Aprendizaje Automático.

Práctica 1. Procesamiento de datos y aprendizaje supervisado.

Autor: Marcos Tendero Carmona. **Fecha:** 26/12/2024

Contenido

Introducción

El desarrollo de la primera práctica de la asignatura tiene como objetivo la puesta en práctica de los conocimientos adquiridos en las unidades 1 y 2 referentes al empleo de algoritmos de aprendizaje supervisado para la resolución de problemas.

Recordamos que el aprendizaje supervisado es un enfoque de aprendizaje automático en el que un modelo aprende a partir de un conjunto de datos etiquetados, es decir, datos que incluyen tanto las entradas como las salidas correctas. En este tipo de aprendizaje, el modelo es entrenado para predecir o clasificar nuevas observaciones basándose en ejemplos previos.

En concreto, se propone la resolución de dos tareas, correspondientes a los dos tipos de problemas de aprendizaje supervisado estudiados: el ejercicio 1 corresponde a un problema de regresión, en el que se predice una variable continua, mientras que el segundo ejercicio consiste en problema de clasificación, en el que se asignan las observaciones a categorías o clases específicas.

El desarrollo técnico de la práctica se ha llevado a cabo en Python, en un Jupyter notebook, siguiendo como referencia el cuaderno de la unidad 2. *En esta memoria se comentan las partes significativas del código. El código completo se adjunta en la carpeta de la entrega.*

Ejercicio 1: estimación del precio de coches usados

En el primer ejercicio, queremos estimar el precio de venta de un coche de segunda mano en base a sus características. Para ello planteamos entrenar un modelo en base a datos históricos de tasación de vehículos, que asocian las características de estos a su precio de venta. Esta descripción nos hace entender que nos encontramos ante un **problema de regresión**.

El dataset de tasación de vehículos con el que trabajamos en este ejercicio contiene las siguientes columnas:

Columna	**Descripción**
ID	Identificador del coche
marca	Marca del vehículo (Ej. Audi, BMW, Skoda...)
modelo	Modelo del coche (Ej. A7)
fecha	Fecha de fabricación del vehículo, informa sobre su antigüedad
tipo_cambio	Tipo de cambio del coche (manual, automático...)
total_km	Kilometraje actual del coche
tipo_combustible	Tipo de combustible (diésel, gasolina...)
consumo	Consumo de combustible del coche, medido en litros por cada 100 km
tipo_motor	Tamaño del motor, relacionado con la potencia (Ej. 3.0)
tasa	Impuesto necesario para circular con el coche, medido en euros (€)
precio	Precio de venta del coche de segunda mano, **variable objetivo**.

El conjunto de datos está dividido en un dataset de entrenamiento del modelo, dataset_coches_train.csv, de 4960 entradas y un conjunto de testeo de 2672 registros, dataset_coches_test.csv, con el que evaluaremos la eficiencia y la capacidad de generalización del modelo.

El conjunto de datos de trabajo, tanto de este ejercicio como del siguiente, viene previamente dividido en dos archivos, por lo que realizaremos el proceso de carga de manera directa.

Modelo propuesto

Se nos pide un modelo de aprendizaje que permita predecir la variable predio con un error absoluto medio (MAE) inferior a 3000€ en el conjunto de testeo.

Se ha propuesto el uso del algoritmo Random Forest, que combina (ensambla) múltiples árboles de decisión que, aplicado a problemas de regresión, predicen un valor continuo para cada entrada dada, calculándose el resultado final como el promedio de los valores predichos por todos los árboles.

A continuación, se muestra la implementación en código del modelo completo:

- En primer lugar, cargamos ambos datasets CSV (entrenamiento y testeo) en variables con Pandas.

```
#Carga de datos
train_cars = pd.read_csv('datasets/dataset_coches_train.csv')
test_cars = pd.read_csv('datasets/dataset_coches_test.csv')
```

Podemos ver información del dataset con funciones de Pandas como *info()* o *describe()*.

