Análisis de Modelos de Bagging y Stacking: Aplicación Práctica

Minería de datos y Modelización Predictiva

Bartolomé Mestre Fons

Miércoles, 5 de Febrero de 2025

Minería de datos y Modelización Predictiva
2024-2025
UCM - Universidad Complutense de Madrid

Índice

1. Objetivos **3**

2. Introducción **3**

3. Depuración de los datos **4**
3.1. Outliers y Valores *Missing* 4
3.2. Transformación de las variables categóricas y numéricas 5
3.3. Estandarización 6
3.4. Selección de Variables significativas 6

4. SVM **9**
4.1. Bagging 11

5. Modelo de *Stacking* **14**

6. Conclusiones **20**

7. Bibliografía **20**

1. Objetivos

En el siguiente documento se desarrolla un modelo predictivo que, basándose en las características de los vehículos en el mercado de segunda mano, determina si originalmente eran blancos o negros. Tras depurar los datos (imputando valores nulos, eliminando *outliers* y normalizando los datos mediante técnicas de *feature engineering*), se dividirá el conjunto en una parte dedicada al entreno del modelo y otra para su testeo. Adicionalmente, se realiza una selección de variables en función de su Explicabilidad Normalizada con el fin de simplificar el modelo. Acto seguido, se emplean técnicas Bayesianas de optimización hiperparamétricas para el modelo de clasificación SVM con 2 *kernels* diferentes: lineal y RBF. Se entrenan ambos modelos y se compara su rendimiento con el fin de obtener un modelo mejorado utilizando la técnica de *bagging*. Además, se realiza un modelo de *Stacking* basándose en los 3 modelos anteriores más 3 adicionales: *RandomForestClassifier*, *KNeighborsClassifier* y *XGBoostClassifier*. Finalmente, se compara el rendimiento de todos los modelos tratados en conjunto. Se adjuntará a lo largo del documento aquellos códigos relevantes para la reproducción de los resultados.

2. Introducción

Se trabajará con 14 variables explicativas más 1 variable objetivo sobre 4340 registros de coches de segunda mano. El conjunto de datos se ha descargado desde la siguiente fuente, filtrado previamente según las indicaciones del enunciado del problema. Se adjunta a continuación un breve análisis:

- **Price**: precio del vehículo. Variable contínua.
- **Levy**: precio del embargo del vehículo. Variable contínua. Aquellos valores nulos se identifican como "-".
- **Manufacturer**: fabricante del vehículo. Variable categórica, con valores HYUNDAI, TOYOTA, MERCEDES-BENZ o LEXUS.
- **Prod. year**: año de fabricación del vehículo. Variable numérica discreta.
- **Category**: categoría del vehículo. Variable categórica, con valores Jeep, Hatchback o Sedan.
- **Leather Interior**: variable dicotómica que indica si el vehículo posee interior de cuero o no. Variable categórica, con valores Yes o No.
- **Fuel type**: combustible del vehículo. Variable categórica, con valores Diesel, Hybrid o Petrol.
- **Engine volume**: tamaño del motor del vehículo. Variable categórica de valores numéricos, algunos de ellos con la terminación "Turbo".
- **Mileage**: número de kilómetros recorridos por el vehículo. Variable categórica de valores numéricos, todos ellos con la terminación "km" .
- **Cylinders**: número de cilindros del vehículo. Variable discreta, con 10 valores únicos.
- **Gear box type**: tipo de caja de cambios del vehículo. Variable categórica, con valores Automatic o Tiptronic.
- **Drive wheels**: tracción del vehículo. Variable categórica, con valores Front, Rear o 4x4.
- **Wheel**: ubicación del volante del vehículo. Variable categórica, con valores Left wheel o Right-hand drive.
- **Color**: color del vehículo. Variable categórica, con valores White o Black.
- **Airbags**: número de airbags del vehículo. Variable numérica discreta, con hasta 16 valores únicos.

Dicho esto, se importan las librerías de las que se vaya a hacer uso a lo largo del documento:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold
from sklearn.feature_selection import SelectKBest, f_classif, RFECV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import BaggingClassifier, StackingClassifier, RandomForestClassifier
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, roc_curve, auc
try:
  from skopt import BayesSearchCV
  from skopt.space import Integer, Real, Categorical
except:
  !pip install scikit-optimize
  from skopt import BayesSearchCV
  from skopt.space import Integer, Real, Categorical

from xgboost import XGBClassifier
```

Se ha optado por utilizar *BayesSearchCV* en sustitución de *GridSearchCV* al ser el primero más eficiente (el último prueba todas las combinaciones posibles dadas, mientras que el primero predice de antemano qué combinaciones de hiperparámetros podrían dar mejores resultados, resultando en un menor número de iteraciones necesarias). Para ello, se requiere la instalación de la librería scikit-optimize. Se lee a continuación el documento que contiene los datos:

```
df = pd.read_excel('datos_tarea25.xlsx')
```

Una vez cargados, se procede a la depuración del dataset.

3. Depuración de los datos

3.1. Outliers y Valores *Missing*

Como se había mencionado anteriormente, la variable Levy posee valores nulos representados de la forma "-". Por ello, se convierte la variable a numérica, tal que para aquellos registros que posean "-" se devolverá un valor NaN:

```
df['Levy'] = pd.to_numeric(df['Levy'], errors='coerce')
```

Ahora, se puede analizar el número de valores nulos de todas las columnas:

```
df.isna().sum()
```

El anterior código devuelve lo siguiente:

```
Price                0
Levy               644
Manufacturer         0
Prod. year           0
Category             0
Leather interior     0
Fuel type            0
Engine volume        0
Mileage              0
Cylinders            0
Gear box type        0
Drive wheels         0
Wheel                0
Color                0
Airbags              0
dtype: int64
```

Se identifican 644 valores nulos en la variable Levy, por lo que se procede a su imputación por la mediana dado que la desviación de los datos respecto a ella no es elevada (parte de los modelos posteriores no funcionan correctamente con valores *missing* en el dataset). Esto se justifica mediante el siguiente gráfico boxplot de la variable Levy (junto a la de Price):

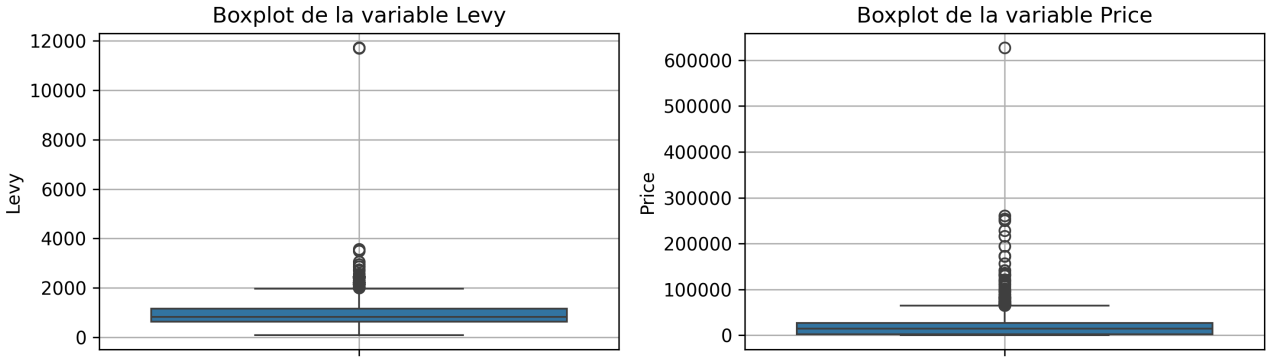

Figura 1: Boxplots de las variables Levy y Price.

Se observan outliers aislados en ambas variables que pueden conducir a imprecisiones posteriores a la hora de entrenar los modelos con estos datos (el modelo puede ajustar sus parámetros a estos valores que caen fuera de la normalidad), por lo que se procede a su eliminación. Así, se decide eliminar aquellos registros cuyas variables Levy y Price se hallen por encima del 99 % restante:

```
limit_price = df_cleaned['Price'].quantile(0.99)
levy_price = df_cleaned['Levy'].quantile(0.99)
df_cleaned = df_cleaned[df_cleaned['Price'] <= limit_price]
df_cleaned = df_cleaned[df_cleaned['Levy'] <= levy_price]
```

Ahora, ya se puede realizar la imputación por la mediana de los valores *missing* en la variable Levy:

```
df['Levy'].fillna(df['Levy'].median(), inplace=True)
```

Es importante aquí mencionar que se ha escogido realizar la imputación por la mediana y no por la media ya que el 20.4 % de los datos se hallan concentrados en los percentiles 0.6 y 0.4, un intervalo pequeño alrededor de la mediana (percentil 0.5) que indica una aglomeración acusada de los datos alrededor de ella. Se halla fácilmente como sigue:

```
total_dim = df.shape[0]
cleaned_dim = df[
(df['Levy'] <= df['Levy'].quantile(0.60))
&
(df['Levy'] >= df['Levy'].quantile(0.40))
].shape[0]
cleaned_dim/total_dim
```

Adicionalmente, se eliminan aquellas filas duplicadas al ser una fuente de *overfitting* para los futuros modelos (cerca del 35 % del dataset total, hallado con df.duplicated().sum()):

```
df = df.drop_duplicates()
```

Con esto, los datos depurados se guardan como una copia para futuros trabajos:

```
df_cleaned = df.copy()
```

3.2. Transformación de las variables categóricas y numéricas

La variable objectivo es Color, una variable categórica con valores Black o White. Se binariza dicho atributo pasándolo a binario, tal que se tendrá 1 si el color es White y 0 si es Black:

```
df_cleaned['Color'] = df_cleaned['Color'].apply(lambda x : 0 if x == 'Black' else 1)
```

Ahora, se procede de la misma manera con el resto de variables categóricas. Puesto que algunas únicamente poseen 2 valores distintos, se evita la colinearidad si directamente en vez de crear 2 atributos separados que puedan tomar valores de 0 o 1 se junta todo en un mismo atributo:

```
df_cleaned['Leather interior'] = pd.Categorical(df_cleaned['Leather interior']).codes
df_cleaned['Wheel'] = pd.Categorical(df_cleaned['Wheel']).codes
df_cleaned['Gear box type'] = pd.Categorical(df_cleaned['Gear box type']).codes
```

Finalmente, se binariza el resto, eliminando la primera categoría con drop_first = True para reducir los efectos de colinearidad:

```
df_dummies = pd.get_dummies(df_cleaned,
                            columns=['Manufacturer', 'Category', 'Drive wheels', 'Fuel type'],
                            drop_first=True, dtype = 'int')
```

Por otra parte, se corrige la variable Mileage para que tenga el formato numérico correcto:

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

En cuanto a la variable Engine volume, algunos registros terminan en Turbo, imposibilitando su tratamiento como datos numéricos. Así, en un nuevo atributo binario se indica si el vehículo posee dicha especificación. Además, se elimina cualquier referencia a dicha característica en la variable Engine volume original:

```
df_dummies['Turbo'] = df_dummies['Engine volume'].astype(str).apply(
    lambda x: 1 if x.endswith(' Turbo') else 0
)
df_dummies['Engine volume'] = df_dummies['Engine volume'].astype(str).apply(
    lambda x: x.replace(' Turbo', '')
)
```

3.3. Estandarización

Para el correcto funcionamiento de los futuros modelos de *machine learning* a aplicar, es recomendable la estandarización de las variables numéricas explicativas. Para ello, se toman 2 estrategias:

- ***StandardScaler***: normaliza los datos para que posean desviación estándar 1 y media 0. Orientada a aquellas variables continuas que muestren un comportamiento gaussiano.
- ***MinMaxScaler***: normaliza los datos con respecto a la cota mínima y máxima. Orientada a aquellas variables discretas de las que se conozca dichas cotas.

Así, se procede a normalizar las diferentes variables numéricas en función de su tipología:

```
discrete_variables = ['Cylinders', 'Airbags']
continuous_variables = ['Price', 'Levy', 'Mileage', 'Engine volume']
minmax_scaler = MinMaxScaler()
standard_scaler = StandardScaler()
df_dummies[discrete_variables] = minmax_scaler.fit_transform(df_dummies[discrete_variables])
df_dummies[continuous_variables] = standard_scaler.fit_transform(df_dummies[continuous_variables])
```

3.4. Selección de Variables significativas

En primer lugar, se estudía la Matriz de Correlación con el fin de adquirir una idea sobre cuán de bien las variables explican la variable objetivo:

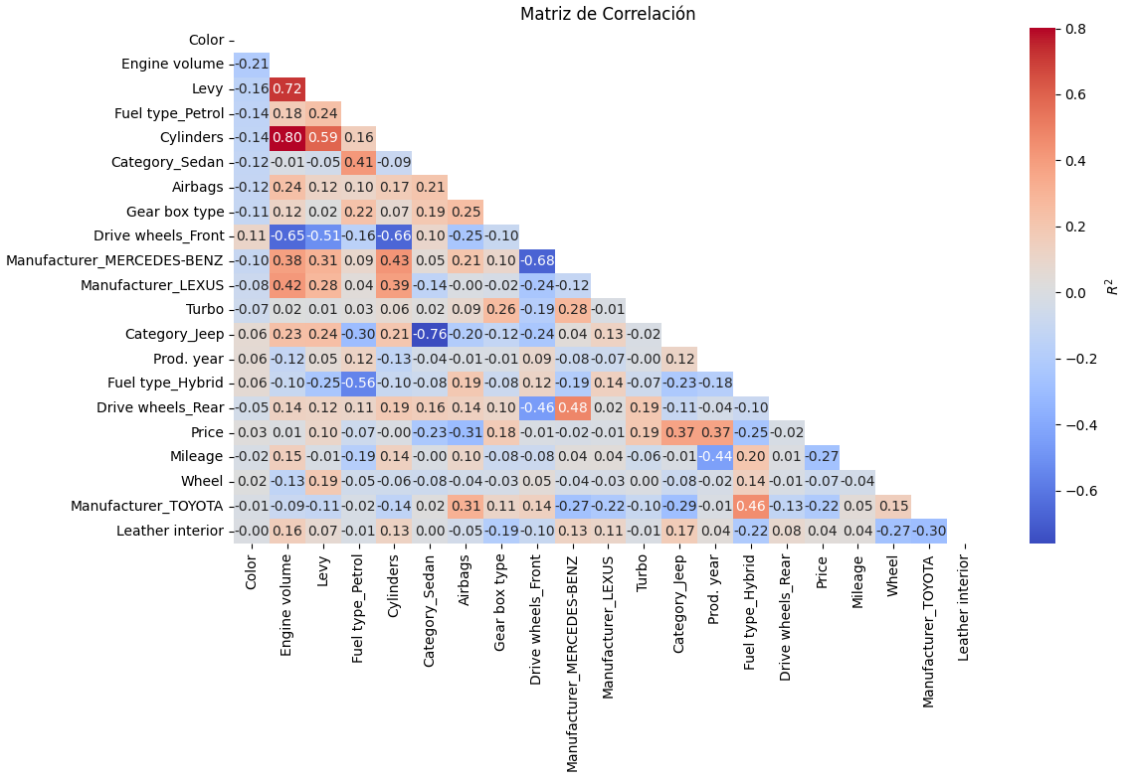

Figura 2: Matriz de Correlación. En la primera columna se identifican las correlaciones entre la variable objetivo y el resto de variables.

Se observa en la imagen superior bajos valores de correlación, por lo que la explicabilidad será pobre y se adelantan resultados no muy precisos. En este punto, se recomienda la creación de variables alternativas y transformarlas mediante técnicas de *feature engineering* con el fin de aumentar la explicabilidad (no se procede a ello para no alargar la extensión del documento excesivamente). Ahora bien, para reducir su complejidad, se representa gráficamente la explicabilidad de cada variable sobre la variable objetivo (tomando las 8 mejores):

