Capstone IV

Machine Learning aplicado a márketing

Luis de la Ossa
Máster en Ciencia de Datos e Ingeniería de Datos en la Nube
Universidad de Castilla-La Mancha

Marta Bellón Castro
Curso 2022-2023

Introducción

En este proyecto se plantean tres supuestos que guardan relación con el área del márketing y la gestión de la relación con el cliente (*Customer Relationship Management*). El primero consiste en el desarrollo de un *pipeline* completo para la predicción del abandono de clientes (*Churn prevention*). En el segundo se desarrollará un modelo que permite estimar el valor potencial de un cliente (*Customer Lifetime Value*). Por último, se recurrirá al aprendizaje no supervisado para segmentar un conjunto de clientes en grupos.

A diferencia de los proyectos anteriores, que consistían exclusivamente en la solución de ejercicios prácticos, en éste se plantearan también algunas preguntas relacionadas con la interpretación y análisis del proceso, y que ***habrán de ser respondidas en la propia libreta***.

> ℹ **Nota**: Aunque el proyecto es fácil, es algo largo ya que, además de bastantes ejercicios, se aportan abundantes explicaciones y comentarios sobre el propio proceso. Especialmente en el planteamiento del primer ejercicio. Recomendamos leerlas atentamente.

Índice

- 1. Prevención del abandono (*churn prevention*)
  - 1.1. Exploración de los datos. Preprocesamiento

    - 1.2. Construcción de un modelo
    - 1.3. Validación sobre nuevos datos
    - 1.4. Comparación con otros modelos
- 2. Predicción del valor potencial de un cliente (*Customer Lifetime Value*)
    - 2.1. Construcción de un árbol de regresión
- 3. Segmentación de clientes

In [1]:
```
# Configuración de la visualización
from IPython.display import display, HTML
display(HTML("<style>.container { width:95% !important; }</style>"))

%matplotlib inline
#%config InlineBackend.figure_format = 'retina'


import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns;
sns.set()
```

1. Prevención del abandono (*churn prevention*)

La prevención del abandono, o *Churn prevention*, es una tarea de *márketing* que consiste en detectar aquellos clientes que podrían abandonar un determinado servicio o mercado, y llevar a cabo campañas específicas de retención con el fin de evitar la pérdida.

Esta parte del proyecto consiste en la elaboración de un modelo basado en aprendizaje supervisado (clasificación) para la detección de clientes en esta situación. Se proporciona un conjunto de datos denominado `Telco-Customer-Churn.csv` , relativo al abandono en una operadora de telecomunicaciones, y obtenido en el sitio de IBM analytics (https://www.ibm.com/analytics). Alguna información adicional, así como *kernels* que tratan estos mismos datos y pueden resultar útiles (*se recomienda que los miréis*) están disponibles en Kaggle (https://www.kaggle.com/blastchar/telco-customer-churn).

In [2]:
```
# Carga los datos
df_churn = pd.read_csv('data/churn/Telco-Customer-Churn.csv', index_col=0);

print("Tamaño del conjunto de datos:  %d" % df_churn.shape[0])
print("Número de variables: %d" % df_churn.shape[1])

if df_churn.index.is_unique:
    print('El índice es único.')
else:
    print('Los índices están duplicados.')

# Visualiza las primeras instancias
df_churn.head()
```

```
Tamaño del conjunto de datos:  7043
Número de variables: 20
El índice es único.
```

Out[2]:

customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	OnlineBackup	DeviceProtection	TechSupport	StreamingTV	Stream
7590-VHVEG	Female	0	Yes	No	1	No	No phone service	DSL	No	Yes	No	No	No	
5575-GNVDE	Male	0	No	No	34	Yes	No	DSL	Yes	No	Yes	No	No	
3668-QPYBK	Male	0	No	No	2	Yes	No	DSL	Yes	Yes	No	No	No	
7795-CFOCW	Male	0	No	No	45	No	No phone service	DSL	Yes	No	Yes	Yes	No	
9237-HQITU	Female	0	No	No	2	Yes	No	Fiber optic	No	No	No	No	No	

Como línea de trabajo, se asumirá que el modelo se desarrolla a partir de un conjunto manejable de datos etiquetados (histórico), pero será utilizado sobre un conjunto de datos actual, no etiquetado, y mucho mayor. Debido a esto, se dividirá el conjunto de datos almacenado en `df_churn` en dos: `df_churn` con el 85% de las entradas, y que será utilizado en el proceso de desarrollo (y validación) del modelo; y `df_churn_new`, con el 15% restante, que permitirá tanto comprobar la correcta definición del flujo de trabajo sobre datos nuevos, como proporcionar medidas de rendimiento del modelo.

El conjunto de datos `df_churn_new` **no será utilizado en ninguna fase del proceso de aprendizaje**, ni siquiera para la exploración, preprocesamiento, o la validación del modelo.

In [3]:
```
# Tamaño del conjunto de entrenamiento
training_size = int(len(df_churn)*0.85)

# Desordena los datos (esto es muy importante).
df_churn = df_churn.sample(frac=1, random_state=0)

# Copia los datos de test.
df_churn_new = df_churn.iloc[training_size:].copy()

# Copia los datos para el entrenamiento del modelo.
df_churn = df_churn.iloc[:training_size].copy()

print("Tamaño del conjunto de datos disponibles: ", len(df_churn))
print("Tamaño del conjunto de nuevos datos: ", len(df_churn_new))
```

```
Tamaño del conjunto de datos disponibles:  5986
Tamaño del conjunto de nuevos datos:  1057
```

En ciencia de datos es frecuente que el punto de partida del trabajo lo constituyan datos desconocidos. El primer paso, *también cuando se trabaja con aprendizaje automático*, consiste en **explorar estos datos**. Además de la familiarización con el problema (que no siempre es posible), como fruto de la exploración surgen decisiones que afectan *principalmente* a la preparación y al preprocesamiento.

En general, el proceso de exploración no es sistemático, sino que se lleva a cabo en base a la información que va arrojando el propio proceso, y es guiado en parte por la intuición y experiencia del analista. En cada paso, surge la necesidad de hacer transformaciones a los datos originales de cara a su uso en la construción del modelo predictivo. Estos cambios pueden ir haciéndose progresivamente. Sin embargo, ***todo el proceso ha de ser registrado de algún modo, ya que ha de ser reproducido para el tratamiento de nuevos datos*** que, en un contexto real, son adquiridos con el formato del conjunto de datos original.

ℹ A lo largo del módulo se ha estudiado como hacer *pipelines* o secuencias de funciones en `scikit-learn`. Aunque *esta funcionalidad es muy útil*, incorporar todo el proceso de los datos dentro de un *pipeline* de `scikit-learn` puede ser tedioso, e incluso puede haber limitaciones que lo impidan o hagan recomendable un modo alternativo de trabajo. En realidad, *es posible apartar (y anticipar) algunas tareas de preparación de datos* del proceso definido en el *pipeline* de `scikit-learn`. Así, operaciones como la eliminación o transformación de columnas, cuya aplicación en nuevos datos es *totalmente independiente del entrenamiento*, pueden definirse y llevarse a cabo en un *pipeline* de `pandas`, anterior a la construcción (y uso) del modelo de clasificación.

> ❗ Ejemplos de operaciones que dependen del entrenamiento son la estándarización o la sustitución de valores perdidos por la media/mediana. Éstas sí deben formar parte del *pipeline* de `scikit-learn` ya que implican instanciar un transformador.

En este proyecto se propone un modo de trabajar en el que sucesivamente, *durante la exploración*, se aplicarán funciones de preprocesamiento a un *DataFrame*. Estas funciones serán almacenadas en una lista que, finalmente, podrá ser utilizada para creación del *pipeline* de `pandas` que implementa la secuencia de acciones de preprocesamiento sobre nuevos datos. Una vez disponible una versión adecuada de los datos, el trabajo con `scikit-learn` se hará según el procedimiento habitual.

> ℹ **Nota**: El modo de trabajo que se propone **no es el único, pero es una alternativa** que puede ser conveniente en algunos supuestos, y distinta a las vistas en clase (implementación de transformadores). Por otra parte, esta tarea se centra en la construcción de modelos, por lo que **la parte de exploración se ha reducido al mínimo imprescindible**. No obstante, podéis incluir pasos adicionales en este sentido si lo veis conveniente.