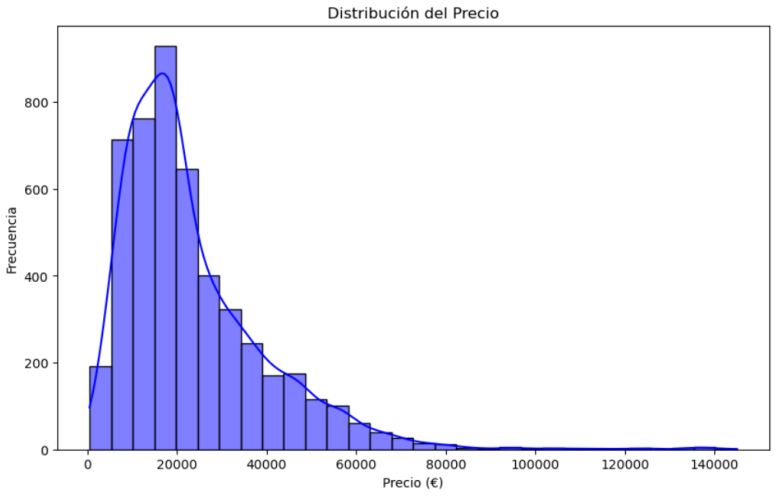
Una vez hemos cargado los datos, comenzamos el análisis contestando a las cuestiones propuestas en el orden en el que se han resuelto en el script:

- ### Pregunta 1. Representar gráficamente la distribución de la variable precio en el conjunto de entrenamiento.

  Para ello, usamos las librerías *matplotlib* y *seaborn* para la visualización gráfica de la distribución de la columna precio de nuestro dataset de entrenamiento. Imprimimos la distribución como un histograma de frecuencia de precios con su estimación de densidad de Kernel:

```
#P1: Distribución de la variable objetivo
plt.figure(figsize=(10, 6))
sns.histplot(train_cars['precio'], kde=True, bins=30, color='blue')
plt.title('Distribución del Precio')
plt.xlabel('Precio (€)')
plt.ylabel('Frecuencia')
plt.show()
```

  Obtenemos la siguiente salida, observando que el precio más repetido se concentra entre los 17500 y los 2000 €:



- Pregunta 2. ¿Cuál es la marca más cara en promedio? ¿Y la más barata? Utilice el conjunto de entrenamiento.

  Para esto, realizamos una secuencia de operaciones de agregación sencillas usando funciones de DataFrames de Pandas: agrupamos el conjunto por precio, nos quedamos con la columna de precios y calculamos el promedio por marca. Una vez hemos realizado esta operación, segmentamos los precios máximo y mínimo con las funciones *max()* y *min()* (y el nombre de la marca con los índices asociados a estos valores, *idxmax()* y *idxmin()*):

```
#P2: Marca más cara y más barata
mean_price_by_brand = train_cars.groupby('marca')['precio'].mean()
print("Marca más barata en promedio:", mean_price_by_brand.idxmin(), "- Precio promedio:", mean_price_by_brand.min(), "€")
print("Marca más cara en promedio:", mean_price_by_brand.idxmax(), "- Precio promedio:", mean_price_by_brand.max(), "€")
```

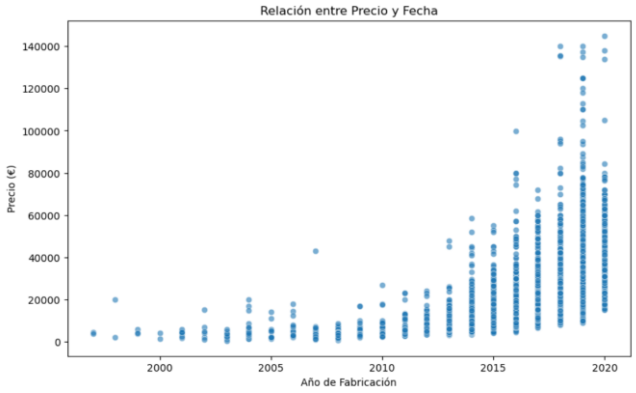
  La salida es:

```
Marca más barata en promedio: vauxhall - Precio promedio: 11884.520146520146 €
Marca más cara en promedio: audi - Precio promedio: 42330.967930029154 €
```

- Pregunta 3. Representar gráficamente la dependencia entre el precio y la fecha. Utilice el conjunto de entrenamiento.

