Trabajo de clasificación binaria

Guillermo Díaz Aguado.
Abril del 2025

Breve aclaración del *.py* subido: Además de incluir el código, he añadido objeto de python que estoy intentando medio "desarrollar" para mí para poder limpiar bien las bases de datos en pocos pasos y también me sirve para aprender yo de programación, OOP, etc… (es el objeto llamado *Pyurification* (nombre feo, lo se))

Introducción del problema.

El enunciado del trabajo nos explica que nosotros debemos crear un grupo de modelos de clasificación basados en Árboles de clasificación que sea capaz de predecir, según las características de los vehículos en el mercado de segunda mano, si eran blancos o negros.

La base de datos proporcionada consta de **4340 observaciones** con las siguientes características:

	Traducción al castellano	Tipo de variable
Price	Precio de venta	Numérica
Levy	Cantidad de impuestos a pagar	Numérica
Manufacturer	Fabricante	Categórica
Prod. year	Año de producción	Numérica
Category	Categoría	Categórica
Leather interior	Interior de cuero	Dicotómica
Fuel type	Tipo de combustible	Categórica
Engine volume	Volumen del motor	Numérica
Mileage	Kilometraje	Numérica
Cilinders	Cilindros	Numérica
Gear box type	Tipo de caja de cambios	Dicotómica
Drive wheels	Ruedas motrices	Categórica
Wheels	Lugar del volante	Dicotómica
Airbags	Airbags	Categórica
Color	Color del coche	Dicotómica

Dentro de las observaciones tenemos etiquetada nuestra variable dependiente: "Color".

Analísis y depuración de la base de datos proporcionada.

Por lo general, todo el dataframe nos viene bastante bien depurado, salvo por alguna que otra columna que debemos hacer algún que otro ajuste:

- "Levy" tiene una alta cantidad de elementos nulos clasificados como "-". Explicaré las opciones sopesadas y la elección final en el siguiente apartado
- "Mileage" tiene al final de cada elemento el string "km", como todas las observaciones están en kilómetros, simplemente quitaremos dicho string y utilizaremos esta variable como una variable numérica.
- "Engine volume" tiene al final de alguna observación una string indicando si tiene "Turbo". Para no perder la información aquellos coches con turbo y sin turbo, he creado una variable dicotómica que indique la presencia de este valor.
- "Manufacturer", "Category", "Fuel type", "Drive wheels" son variables categóricas, como los árboles que vamos a utilizar son Árboles binarios de clasificación (debido a su fácil implementación y mejor rendimiento computacional), todas estas variables las transformaré en dicotómicas

Missing data

En nuestro dataset se presentan bastantes valores perdidos (variable "Levy"). Podemos actuar de tres formas distintas:

- Podríamos eliminar los datos que tienen elementos perdidos, pero esto puede conllevar una reducción del set de entrenamiento.

- De forma alternativa podemos imputar estos valores, de manera que para las variables categóricas usaremos el valor más repetido, y en las numéricas podemos usar la media o la mediana. También está la opción de buscar la columna más parecida e imputar los valores de esta columna, aunque se crearía algo de dependencia en estas dos columnas.
  También se pueden imputar después de crear un árbol base y en los nodos que "deberían" colocarse estas observaciones se imputan valores en función del recorrido que ha pasado la observación, esto puede crear overfitting.

- La forma que voy a usar (ya que creo que es la más adecuada para el caso de árboles de decisión) es la de imputar valores en estas observaciones para poder usar de manera usual esta columna y después genero una nueva variable independiente donde me muestre si tiene valor nulo o no. De esta forma me verifico que todos los datos se mantienen en mi dataset sin perder información, aunque sí que es verdad que la imputación de valores puede generar error. Al crear esta nueva variable podemos descubrir nuevas relaciones.

Atípicos

Para los valores atípicos de las variables, he decidido no realizar ningún cambio puesto que los árboles son muy robustos a la hora de entender las relaciones sin depender de los propios valores.

Normalización de las variables.

Tampoco es necesario la normalización de las variables, los árboles al no tratar en distancias ni varianzas no les afecta si una variable tiene el rango de 0-1 o de 0-1000. La normalización puede crear confusión a la hora de interpretar el árbol, el cuál es uno de los puntos más fuertes de este tipo de modelos.

Selección de variables.

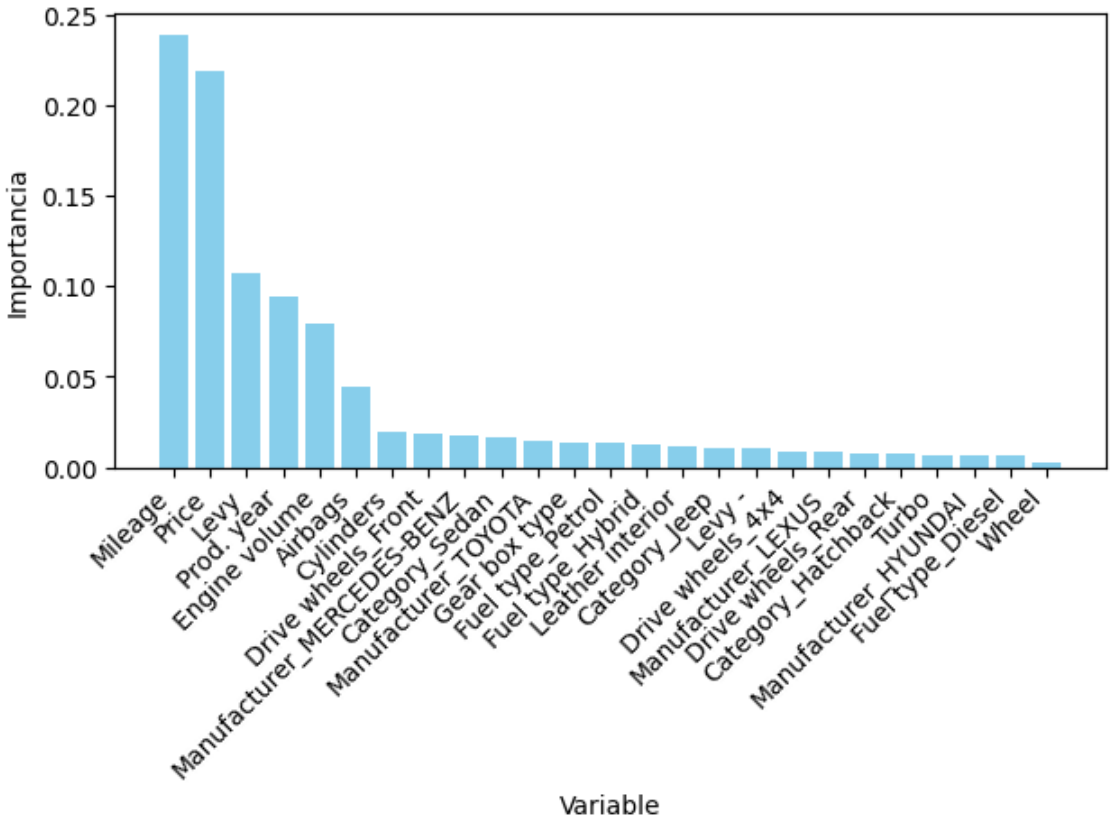
A la hora de realizar la discriminación de características, me he decantado por un método basado en los **Random forest.**

