# Módulo 11, 12 y 13: Machine Learning


Profesores: Inmaculada Gutiérrez, Daniel Gómez y Javier Castro
Alumno: Pablo Pérez Calvo

# Índice

# 1. Introducción

El objetivo de este proyecto es realizar varios modelos predictivos de clasificación binaria para determinar si un coche debe ser repintado de blanco o negro. El enunciado del problema a resolver y la descripción de la base de datos a utilizar es la siguiente.

**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:

*Precio de venta, Cantidad de Impuestos a pagar, Fabricante, Año de fabricación,Categoría, Interior de cuero, Tipo de combustible, Volumen del motor, Kilometraje, Cilindros, Tipo de caja de cambios, Ruedas motrices, Lugar del volante, Número de Airbags*

Y de la variable dependiente **Color**. La decisión final es si el coche debe pintarse de blanco o no.

# 2. Análisis y depuración de la base de datos

**Ejercicio 1**

Analiza y depura la base de datos proporcionada, justificando todos los procesos de *feature engineering* y su necesidad en posteriores procesos de predicción

En primer lugar, importamos la base de datos que vamos a utilizar llamada "datos_tarea25.xlsx" y eliminamos aquellos registros que estén duplicados y que por tanto, no nos aportarán valor al futuro análisis.

```
data = pd.read_excel('datos_tarea25.xlsx')
data = data.drop_duplicates().reset_index(drop = True)
```

## 2.1. Corrección de tipos en las variables

En segundo lugar, observaremos los tipos de cada una de las variables para ver que los datos se han leído correctamente.

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

Las variables *Levy*, *Engine volume* y *Mileage* aparecen como categóricas cuando deberían ser numéricas, vamos a corregirlo.

- La variable *Levy* se lee como objeto debido a que presenta algunos valores "-", para tener una columna numérica transformaremos estos valores a missings.

- La variable *Engine volume* presenta algunos valores numéricos seguidos de la palabra "Turbo", debido a esto crearemos otra nueva variable que indique si el motor tiene turbo o no. De modo que nos quedaría la variable *Engine volume* con valores numéricos y una nueva variable llamada *Engine type* que indica si tiene turbo.

- La variable *Mileage* presenta la palabra "km" tras el número de kilómetros recorridos. Como todos los registros de esta variable tienen la misma unida de medida, kilómetros, procedemos a eliminar esta palabra para obtener una columna numérica.

- La variable *Cylinders* al tener pocas categorías, la pasaremos de númerica a categórica.

```
# Correcion de errores en el tipo

data['Levy'] = pd.to_numeric(data['Levy'], errors='coerce')  # Convierte y pone NaN donde no
    pueda

# Creacion de una nueva variable para determinar si tiene turbo o no
data[['Engine volume', 'Engine type']] = data['Engine volume'].str.split(' ', n=1, expand=True)
data['Engine volume'] = data['Engine volume'].astype(float)
data['Engine type'] = data['Engine type'].fillna('No Turbo')

data['Mileage']=data['Mileage'].str.replace('km',' ').astype('int')

data['Cylinders']=data['Cylinders'].astype('object')
```

## 2.2. Análisis descriptivo de las variables numéricas y detección de outliers.

Una vez corregido los errores de los tipos de cada variable, realizaremos un análisis descriptivo de las variables numéricas con el uso de la función *describe()*.

| Index | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| Price | 2806 | 23913.6 | 26028.7 | 3 | 9408 | 18821.5 | 31361 | 627220 |
| Levy | 2172 | 900.43 | 505.682 | 87 | 639 | 781 | 1053 | 11714 |
| Prod. year | 2806 | 2012.81 | 3.92638 | 1943 | 2011 | 2013 | 2015 | 2020 |
| Engine volume | 2806 | 2.37965 | 0.902089 | 0 | 1.8 | 2 | 2.5 | 6.3 |
| Mileage | 2806 | 521707 | 2.09735e+07 | 0 | 62920.8 | 113600 | 165166 | 1.11111e+09 |
| Airbags | 2806 | 7.53671 | 4.20846 | 0 | 4 | 8 | 12 | 16 |

Figura 1: Análisis descriptivo de las variables cuantitativas

Para analizar mejor la presencia de outliers, vamos a representar gráficamente los boxplots de cada una de las variables para observar sus distribuciones.

