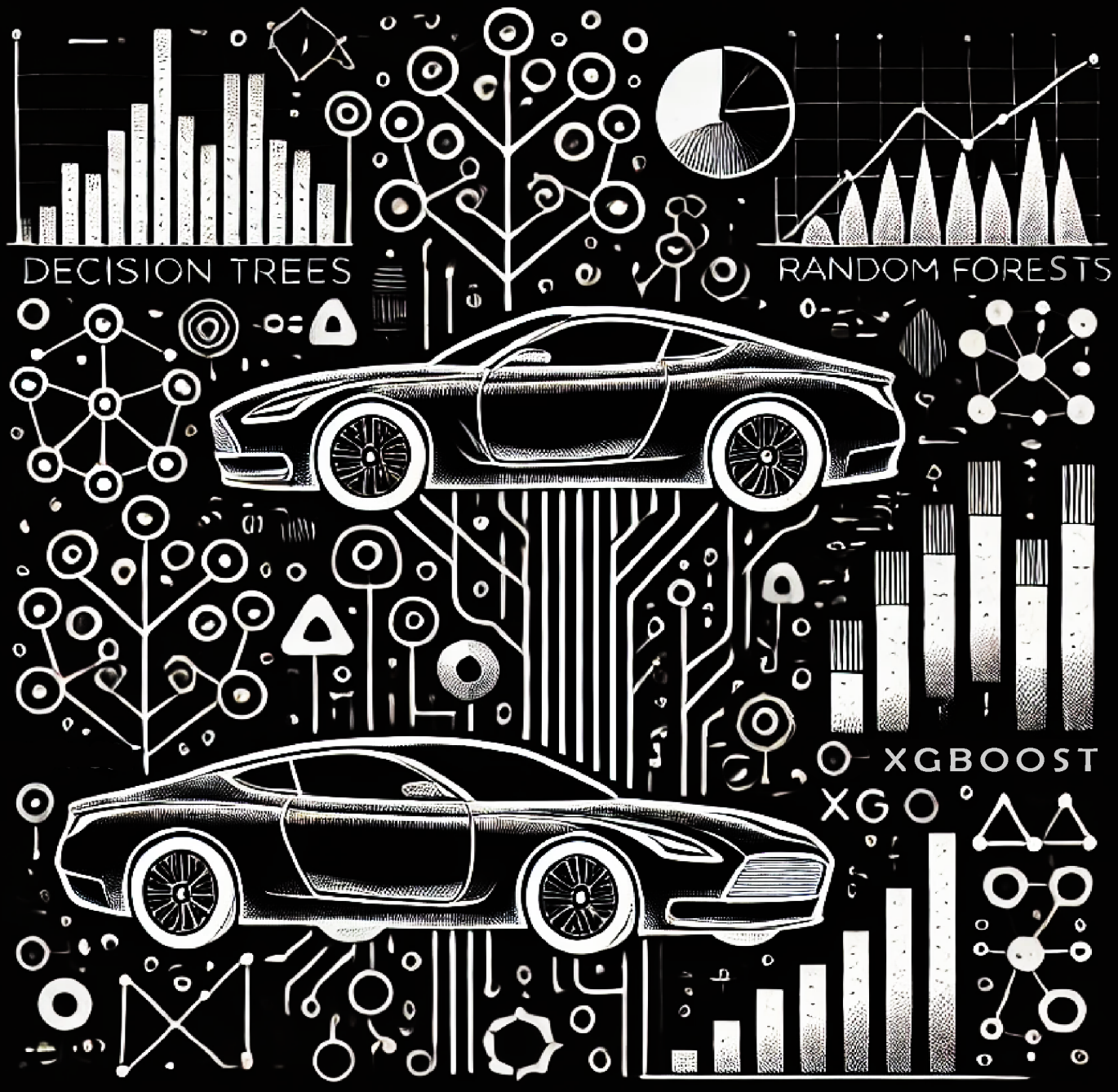
Random Forest y XGBoost

Práctica 1 de el módulo de Machine Learning

Laura Rodríguez Ropero



Máster en Big Data, Data Science e IA

Random Forest y XGBoost

Práctica 1 de el módulo de Machine Learning

por

Laura Rodríguez Ropero

24/04/2025

Índice

Profesores: Inmaculada Gutiérrez, Daniel Gómez, y Juan Antonio Guevara
Facultad: Facultad de Estudios Estadísticos
Universidad: Universidad Complutense de Madrid
Máster: Big Data, Data Science e Inteligencia Artificial

1

Análisis y depuración de la base de datos

1.1. Enunciado

Una empresa dedicada a la venta de coches usados se enfrenta al desafío de determinar el color óptimo para repintar vehículos que llegan en condiciones deficientes. Tras evaluar las opciones, decide limitarse a los colores blanco y negro, por ser los más comunes en el mercado. Para decidir el color de repintado de cada coche, la empresa planea desarrollar un modelo predictivo que, basándose en las características de los vehículos en el mercado de segunda mano, determine si originalmente eran blancos o negros. La base de datos disponible incluye las siguientes variables independientes:

- **Price**: Precio del vehículo.
- **Levy**: Impuesto o recargo asociado al automóvil.
- **Manufacturer**: Marca del automóvil.
- **Prod. year**: Año de fabricación.
- **Category**: Tipo de carrocería.
- **Leather interior**: Indica si el vehículo tiene interior de cuero.
- **Fuel type**: Tipo de combustible.
- **Engine volume**: Capacidad del motor.
- **Mileage**: Kilometraje del vehículo.
- **Cylinders**: Número de cilindros del motor.
- **Gear box type**: Tipo de transmisión.
- **Drive wheels**: Tracción del vehículo.
- **Wheel**: Lado del volante.
- **Airbags**: Número de airbags.
- **Color**: Blanco o negro. Se trata de nuestra variable objetivo.

