UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE ESTUDIOS ESTADÍSTICOS

TAREA EVALUABLE

MÓDULO: MACHINE LEARNING - CLASIFICACIÓN BINARIA

Jorge González Perea 51553561G

Máster en Big Data, Data Science & Inteligencia Artificial

Curso académico 2024-2025

Índice

1. Introducción.

En esta práctica se va a tratar una base de datos con 4340 registros de información perteneciente a distintos vehículos de segunda mano. Las variables disponibles son las siguientes:

- **Price**: variable numérica. Precio del vehículo.
- **Levy**: variable numérica. Cantidad de impuestos a pagar.
- **Manufacturer**: variable categórica. Empresa fabricante del vehículo.
- **Prod. year**: variable numérica. Año de fabricación.
- **Category**: variable categórica. Tipo de vehículo.
- **Leather interior**: Variable categórica. Indica si el interior es de cuero o no.
- **Fuel type**: variable categórica. Indica el tipo de combustible del vehículo.
- **Engine volume**: variable categórica. Volumen de desplazamiento de los cilindros del motor. También indica si el motor tiene acoplado un turbo.
- **Mileage**: variable categórica. Kilometraje del vehículo.
- **Cylinders**: variable numérica. Número de cilindros del bloque.
- **Gear box Type**: variable categórica. Tipo de transmisión del vehículo.
- **Drive wheels**: variable categórica. Indica las ruedas que traccionan.
- **Wheel**: variable categórica. Indica el lado en el que el volante está situado.
- **Color**: variable categórica. Color del vehículo.
- **Airbags**: variable numérica. Número de airbags en el vehículo.

El objetivo es emplear las 14 variables restantes para poder predecir la variable objetivo **Color**, de forma que se pueda saber si conviene pintar el coche de color blanco o no. Para ello se va a llevar a cabo un análisis y una limpieza de la base de datos. Posteriormente se van a desarrollar distintos modelos de árboles de regresión que ayude con la toma de decisiones para la variable objetivo: un árbol simple, un *Random Forest* y *XGBoost*. Por último, estos modelos serán comparados con el fin de justificar cuál de ellos es mejor para esta BBDD. Todo ello será llevado a cabo con *Python* y con las librerías de funciones que se pueden encontrar en la sección 7 (anexo).

La metodología para el desarrollo de los modelos es la siguiente en los tres casos:

- Primero se crea un clasificador base con parámetros obtenidos de forma arbitraria y que minimicen el sobreajuste sin pasarse. Este estimador ya se puede usar como modelo.
- El segundo paso consiste en hacer una búsqueda de los mejores parámetros mediante la función *GridSearchCV* sobre el estimador base del paso anterior. El mejor modelo se obtiene con el método *grid_search.best_estimator*.
- Otro modelo prometedor se puede encontrar haciendo una búsqueda manual mediante *boxplots* en los resultados de *GridSearchCV*.
- La cuarta y última manera empleada para buscar parámetros óptimos es visualizado la dependencia de la precisión de los modelos en función de estos. Para ello se crean gráficas manualmente y se eligen los mejores parámetros.

2. Sección 2 - Análisis y depuración de los datos.

En primer lugar, para poder trabajar con la base de datos de forma más eficiente es imprescindible familiarizarse con ella y llevar a cabo un análisis descriptivo de las variables. Tras este primer análisis se procede con una limpieza de los datos, ya que puede haber registros *missing* o valores anómalos. Por último se lleva a cabo la transformación de variables (*feature engineering*) de forma que la tabla de datos se ha simplificado lo máximo posible mediante codificaciones y eliminación de información redundante.

Con el siguiente código se lee el archivo de datos y se muestran los formatos de las distintas variables:

```
datos = pd.read_excel("datos_tarea25.xlsx")
print(datos.dtypes)
```

Listing 1: Lectura de datos y análisis incial.

El método **datos.dtypes** proporciona los formatos de las variables del *DataFrame*. En este caso se observan algunas incongruencias:

```
Price              int64
Levy              object
Manufacturer      object
Prod. year         int64
Category          object
Leather interior  object
Fuel type         object
Engine volume     object
Mileage           object
Cylinders          int64
Gear box type     object
Drive wheels      object
Wheel             object
Color             object
Airbags            int64
```

Figura 1: Formato de las variables

Tal y como se puede ver, hay variables categóricas como **Levy**, **Engine volume** o **Mileage** que deberían ser numéricas. Por otro lado, otras son numéricas a pesar de poder ser codificadas tal y como se verá más adelante.

Para trabajar con esta base de datos se van a analizar todas las variables por separado, de forma que se puedan controlar con seguridad los formatos y los valores perdidos o anómalos de cada una.

2.1. Variables numéricas.

En esta sección se van a analizar las variables numéricas o aquellas que deberían ser numéricas. En este caso son **Price**, **Levy**, **Prod. year**, **Engine volume**, **Mileage**, **Cylinders**, **Airbags**. Algunas están en formato numérico y otras en formato de texto tal y como se indica en la imagen 1, por lo que será conveniente convertir todas a formato numérico antes de comenzar con el verdadero análisis.

En el caso de la variable **Price** no se observan datos faltantes o perdidos, pero sí un registro con un valor anormalmente alto (se puede observar en la imagen 24 del anexo)

```
var = 'Price'
print(len(datos[var].unique()))
print(datos[var].isna().sum())
datos[var].hist(bins = 30, color = 'mediumorchid')
plt.ylim(top = 20)
plt.show()
```

Listing 2: Tratamiento de la variable **Price**.

Por el contrario, la variable **Levy** contiene valores faltantes identificados con el carácter ". Por ello, se cambia el formato de la columna a número y se sustituyen los registros faltantes por *NaN*:

```
var = 'Levy'

print(len(datos[var].unique()))
print(datos[var].isna().sum())
print(datos[var].unique())

datos[var] = datos[var].replace("-", np.nan)
num_missing = datos[var].isna().sum()
levy = [float(dato) for dato in datos[var].tolist()]
datos[var] = levy
porcentaje = round(100*num_missing/len(datos[var].tolist()), 2)
print(f"\nLa variable Levy tiene {num_missing} valores perdidos (un {porcentaje}%)")

datos[var].hist(bins = 30, color = 'mediumorchid')
plt.ylim(top = 50)
plt.show()
```

Listing 3: Tratamiento de la variable **Levy**.

La siguiente variable a analizar es **Prod. year**. Esta columna se encuentra en el formato correcto y sólo contiene un *outlier* que no supone un valor perdido/anómalo.

```
var = 'Prod. year'

print(len(datos[var].unique()))
print(datos[var].isna().sum())
datos[var].hist(bins = 30, color = 'mediumorchid')
plt.ylim(top = 20)
plt.show()
```

Listing 4: Tratamiento de la variable **Prod. year**.

En la columna **Engine volume** se tienen 58 valores únicos que se pueden pasar a formato numérico. Para ello es necesario separar las cadenas de aquellos vehículos cuyo motor contiene un turbocompresor. Esta información se recoge en una nueva variable categórica y binaria llamada **Turbo**, que contiene un 1 o un 0 en el caso de que el vehículo registrado contenga turbo o no respectivamente.

```
var = 'Engine volume'
print(len(datos[var].unique()))
print(datos[var].unique())
engine_volume = []
turbo = []
for dato in datos[var]:
    cadenas = dato.split()
    engine_volume.append(float(cadenas[0]))
    if len(cadenas) > 1:
        turbo.append(str(1))
    else:
        turbo.append(str(0))
datos[var] = engine_volume
datos["Turbo"] = turbo

datos[var].hist(color = 'mediumorchid')
```

Listing 5: Tratamiento de la variable **Engine volume**.

La variable **Mileage** contiene la unidad de medida en cada registro (la cadena "km"), por lo que se elimina la unidad y se transforman los datos a formato numérico. Al generar el histograma de los datos se observa un valor anómalo extremadamente alto (con un valor de más de 10^9 km, siendo la vida media de un vehículo unos 3×10^5 km). Por tanto, es necesario sustituir este valor por *NaN* para su posterior imputación aleatoria.

Por último, las variables **Cylinders** y **Airbags** no presentan valores anómalos o perdidos y están en el formato correcto, por lo que no es necesario modificar nada.

El siguiente y último paso para el tratamiento de las variables numéricas consiste en imputar los datos faltantes. Para ello se hace uso de la función *ImputacionCuant* de la librería *FuncionesMineria.py* (el código

de la función se puede encontrar en el anexo). El método de imputación consiste en la selección de valores aleatorios en función de la distribución de la variable. Las únicas columnas que contienen valores nulos son **Levy** (644) y **Mileage** (1):

```
for col in list(datos.columns):
    print(f"{col}: {datos[col].isna().sum()}")
print()

columnas = ["Levy", "Mileage"]
for var in columnas:
    datos[var] = ImputacionCuant(datos[var], 'aleatorio')

for col in columnas:
    print(f"{col}: {datos[col].isna().sum()}")
```

Listing 6: Imputación de datos faltantes.

2.2. Variables categóricas.

En esta sección se van a analizar las columnas no numéricas de la base de datos para comprobar si contienen valores no válidos o están en formatos no correspondientes. Con el siguiente código se pueden comprobar los valores únicos de cada una:

```
numericas = datos.select_dtypes(include = ['int', 'int32', 'int64', 'float', 'float32',
            'float64']).columns.tolist()
categoricas = [col for col in datos.columns.tolist() if col not in numericas]

for var in categoricas:
    uv = len(datos[var].unique())
    print(f"La variable {var} tiene {uv} valores únicos:")
    print(datos[var].unique())
    print()
```

Listing 7: Variables categóricas.

Estas variables contienen una baja cantidad de valores únicos, por lo que podrían ser codificadas. Sin embargo, el desarrollo de los modelos de árboles hace que no sea necesario ese proceso en el tratamiento de los datos, ya que a la hora de clasificar los datos mediante ramas, estas actúan como codificación. Sin embargo, las variables **Leather interior** y **Color** sí se han modificado para una mayor claridad en la clasificación de las mismas:

- La variable **Leather interior** se puede codificar para que los registros contengan 1 en lugar de Yes y 0 en lugar de No.
- La variable **Gear box type** se puede codificar para que los registros contengan 1 en lugar de $Automatic$ y 0 en lugar de $Tiptronic$. También se ha cambiado el nombre de la columna a **Automatic gear box**.
- La variable **Wheel** se puede codificar para que los registros contengan 1 en lugar de $Left wheel$ y 0 en lugar de $Right - handdrive$. Tmabién se ha cambiado el nombre de la columna a **Left wheel**.
- La variable **Color** se puede codificar para que los registros contengan 1 en lugar de $White$ y 0 en lugar de $Black$.

El código es el siguiente:

```
two_value_variables = ['Leather interior', 'Gear box type', 'Wheel', 'Color']
ones = ['Yes', 'Automatic', 'Left wheel', 'White']

for var in two_value_variables:
    encoded_data = []
    for dato in datos[var]:
        if dato in ones:
            encoded_data.append('1')
        else:
            encoded_data.append('0')
    datos[var] = encoded_data
```

```
12    datos = datos.rename(columns={'Gear box type': 'Automatic gear box',
13                'Wheel': 'Left wheel'})
14    print(datos.dtypes)
15    datos.to_excel("datos_tarea25_clean.xlsx")
```

Listing 8: Codificación de variables categóricas.