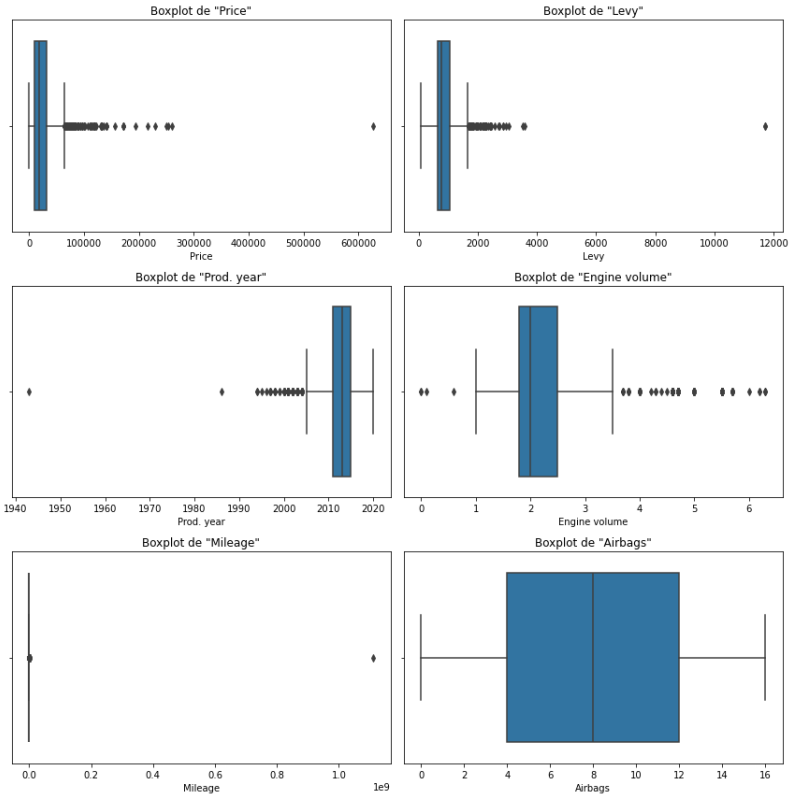



Figura 2: Distribución de las variables numéricas

En el gráfico anterior se pueden identificar claramente valores anómalos en todas las variables excepto en *"Airbags"*.

La manera de proceder con estos valores atípicos será tratar como valores missing todos aquellos que se encuentren fuera del rango intercuartílico.

```
for column in data[numericas].columns:
    # Calculo del rango intercuartilico
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1

    # Todos los datos fuera del rango intercuartilico los tratamos como missings
    data[column] = data[column].where((data[column] >= Q1 - 1.5 * IQR) & (data[column] <= Q3 +
        1.5 * IQR), np.nan)
```

## 2.3. Análisis de la distribución de las variables categóricas y agrupación de categorías.

Análogamente realizaremos ahora un análisis de la distribución de las variables categóricas haciendo uso de la función *countplot()*, con el objetivo de observar categorías poco representadas y agruparlas para reducir el número de categorías.