```
x = df_dummies.drop(columns = ['Color'])
y = df_dummies['Color']

selector = SelectKBest(score_func=f_classif, k=8)
selector.fit(x, y)

sns.barplot(
    x = selector.scores_ / selector.scores_.sum(),
    y = x.columns,
    palette = 'coolwarm',
    hue = selector.scores_ / selector.scores_.sum()
)
plt.grid()
plt.xlabel('Significancia')
plt.ylabel('Variables Explicativas')
plt.legend([],[], frameon=False)
plt.title('Significancia Normalizada por Variable Explicativa')
plt.show()
```

La explicabilidad normalizada toma la siguiente forma:

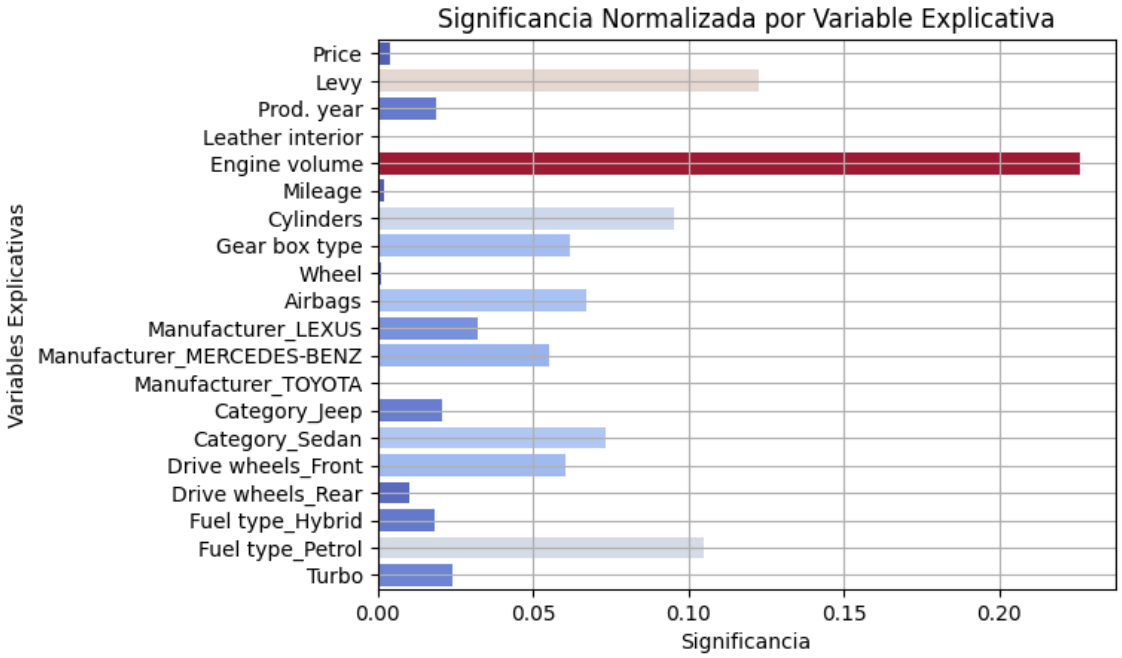

Figura 3: Significancia Normalizada por variable explicativa.

Engine volume es la variable que mayor significancia posee de entre el resto, seguida por Levy. De entre todas, las 8 principales son: Levy, Engine Volume, Cyliders, Gear box type, Airbags, Category_Sedan, Drive wheels_Front y Fuel type_Petrol. Así, las que mayor significancia aportan se escogen del dataset:

```
x_cut = x[x.columns[selector.get_support()].tolist()]
```

A continuación, nuevamente se filtra el número de variables relevantes mediante la aplicación del método RFECV (*Recursive Feature Elimination Cross Validation*), método por el cual se eliminan variables recursivamente en base a si mejoran la Accuracy. Como prueba, se toma el modelo RandomForestClassifier para escoger qué variables eliminar y cuales no, con validación cruzada estratificada (tal que cada sección posea el mismo número de cada clase de la variable objetivo):

```
kfold = StratifiedKFold(n_splits = 10, shuffle = True, random_state = 1234)
estimator = RandomForestClassifier(n_estimators=50, random_state=1234)
selector = RFECV(estimator, step=1, cv=kfold, scoring="accuracy", n_jobs=-1)
selector = selector.fit(x_cut, y)

mean_scores = selector.cv_results_['mean_test_score']

skf = pd.DataFrame(zip(selector.ranking_, mean_scores, x_cut.columns),
                   columns=['Importancia', 'Score', 'Variable'])
skf = skf.sort_values('Importancia')

fig, ax = plt.subplots(figsize=(10, 6))
ax.barh(y=skf["Variable"], width=skf["Score"])
ax.set_xlabel('Score')
ax.set_title('Variables seleccionadas y su Importancia en la variable objetivo')
plt.tight_layout()
plt.show()
```

El resultado se muestra a continuación:

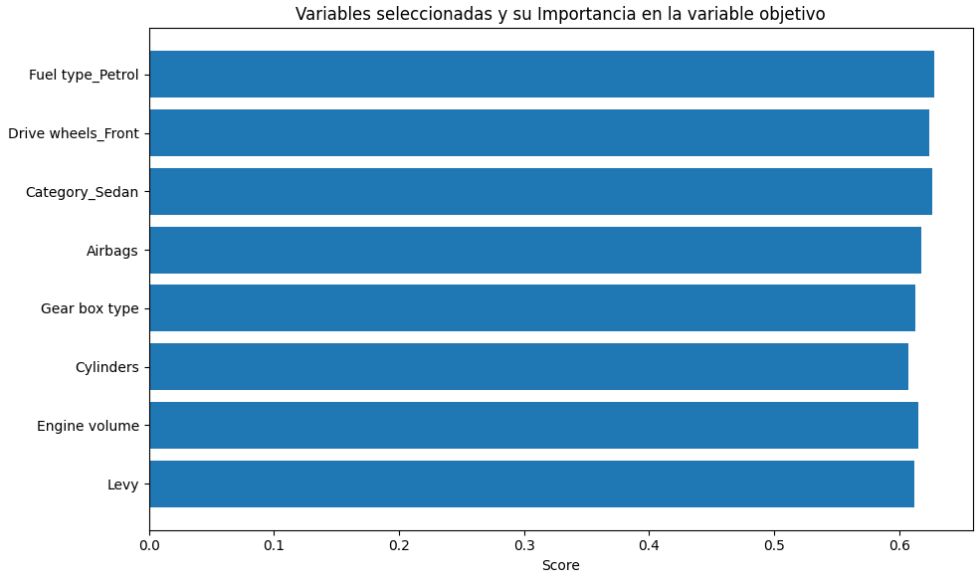

Figura 4: Variables numéricas más significativas.

Así, tras el segundo filtrado el modelo considera que la mejor combinación de variables es tomar las 8 a la vez (hallado con selector.n_features_), por lo que el dataset se halla finalmente listo para su tratamiento en la fase de modelaje.

4. SVM

Se propone aplicar la técnica SVM para predecir la variable objetivo. Esta proposición se refuerza sobre el hecho de que la variable objetivo es dicotómica y se poseen múltiples atributos explicativos. Además, la extensión del dataset no es relativamente extensa (inferior a 100000 muestras). Cabe mencionar que dicho algoritmo no funciona bien con valores *missing* y es sensible a datos no escalados (aquí el preprocesamiento realizado anteriormente es importante). Así, se analizará el rendimiento del modelo dados 2 *kernels* diferentes, evaluándolos mediante métricas como *Accuracy* y AUC. La primera da una estimación de la tasa de aciertos del modelo de clasificación para clasificar tanto resultados positivos como negativos, mientras que la segunda es la integración de la curva ROC, ofreciendo un método de cuantificar el poder discriminatorio entre clases del modelo. Así, al integrarse la curva ROC para obtener el AUC, se obtiene una métrica independiente del *threeshold* (umbral probabilístico a partir del cual una predicción cae dentro de una clase u otra) escogido por el modelo.
Primero, se separan los datos en una parte dedicada al entreno del modelo y otra para el testeo. Así, se dividen los datos entre una parte de entreno (80 %) y otra de prueba (20 %):