El resultado es un *DataFrame* con los datos limpios y los siguientes formatos:

```
Price                int64
Levy               float64
Manufacturer        object
Prod. year           int64
Category            object
Leather interior    object
Fuel type           object
Engine volume      float64
Mileage            float64
Cylinders            int64
Automatic gear box  object
Drive wheels        object
Left wheel          object
Color               object
Airbags              int64
Turbo               object
```

Figura 2: Formato de las variables tras la depuración.

3. Modelo de árbol de decisiones.

El primer modelo a crear es un árbol de decisiones que permita clasificar y predecir la variable objetivo **Color**. En el siguiente listado de código se lee el nuevo archivo de datos con la información depurada y se muestra el *DataFrame*. También se imprime por pantalla la frecuencia absoluta de cada valor de la variable objetivo.

```
df = pd.read_excel('datos_tarea25_clean.xlsx')
display(df.head())
print(f'\nLa frecuencia de cada clase es: \n{df.Color.value_counts()}')
```

Listing 9: Lectura de datos y análisis de la variable objetivo.

A continuación es necesario separar la variable objetivo de las predictoras. Para ello se separan los datos en una muestra de entrenamiento y una de prueba siguiendo el criterio $80-20$ y se comprueba que ambas submuestras siguen una distribución parecida.

```
variables = df.columns.tolist()
objetivo = ['Color']
predictoras = [col for col in variables if col not in objetivo]
X = df[predictoras]
y = df[objetivo]
```

Listing 10: Separación de las variables y de la muestra.

El primer modelo de árbol simple es el siguiente. Para crearlo, se han escogido los valores de los parámetros de forma que se minimiza el sobreajuste (la diferencia entre las predicciones de *train* y *test* se hace mínima) sin que conlleve pérdidas en la eficacia de las predicciones.

```
arbol1 = DecisionTreeClassifier(min_samples_split = 200,
            criterion = 'entropy', max_depth = 25)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
            random_state=42)
print(f'Frecuencia en train es: \n{y_train.value_counts(normalize=True)}')
print(f'\nFrecuencia en test es: \n{y_test.value_counts(normalize=True)}')
arbol1.fit(X_train, y_train)
y_train_pred = arbol1.predict(X_train)
y_test_pred = arbol1.predict(X_test)
print(f'Accuracy para train de: {accuracy_score(y_train,y_train_pred)}')
print(f'Accuracy para test de: {accuracy_score(y_test,y_test_pred)}')
print(f'Diferencia de precisión entre train y test:
            {accuracy_score(y_train,y_train_pred)-accuracy_score(y_test,y_test_pred)}')
accuracy_arbol_base = accuracy_score(y_test, y_test_pred)
```

Listing 11: Primer modelo de árbol de regresión.

El siguiente paso consiste en buscar los parámetros y las métricas para la evaluación de la bondad del modelo. En este caso se han elegido distintas profundidades máximas entre 5 y 50 para compensar la pérdida de capacidad predictiva con el riesgo de sobreajuste, así como número mínimo de casos (*min_samples_split*) entre 10 y 300. Los criterios elegidos son el de Gini (desigualdad) y el de entropía (aleatoriedad). Las métricas objetivo son las incluidas en la variable *scoring_metrics*.

```
params = {
    'max_depth': [5, 10, 15, 20, 25, 30, 40, 50],
    'min_samples_split': [10, 15, 30, 50, 100, 200, 300],
    'criterion': ["gini", "entropy"]}
scoring_metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
grid_search = GridSearchCV(estimator=arbol1, param_grid=params, cv=4,
            scoring = scoring_metrics, refit='accuracy')
grid_search.fit(X_train, y_train)
results = pd.DataFrame(grid_search.cv_results_)
best_model = grid_search . best_estimator_
print('Parámetros del mejor modelo de árbol por GridSearchCV:')
print(best_model)
y_train_pred = best_model.predict(X_train)
y_test_pred = best_model.predict(X_test)
accuracy_arbol_gscv = accuracy_score(y_test, y_test_pred)
print(f'Precisión del mejor modelo de árbol por GridSearchCV: {accuracy_arbol_gscv}')
```

Listing 12: Resultados del proceso *GridSearchCV*.

Se obtiene que el mejor modelo, según *GridSearchCV* es aquel con profundidad **max_depth** = 10, divisiones **min_samples_split** = 15 y **criterion** = *entropy*. Al repetir este proceso para cada una de las métricas se puede obtener una estimación del mejor modelo en cada caso, que es aquel que maximiza estas. Para ello se hace una nueva validación cruzada cambiando los parámetros en la variable *refit*:

```
    # Accuracy:
    grid_search = GridSearchCV(estimator=arbol1, param_grid=params, cv=4,
                    scoring = scoring_metrics, refit='accuracy')
    grid_search.fit(X_train, y_train)
    best_model = grid_search.best_estimator_
    print("El mejor modelo para 'Accuracy' es:")
    print(grid_search.best_estimator_)
    print()
    # Precision:
    grid_search = GridSearchCV(estimator=arbol1, param_grid=params, cv=4,
                    scoring = scoring_metrics, refit='precision_macro')
    grid_search.fit(X_train, y_train)
    best_model = grid_search.best_estimator_
    print("El mejor modelo para 'Precision' es:")
    print(grid_search.best_estimator_)
    print()
    # Recall:
    grid_search = GridSearchCV(estimator=arbol1, param_grid=params, cv=4,
                    scoring = scoring_metrics, refit='recall_macro')
    grid_search.fit(X_train, y_train)
    best_model = grid_search.best_estimator_
    print("El mejor modelo para 'Recall' es:")
    print(grid_search.best_estimator_)
    print()
    # F1:
    grid_search = GridSearchCV(estimator=arbol1, param_grid=params, cv=4,
                    scoring = scoring_metrics, refit='f1_macro')
    grid_search.fit(X_train, y_train)
    best_model = grid_search.best_estimator_
    print("El mejor modelo para 'F1' es:")
    print(grid_search.best_estimator_)
```

Listing 13: Búsqueda del mejor modelo para cada métrica.

Los resultados de esta búsqueda se pueden encontrar en la siguiente tabla:

Métrica	**Criterio**	**max_depth**	**min_samples_split**
Accuracy	*entropy*	20	10
Precision	*entropy*	20	10
Recall	*entropy*	10	10
F1	*entropy*	10	10

Tabla 1: Mejores modelos para cada métrica según el método *grid_search.best_estimator_*

A partir de esta tabla se puede empezar a considerar un modelo que siga el criterio de entropía, con una profundidad máxima de entre 10 y 20 y un número mínimo de *splits* de 10. Sin embargo, conviene comprobar el comportamiento de las distintas métricas en función de los parámetros *max_depth*, *min_samples_split* con el fin de evitar el sobreajuste o la pérdida de capacidad predictiva ya mencionados.

En las gráficas de la figura 3 se puede observar que, para todas las métricas, se alcanza un máximo local en el promedio cuando la profundidad máxima del árbol es igual a 10. Además, en el caso de 10 *splits*, el máximo absoluto se alcanza para una profundidad de 20 nodos. A juzgar por las gráficas, los valores óptimos para maximizar las métricas sin perder información (esto es, eligiendo una profundidad o un número de *splits* muy bajos) sería **max_depth** = 10 y **min_samples_split** = 10. Esta elección de parámetros se asimila a la búsqueda *grid search* del mejor estimador para la métrica **accuracy** y no se aleja de los valores de las búsquedas restantes (ver tabla 1).

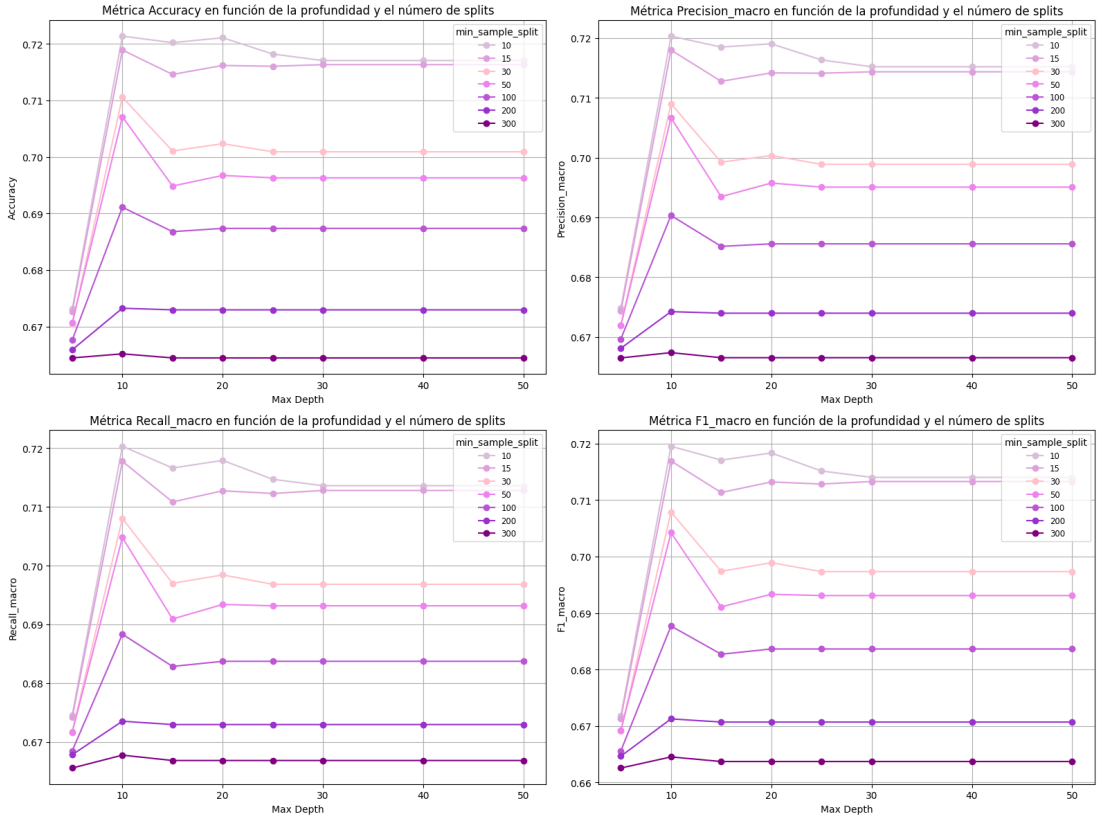

Figura 3: Evolución de las métricas del árbol en función de los parámetros.

Por lo tanto, se van a considerar todos los modelos con una profundidad máxima de entre 10 y 20 nodos (incluidos), así como un mínimo de divisiones de 10 y que sigan el criterio de aleatoriedad (entropía):

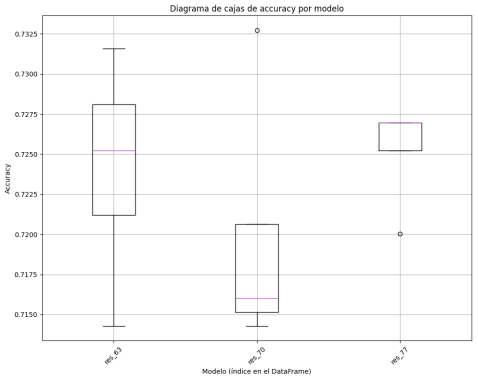

Figura 4: Diagrama de cajas para accuracy del árbol. Se muestran aquellos modelos con $10 <$ **max_depth** $<$ 20 y **min_samples_split** $= 10$.