Ya que este tipo de modelo es bastante "inteligente" para lidiar con columnas que no aportan mucho o que solo meten ruido. El Random Forest evalúa cada variable y le da una puntuación según lo útil que haya sido para tomar decisiones en los árboles. Además, no se limita a relaciones simples o lineales como hacen otros algoritmos, sino que puede detectar conexiones complejas entre las variables, incluso si no son tan evidentes a simple vista. Esto lo hace ideal para identificar qué columnas realmente ayudan a predecir y cuáles se pueden dejar fuera sin miedo a perder precisión. Además, viendo de qué va la práctica me ha parecido lo mas adecuado.

También estaba la opción de usar un **Arbol de clasificación**, pero el Random Forest suele funcionar mejor que usar un solo árbol de clasificación. Un árbol por sí solo puede cambiar mucho si los datos cambian un poco, lo que puede llevarte a conclusiones poco fiables. En cambio, el Random Forest combina muchos árboles distintos y saca un promedio, lo que hace que los resultados sean más estables y precisos. También es mucho mejor captando relaciones más complejas entre las variables, sin quedarse solo con lo más evidente. Por eso, si quieres saber qué columnas de tu dataset realmente están ayudando a predecir bien, el Random Forest es una herramienta más sólida y práctica que usar un solo árbol.

```
from sklearn.ensemble import RandomForestClassifier

model_feature = RandomForestClassifier(random_state=42)
model_feature.fit(X, y)

importancia = pd.Series(model_feature.feature_importances_, index=X.columns)
importancia = importancia.sort_values(ascending=False)


importancia.plot(kind='barh')
plt.gca().invert_yaxis()
plt.title("Las variables más importantes")
plt.show()
```

Quedándonos así, una gráfica como la siguiente:



Según el *elbow method*, a partir de la variable “Airbags”, no existe una fuerte importancia de estas variables demás variables.

	Importancia	Importancia Acumulada
Mileage	0.239187	0.239187
Price	0.218889	0.458077
Levy	0.107612	0.565689
Prod. year	0.094007	0.659696
Engine volume	0.079431	0.739128
Airbags	0.044945	0.784073
Cylinders	0.019974	0.804046
Drive wheels_Front	0.018688	0.822734
Manufacturer_MERCEDES-BENZ	0.017710	0.840443
Category_Sedan	0.016805	0.857249
Manufacturer_TOYOTA	0.014212	0.871461
Gear box type	0.014010	0.885471
Fuel type_Petrol	0.013293	0.898764
Fuel type_Hybrid	0.012657	0.911421
Leather interior	0.011786	0.923208
Category_Jeep	0.011003	0.934210
Levy -	0.010819	0.945029
Drive wheels_4x4	0.008618	0.953647
Manufacturer_LEXUS	0.008334	0.961981
Drive wheels_Rear	0.008036	0.970017
Category_Hatchback	0.007238	0.977255
Turbo	0.006858	0.984113
Manufacturer_HYUNDAI	0.006748	0.990861
Fuel type_Diesel	0.006575	0.997437