```
x_train, x_test, y_train, y_test = train_test_split(
    x_cut, y, test_size = 0.2, random_state = 1234
)
```

Es importante asegurarse de que se mantiene la misma proporción o parecida de valores Positivos y Negativos tanto en los datos de prueba como de entreno. Se comprueba con value_counts(normalize = True) en ambas secciones de datos que así se cumple: 50.7 % y 52.4 % de Positivos y 49.3 % y 47.6 % de Negativos, respectivamente para los datos de entreno y prueba. Por otra parte, se propone optimizar los hiperparámetros de los modelos dados 2 *kernels* distintos:

- ***Lineal***: utilizado cuando los datos son linealmente separables, corresponde a una función de decisión lineal. Tiene un bajo coste computacional. Se rige por el hiperparámetro C, que controla el balance entre maximizar la precisión sin caer en *overfitting*.
- ***Radial Basis Function (RBF)***: utilizado cuando los datos no son linealmente separables, corresponde a una función de decisión exponencial. Tiene un mayor coste computacional al mapear los datos en un espacio dimensionalmente superior. Se rige en función de los hiperparámetros C y γ, donde el último indica el peso ponderado de cada observación dado su distanciamiento respecto al resto.

Se entrenan ambos modelos con el siguiente código, donde mediante 10 validaciones cruzadas estratificadas en el algoritmo BayesSearchCV se hallarán los parámetros que ofrecen mejores resultados:

```
param_grid_linear = {
    'C': Real(0.001, 0.1)
}
param_grid_rbf = {
    'C': Real(0.001,10),
    'gamma': Real(0.01, 10)
}
svm_linear = SVC(kernel='linear')
svm_rbf = SVC(kernel='rbf')

bayes_search_linear = BayesSearchCV(svm_linear, param_grid_linear, n_iter=30,
 scoring='accuracy', cv=kfold)
bayes_search_rbf = BayesSearchCV(svm_rbf, param_grid_rbf, n_iter=30,
 scoring='accuracy', cv=kfold)

bayes_search_linear.fit(x_train, y_train)
bayes_search_rbf.fit(x_train, y_train)

best_params_linear = bayes_search_linear.best_params_
best_params_rbf = bayes_search_rbf.best_params_
```

En las 2 últimas líneas se almacenan los hiperparémtros optimizados, siendo $C = 0{,}1$ para el modelo Lineal y $C = 1{,}41$ y $\gamma = 9{,}33$ para el segundo. Con esto, se calculan las métricas mencionadas anteriormente más algunas de interés (*precision*, *recall* y *f1*) para evaluar el rendimiento de ambos modelos, almacenándolas en un DataFrame:

```
svm_best_linear = bayes_search_linear.best_estimator_
svm_best_rbf = bayes_search_rbf.best_estimator_

scorings = ['accuracy', 'roc_auc', 'precision', 'recall', 'f1']
std_columns = [col + "_std" for col in scorings]
all_columns = ["Model"] + scorings + std_columns
metrics = pd.DataFrame(columns=all_columns)

new_row = {'Model':'SVM Lineal'}

for metric in scorings:
    metric_svmlineal = cross_val_score(svm_best_linear, x_test, y_test, cv = kfold,
 scoring = metric)
    metric_svmlineal_mean, metric_svmlineal_std = metric_svmlineal.mean(), metric_svmlineal.std()
    new_row[f'{metric}'] = metric_svmlineal_mean
    new_row[f'{metric}_std'] = metric_svmlineal_std

metrics.loc[len(metrics)] = new_row

new_row = {'Model':'SVM RBF'}

for metric in scorings:
    metric_svmrbf = cross_val_score(svm_best_rbf, x_test, y_test, cv = kfold, scoring = metric)
    metric_svmrbf_mean, metric_svmrbf_std = metric_svmrbf.mean(), metric_svmrbf.std()
    new_row[f'{metric}'] = metric_svmrbf_mean
    new_row[f'{metric}_std'] = metric_svmrbf_std

metrics.loc[len(metrics)] = new_row
```

Los resultados son los siguientes:

Modelo	Accuracy	AUC	Precision	Recall	F1
SVM Lineal	0.627 ± 0.080	0.644 ± 0.070	0.614 ± 0.070	0.782 ± 0.090	0.687 ± 0.070
SVM RBF	0.588 ± 0.090	0.592 ± 0.110	0.613 ± 0.110	0.614 ± 0.120	0.607 ± 0.090

Cuadro 1: Rendimiento de los modelos SVM.

Gráficamente, se ilustran las métricas de mayor interés (*Accuracy* y *AUC*, con sus respectivas barras de error debido a las validaciones cruzadas):

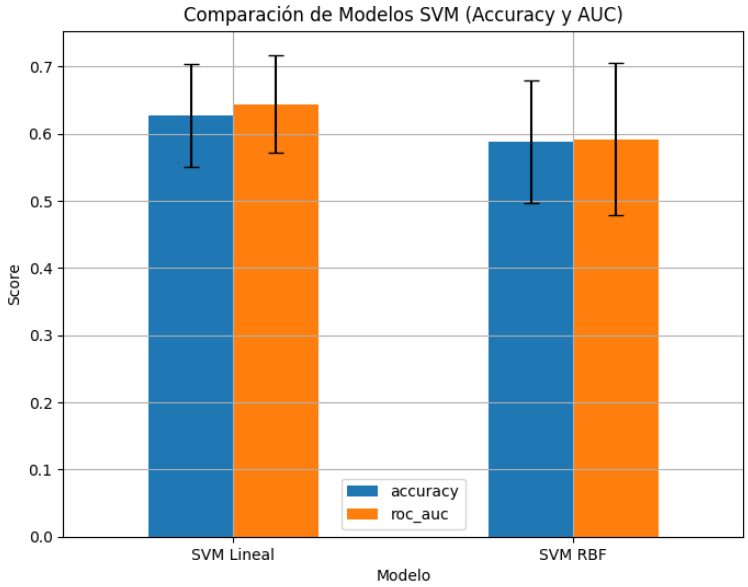

Figura 5: Comparativa gráfica del rendimiento de los modelos SVM.

Como se puede observar, el modelo que mejores resultados ofrece es aquel que ha sido ejectuado con el *kernel* Lineal, por lo que se toma este como modelo ganador. Este hecho indica un comportamiento lineal en cuanto a la distribución de los datos, de aquí que el *kernel* RBF ofrezca peores resultados. Cabe aquí mencionar que se han tomado únicamente estas 2 métricas al priorizar obtener el mayor número posible de Verdaderos Positivos y Verdaderos Negativos (en aquellos casos donde se prefiera reducir el número de Falsos Negativos, es conveniente utilizar otras métricas como *Recall* o F1).

4.1. Bagging

Se procede a aplicar la técnica de *bagging* sobre el modelo ganador del caso anterior, el modelo SVM de *kernel* lineal. Corresponde a una estrategia de ensamblaje que combina múltiples versiones del mismo modelo para producir predicciones más estables y precisas. Permite reducir la varianza del modelo y el *overfitting* promediando las predicciones. Esto conduce a un modelo más robusto que generaliza mejor para nuevos datos. Siendo el modelo ganador el SVM Lineal, se aplica el *bagging* sobre este:

```
bagging = BaggingClassifier(estimator=svm_best_linear, n_estimators=50, random_state=1234)
bagging.fit(x_train, y_train)

new_row = {'Model':'SVM Bagging'}

for metric in scorings:
    metric_bagging = cross_val_score(bagging, x_test, y_test, cv = cv, scoring = metric)
    metric_bagging_mean, metric_bagging_std = metric_bagging.mean(), metric_bagging.std()
    new_row[f'{metric}'] = metric_bagging_mean
    new_row[f'{metric}_std'] = metric_bagging_std

metrics.loc[len(metrics)] = new_row
```

Para una comparación visual con los modelos anteriores, se añaden estos resultados al gráfico anterior:

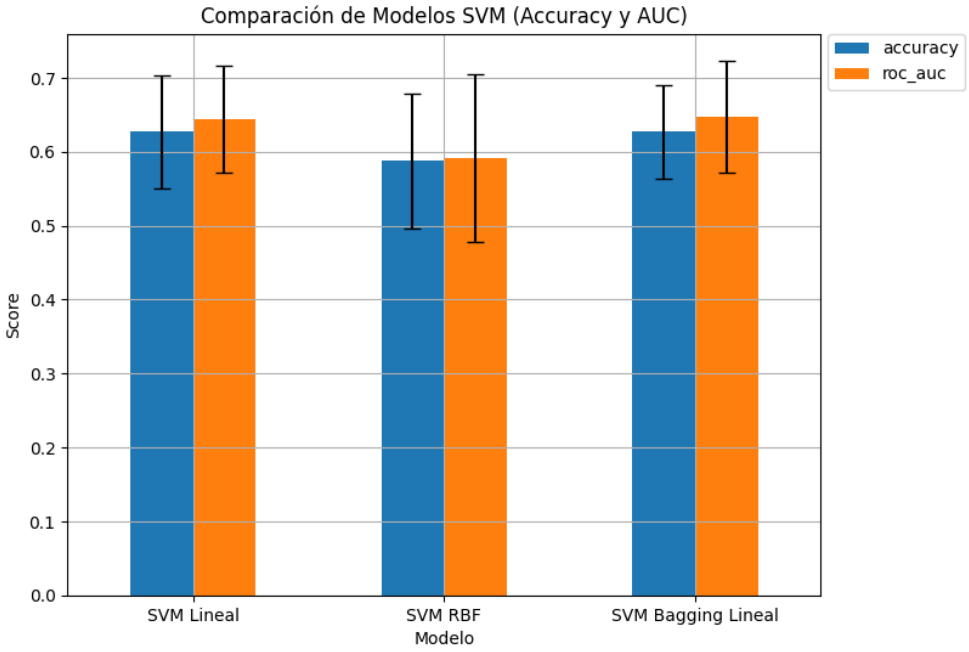

Figura 6: Comparativa gráfica del rendimiento de los modelos SVM junto al modelo con *bagging*.

Numéricamente, se comparan los 3 modelos SVM en la siguiente tabla:

Modelo	**Accuracy**	**AUC**	**Precision**	**Recall**	**F1**
SVM Lineal	0.627 ± 0.080	0.644 ± 0.070	0.614 ± 0.070	0.782 ± 0.090	0.687 ± 0.070
SVM RBF	0.588 ± 0.090	0.592 ± 0.110	0.613 ± 0.110	0.614 ± 0.120	0.607 ± 0.090
SVM Bagging Lineal	0.627 ± 0.060	0.648 ± 0.080	0.620 ± 0.060	0.746 ± 0.100	0.675 ± 0.060

Cuadro 2: Rendimiento de los modelos SVM.

El modelo ganador junto a su versión *bagging* apenas ofrece mejores resultados que el primero, por lo que en función del criterio que se priorice se escogerá uno u otro.

Se calcula la Matriz de Confusión sobre este último con el fin de analizar cuán de bien predice el modelo el color de los coches:

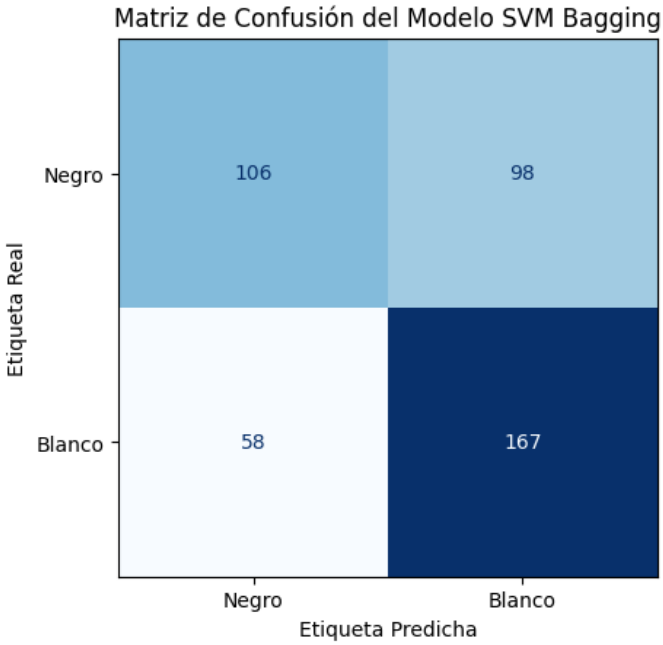

Figura 7: Matriz de Confusión del Modelo SVM *Bagging*.

De aquí, se extrae que cerca del 74.2 % de las veces el color Blanco se predice con éxito (Verdaderos Positivos), mientras que el color Negro solo se predice correctamente el 52.0 % de las veces (Verdaderos Negativos).
La Matriz de Confusión se ha obtenido con el siguiente código:

```
y_pred = bagging.predict(x_test)

cm = confusion_matrix(y_test, y_pred)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels = ['Negro', 'Blanco'])
disp.plot(cmap='Blues', colorbar = False)
disp.ax_.set_title("Matriz de Confusión del Modelo SVM Bagging")
disp.ax_.set_xlabel("Etiqueta Predicha")
disp.ax_.set_ylabel("Etiqueta Real")
plt.show()
```

Por último, se analiza como evoluciona el ratio de Verdaderos Positivos y Falsos Positivos del modelo de *bagging* en función del punto de corte escogido mediante la Curva ROC:

```
y_probs = bagging.predict_proba(x_test)[:, 1]

fpr, tpr, thresholds = roc_curve(y_test, y_probs)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'Curva ROC (AUC = {roc_auc:.2f})', color='blue')
plt.plot([0, 1], [0, 1], 'k--', label='Clasificación Aletoria')
plt.xlabel('Ratio de Falsos Positivos')
plt.ylabel('Ratio de Verdaderos Positivos')
plt.title('Curva ROC')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

El resultado es el siguiente:

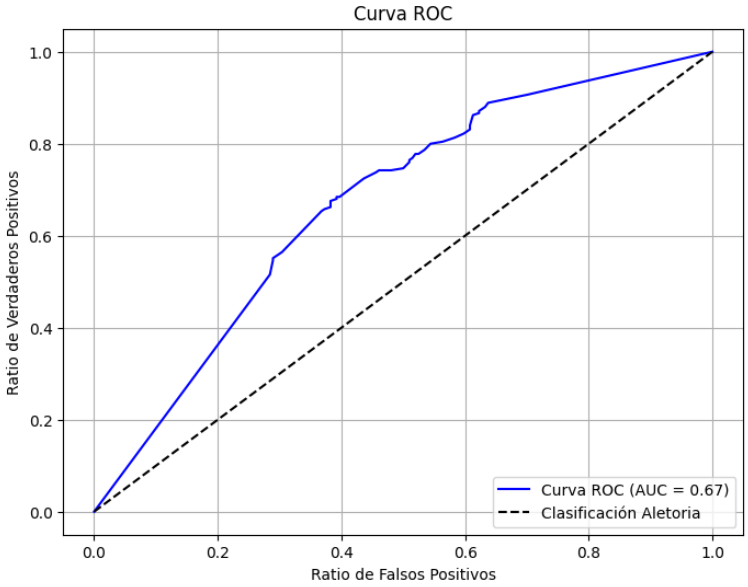

Figura 8: Curva ROC del Modelo SVM *Bagging*. En azul se representa el caso donde se tomase el valor del *threshold*=0.67, el que mejores resultados ofrece comprobado experimentalmente.

