1-5-2025

MACHINE LEARNING

Clasificación Binaria

Jacobo Álvarez Gutiérrez
UNIVERSIDAD COMPLUTENSE DE MADRID

Índice

1. Introducción

El presente documento tiene como objetivo detallar el desarrollo de un modelo de clasificación binaria orientado a predecir si un vehículo debe ser repintado de color blanco o no, a partir de una base de datos con características técnicas y comerciales de automóviles. Este problema se aborda como un ejercicio práctico de aprendizaje supervisado, combinando técnicas de depuración, análisis descriptivo, imputación de valores faltantes y ajuste de modelos predictivos. La variable objetivo se construye a partir del campo original *'Color'*, limitado exclusivamente a las categorías blanco y negro. A lo largo del trabajo se aplicarán diversos algoritmos de clasificación, incluyendo máquinas de vectores soporte (SVM) y modelos de ensamblado *bagging* y *stacking*, prestando especial atención a la preparación adecuada de los datos y a la evaluación comparativa del rendimiento de los modelos. Además, la finalidad de este proyecto no es solo alcanzar un alto grado de precisión, sino también justificar cada una de las decisiones metodológicas adoptadas en el proceso.

2. Depuración de la base de datos

2.1. Consideraciones generales sobre las variables

En primer lugar, definimos el directorio de trabajo y cargamos la base de datos para hacer un análisis visual de la misma.

```
os.chdir('C:/Users/jacob/Desktop/master_ucm/mod8_machine_learning/Tarea')
datos = pd.read_excel('datos_tarea25.xlsx')
```

Además, combinamos esto con el comando *'datos.dtypes()'* para observar la tipología predefinida para cada variable. Tras esta primera visualización, vemos que es necesario hacer las siguientes correcciones:

- La variable *'Levy'* es un número entero, así que debemos cambiar su tipología a *'Int64'* (porque *'Int64'* permite gestionar valores perdidos y *'int64'* no).
- La variable *'Engine volume'* tiene, en algunos registros, una caracterización adicional para especificar si un motor es turbo. Por ende, se añadirá esta caracterización en una columna adicional a la base de datos y se transformarán los valores del volumen del motor a *floats*.
- La variable *'Mileage'* contiene en cada registro la cadena de texto *'km'*. Debemos suprimir la misma y transformar la tipología de la variable a numérica.
- La variable objetivo *'Color'* la convertimos en una variable binaria asignando el valor 1 para '*White'* y 0 para '*Black*'. De forma análoga, las variables '*Leather interior*', '*Wheel*' y '*Gear box type*' también se pueden convertir en variables binarias.

Todas las consideraciones hechas sobre la base de datos se solventan con las siguientes líneas de código:

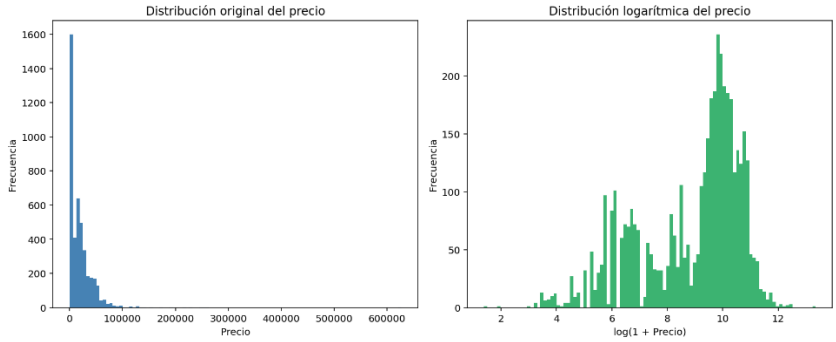
```
# Convertimos 'Levy' a entero (Int64 para poder manejar valores missing)
datos['Levy'] = datos['Levy'].replace('-', np.nan).astype('Int64')

# Convertimos 'Engine volume' a float y añadimos una variable binaria para
# especificiar si el motor es turbo
datos['Turbo'] = (datos['Engine volume']
                  .str.contains('Turbo')
                  .map({True: 1, False: 0}))

datos['Engine volume'] = (datos['Engine volume']
                          .str.replace('Turbo', '', regex = True)
                          .astype(float))

# Suprimimos la cadena 'km' de la variable 'Mileage' y convertimos a entero
datos['Mileage'] = (datos['Mileage']
                    .str.replace('km', '', regex = True)
                    .astype(int))

# Convertimos la variable 'Color' en binaria (Int64), siendo 1 blanco y 0 negro
datos['Color'] = datos['Color'].map({'White': 1, 'Black': 0})

# Convertimos la variable 'Leather interior' en binaria, siendo 1 yes y 0 no
datos['Leather interior'] = datos['Leather interior'].map({'Yes': 1, 'No': 0})

# Convertimos la variable 'Wheel' en binaria, siendo 1 left y 0 right
datos['Wheel'] = datos['Wheel'].map({'Left wheel':1, 'Right-hand drive':0})

# Convertimos la variable 'Gear box type' en binaria, siendo:
datos['Gear box type'] = datos['Gear box type'].map({'Automatic':1,
                                                     'Tiptronic':0})
```

2.2. Búsqueda y tratamiento de valores atípicos

A continuación, consideramos el comando *'datos.describe().T'* con el objetivo de encontrar valores atípicos o alguna otra característica destacable a simple vista. Se observa con esto que el valor mínimo de la variable *'Engine volume'* es cero, lo cual no tiene sentido físico. Al mirar en detalle se aprecian dos registros con un volumen nulo, otro con un valor 0.1 y otro con un valor 0.6. Por otra parte, también hay registros de coches con 1, 2 o 7 cilindros, lo cual son valores completamente inusuales para coches. De forma análoga, existe un valor anómalo en la variable *'Mileage'* ($\approx 1{,}111 \times 10^9$ Km). Además, teniendo en cuenta que únicamente conforman un total de 11 registros, se ha decidido suprimirlos tal que:

```
# Eliminamos algunos valores atípicos (11 registros en total)
datos = datos[~datos['Engine volume'].isin([0, 0.1, 0.6])].copy()
datos = datos[~datos['Cylinders'].isin([1, 2, 7])].copy()
datos = datos[datos['Mileage'] != 1111111111]
```

Adicionalmente, la variable *'Price'* también muestra algunos valores un tanto sospechosos, aunque no de manera tan aislada. Veamos una representación en histogramas de esta variable en su escala natural y en escala logarítmica (por simplicidad se omiten las líneas de código):