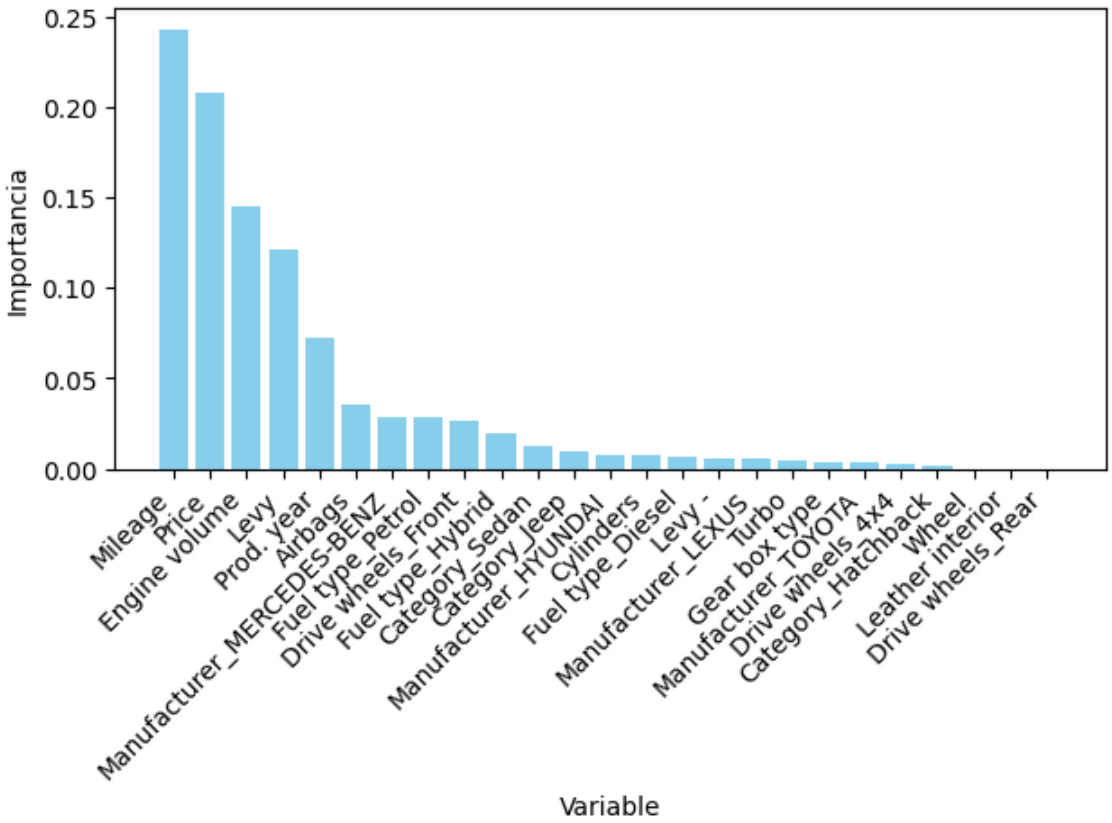
El problema es que la importancia acumulada de estas variables se quedaría en un 78%, el cual no es mal número pero creo que se perderá gran parte de la precisión de entrenamiento, finalmente decidí usar las 12 primeras variables, que acumulan un 88%.

La selección de variables en los árboles de clasificación no es estrictamente necesaria para que el modelo funcione, ya que estos algoritmos pueden manejar grandes cantidades de variables y tienden a ignorar automáticamente las menos relevantes.

He decidido mantener estas variables para tos modelos de **Random forest** y **XGBoost** , puesto que estos requieren mucha más capacidad computacional que los **árboles de clasificación**.

Árboles de clasificación.

Al momento de construir el modelo con árboles de decisión, lo primero que haré será entrenar un árbol sencillo, de baja profundidad, con el objetivo de identificar cuáles son las variables más relevantes del conjunto de datos. Este paso inicial me servirá para tener una idea clara de qué características aportan más al modelo. Luego, pasaré a realizar el tuneo de hiperparámetros usando todas las variables, ya que por lo que se verá en la siguiente gráfica las variables menos importantes se siguen quedando como menos importantes.



Como se puede ver, las variables más importantes en el árbol de decisión se mantienen más o menos en los primeros puestos, en comparación con el análisis hecho con el modelo de Random Forest. Lo que sí cambia es el nivel de importancia que se le asigna a cada una. Esto se debe a que Random Forest calcula la importancia como un promedio basado en muchos árboles construidos sobre distintos subconjuntos de los datos, lo que da un resultado más estable. En cambio, un árbol de decisión trabaja con todos los datos directamente en una sola estructura, por lo que las importancias pueden variar más.

Los cuatro métodos de validación de la bondad estudiados en clase:

- “accuracy” (Exactitud): mide la proporción de predicciones correctas sobre el total de muestras.
- “precision_macro” (Precisión macro-promediada): mide el promedio no ponderado de la precisión por clase.
- recall_macro (Exhaustividad macro-promediada): mide el promedio no ponderado del recall (sensibilidad) por clase.
- f1_macro: mide el promedio no ponderado del F1-score por clase.

Árbol de clasificación con todas las variables.

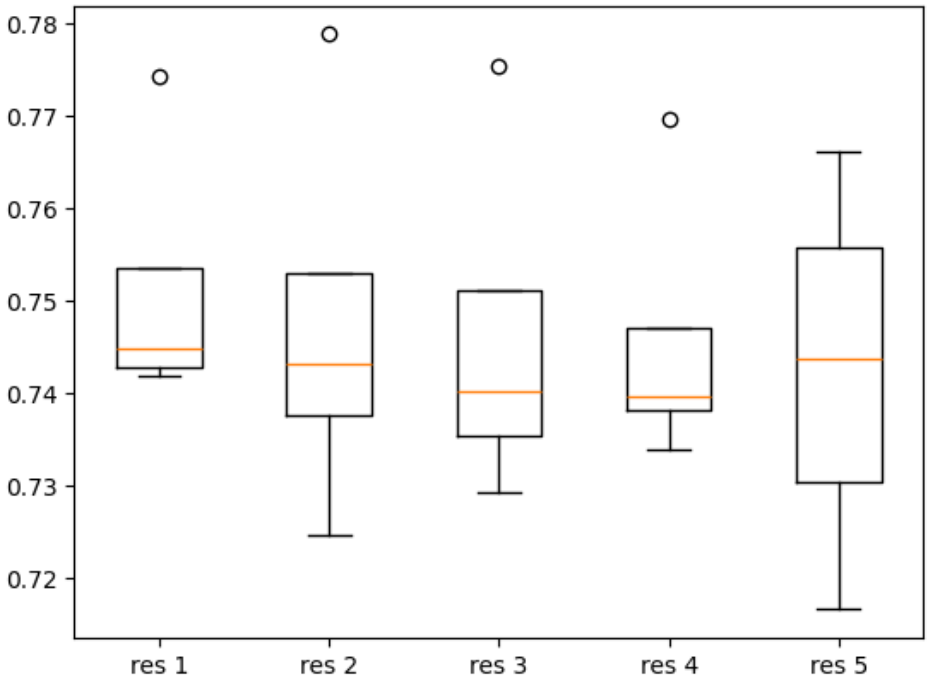
He llevado a cabo una *grid search* utilizando los siguientes parámetros:

```python
# Tuneo
params={
    "max_depth":[2, 3, 5, 10, 20],
    "min_samples_split":[5, 10, 20, 50, 100],
    "criterion": ["gini", "entropy"]
}
scoring_metrics = ["accuracy", "precision_macro", "recall_macro", "f1_macro"]


#Cross Validation
decision_tree = DecisionTreeClassifier()
grid_search = GridSearchCV(estimator=decision_tree,
                           param_grid=params,
                           cv=4,
                           scoring=scoring_metrics,
                           refit="accuracy")
grid_search.fit(X_train, y_train)


# Obtenemos los resultados del grid search.
results = pd.DataFrame(grid_search.cv_results_)

results.sort_values(by="mean_test_accuracy", ascending=False, inplace=True)
results[["params", "mean_test_accuracy", "mean_test_precision_macro", "mean_test_recall_macro", "mean_test_f1_macro"]].head(5)
```