De todos ellos, el mejor candidato es el que tiene índice 63 (*res_63*) ya que, aunque no tiene la mayor mediana, posee un rango intercuartílico no demasiado amplio y no tiene valores anómalos (no como otros candidatos parecidos, como el resultado *res_77*. El modelo 63 posee los parámetros **max_depth** $= 10$ y **min_samples_split** $= 10$.

```
print(results.iloc[63].params)
modelo_63 = grid_search.cv_results_['params'][63]
best_model = grid_search.estimator.set_params(**modelo_63)
y_test_pred = best_model.predict(X_test)
accuracy_arbol_bmanual = accuracy_score(y_test, y_test_pred)
print(f'Precisión del mejor modelo de árbol: {accuracy_arbol_bmanual}')
```

Listing 14: Elección del mejor modelo por búsqueda manual.

Por otro lado, si se ordenan los resultados y se genera otro diagrama de cajas, se puede obtener una nueva estimación de cuál es el mejor modelo según *GridSearchCV*. En este caso, el mejor modelo obtenido (*res_2*) coincide con el mejor modelo del *boxplot* de la figura 4:

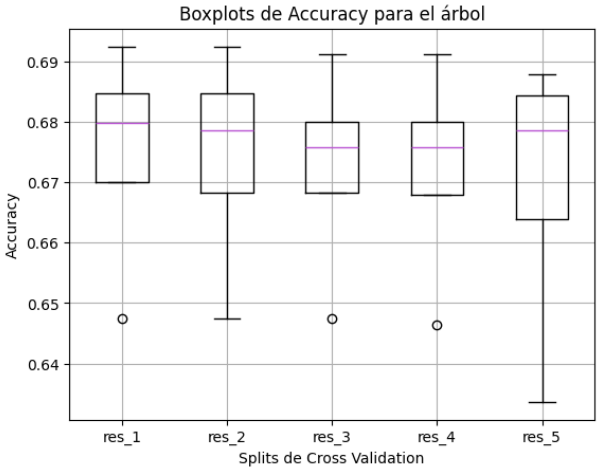

Figura 5: Diagrama de cajas para accuracy del árbol. Se muestran aquellos modelos con $10 <$ **max_depth** $<$ 20 y **min_samples_split** $= 10$.

3.1. Resultados del modelo de árbol.

La siguiente tabla consiste en una recopilación de los mejores modelos según los distintos tipos de búsqueda de parámetros realizadas (mediante *GridSearchCV*, manual o con gráficas, mediante *boxplot*, etc).

Modelo	**criterion**	**max_depth**	**min_samples_split**
Árbol base	*entropy*	25	200
GridSearchCV	*entropy*	20	10
Manual	*entropy*	10	10
Boxplot	*entropy*	10	10

Tabla 2: Parámetros de los mejores modelos de árbol encontrados.

Y los resultados para **accuracy** de estos modelos se pueden encontrar en la siguiente imagen:

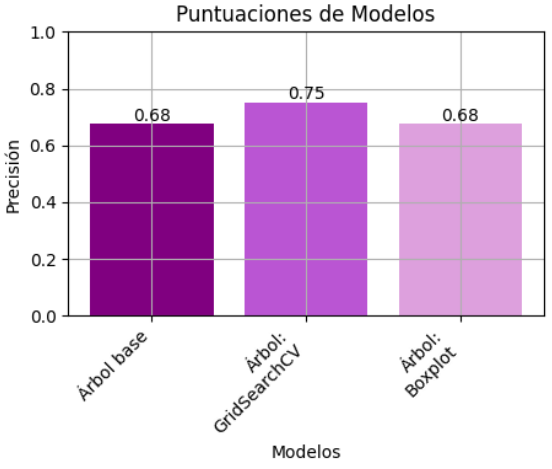

Figura 6: Resultados de los modelos de árbol.

Por lo tanto, el mejor modelo es aquel con **max_depth** $= 20$ y **min_samples_split** $=$ **10**. Los resultados para ambas submuestras según este modelo son los siguientes:

```
Matriz de Confusión:
[[1735  150]
 [ 127 1460]]

Medidas de Desempeño:
              precision    recall  f1-score   support

           0       0.93      0.92      0.93      1885
           1       0.91      0.92      0.91      1587

    accuracy                           0.92      3472
   macro avg       0.92      0.92      0.92      3472
weighted avg       0.92      0.92      0.92      3472
```

```
Matriz de Confusión:
[[346  96]
 [119 307]]

Medidas de Desempeño:
              precision    recall  f1-score   support

           0       0.74      0.78      0.76       442
           1       0.76      0.72      0.74       426

    accuracy                           0.75       868
   macro avg       0.75      0.75      0.75       868
weighted avg       0.75      0.75      0.75       868
```

Figura 7: Medidas de bondad del ajuste en la submuestra de entrenamiento (izqa) y de prueba (dcha).

Y las curvas ROC para ambas submuestras son las siguientes:

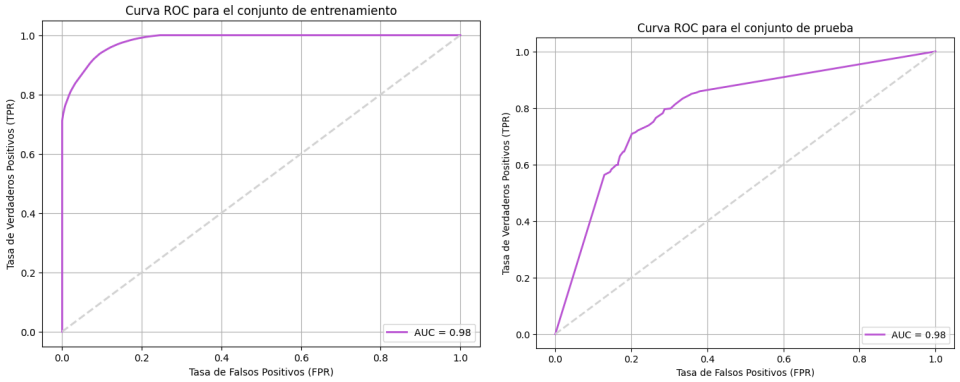

Figura 8: Curvas ROC para la submuestra de entrenamiento (izqa) y de prueba (dcha).

La importancia de las variables para el árbol ganador es la siguiente:

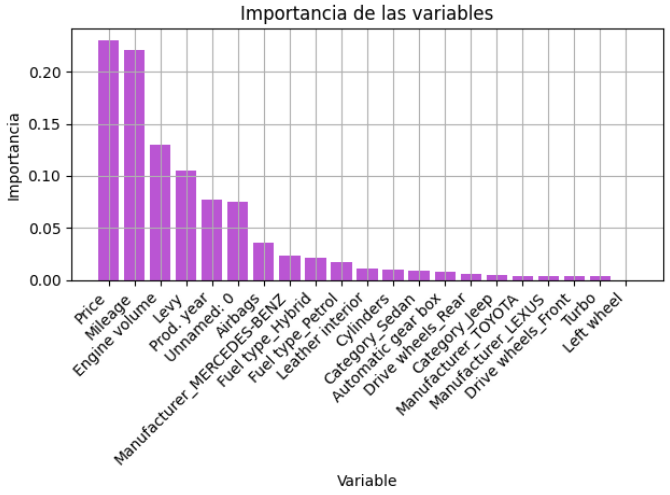

Figura 9: Importancia de las variables para la toma de decisiones en el árbol de la figura **??**.

En esta gráfica se puede observar que las variables **Price**, **Mileage** y **Engine volume** son las que más contribuyen al desarrollo del árbol (con contribuciones menos importantes pero notables de **Levy**, **Prod. year**, etcétera).

Y por último, el árbol de clasificación asociado, con profundidad 20, un número mínimo de divisiones de 10 y según el criterio de entropía, se genera visualmente con el siguiente código. Es un árbol en el que el nodo padre se divide en numerosos nodos hijo, dando lugar a un esquema muy profundo y difícil de interpretar::

```
plt.figure(figsize=(20, 15))
plot_tree(best_model, feature_names=X.columns.tolist(), filled=True, proportion = True)
plt.show()
```

Listing 15: Código para generar el árbol.

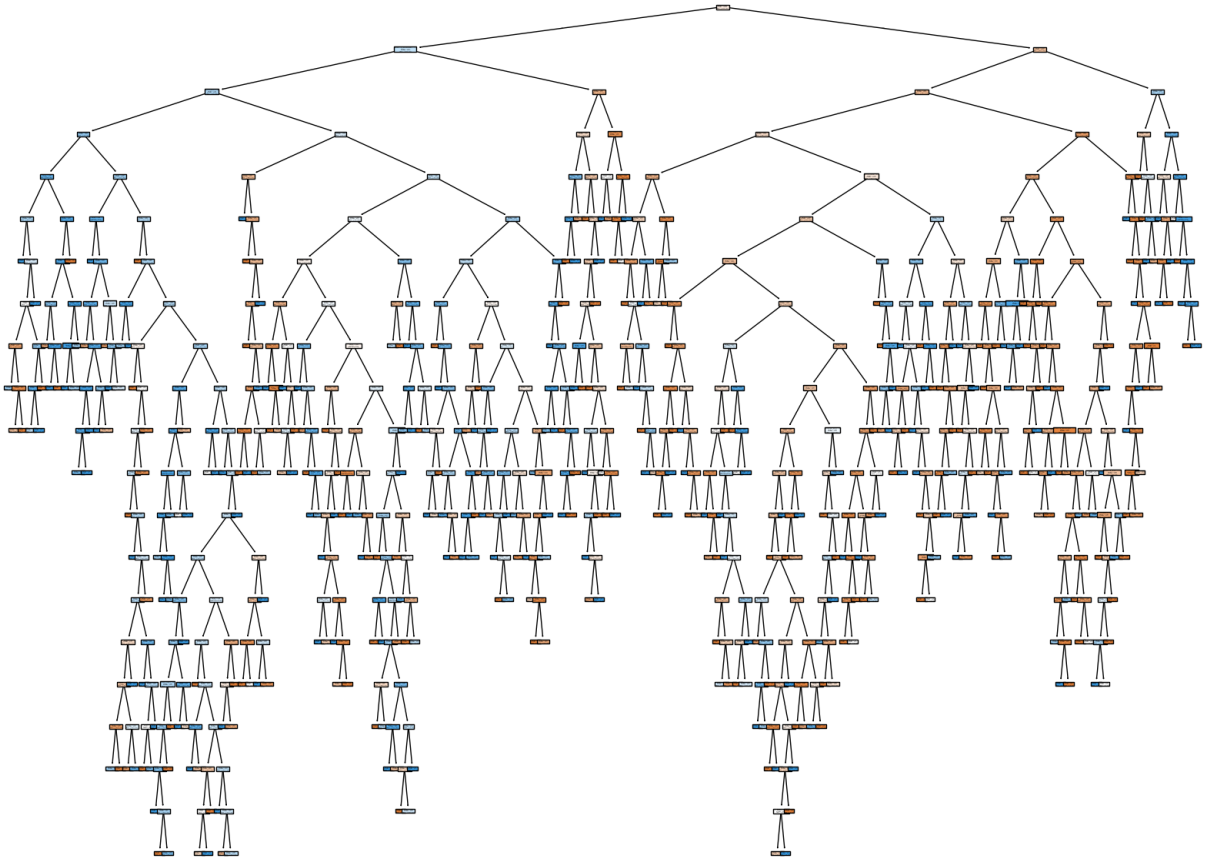

Figura 10: Árbol de decisiones asociado al mejor estimador elegido.