  Esta vez usamos *scatterplot* de *seaborn* con *x=fecha* e *y=precio*. Podemos observar una tendencia de crecimiento casi exponencial del precio de los vehículos con el paso del tiempo:

```
#P3: Relación entre precio y fecha
plt.figure(figsize=(10, 6))
sns.scatterplot(x=train_cars['fecha'], y=train_cars['precio'], alpha=0.6)
plt.title('Relación entre Precio y Fecha')
plt.xlabel('Año de Fabricación')
plt.ylabel('Precio (€)')
plt.show()
```



Continuamos con la implementación del algoritmo Random Forest. En primer lugar, tenemos que hacer un procesamiento de datos para poder emplear el modelo, consistente en codificar las variables categóricas del dataset, ya que en *scikit-learn* los modelos requieren que todas las características sean numéricas. Para ello, usamos codificación One Hot con la función *OneHotEncoder* de *scikit-learn (NOTA: he leído que el uso de drop=first no es necesario para el algoritmo Random Forest ya que no es sensible a colinealidad, pero es recomendable si se plantea expandir el análisis a otros modelos que si lo fueran, como regresión lineal o logística)*. Se usa la función *get_feature_names_out* para la generación de nombres para las nuevas columnas binarias. Por otro lado, eliminamos también columnas que no necesitamos en el análisis, como el ID.

```
#Procesamiento de datos
#Eliminamos columnas no relevantes
train_cars = train_cars.drop(['ID'], axis=1)
test_cars = test_cars.drop(['ID'], axis=1)

#Codificación de variables categóricas
categorical_features = ['marca', 'modelo', 'tipo_cambio', 'tipo_combustible']
encoder = OneHotEncoder(drop='first', sparse_output=False)
encoded_train = pd.DataFrame(encoder.fit_transform(train_cars[categorical_features]))
encoded_test = pd.DataFrame(encoder.transform(test_cars[categorical_features]))

#Las agregamos al dataset
encoded_train.columns = encoder.get_feature_names_out(categorical_features)
encoded_test.columns = encoder.get_feature_names_out(categorical_features)
train_cars = pd.concat([train_cars.drop(categorical_features, axis=1), encoded_train], axis=1)
test_cars = pd.concat([test_cars.drop(categorical_features, axis=1), encoded_test], axis=1)
```

Fijémonos en el uso de las funciones *fit_transform* y *trasform* para hacer la transformación de los datasets con el codificador. La primera hace que el conjunto de datos 'aprenda' nuevos parámetros añadidos por el encoder, se aplica al conjunto de entrenamiento. La segunda se usa en el conjunto de testeo y simplemente usa los parámetros aprendidos en el conjunto de entrenamiento para la transformación.

Por otro lado, se normalizan las variables numéricas. *He probado a usar la normalización estándar, para diferir del notebook de referencia de la Unidad 2*:

```
#Normalización estándar de variables numéricas
scaler = StandardScaler()
numerical_features = ['total_km', 'consumo', 'tipo_motor', 'tasa']
train_cars[numerical_features] = scaler.fit_transform(train_cars[numerical_features])
test_cars[numerical_features] = scaler.transform(test_cars[numerical_features])
```

Finalmente, separamos las variables características del target y entrenamos el modelo con el conjunto de entrenamiento:

```
#Separación de X (características) e Y (target)
X_train = train_cars.drop('precio', axis=1)
y_train = train_cars['precio']
X_test = test_cars.drop('precio', axis=1)
y_test = test_cars['precio']

#Entrenamiento del modelo
model = RandomForestRegressor(random_state=42)
model.fit(X_train, y_train)
```

Una vez hemos entrenado el modelo, realizamos la predicción (con la función *predict*) con el conjunto de testeo y comprobamos su eficiencia con el error absoluto medio, obteniendo un resultado exitoso (<3000€):

```
#Predicción y evaluación
predictions = model.predict(X_test)
mae = mean_absolute_error(y_test, predictions)
print("MAE del conjunto de validación:", mae, "€")
```