Como resultado nos salen los siguientes posibles arboles como candidatos:

	params	mean_test_accuracy	mean_test_precision_macro	mean_test_recall_macro	mean_test_f1_macro
45	{'criterion': 'entropy', 'max_depth': 20, 'min_samples_split': 5}	0.751440	0.751050	0.750560	0.750344
40	{'criterion': 'entropy', 'max_depth': 10, 'min_samples_split': 5}	0.747408	0.746851	0.746344	0.746277
46	{'criterion': 'entropy', 'max_depth': 20, 'min_samples_split': 10}	0.746256	0.745768	0.745645	0.745294
15	{'criterion': 'gini', 'max_depth': 10, 'min_samples_split': 5}	0.745680	0.744832	0.745046	0.744831
20	{'criterion': 'gini', 'max_depth': 20, 'min_samples_split': 5}	0.742512	0.741823	0.740641	0.740917

Por lo que parece no se llevan mucha diferencia entre todos ellos, pero no nos podemos fijar solo en el valor medio de precisión (o los demás métodos de validación), para ello vamos a graficar los diagramas de *caja y bigotes* para estos modelos:



Y, por lo que se puede ver en este gráfico, el resultado 1 del diagrama de caja y bigotes parece ser el mejor, ya que tiene la media más alta y, además, es uno de los que presenta menos

variación entre los valores. Así que nos quedaremos con el “res 1”, el cual utiliza los siguientes parámetros:

```
DecisionTreeClassifier(criterion='entropy', max_depth=20,
                       min_samples_split=5)
```

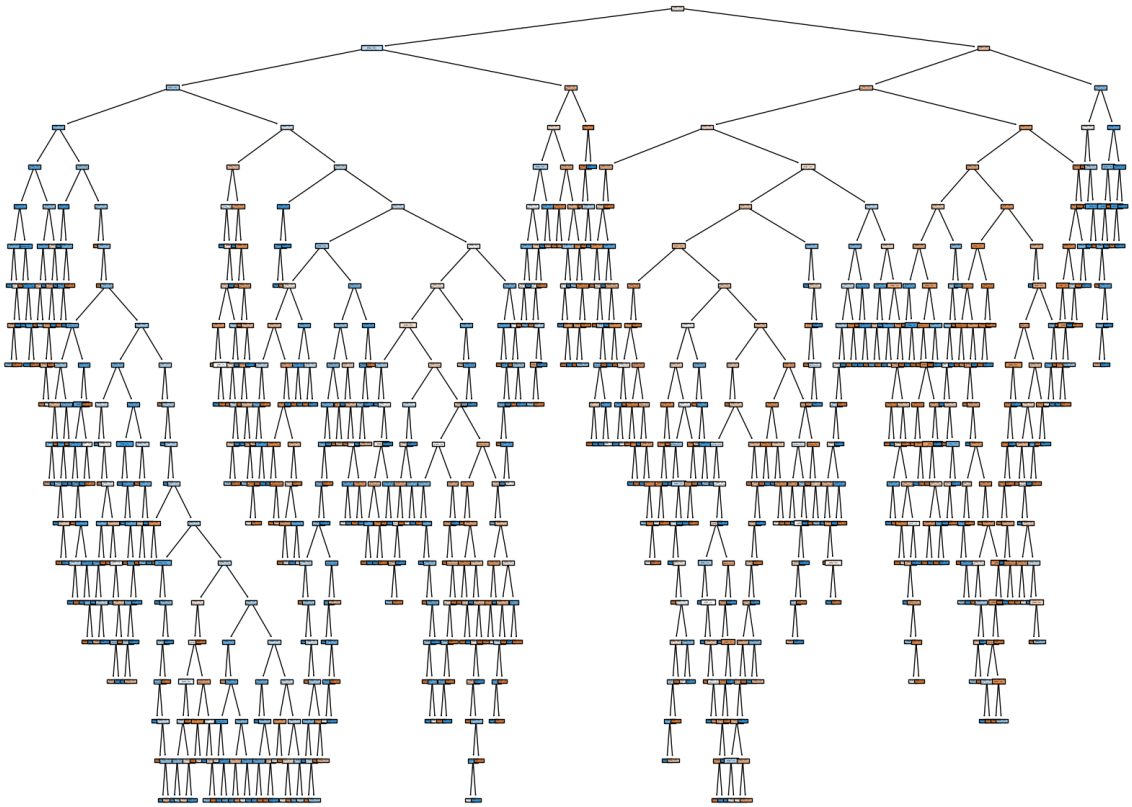
Con las siguientes medidas de desempeño:

Medidas de desempeño

	precision	recall	f1-score	support
0	0.94	0.95	0.95	1845
1	0.94	0.94	0.94	1627
accuracy			0.94	3472
macro avg	0.94	0.94	0.94	3472
weighted avg	0.94	0.94	0.94	3472

Este modelo tiene un rendimiento muy bueno y equilibrado en ambas clases, con un accuracy del 94%, y un f1-score de 0.94 en ambas clases, lo cual indica que tanto precisión como recall están bien compensados.

La gráfica del árbol se nos queda así, la cual cuesta muchísimo interpretar debido a su alta profundidad. Temiendo a que mi árbol se quede muy **sobreajustado** voy a realizar un tuneo con menores profundidades.