El dataset contiene 4,340 observaciones en total. Las variables incluyen tanto datos numéricos como categóricos. A continuación, se presenta un primer análisis descriptivo de la información, destacando la distribución de cada variable y la posible existencia de valores atípicos o perdidos.

1.2. Análisis exploratorio de datos (EDA)

Column	Non-Null Count	Dtype
Price	4340 non-null	int64
Levy	4340 non-null	object
Manufacturer	4340 non-null	object
Prod. year	4340 non-null	int64
Category	4340 non-null	object
Leather interior	4340 non-null	object
Fuel type	4340 non-null	object
Engine volume	4340 non-null	object

Column	Non-Null Count	Dtype
Mileage	4340 non-null	object
Cylinders	4340 non-null	int64
Gear box type	4340 non-null	object
Drive wheels	4340 non-null	object
Wheel	4340 non-null	object
Color	4340 non-null	object
Airbags	4340 non-null	int64

Como podemos observar no exiten valores nulos en ninguna de las variables. Por otro lado, la frecuencia de la clase Black es de 2327, y de la clase White es de 2013. Siendo esto así vamos a considerar que ambas clases están bien representadas y continuamos con el ejercicio.

Es la hora de estudiar un poco más a fondo nuestros datos. Lo primero que vamos a hacer va a ser establecer Mileage como variable numérica pues al estar terminada en 'km' ha sido asignada automáticamente como categórica:

```
df['Mileage'] = df['Mileage'].str.replace(' km', '', regex=True).astype(int)
df.head()
```

Price	Levy	Manufacturer	Prod. year	Category	Leather interior	Fuel type
39493	891	HYUNDAI	2016	Jeep	Yes	Diesel
1803	761	TOYOTA	2010	Hatchback	Yes	Hybrid
1098	394	TOYOTA	2014	Sedan	Yes	Hybrid
941	1053	MERCEDES-BENZ	2014	Sedan	Yes	Diesel
1019	1055	LEXUS	2013	Jeep	Yes	Hybrid

Engine volume	Mileage	Cylinders	Gear box type	Drive wheels	Wheel	Color	Airbags
2	160931	4	Automatic	Front	Left wheel	White	4
1.8	258909	4	Automatic	Front	Left wheel	White	12
2.5	398069	4	Automatic	Front	Left wheel	Black	12
3.5	184467	6	Automatic	Rear	Left wheel	White	12
3.5	138038	6	Automatic	Front	Left wheel	White	12

Podemos ver que las variables Levy y Engine volume también habían sido mal asignadas como object. (En el caso de Engine volume vamos a descartar las versiones Turbo de un mismo motor para simplificar.)

```
df["Levy"] = pd.to_numeric(df["Levy"], errors="coerce")
df["Engine volume"] = df["Engine volume"].str.extract(r"([\d.]+)").astype(float)
```

Por último vamos a separar las nuevas variables numéricas de las categóricas y a trasformar estas últimas mediante la regla One-Hot Encoding. También vamos a dividir los datos en Train y Test.

```
nominales = ['Manufacturer', 'Category', 'Leather interior', 'Fuel type', 'Gear box type', '
    Drive wheels', 'Wheel']
df = pd.get_dummies(df, columns=nominales, drop_first=True)
X = df.drop(columns=['Color'])
y = df['Color']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

La frecuencia de cada clase en train es: Black 0.542915 White 0.457085

La frecuencia de cada clase en test es: Black 0.509217 White 0.490783

Por tanto, la distribución de la variable depediente es similar en ambos sets.

2

Búsqueda paramétrica y selección del mejor árbol de decisión

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.tree import DecisionTreeClassifier, export_text, plot_tree
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_curve, auc
```

2.1. Búsqueda de hiperparámetros

Los hiperparámetros a optimizar son los siguientes:

- Profundidad máxima (**max_depth**).
- Mínimo de muestras en hojas (**min_samples_leaf**).
- Mínimo de muestras para dividir un nodo (**min_samples_split**).
- Criterio de partición (**criterion**: gini, entropy).