5. Modelo de *Stacking*

A continuación, se propone la implementación de la técnica de *Stacking* con el fin de reforzar el poder predictivo de un modelo en particular a partir del rendimiento de otros. Para ello, a parte de los 3 modelos vistos hasta ahora, se implementan 3 modelos adicionales:

- ***RandomForestClassifier***: utiliza un conjunto de árboles de decisión para realizar predicciones de valores binarios, combinándolos para obtener la mejor estimación. No es sensible a datos no escalados y lidia bien con valores *missing*. Algunos parámetros que rigen su rendimiento son: *n_estimators* (número de árboles a generar), *max_depth* (profundidad máxima de cada árbol), *min_samples_leaf* (mínimo número de muestras necesarias en una hoja del árbol), *max_features* (número máximo de características a considerar para dividir un nodo) y *bootstrap* (si se utilizan muestras con reemplazo para construir cada árbol).
- ***KNeighborsClassifier***: se clasifican observaciones similares en función de las etiquetas asociadas a sus vecinos más cercanos. Es sensible a datos no escalados y lidia mal con valores *missing*. Sus principales parámetros son: *n_neighbors* (número de vecinos a considerar), *weights* (función de ponderación), *metric* (métrica a utilizar para medir las distancias entre vecinos) y *p* (parámetro de la métrica de Minkowsky, donde en función de su valor se desglosará en las métricas de Manhattan y Euclidiana).
- ***XGBoostClassifier***: modelo de clasificación basado en el algoritmo de *gradient boosting*, que construye un conjunto de árboles de decisión de manera secuencial, optimizando los errores de predicción en cada iteración. No es sensible a datos no escalados y lidia bien con valores *missing*. Sus principales parámetros son: *n_estimators* (número de árboles a generar), *eta* (tasa de aprendizaje que controla la contribución de cada árbol), *max_depth* (profundidad máxima de los árboles) y *gamma* (reducción mínima de pérdida requerida para realizar una partición adicional en un nodo).

Véase que no se abordan modelos de redes neuronales dada su limitación a datasets de gran tamaño. En general, para datos de corta extensión, modelos "simples" de *machine learning* funcionan mejor.
Así, se entrenan los 3 modelos mencionados arriba para comparar su rendimiento con los modelos SVM vistos hasta ahora, aplicando de nuevo el algoritmo de optimización *BayesSearchCV* sobre los nuevos modelos así como validación cruzada. Para implementar el modelo de *RandomForestClassifier* se realiza lo siguiente:

```
forest_model = RandomForestClassifier(random_state = 1234, n_estimators = 1200)

param_grid_randomforest = {
    'max_depth': Integer(2,15),
    'min_samples_split': Integer(2, 10),
    'min_samples_leaf': Integer(1, 4)
```

```
}


bayes_search_randomforest = BayesSearchCV(forest_model, param_grid_randomforest, n_iter = 60,
 scoring='accuracy', cv=kfold)
bayes_search_randomforest.fit(x_train, y_train)

best_params_forest = bayes_search_randomforest.best_params_

forest_model = bayes_search_randomforest.best_estimator_
new_row = {'Model':'RandomForest'}

for metric in scorings:
    metric_randomforest = cross_val_score(forest_model, x_test, y_test, cv = kfold,
 scoring = metric)
    metric_randomforest_mean, metric_randomforest_std = metric_randomforest.mean(),
 metric_randomforest.std()
    new_row[f'{metric}'] = metric_randomforest_mean
    new_row[f'{metric}_std'] = metric_randomforest_std

metrics.loc[len(metrics)] = new_row
```

Los parámetros óptimos hallados para el modelo son: $max\_depth = 8$, $min\_samples\_leaf = 4$, $min\_samples\_split = 10$. Para el modelo *KNeighborClassifier* se ejecuta el siguiente código:

```
neighbor_model = KNeighborsClassifier()

param_grid_neighbor = {
    'n_neighbors': Integer(3,15),
    'weights': Categorical(['uniform', 'distance']),
    'p': Real(1, 2)
}

bayes_search_neighbor = BayesSearchCV(neighbor_model, param_grid_neighbor, n_iter = 60,
 scoring='accuracy', cv=kfold)
bayes_search_neighbor.fit(x_train, y_train)

best_params_neighbor = bayes_search_neighbor.best_params_

neighbor_model = bayes_search_neighbor.best_estimator_

new_row = {'Model':'KNeighbors'}

for metric in scorings:
    metric_neighbors = cross_val_score(neighbor_model, x_test, y_test, cv = kfold,
 scoring = metric)
    metric_neighbors_mean, metric_neighbors_std = metric_neighbors.mean(), metric_neighbors.std()
    new_row[f'{metric}'] = metric_neighbors_mean
    new_row[f'{metric}_std'] = metric_neighbors_std

metrics.loc[len(metrics)] = new_row
```

Sus parámetros optimizados son: $n\_neighbors = 11$, $p = 1{,}0$ y $weights = distance$. Finalmente, el algoritmo restante *XGBoostClassifier* tomará forma con la siguiente script:

```
xgboost_model = XGBClassifier(n_estimators = 1200)

param_grid_xgboost = {
    'eta': Real(0.1, 0.7),
    'gamma': Real(0.1, 1),
    'max_depth': Integer(1,5)
}
```

```
bayes_search_xgboost = BayesSearchCV(xgboost_model, param_grid_xgboost, n_iter = 60,
 scoring='accuracy', cv=kfold)
bayes_search_xgboost.fit(x_train, y_train)

best_params_xgboost = bayes_search_xgboost.best_params_

xgboost_model = bayes_search_xgboost.best_estimator_

new_row = {'Model':'XGBoostClassifier'}

for metric in scorings:
    metric_xgboost = cross_val_score(xgboost_model, x_test, y_test, cv = kfold, scoring = metric)
    metric_xgboost_mean, metric_xgboost_std = metric_xgboost.mean(), metric_xgboost.std()
    new_row[f'{metric}'] = metric_xgboost_mean
    new_row[f'{metric}_std'] = metric_xgboost_std

metrics.loc[len(metrics)] = new_row
```

Tras la optimización del anterior modelo, los hiperparámetros escogidos son: $\eta = 0{,}246$, $\gamma = 0{,}962$, $max\_depth = 2$. En la siguiente tabla se realiza una comparativa del rendimiento de todos los modelos estudiados hasta ahora:

Modelo	**Accuracy**	**AUC**	**Precision**	**Recall**	**F1**
SVM Lineal	0.627 ± 0.080	0.644 ± 0.070	0.614 ± 0.070	0.782 ± 0.090	0.687 ± 0.070
SVM RBF	0.588 ± 0.090	0.592 ± 0.110	0.613 ± 0.110	0.614 ± 0.120	0.607 ± 0.090
SVM Bagging Lineal	0.627 ± 0.060	0.648 ± 0.080	0.620 ± 0.060	0.746 ± 0.100	0.675 ± 0.060
RandomForest	0.616 ± 0.060	0.625 ± 0.090	0.630 ± 0.070	0.662 ± 0.090	0.642 ± 0.060
KNeighbors	0.546 ± 0.050	0.571 ± 0.060	0.570 ± 0.060	0.573 ± 0.080	0.568 ± 0.050
XGBoost	0.618 ± 0.070	0.628 ± 0.100	0.622 ± 0.070	0.716 ± 0.090	0.662 ± 0.060

Cuadro 3: Rendimiento de los modelos.