El nuevo tuneo se queda así:

```
# Tuneo
params={
    "max_depth":[2, 3, 5, 7],
    "min_samples_split":[5, 10, 20, 50, 100],
    "criterion": ["gini", "entropy"]
}
scoring_metrics = ["accuracy", "precision_macro", "recall_macro", "f1_macro"]


#Cross Validation
decision_tree = DecisionTreeClassifier()
grid_search = GridSearchCV(estimator=decision_tree,
                           param_grid=params,
                           cv=4,
                           scoring=scoring_metrics,
                           refit="accuracy")
grid_search.fit(X_train, y_train)


# Obtenemos los resultados del grid search.
results = pd.DataFrame(grid_search.cv_results_)

results.sort_values(by="mean_test_accuracy", ascending=False, inplace=True)
results[["params", "mean_test_accuracy", "mean_test_precision_macro", "mean_test_recall_macro", "mean_test_f1_macro"]].head(5)
```

El cual tiene menos precisión y mas variabilidad, pero es más generalizado.

El modelo sería:

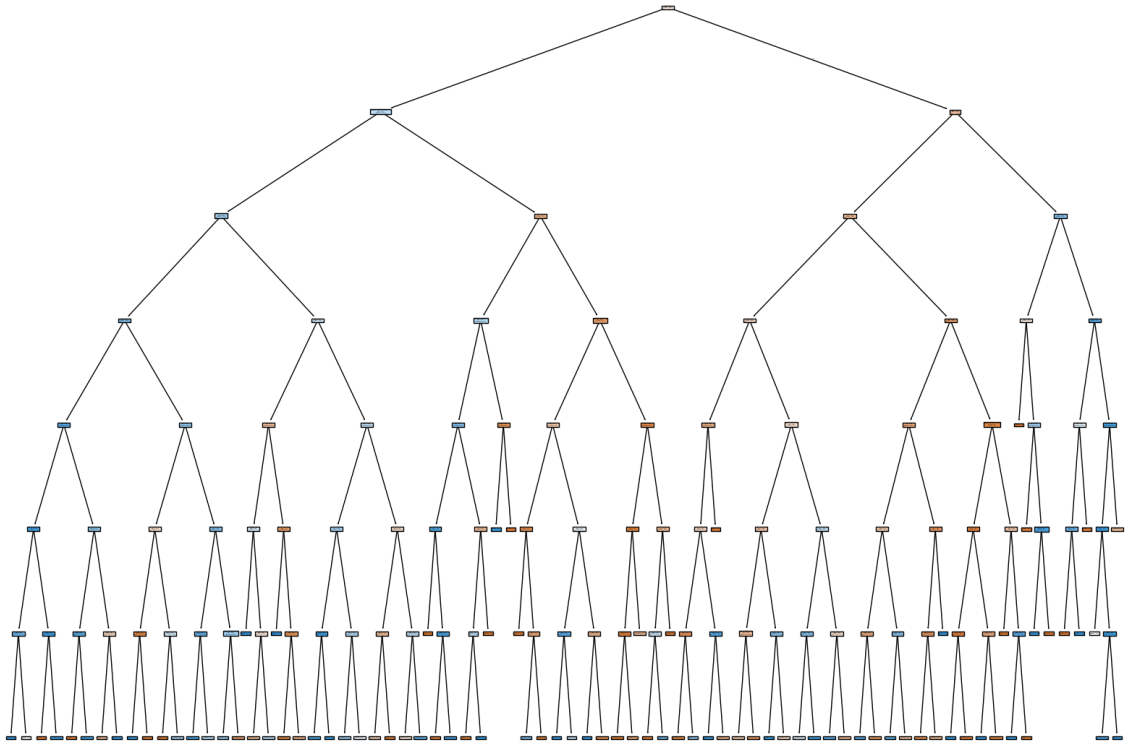
```
DecisionTreeClassifier(max_depth=7, min_samples_split=5)
```

Y se nos quedaría unas medidas de desempeño tal que:

Medidas de desempeño

	precision	recall	f1-score	support
0	0.80	0.77	0.78	1845
1	0.75	0.78	0.76	1627
accuracy			0.77	3472
macro avg	0.77	0.77	0.77	3472
weighted avg	0.78	0.77	0.77	3472

Para el nuevo modelo, se nos queda un árbol tal que así:

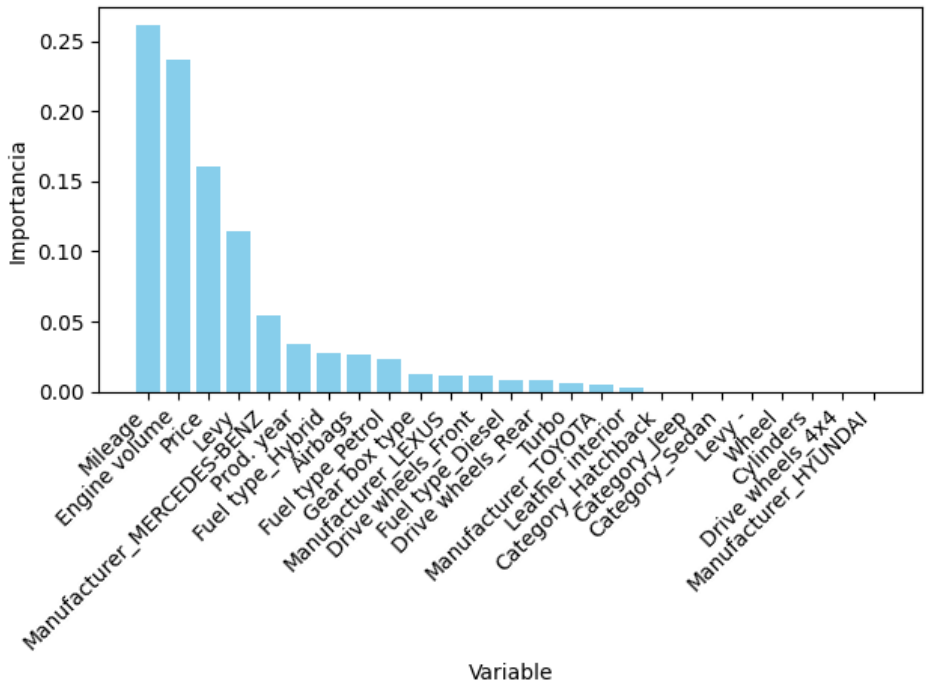


Donde las hojas finales tienen colores que indican que están bien separadas. Me podría meter mas de lleno en cada una de las hojas pero por lo que se ve a simple vista, hace un trabajo decente, con un accuracy de alrededor de 72%.