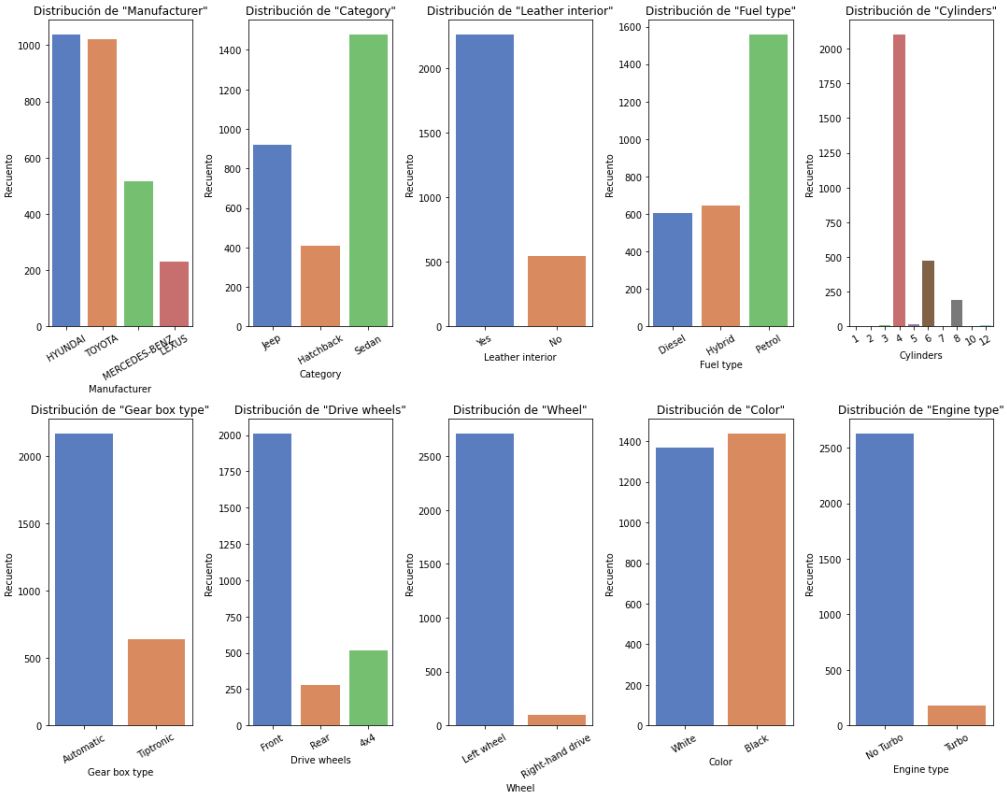



Figura 3: Distribución de variables categóricas

Con el objetivo de quedarnos como máximo con 3 categorías en cada variable, agruparemos "Mercedes-Benz" y "Lexus" de la columna "Manufacturer" en una categoría llamada "Otro". Además, como en la variable "Drive-wheels" hay dos categorías poco representadas, "Roar" y "4x4", en comparación con "Front", las agruparemos en otra categoría llamada "Otro". La categorías de "Cylinders" se agruparan en dos, menos de 5 y 5 o más.

```
data['Manufacturer'] = data['Manufacturer'].replace({'MERCEDES-BENZ': 'Otro', 'LEXUS': 'Otro'})
data['Drive wheels'] = data['Drive wheels'].replace({'4x4': 'Otro', 'Rear': 'Otro'})
data['Cylinders'] = data['Cylinders'].replace({1: 'Menos de 5', 2: 'Menos de 5', 3: 'Menos de 5'
    , 4: 'Menos de 5', 5: '5 o mas', 6: '5 o mas', 7: '5 o mas', 8: '5 o mas', 9: '5 o mas', 10:
     '5 o mas', 12: '5 o mas'})
```

## 2.4. Tratamiento de valores perdidos.

Primero de todo, veamos cual es la distribución de missing values y la proporción respecto al total en nuestro conjunto de datos actual.

```
missings = data.isnull().sum()
prop_missing= missings/len(data)

missing_values = pd.DataFrame({
    'Valores omitidos': missings,
    '% del total': prop_missing.round(2)})
```

| Index | Valores omitidos | % del total |
|---|---|---|
| Price | 129 | 0.05 |
| Levy | 741 | 0.26 |
| Manufacturer | 0 | 0 |
| Prod. year | 93 | 0.03 |
| Category | 0 | 0 |
| Leather interior | 0 | 0 |
| Fuel type | 0 | 0 |
| Engine volume | 203 | 0.07 |
| Mileage | 81 | 0.03 |
| Cylinders | 0 | 0 |
| Gear box type | 0 | 0 |
| Drive wheels | 0 | 0 |
| Wheel | 0 | 0 |
| Color | 0 | 0 |
| Airbags | 0 | 0 |
| Engine type | 0 | 0 |

Figura 4: Recuento de missing values

Los valores perdidos que encontramos son aquellos que pasamos a missing cuando los identificamos como outliers en la sección 2.2 y los valores de "Levy" que inicialmente se presentaban como "-". La variable con más valores perdidos es "Levy" pero como ninguna supera el $50\,\%$ no eliminaremos ninguna variable.

Estos valores missings los vamos a imputar con la técnica de los $K$ vecinos más cercanos.

```
# Imputacion de datos perdidos mediante KNN
imputer = KNNImputer(n_neighbors=3, metric='nan_euclidean')
df = data.copy()
data_imputed = pd.DataFrame(imputer.fit_transform(df[numericas]))
# Recuperamos los nombres de las columnas
data_imputed=data_imputed.set_axis([numericas], axis=1)
```

Finalmente volvemos a agrupar en un mismo dataframe, las variables numéricas sin missings junto a las variables categóricas.

```
data = pd.concat([data_imputed, data[categoricas]], axis=1)
data = data.set_axis(variables, axis=1)
```