Nota: Las normas del árbol, que contienen toda la información sobre los nodos de este, se obtienen ejecutando el código del siguiente listado. Debido a la extensión del texto, estas no se incluyen en este documento, pero sí en un archivo .txt adjunto.

```
rules = export_text(best_model, feature_names = list(X.columns), show_weights = True)
with open('decision_tree_rules.txt', 'w', encoding='utf-8') as f:
    f.write(rules)
print(rules)
```

Listing 16: Normas de clasificación del árbol de la figura 10.

4. Modelo de *Random Forest.*

En esta sección se va a desarrollar un modelo de *Random Forest*. Para ello es necesario llevar a cabo un proceso de ensamblado de cierto número de árboles conocido como *Bagging* (*Bootsrap Averaging*). En este proceso se utilizan distintas submuestras de los datos para el desarrollo de los árboles, alternando entre diferentes parámetros como el tamaño de dichas submuestras, el número de árboles desarrollados, si hay reemplazamiento de las submuestras o no, etc. Este modelo de múltiples árboles permite paliar las desventajas que los árboles simples conllevan, como la poca fiabilidad del ajuste o la eficacia de las predicciones.

Con un modelo básico de *Random Forest* con una elección arbitraria de parámetros que den lugar a un sobreajuste bajo se consigue un valor de **accuracy** parecido, tal y como se puede ver con las siguientes líneas. Estas incluyen también valores de precisión para las predicciones de ambos conjuntos (entrenamiento y prueba):

```
RF_model = RandomForestClassifier(n_estimators = 100, bootstrap = True, max_depth = 25,
            min_samples_split = 10, criterion = 'entropy', min_samples_leaf = 10,
            random_state=123)
RF_model.fit(X_train, y_train)
y_pred_rf = RF_model.predict(X_test)
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f'Precisión del modelo con RF estándar: {accuracy_rf}')
y_train_pred = RF_model.predict(X_train)
y_test_pred = RF_model.predict(X_test)
print(f'Se tiene un accuracy para train de: {accuracy_score(y_train,y_train_pred)}')
print(f'Se tiene un accuracy para test de: {accuracy_score(y_test,y_test_pred)}')
```

Listing 17: Primer modelo de *Random Forest*.

Al igual que en la sección anterior, se puede hacer una búsqueda de los parámetros óptimos:

```
params = {
    'n_estimators' : [50, 100, 200],
    'max_depth': [10, 15, 20, 25],
    'bootstrap': [True, False],
    'min_samples_leaf' : [5, 10, 20, 30],
    'min_samples_split': [2, 5, 10, 15, 20, 30, 50, 100],
    'criterion': ["gini", "entropy"]}

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

Listing 18: Búsqueda *grid search* para el modelo de *Random Forest*.

El mejor modelo dado por la validación cruzada es el siguiente, que además es fruto de la combinación de los parámetros de la tabla 3. El código también incluye los resultados de las predicciones de dicho modelo:

```
best_model_RF = grid_search_RF.best_estimator_
print(grid_search_RF.best_estimator_)
y_pred_rf = best_model_RF.predict(X_test)
accuracy_rf_gscv = accuracy_score(y_test, y_pred_rf)
print(f'Precisión del modelo: {accuracy_rf_gscv}')

y_train_pred = best_model_RF.predict(X_train)
y_test_pred = best_model_RF.predict(X_test)
print(f'Se tiene un accuracy para train de: {accuracy_score(y_train,y_train_pred)}')
print(f'Se tiene un accuracy para test de: {accuracy_score(y_test,y_test_pred)}')
```

Listing 19: Mejor modelo por validación cruzada de *GridSearchCV* para *Random Forest*.

Y si se ordenan de forma manual los resultados de la validación cruzada, se puede obtener el siguiente diagrama de cajas para la elección de modelos:

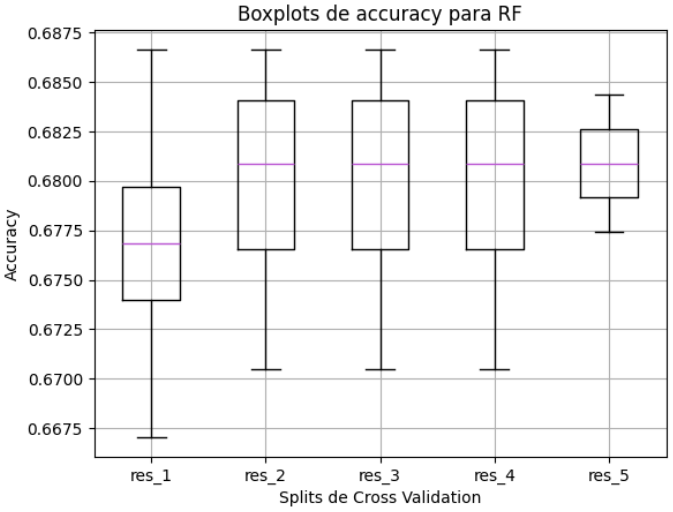

Figura 11: Diagrama de cajas para **accuracy** del modelo de *Random Forest*.

Nuevamente, es posible generar gráficas que ayuden a la elección de parámetros para maximizar **accuracy**. En este caso esas gráficas tienen la siguiente forma:

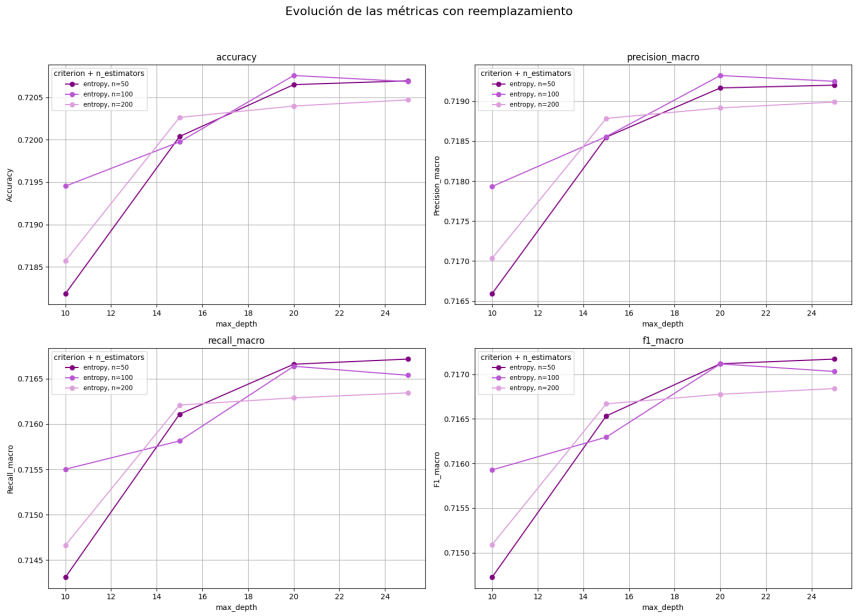

Figura 12: Evolución de la precisión del modelo de RF en función de los parámetros sin reemplazamiento.

Se puede observar que los resultados tienen una representación diferente y menos intuitiva que en los anteriores casos. Por otro lado, si se representan las mismas medidas para $bootstrap = False$ se obtiene:

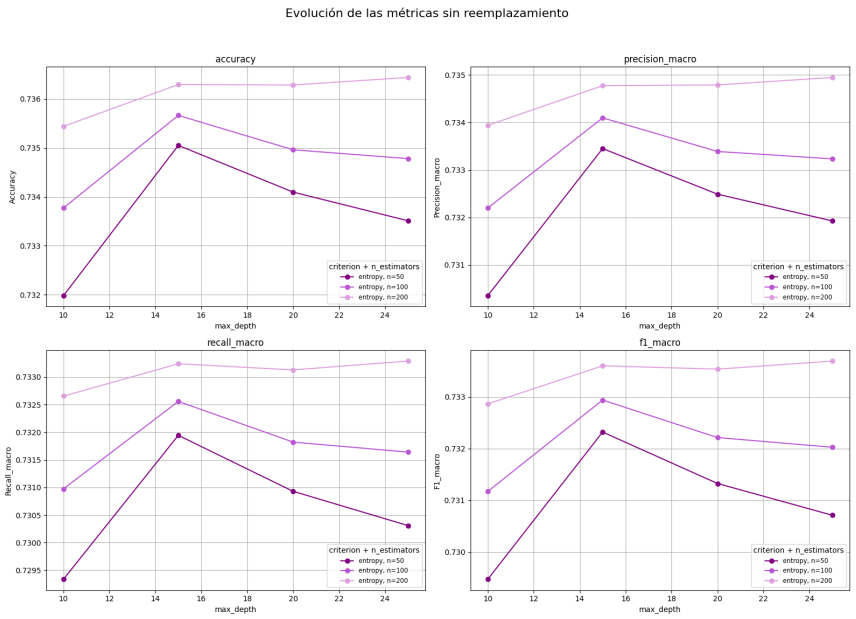

Figura 13: Evolución de la precisión del modelo de RF en función de los parámetros sin reemplazamiento.

Como en la imagen 13 hay mejores resultados, y estos no varían mucho entre 50 y 200 estimadores, se eligen los parámetros que maximicen la precisión sin elevar el tiempo de computación de forma excesiva.

De todos los modelos que cumplen con las condiciones de los máximos de las figuras 12 y 13, los mejores modelos son los siguientes:

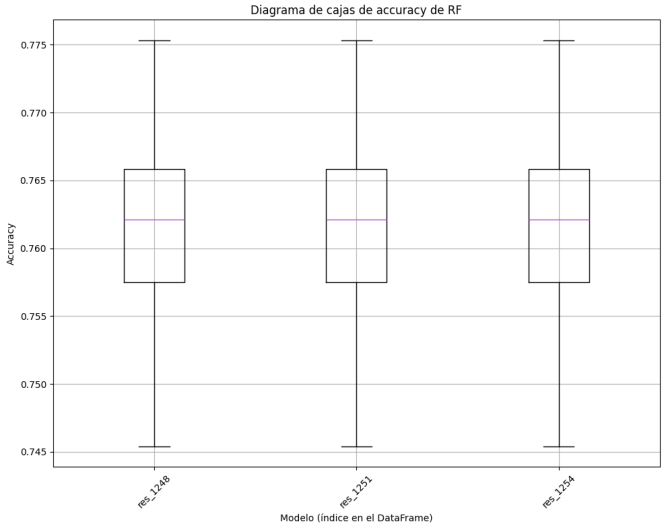

Figura 14: Diagrama de cajas de accuracy para RF. Se muestran modelos con **n_estimators** = 50, **max_depth** = 15 y media de **accuracy** alta.

Los resultados de estos modelos son idénticos y sus parámetros sólo difieren en **min_samples_split**. Se puede escoger cualquiera de los tres:

```
print(results.iloc[1254].params)
modelo_1254 = grid_search_RF.cv_results_['params'][1254]
best_model = grid_search_RF.estimator.set_params(**modelo_1254)
y_train_pred = best_model.predict(X_train)
y_test_pred = best_model.predict(X_test)
accuracy_modelo_1254 = accuracy_score(y_test, y_test_pred)
print(f'Precisión del modelo 1254 por búsqueda manual: {accuracy_modelo_1254}')
```

Listing 20: Mejor modelo por búsqueda manual de RF.

4.1. Resultados del modelo de *Random Forest*.

Los mejores modelos obtenidos son:

Modelo	n	max_depth	min_samples_leaf	min_samples_split	*bootstrap*
Base	100	25	10	10	True
GridSearchCV	50	15	5	2	False
Boxplot	50	10	30	100	True
Manual	50	15	5	10	False

Tabla 3: Búsqueda de parámetros para *Random Forest*. Todos con **criterion** = *entropy*.

La precisión de cada modelo se puede ver en la imagen 15. Más adelante se verá una comparación con la figura 6 para comprobar que la precisión de los modelos aumenta para el algoritmo de *Random Forest*.

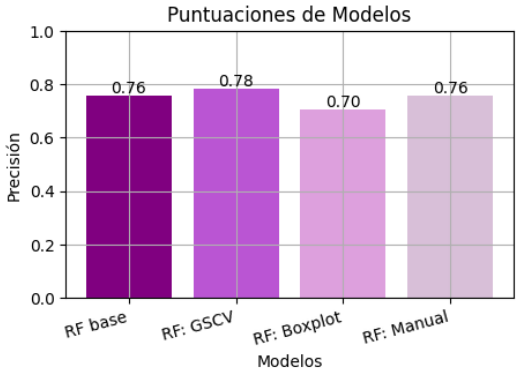

Figura 15: Resultados de los modelos de RF.

Tal y como se puede observar, el mejor modelo es aquel encontrado mediante *GridSearchCV* y el método *best_ estimator*. La importancia de las variables es otra medida que conviene comprobar una vez escogido el modelo ganador. En este caso las variables más importantes son las mismas que en el caso del árbol simple, aunque en otro orden.

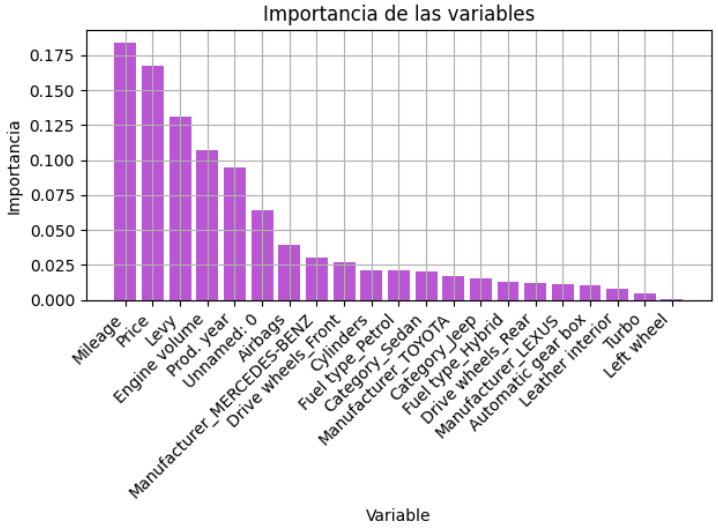

Figura 16: Importancia de las variables en el modelo ganador de RF.

5. Modelo de *XGBoost*.

El último modelo emplea un algoritmo de *Extreme Gradient Boosting* o *XGBoost*. Este consiste en una variante del método de *Gradient Boosting* más rápido y eficiente. Este proceso de ensamblado construye árboles que se emplean como una mejora del árbol anterior. Tienen mayor riesgo de sobreajuste que los modelos anteriores, por lo que es más difícil encontrar los parámetros óptimos.

Un modelo inicial es el siguiente. Los parámetros que dan lugar al clasificador inicial se han elegido de forma arbitraria para reducir el sobreajuste de las predicciones lo máximo posible.

```
xgb_classifier = XGBClassifier(booster = 'gbtree', n_estimators = 200,
                            eta = 0.05, gamma = 3.5, random_state=123, max_depth = 25,
tree_method = 'hist')
xgb_classifier.fit(X_train, y_train)
y_pred_base = xgb_classifier.predict(X_test)
accuracy_xgb = accuracy_score(y_test, y_pred_base)
y_train_pred = xgb_classifier.predict(X_train)
y_test_pred = xgb_classifier.predict(X_test)
print(f'Se tiene un accuracy para train de: {accuracy_score(y_train,y_train_pred)}')
print(f'Se tiene un accuracy para test de: {accuracy_score(y_test,y_test_pred)}')
print()
print(f'Diferencia de precisión entre train y test:
        {accuracy_score(y_train,y_train_pred) - accuracy_score(y_test,y_test_pred)}')
```

Listing 21: Clasificador inicial del modelo XGB.

Sobre este clasificador inicial se construye una red de árboles sobre la cual se busca la mejor combinación de parámetros para maximizar la precisión del modelo. Se ha comprobado que el número de estimadores (árboles) y la profundidad de los mismos no contribuyen al sobreajuste cuando el resto de parámetros tienen valores parecidos a los siguientes:

```
params = {
    'n_estimators': [150, 200, 250],
    'eta' : [0.01, 0.05, 0.1, 0.15, 0.2],
    'gamma' : [0.5, 1, 1.5, 2],
    'max_depth': [3, 5, 10, 15]}