Para cada combinación, se utiliza validación cruzada de 4 folds y se mide la media de *Accuracy*, *Precisión*, *Recall* y *F1-score*, seleccionando finalmente el modelo con mejor balance en dichas métricas.

```
params = {'max_depth': [2, 3, 5, 10, 20],
          'min_samples_leaf': [1, 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']
grid_search = GridSearchCV(estimator=DecisionTreeClassifier(), param_grid=params,
                           cv=4, scoring=scoring_metrics, refit='accuracy')
grid_search.fit(X_train, y_train)
```

```
print(grid_search.best_estimator_)
```

Output: (criterion='entropy', max_depth=20, min_samples_leaf=2, min_samples_split=5)

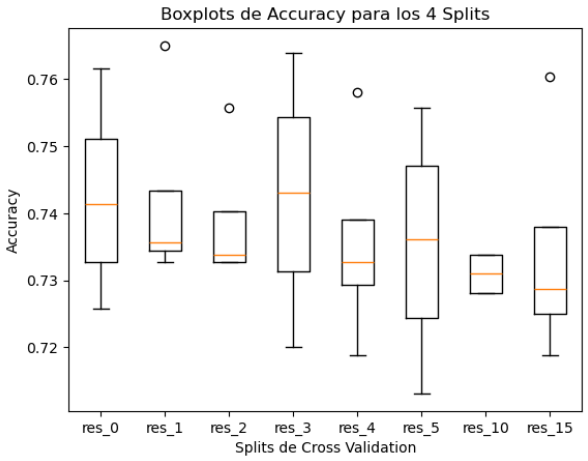
En vista a este primer resultado vamos a realizar un segundo GridSearch un tanto más fino.

```
params = {'max_depth': [10, 15, 20, 25, 30],
          'min_samples_leaf': [1, 2, 3, 5, 10],
          'min_samples_split': [2, 3, 4, 5, 6, 7],
          'criterion': ["gini", "entropy"]}
scoring_metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
grid_search = GridSearchCV(estimator=DecisionTreeClassifier(), param_grid=params,
                           cv=4, scoring=scoring_metrics, refit='accuracy')
grid_search.fit(X_train, y_train)
```

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

res_0 = results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy', '
    split3_test_accuracy']].iloc[0]
res_1 = results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy', '
    split3_test_accuracy']].iloc[1]
res_2 = results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy', '
    split3_test_accuracy']].iloc[2]
res_3 = results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy', '
    split3_test_accuracy']].iloc[3]
res_4 = results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy', '
    split3_test_accuracy']].iloc[4]
res_5 = results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy', '
    split3_test_accuracy']].iloc[5]
res_10 = results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy', '
    split3_test_accuracy']].iloc[10]
res_15 = results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy', '
    split3_test_accuracy']].iloc[15]
```

```
plt.boxplot([res_0.values,res_1.values,res_2.values,res_3.values,res_4.values,res_5.values,
    res_10.values,res_15.values], labels = ['res_0','res_1','res_2','res_3','res_4','res_5','
    res_10','res_15'])
plt.title('Boxplots de Accuracy para los 4 Splits')
plt.xlabel('Splits de Cross Validation')
plt.ylabel('Accuracy')
plt.show()
```



En vista a los boxplot impresos vamos a quedarnos con la combinación de parámetros del res_10 porque a simple vista se puede apreciar que refleja el modelo más robusto de los 8, y la diferencia en el accuracy es despleciable.

2.2. Estructura del árbol elegido:

En vista a la elección que se ha tomado, procedemos a ajustar este modelo con todo el conjunto de entrenamiento, y a hacer predicciones.

```
params = results['params'].iloc[10]
best_model = DecisionTreeClassifier(**params)
best_model.fit(X_train, y_train)
y_train_pred = best_model.predict(X_train)
y_test_pred = best_model.predict(X_test)
```

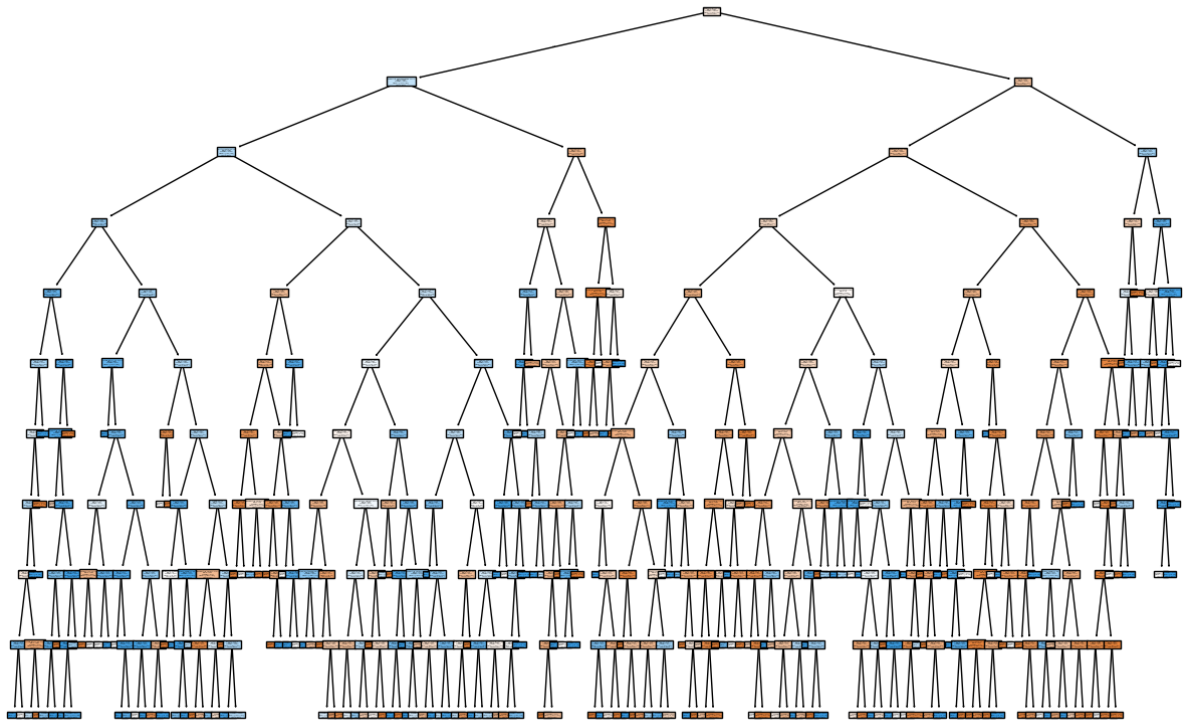
En este momento, se puede obtener información detallada de cada nodo y las reglas de decisión.