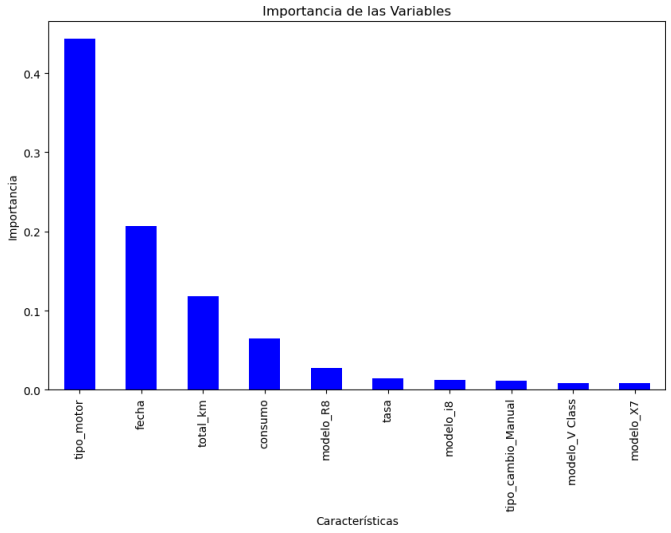
```
MAE del conjunto de validación: 1895.7567118113243 €
```

- Pregunta 4. Analizar las variables más significativas, ya sea mediante un test estadístico o analizando el modelo entrenado.

  Para obtener las variables más significativas del dataset en nuestro modelo, usamos la función *feature_importances_* del módulo *ensemble* de *scikit-learn*. Imprimimos las 10 más significativas:

```python
#P4: Variables más significativas
feature_importances = pd.Series(model.feature_importances_, index=X_train.columns).sort_values(ascending=False)
plt.figure(figsize=(10, 6))
feature_importances.head(10).plot(kind='bar', color='blue')
plt.title('Importancia de las Variables')
plt.xlabel('Características')
plt.ylabel('Importancia')
plt.show()
print("Variables más importantes:\n", feature_importances.head(10))
```

Podemos comprobar que con diferencia, el tipo de motor es la variable de mayor importancia, seguida de la fecha y el número de kilómetros.




```
Variables más importantes:
 tipo_motor            0.442982
fecha                  0.206486
total_km               0.118296
consumo                0.064836
modelo_R8              0.027290
tasa                   0.014631
modelo_i8              0.012197
tipo_cambio_Manual     0.011048
modelo_V Class         0.007962
modelo_X7              0.007936
```

- Pregunta 5. Un amigo quiere vender un Audi A7 de 2020 con 5000 km, cambio automático, combustible híbrido, consumo de 5.5 l/100km y motor 4.0. Su tasa de circulación es de 200€. ¿A cuánto debería venderlo?

  Finalmente, con el modelo entrenado y testeado, realizamos una predicción con un caso de tasación de un modelo específico. Lo definimos como diccionario:

```python
#P5: Predicción del precio de un Audi A7
new_car = pd.DataFrame({
    'total_km': [5000],
    'consumo': [5.5],
    'tipo_motor': [4.0],
    'tasa': [200],
    'fecha': [2020],
    'marca_Audi': [1],
    'modelo_A7': [1],
    'tipo_cambio_automático': [1],
    'tipo_combustible_híbrido': [1]
})
#Predicción del precio
predicted_price = model.predict(new_car)[0]
print(f"El precio recomendado para el Audi A7 es: {predicted_price:.2f} €")
```

```
El precio recomendado para el Audi A7 es: 63183.74 €
```

Ejercicio 2: predicción del resultado de una campaña de marketing telefónica

El caso de estudio del segundo ejercicio consiste en la campaña anual de marketing telefónico de un banco, cuyo objetivo es ofrecer depósitos a plazo fijo a sus clientes. Disponemos de datos recopilados los últimos tres años, a partir de los cuales se nos pide desarrollar un modelo predictivo para mejorar los resultados de contratación (tasa de conversión de la campaña) enfocándose en los clientes con mayor probabilidad de contratación del producto. Dado a que la variable objetivo es la contratación o no del producto, estamos ante un **problema de clasificación**.