Se ilustra el rendimiento de cada modelo en un diagrama de barras:

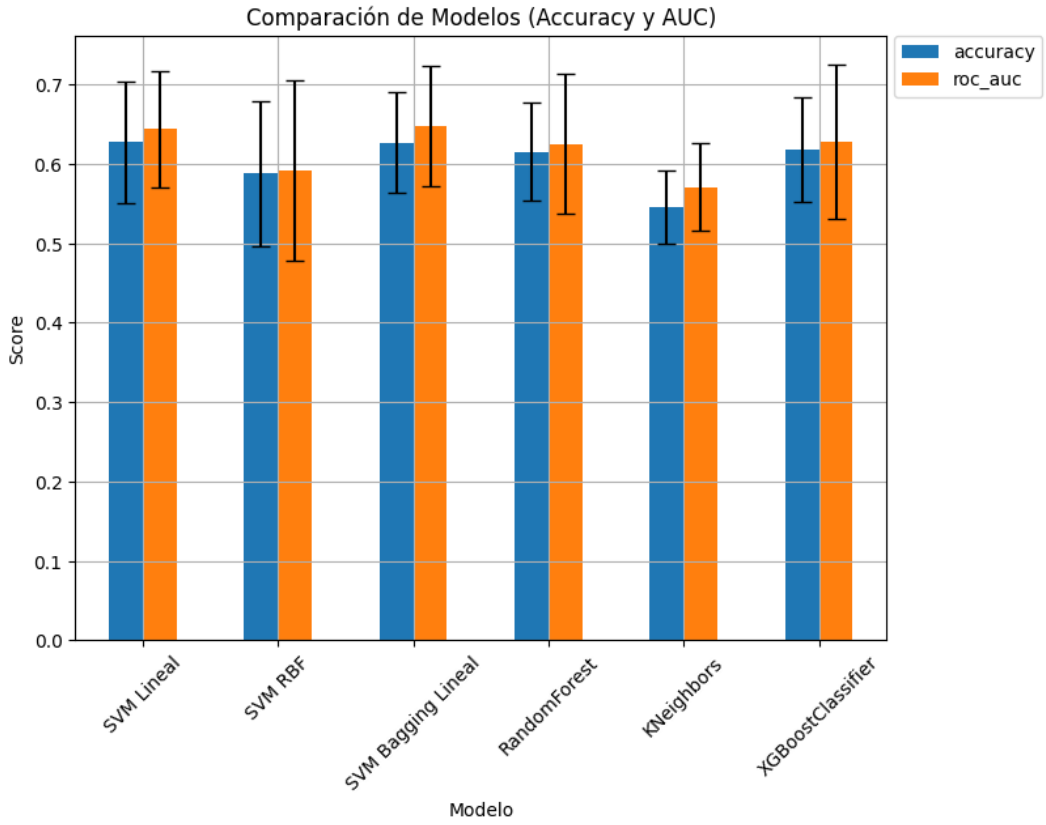

Figura 9: Comparativa gráfica del rendimiento de los modelos SVM, SVM *bagging*, *RandomForest*, *KNeighbors* y *XGBoost*.

Se observa como los modelos recientemente incluidos no mejoran las predicciones de los SVM. En especial, el modelo *KNeighbors* es el que peor rinde. A la hora de elegir el modelo final sobre el que construir el *Stacking*, es importante probar diferents modelos finales con el fin de obtener el mejor rendimiento. Así, se procede en primera instancia a la implementación de este algoritmo tomando como algoritmo final *SVM Bagging Lineal*. Además, únicamente se considerarán aquellos otros modelos que mejor rendimiento han tenido (*SVM Lineal*, *RandomForestClassifier* y *XGBoostClassifier*):

```
models = [
    ('SVM lineal', svm_best_linear),
    ('RandomForest', forest_model),
    ('XGBoost', xgboost_model)
]

stacking_model = StackingClassifier(
    estimators = models,
    final_estimator = bagging
)

stacking_model.fit(x_train, y_train)

new_row = {'Model':'Stacking SVM Bagging Lineal'}

for metric in scorings:
    metric_stacking = cross_val_score(stacking_model, x_test, y_test, cv = kfold, scoring = metric)
    metric_stacking_mean, metric_stacking_std = metric_stacking.mean(), metric_stacking.std()
    new_row[f'{metric}'] = metric_stacking_mean
    new_row[f'{metric}_std'] = metric_stacking_std

metrics.loc[len(metrics)] = new_row
```

Como se ha hecho en los anteriores casos, se aplica Validación Cruzada sobre el modelo a tratar. Los nuevos

resultados se añaden al DataFrame que recoge el rendimiento de cada modelo, donde como exposición final se adjuntan todos los estadísticos calculados para cada modelo:

Modelo	**Accuracy**	**AUC**	**Precision**	**Recall**	**F1**
SVM Lineal	0.627 ± 0.080	0.644 ± 0.070	0.614 ± 0.070	0.782 ± 0.090	0.687 ± 0.070
SVM RBF	0.588 ± 0.090	0.592 ± 0.110	0.613 ± 0.110	0.614 ± 0.120	0.607 ± 0.090
SVM Bagging Lineal	0.627 ± 0.060	0.648 ± 0.080	0.620 ± 0.060	0.746 ± 0.100	0.675 ± 0.060
RandomForest	0.616 ± 0.060	0.625 ± 0.090	0.630 ± 0.070	0.662 ± 0.090	0.642 ± 0.060
KNeighbors	0.546 ± 0.050	0.571 ± 0.060	0.570 ± 0.060	0.573 ± 0.080	0.568 ± 0.050
XGBoost	0.618 ± 0.070	0.628 ± 0.100	0.622 ± 0.070	0.716 ± 0.090	0.662 ± 0.060
Stacking SVM Bagging Lineal	0.632 ± 0.070	0.634 ± 0.080	0.619 ± 0.060	0.782 ± 0.090	0.689 ± 0.060

Cuadro 4: Rendimiento final de los modelos.

Se puede analizar además como varía el *Stacking* si se escoge como modelo final otro de los anteriormente seleccionados. Ordenados en función del valor de *Accuracy* descendentemente, se obtiene lo siguiente:

Modelo de *Stacking*					
Modelo final	**Accuracy**	**AUC**	**Precision**	**Recall**	**F1**
SVM Bagging Lineal	0.632 ± 0.070	0.634 ± 0.080	0.619 ± 0.060	0.782 ± 0.090	0.689 ± 0.060
SVM Lineal	0.620 ± 0.070	0.654 ± 0.100	0.614 ± 0.060	0.742 ± 0.100	0.670 ± 0.060
XGBoost	0.606 ± 0.060	0.628 ± 0.070	0.610 ± 0.060	0.688 ± 0.110	0.644 ± 0.070
RandomForest	0.587 ± 0.060	0.608 ± 0.060	0.605 ± 0.070	0.640 ± 0.090	0.618 ± 0.060

Cuadro 5: Rendimiento del modelo de *Stacking* en función del modelo final escogido.