scoring_metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
grid_search_XGB = GridSearchCV(estimator=xgb_classifier,
                            param_grid=params,
                            cv=4, scoring = scoring_metrics, refit='accuracy')
grid_search_XGB.fit(X_train, y_train)
modelo_XGB = grid_search_XGB.best_estimator_
y_train_pred_xgb = modelo_XGB.predict(X_train)
y_test_pred_xgb = modelo_XGB.predict(X_test)
accuracy_xgb_gscv = accuracy_score(y_test,y_test_pred_xgb)
print(f'Se tiene un accuracy para train de: {accuracy_score(y_train,y_train_pred_xgb)}')
print(f'Se tiene un accuracy para test de: {accuracy_score(y_test,y_test_pred_xgb)}')
print()
print('Resultados para Modelo')
print(classification_report(y_test, y_test_pred_xgb))
print()
print('Parámetros del mejor estimador:')
params_interes = ['n_estimators', 'max_depth', 'eta', 'gamma']
for param in params_interes:
    valor = modelo_XGB.get_params()[param]
    print(f'- {param}: {valor}')
```

Listing 22: Clasificador inicial del modelo XGB y búsqueda *GridSearchCV*.

Se puede comprobar que a pesar de intentar paliar el sobreajuste entre las predicciones de entrenamiento y de prueba, el modelo da lugar a mucha diferencia en la precisión de estas. Si se genera un diagrama de cajas con los mejores resultados de la búsqueda se obtiene lo siguiente:

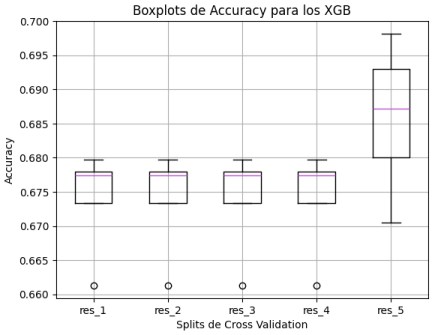

Figura 17: Diagrama de cajas para accuracy del modelo XGB.

Por otro lado, la evolución de la precisión del modelo en función de estos parámetros es la siguiente:

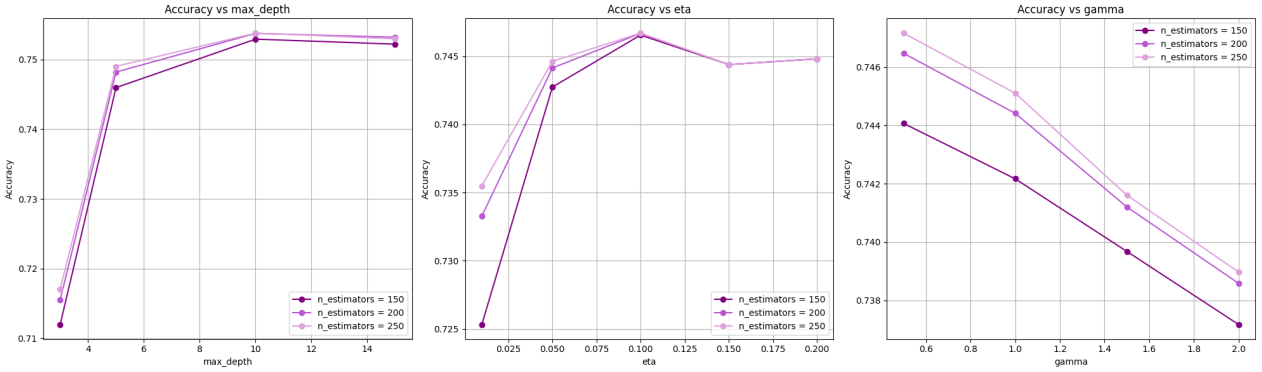

Figura 18: Evolución de la precisión del modelo XGB en función de los parámetros.

El número de árboles no afecta de manera significativa a la precisión del modelo, por lo que es mejor escoger el valor que minimice el tiempo de computación (el menor de todos). Sin embargo, la elección de otros parámetros se puede optimizar. Por ejemplo, los mejores parámetros para **accuracy** son **max_depth** = 10, **eta** = 0,1 y **gamma** = 0,5. La búsqueda manual proporciona los siguientes resultados, donde claramente el mejor modelo es aquel con índice 203 debido a su poca dispersión y bajo rango intercuartílico:

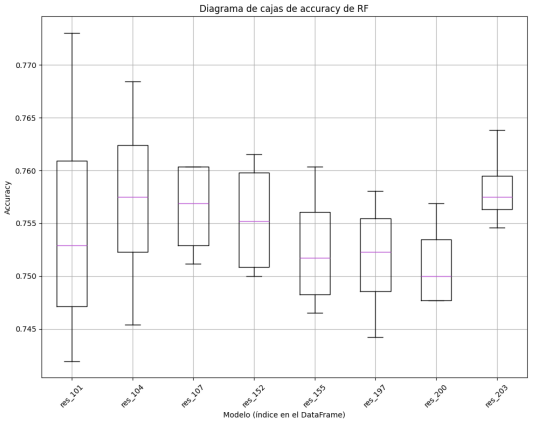

Figura 19: Diagrama de cajas de **accuracy** para XGB. Se muestran modelos con los parámetros determinados por la figura 18.

5.1. Resultados del modelo de *XGBoost*.

Los parámetros de todos los modelos medidos en esta sección se encuentran en la siguiente tabla:

Búsqueda	**n_estimators**	**max_depth**	**eta**	**gamma**
Base	200	25	3,5	0,05
GridSearchCV	150	10	0,20	1,00
Boxplot	200	3	0,01	1,50
Manual	250	15	0,20	0,50

Tabla 4: Búsqueda de parámetros para XGB.

Estos modelos se pueden ver comparados en cuanto a su precisión en el siguiente gráfico de barras:

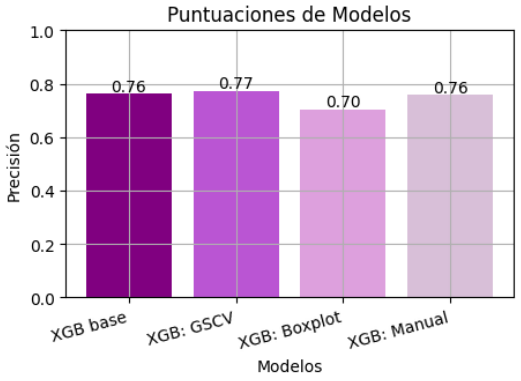

Figura 20: Puntuaciones de los modelos de XGB.

Por lo que el modelo ganador vuelve a ser aquel encontrado por la búsqueda con *GridSearchCV*.

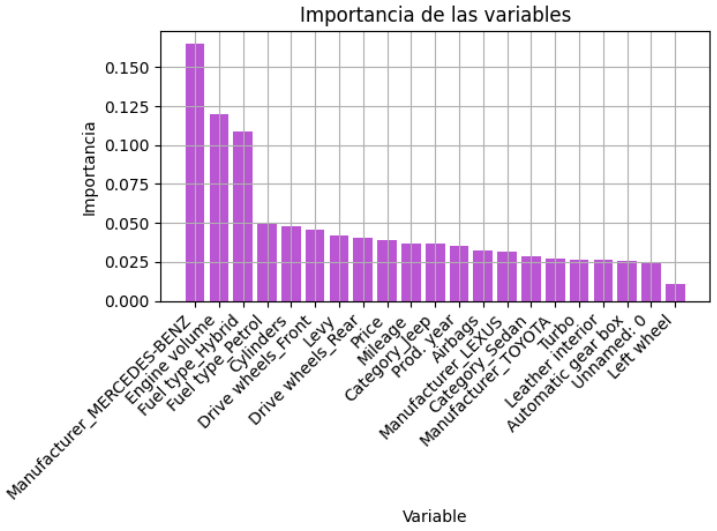

Figura 21: importancia de las variables para el modelo ganador de XGB.

Se puede comprobar que en este caso, las variables **Price** y **Mileage** han perdido importancia (tal y como se esperaba según el método de *Extreme Gradient Boosting*). Las variables más importantes en el desarrollo de este modelo son, en orden de importancia: **Manufacturer_MERCEDES-BENZ** (variable *dummie*), **Engine volume** y **Fuel type_Hybrid** (*dummie*).

6. Comparación de modelos y conclusiones.

6.1. Comparación.

A continuación se encuentra una comparación en forma de gráfico de barras que contiene las precisiones de los distintos modelos ganadores de cada sección:

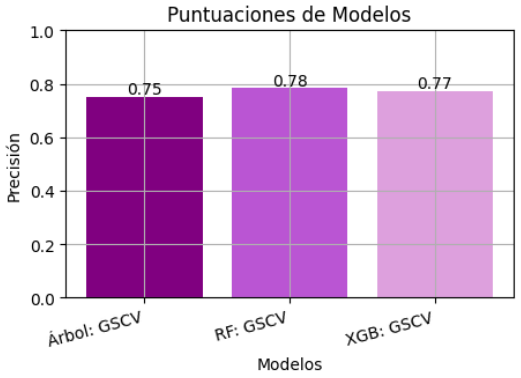

Figura 22: Precisión de los modelos ganadores.

Tal y como se observa, en los tres casos el modelo ganador ha sido aquel obtenido mediante el método *best_ estimator* de la función *GridSearchCV*. El modelo con mayor precisión es el de *Random Forest*, aunque las diferencias con los otros dos no son significativas. Esto se debe, probablemente, a la búsqueda manual de parámetros en los estimadores o clasificadores base, que se ha llevado a cabo para reducir el sobreajuste en las primeras predicciones hechas en cada sección. La curva ROC de estos tres modelos ganadores son las siguientes:

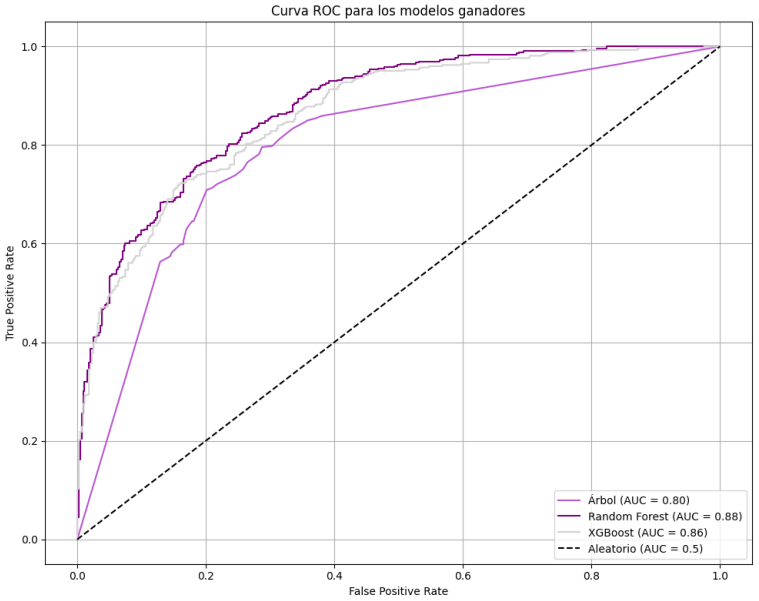

Figura 23: Curvas ROC de los modelos ganadores.

Donde el modelo de *Random Forest* presenta la mayor superficie bajo la curva ROC.

6.2. Conclusiones.

En este trabajo se han comparado tres algoritmos de clasificación: árboles de decisión y *Random Forest* y *Extreme Gradient Boosting* (*XGBoost* o XGB). Cada uno presenta ventajas y limitaciones que se han evaluado en función del coste computacional, precisión del modelo, importancia de variables y análisis de las curvas ROC.

Coste computacional: el árbol de decisiones es el modelo más ligero en términos de tiempo de entrenamiento y consumo de recursos, lo que lo convierte en una buena opción cuando se requiere rapidez o se trabaja con entornos con limitaciones computacionales. En cambio, el "bosque.ºbtenido por RF supone un mayor coste, especialmente al aumentar la profundidad, ya que implica muchas particiones recursivas sobre el conjunto de datos. Sin embargo, el modelo más costoso es el de XGB, debido a la necesidad de entrenar múltiples árboles (hasta 250 en este caso), lo que implica una mayor demanda computacional tanto en tiempo como en memoria.

Precisión de los modelos: En términos de rendimiento, RF ha obtenido la mayor precisión en el conjunto de prueba, alcanzando un 78.45 %, seguido por XGB con un 77,30 %, y el árbol de decisión con un 75,23 %. Aunque la diferencia entre modelos no es abismal, sí se observa una mejora progresiva al incorporar modelos más complejos y robustos frente al sobreajuste. *Random Forest* destaca especialmente por su capacidad de generalización y su resistencia al sobreajuste frente a árboles individuales.

Importancia de las variables: En la regresión logística, la interpretación de los coeficientes permite identificar qué variables tienen mayor peso en la predicción, destacando por ejemplo las variables relacionadas con la antigüedad/uso y el precio. En el árbol de decisión, la importancia se deriva del orden de las divisiones, y variables como **Price** y **Mileage** aparecen como claves. Para *Random Forest*, al combinar múltiples árboles, se ofrece una medida más estable y agregada de importancia, donde nuevamente variables como **Price**, **Mileage** o **Levy** parecen ser determinantes. En el caso del modelo ganador por XGB, Las variables más importantes son más generales y no tienen tanto que ver con los datos propios de cada vehículo. Estas son **Manufacturer**, **Engine volume** y **Fuel Type**.

Curvas ROC: Las curvas ROC de los tres modelos muestran cómo se comportan frente a distintas tasas de falsos positivos. El árbol de decisiones tiene $AUC = 0{,}80$, XGB alcanza $AUC = 0{,}86$ y RF ha logrado $AUC = 0{,}88$, el más alto de todos. Esto refuerza su capacidad para separar correctamente las clases, con una alta sensibilidad y especificidad. Las curvas indican que *Random Forest* no solo tiene mayor precisión sino también mayor robustez en la clasificación.

En resumen, aunque el árbol simple es adecuado por su simplicidad y bajo coste, el algoritmo de *Random Forest* ofrece el mejor equilibrio entre rendimiento y fiabilidad, a cambio de un coste computacional superior. La elección del modelo dependerá del contexto y de los recursos disponibles.

7. Anexo.

En esta sección se pueden encontrar listados de códigos, tablas e imágenes que aportan información adicional a la práctica pero no son de gran importancia. **NO HACE FALTA CORREGIR ESTA SECCIÓN**.

7.1. Librerías.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn
from sklearn.tree import plot_tree
import seaborn as sns
from sklearn.tree import DecisionTreeClassifier, export_text, DecisionTreeRegressor
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, roc_curve, auc
from sklearn.metrics import make_scorer, mean_absolute_error, mean_squared_error, r2_score
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier, BaggingRegressor
from sklearn.ensemble import GradientBoostingClassifier,GradientBoostingRegressor
from xgboost import XGBClassifier, XGBRegressor
from sklearn.model_selection import learning_curve

import warnings
warnings.filterwarnings('ignore')
```

Listing 23: Librerías usadas en la práctica.

7.2. Funciones.

```
def ImputacionCuant(var, tipo):
    """
    Esta función realiza la imputación de valores faltantes en una variable cuantitativa.
    Datos de entrada:
    - var: Serie de datos cuantitativos con valores faltantes a imputar.
    - tipo: Tipo de imputación ('media', 'mediana' o 'aleatorio').
    Datos de salida:
    - Una nueva serie con valores faltantes imputados.
    """
    # Realiza una copia de la variable para evitar modificar la original
    vv = var.copy()
    if tipo == 'media':
        # Imputa los valores faltantes con la media de la variable
        vv[np.isnan(vv)] = round(np.nanmean(vv), 4)
    elif tipo == 'mediana':
        # Imputa los valores faltantes con la mediana de la variable
        vv[np.isnan(vv)] = round(np.nanmedian(vv), 4)
    elif tipo == 'aleatorio':
        # Imputa los valores faltantes de manera aleatoria basada en la distribución de valores existentes
        x = vv[~np.isnan(vv)]
        frec = x.value_counts(normalize=True).reset_index()
        frec.columns = ['Valor', 'Frec']
        frec = frec.sort_values(by='Valor')
        frec['FrecAcum'] = frec['Frec'].cumsum()
        random_values = np.random.uniform(min(frec['FrecAcum']), 1, np.sum(np.isnan(vv)))
        imputed_values = list(map(lambda x: list(frec['Valor'][frec['FrecAcum'] <= x])[-1], random_values))
        vv[np.isnan(vv)] = [round(x, 4) for x in imputed_values]

    return vv
```

Listing 24: Funciones importadas en la práctica.

7.3. Imágenes.

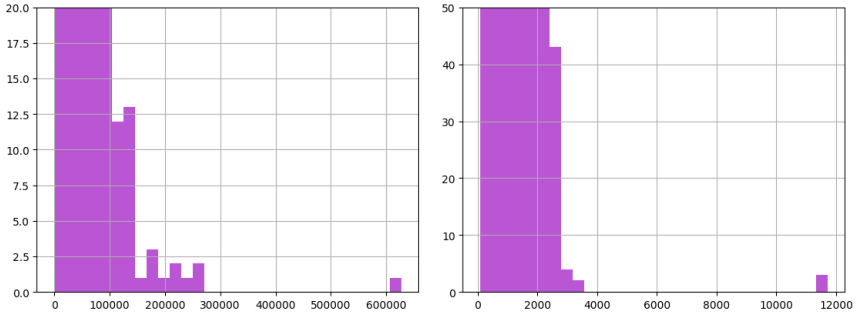

Figura 24: Histograma de las variables **Price** y **Levy**.