Dichos datos se almacenan en un dataset cuyas columnas se listan y describen en la siguiente tabla:

Columna	**Descripción**
edad	Edad del cliente, en años
empleo	Tipo de empleo del cliente
estado	Estado civil del cliente: soltero, casado, divorciado, etc.
educación	Nivel de estudios del cliente
impago	Indica si el cliente tiene algún impago pendiente
hipoteca	Indica si el cliente tiene contratada una hipoteca
préstamo	Indica si el cliente tiene contratado un préstamo
tipo_contacto	Indica si se contactó con el cliente por teléfono fijo o móvil (cellular)
mes	Mes en el que se contactó con el cliente
dia_semana	Día de la semana en el que se contactó con el cliente
contactos_actual	Número de veces que se ha contactado con el cliente en la campaña actual
contactos_anterior	Número de veces que se ha contactado con el cliente en la campaña anterior
resultado_anterior	Resultado de la campaña anterior: success, failure o nonexistent
tasa_var_empleo_3m	Indicador macroeconómico: variación en la tasa de empleo trimestral
euribor_3m	Indicador macroeconómico: valor del Euribor trimestral
ipc_1m	Indicador macroeconómico: valor del IPC (Índice de Precios al Consumidor) mensual
target	Resultado de la campaña: ‘yes’ si contrató el producto, ‘no’ en caso contrario

De nuevo, en nuestra situación de partida disponemos del dataset previamente separado entre conjunto de entrenamiento y conjunto de validación, respectivamente: El fichero dataset_marketing_train.csv con 32652 registros y el fichero dataset_marketing_test.csv con 8536.

Objetivo. Modelo propuesto

De nuevo, se propone el uso del algoritmo Random Forest, esta vez enfocado a un problema de clasificación. En este caso, cada árbol de decisión predice una clase para una entrada dada, siendo el resultado final para dicha entrada la clase obtenida por la mayoría de los árboles.

Tras la carga de datos (que omito, ya que es análoga a la del ejercicio 1) volvemos a hacer un procesamiento del dataset para poder aplicar el modelo. Este procesamiento comienza con el mapeo de la variable objetivo a formato binario, sigue con la separación de características y target y finaliza con la codificación One Hot que se describe en la Pregunta 3.

```
#Transformación de la columna target
train_marketing['target'] = train_marketing['target'].map({'yes': 1, 'no': 0})
test_marketing['target'] = test_marketing['target'].map({'yes': 1, 'no': 0})

#Separación de características (X) y variable objetivo (y)
y_train = train_marketing['target']
y_test = test_marketing['target']
X_train = train_marketing.drop('target', axis=1)
X_test = test_marketing.drop('target', axis=1)
```

- Pregunta 3. Transformar, al menos, las variables estado y resultado_anterior utilizando one-hot encoding.

  De la misma forma que en el ejercicio anterior, necesitamos codificar las variables categóricas a variables numéricas para que puedan ser entrada del modelo de AA. Para ello, volvemos a detectar las variables categóricas (tipo = objeto) y las transformamos con el codificador One Hot con las funciones *fit_transform* y *transform.*

  He tenido que añadir una transformación intermedia que pasa las variables categóricas a strings ya que me saltaba un error (lo solucioné con ChatGPT).

```
#Identificar columnas categóricas y convertirlas a string
categorical_features = X_train.select_dtypes(include=['object']).columns
X_train[categorical_features] = X_train[categorical_features].astype(str)
X_test[categorical_features] = X_test[categorical_features].astype(str)

#P3: Transformación de las variables estado y resultado_anterior
encoder = OneHotEncoder(drop='first', sparse_output=False)

#Codificación de todas las variables categóricas seleccionadas
encoded_train = pd.DataFrame(encoder.fit_transform(X_train[categorical_features]))
encoded_test = pd.DataFrame(encoder.transform(X_test[categorical_features]))
```

Una vez hemos dispuesto el formato de las variables, entrenamos el modelo con *fit()*:

```
#Entrenamiento del modelo
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)
```

Finalmente, evaluamos la precisión del modelo con el conjunto de testeo, obteniendo un resultado satisfactorio con el algoritmo seleccionado (precisión > 60%):

```
#Evaluación de la precisión del modelo
predictions = model.predict(X_test)
precision = precision_score(y_test, predictions)
print("Precisión en conjunto de validación:", precision)
```

```
Precisión en conjunto de validación: 0.7459807073954984
```

- Pregunta 1. Calcular el ratio de conversión (porcentaje de clientes que contratan) de la campaña en el conjunto de entrenamiento.