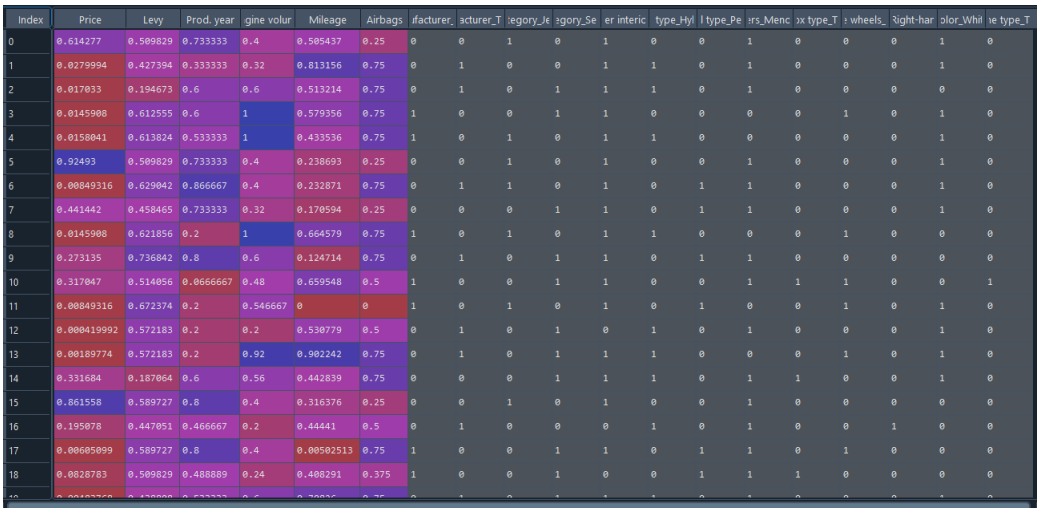
Por otro lado una de las fases más importantes del *feature engineering* es la estandarización de las variables numéricas. Aplicaremos el método MinMaxScaler() para asegurarnos que los valores sean no negativos.

```
# Aplicar MinMaxScaler para asegurarse de que los valores sean no negativos
scaler = MinMaxScaler()

# Escalamos las variables numericas
num_scaled = scaler.fit_transform(data[numericas])
num_scaled = pd.DataFrame(num_scaled)

# Unimos las columnas numericas escaladas con las categoricas
df_depurada = pd.concat([num_scaled, data[categoricas]], axis=1)

# Recuperamos los nombres de las columnas
df_depurada = df_depurada.set_axis(variables, axis=1)
```

Por último, convertimos en dummies todas las variables categóricas.

```
# Creamos dummies para las variables categoricas
df_depurada_dummies = pd.get_dummies(df_depurada,columns=categoricas, drop_first= True)
```

Ya tenemos nuestra base de datos depurada, estandarizada y con variables dummies preparada para realizar el análisis en los siguiente apartados.



| Index | Price | Levy | Prod. year | gine volur | Mileage | Airbags | facturer_ | acturer_T | egory_Je | egory_Se | er interic | type_Hyl | l type_Pe | ers_Menc | ox type_T | s wheels_ | Right-har | olor_Whit | ne type_T |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.614277 | 0.509829 | 0.733333 | 0.4 | 0.505437 | 0.25 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0.0279994 | 0.427394 | 0.333333 | 0.32 | 0.813156 | 0.75 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0.017033 | 0.194673 | 0.6 | 0.6 | 0.513214 | 0.75 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0.0145908 | 0.612555 | 0.6 | 1 | 0.579356 | 0.75 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0.0158041 | 0.613824 | 0.533333 | 1 | 0.433536 | 0.75 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 5 | 0.92493 | 0.509829 | 0.733333 | 0.4 | 0.238693 | 0.25 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 6 | 0.00849316 | 0.629042 | 0.866667 | 0.4 | 0.232871 | 0.75 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 7 | 0.441442 | 0.458465 | 0.733333 | 0.32 | 0.170594 | 0.25 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 8 | 0.0145908 | 0.621856 | 0.2 | 1 | 0.664579 | 0.75 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 9 | 0.273135 | 0.736842 | 0.8 | 0.6 | 0.124714 | 0.75 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0.317047 | 0.514056 | 0.0666667 | 0.48 | 0.659548 | 0.5 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 11 | 0.00849316 | 0.672374 | 0.2 | 0.546667 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 12 | 0.000419992 | 0.572183 | 0.2 | 0.2 | 0.530779 | 0.5 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 13 | 0.00189774 | 0.572183 | 0.2 | 0.92 | 0.902242 | 0.75 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 14 | 0.331684 | 0.187064 | 0.6 | 0.56 | 0.442839 | 0.75 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 15 | 0.861558 | 0.589727 | 0.8 | 0.4 | 0.316376 | 0.25 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0.195078 | 0.447051 | 0.466667 | 0.2 | 0.44441 | 0.5 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 17 | 0.00605099 | 0.589727 | 0.8 | 0.4 | 0.00502513 | 0.75 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 18 | 0.0828783 | 0.509829 | 0.488889 | 0.24 | 0.408291 | 0.375 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

Figura 5: Muestra de la base de datos depurada

# 3. Modelo SVM

**Ejercicio 2**

Obtener el mejor modelo posible con una máquina de vector soporte. Para ello deberás realizar un ajuste paramétrico del SVM con al menos dos kernels haciendo una representación gráfica de esa búsqueda usando como medidas de precisión accuracy y AUC. Justificar todos los pasos que vas haciendo, así como la comparación entre los dos mejores kernels encontrados. Realizar y ajustar un bagging al mejor modelo encontrado explicando cual es el objetivo y resultado que se obtiene.