Considerando el planteamiento descrito, es necesario registrar las transformaciones que se irán haciendo sobre el *DataFrame* y el tratamiento que se hará finalmente a cada variable dentro del *pipeline* de `scikit-learn`. Para ello, se utilizarán tres listas:

- `churn_data_prep_pipeline`, que guardará la secuencia de funciones de transformación (referencias) aplicadas al *DataFrame*.
- `cat_features`, que guardará los nombres de las columnas que contienen valores discretos, y que han de ser tratadas como categóricas en el *pipeline* de `scikit-learn`.
- `num_features`, que guardará los nombres de las columnas que han de ser tratadas como numéricas en el *pipeline* de `scikit-learn`.

In [4]:
```
# Lista de funciones aplicadas en la preparación de los datos
churn_data_prep_pipeline = []

# Características que serán consideradas categóricas y numéricas en el pipeline
cat_features = []
num_features = []
```

1.1 Exploración de los datos. Preprocesamiento.

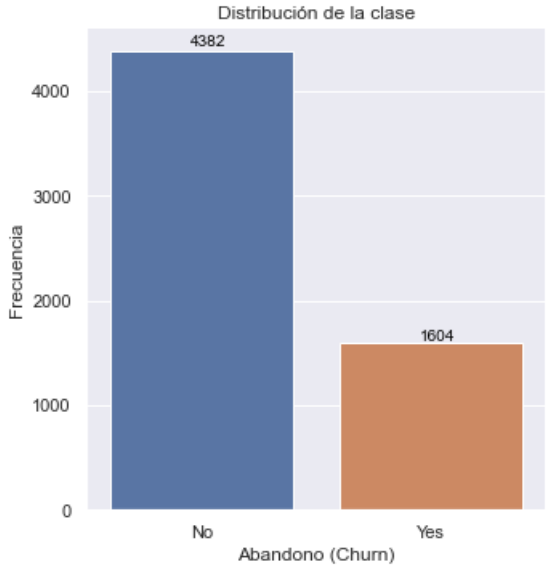
El objetivo principal de este trabajo es predecir la baja de un determinado usuario mediante un modelo de clasificación. Por eso, uno de los pasos más importantes en la exploración consiste en el análisis e interpretación de la distribución de las clases. Este factor puede resultar de interés en el entrenamiento, pero también de cara a la evaluación.

Ejercicio 1

Dibujar un diagrama de barras (`sns.catplot()` o `sns.countplot()`) con la distribución de las clases (columna `Churn`) sobre las muestras. Obtener los posibles valores de las mismas. Comentar qué se aprecia en la gráfica, en qué modo puede afectar esto al rendimiento del clasificador, y si esto influye en el modo en que se ha de llevar a cabo la evaluación.

In [5]:
```
# Creo la gráfica de barras
g = sns.catplot(x="Churn", kind="count", data=df_churn);

# Inserto los valores de cada barra
ax = g.facet_axis(0,0)
for p in ax.patches:
    ax.text(p.get_x() +0.35,
            p.get_height() * 1.012,
            '{0:.0f}'.format(p.get_height()),
            color='black',
            rotation='horizontal',
            size=10)

# Añado etiquetas y títulos al gráfico
plt.xlabel("Abandono (Churn)")
plt.ylabel("Frecuencia")
plt.title("Distribución de la clase")

# Imprimo la información
print("Clases: ", df_churn['Churn'].unique())
print(f"El porcentaje de muestras positivas es {(sum(df_churn['Churn']=='Yes') / df_churn.shape[0]):.2f}")
```

```
Clases:  ['No' 'Yes']
El porcentaje de muestras positivas es 0.27
```



Respuesta

En el gráfico podemos ver que el data set no está balanceado, es decir, hay más datos de una categoría que de la otra: 27% de los datos corresponden a "abandono" y un 73% a "no abandono". Es importante tener esto en cuenta al evaluar nuestro algoritmo de predicción, ya que necesitamos que pueda predecir ambas categorías correctamente.

☑

Puede apreciarse que las clases están codificadas como *Strings*. Para trabajar con `scikit-learn` se necesita convertirlas a formato numérico.

> ❗ Este paso no será incluido en el *pipeline*. El motivo es que (tal y como se comentó anteriormente) una vez diseñado, se debería implantar para el procesamiento de datos nuevos que ***no*** contienen información relativa a la clase.

A continuación, se convierte la columna correspondiente a la clase a formato numérico.

In [6]:
```
df_churn['Churn'] = (df_churn['Churn']=='Yes').astype(int)
df_churn.head()
```

Out[6]:

customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	OnlineBackup	DeviceProtection	TechSupport	StreamingTV	Stream
6296-DDOOR	Female	0	No	No	19	Yes	No	DSL	No	Yes	No	No	Yes	
3680-CTHUH	Male	0	No	No	60	Yes	Yes	Fiber optic	Yes	Yes	Yes	Yes	Yes	
4931-TRZWN	Female	0	No	No	13	Yes	No	DSL	Yes	No	No	No	Yes	
8559-WNQZS	Male	0	No	No	1	Yes	No	No	No internet service	No internet service	No internet service	No internet service	No internet service	
3537-HPKQT	Female	0	Yes	No	55	Yes	Yes	Fiber optic	No	Yes	No	No	No	

Una vez explorada la clase, se procede con las características, para lo cual se comprobarán primero los tipos.

In [7]:
```
df_churn.dtypes
```

Out[7]:
```
gender              object
SeniorCitizen        int64
Partner             object
Dependents          object
tenure               int64
PhoneService        object
MultipleLines       object
InternetService     object
OnlineSecurity      object
OnlineBackup        object
DeviceProtection    object
TechSupport         object
StreamingTV         object
StreamingMovies     object
Contract            object
PaperlessBilling    object
PaymentMethod       object
MonthlyCharges     float64
TotalCharges        object
Churn                int32
dtype: object
```

Puede observarse que hay columnas de enteros, flotantes y también *Strings* (objetos). Se dividirán las columnas por tipo, y se procederá a examinar y tratar cada grupo por separado.

> ❗ En este caso de estudio se puede proceder así, ya que **el número de variables es manejable**. En casos con más variables, habría que hacer una exploración de carácter más superficial, y definir métodos de carácter general.

A continuación se almacenan los nombres de las columnas numéricas del *DataFrame* en una lista denominada `num_df_columns`, y el resto en otra denominada `cat_df_columns`.

> ❗ ***Observad*** que éstas listas son distintas de `num_features` y `cat_features`, que contienen los nombres de las características en función de ***cómo serán tratadas finalmente*** en el *pipeline* de `scikit-learn`.

In [8]:
```
dis_df_columns = df_churn.select_dtypes(exclude=np.number).columns
num_df_columns = df_churn.select_dtypes(include=np.number).columns

print('Discretas: ',dis_df_columns)
print('\nNuméricas: ',num_df_columns)
```

```
Discretas:  Index(['gender', 'Partner', 'Dependents', 'PhoneService', 'MultipleLines',
       'InternetService', 'OnlineSecurity', 'OnlineBackup', 'DeviceProtection',
       'TechSupport', 'StreamingTV', 'StreamingMovies', 'Contract',
       'PaperlessBilling', 'PaymentMethod', 'TotalCharges'],
      dtype='object')

Numéricas:  Index(['SeniorCitizen', 'tenure', 'MonthlyCharges', 'Churn'], dtype='object')
```

ℹ Se comenzará tratando las variables discretas, ya que a veces conviene convertirlas a numéricas, y luego se pueden incluir en la exploración y análisis de las variables numéricas.

Tratamiento de las columnas discretas

ℹ En relación a estas columnas, dos aspectos muy relevantes de cara a la construcción de un modelo con `scikit-learn` son: el número de valores que puede tomar cada una; y si existe una relación de orden entre estos valores. Estos factores determinan el tipo de transformación que se ha de hacer. Existen ***cuatro posibilidades***:

- Cuando la columna toma dos valores, se puede binarizar y convertir a numérica diréctamente.
- Si el tamaño del conjunto de valores es mayor que dos, y no existe una relación de orden entre ellos, se aplica *One Hot Encoding* (se aplicará posteriormente en el *pipeline* de transformaciones).
- Si existe una relación de orden, los valores se transforman a numéricos, sustituyendo cada valor por su orden.
- Si el conjunto de valores extremadamente grande se ha de explorar, ya que es muy posible que se trate de un identificador, o de un error.

A continuación, se obtiene el número de valores para cada una de las variables discretas y se almacena en una lista denominada `num_values_dis_df_col` . Cada elemento de la lista es una tupla con el nombre de la columna y el número de variables.