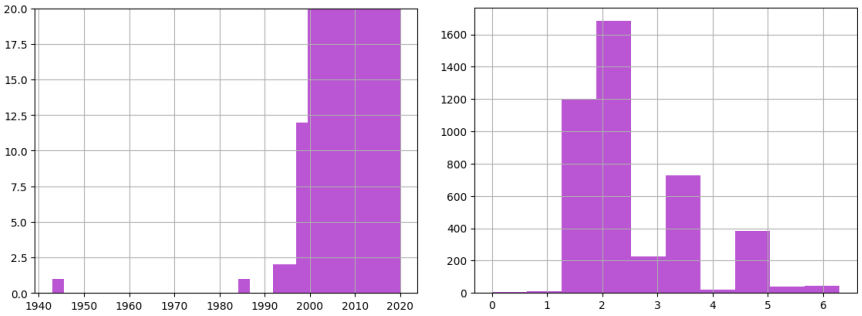

Figura 25: Histograma de las variables **Prod. year** y **Engine volume**.

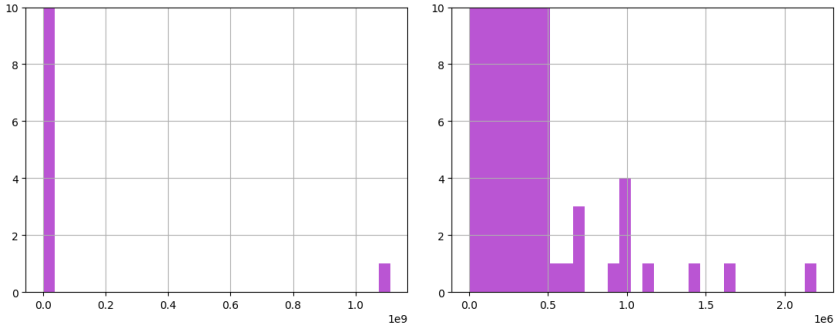

Figura 26: Histograma de la variable **Mileage**.

7.4. Códigos para generar imágenes.

```
    grid_search = GridSearchCV(estimator=arbol1, param_grid=params, cv=4,
                scoring = scoring_metrics, refit='accuracy')
    grid_search.fit(X_train, y_train)
    results = pd.DataFrame(grid_search.cv_results_)
    scoring_metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
    fig, axes = plt.subplots(2, 2, figsize = (16, 12))
    axes = axes.flatten()
    palette = ['thistle', 'plum', 'pink', 'violet', 'mediumorchid', 'darkorchid', 'purple']

    for i, metric in enumerate(scoring_metrics):
        ax = axes[i]
        mean_col = f'mean_test_{metric}'
        std_col = f'std_test_{metric}'
        grouped = results.groupby(['param_max_depth', 'param_min_samples_split'])[[mean_col,
```

```
                    std_col]].mean().reset_index()
        min_split_vals = sorted(grouped['param_min_samples_split'].unique())

        for j, val in enumerate(min_split_vals):
            subset = grouped[grouped['param_min_samples_split'] == val]
            x = subset['param_max_depth']
            y = subset[mean_col]
            ax.plot(x, y, marker = 'o', label = f'{val}', color = palette[j])

        ax.set_title(f'Métrica {metric.capitalize()} en función de la profundidad
                    y el núumero de splits')
        ax.set_xlabel('Max Depth')
        ax.set_ylabel(metric.capitalize())
        ax.grid(True)
        ax.legend(title ='min_sample_split', loc = 'upper right', fontsize = 'small')
    plt.tight_layout()
    plt.show()
```

Listing 25: Código para generar la figura 3.

```
filtro = (
    (results['param_max_depth'] >= 10) &
    (results['param_max_depth'] <= 20) &
    (results['param_min_samples_split'] == 10) &
    (results['param_criterion'] == 'entropy')
)
subset = results[filtro]
split_cols = ['split0_test_accuracy', 'split1_test_accuracy', 'split2_test_accuracy',
        'split3_test_accuracy']

data = [row[split_cols].values for _, row in subset.iterrows()]
labels = [f'res_{i}' for i in subset.index]
plt.figure(figsize = (10, 8))
plt.boxplot(data, labels = labels, medianprops = {'color': 'mediumorchid'})
plt.title('Diagrama de cajas de accuracy por modelo (max_depth = 20, min_samples_split =
 15,
        criterion = "entropy")')
plt.ylabel('Accuracy')
plt.xlabel('Modelo (índice en el DataFrame)')
plt.xticks(rotation = 45)
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 26: Código para generar la figura 4.

```
sorted_results = results.sort_values(by='rank_test_accuracy', ascending=True).head(5)
res_1 = results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy',
'split3_test_accuracy']].iloc[0]
res_2 = results[['split0_test_accuracy', 'split1_test_accuracy',
            'split2_test_accuracy', 'split3_test_accuracy']].iloc[1]
res_3 = results[['split0_test_accuracy', 'split1_test_accuracy',
            'split2_test_accuracy', 'split3_test_accuracy']].iloc[2]
res_4 = results[['split0_test_accuracy', 'split1_test_accuracy',
            'split2_test_accuracy', 'split3_test_accuracy']].iloc[3]
res_5 = results[['split0_test_accuracy', 'split1_test_accuracy',
            'split2_test_accuracy', 'split3_test_accuracy']].iloc[4]

plt.boxplot(
    [res_1.values, res_2.values, res_3.values, res_4.values, res_5.values],
    labels = ['res_1','res_2','res_3','res_4','res_5'],
    medianprops = {'color': 'mediumorchid'})
plt.title('Boxplots de Accuracy para los 4 Splits')
plt.xlabel('Splits de Cross Validation')
plt.ylabel('Accuracy')
plt.grid()
plt.show()

sorted_results['params'].iloc[1]
```

Listing 27: Código para generar la figura 5.

```
        model_names = ['Árbol base', 'Árbol:\nGridSearchCV', 'Árbol:\nBoxplot']
    scores = [accuracy_arbol_base, accuracy_arbol_gscv, accuracy_arbol_bmanual]
    plt.figure(figsize=(5, 3))
    bars = plt.bar(np.arange(len(model_names)), scores, color=['purple', 'mediumorchid', '
    plum'])
    plt.xticks(np.arange(len(model_names)), model_names, rotation=45, ha='right')
    plt.ylim(0, 1.0)
    plt.title('Puntuaciones de Modelos')
    plt.xlabel('Modelos')
    plt.ylabel('Precisión')
    plt.grid()
    for bar, score in zip(bars, scores):
        plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 0.01, f'{score:.2f}',
     ha='center', color='black')
    plt.show()
```

Listing 28: Código para generar la figura 6.

```
    y_train_auc = pd.get_dummies(y_train,drop_first=True)
    y_prob_train = best_model.predict_proba(X_train)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_train_auc, y_prob_train)
    roc_auc = auc(fpr, tpr)
    print(f"\nÁrea bajo la curva ROC (AUC): {roc_auc:.2f}")
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='mediumorchid', lw=2, label=f'AUC = {roc_auc:.2f}')
    plt.plot([0, 1], [0, 1], color='lightgrey', lw=2, linestyle='--')
    plt.xlabel('Tasa de Falsos Positivos (FPR)')
    plt.ylabel('Tasa de Verdaderos Positivos (TPR)')
    plt.title('Curva ROC para el conjunto de entrenamiento')
    plt.legend(loc="lower right")
    plt.grid()
    plt.show()

    y_test_auc = pd.get_dummies(y_test,drop_first=True)
    y_prob_test = best_model.predict_proba(X_test)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_test_auc, y_prob_test)
    roc_auc_test = auc(fpr, tpr)
    print(f"\nÁrea bajo la curva ROC (AUC): {roc_auc:.2f}")
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='mediumorchid', lw=2, label=f'AUC = {roc_auc:.2f}')
    plt.plot([0, 1], [0, 1], color='lightgrey', lw=2, linestyle='--')
    plt.xlabel('Tasa de Falsos Positivos (FPR)')
    plt.ylabel('Tasa de Verdaderos Positivos (TPR)')
    plt.title('Curva ROC para el conjunto de prueba')
    plt.legend(loc="lower right")
    plt.grid()
    plt.show()
```

Listing 29: Código para generar la figura 8.

```
    df_importancia = pd.DataFrame({'Variable': best_model.feature_names_in_,
            'Importancia':best_model.feature_importances_}).sort_values(by='Importancia',
            ascending=False)
    plt.bar(df_importancia['Variable'], df_importancia['Importancia'], color='mediumorchid')
    plt.xlabel('Variable')
    plt.ylabel('Importancia')
    plt.title('Importancia de las variables')
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout()
    plt.grid()
    plt.show()
```

Listing 30: Código para generar la figura 9.

```
    results = pd.DataFrame(grid_search_RF.cv_results_)
    sorted_results = results.sort_values(by='mean_test_accuracy', ascending=True).head(5)

    res_1 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy',
                'split2_test_accuracy', 'split3_test_accuracy']].iloc[0]
    res_2 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy',
```

```
                'split2_test_accuracy', 'split3_test_accuracy']].iloc[1]
res_3 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy',
                'split2_test_accuracy', 'split3_test_accuracy']].iloc[2]
res_4 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy',
                'split2_test_accuracy', 'split3_test_accuracy']].iloc[3]
res_5 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy',
                'split2_test_accuracy', 'split3_test_accuracy']].iloc[4]

plt.boxplot([res_1.values,res_2.values,res_3.values,res_4.values,res_5.values],
                labels = ['res_1','res_2','res_3','res_4','res_5'],
                medianprops = {'color': 'mediumorchid'})
plt.title('Boxplots de Accuracy para los 4 Splits')
plt.xlabel('Splits de Cross Validation')
plt.ylabel('Accuracy')
plt.grid()
plt.show()

best_model_rf_boxplot = RandomForestClassifier(**sorted_results['params'].iloc[4],
random_state=123)
best_model_rf_boxplot.fit(X_train, y_train)
print('Parámetros del mejor modelo de RF por boxplot:')
print(best_model_rf_boxplot.get_params)
print()
y_train_pred = best_model_rf_boxplot.predict(X_train)
y_test_pred = best_model_rf_boxplot.predict(X_test)
print(f'Se tiene un accuracy para train de: {accuracy_score(y_train,y_train_pred)}')
print(f'Se tiene un accuracy para test de: {accuracy_score(y_test,y_test_pred)}')
print(f'Diferencia de precisión entre train y test: {accuracy_score(y_train,y_train_pred
)-accuracy_score(y_test,y_test_pred)}')
print()
accuracy_rf_boxplot = accuracy_score(y_test, y_test_pred)
print(f'Precisión del modelo: {accuracy_rf_boxplot}')
print(f'Resultados para el modelo: {classification_report(y_test, y_test_pred)}')
```

Listing 31: Código para generar la figura 11.