Como se puede apreciar, hay un claro rango de valores donde la frecuencia es mayor. Vamos a usar la versión logarítmica para cribar algunos valores atípicos a partir del rango intercuartílico. Una vez hecho esto, vemos que la cantidad de registros fuera de ese rango intercuartílico es ínfima, así que se ha decidido suprimirlos. Esto se hace a partir de las siguientes líneas de código:

```python
# Rango intercuartílico en log(Price)
Q1 = np.log1p(datos['Price']).quantile(0.25)
Q3 = np.log1p(datos['Price']).quantile(0.75)
IQR = Q3 - Q1

lim_inf = np.exp(Q1 - 1.5 * IQR)
lim_sup = np.exp(Q3 + 1.5 * IQR)

fuera_rango = datos[(datos['Price'] < lim_inf) | (datos['Price'] > lim_sup)]
print(f"Porcentaje eliminado: {100 * len(fuera_rango) / len(datos):.3f}%")

#Como son muy pocos valores se eliminarán los registros asociados (7 en total)
datos = datos[(datos['Price'] >= lim_inf) & (datos['Price'] <= lim_sup)].copy()
```

Antes del siguiente apartado, conviene señalar que, aunque no se añadirán las líneas de código asociadas por simplicidad, se ha comprobado que ninguna categoría de cada variable categórica está poco representada en la muestra.

2.3. Tratamiento de valores *missing*

Llegados a este punto donde no se aprecia ninguna otra anomalía destacable, vamos a hacer un tratamiento de los valores *missing*. Al aplicar *'datos.isnull().sum()',* se comprueba que la única variable con valores perdidos es '*Levy*'. No obstante, antes de imputar por valores centrales y dar por concluido este paso, se ha decidido observar si existe alguna justificación para que esos valores no estén en la base de datos. Para ello, se ha añadido la variable binaria '*Levy_missing*' y se ha estudiado la correlación de esta y la variable original con todas las demás, obteniendo los siguientes resultados (por simplicidad no se añaden las líneas de código):

Índice	Cylinders	ngine volum	Mileage	ather interi	Price	Turbo	Gear box type	Airbags	Color	Prod. year	Wheel
Levy	0.706669	0.69294	0.111579	0.0916928	0.0307638	-0.00942724	-0.0256041	-0.0476683	-0.128669	-0.228764	-0.249047
Levy_missing	0.0528859	-0.0172555	0.0157247	-0.319833	0.151213	0.262494	-0.36482	-0.00902622	-0.0499254	-0.352232	-0.191795

Nótese que las principales razones para el aumento de los impuestos asociados a un coche se deben al número de cilindros y la capacidad del motor, es decir, a la potencia de este. Por otra parte, parece que el año de fabricación y el tipo de caja de cambios están relacionadas con los valores faltantes de impuestos. Concretamente, parece que estos valores faltantes se deban a coches muy antiguos (quizás descatalogados) o algunos de los más modernos.

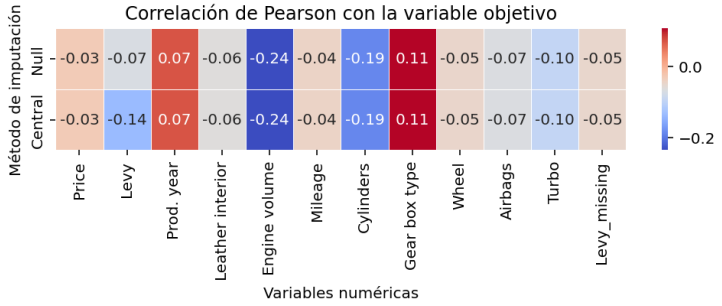
Esta situación plantea la duda de si se trata de un error de la base de datos o si, por el contrario, esos vehículos están exentos de impuestos o aún no se les ha registrado un valor. En este contexto, la imputación con un valor nulo puede estar justificado. Atendiendo a esta observación, vamos a confeccionar dos bases de datos depuradas de diferente manera. Por un lado, probaremos con la imputación de valores nulos para cada uno de esos registros perdidos y, por otra parte, atendiendo a la alta correlación con el número de cilindros, vamos a imputar por valores centrales, pero separando por grupos en base a esta característica. Compararemos posteriormente los diferentes métodos de imputación. Todo el proceso anterior se resume en las siguientes líneas de código:

```python
# Imputación de valores nulos
datos_null_imput = datos.copy()
datos_null_imput['Levy'] = datos_null_imput['Levy'].fillna(0)

# Imputación de valores centrales según número de cilindros
medianas_cylinders = datos.groupby('Cylinders')['Levy'].median().round()
datos_central_imput = datos.copy()
datos_central_imput['Levy'] = datos_central_imput['Levy'].fillna(
    datos_central_imput['Cylinders'].map(medianas_cylinders))
```

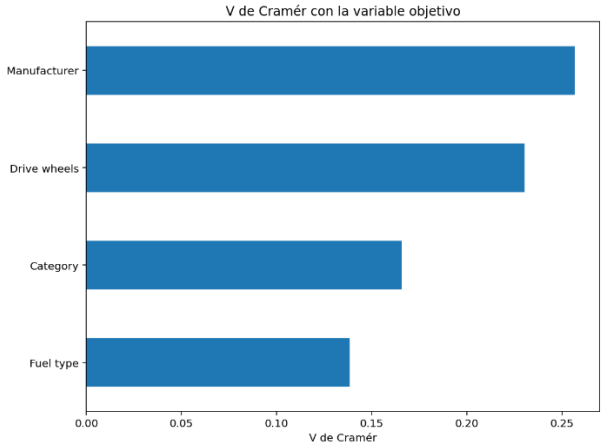
2.4. Selección de variables

En esta sección se estudiará la relevancia del conjunto de variables predictoras sobre la variable objetivo para los diferentes métodos de imputación (se omiten líneas de código por simplicidad).

En lo que respecta a la relación con las variables numéricas se obtuvieron los siguientes valores para el coeficiente de correlación de Pearson:



Por otra parte, en lo que respecta a las variables categóricas, se obtuvo la siguiente representación gráfica para el parámetro V de Cramer:



Como se puede apreciar de ambas representaciones, parece que ninguna de las variables esté estrictamente relacionada con la variable objetivo de manera individual. Por esta razón y atendiendo a que tampoco son demasiadas variables, la elaboración de los modelos de este proyecto se realizará con todo el conjunto de estas. Además, como el método de imputación tampoco ha sido determinante por el momento, seguiremos comparando la efectividad de los modelos predictivos en cada caso.