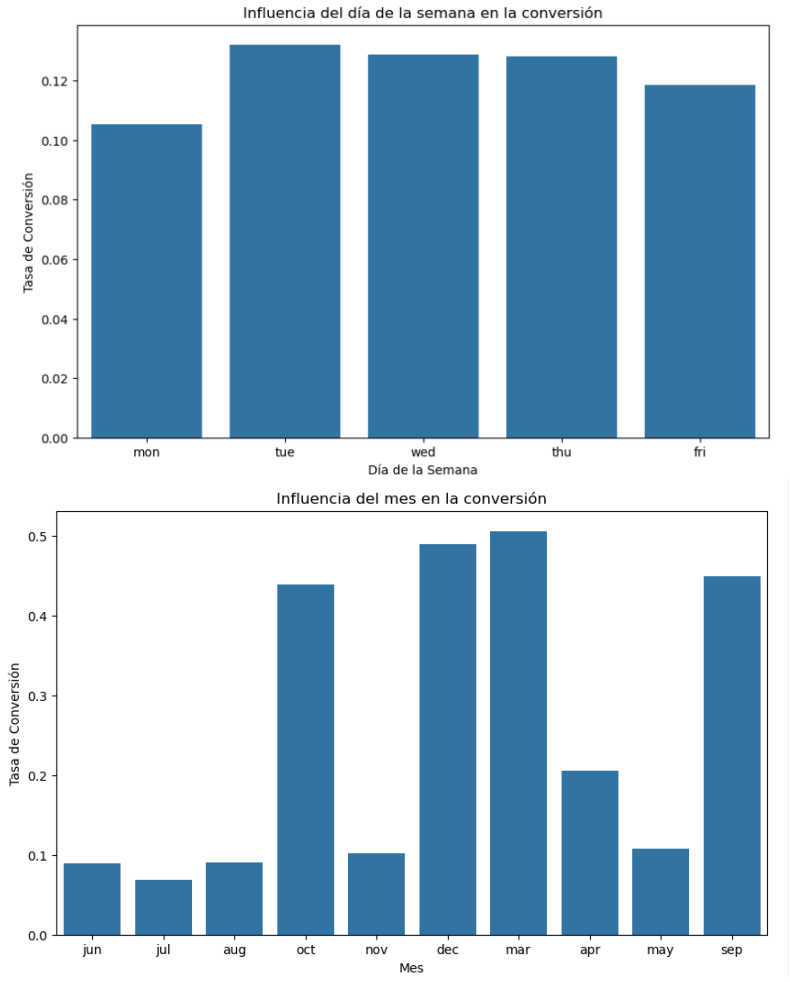
  Con el modelo entrenado, calculamos la tasa de conversión como la media de la variable objetivo, ya que está dispuesta como una variable binaria:

```
#P1: Ratio de conversión
conversion_rate = y_train.mean()
print("Ratio de conversión:", conversion_rate * 100, "%")
```

```
Ratio de conversión: 12.241210339336028 %
```

- Pregunta 2. ¿Cómo influye el día de la semana de contacto en el resultado de la campaña? ¿Y el mes? Utilice el conjunto de entrenamiento.

  Omitiendo el código por analogía al primer ejercicio, creamos un diagrama de barras que distribuya la tasa de conversión por día de la semana y por més. Entre los resultados destaca que el lunes es el día de menor contratación y que Marzo, Septiembre, Octubre y Diciembre son los meses más exitosos.



- Pregunta 4. Representar gráficamente la curva ROC del modelo y calcular su AUC en el conjunto de validación.

  Para calcular la curva ROC empleamos la función *roc_curve*, que toma como entrada el target del conjunto de testeo y las probabilidades de cada clase estimadas por el modelo con la función *predict_proba*. Esta función devuelve la tasa de falsos positivos (fpr), la tasa de verdaderos positivos y los tresholds de puntos de corte.