```
scoring_metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
palette = ['purple', 'mediumorchid', 'plum', 'orchid', 'hotpink', 'violet', 'darkorchid'
]
titles = ['con reemplazamiento', 'sin reemplazamiento']
results['param_bootstrap'] = results['param_bootstrap'].astype(bool)
results['param_criterion'] = results['param_criterion'].astype(str)
index = 0
for boot in [True, False]:
    fig, axes = plt.subplots(2, 2, figsize=(16, 12))
    axes = axes.flatten()

    for crit in ['gini', 'entropy']:
        filtered = results[
            (results['param_bootstrap'] == boot) &
            (results['param_criterion'] == crit)
        ]

        for i, metric in enumerate(scoring_metrics):
            ax = axes[i]
            mean_col = f'mean_test_{metric}'

            grouped = filtered.groupby(
                ['param_max_depth', 'param_n_estimators']
            )[mean_col].mean().reset_index()

            for j, n_est in enumerate(sorted(grouped['param_n_estimators'].unique())):
                subset = grouped[grouped['param_n_estimators'] == n_est]
                ax.plot(subset['param_max_depth'], subset[mean_col],
                        marker='o', label=f'{crit}, n={n_est}',
                        color=palette[j % len(palette)])

            ax.set_title(f'{metric}')
            ax.set_xlabel('max_depth')
            ax.set_ylabel(metric.capitalize())
```

```
                ax.grid(True)
                ax.legend(title='criterion + n_estimators', fontsize='small')

        plt.suptitle(f'Evolución de las métricas {titles[index]}', fontsize=16)
        index += 1
        plt.tight_layout(rect=[0, 0.03, 1, 0.95])
        plt.show()
```

Listing 32: Código para generar las figuras 12 y 13.

```
filtro = (
    (results['param_max_depth'] == 15) &
    (results['param_n_estimators'] == 50) &
    (results['param_criterion'] == 'entropy')
)
subset = results[filtro]
split_cols = ['split0_test_accuracy', 'split1_test_accuracy', 'split2_test_accuracy', '
split3_test_accuracy']
subset['mean_accuracy'] = subset[split_cols].mean(axis = 1)
subset = subset[subset['mean_accuracy'] > 0.76]
data = [row[split_cols].values for _, row in subset.iterrows()]
labels = [f'res_{i}' for i in subset.index]
plt.figure(figsize = (10, 8))
plt.boxplot(data, labels = labels, medianprops = {'color': 'mediumorchid'})
plt.title('Diagrama de cajas de accuracy por modelo (media > 0.7)')
plt.ylabel('Accuracy')
plt.xlabel('Modelo (índice en el DataFrame)')
plt.xticks(rotation = 45)
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 33: Código para generar la figura 14.

```
model_names = ['RF base', 'RF: GSCV', 'RF: Boxplot', 'RF: Manual']
scores = [accuracy_rf_base, accuracy_rf_gscv, accuracy_rf_boxplot, accuracy_modelo_1254]
plt.figure(figsize=(5, 3))
bars = plt.bar(np.arange(len(model_names)), scores, color=['purple', 'mediumorchid', '
plum', 'thistle'])
plt.xticks(np.arange(len(model_names)), model_names, rotation=15, ha='right')
plt.ylim(0, 1.0)
plt.title('Puntuaciones de Modelos')
plt.xlabel('Modelos')
plt.ylabel('Precisión')
plt.grid()
for bar, score in zip(bars, scores):
    plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 0.01, f'{score:.2f}',
 ha='center', color='black')
plt.show()
```

Listing 34: Código para generar la figura 15.

```
best_model = grid_search_RF.best_estimator_
df_importancia = pd.DataFrame({'Variable': best_model.feature_names_in_, 'Importancia':
best_model.feature_importances_}).sort_values(by='Importancia', ascending=False)
plt.bar(df_importancia['Variable'], df_importancia['Importancia'], color='mediumorchid')
plt.xlabel('Variable')
plt.ylabel('Importancia')
plt.title('Importancia de las variables')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.grid()
plt.show()
```

Listing 35: Código para generar la figura 16.

```
results = pd.DataFrame(grid_search_XGB.cv_results_)

sorted_results = results.sort_values(by='mean_test_accuracy', ascending=True).head(5)
res_1 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy',
```

```
                'split2_test_accuracy', 'split3_test_accuracy']].iloc[0]
    res_2 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy',
                'split2_test_accuracy', 'split3_test_accuracy']].iloc[1]
    res_3 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy',
                'split2_test_accuracy', 'split3_test_accuracy']].iloc[2]
    res_4 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy',
                'split2_test_accuracy', 'split3_test_accuracy']].iloc[3]
    res_5 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy',
                'split2_test_accuracy', 'split3_test_accuracy']].iloc[4]

    plt.boxplot([res_1.values,res_2.values,res_3.values,res_4.values,res_5.values], labels =
     ['res_1','res_2','res_3','res_4','res_5'], medianprops = {'color': 'mediumorchid'})
    plt.title('Boxplots de Accuracy para los 4 Splits')
    plt.xlabel('Splits de Cross Validation')
    plt.ylabel('Accuracy')
    plt.grid()
    plt.show()
    print('Parámetros del mejor estimador:')
    print(sorted_results['params'].iloc[4])
```

Listing 36: Código para generar la figura 17.

```
    results = pd.DataFrame(grid_search_XGB.cv_results_)
fig, axes = plt.subplots(1, 3, figsize = (20, 6))
axes = axes.flatten()
palette = ['purple', 'mediumorchid', 'plum']

for i, n in enumerate(sorted(results['param_n_estimators'].unique())):
    subset = results[results['param_n_estimators'] == n]
    grouped = subset.groupby('param_max_depth')['mean_test_accuracy'].mean().reset_index()
    axes[0].plot(grouped['param_max_depth'], grouped['mean_test_accuracy'],
                label = f'n_estimators = {n}', marker = 'o', color = palette[i])
axes[0].set_title('Accuracy vs max_depth')
axes[0].set_xlabel('max_depth')
axes[0].set_ylabel('Accuracy')
axes[0].legend()
axes[0].grid(True)
for i, n in enumerate(sorted(results['param_n_estimators'].unique())):
    subset = results[results['param_n_estimators'] == n]
    grouped = subset.groupby('param_eta')['mean_test_accuracy'].mean().reset_index()
    axes[1].plot(grouped['param_eta'], grouped['mean_test_accuracy'],
                label = f'n_estimators = {n}', marker = 'o', color = palette[i])
axes[1].set_title('Accuracy vs eta')
axes[1].set_xlabel('eta')
axes[1].set_ylabel('Accuracy')
axes[1].legend()
axes[1].grid(True)
for i, n in enumerate(sorted(results['param_n_estimators'].unique())):
    subset = results[results['param_n_estimators'] == n]
    grouped = subset.groupby('param_gamma')['mean_test_accuracy'].mean().reset_index()
    axes[2].plot(grouped['param_gamma'], grouped['mean_test_accuracy'],
                label = f'n_estimators = {n}', marker = 'o', color = palette[i])
axes[2].set_title('Accuracy vs gamma')
axes[2].set_xlabel('gamma')
axes[2].set_ylabel('Accuracy')
axes[2].legend()
axes[2].grid(True)
plt.tight_layout()
plt.show()
```

Listing 37: Código para generar la figura 18.

```
filtro = (
    (results['param_max_depth'] >= 5) &
    (results['param_max_depth'] <= 15) &
    (results['param_eta'] >= 0.1) &
    (results['param_gamma'] == 0.5) &
    (results['param_n_estimators'] == 250))
subset = results[filtro]
```

```
split_cols = ['split0_test_accuracy', 'split1_test_accuracy', 'split2_test_accuracy', '
    split3_test_accuracy']
subset['mean_accuracy'] = subset[split_cols].mean(axis = 1)
subset = subset[subset['mean_accuracy'] > 0.75]
data = [row[split_cols].values for _, row in subset.iterrows()]
labels = [f'res_{i}' for i in subset.index]
plt.figure(figsize = (10, 8))
plt.boxplot(data, labels = labels, medianprops = {'color': 'mediumorchid'})
plt.title('Diagrama de cajas de accuracy de RF')
plt.ylabel('Accuracy')
plt.xlabel('Modelo (índice en el DataFrame)')
plt.xticks(rotation = 45)
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 38: Código para generar la figura 19.

```
    model_names = ['XGB base', 'XGB: GSCV', 'XGB: Boxplot', 'XGB: Manual']
    scores = [accuracy_xgb_base, accuracy_xgb_gscv, accuracy_xgb_boxplot,
    accuracy_modelo_203]
    plt.figure(figsize=(5, 3))
    bars = plt.bar(np.arange(len(model_names)), scores, color=['purple', 'mediumorchid', '
    plum', 'thistle'])
    plt.xticks(np.arange(len(model_names)), model_names, rotation=15, ha='right')
    plt.ylim(0, 1.0)
    plt.title('Puntuaciones de Modelos')
    plt.xlabel('Modelos')
    plt.ylabel('Precisión')
    plt.grid()
    for bar, score in zip(bars, scores):
        plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 0.01, f'{score:.2f}',
     ha='center', color='black')
    plt.show()
```

Listing 39: Código para generar la figura 20.

```
    best_model = grid_search_XGB.best_estimator_
    df_importancia = pd.DataFrame({'Variable': best_model.feature_names_in_,
            'Importancia': best_model.feature_importances_}).sort_values(by='Importancia',
    ascending=False)
    plt.bar(df_importancia['Variable'], df_importancia['Importancia'], color='mediumorchid')
    plt.xlabel('Variable')
    plt.ylabel('Importancia')
    plt.title('Importancia de las variables')
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout()
    plt.grid()
    plt.show()
```

Listing 40: Código para generar la figura 21.

```
    model_names = ['Árbol: GSCV', 'RF: GSCV', 'XGB: GSCV']
    scores = [accuracy_arbol_gscv, accuracy_rf_gscv, accuracy_xgb_gscv]
    plt.figure(figsize=(5, 3))
    bars = plt.bar(np.arange(len(model_names)), scores, color=['purple', 'mediumorchid', '
    plum', 'thistle'])
    plt.xticks(np.arange(len(model_names)), model_names, rotation=15, ha='right')
    plt.ylim(0, 1.0)
    plt.title('Puntuaciones de Modelos')
    plt.xlabel('Modelos')
    plt.ylabel('Precisión')
    plt.grid()
    for bar, score in zip(bars, scores):
        plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 0.01, f'{score:.2f}',
     ha='center', color='black')
```

Listing 41: Código para generar la figura 22.

```python
# Modelo 1: árbol
modelo_1 = grid_search.best_estimator_
modelo_1.fit(X_train, y_train)
y_prob1 = modelo_1.predict_proba(X_test)[:, 1]
fpr1, tpr1, _ = roc_curve(y_test, y_prob1)
roc_auc1 = auc(fpr1, tpr1)

# Modelo 2: RF
modelo_2 = grid_search_RF.best_estimator_
modelo_2.fit(X_train, y_train)
y_prob2 = modelo_2.predict_proba(X_test)[:, 1]
fpr2, tpr2, _ = roc_curve(y_test, y_prob2)
roc_auc2 = auc(fpr2, tpr2)

# Modelo 3: XGB
modelo_3 = grid_search_XGB.best_estimator_
modelo_3.fit(X_train, y_train)
y_prob3 = modelo_3.predict_proba(X_test)[:, 1]
fpr3, tpr3, _ = roc_curve(y_test, y_prob3)
roc_auc3 = auc(fpr3, tpr3)

# Gráfica ROC
plt.figure(figsize=(10, 8))
plt.plot(fpr1, tpr1, label = f'Árbol (AUC = {roc_auc1:.2f})', color='mediumorchid')
plt.plot(fpr2, tpr2, label = f'Random Forest (AUC = {roc_auc2:.2f})', color='purple')
plt.plot(fpr3, tpr3, label = f'XGBoost (AUC = {roc_auc3:.2f})', color='lightgrey')
plt.plot([0, 1], [0, 1], 'k--', label = 'Aleatorio (AUC = 0.5)')

plt.title('Curva ROC para los modelos ganadores')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc = 'lower right')
plt.grid(True)
plt.tight_la
```

Listing 42: Código para generar la figura 23.