In [9]:
```
num_values_dis_df_col = list(map(lambda col: (col,len(df_churn[col].value_counts())), dis_df_columns))
num_values_dis_df_col
```

Out[9]:
```
[('gender', 2),
 ('Partner', 2),
 ('Dependents', 2),
 ('PhoneService', 2),
 ('MultipleLines', 3),
 ('InternetService', 3),
 ('OnlineSecurity', 3),
 ('OnlineBackup', 3),
 ('DeviceProtection', 3),
 ('TechSupport', 3),
 ('StreamingTV', 3),
 ('StreamingMovies', 3),
 ('Contract', 3),
 ('PaperlessBilling', 2),
 ('PaymentMethod', 4),
 ('TotalCharges', 5608)]
```

Variables binarias

Como se comentó anteriormente, las variables discretas que toman dos valores se pueden binarizar directamente, salvo en el caso de no proporcionar información útil, que se pueden eliminar.

A partir de la lista `num_values_dis_df_col` se obtienen los nombres de las columnas binarias y se almacenan en una lista denominada `dis_df_col_bin` .

In [10]:
```
dis_df_col_bin = list(map(lambda cv: cv[0],(filter(lambda cv: cv[1]==2,num_values_dis_df_col))))
dis_df_col_bin
```

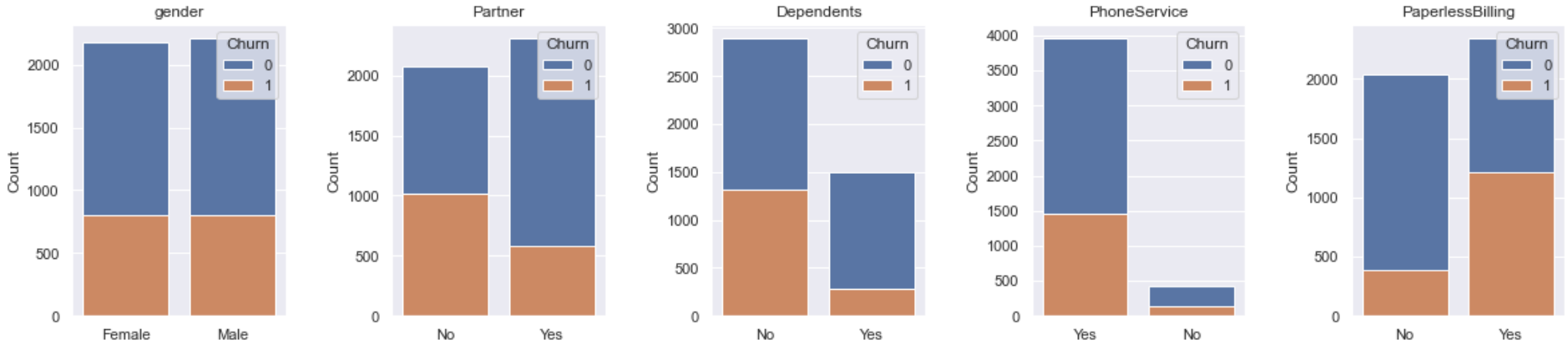
Out[10]:
```
['gender', 'Partner', 'Dependents', 'PhoneService', 'PaperlessBilling']
```

Ejercicio 2

Dibujar la distribución de cada una de estas variables utilizando una gráfica de tipo `sns.countplot()` para cada una de ellas. Es posible visualizar también la variable de clase (con `hue='Churn', dodge=False`).

En este ejercicio se crea una figura con cuatro gráficas, y en cada una de ellas hay que dibujar el `sns.countplot()` correspondiente a cada una de las columnas, utilizando el color de la gráfica para distinguir la variable `Churn` .

In [11]:
```
# Creo una figura con cuatro gráficas
fig, axs = plt.subplots(1, 5, figsize=(20, 4))

# En la variable 'columns' introduzco las columnas a visualizar
columns = dis_df_col_bin

# Itero sobre cada una de las columnas para visualizarlo
for i, col in enumerate(columns):
    sns.countplot(x=col, data=df_churn, hue='Churn', dodge=False, ax=axs[i])
    axs[i].set_title(col)
    axs[i].set_xlabel(None)
    axs[i].set_ylabel('Count')

# Añado separación entre gráficas
plt.subplots_adjust(wspace=0.5)
plt.show()
```



☑

Independientemente de que lo relevantes que puedan ser a la hora de clasificar, parece que todas las variables y sus valores tienen sentido, por lo que se van a preservar.

La siguiente función, `churn_binarize_dis` , recibe un *DataFrame* y transforma las columnas de `dis_df_col_bin` a un entero binario.

In [12]:
```
# Para gender --> Male=0, Female=1
# Pare el resto: --> No=0, Yes=1
def churn_binarize_dis(df):
    df['gender'] = (df['gender']=='Female').apply(int)
    df['Partner'] = (df['Partner']=='Yes').apply(int)
    df['Dependents'] = (df['Dependents']=='Yes').apply(int)
    df['PhoneService'] = (df['PhoneService']=='Yes').apply(int)
    df['PaperlessBilling'] = (df['PaperlessBilling']=='Yes').apply(int)
    return df # No hace falta

# Comprueba si la función está bien sobre una copia de los datos
display(churn_binarize_dis(df_churn.head(5).copy()))
```

customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	OnlineBackup	DeviceProtection	TechSupport	StreamingTV	Stream
6296-DDOOR	1	0	0	0	19	1	No	DSL	No	Yes	No	No	Yes	
3680-CTHUH	0	0	0	0	60	1	Yes	Fiber optic	Yes	Yes	Yes	Yes	Yes	
4931-TRZWN	1	0	0	0	13	1	No	DSL	Yes	No	No	No	Yes	
8559-WNQZS	0	0	0	0	1	1	No	No	No internet service	No internet service	No internet service	No internet service	No internet service	
3537-HPKQT	1	0	1	0	55	1	Yes	Fiber optic	No	Yes	No	No	No	

A continuacion se aplica la función `churn_binarize_dis` a los datos, y se añade a la lista de acciones de preprocesamiento, `churn_data_prep_pipeline` .

In [13]:
```
# Lo aplica
churn_binarize_dis(df_churn)

# Lo añade.
churn_data_prep_pipeline.append(churn_binarize_dis)
```

Las variables originalmente discretas que han sido transformadas a binarias, se han de tratar como numéricas, por lo que se añaden a la lista `num_features` .

In [14]:
```
# Añade las variables
num_features.extend(dis_df_col_bin)
# Muestra las variables numéricas
print('Variables numéricas: ', num_features)
```

```
Variables numéricas:  ['gender', 'Partner', 'Dependents', 'PhoneService', 'PaperlessBilling']
```

Variables categóricas

Las variables discretas que toman más de dos valores se pueden tratar como categóricas con *One Hot Encoding* (posteriormente) o, si son ordinales, se pueden transformar a numéricas. En primer lugar, hay que explorar el tipo y distribución de valores de cada una de ellas, y ver si se pueden considerar como ordinales.

A partir de `num_values_dis_df_col`, se pueden obtener los nombres de las columnas discretas que tienen más de dos valores y menos de cinco. Se almacenarán en una lista denominada `dis_df_columns_cat`.

In [15]:
```
dis_df_columns_cat = list(map(lambda cv: cv[0],(filter(lambda cv: cv[1]>2 and cv[1]<5, num_values_dis_df_col))))
dis_df_columns_cat
```

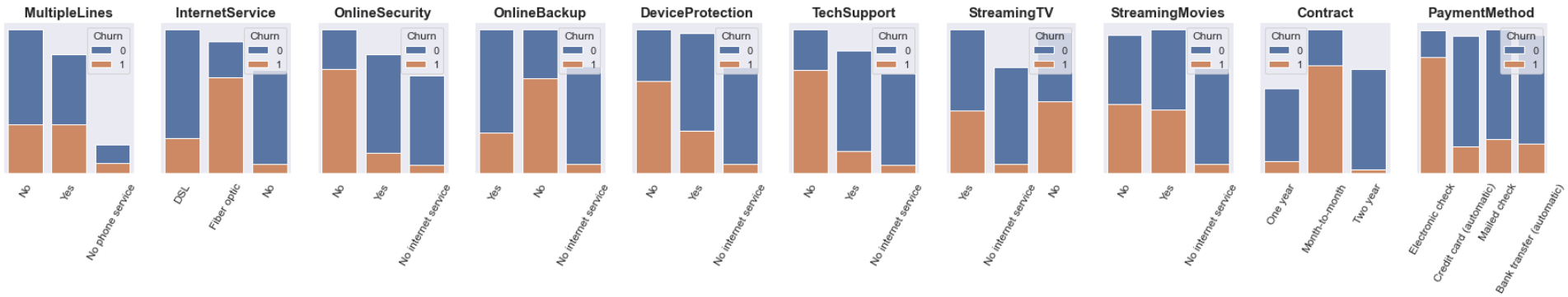
Out[15]:
```
['MultipleLines',
 'InternetService',
 'OnlineSecurity',
 'OnlineBackup',
 'DeviceProtection',
 'TechSupport',
 'StreamingTV',
 'StreamingMovies',
 'Contract',
 'PaymentMethod']
```

Ejercicio 3

Dibujar la distribución de cada una de estas variables utilizando una gráfica de tipo `sns.countplot()`. Para cada una de ellas para comprobar que todos los valores son correctos.