En primer lugar, separamos la variable objetivo "Color", en este caso al pasarla a dummies "Color_White", del dataframe depurado.

```
X = df_depurada_dummies.drop(columns=['Color_White'])
y = df_depurada_dummies['Color_White']
```

En segundo lugar, seleccionaremos las $9$ variables más importantes mediante el criterio del test chi-cuadrado, debido a que las variables númericas están escaladas entre $0$ y $1$.

```
selector = SelectKBest(score_func=chi2, k=9)
X_new = selector.fit_transform(X, y)

# Obtener los nombres de las caracteristicas seleccionadas
selected_columns = X.columns[selector.get_support()]
X = df_depurada_dummies[selected_columns]

print("Variables seleccionadas:", selected_columns)
```

```
Variables seleccionadas: Index(['Engine volume', 'Manufacturer_Otro', 'Category_Sedan','
    Fuel type_Hybrid', 'Fuel type_Petrol', 'Cylinders_Menos de 5', 'Gear box type_Tiptronic
    ', 'Drive wheels_Otro', 'Engine type_Turbo'], dtype='object')
```

Por último, dividimos el conjunto de datos en train y test, un $70\,\%$ para entrenamiento y un $30\,\%$ para test.

```
seed = 123
[X_train, X_test, y_train, y_test] = train_test_split(X, y, test_size = 0.30, random_state =
    seed, stratify=y)
```

## 3.1. Búsqueda con Kernel lineal

En el caso del kernel lineal el único parámetro significativo es $C$ que representa la tolerancia a errores o la penalización a las malas clasificaciones. Vamos a buscar cual es el $C$ con el que se consiguen mejores resultados mediante un *GridSearch*.

```
# Rango del parametro C
param_grid_lineal = {'C': [0.1, 0.2, 0.3, 0.4, 0.5, 1, 2, 5, 10, 50, 100] }

grid = GridSearchCV(SVC(kernel='linear', random_state = seed), param_grid_lineal, refit = True,
    cv=10, verbose = 3)

# Entrenamos y buscamos en TRAIN
resultados = grid.fit(X_train, y_train)
```

```
print(grid.best_params_)
```

```
{'C': 1}
```

Como el parámetro con el que se ha obtenido mayor accuracy es con $C = 1$ buscaremos en un entorno más cercano a este.

```
param_grid_lineal = {'C': [0.25, 0.5, 0.75, 0.9, 1, 1.1, 1.25, 1.5, 1.75]}
```

Creamos un gráfico de dispersión del accuracy medio para cada valor del parámetro.

```
aux = pd.DataFrame(resultados.cv_results_)
plt.scatter(aux[['param_C']], aux[['mean_test_score']], color='b', alpha=0.9)
plt.xlabel('Parametro C')
plt.ylabel('Accuracy ')
plt.xlim(0, 2)
plt.title('Precision media SVM en funcion del parametro C ')
plt.show()
```

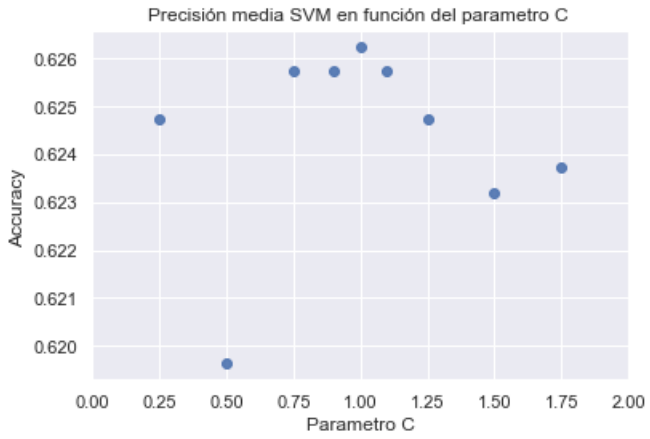



Figura 6: Búsqueda de parámetros kernel lineal

Oberservamos en el gráfico que para $C = 1$ se obtiene el mayor valor de accuracy.

```
print(grid.best_params_)
```

```
{'C': 1}
```

Usamos ahora el mejor modelo lineal para predecir y obtenemos las métricas del mismo.

```
grid_predictions = grid.predict(X_test)
```

Los resultados del modelo lineal son los siguientes.

```
print(classification_report(y_test, grid_predictions))
              precision    recall  f1-score   support

           0       0.66      0.66      0.66       432
           1       0.64      0.64      0.64       410

accuracy                           0.65       842
macro avg      0.65      0.65      0.65       842
weighted avg   0.65      0.65      0.65       842
print('accuracy' , accuracy)
accuracy 0.6520190023752969
print('AUC', auc)
AUC 0.6901168699186992
```