2.5. Normalización de variables numéricas y recodificación de las categóricas

Finalmente, para concluir con este proceso de depuración de la base de datos, vamos a normalizar todas las variables numéricas y se aplicará una codificación *one-hot* a todas las categorías de cada variable categórica. Aunque distintos modelos tienen distintos requisitos en cuanto a la escala de las variables, se ha optado por una normalización tipo MinMax (muy adecuada para la Máquina de Vectores Soporte SVM que se implementará en primer lugar) para todos los modelos que se desarrollarán posteriormente, pues permite mantener un flujo de trabajo unificado sin perjudicar significativamente el rendimiento de los modelos base que no requieren normalización (como los árboles de decisión). Además, aunque esta transformación puede reducir la interpretabilidad de algunos modelos, estos se usarán únicamente como modelos base del ensamblado, así que las predicciones individuales de estos no se analizarán en detalle. A continuación, para finalizar este proceso depurativo, se muestran las líneas de código asociadas a esta última parte:

```python
def transformar_y_guardar(X, y, num_cols, cat_cols, nombre_archivo):
    # Pipeline de transformación
    preprocesador = ColumnTransformer(transformers=[
        ('num', MinMaxScaler(), num_cols),
        ('cat', OneHotEncoder(drop='first', sparse_output=False), cat_cols)
    ])

    # Aplicar transformaciones
    X_transformado = preprocesador.fit_transform(X)

    # Obtener nombres de columnas finales
    columnas_cat = (preprocesador.named_transformers_['cat']
                    .get_feature_names_out(cat_cols))
    columnas_finales = num_cols + list(columnas_cat)

    # Crear DataFrame resultante
    df_resultado = pd.DataFrame(X_transformado, columns=columnas_finales,
                                index=X.index)
    df_resultado['Color'] = y

    # Guardar en Excel
    df_resultado.to_excel(nombre_archivo, index=False)

transformar_y_guardar(X_null, y, num_cols, cat_cols, "datos_null_transf.xlsx")
transformar_y_guardar(X_central, y, num_cols, cat_cols,
                      "datos_central_transf.xlsx")
```

Para trabajar posteriormente con estas bases de datos ya depuradas simplemente hacemos una llamada a estos archivos generados tal que:

```python
os.chdir('C:/Users/jacob/Desktop/master_ucm/mod8_machine_learning/tema_1/Tarea')
data_null = pd.read_excel('datos_null_transf.xlsx')
data_central = pd.read_excel('datos_central_transf.xlsx')
```

3. Modelo SVM

El primer paso para empezar con la elaboración de cualquier modelo es hacer una partición *train/test* de cada base de datos. Esto lo hacemos a partir de las siguientes líneas de código:

```python
def particion(data):
    y = data['Color']
    X = data.drop(columns=['Color'])

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=123, stratify=y)

    return X_train, X_test, y_train, y_test

X_null_train, X_null_test, y_null_train, y_null_test = particion(data_null)
(X_central_train, X_central_test,
 y_central_train, y_central_test) = particion(data_central)
```

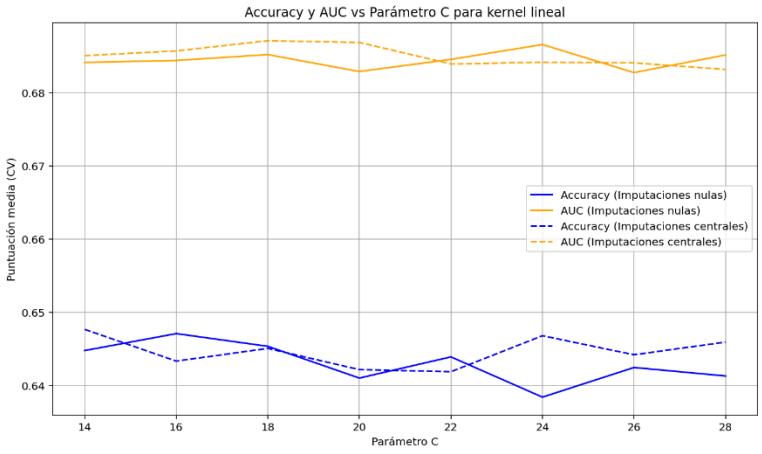
Por otra parte, en lo que respecta al fundamento de las máquinas SVM, sabemos que este reside en encontrar el hiperplano que mejor separa las clases en el espacio de características, maximizando el margen entre las observaciones más cercanas a dicho hiperplano. Compararemos dos variantes del modelo: una con un *kernel* lineal y otra con *kernel* radial de base gaussiana (RBF). Con esta elección se pretende comprobar si el problema es linealmente separable de manera eficiente o, en su defecto, existen relaciones no lineales relevantes. Además, ambos modelos se ajustarán mediante una parrilla de hiperparámetros buscando maximizar las métricas de evaluación.

3.1. Elección del kernel

Se muestra a continuación la definición de cada *kernel,* así como de la parrilla inicial de parámetros (esta última se ajustará según las necesidades del modelo).