Este ejercicio es parecido al anterior. Se trata crear una figura con diez gráficas, una por columna, y dibujar una gráfica en cada una de ellas.

In [16]:
```
# Creo una figura con 10 gráficas
fig, axs = plt.subplots(1, 10, figsize=(30, 3))
plt.subplots_adjust(hspace=0.7, wspace=0.2)

# Itero sobre cada gráfica
for col, ax in enumerate(axs.flatten()):

    sns.countplot(x = dis_df_columns_cat[col], data=df_churn, hue="Churn", dodge=False, ax=ax)

    # Cambio parámetros de visualización
    ax.set_title(dis_df_columns_cat[col], fontsize=15, fontweight='bold')
    ax.set_xticklabels(ax.get_xticklabels(), rotation=60, fontsize=12)
    ax.set_yticks([])
    ax.set_xlabel(None)
    ax.set_ylabel(None)
```

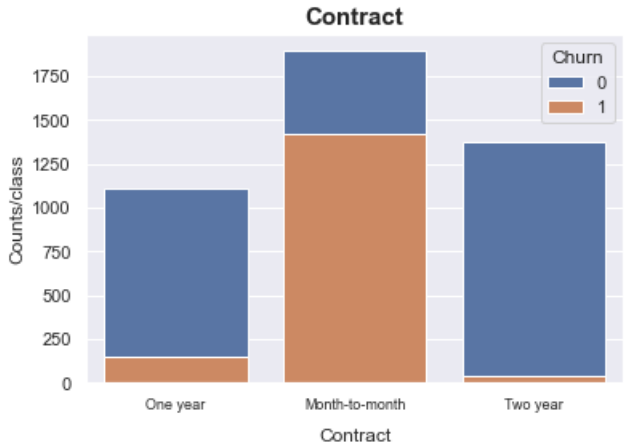


☑

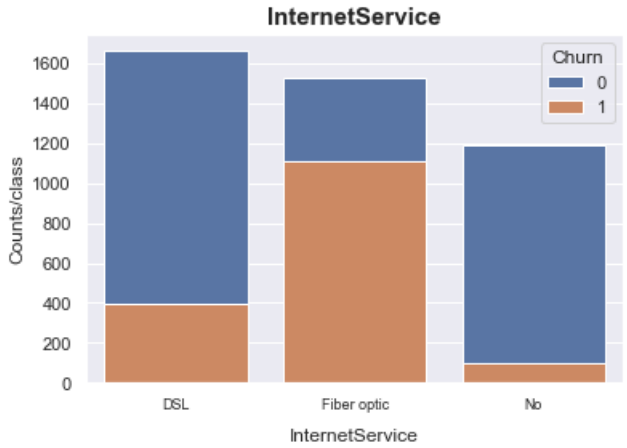
Ejercicio 4

Explorar estas variables (de una en una) mediante `sns.countplot()`, utilizando hue la relación de las variables con `Churn` (con `hue='Churn', dodge=False`). ¿Qué variable o variables parecen más relevantes? Dibujar solo una gráfica cada vez, y finalmente dejar solamente la correspondiente a la variable más relevante (si la hubiera) y comentar por qué lo es.

In [17]:
```
# Variable 'Contract'=8
col_name = dis_df_columns_cat[8]

# Creo la figura
fig, ax = plt.subplots(1,1)
sns.countplot(x = col_name, data=df_churn, hue="Churn", dodge=False, ax=ax)
ax.set_title(col_name, fontsize=15, fontweight='bold')

# Modifico la visualización
plt.xticks(rotation=0, fontsize=9)
plt.xlabel(col_name, labelpad=10)
plt.ylabel('Counts/class')
plt.show()
```



In [18]:
```
# Variable 'InternetService'=1
col_name = dis_df_columns_cat[1]

# Creo la figura
fig, ax = plt.subplots(1,1)
sns.countplot(x = col_name, data=df_churn, hue="Churn", dodge=False, ax=ax)
ax.set_title(col_name, fontsize=15, fontweight='bold')

# Modifico la visualización
plt.xticks(rotation=0, fontsize=9)
plt.xlabel(col_name, labelpad=10)
plt.ylabel('Counts/class')
plt.show()
```



Respuesta:

Tras analizar los gráficos, se puede concluir que la variable 'Contract' es la más relevante, ya que los contratos de mayor duración presentan una tasa de abandono menor. Esto puede ocurrir por:
1. Los requisitos de contratación.
2. La confianza en la compañía.
3. La satisfacción con el servicio que recibido.

También se puede observar en la variable 'InternetService' que los usuarios que no tienen acceso a internet abondonan menos la compañía. Esto podría deberse a:
1. Vivir en areas que no poseen acceso a internet.
2. Lealtad a la compañía y resistencia a cambiar a otra.
3. Falta de información sobre otras opciones disponibles.
4. Dificultades con la tecnología (personas mayores), siendo reacios a aprender a utilizar nuevos dispositivos y servicios ofrecidos por otras compañías.

☑

En principio, parece que todas las columnas se podrían tratar como categóricas, aunque la columna `Contract` se *podría* considerar ordinal. Por tanto, se añadirán todas las variables, menos `Contract`, a la lista `cat_features`.

In [19]:
```
cat_features = dis_df_columns_cat[:] # Copia la lista
cat_features.remove('Contract')      # Borra Contract
cat_features
```

Out[19]:
```
['MultipleLines',
 'InternetService',
 'OnlineSecurity',
 'OnlineBackup',
 'DeviceProtection',
 'TechSupport',
 'StreamingTV',
 'StreamingMovies',
 'PaymentMethod']
```

La función `churn_transform_contract` toma como entrada un *DataFrame* y convierte la columna `Contract` en numérica asignando los valores a partir del diccionario `con_to_ordinal`.

In [20]:
```
con_to_ordinal = {'Month-to-month':1, 'One year':2, 'Two year':3}

def churn_transform_contract(df):
    df['Contract'] = df['Contract'].map(con_to_ordinal)

    return df # No hace falta

# Comprueba si la función está bien
display(churn_transform_contract(df_churn.head(5).copy()))
```

customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	OnlineBackup	DeviceProtection	TechSupport	StreamingTV	Stream
6296-DDOOR	1	0	0	0	19	1	No	DSL	No	Yes	No	No	Yes	
3680-CTHUH	0	0	0	0	60	1	Yes	Fiber optic	Yes	Yes	Yes	Yes	Yes	
4931-TRZWN	1	0	0	0	13	1	No	DSL	Yes	No	No	No	Yes	
8559-WNQZS	0	0	0	0	1	1	No	No	No internet service	No internet service	No internet service	No internet service	No internet service	
3537-HPKQT	1	0	1	0	55	1	Yes	Fiber optic	No	Yes	No	No	No	

Ejercicio 5

Para esta variable, se ha hecho una asignación de valores `{'Month-to-month':1, 'One year':2, 'Two year':3}` . Además de que es intuitiva, parece consistente con la información que

Respuesta:

La clasificación por duración de contratos es coherente con la gráfica, mostrando que la tasa de abandono de 'Month-to-month' es mayor que la de 'One year' y que a su vez es mayor que 'Two year', es decir, a medida que la duración del contrato aumenta, la probabilidad de cancelación disminuye.
Desde mi punto de vista, transformar la duración a meses sería mas inuitivo, aunque es cierto que en este caso no existe información sobre valores intermedios.

☑

En este punto, se debe añadir la función `churn_transform_contract` a `churn_data_prep_pipeline` , aplicarla al `DataFrame` `df_churn` .

In [21]:
```
# Lo aplica
churn_transform_contract(df_churn)

# Lo añade.
churn_data_prep_pipeline.append(churn_transform_contract)
```

Por otra parte, `Contract` se tratará como una variable numérica, por lo que se añade a `num_features` .