```
tree_rules = export_text(best_model, feature_names=list(X.columns),show_weights=True)
print(tree_rules)
```

Output:

|— Engine volume <= 2.30
| |— Manufacturer_MERCEDES-BENZ <= 0.50
| | |— Fuel type_Petrol <= 0.50
| | | |— Price <= 2116.50
| | | | |— Airbags <= 7.00
| | | | | |— Mileage <= 190438.50
| | | | | | |— Levy <= 615.50
| | | | | | | |— Levy <= 585.50
| | | | | | | | |— Mileage <= 108000.00
| | | | | | | | | |— Price <= 55.00
| | | | | | | | | | |— weights: [0.00, 5.00] class: White
| | | | | | | | | |— Price > 55.00
| | | | | | | | | | |— weights: [2.00, 2.00] class: Black
...

Este árbol, con criterio “gini”, profundidad máxima = 10, min_samples_leaf = 2 y min_samples_split = 6, se puede ver de la siguiente manera:



2.3. Importancia de las variables

A continuación, vamos a estudiar la importancia, o valor predictivo, de cada variable en el modelo.

```
print(pd.DataFrame({'nombre': best_model.feature_names_in_, 'importancia': best_model.
    feature_importances_}))
```

nombre	importancia
Price	0.209846
Levy	0.131902
Prod. year	0.069679
Engine volume	0.181245
Mileage	0.214719
Cylinders	0.008395
Airbags	0.033326
Manufacturer_LEXUS	0.000000
Manufacturer_M-BENZ	0.037784
Manufacturer_TOYOTA	0.016811

nombre	importancia
Category_Jeep	0.007272
Category_Sedan	0.002083
Leather interior_Yes	0.004476
Fuel type_Hybrid	0.031479
Fuel type_Petrol	0.022474
Gear box type_Tiptronic	0.005203
Drive wheels_Front	0.012399
Drive wheels_Rear	0.008479
Wheel_Right-hand drive	0.002428

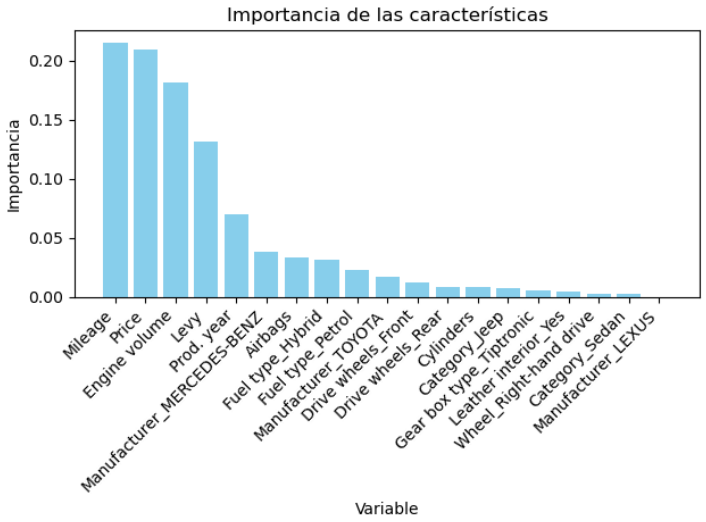
De manera general Price (0.209846) y Mileage (0.214719) aparecen como las más relevantes, seguidas muy de cerca por Engine volume (0.181245). También destaca Levy (0.131902) con un valor notable. Estas magnitudes más altas indican que el modelo se apoya fuertemente en estas variables para realizar las divisiones o la predicción.

Otras variables relevantes son también Prod. year (0.069679) y varios indicadores relacionados con el fabricante y el tipo de combustible (p. ej., “Fuel type_Hybrid (0.031479)”, “Manufacturer_MERCEDES-BENZ (0.037784)”), contribuyendo pero con menor peso que las primeras.

En la zona inferior de la tabla aparecen variables como “Wheel_Right-hand drive (0.002428)”, “Manufacturer_LEXUS (0.000000)” o “Category_Sedan” (0.002082)”, con valores más bajos. Esto sugiere que, en este modelo, su influencia en la decisión final es relativamente limitada.

En resumen, esta tabla sugiere que, en el modelo concreto, el precio, el kilometraje recorrido, el volumen del motor y la tasa/concepto “Levy” son factores fundamentales. El resto de variables aporta información adicional pero, según el modelo, tienen un impacto menor en la predicción o clasificación que se esté llevando a cabo.