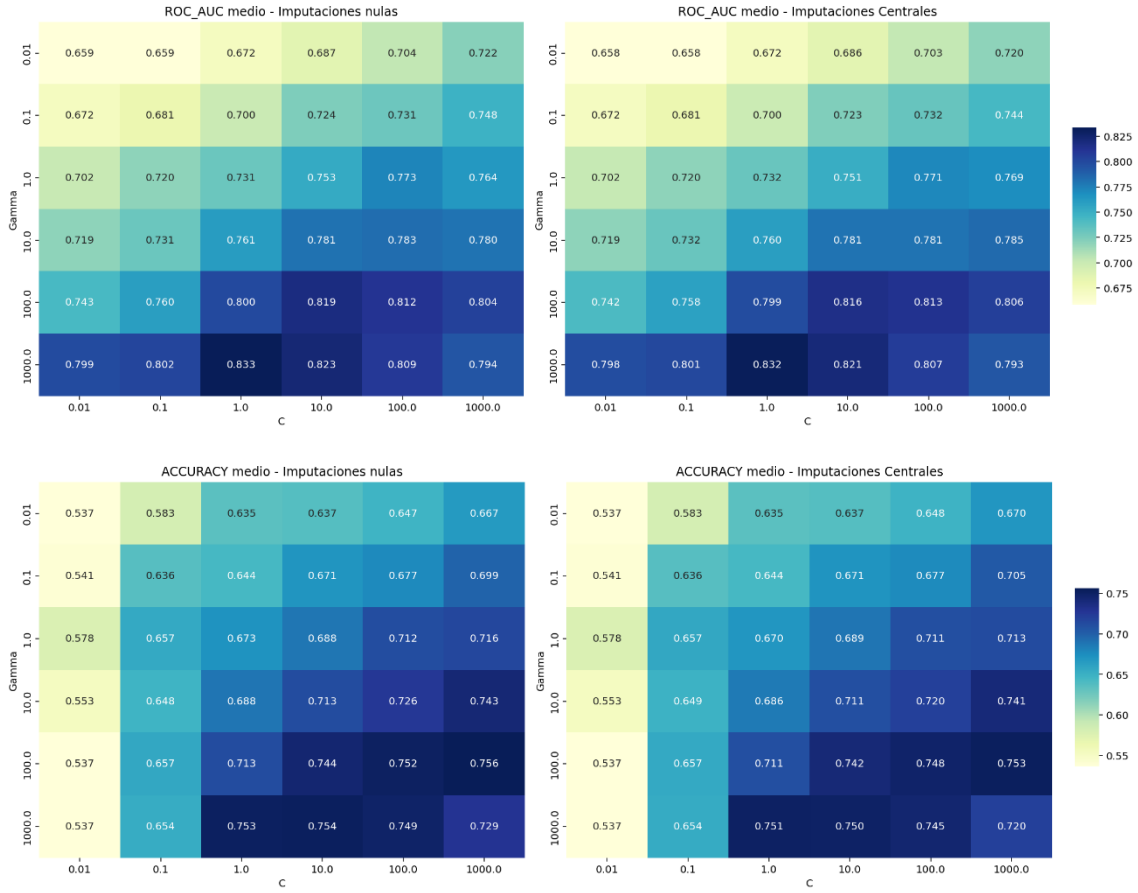
```python
def kernel_lineal(x, y):
    svc_linear = SVC(kernel='linear', probability=True,
                     max_iter=10000, tol=1e-3, random_state=123)

    # Parrilla original: [0.01, 0.1, 1, 10, 100, 1000]
    param_grid_linear = {'C': [14, 16, 18, 20, 22, 24, 26, 28]}

    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

    grid_linear = GridSearchCV(estimator=svc_linear,
                               param_grid=param_grid_linear,
                               scoring=['accuracy', 'roc_auc'],
                               refit='roc_auc', cv=cv,
                               return_train_score=True,
                               n_jobs=1, verbose=2)

    grid_linear.fit(x, y)
    return grid_linear


def kernel_rbf(x, y):
    svc_rbf = SVC(kernel='rbf', probability=True, max_iter=10000, tol=1e-3,
                  random_state=123)

    param_grid_rbf = {'C': [0.01, 0.1, 1, 10, 100, 1000],
                      'gamma': [0.01, 0.1, 1, 10, 100, 1000]}

    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

    grid_rbf = GridSearchCV(estimator=svc_rbf, param_grid=param_grid_rbf,
                            scoring=['accuracy', 'roc_auc'], refit='roc_auc',
                            cv=cv, return_train_score=True,
                            n_jobs=1, verbose=2)

    grid_rbf.fit(x, y)
    return grid_rbf
```

Aunque por simplicidad se omiten las líneas de código asociadas a la representación, estos fueron los resultados obtenidos para los parámetros '*Accuracy*' y '*AUC*' en función del parámetro '*C*' para el *kernel* lineal:



Vemos que, para ambas bases de datos, el valor más alto del parámetro *AUC* y de *Accuracy* se obtiene para valores de *C* en el intervalo (14, 28). Atendiendo entonces a que las variaciones de los parámetros en ese intervalo son insignificantes y que el comportamiento entre ambos métodos de imputación es muy similar, se ha decidido adoptar como valor óptimo $C = 20$ para todos los casos.

Repitiendo el proceso para el *kernel* RBF y omitiendo de nuevo las líneas de código, se obtiene la siguiente representación de *AUC* y *Accuracy* en función de los parámetros *C* y *gamma*:

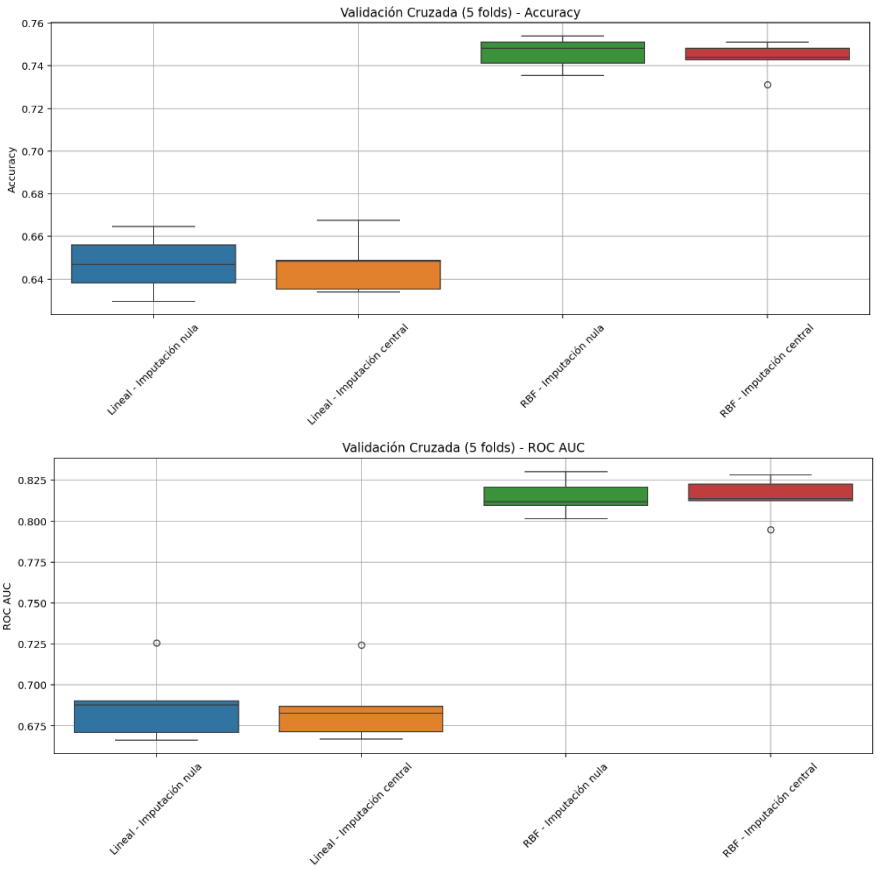


Nótese que ambas métricas presentan una evolución coherente, aumentando el rendimiento del modelo de manera progresiva a medida que aumentan los valores de los parámetros *C* y *gamma*. Vemos que la región donde se obtienen los mejores resultados es donde $c \geq 1$ y $gamma \geq 10$, aunque después de los valores $C = 10$ y $gamma = 100$ las mejoras ya no son significativas, así que se tomará esta configuración como la óptima.

Se concluye entonces con la siguiente definición de modelos óptimos, así como con su fase de entrenamiento con el conjunto de datos adecuado:

```
# Kernel lineal
SVM_ln_null = SVC(kernel='linear', C=20., probability=True, random_state=123)
SVM_ln_central = SVC(kernel='linear', C=20., probability=True,
                     random_state=123)

# Kernel RBF
SVM_rbf_null = SVC(kernel='rbf', C=10., gamma=100., probability=True,
                   random_state=123)
SVM_rbf_central = SVC(kernel='rbf', C=10., gamma=100., probability=True,
                      random_state=123)
```

A continuación, vamos a evaluar la influencia de los diferentes métodos de imputación, así como de los diferentes *kernels* empleados. Para ello, se obtendrán los valores de las métricas *AUC* y *Accuracy* para cada uno de los cuatro casos empleando un proceso de validación cruzada. Por simplificar este documento de texto, no se añadirán las líneas de código asociadas a las gráficas que siguen a continuación:



A partir de los resultados obtenidos, pueden extraerse dos conclusiones relevantes. En primer lugar, las métricas de evaluación (*Accuracy* y *AUC*) presentan valores prácticamente idénticos entre los dos métodos de imputación considerados, lo que indica que la estrategia de imputación no tiene un impacto significativo en el rendimiento del modelo. De hecho, de ahora en adelante se trabajará únicamente con la base de datos con imputaciones nulas para simplificar. En segundo lugar, se observa una mejora clara en ambas métricas al emplear el *kernel* RBF en lugar del *kernel* lineal, lo que permite concluir que el uso de un *kernel* no lineal resulta más adecuado para capturar la complejidad del problema.

Llegados a este punto, solo queda declarar el modelo ganador, entrenarlo y obtener las predicciones (estas se comentan en el siguiente apartado):

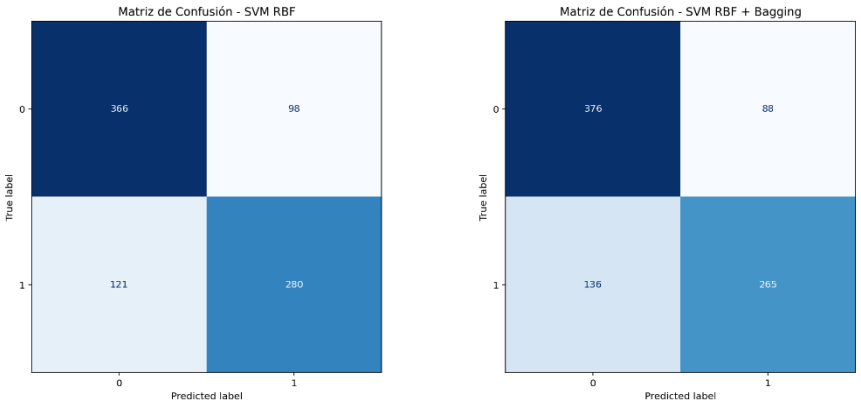
```
best_SVM = SVC(kernel='rbf', C=10.0, gamma=100.0, probability=True,
               random_state=123)
best_SVM.fit(X_null_train, y_null_train)
y_pred_SVM = best_SVM.predict(X_null_test)
y_prob_SVM = best_SVM.predict_proba(X_null_test)[:, 1]
```

3.2. *Bagging* para el mejor modelo obtenido

A continuación, vamos a implementar una técnica de ensamblado *Bagging* para intentar mejorar la capacidad predictiva del modelo con el que concluía el apartado anterior. Como sabemos, esta técnica consistirá en entrenar múltiples versiones del mismo modelo, pero sobre subconjuntos aleatorios del conjunto de datos de entrenamiento generados mediante un muestreo con reemplazo. Así, posteriormente se podrá hacer una combinación de las predicciones. Tanto la definición de este modelo como su proceso de entrenamiento y predicción se llevan a cabo con las siguientes líneas de código:

```
SVM_bagging = BaggingClassifier(
    estimator = SVC(kernel='rbf', C=10.0, gamma=100.0, probability=True,
                    random_state=123),
    n_estimators=10, max_samples=0.8, bootstrap=True, random_state=123,
    n_jobs=1)
SVM_bagging.fit(X_null_train, y_null_train)
y_pred_bagging = SVM_bagging.predict(X_null_test)
y_prob_bagging = SVM_bagging.predict_proba(X_null_test)[:, 1]
```

Una vez hechas las predicciones tanto con el modelo base como con el modelo *bagging*, este es el resumen de los resultados obtenidos (por simplicidad se omiten las líneas de código):



Modelo	Accuracy	AUC
SVM RBF	0.747	0.81
SVM RBF + Bagging	0.741	0.816

Tal y como se puede apreciar, no existe una mejora significativa para ninguna de las métricas al aplicar la técnica bagging. Aun así, podemos justificar la implementación de esta técnica como una estrategia efectiva para consolidar el rendimiento del modelo final. Asimismo, vemos que para ambos modelos, la matriz de confusión refleja una distribución relativamente equilibrada entre clases, lo cual sugiere que los modelos mantienen una buena capacidad para identificar correctamente tanto a los vehículos que deben ser repintados de color blanco como los que no. Además, el número de falsos positivos y falsos negativos es moderado, lo que indica un buen compromiso entre sensibilidad y especificidad.