In [22]:
```
# Añade las variables
num_features.append('Contract')

# Las muestra
print('Variables numéricas: ', num_features)
```

```
Variables numéricas:  ['gender', 'Partner', 'Dependents', 'PhoneService', 'PaperlessBilling', 'Contract']
```

Total Charges

La columna `TotalCharges` , a pesar de estar representada como un objeto, contiene datos numéricos. Debido a esto, se va a transformar en numérica, asignando valores `NaN` a las entradas que no se han podido transformar en numéricas al leer el *DataFrame*. La función `churn_charges_to_numeric` toma como entrada un *DataFrame* y convierte la columna `TotalCharges` en numérica.

In [23]:
```
def churn_charges_to_numeric(df):
    df['TotalCharges'] = pd.to_numeric(df_churn['TotalCharges'], errors='coerce')
    return df # No hace falta

# Comprueba si la función está bien
display(churn_charges_to_numeric(df_churn.head(5).copy()))
```

customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	OnlineBackup	DeviceProtection	TechSupport	StreamingTV	Stream
6296-DDOOR	1	0	0	0	19	1	No	DSL	No	Yes	No	No	Yes	
3680-CTHUH	0	0	0	0	60	1	Yes	Fiber optic	Yes	Yes	Yes	Yes	Yes	
4931-TRZWN	1	0	0	0	13	1	No	DSL	Yes	No	No	No	Yes	
8559-WNQZS	0	0	0	0	1	1	No	No	No internet service	No internet service	No internet service	No internet service	No internet service	
3537-HPKQT	1	0	1	0	55	1	Yes	Fiber optic	No	Yes	No	No	No	

La función `churn_charges_to_numeric` también se añade `churn_data_prep_pipeline` y se aplica `df_churn` .

In [24]:
```
# Lo aplica
churn_charges_to_numeric(df_churn)

# Lo añade.
churn_data_prep_pipeline.append(churn_charges_to_numeric)
```

Se ha de incluir `TotalCharges` en la lista de columnas numéricas, `num_features` .

In [25]:
```
# Añade las variables
num_features.append('TotalCharges')

# Las muestra
print('Variables numéricas: ', num_features)
```

```
Variables numéricas:  ['gender', 'Partner', 'Dependents', 'PhoneService', 'PaperlessBilling', 'Contract', 'TotalCharges']
```

Hasta el momento, estos son los pasos que se han llevado a cabo en el preprocesamiento.

In [26]:
```
print("\nPasos de preprocesamiento: ")
for step, function in enumerate(churn_data_prep_pipeline):
    print("\t {:d}: {:s}".format(step, function.__name__))
```

```
Pasos de preprocesamiento:
         0: churn_binarize_dis
         1: churn_transform_contract
         2: churn_charges_to_numeric
```

Y estas son las características (*originalmente discretas*) que serán tratadas como categóricas y como numéricas.

In [27]:
```
print('Variables que se tratarán como categóricas: \n\t',cat_features, end='\n\n')
print('Variables que se tratarán como numéricas: \n\t',num_features)
```

```
Variables que se tratarán como categóricas:
         ['MultipleLines', 'InternetService', 'OnlineSecurity', 'OnlineBackup', 'DeviceProtection', 'TechSupport', 'StreamingTV', 'StreamingMovies', 'Payme
ntMethod']

Variables que se tratarán como numéricas:
         ['gender', 'Partner', 'Dependents', 'PhoneService', 'PaperlessBilling', 'Contract', 'TotalCharges']
```

Tratamiento de las columnas numéricas

El tratamiento de las columnas numéricas es relativamente sencillo, y se puede descomponer en varias etapas:

- Comprobar que, efectivamente, corresponden a características numéricas.
- Detección y tratamiento de outliers.
- Detección y tratemiento de valores perdidos.
- Exploración de las variables.

> ❗ Esta es una posible descomposición, y ha de considerarse como sugerencia. Si los *outliers* se sustituyen por valores perdidos, deben tratarse antes que estos.

Comprobación de tipos

En algunos casos, las columnas numéricas pueden representar características categóricas. Para detectar esta situación es posible apoyarse, además de en el nombre (descriptivo) de las columnas, en el número de valores que éstas toman.

> ℹ Otra opción consiste en elaborar gráficas (pero se harán después porque en este caso no es necesario).

A continuación, se obtiene el número de valores para cada una de las variables que han sido almacenadas como numéricas en el *DataFrame* original (`num_df_columns`).

In [28]:
```
list(map(lambda col: "{:s}: {:d}".format(col,len(df_churn[col].value_counts())), num_df_columns))
```

Out[28]:
```
['SeniorCitizen: 2', 'tenure: 73', 'MonthlyCharges: 1529', 'Churn: 2']
```

Parece dos de las variables son binarias, y otras dos son numéricas, por lo que no hay que hacer cambios.

Outliers

Existen varios métodos para llevar a cabo la detección de outliers. Por ejemplo, en el caso de que el número de características sea reducido, y que las escalas sean similares, se podría utilizar un gráfico de cajas. Otra posibilidad consite en visualizar los rangos que toman las variables.

In [29]:
```
df_churn[num_df_columns.to_list()+['TotalCharges']].describe(percentiles=[0.01,0.25,0.5,0.75,0.99])
```

Out[29]:

	SeniorCitizen	tenure	MonthlyCharges	Churn	TotalCharges
count	5986.000000	5986.000000	5986.000000	5986.000000	5977.000000
mean	0.166889	32.186435	64.821024	0.267959	2273.195483
std	0.372908	24.503081	30.033744	0.442933	2258.880108
min	0.000000	0.000000	18.250000	0.000000	18.800000
1%	0.000000	1.000000	19.150000	0.000000	19.850000
25%	0.000000	9.000000	35.825000	0.000000	400.000000
50%	0.000000	28.000000	70.400000	0.000000	1396.250000
75%	0.000000	55.000000	89.850000	1.000000	3772.650000
99%	1.000000	72.000000	114.557500	1.000000	8027.362000
max	1.000000	72.000000	118.750000	1.000000	8684.800000

Parece que los valores mínimo y máximo en las variables `tenure`, `MonthyCharges` y `TotalCharges` no se alejan excesivamente de los rangos razonables en ningún caso. Por tanto, no hay que proceder en este caso.

Valores perdidos

A continuación, se comprueba si existen valores perdidos para alguna de las variables numéricas.

In [30]:
```
df_churn[num_df_columns.to_list()+['TotalCharges']].isna().any()
```

Out[30]:
```
SeniorCitizen     False
tenure            False
MonthlyCharges    False
Churn             False
TotalCharges       True
dtype: bool
```

Puede apreciarse que solo los hay para `TotalCharges`.

> ❗ El tratamiento de los valores perdidos es conveniente hacerlo en el *pipeline* de `scikit-learn`, ya que en muchos casos se usará la media o mediana de la variable en el conjunto de entrenamiento, y este valor se ha de almacenar.

Exploración

Por último, puede resultar de interés llevar a cabo una pequeña exploración de las variables numéricas para ver tanto sus distribuciones como su relación con la clase. Es importante recordar que anteriormente se convirtieron algunas variables discretas a numéricas. Por tanto, se actualizará la lista `num_features`, añadiendole los elementos de `num_df_colums`, es decir, de las columnas que eran numéricas inicialmente.

In [31]:
```
num_features = num_features + num_df_columns.to_list()
num_features
```

Out[31]:
```
['gender',
 'Partner',
 'Dependents',
 'PhoneService',
 'PaperlessBilling',
 'Contract',
 'TotalCharges',
 'SeniorCitizen',
 'tenure',
 'MonthlyCharges',
 'Churn']
```