```
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='skyblue')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



2.4. Métricas de rendimiento:

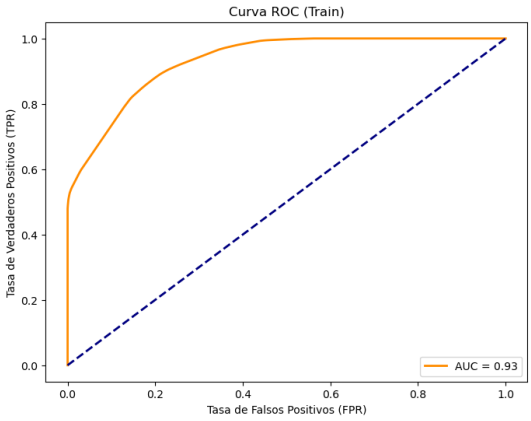
```
conf_matrix = confusion_matrix(y_train, y_train_pred)
print("Matriz de Confusión (Train):")
print(conf_matrix)
print("\nMedidas de Desempeño (Test):")
print(classification_report(y_train, y_train_pred))
```

```
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)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'AUC = {roc_auc:.2f}')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('Tasa de Falsos Positivos (FPR)')
plt.ylabel('Tasa de Verdaderos Positivos (TPR)')
plt.title('Curva ROC (Train)')
plt.legend(loc="lower right")
plt.show()
```

Matriz de Confusión (Train)

	Pred: Black	Pred: White
True: Black	1605	280
True: White	281	1306

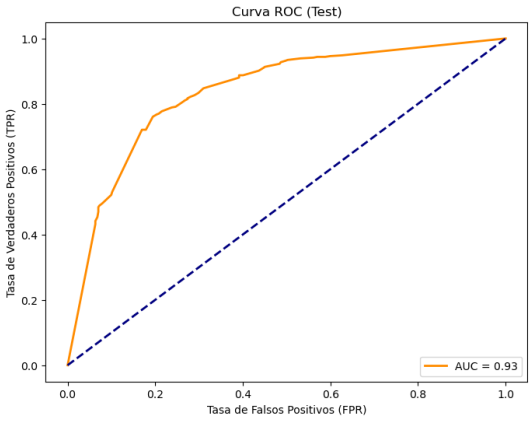


Medidas de Desempeño (Test)

	Prec.	Recall	F1	Support
Black	0.85	0.85	0.85	1885
White	0.82	0.82	0.82	1587
Accuracy			0.84	3472
Macro avg	0.84	0.84	0.84	3472
Weighted avg	0.84	0.84	0.84	3472

Matriz de Confusión (Train)

	Pred: Black	Pred: White
True: Black	347	95
True: White	95	331



Medidas de Desempeño (Test)

	Prec.	Recall	F1	Support
Black	0.79	0.79	0.79	442
White	0.78	0.78	0.78	426
Accuracy			0.78	868
Macro avg	0.78	0.78	0.78	868
Weighted avg	0.78	0.78	0.78	868

En conclusión, el train tiene mejor desempeño con un accuracy de 0.84 en comparación con 0.78 en test. Sin embargo, ambos tienen un AUC de 0.93, lo que indica que el modelo discrimina bien entre las clases.

El modelo no parece sobreajustado, ya que los valores en prueba y entrenamiento son similares.

3

Búsqueda paramétrica para Random Forest y XGBoost según Accuracy

3.1. Configuración y comparación de modelos

Para **Random Forest**, se exploraron diferentes configuraciones ajustando los valores de:

- `n_estimators`: Número de árboles (p. ej., 50, 100, 200).
- `max_depth`: Profundidad máxima de cada árbol (p. ej., 5, 10, 20).
- `bootstrap`: Uso de muestreo con reemplazo o no (`True` o `False`).
- `min_samples_leaf`: Mínimo de muestras requeridas para formar una hoja (p. ej., 2, 5, 10).
- `min_samples_split`: Mínimo de muestras para dividir un nodo (p. ej., 2, 5, 10).
- `criterion`: Criterio para medir la calidad de la división (`gini`, `entropy` o `log_loss`).

Para **XGBoost**, se evaluaron distintas combinaciones de:

- `booster`: Tipo de modelo base (p. ej., `gbtree`, `gblinear`).
- `n_estimators`: Número de árboles (p. ej., 50, 100, 200).
- `eta`: Tasa de aprendizaje (p. ej., 0.01, 0.1, 0.3).
- `gamma`: Mínima reducción de pérdida requerida para hacer una partición adicional (p. ej., 0.1, 1).
- `max_depth`: Profundidad máxima de cada árbol (p. ej., 5, 10, 15).
- `tree_method`: Método de crecimiento de los árboles (p. ej., `auto`, `hist`, o `approx`).

En ambos casos, se utilizó una validación cruzada de 4 *folds*, evaluando las configuraciones con base en cuatro métricas de interés: `accuracy`, `precision_macro`, `recall_macro` y `f1_macro`. Sin embargo, la métrica objetivo sobre la cual finalmente se refina el modelo (indicada con `refit`) es `accuracy`, lo que significa que se seleccionará la configuración de hiperparámetros que maximice el *accuracy* promedio obtenido durante la validación cruzada.