4. Modelo *Stacking*

El *Stacking* es otra técnica de ensamblado que emplea las predicciones de varios modelos base como *inputs* de un modelo final. Con el objetivo de optimizar al máximo este último modelo, es conveniente que los modelos base sean diversos en naturaleza, para así poder captar diferentes patrones y relaciones entre los datos. En este caso particular, la elección de modelos base ha sido la siguiente:

- ***Regresión Logística:*** modelo lineal, simple y rápido.
- ***Random Forest:*** modelo de árboles no lineal, robusto ante el sobreajuste y que permite captar algunas relaciones más complejas.
- ***XGBoost:*** otro modelo basado en árboles que complementa al *Random Forest* por su enfoque secuencial.
- ***Máquina de Vector Soporte (SVM):*** definiremos de nuevo el mismo modelo SVM con *kernel* RBF que resultó ganador en el apartado 3.1 de este documento.

A continuación, se detallará por separado la definición de cada uno de los modelos base para su posterior comparación. Finalmente, se realizará el proceso de ensamblado de *stacking*.

4.1. Regresión Logística

Las siguientes líneas de código reflejan la búsqueda de los parámetros óptimos, así como la definición del modelo con estos valores:

```
lr_param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],
    'solver': ['liblinear', 'lbfgs'],
    'penalty': ['l2'],
    'class_weight': ['balanced']}

lr_model = LogisticRegression(max_iter=10000, random_state=123)
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=123)

lr_grid = GridSearchCV(estimator=lr_model, param_grid=lr_param_grid,
                       scoring=['accuracy', 'roc_auc'], refit='roc_auc',
                       cv=cv, return_train_score=True, n_jobs=1, verbose=2)

lr_grid.fit(X_null_train, y_null_train)
print(lr_grid.best_params_)
# {'C': 100, 'class_weight': 'balanced', 'penalty': 'l2', 'solver': 'lbfgs'}

lr_base = LogisticRegression(**lr_grid.best_params_,
                             max_iter=10000,
                             random_state=123)
```

Tal y como se aprecia en el código, el mejor modelo se obtuvo para $C = 100$ y *class_weight=balanced*, indicando que el modelo requiere baja regularización y que la ponderación de clases contribuye a mejorar su rendimiento dado el leve desbalanceo de la variable objetivo.

4.2. Random Forest

De forma análoga:

```
rf_param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 15, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2'],
    'class_weight': [None, 'balanced']
}

rf_model = RandomForestClassifier(random_state=123)
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=123)

rf_grid = GridSearchCV(estimator=rf_model, param_grid=rf_param_grid,
                       scoring=['accuracy', 'roc_auc'], refit='roc_auc',
                       cv=cv, return_train_score=True, n_jobs=1, verbose=2)

rf_grid.fit(X_null_train, y_null_train)
print(rf_grid.best_params_)
# {'class_weight': None, 'max_depth': 15, 'max_features': 'sqrt',
# 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 300}

rf_base = RandomForestClassifier(**rf_grid.best_params_, random_state=123)
```

Como vemos, se definió una parrilla de parámetros con valores diversos para el número de árboles (*n_estimators*), la profundidad máxima (*max_depth*), los mínimos de muestras para dividir nodos y hojas, y dos métodos de selección de características (*max_features*). Así, se pretende explorar modelos poco complejos y otros más flexibles y con mayor capacidad de ajuste. Además, se ha vuelto a evaluar la influencia del ligero desbalanceo de clases en la variable objetivo.

Tras el ajuste de parámetros, vemos que la mejor combinación se obtiene para $n\_estimators = 300$ y $\max\_depth = 15$, lo cual sugiere que el modelo se beneficia de una mayor cantidad de árboles y una profundidad moderada, capaz de capturar relaciones no lineales complejas sin llegar al sobreajuste. Además, vemos que $class\_weight = None$, que indica que el modelo ha sido capaz de trabajar eficientemente sin necesidad de ajustar los pesos de las clases de la variable objetivo.

4.3. XGBoost

Nuevamente:

```
xgb_param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.8, 1],
    'colsample_bytree': [0.8, 1],
    'gamma': [0, 0.1, 0.2],
    'reg_lambda': [1, 5],
    'scale_pos_weight': [1.158]
    }

xgb_model = XGBClassifier(objective='binary:logistic', use_label_encoder=False,
                          eval_metric='logloss', random_state=123)
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=123)

xgb_grid = GridSearchCV(estimator=xgb_model, param_grid=xgb_param_grid,
                        scoring=['accuracy', 'roc_auc'], refit='roc_auc',
                        cv=cv, return_train_score=True, n_jobs=-1, verbose=2)

xgb_grid.fit(X_null_train, y_null_train)
print(xgb_grid.best_params_)
# {'colsample_bytree': 0.8, 'gamma': 0, 'learning_rate': 0.1, 'max_depth': 7,
# 'n_estimators': 200, 'reg_lambda': 1, 'scale_pos_weight': 1.158,
# 'subsample': 0.8}

xgb_base = XGBClassifier(**xgb_grid.best_params_, objective='binary:logistic',
                         use_label_encoder=False, eval_metric='logloss',
                         random_state=123)
```

Una vez más, se definió una parrilla de parámetros que permitiese explorar diferentes niveles de complejidad y estrategias de regularización. Se variaron el número de árboles (*n_estimators*), la profundidad máxima de los árboles (*max_depth*), la tasa de aprendizaje (*learning_rate*) y los parámetros subsampling (*subsample* y *colsample_bytree*) para controlar el sobreajuste. También se ajustaron *gamma* y *reg_lambda* como técnicas de regularización adicionales. Finalmente, se incluyó el parámetro *scale_pos_weight* con el valor 1.158, calculado para compensar el leve desbalanceo de clases presente en el conjunto de datos.

La mejor combinación de parámetros sugiere que el modelo se beneficia de árboles con profundidad moderada (*max_depth=7*), un aprendizaje con tasa intermedia (*learning_rate=0.1*) y una ligera reducción de complejidad mediante subsampling (*subsample=0.8* y *colsample_bytree=0.8*). La regularización adicional (*gamma=0* y *reg_lambda=1*) no resultó necesaria, lo cual es indicativo de que el modelo no presentó un sobreajuste significativo.