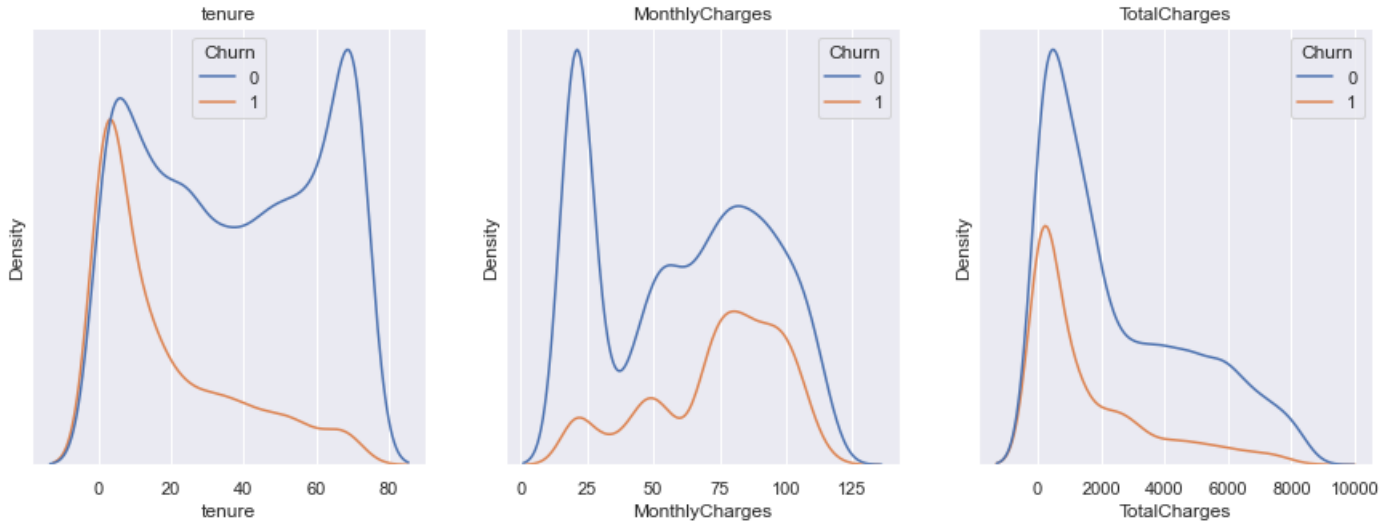
Ejercicio 6

Dibujar, para las variables `tenure`, `MonthlyCharge` y `TotalCharges` (las que no son binarias), un gráfico del tipo `sns.kdeplot()` que muestre la distribución de valores para cada una de las clases (dos distribuciones por gráfica). Utilizar una figura de 1×3 gráficas. Comentar qué se observa en la gráfica.

> ❗ Si da error, utilizar como fuente de datos `df_churn.dropna()`.

ℹ En este caso hay que dibujar tres gráficas. En cada una de ella se representa la distribución de los valores de la columna, pero se utiliza el color para diferenciar entre las dos categorías que puede tomar `Churn`.

In [32]:
```
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
features = ['tenure', 'MonthlyCharges','TotalCharges']

for col, ax in enumerate(axs.flatten()):

    # Dibujo la grafica
    col_name = features[col]
    sns.kdeplot(data=df_churn, x=features[col], hue="Churn", ax=axs[col])

    # Cambio titulo y quito los valores del eje Y
    ax.set_title(col_name);
    ax.set_yticks([])
```



Respuesta:

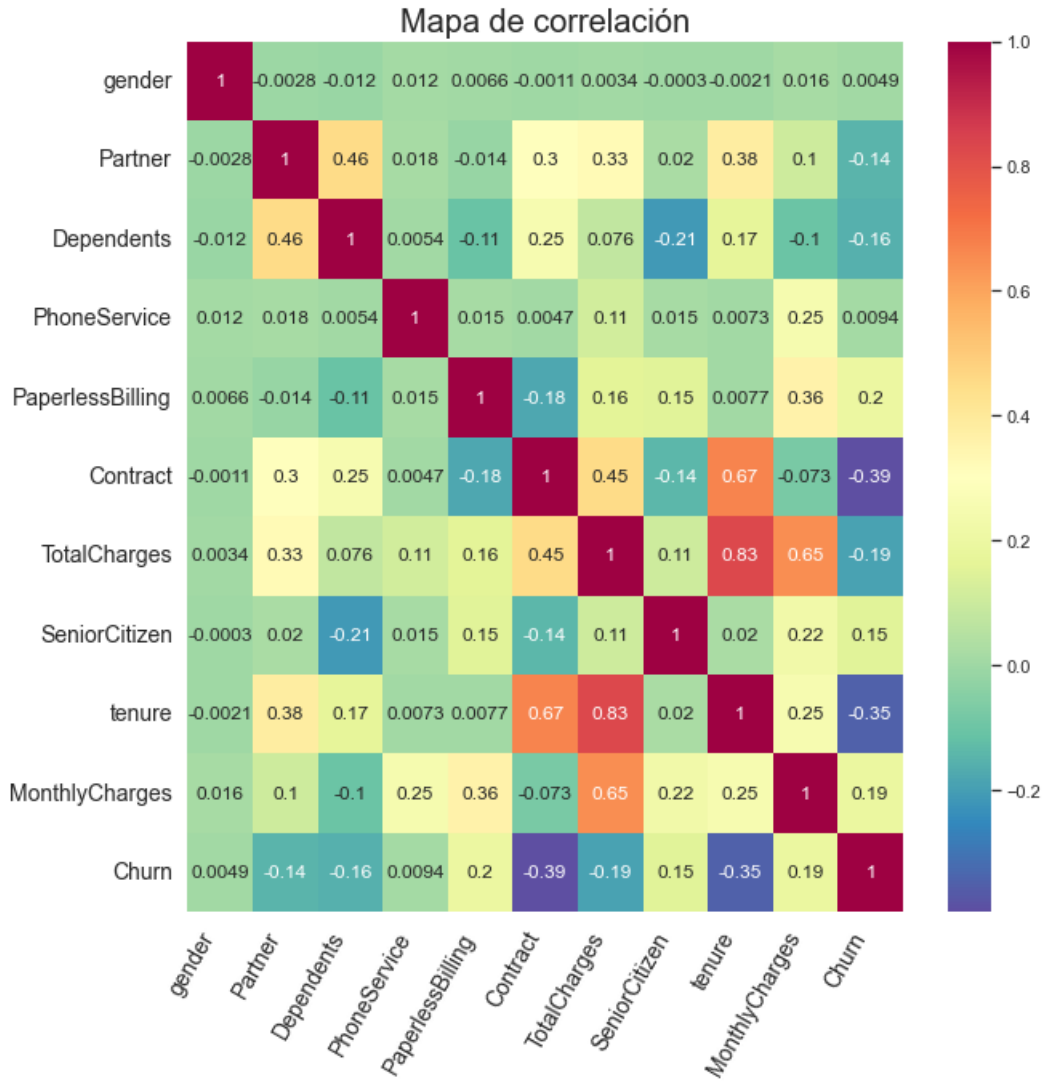
Dado que solo el 27% de las muestras presentan cancelaciones de suscripciones (Churn=1), es más útil analizar las gráficas en términos de tendencias que en términos absolutos.
- En cuanto a la variable 'Tenure', se observa un descenso de cancelaciones a medida que aumenta la permanencia.
- En el caso de 'MonthlyCharges', las curvas son muy similares. Sin embargo, en los rangos más bajos las cancelaciones son muy bajas, lo que sugiere que estos usuarios podrían estar acogidos a alguna oferta ligada a un contrato de permanencia.
- En cuanto a 'TotalCharges', las curvas de cancelaciones y no cancelaciones presentan una evolución muy similar.

☑

Ejercicio 7

Calcular la correlacción entre las variables numéricas y mostrarla en una gráfica `sns.heatmap()` .

In [33]:
```
# Calculo la matriz de correlación utilizando el método 'pearson'
corrmat = df_churn[num_features].corr(method='pearson')

# Creo un dataframe para filtrar las correlaciones duplicadas y autocorrelaciones
corr_matrix = pd.DataFrame(corrmat.unstack(), columns=['corr_value']).reset_index()
corr_matrix.columns = ['var1', 'var2', 'corr_value']
corr_matrix = corr_matrix[(corr_matrix['var1'] != corr_matrix['var2']) & (corr_matrix['corr_value'] != 1.0)]

# Ordeno el dataframe de correlaciones en orden descendente y ascendente
corr_matrix_desc = corr_matrix.sort_values(by='corr_value', ascending=False).round(2)
corr_matrix_asc = corr_matrix.sort_values(by='corr_value', ascending=True).round(2)

# Elimino las correlaciones duplicadas y las guardamos en una nueva columna
corr_matrix_desc['variables'] = corr_matrix_desc[['var1', 'var2']].apply(lambda x: '-'.join(sorted(x)), axis=1)
corr_matrix_asc['variables'] = corr_matrix_asc[['var1', 'var2']].apply(lambda x: '-'.join(sorted(x)), axis=1)
corr_matrix_desc = corr_matrix_desc.drop_duplicates(subset='variables')
corr_matrix_asc = corr_matrix_asc.drop_duplicates(subset='variables')

# Muestro el mapa de calor utilizando seaborn
plt.figure(figsize=(10, 10))
sns.heatmap(corrmat, annot=True, cmap='Spectral_r')
plt.title("Mapa de correlación",fontsize=20)
plt.xticks(fontsize=14, rotation=60, ha='right')
plt.yticks(fontsize=14)
plt.show()

# Muestro las primeras 10 correlaciones en ambos sentidos
print("Top 10 correlaciones más fuertes:")
display(corr_matrix_desc.head(10))

print("\nTop 10 correlaciones más débiles:")
display(corr_matrix_asc.head(10))
```