3.2. Flujo de trabajo para la configuración y evaluación de modelos de Random Forest y XGBoost

Se parte de un modelo sencillo (en el caso del RandomForest, compuesto por un único árbol obtenido en la sección anterior). Este modelo actúa como base para comparar los resultados que se obtendrán tras una búsqueda sistemática de hiperparámetros.

```
RF_model = RandomForestClassifier(
    n_estimators=60, bootstrap=True,
    max_depth=10, min_samples_split=6,
    criterion='gini', min_samples_leaf=2,
    random_state=123)
RF_model.fit(X_train, y_train)
```

```
xgb_classifier = XGBClassifier(booster = '
    gbtree', n_estimators = 200, eta = 0.1,
    gamma = 1, random_state=123, max_depth =
     20, tree_method = 'hist')
xgb_classifier.fit(X_train, y_train)
```

Se especifica un espacio de posibles valores para cada hiperparámetro del modelo. Adicionalmente, se definen las métricas de evaluación que `GridSearchCV` calculará al validar el desempeño de cada combinación. En este caso, se indican `accuracy`, `precision_macro`, `recall_macro` y `f1_macro`. A través de `GridSearchCV`, se recorre la rejilla definida y se entrena cada configuración de parámetros. Se emplea una validación cruzada de *4 folds*, de manera que:

- El conjunto de entrenamiento se divide en 4 bloques (*folds*).
- En cada iteración, se reservan 3 pliegues para entrenar y 1 pliegue para validar.
- Se obtienen métricas medias (y desviaciones) de desempeño.

Por último, se usa `refit='accuracy'` para indicar que, una vez probadas todas las combinaciones, se retome aquella con la mayor `accuracy` promedio, ajustando el modelo final con todos los datos de entrenamiento.

Una vez terminada la búsqueda. Se obtiene el mejor estimador con grid_search_RF.best_estimator_. Y se calculan sus predicciones sobre el conjunto de prueba.

```
params = {
    'n_estimators' : [50, 100, 150, 200],
    'max_depth': [2, 5, 10, 20, 25],
    'bootstrap': [True, False],
    'min_samples_leaf' : [1, 3, 5, 10, 20],
    'min_samples_split': [5, 10, 20, 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)
```

```
params = {
    'booster' : ['gbtree', 'gblinear'],
    'n_estimators': [50, 100, 150, 200,
        250],
    'eta' : [0.01, 0.1, 0.2, 0.3],
    'gamma' : [0, 0.1, 0.5, 1],
    'max_depth': [5, 10, 15, 20, 25],
    'tree_method' : ['auto', 'approx', 'hist
        ']
}
scoring_metrics = ['accuracy', '
    precision_macro', 'recall_macro', '
    f1_macro']
grid_search_XGB = GridSearchCV(estimator=
    XGBClassifier(), param_grid=params, cv
    =4, scoring = scoring_metrics, refit='
    accuracy')
grid_search_XGB.fit(X_train, y_train)
```

```
best_model_RF = grid_search_RF.
    best_estimator_
y_train_pred = best_model_RF.predict(X_train
    )
y_test_pred = best_model_RF.predict(X_test)
```

```
best_model_XGB = grid_search_XGB.
    best_estimator_
y_train_pred = best_model_XGB.predict(
    X_train)
y_test_pred = best_model_XGB.predict(X_test)
```

Output:
RandomForestClassifier(criterion='entropy', max_depth=20, min_samples_split=5, n_estimators=50, random_state=123)
Accuracy para train de: 0.972926267281106
Accuracy para test de: 0.782258064516129

Output:
XGBClassifier(booster='gbtree', n_estimators=50, eta=0.2, gamma=0, max_depth=20, tree_method='approx', ...)
Accuracy para train de: 0.9994239631336406
Accuracy para test de: 0.7764976958525346

Parece haber claros indicios de sobreajuste en ambos modelos. Precedemos a redefinir la grilla.

Los ajustes en la grilla de hiperparámetros del modelo Random Forest tienen como objetivo reducir el sobreajuste limitando la complejidad del modelo. Se ha reducido el rango de max_depth para evitar árboles excesivamente profundos. Además, se han aumentado los valores de min_samples_leaf y min_samples_split con el fin de hacer los árboles más generales y menos sensibles a pequeñas variaciones en los datos. Se ha incrementado el número de n_estimators para mejorar la estabilidad y reducir la varianza del modelo. También se ha incorporado el parámetro max_features, con valores como 'sqrt' y 'log2', para introducir aleatoriedad en las divisiones y minimizar el riesgo de sobreajuste. Finalmente, se ha fijado bootstrap a True para promover la generalización mediante muestreos con reemplazo. Estos ajustes buscan mejorar la capacidad de generalización y reducir el sobreajuste.