Se observa que el modelo de *Stacking* con mejores resultados es el que se basa en el algoritmo *SVM Bagging Lineal*, por lo que de ahora en adelante para el *Stacking* se tratará únicamente con el mencionado modelo. El rendimiento final de los algoritmos se grafica a continuación:

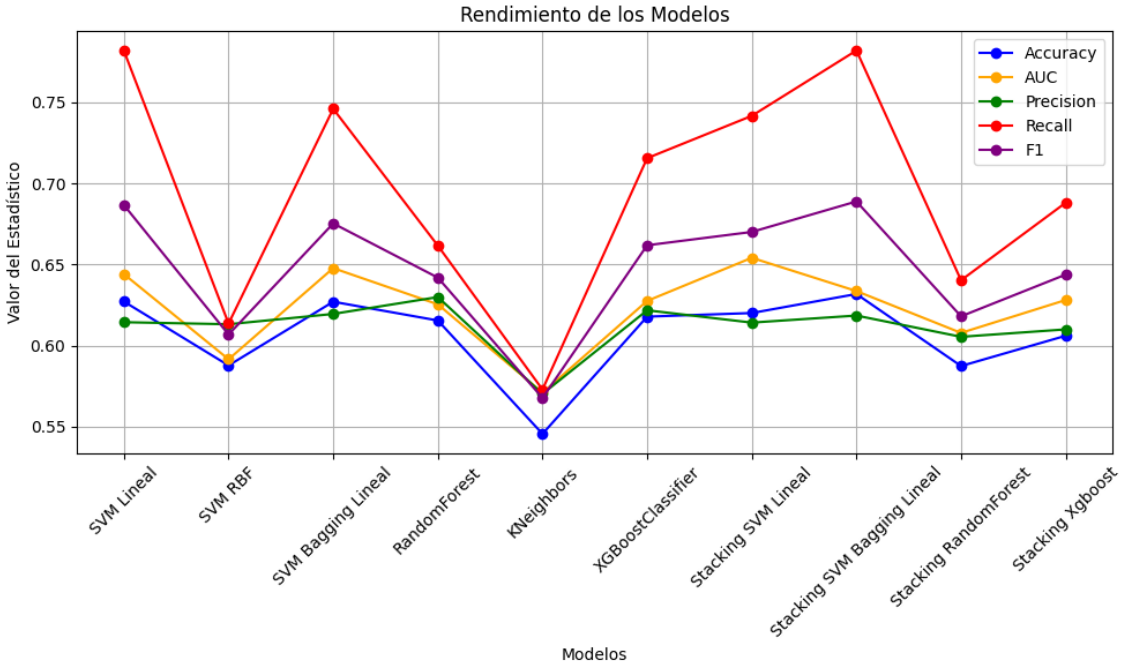

Figura 10: Comparativa gráfica del rendimiento de los modelos.

Se observa como cada modelo destaca en una métrica en particular (*SVM Bagging Lineal* en *Accuracy*, *Recall* y *F1*, *Stacking SVM Lineal* en *AUC* y *RandomForest* en *Precision*). Con esto, en función de la métrica que se priorice se escogerá un modelo u otro. Ahora bien, en general el modelo *Stacking SVM Bagging Lineal* es el que mejor rinde en la mayoría de las métricas de precisión, por lo que se toma dicho modelo como el ganador

de entre todos los barajados. Esta proposición refuerza el hecho de que la aplicación de técnicas de *Stacking* mejora en general la precisión del modelo.

Se representa a continuación la Matriz de Confusión del modelo de *Stacking* siguiendo el método mostrado en el apartado anterior:

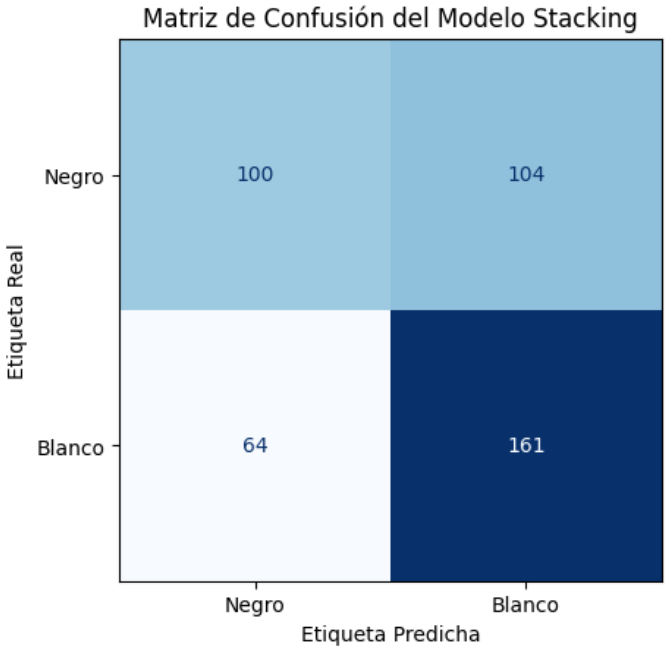

Figura 11: Matriz de Confusión del Modelo *Stacking*.

Así, el 49.0 % de las veces se predice correctamente el color Negro del coche, mientras que en cuanto al color Blanco se acierta un 71.1 %. Es importante notar aquí que si uno calcula a partir de las Matrices de Confusión de los modelos de *bagging* y *Stacking* el estadístico *Accuracy* es diferente a los tabulados. Esto se debe a que estas Matrices de Confusión se han calculado a partir del conjunto total de los datos de prueba, mientras los resultados tabulados se han hallado sobre los datos de testeo, resultados más fiables.

Finalmente, se grafica la curva ROC al igual que como se hizo con el modelo de *bagging* ganador:

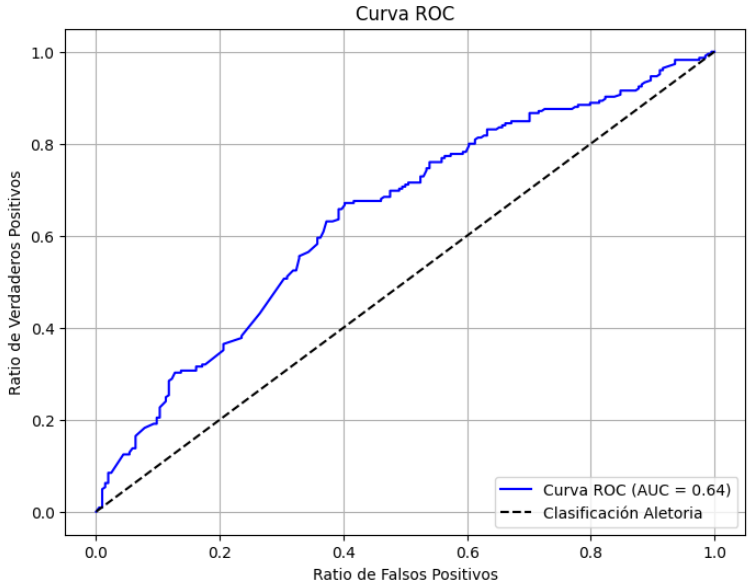

Figura 12: Curva ROC del *Modelo Stacking SVM Bagging Lineal*. En azul se representa el caso donde se tomase el valor del *threshold*=0.64, el que mejores resultados ofrece comprobado experimentalmente.

A diferencia del caso de *bagging* el punto de corte óptimo en este caso es de 0.64 en vez de 0.67 como en el caso anterior, por lo que el modelo de *Stacking* tiende a identificar una observación como Positiva ligeramente antes que el modelo de *bagging*.

6. Conclusiones

Los hallazgos y observaciones realizados hasta ahora se resumen en las siguientes conclusiones:

1. Un *kernel* Lineal en un modelo SVM ofrece mejores resultados que un *kernel* RBF en cuanto a la predicción del color del coche, reforzando la hipótesis de la distribución lineal de los datos.
2. El modelo de SVM *bagging* con *kernel* lineal ofrece mejores resultados que el modelo SVM con *kernels* RBF, y similares respecto al modelo SVM con *kernel* lineal.
3. Los modelos de *XGBoostClassifier*, *RandomForestClassifier* y *KNeighborsClassifier* no consiguen superar modelos más básicos como el SVM con *kernel* lineal.
4. La implementación del *Stacking* tomando como modelo final el *SVM Bagging Lineal* es el que mejores resultados ofrece en gran parte de los estadísticos analizados. En función de la métrica que se priorice, se utilizará este modelo u otro.

7. Bibliografía

- Car Price Prediction Callenge. (2025), *Kaggle.com*. Link.
- Gutierrez, Inmaculada. Apuntes de la Asignatura Machine Learning, Redes Neuronales.(2025). Universidad Complutense de Madrid.
- Gómez, Daniel. Apuntes de la Asignatura Machine Learning, SVM, Ensemble.(2025). Universidad Complutense de Madrid.