Top 10 correlaciones más fuertes:

	var1	var2	corr_value	variables
74	TotalCharges	tenure	0.83	TotalCharges-tenure
93	tenure	Contract	0.67	Contract-tenure
105	MonthlyCharges	TotalCharges	0.65	MonthlyCharges-TotalCharges
23	Dependents	Partner	0.46	Dependents-Partner
61	Contract	TotalCharges	0.45	Contract-TotalCharges
19	Partner	tenure	0.38	Partner-tenure
53	PaperlessBilling	MonthlyCharges	0.36	MonthlyCharges-PaperlessBilling
17	Partner	TotalCharges	0.33	Partner-TotalCharges
56	Contract	Partner	0.30	Contract-Partner
27	Dependents	Contract	0.25	Contract-Dependents

```
Top 10 correlaciones más débiles:
```

	var1	var2	corr_value	variables
65	Contract	Churn	-0.39	Churn-Contract
118	Churn	tenure	-0.35	Churn-tenure
79	SeniorCitizen	Dependents	-0.21	Dependents-SeniorCitizen
116	Churn	TotalCharges	-0.19	Churn-TotalCharges
59	Contract	PaperlessBilling	-0.18	Contract-PaperlessBilling
32	Dependents	Churn	-0.16	Churn-Dependents
21	Partner	Churn	-0.14	Churn-Partner
62	Contract	SeniorCitizen	-0.14	Contract-SeniorCitizen
46	PaperlessBilling	Dependents	-0.11	Dependents-PaperlessBilling
31	Dependents	MonthlyCharges	-0.10	Dependents-MonthlyCharges

Ejercicio 8

Comentar qué se observa en la gráfica (si se observa o no alguna circunstancia relevante).

Respuesta:

Hay algunas relaciones entre variables que están positivamente correlacionadas entre sí, lo que significa que cuando una variable aumenta, la otra variable también aumenta. Por ejemplo, la variable 'TotalCharges' está positivamente correlacionada con la variable 'tenure', lo que sugiere que el gasto total de un usuario está estrechamente relacionado con el tiempo que ha estado en la compañía.
En cuanto a la relación entre la variable 'Partner' y la variable 'Dependents' (correlación directa), indica que aquellos que tienen una pareja es probable que también tengan hijos.

Otras relaciones entre variables están inversamente correlacionadas, lo que significa que cuando una variable aumenta, la otra variable disminuye. Por ejemplo, la variable 'Contract' está

☑

Preprocesamiento

Llegados a este punto, se han llevado a cabo las acciones de preprocesamiento necesarias. En el *DataFrame* resultante quedan características numéricas, y también categóricas, que serán tratadas como tal en pasos posteriores. Con respecto a las numéricas, ha de eliminarse `Churn`.

In [34]:
```
num_features.remove('Churn')

print("Variables numéricas: ")
print(num_features)
print("\nVariables categóricas")
print(cat_features)

df_churn.head()
```

```
Variables numéricas:
['gender', 'Partner', 'Dependents', 'PhoneService', 'PaperlessBilling', 'Contract', 'TotalCharges', 'SeniorCitizen', 'tenure', 'MonthlyCharges']

Variables categóricas
['MultipleLines', 'InternetService', 'OnlineSecurity', 'OnlineBackup', 'DeviceProtection', 'TechSupport', 'StreamingTV', 'StreamingMovies', 'PaymentMetho
d']
```

Out[34]:

customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	OnlineBackup	DeviceProtection	TechSupport	StreamingTV	Stream
6296-DDOOR	1	0	0	0	19	1	No	DSL	No	Yes	No	No	Yes	
3680-CTHUH	0	0	0	0	60	1	Yes	Fiber optic	Yes	Yes	Yes	Yes	Yes	
4931-TRZWN	1	0	0	0	13	1	No	DSL	Yes	No	No	No	Yes	
8559-WNQZS	0	0	0	0	1	1	No	No	No internet service	No internet service	No internet service	No internet service	No internet service	
3537-HPKQT	1	0	1	0	55	1	Yes	Fiber optic	No	Yes	No	No	No	

En relación al preprocesamiento, se han aplicado, sucesivamente, las siguientes funciones (deben aparecer las tres).

In [35]:
```
print("\nPasos de preprocesamiento: ")
for step, function in enumerate(churn_data_prep_pipeline):
    print("\t {:d}: {:s}".format(step, function.__name__))
```

```
Pasos de preprocesamiento:
         0: churn_binarize_dis
         1: churn_transform_contract
         2: churn_charges_to_numeric
```

La siguiente función, denominada `preprocess_data` , admite como parámetros un *DataFrame* y una lista de funciones como la anterior, y las aplica sucesivamente sobre el *DataFrame*.

In [36]:
```
def preprocess_data(df, churn_data_prep_pipeline):
    for function in churn_data_prep_pipeline:
        function(df)
    return df
```

> Una alternativa a este método sería utilizar `DataFrame.pipe()`, pero en este caso no es necesario.

La siguiente celda aplica la función `preprocess_data` sobre un *DataFrame* auxiliar. Como puede observarse, el formato devuelto es similar al de `df_churn` una vez hecho el preprocesamiento.

In [37]:
```
df_aux = pd.read_csv('data/churn/Telco-Customer-Churn.csv', index_col=0).sample(n=5, random_state=0)
preprocess_data(df_aux, churn_data_prep_pipeline)
```

Out[37]:

customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	OnlineBackup	DeviceProtection	TechSupport	StreamingTV	Stream
6296-DDOOR	1	0	0	0	19	1	No	DSL	No	Yes	No	No	Yes	
3680-CTHUH	0	0	0	0	60	1	Yes	Fiber optic	Yes	Yes	Yes	Yes	Yes	
4931-TRZWN	1	0	0	0	13	1	No	DSL	Yes	No	No	No	Yes	
8559-WNQZS	0	0	0	0	1	1	No	No	No internet service	No internet service	No internet service	No internet service	No internet service	
3537-HPKQT	1	0	1	0	55	1	Yes	Fiber optic	No	Yes	No	No	No	

Otra alternativa para implementar el pipeline de preprocesamiento es construir a posteriori el pipeline utilizando `DataFrame.pipe()`.

In [38]:
```
df_aux = pd.read_csv('data/churn/Telco-Customer-Churn.csv', index_col=0).sample(n=5, random_state=0)

def preprocess_data_pipe(df):
    return (df.pipe(churn_binarize_dis).
               pipe(churn_transform_contract).
               pipe(churn_charges_to_numeric))


preprocess_data_pipe(df_aux)
```

Out[38]:

customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	OnlineBackup	DeviceProtection	TechSupport	StreamingTV	Stream
6296-DDOOR	1	0	0	0	19	1	No	DSL	No	Yes	No	No	Yes	
3680-CTHUH	0	0	0	0	60	1	Yes	Fiber optic	Yes	Yes	Yes	Yes	Yes	
4931-TRZWN	1	0	0	0	13	1	No	DSL	Yes	No	No	No	Yes	
8559-WNQZS	0	0	0	0	1	1	No	No	No internet service	No internet service	No internet service	No internet service	No internet service	
3537-HPKQT	1	0	1	0	55	1	Yes	Fiber optic	No	Yes	No	No	No	

1.2. Construcción de un modelo

En este punto, se ha definido un flujo de acciones de preprocesamiento, y se ha preparado un *DataFrame* a partir del cual se puede construir y validar el modelo. Este conjunto de datos será el punto de partida para la definición de un *Pipeline* con `scikit-learn`, que también incluye ciertas acciones de transformación, y es "independiente" del proceso anterior. No obstante, sí que será necesario utilizar las listas de características elaboradas anteriormente. Por otra parte, se almacenarán las columnas de entrada en `X` y la clase en `y`.

In [39]:
```
X = df_churn.drop('Churn',axis=1)
y = df_churn['Churn']
```

Creación de un Pipeline para la transformación

El primer paso en la creación del *Pipeline* consiste en la transformación de los datos. Uno de los problemas con los que tradicionalmente se ha lidiado en este sentido, es que se han de tratar de manera separada los datos de distintas columnas, que luego han de ser unidas. Esto conlleva la implementación de transformadores. Desde la versión 0.20 `scikit-learn` proporciona el objeto `ColumnTransformer` (https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html#sklearn.compose.ColumnTransformer), que permite tratar por separado las columnas, y facilita enormemente la tarea de transformación.