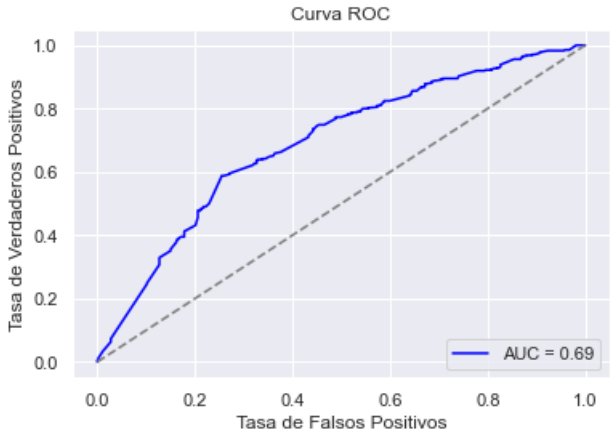



Figura 7: Curva ROC mejor modelo lineal

## 3.2. Búsqueda con Kernel Gaussiano

En el caso del kernel gaussiano, además del parámetro $C$ también existe el parámetro *gamma* el cual controla la influencia de cada punto en la clasificación.

```
param_grid_gausiano = {'C': [0.1,0.5, 1, 2, 5, 10, 100],
             'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['rbf']}

grid_gausiano = GridSearchCV(SVC(), param_grid_gausiano, refit = True, cv=5, verbose = 3)
# Entrenamos y buscamos en TRAIN
resultados = grid_gausiano.fit(X_train, y_train)
```

```
print(grid_gausiano.best_params_)
```

```
{'C': 2, 'gamma': 1, 'kernel': 'rbf'}
```

Obtenemos los mejores valores de accuracy para $C = 2$ y *gamma* $= 1$, vamos a seguir buscando en un entorno de estos valores.

```
param_grid_gausiano =  {'C': [1, 1.25, 1.5, 1.75, 1.9, 2, 2.1, 2.25, 2.5, 2.75, 3],
        'gamma': [0.5 ,0.75 ,1 ,1.25, 1.5],
        'kernel': ['rbf']}
```

Mostramos graficamente los valores medio de accuracy para cada valor de los parámetros.

```
# Graficamos los resultados del modelo gaussiano
aux = pd.DataFrame(resultados.cv_results_)

# Factorizamos la variable sigma en categorias
categorias, valores_enteros = pd.factorize(aux['param_gamma'])

# Creamos el grafico de dispersion con colores basados en las categorias de sigma
plt.scatter(aux[['param_C']], aux[['mean_test_score']], c=categorias , cmap='viridis')
plt.xlabel('Parametro C')
plt.ylabel('Accuracy ')
plt.xlim(0, 2)
plt.title('Precision media SVM en funcion del parametro C ')

plt.show()
```

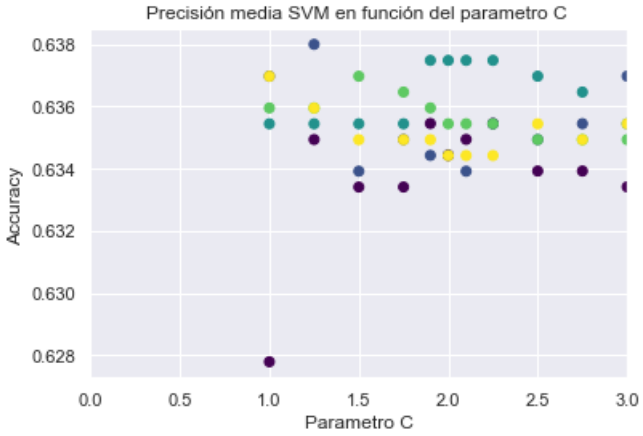



Figura 8: Búsqueda de parámetros kernel gaussiano

El mejor resultado obtenido es con la siguiente configuración de parámetros.

```
print(grid_gausiano.best_params_)
```

```
{'C': 1.25, 'gamma': 0.75, 'kernel': 'rbf'}
```

Si entrenamos ahora usando el mejor modelo encontrado tenemos estos resultados.

```
grid_predictions = grid_gausiano.predict(X_test)

cm = confusion_matrix(y_test, grid_predictions)
accuracy=(cm[0,0]+cm[1,1])/(cm[0,1]+cm[1,1]+ cm[1,0]+cm[0,0])
y_scores = grid_gausiano.decision_function(X_test)
auc = roc_auc_score(y_test, y_scores)
```

```
print(classification_report(y_test, grid_predictions))
              precision    recall  f1-score   support

           0       0.65      0.61      0.63       432
           1       0.61      0.66      0.63       410

    accuracy                           0.63       842
   macro avg       0.63      0.63      0.63       842
weighted avg       0.63      0.63      0.63       842

print('accuracy' , accuracy)
accuracy 0.6318289786223278
print('AUC', auc)
AUC 0.6668275745257453
```