En cuanto al modelo XGBoost, los cambios en la grilla de hiperparámetros se centran en mejorar la generalización y mitigar el sobreajuste. Se ha reducido el rango de n_estimators para evitar entrenar un número excesivo de árboles, lo cual podría inducir sobreajuste. Asimismo, se ha limitado la profundidad máxima de los árboles (max_depth) para prevenir una excesiva complejidad. El parámetro eta (tasa de aprendizaje) se ha ajustado a valores más bajos, permitiendo un aprendizaje más controlado y evitando que el modelo se sobreajuste a los datos de entrenamiento. El parámetro gamma también se ha moderado para evitar divisiones demasiado específicas. Además, se han añadido los parámetros reg_alpha y reg_lambda para aplicar regularización, reduciendo así la complejidad del modelo y mitigando el sobreajuste. Por último, se ha fijado el parámetro tree_method a 'hist', un método más eficiente para conjuntos de datos grandes que favorece la generalización.

```
params2 = {
    'n_estimators': [100, 150, 200, 300],
    'max_depth': [2, 5, 8, 10],
    'min_samples_leaf': [20, 40, 60, 80],
    'min_samples_split': [50, 80, 120, 160],
    'max_features': ['sqrt', 'log2'],
    'bootstrap': [True],
    '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)
```

```
params2 = {'n_estimators': [50, 100, 150,
    200], 'max_depth': [2, 5, 8, 10], 'eta':
     [0.001, 0.005, 0.01, 0.05], 'gamma':
    [0, 0.1, 0.2, 0.3],  'reg_alpha': [1,
    10, 50, 100], 'reg_lambda': [1, 10, 50,
    100],
'tree_method': ['hist']
}
scoring_metrics = ['accuracy', '
    precision_macro', 'recall_macro', '
    f1_macro']
grid_search_XGB = GridSearchCV(estimator=
    XGBClassifier(), param_grid=params, cv
    =4, scoring = scoring_metrics, refit='
    accuracy')
grid_search_XGB.fit(X_train, y_train)
```

```
best_model_RF = grid_search_RF.
    best_estimator_
y_train_pred = best_model_RF.predict(X_train
    )
y_test_pred = best_model_RF.predict(X_test)
```

Output:
RandomForestClassifier(criterion='entropy', max_depth=8, min_samples_leaf=20, min_samples_split=50, n_estimators=300, random_state=123)
Accuracy para train de: 0.7546082949308756
Accuracy para test de: 0.6970046082949308

```
best_model_XGB = grid_search_XGB.
    best_estimator_
y_train_pred = best_model_XGB.predict(
    X_train)
y_test_pred = best_model_XGB.predict(X_test)
```

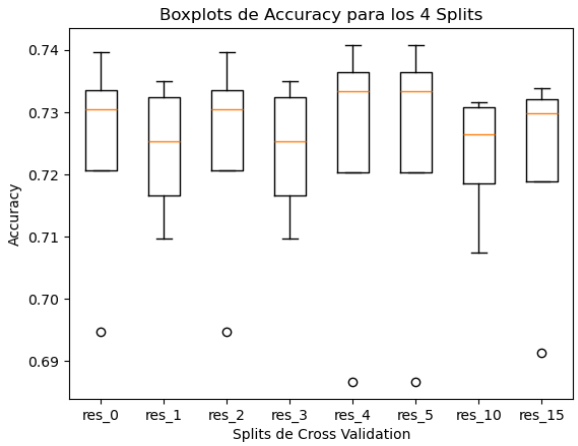
Output:
XGBClassifier(booster='gbtree', n_estimators=100, eta=0.05, gamma=0, max_depth=10, reg_alpha=1, reg_lambda=1, tree_method='hist', ...)
Accuracy para train de: 0.9398041474654378
Accuracy para test de: 0.7626728110599078

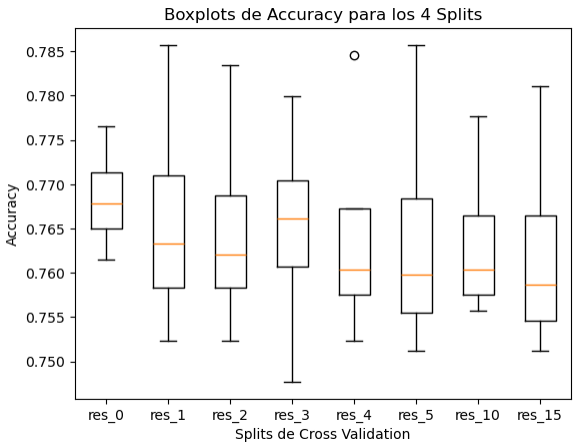
```
results = pd.DataFrame(grid_search_RF.
    cv_results_)
```

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

```
sorted_results = results.sort_values(by='mean_test_accuracy', ascending=True)
res_0 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy'
    , 'split3_test_accuracy']].iloc[0]
res_1 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy'
    , 'split3_test_accuracy']].iloc[1]
```

```
res_2 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy'
    , 'split3_test_accuracy']].iloc[2]
res_3 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy'
    , 'split3_test_accuracy']].iloc[3]
res_4 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy'
    , 'split3_test_accuracy']].iloc[4]
res_5 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy'
    , 'split3_test_accuracy']].iloc[5]
res_10 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy
    ', 'split3_test_accuracy']].iloc[10]
res_15 = sorted_results[['split0_test_accuracy', 'split1_test_accuracy','split2_test_accuracy
    ', 'split3_test_accuracy']].iloc[15]
plt.boxplot([res_0.values,res_1.values,res_2.values,res_3.values,res_4.values,res_5.values,
    res_10.values,res_15.values], labels = ['res_0','res_1','res_2','res_3','res_4','res_5','
    res_10','res_15'])
plt.show()
```





Voy a elegir el modelo 15 pues es el que parece más robusto.