4.4. SVM

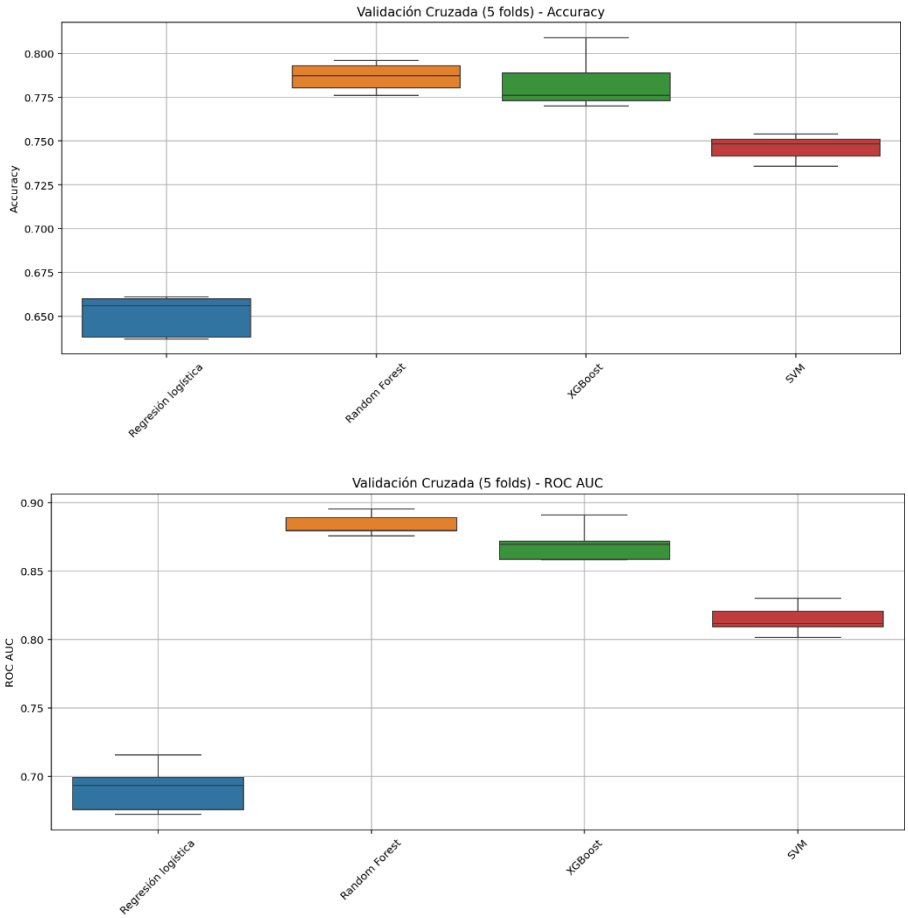
En este caso particular se omite la búsqueda de parámetros porque vamos a reutilizar la máquina de vector soporte que obtuvimos en el apartado 3.1. Tendríamos simplemente:

```
svm_base = SVC(kernel='rbf', C=10.0, gamma=100.0, probability=True,
               random_state=123)
```

Esta elección resulta adecuada puesto que el modelo SVM con kernel RBF y esa elección de parámetros resultó ser el más efectivo entre las variantes de SVM evaluadas.

4.5. Comparación de modelos base

A continuación, se muestran los resultados de una validación cruzada para las métricas *'Accuracy'* y *'AUC'* con los datos de entrenamiento para cada uno de los modelos base generados (se omiten las líneas de código por simplicidad):



Los resultados de la validación cruzada muestran que los modelos Random Forest y XGBoost ofrecen el mejor rendimiento en términos de *accuracy* y *AUC*, con valores muy similares entre ambos. Estos resultados reflejan la capacidad de los modelos de árboles para capturar relaciones no lineales y complejas presentes en los datos.

El modelo SVM con kernel RBF también presenta un rendimiento competitivo, aunque ligeramente inferior al de los modelos de árboles, lo cual puede deberse a las limitaciones impuestas por los parámetros seleccionados o a la naturaleza específica del problema. Finalmente, la regresión logística muestra resultados aceptables, pero inferiores al resto, como era esperable dada su naturaleza lineal.

Independientemente del desempeño individual de cada modelo, la diversidad en la elección de los mismos asegura que el ensamblado *stacking* pueda combinar distintos enfoques de aprendizaje y aprovechar las fortalezas individuales de cada modelo.

4.6. Ensamblado *Stacking*

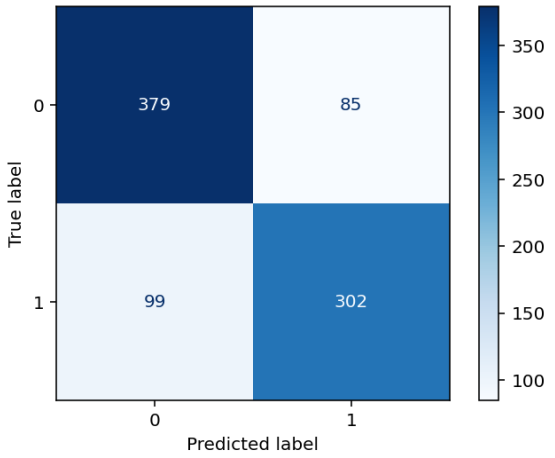
Tal y como se comentó anteriormente, la técnica de ensamblado *Stacking* consiste en tomar las predicciones de los modelos base que hemos definido y usarlas como *inputs* de otro modelo. En este caso particular se aplicará una regresión logística, pues teniendo en cuenta que el número

de predictores es reducido, no se requiere de un modelo altamente complejo para poder capturar patrones y así ganamos en eficiencia computacional. Asimismo, atendiendo de nuevo al poco número de predictores para este modelo final y considerando que ya hemos hecho una optimización de parámetros en cada uno de los modelos base, no se considerará dicho ajuste en este modelo. En su lugar, se utilizará una configuración estándar con regularización *L2* ($C = 1.0$), priorizando estabilidad frente a flexibilidad.