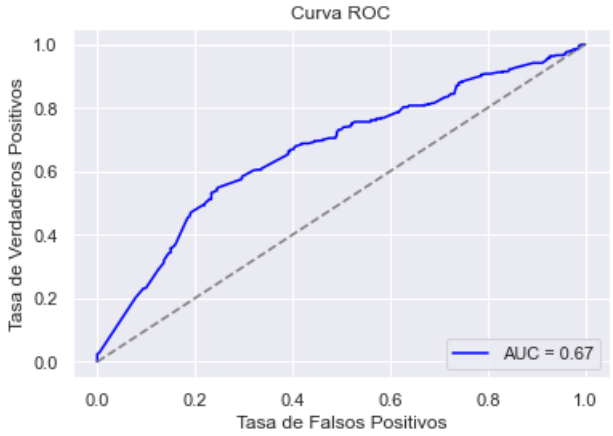



Figura 9: Curva ROC mejor modelo kernel gaussiano

Como el modelo SVM con kernel lineal ofrece mayor valor de accuracy y AUC que el creado con el kernel gaussiano, lo usaremos como modelo ganador.

```
best_svm = best_linear_svm

best_svm.fit(X_train,y_train)
print('Model test Score: %.3f, ' %best_svm.score(X_test, y_test),
      'Model training Score: %.3f' %best_svm.score(X_train, y_train))
```

```
Model test Score: 0.652,  Model training Score: 0.630
```

Podemos ver como el accuracy para el conjunto de entrenamiento y de prueba son similares, lo que nos indica que es un buen modelo en términos de ajuste.

### 3.3. Ensamblado de Bagging

Vamos a utilizar como modelo base el modelo lineal ganador del anterior apartado y utilizaremos la función *BaggingClassifier()* con 100 estimadores.

```
bagging = BaggingClassifier(best_svm, n_estimators=100, random_state=seed)
# Entrenamos el bagging
bagging.fit(X_train, y_train)

# Resultados
print('Model test Score: %.3f, ' %bagging.score(X_test, y_test),
      'Model training Score: %.3f' %bagging.score(X_train, y_train))
```

```
Model test Score: 0.646,  Model training Score: 0.633
```

Podemos observar con estos resultados que el Bagging no aporta mejora significativa al modelo base, es más, el modelo base generaliza mejor. Como el modelo base ya es estable no se beneficia del bagging.

## 4. Modelo Stacking

**Ejercicio 3**

Realizar un modelo de stacking en profundidad, comparando en primer lugar los clasificadores base empleados, así como el resultado final del stacking obtenido. Explica bien todos los pasos que das.

Para realizar el stacking utilizaremos 3 modelos bases diversos: un modelo lineal *LogisticRegression*, un modelo basado en distancias *KNN* y uno basado en árboles *RandomForest*. Es una buena selección de modelos porque cada uno aprende de manera distinta y no habrá redundancia.

El meta-modelo usado será un *LogisticRegression* debido a su alta explicabilidad y generalización.

```
# Inicializar los modelos base
model_1 = LogisticRegression()
model_2 =  KNeighborsClassifier(n_neighbors=5) #GradientBoostingClassifier()
model_3 = RandomForestClassifier() #GaussianNB()#DecisionTreeClassifier()

all_models = [('lr', model_1), ('Kn', model_2), ('rf', model_3)]

# Inicializar el modelo de stacking
stacking_model = StackingClassifier(
    estimators= all_models,
    final_estimator=  LogisticRegression(solver='liblinear') #best_svm
)
```

Procedemos a entrenar el modelo y a predecir los valores para el conjunto de prueba.

```
# Entrenar el modelo de stacking
stacking_model.fit(X_train, y_train)

# Realizar predicciones en el conjunto de prueba
y_pred = stacking_model.predict(X_test)
```

Los resultados del stacking son los siguientes.

```
print(classification_report(y_test, y_pred))
              precision    recall  f1-score   support

           0       0.66      0.71      0.68       432
           1       0.67      0.61      0.64       410

    accuracy                           0.66       842
   macro avg       0.66      0.66      0.66       842
weighted avg       0.66      0.66      0.66       842
print("Accuracy:", accuracy_score(y_test, y_pred))
Accuracy: 0.66270783847981
auc = roc_auc_score(y_test, y_pred)
print('AUC', auc)
AUC 0.6613595302619694
```