Me quedo con el modelo 0, por el mismo criterio de robustez.

```
random_f_def = RandomForestClassifier(**
    sorted_results['params'].iloc[15],
    random_state=123)
modelo_RF = random_f_def.fit(X_train,
    y_train)
```

```
xgb_def = XGBClassifier(**sorted_results['
    params'].iloc[0],random_state=123)
modelo_XGB = xgb_def.fit(X_train, y_train)
```

3.3. Resultados de los mejores modelos

Finalmente, vamos a evaluar nuestros modelos.

```
y_train_pred = modelo_RF.predict(X_train)
y_test_pred = modelo_RF.predict(X_test)
print(classification_report(y_test,
    y_test_pred))
```

```
y_train_pred_xgb = modelo_XGB.predict(
    X_train)
y_test_pred_xgb = modelo_XGB.predict(X_test)
print(classification_report(y_test,
    y_test_pred_xgb))
```

Se tiene un accuracy para train de: 0.7730414746543779. Y un accuracy para test de: 0.717741935483871

Class	P	R	F1	S
Black	0.75	0.73	0.73	467
White	0.69	0.71	0.70	401
Acc			0.72	868
M-avg	0.72	0.72	0.72	868
W-avg	0.72	0.72	0.72	868

Se tiene un accuracy para train de: 0.9032258064516129. Y un accuracy para test de: 0.7695852534562212.

Class	P	R	F1	S
Black	0.78	0.79	0.79	467
White	0.75	0.74	0.75	401
Acc			0.77	868
M-avg	0.77	0.77	0.77	868
W-avg	0.77	0.77	0.77	868

4

Proceso comparativo de los modelos y conclusiones

4.1. Comparativa entre Árbol de Decisión, Random Forest y XGBoost

A lo largo del proceso se entrenaron y ajustaron tres modelos principales: Árbol de Decisión, Random Forest y XGBoost. La comparación se realizó en función de métricas clásicas de clasificación: accuracy, precision, recall y F1-score, evaluadas sobre el conjunto de test.

- **Árbol de Decisión:** Obtuvo un accuracy del 78% en test, y valores similares en precision, recall y F1-score (todos de 0.78). Además, mostró un rendimiento muy equilibrado entre train y test (84% vs. 78%), lo que indica una buena capacidad de generalización. Su principal ventaja es la interpretabilidad, ya que permite explicar fácilmente sus decisiones mediante reglas.
- **Random Forest:** A pesar de ser un modelo de ensamble que suele superar a los árboles individuales, en este caso su configuración priorizó la generalización y simplicidad, lo que redujo su desempeño. Su accuracy fue de 71.8% en test y 77.3% en entrenamiento, con un F1-score de 0.72. Aunque no presenta overfitting, su rendimiento final quedó por debajo del árbol de decisión..
- **XGBoost:** Presentó un accuracy del 76.9% y un F1-score de 0.77 en test. Sin embargo, su accuracy en entrenamiento fue de 90.3%, una diferencia significativa que sugiere sobreajuste. A pesar de incluir regularización (reg_alpha, reg_lambda) y una tasa de aprendizaje baja (eta = 0.05), el modelo aún mostró tendencia a memorizar el entrenamiento.

En resumen, el árbol de decisión fue el más equilibrado, con el mejor rendimiento en test y sin indicios de sobreajuste. XGBoost mostró buen desempeño, pero con señales claras de overfitting, mientras que Random Forest fue el menos eficaz en este caso.

4.2. Elección final y justificación

Dado que el objetivo es obtener un modelo preciso y estable en nuevos datos, el Árbol de Decisión es el modelo más adecuado. No solo obtuvo el accuracy más alto en test (78%), sino que también presentó un comportamiento consistente entre entrenamiento y prueba, lo que indica una buena capacidad de generalización. Además, su estructura clara lo hace ideal para contextos donde la interpretabilidad es importante.

Por otro lado, aunque XGBoost mostró un F1-score alto (0.77), su elevado rendimiento en entrenamiento frente al test (90% vs. 77%) indica una tendencia al sobreajuste, por lo que conviene tomarlo con cautela si el modelo va a usarse en producción. Random Forest, pese a su potencial, no logró superar al árbol ni en precisión ni en balance general.

4.3. Posibles extensiones y siguientes pasos

Para futuras investigaciones y mejoras del modelo, se proponen las siguientes líneas de trabajo:

- Optimizar la búsqueda de hiperparámetros mediante Bayesian Optimization, lo que podría reducir significativamente el tiempo computacional respecto a Grid Search.
- Explorar nuevos algoritmos, como redes neuronales profundas, especialmente si se dispone de mayor volumen de datos.
- Evaluar los modelos con datos completamente nuevos o en un entorno productivo real, para comprobar su capacidad de generalización.
- Realizar ingeniería de características más avanzada, generando variables nuevas o combinaciones de las existentes que capturen mejor las relaciones entre atributos y la variable objetivo.