  Estas dos tasas son la entrada de la función *auc*, que devuelve la AUC asociada a dicha curva. *NOTA: me parece curioso el orden lógico de estas funciones. A partir de la ROC se obtienen las tasas y umbrales de puntos de corte (y no al revés) y es a partir de estos puntos de corte que se obtiene el área bajo la curva. Entiendo que es la lógica de programación.*

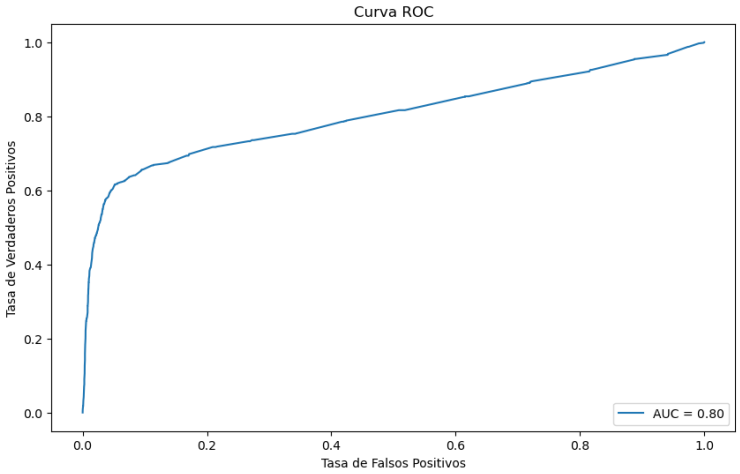
```python
#P4: Curva ROC y AUC
probs = model.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, probs)

auc = auc(fpr, tpr)
print("AUC:", auc)

plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, label=f'AUC = {auc:.2f}')
plt.title('Curva ROC')
plt.xlabel('Tasa de Falsos Positivos')
plt.ylabel('Tasa de Verdaderos Positivos')
plt.legend(loc='lower right')
plt.show()
```

Los resultados obtenidos son los siguientes:

```
AUC: 0.8020722931258459
```



- Pregunta 5. El banco quiere optimizar los costes de la campaña utilizando el modelo que acabas de entrenar.

  Para ello, te indica que cada llamada que se realiza tiene un coste de 5€ para ellos, y que el beneficio obtenido al contratar el producto es de 50€. Por tanto, el balance neto (beneficios-gastos) de la campaña es:

  Balance = 50€ x Clientes que contratan- 5€ x Clientes contactados

  Para maximizar el balance se necesita que el mayor número posible de llamadas terminen en contratación. Es decir, que la precisión sea lo más alta posible. Por ejemplo: si se llama a 10 clientes y ninguno contrata el balance es-50€, y si contratan 5 el balance sería 200€.

  El banco va a utilizar tu modelo para decidir a qué clientes llama y a cuáles no: llamará sólo a aquellos que tengan una probabilidad de contratación superior a un cierto umbral. Utilizando el conjunto de validación, deberás hallar cuál es el umbral que maximiza el balance de la campaña teniendo en cuenta los beneficios y los gastos.

  Nota: El conjunto de validación contiene 8,536 clientes contactados, de los cuales 643 contrataron. Usando la fórmula anterior, podemos ver que el balance de la campaña fue de -10,530€. ¡Fue un desastre! Lo que se pide en este ejercicio es hacer un backtesting, es decir, simular qué hubiera pasado si se hubiese utilizado tu modelo. Es decir, para resolver esta pregunta debes calcular tanto el número de clientes contactados, en función de las probabilidades de tu modelo, como los que contratan.

Se nos pide comprobar qué resultados hubiéramos obtenido si hubiéramos usado nuestro modelo. A efectos prácticos y teniendo en cuenta la fórmula para calcular el balance, debemos comprobar que las predicciones de nuestro modelo hacen que se llame a clientes con una potencialmente alta probabilidad de contratación del servicio.