Podemos ver como obtenemos buenos valores de precision tanto para la clase $0$ como para la clase $1$. Veamos ahora la influencia y el comportamiento de los modelos bases en el stacking detalladamente.

```
# Obtener las caracteristicas metaaprendidas por cada clasificador base
X_train_meta = stacking_model.transform(X_train)

# Evaluar el rendimiento de cada clasificador base individualmente
for (name, clf) in stacking_model.named_estimators_.items():
    y_pred_base = clf.predict(X_test)
    print(f"\nResultados del Clasificador Base {name}:")
    print(classification_report(y_test, y_pred_base))
```

```
Resultados del Clasificador Base lr:
              precision    recall  f1-score   support

           0       0.66      0.64      0.65       432
           1       0.64      0.66      0.65       410

    accuracy                           0.65       842
   macro avg       0.65      0.65      0.65       842
weighted avg       0.65      0.65      0.65       842


Resultados del Clasificador Base Kn:
              precision    recall  f1-score   support

           0       0.69      0.55      0.61       432
           1       0.61      0.74      0.67       410

    accuracy                           0.64       842
   macro avg       0.65      0.64      0.64       842
weighted avg       0.65      0.64      0.64       842


Resultados del Clasificador Base rf:
              precision    recall  f1-score   support

           0       0.67      0.63      0.65       432
           1       0.63      0.67      0.65       410

    accuracy                           0.65       842
   macro avg       0.65      0.65      0.65       842
weighted avg       0.65      0.65      0.65       842
```

Tanto *LogisticRegression* como *Random Forest* son modelos equilibrados con precision y recall parejos

en ambas clases. El modelo *KNN*, sin embargo, favorece a la clase 1. Esta diversidad de errores enriquece el stacking porque cada modelo aporta algo distinto.

Por último, evaluaremos el stacking mediante validación cruzada para tener una visión mejor de la validez del modelo.

```
cv = StratifiedKFold ( n_splits =5 , shuffle = True , random_state = seed )

 # Obtener predicciones de la validacion cruzada
y_pred_cv_train = cross_val_predict ( stacking_model , X_train , y_train , cv = cv ,
method ='predict')
y_pred_cv_test = cross_val_predict ( stacking_model , X_test , y_test , cv = cv ,
method ='predict')

# Calcular metricas
accuracy = accuracy_score( y_train , y_pred_cv_train )
precision = precision_score( y_train , y_pred_cv_train )
recall = recall_score( y_train , y_pred_cv_train )
f1 = f1_score( y_train , y_pred_cv_train )

# Imprimir las m tricas
print('M tricas en train:')
print(f'Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}\n')


accuracy = accuracy_score( y_test , y_pred_cv_test )
precision = precision_score( y_test , y_pred_cv_test )
recall = recall_score( y_test , y_pred_cv_test )
f1 = f1_score( y_test , y_pred_cv_test )


print('Metricas en test:')
print(f'Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
```

```
Metricas en train:
Accuracy: 0.6247454175152749
Precision: 0.6145077720207254
Recall: 0.6189979123173278
F1 Score: 0.6167446697867915

Metricas en test:
Accuracy: 0.6353919239904988
Precision: 0.6235011990407674
Recall: 0.6341463414634146
F1 Score: 0.6287787182587666
```

Los resultados obtenidos tras la validación cruzada nos indican que el stacking es sólido y equilibrado. No hay ninguna métrica que sobresalga de las otras y estas son similares para test y para train por lo que indica un buen ajuste.

# 5. Estudio comparativo

**Ejercicio 4**

Desarrolla un proceso comparativo completo de los modelos y procesos seleccionados en los apartados 2) y 3), justificando y argumentando las decisiones y resultados.

Para realizar el proceso comparativo completo de los modelos estudiados anteriormente, entrenaremos cada uno con el mismo conjunto de datos y luego obtendremos sus métricas de predicción: accuracy, precission, recall y F1 score.

| Modelos | Accuracy | Precission | Recall | F1 score |
|---|---|---|---|---|
| SVM Lineal | **0.6520** | **0.6430** | **0.6415** | **0.6422** |
| SVM Gaussiano | **0.6318** | **0.6142** | **0.6561** | **0.6345** |
| Bagging | **0.6461** | **0.6353** | **0.6415** | **0.6383** |
| Stacking | **0.6603** | **0.6615** | **0.6195** | **0.6398** |

El stacking es el que presenta mayor accuracy, precisión y F1 score, sin embargo, la mejora respecto al modelo simple de SVM con kernel lineal no es significativa.

El SVM Gaussiano es el que ofrece mayor sensibilidad pero es el peor en las demás métricas. Entre los dos modelos SVM, nos quedamos con el del kernel lineal como ya vimos en el apartado 2. Además, al hacer el bagging utilizando este svm lineal como modelo base, no mejoramos el valor en ninguna métrica.

Por lo comentado anteriormente, el mejor modelo creado es el stacking que aunque tiene el peor valor de recall, esto no es una gran desventaja debido a que el objetivo del problema es clasificar el color de un coche, este problema no es sensible a falsos negativos como lo sería predecir enfermedades. Aunque viendo las buenas soluciones que aporta el modelo simple de SVM Lineal, siendo insignificante la mejora en el stacking, un modelo más complejo, puede considerarse un modelo igualmente válido.