22 72369.60
	H2O. 11 GBM y 8 DeepLearning models. 5000s de ejecución.	**max_runtime_secs = 5000**	**31/12/2024 - 11:00**	**72592.53 63203.32 72348.35**
Preprocesados anteriores de de cada modelo	CatBoost + LightGBM + RandomForest + Red Neuronal + XGBoost + H2O autoML (5 min)	Ensamble de los 5 modelos (50%) y H2O (50%)	29/12/2024 - 17:00	68716.39 63246.27 72405.55
	CatBoost + LightGBM + RandomForest + Red Neuronal + XGBoost + H2O autoML (50 min)	Ensamble de los 5 modelos (50%) y H2O (50%)	29/12/2024 - 19:30	68685.43 63222.20 72377.14

Tabla 12. Esquema de todas las pruebas realizadas para el problema de regresión del precio de coches usados.

4. Bibliografía

- Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. (2018). CatBoost: unbiased boosting with categorical features. *Advances in Neural Information Processing Systems*, 31. Recuperado de https://arxiv.org/abs/1706.09516

- Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... & Liu, T. Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. *Advances in Neural Information Processing Systems*, 30. Recuperado de https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf

- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5-32. https://doi.org/10.1023/A:1010933404324

- Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61, 85-117. https://doi.org/10.1016/j.neunet.2014.09.003

- Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. En *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785-794). https://doi.org/10.1145/2939672.2939785

- LeDell, E., & Poirier, S. (2020). H2O AutoML: Scalable automatic machine learning. En *Proceedings of the AutoML Workshop at ICML*. https://www.automl.org/wp-content/uploads/2020/07/AutoML_2020_paper_61.pdf

- Analytics Vidhya. (2018, junio 28). *Comprehensive guide to ensemble learning models*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-for-ensemble-models/