Sus reglas en formato texto son unas 200 lineas de texto, así que mostraré unas cuantas:

```
|--- Engine volume <= 2.30
|   |--- Manufacturer_MERCEDES-BENZ <= 0.50
|   |   |--- Fuel type_Petrol <= 0.50
|   |   |   |--- Mileage <= 52351.50
|   |   |   |   |--- Prod. year <= 2017.50
|   |   |   |   |   |--- Prod. year <= 2011.50
|   |   |   |   |   |   |--- Price <= 10636.00
|   |   |   |   |   |   |   |--- weights: [0.00, 12.00] class: 1
|   |   |   |   |   |   |--- Price >  10636.00
|   |   |   |   |   |   |   |--- weights: [5.00, 5.00] class: 0
|   |   |   |   |   |--- Prod. year >  2011.50
|   |   |   |   |   |   |--- Airbags <= 1.00
|   |   |   |   |   |   |   |--- weights: [1.00, 0.00] class: 0
|   |   |   |   |   |   |--- Airbags >  1.00
|   |   |   |   |   |   |   |--- weights: [1.00, 67.00] class: 1
|   |   |   |   |--- Prod. year >  2017.50
|   |   |   |   |   |--- Mileage <= 19260.50
|   |   |   |   |   |   |--- Mileage <= 60.00
|   |   |   |   |   |   |   |--- weights: [2.00, 0.00] class: 0
|   |   |   |   |   |   |--- Mileage >  60.00
|   |   |   |   |   |   |   |--- weights: [1.00, 16.00] class: 1
|   |   |   |   |   |--- Mileage >  19260.50
|   |   |   |   |   |   |--- Mileage <= 47846.00
|   |   |   |   |   |   |   |--- weights: [7.00, 3.00] class: 0
|   |   |   |   |   |   |--- Mileage >  47846.00
|   |   |   |   |   |   |   |--- weights: [0.00, 2.00] class: 1
|   |   |   |--- Mileage >  52351.50
|   |   |   |   |--- Mileage <= 71885.50
|   |   |   |   |   |--- Price <= 12157.00
|   |   |   |   |   |   |--- Levy <= 392.00
|   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |--- Levy >  392.00
|   |   |   |   |   |   |   |--- weights: [13.00, 0.00] class: 0
|   |   |   |   |--- Price >  12157.00
|   |   |   |   |   |   |--- Mileage <= 56390.50
|   |   |   |   |   |   |   |--- weights: [6.00, 0.00] class: 0
```

Donde se puede verificar ahora si la importancia de “Engine volume”, “Mileage”, “Levy”(que aunque no estén arriba, si que están en los nodos mas cercanos a las hojas)



Random Forest

Como ya he explicado antes, por temas computacionales tengo que disminuir la cantidad de variables con las que entrenaré el modelo, puesto que mi ordenador no da para más. En el primer apartado ya hicimos una selección de variable y usaremos esas explicadas.

```
    X = X[variables_importantes]

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=random_state)
    print(f"La frecuencia de cada clase en train es: \n{y_train.value_counts(normalize=True)}")
    print(f"La frecuencia de cada clase en test es: \n{y_test.value_counts(normalize=True)}")
```

```
La frecuencia de cada clase en train es:
Color
0    0.531394
1    0.468606
Name: proportion, dtype: float64
La frecuencia de cada clase en test es:
Color
0    0.5553
1    0.4447
Name: proportion, dtype: float64
```

El tuneo junto con el *grid search* realizado es:

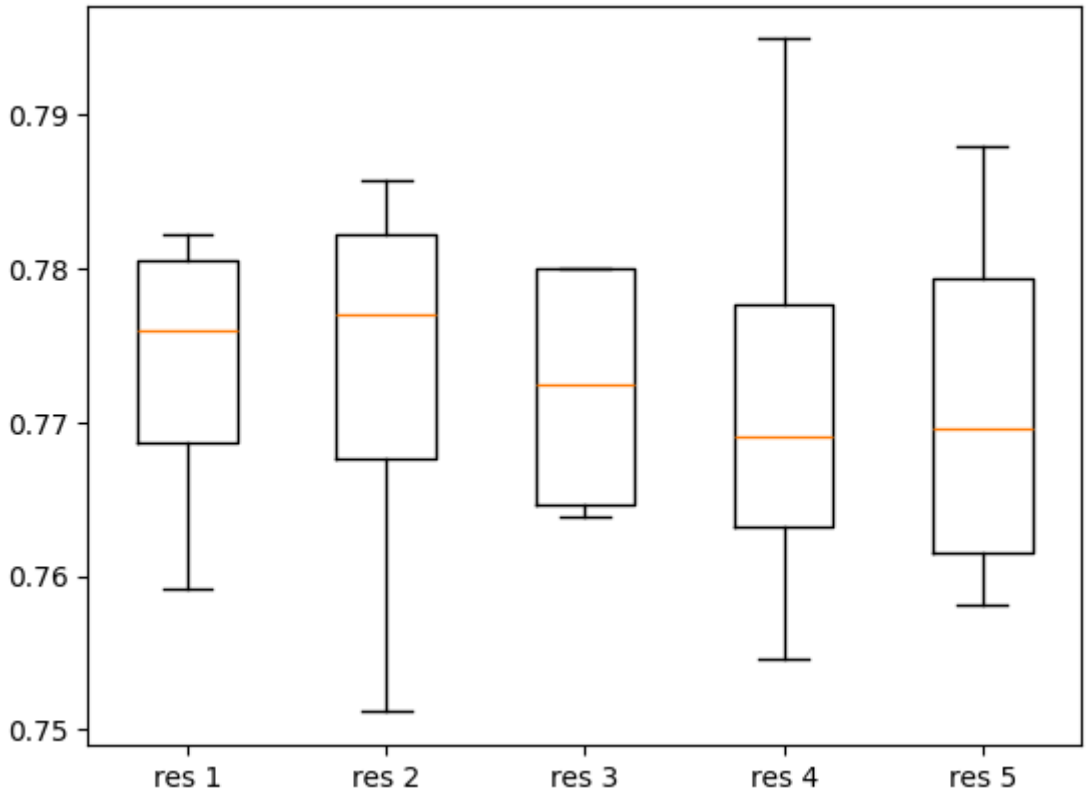
```
RF_model = RandomForestClassifier()
# Tuneo de los hiperparametros
params = {
    'n_estimators' : [50, 70, 80, 100],
    'max_depth': [2, 3, 5, 10, 20],
    'bootstrap': [True, False],
    'min_samples_leaf' : [3,10,30],
    'min_samples_split': [5, 10, 20, 50, 100],
    'criterion': ["gini", "entropy"]
}

scoring_metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
# cv = crossvalidation
grid_search_RF = GridSearchCV(estimator=RF_model,
                              param_grid=params,
                              cv=4,
                              scoring = scoring_metrics,
                              refit='accuracy')
grid_search_RF.fit(X_train, y_train)
```

He probado a realizar alguno con n_estimators de 150, el cual daba mejores resultados, pero seguramente estuviera sobreajustado y además me duro 1h y algo (por lo que he decidido no repetirlo). También he de indicar que hice pruebas usando todas las variables, lo cual hizo que tuviera que esperar muchísimo, y que además tiene un **accuracy** parecido.

Los mejores modelos que han salido son los siguientes:

	params	mean_test_accuracy	mean_test_precision_macro	mean_test_recall_macro	mean_test_f1_macro
1143	{'bootstrap': False, 'criterion': 'entropy', 'max_depth': 20, 'min_samples_leaf': 3, 'min_samples_split': 5, 'n_estimators': 100}	0.773329	0.772663	0.772005	0.772183
841	{'bootstrap': False, 'criterion': 'gini', 'max_depth': 20, 'min_samples_leaf': 3, 'min_samples_split': 5, 'n_estimators': 70}	0.772753	0.772139	0.771421	0.771565
842	{'bootstrap': False, 'criterion': 'gini', 'max_depth': 20, 'min_samples_leaf': 3, 'min_samples_split': 5, 'n_estimators': 80}	0.772177	0.771826	0.770738	0.770918
1142	{'bootstrap': False, 'criterion': 'entropy', 'max_depth': 20, 'min_samples_leaf': 3, 'min_samples_split': 5, 'n_estimators': 80}	0.771889	0.771251	0.770615	0.770767
1146	{'bootstrap': False, 'criterion': 'entropy', 'max_depth': 20, 'min_samples_leaf': 3, 'min_samples_split': 10, 'n_estimators': 80}	0.771313	0.770696	0.769928	0.770145

Que al graficarlo en una boxplot, nos muestra que también el mejor modelo por accuracy, es el mejor de todos:



Quedándonos pues un modelo tal que :

```
RandomForestClassifier(bootstrap=False, criterion='entropy',
    max_depth=20,min_samples_leaf=3, min_samples_split=5)
```

El cual tiene unas muy buenas medidas de desempeño.

Medidas de desempeño

	precision	recall	f1-score	support
0	0.97	0.95	0.96	1845
1	0.95	0.97	0.96	1627
accuracy			0.96	3472
macro avg	0.96	0.96	0.96	3472
weighted avg	0.96	0.96	0.96	3472

Estas medidas de desempeño son bastante altas, seguramente estemos pecando de sobreajuste como siempre, pero al haber realizado tantas pruebas, podremos confiar en estos resultados.