Los umbrales de probabilidad de corte (contratación, en este caso) nos los devolvió la función de obtención de la ROC. Se plantea un bucle que recorra estos umbrales y 'contacte' a los clientes con probabilidad de contratación superior a dicho umbral. Se declararán casos exitosos los que, de esos clientes contactados, contrataron el servicio. Finalmente se calcula el balance económico con la fórmula y se almacena en una variable junto con el umbral evaluado. Si en una nueva iteración se obtiene un mejor balance, se actualizan ambas variables (*NOTA usé ChatGPT ya que yo inicialicé la variable best_balance a 0, pero me dio error ya que, evidentemente, el balance puede dar negativo. No conocía el uso de -float('inf'), pero me ha parecido muy curioso y lógico)*:

```
#P5: Optimización de balance
best_balance = -float('inf')
best_threshold = 0
for threshold in thresholds:
    predicted = (probs >= threshold).astype(int) #evaluamos qué clientes son contactados para cada umbral
    contacts = predicted.sum()
    successes = ((predicted == 1) & (y_test == 1)).sum()
    balance = 50 * successes - 5 * contacts
    if balance > best_balance:
        best_balance = balance
        best_threshold = threshold

print("Mejor umbral:", best_threshold)
print("Balance óptimo de la campaña:", best_balance, "€")
```

Se obtiene un resultado muy positivo, aumentando el beneficio de la campaña en más de 25000 €. El modelo estima que el mejor umbral es el de clientes con una probabilidad de contratación del 23.5%.

```
Mejor umbral: 0.235
Balance óptimo de la campaña: 15780 €
```

Ajuste de hiperparámetros

He probado a ajustar hiperparámetros en ambos modelos para ver si mejora su eficiencia. Para ello he usado GridSearchCV, creando un grid con los hiperparámetros más destacables de Random Forest:

Hiperparámetro	**Descripción**
n_estimators	Número de árboles. Más árboles pueden mejorar el rendimiento, pero aumentan el tiempo de cómputo.
max_depth	Profundidad máxima de los árboles. Limitarla ayuda a prevenir sobreajuste, dejarla como None permite que crezcan completamente.
min_samples_split	Número mínimo de muestras necesarias para dividir un nodo.
min_samples_leaf	Número mínimo de muestras que debe contener una hoja. Valores altos mejoran la generalización.
max_features	Número máximo de características consideradas para dividir en cada nodo.

Para ambos modelos, definimos el siguiente grid de hiperparámetros con posibles valores:

```
#Ealuación de la precisión ajustando hiperparámetros
param_grid = {
    'n_estimators': [100, 200],      #Menos valores para el número de árboles
    'max_depth': [None, 20],         #Menos opciones de profundidad
    'min_samples_split': [2, 10],    #División mínima
    'min_samples_leaf': [1, 2],      #Menos valores para hojas mínimas
    'max_features': ['sqrt']         #Fijar un valor común para max_features
}
```

Seguimos configurando GridSearchCV, especificando el modelo, el grid de hiperpatrámetros, la métrica de evaluación (en este caso, al ser el modelo regresor, es el MAE. Para el clasificador usamos la precisión) y la valoración cruzada (número de particiones). El resto de parámetros son por defecto (texto de progreso y uso de núcleos):

```
#Configuramos GridSearchCV
grid_search = GridSearchCV(
    estimator=model,
    param_grid=param_grid,
    scoring='neg_mean_absolute_error',
    cv=5,
    verbose=1,
    n_jobs=-1
)
```

Finalmente, entrenamos el nuevo modelo con GridSearchCV, obtenemos el mejor modelo con *best_estimator_*, realizamos la predicción y evaluamos el modelo:

```
#Entrenar el nuevo modelo
grid_search.fit(X_train, y_train)

#Evaluación del mejor modelo
best_model = grid_search.best_estimator_
predictions = best_model.predict(X_test)
best_mae = mean_absolute_error(y_test, predictions)

print("Error Absoluto Medio (MAE) del mejor modelo:", best_mae)
```

Para el ejercicio 1 no obtenemos mejores resultados:

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
Error Absoluto Medio (MAE) del mejor modelo: 1903.0840985879533
```

Para el ejercicio 2, tampoco:

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
Precisión del mejor modelo en el conjunto de validación: 0.738562091503268
```