A continuación, se muestran las líneas de código asociadas a la implementación del ensamblado.

```
# Definimos nuevamente los mismos modelos base para que no estén entrenados
estimators = [
    ('lr', LogisticRegression(**lr_grid.best_params_,
                              max_iter=10000,
                              random_state=123)),
    ('rf', RandomForestClassifier(**rf_grid.best_params_, random_state=123)),
    ('xgb', XGBClassifier(**xgb_grid.best_params_, objective='binary:logistic',
                          use_label_encoder=False, eval_metric='logloss',
                          random_state=123)),
    ('svm', SVC(kernel='rbf', C=10.0, gamma=100.0, probability=True))
]

meta_model = LogisticRegression(penalty='l2', C=1.0, solver='lbfgs',
                                max_iter=10000, random_state=123)

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=123)

stacking_model = StackingClassifier(estimators=estimators,
                                    final_estimator=meta_model,
                                    cv = cv, stack_method='predict_proba',
                                    passthrough=False, n_jobs=1, verbose=2)

stacking_model.fit(X_null_train, y_null_train)

y_pred = stacking_model.predict(X_null_test)
y_pred_proba = stacking_model.predict_proba(X_null_test)[:, 1]
```

A continuación, se muestran los resultados de la validación del modelo (se omiten las líneas de código por simplicidad):



Accuracy	ROC-AUC
0.7873	0.8836

Los resultados obtenidos con el ensamblado *stacking* muestran una mejora en ambas métricas respecto a los modelos base individuales. La *accuracy* alcanza un valor de 0.7873 y la métrica *ROC AUC* asciende a 0.8836, superando ligeramente a los valores obtenidos por Random Forest y XGBoost de manera aislada. Esta mejora, aunque modesta, es coherente con lo esperado en ensamblados de este tipo, donde los modelos base ya presentaban un rendimiento elevado.

Además, se analizó la matriz de confusión obtenida para este modelo, observándose que los errores de clasificación están relativamente equilibrados entre las dos clases. Esto indica que el modelo no presenta un sesgo significativo hacia ninguna de las categorías y mantiene un buen equilibrio entre sensibilidad y especificidad, lo que respalda los valores elevados de *AUC* obtenidos.

5. Conclusiones

El desarrollo de este proyecto ha permitido abordar de manera integral todas las fases necesarias para resolver un problema de clasificación binaria en el ámbito de los datos comerciales y técnicos de vehículos. A lo largo del trabajo, se ha combinado el tratamiento riguroso de los datos con la aplicación y comparación de diversos modelos de *machine learning*, siguiendo una metodología reproducible y justificada en cada una de sus etapas.

En primer lugar, el proceso de depuración de datos evidenció la importancia crítica de una limpieza adecuada antes de aplicar modelos predictivos. Se identificaron y eliminaron valores atípicos significativos, particularmente en las variables '*Engine volume*', '*Cylinders*', '*Mileage*' y '*Price*'. Estas acciones no solo eliminaron registros inconsistentes o imposibles desde un punto de vista físico, sino que también ayudaron a prevenir sesgos y problemas de sobreajuste en etapas posteriores. Asimismo, se normalizaron las variables numéricas y se recodificaron las categóricas utilizando *one-hot encoding*, con el fin de preparar un conjunto de datos compatible con los algoritmos seleccionados.

En cuanto al tratamiento de valores perdidos, el análisis detallado de la variable '*Levy*' permitió no solo detectar su patrón de valores faltantes, sino también diseñar dos estrategias de imputación fundamentadas: asignación de valores nulos para representar exención de impuestos y una imputación central basada en agrupaciones por número de cilindros. La comparación posterior de ambas estrategias mostró que el método de imputación no tuvo un impacto significativo en el rendimiento de los modelos, lo que justifica el uso de imputaciones nulas por su simplicidad y capacidad explicativa.

En la fase de modelado, se aplicó inicialmente una Máquina de Vectores Soporte (SVM) con búsqueda de parámetros tanto para un *kernel* lineal como para un *kernel* radial de base gaussiana (RBF). Los resultados indicaron que el *kernel* RBF superó consistentemente al *kernel* lineal en términos de *accuracy* y *AUC*, reflejando la presencia de relaciones no lineales relevantes en los datos. Sin embargo, la aplicación de técnicas de *bagging* no logró mejorar sustancialmente el rendimiento del modelo SVM, lo que sugiere que este algoritmo ya ofrecía una buena combinación entre sesgo y varianza en su configuración óptima.

Posteriormente, se desarrolló un modelo de ensamblado *stacking*, seleccionando como modelos base una regresión logística, un Random Forest, un XGBoost y una SVM con *kernel* RBF. Para cada uno de ellos se realizó una búsqueda de parámetros cuidadosamente diseñada. La regresión logística, aunque mostró un rendimiento aceptable, se vio superada por los modelos no lineales. Random Forest y XGBoost destacaron como los clasificadores con mejor rendimiento individual, seguidos de cerca por la SVM. Esta diversidad en el desempeño de los modelos base confirmó la idoneidad de combinarlos en un ensamblado, maximizando su capacidad colectiva para capturar distintos patrones en los datos.

El modelo *stacking* final, con una regresión logística como meta-modelo, logró una *accuracy* de 0.7873 y un *AUC* de 0.8836, mejorando ligeramente los resultados de los modelos base

individuales. Además, el análisis de la matriz de confusión demostró que los errores de clasificación estaban equilibrados entre las dos clases, lo que indica que el modelo no introdujo sesgos relevantes hacia ninguna categoría y mantuvo una buena capacidad de discriminación.

De manera general, puede concluirse que la estrategia metodológica adoptada (desde la depuración y tratamientos de datos hasta la selección y ajuste de modelos) fue eficaz y permitió alcanzar un rendimiento sólido. El uso de técnicas de ensamblado, especialmente *stacking*, resultó beneficioso incluso cuando los modelos base ya ofrecían métricas elevadas. Este enfoque no solo mejoró ligeramente las métricas de evaluación, sino que también consolidó la estabilidad del modelo y redujo su varianza.

En conjunto, el trabajo realizado ha demostrado que una estrategia metodológica cuidadosamente diseñada es fundamental para desarrollar modelos de clasificación robustos y eficaces. Más allá de las métricas específicas obtenidas, el proyecto ha puesto de manifiesto la importancia de considerar todas las fases del ciclo de *machine learning*, donde el tratamiento de datos, la selección informada de algoritmos y la justificación de cada decisión adoptada son tan relevantes como los resultados finales. Asimismo, los resultados confirman que, incluso cuando los modelos base presentan un alto rendimiento, el uso de técnicas de ensamblado como el *stacking* puede aportar mejoras adicionales en la capacidad de generalización y estabilidad del modelo. En definitiva, el modelo desarrollado ofrece una solución robusta y bien fundamentada al problema planteado, y la metodología seguida sienta unas bases sólidas para abordar futuros proyectos de clasificación supervisada.