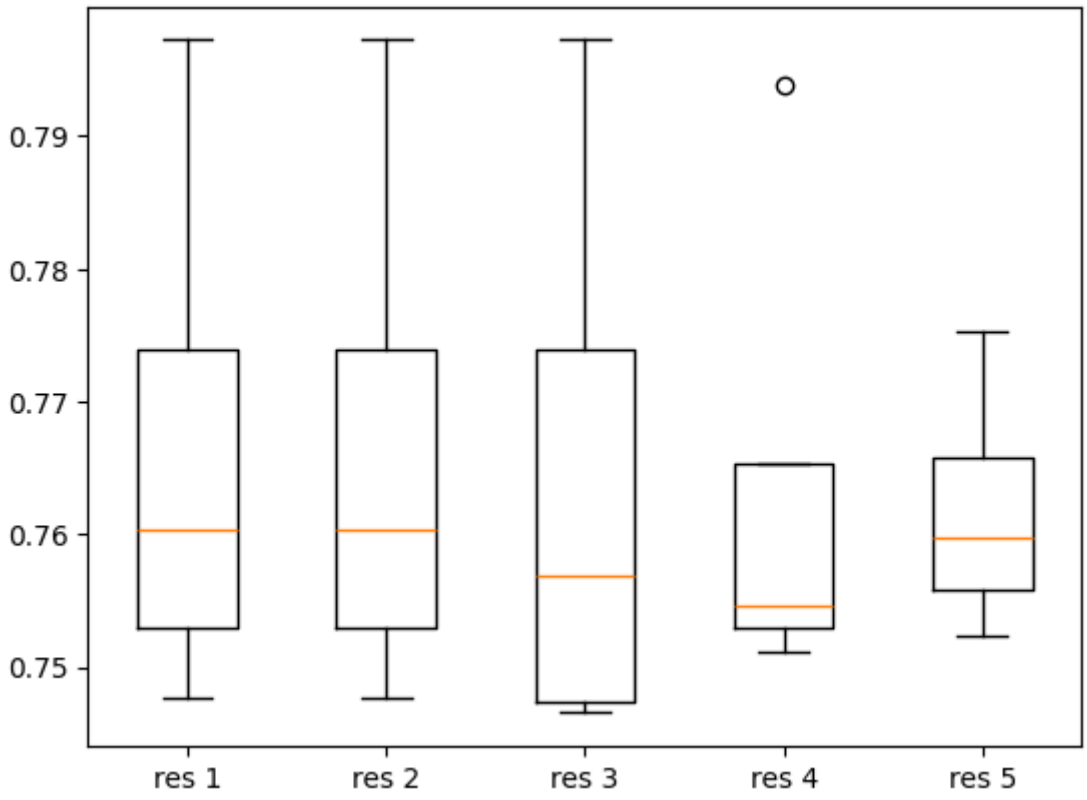
XGBoost

Seguimos usando las variables mas importantes sacadas del Random Forest del inicio del proyecto. Para este grid search he usado los siguientes parámetros:

```
xgb = XGBClassifier()

params = {
    'n_estimators': [100,200,300],
    'eta' : [0.1,0.4,0.7],
    'gamma' : [0.1,0.5,1],
    'max_depth': [5, 10]
}

scoring_metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']

grid_search_XGB = GridSearchCV(estimator=xgb,
                               param_grid=params,
                               cv=4, scoring = scoring_metrics, refit='accuracy')
grid_search_XGB.fit(X_train, y_train)
```

Dandonos como resultado los siguientes valores:

	params	mean_test_accuracy	mean_test_precision_macro	mean_test_recall_macro	mean_test_f1_macro
4	{'eta': 0.1, 'gamma': 0.1, 'max_depth': 10, 'n_estimators': 200}	0.766417	0.765784	0.765755	0.765560
5	{'eta': 0.1, 'gamma': 0.1, 'max_depth': 10, 'n_estimators': 300}	0.766417	0.765784	0.765755	0.765560
3	{'eta': 0.1, 'gamma': 0.1, 'max_depth': 10, 'n_estimators': 100}	0.764401	0.763793	0.763748	0.763530
1	{'eta': 0.1, 'gamma': 0.1, 'max_depth': 5, 'n_estimators': 200}	0.763537	0.762903	0.763045	0.762758
21	{'eta': 0.4, 'gamma': 0.1, 'max_depth': 10, 'n_estimators': 100}	0.761809	0.761125	0.760873	0.760824



Viendo la gráfica de *caja y bigotes* posiblemente el mejor modelo es el del resultado 5 puesto que es el que tiene la menor variabilidad y mayor accuracy. También he sopesado el resultado 4 pero tal vez ese valor atípico le tenga mas importancia de lo que parece.

El modelo realizado será:

```
XGBClassifier(eta=0.4, gamma=0.1, max_depth=10, n_estimators=100, ...)
```

El caul tiene unas medidas de desempeño casi perfectas:

Medidas de desempeño

	precision	recall	f1-score	support
0	0.99	0.99	0.99	1845
1	0.99	0.99	0.99	1627
accuracy			0.99	3472
macro avg	0.99	0.99	0.99	3472
weighted avg	0.99	0.99	0.99	3472

Desarrolla un proceso comparativo completo de los modelos

Compararé estos modelos en función de:

- Accuracy
- Explicatividad
- Desempeño computacional
- Robustez

Precisión (Accuracy)

- **Árbol de Decisión**: ofrece una precisión razonable, tiende a sobreajustarse si no se controla su crecimiento, es por ello que con 20 de profundidad, este modelo daba grandes valores de precisión.
- **Random Forest**: mejora significativamente la precisión al reducir la varianza mediante el promedio de múltiples árboles, dando así un resultado mejor para casos predictivos.
- **XGBoost**: generalmente logra la **mejor precisión**, ya que optimiza el aprendizaje de errores paso a paso, aunque puede requerir más ajuste de parámetros.

Explicatividad

- **Árbol de Decisión**: altamente interpretable, se puede visualizar como un gráfico y comprender fácilmente.
- **Random Forest**: menos interpretable debido a la cantidad de árboles involucrados, aunque permite obtener **importancia de características**, que es por lo que nosotros hemos realizado la elección de variables con este método.
- **XGBoost**: Es la menos intuitiva de todos.

Desempeño Computacional

- **Árbol de Decisión**: muy rápido de entrenar y predecir.
- **Random Forest**: es el que más tiempo me ha requerido, puede que sea por el n_estimators pero y porque mi ordenador no tiene gráfica, pero es sin duda el mas costoso.
- **XGBoost**: Se podría decir que para lo bien que lo hace, tarda relativamente poco (en el orden de los 10 min)

Robustez

- **Árbol de Decisión**: no es robusto frente a ruido o datos desbalanceados.
- **Random Forest**: mucho más robusto, gracias al promedio de múltiples modelos.
- **XGBoost**: extremadamente robusto y preciso, aunque puede sobreajustar si no se controla bien.

Conclusión

Después de comparar los tres modelos: Árbol de Decisión, Random Forest y XGBoos. Se pueden sacar conclusiones claras sobre en qué destaca cada uno.

En primer lugar, XGBoost es el modelo que ofrece la mejor capacidad de predicción. XGBoost suele lograr una precisión más alta, especialmente en conjuntos de datos complejos o con ruido.

Por otro lado, el modelo más explicativo es el Árbol de Decisión tradicional. Esto se debe a que su estructura es muy fácil de entender: cada decisión está representada por una división clara basada en una variable, y se puede visualizar en forma de árbol.

Finalmente, el mejor modelo para analizar la importancia de las variables (selección de características) es Random Forest.