Podéis leer un par de artículos sobre `ColumnTransformer` en estos enlaces: post 1 (https://medium.com/vickdata/easier-machine-learning-with-the-new-column-transformer-from-scikit-learn-c2268ea9564c) (básico) y post 2 (https://medium.com/dunder-data/from-pandas-to-scikit-learn-a-new-exciting-workflow-e88e2271ef62) (más complejo pero muy interesante).

In [40]:
```
#!pip install --upgrade sklearn
```

En este contexto, el preprocesamiento para todas las variables de un mismo tipo será similar, por lo que serán sometidas a la misma secuencia de transformaciones mediante un *Pipeline*. Una vez definido el *Pipeline* para cada tipo de datos, se aplicarán ambos a las columnas correspondientes mediante `ColumnTransformer` .

En primer lugar, se va a definir el proceso de transformación para las variables numéricas, que consistirá en:

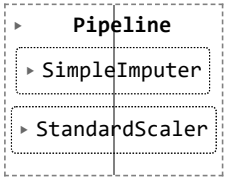
- Imputación de valores perdidos (`SimpleImputer`).
- Normalización a media cero y desviación uno (`StandardScaler`).

Ejercicio 9

Crear un `Pipeline` , denominado `num_transformer` , que consista en las dos transformaciones descritas anteriormente.

In [41]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

num_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
    ])

num_transformer
```

Out[41]:



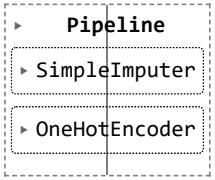
Importante. La normalización de las características mediante `StandardScaler` es, por ejemplo, una de las transformaciones que hay que hacer dentro del *pipeline*, ya que utiliza las medias y desviaciones estándar de las características en el conjunto de entrenamiento, y las almacena para reescalar en nuevos datos.

Ejercicio 10

Crear otro *Pipeline*, denominado `cat_transformer`, que defina la secuencia de transformaciones para las variables categóricas. Este debe estar formado por un objeto `SimpleImputer` que reemplace los valores perdidos por la etiqueta `missing` (`strategy='constant', fill_value='missing'`), y otro objeto, `OneHotEncoder` que transforme las variables categóricas a

In [42]:
```
from sklearn.preprocessing import OneHotEncoder

cat_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ])

cat_transformer
```

Out[42]:

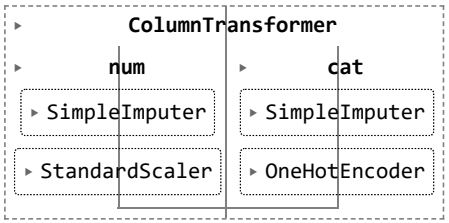


Ejercicio 11

Crear un objeto de tipo `ColumnTransformer`, denominado `churn_trans`, que aplique las dos secuencias anteriores de transformación sobre las características correspondientes.

In [43]:
```
from sklearn.compose import ColumnTransformer

churn_trans = ColumnTransformer(
    transformers=[
        ('num', num_transformer, num_features),
        ('cat', cat_transformer, cat_features)
        ]
    )

churn_trans
```

Out[43]:



Creación de un Pipeline con un modelo de regresión logística

Una vez defenidas las operaciones de transformación, se creará un *pipeline* que será utilizado para aprender un modelo de regresión logística.

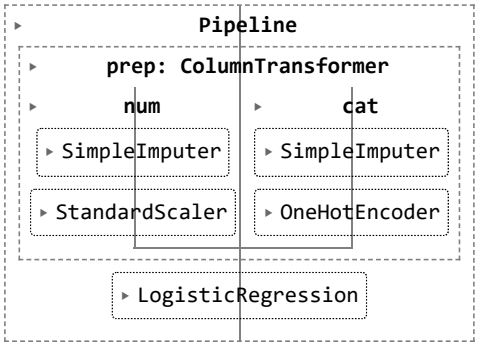
Ejercicio 12

Crear un objeto `Pipeline` que encadene la transformación definida en el objeto `churn_trans` anterior, y un modelo de regresión logística. Este modelo (*Pipeline*) se denominará `churn_pipe_logr`, y los pasos de transformación y clasificación, `prep` y `clas` respectivamente.

Crear un objeto `LogisticRegression` con `max_iter=200`.

In [44]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

logr_model = LogisticRegression(max_iter=200);

churn_pipe_logr = Pipeline(steps=[
    ('prep', churn_trans),
    ('clas', LogisticRegression(max_iter=200))
])

churn_pipe_logr
```

Out[44]:



De cara a aprender el modelo predictivo (*Pipeline* `churn_pipe_logr`), se han de ajustar varios parámetros. En concreto:

- El valor utilizado para imputar valores perdidos numéricos `strategy`, que podrá ser `mean` o `median`.
- La constante de regularización para regresión logística, `C`, que puede ser `[10e-3, 10e-2, 10e-1, 1, 10, 100, 1000]`.
- El parámetro `class_weight` en regresión logística, que puede ser `[None, 'balanced']`, y determina si el peso de cada ejemplo en la función del coste es el mismo o no.

Ejercicio 13

Entrenar un objeto `GridSearchCV` para determinar cual es la mejor configuración; almacenar el resultado en la variable `GS`. Utilizar validación cruzada de 5 folds, y la función `scoring` adecuada. Almacenar el mejor modelo encontrado en la variable `churn_pipe_logr` (la definida anteriormente).

> El parámetro `refit=True` (valor por defecto) hace que, una vez determinada la mejor configuración de parámetros, se entrene el modelo con todos los datos.

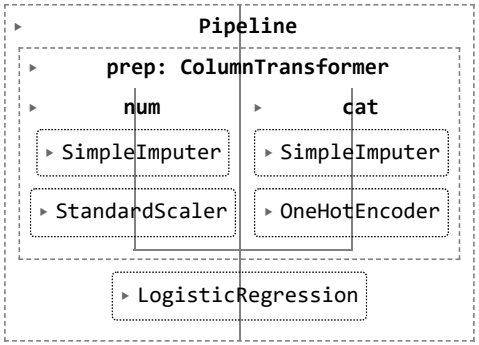
In [45]:
```
parameters = {}
parameters['prep__num__imputer__strategy'] = ['mean', 'median']
parameters['clas__C'] = [10e-3, 10e-2, 10e-1, 1, 10, 100, 1000]
parameters['clas__class_weight'] = [None, 'balanced']

GS = GridSearchCV(churn_pipe_logr,
                  param_grid=parameters,
                  scoring="recall",
                  cv=5,
                  refit=True)
GS.fit(X, y)

print("Mejor score: ", GS.best_score_)
print("Mejor configuración de parámetros: ", GS.best_params_)

churn_pipe_logr = GS.best_estimator_
```

```
Mejor score:  0.8111078660436137
Mejor configuración de parámetros:  {'clas__C': 100, 'clas__class_weight': 'balanced', 'prep__num__imputer__strategy': 'median'}
```

Es posible visualizar el `Pipeline` resultante.

In [46]:
```
churn_pipe_logr
```

Out[46]:



Ejercicio 14

Contestar a las siguientes preguntas.

1) ¿Para qué sirve el parámetro C (Equivale a $1/\lambda$, siendo λ el parámetro visto en clase)?
Respuesta:
El parámetro C es el inverso de la regularización λ. La regularización intenta encontrar un equilibrio entre ajustar el modelo a los datos de entrenamiento y predecir datos nuevos.
En un modelo de aprendizaje automático C determina qué tan riguroso es el modelo a la hora de penalizar errores en la predicción.
- Un valor más alto de C indica una mayor penalización por errores (modelo sobreajustado): el modelo va a poder predecir con mucha exactitud los datos de entrenamiento, pero no otros datos nuevos.
- Un valor más bajo de C indica una menor penalización por errores: el modelo no se ajusta tan bien a los datos de entrenamiento, pero que puede generalizar mejor a nuevos datos.

2) ¿Por qué se utilizan todos los datos `refit=True` para entrenar el mejor modelo una vez encontrados los parámetros?
Respuesta:
Cuando se establece refit=True, el modelo se entrenará con todos los datos disponibles después de encontrar los mejores parámetros utilizando la validación cruzada, lo que puede mejorar la precisión del modelo en la predicción de nuevos datos. Sin embargo, si el parámetro se establece refit=False, el modelo solo se entrenará con un subconjunto de los datos, lo que puede evitar que el modelo alcance su máximo potencial de precisión en la predicción de nuevos datos.

3) ¿Qué función has utilizado como `scoring` ? ¿Por qué?
Respuesta:
He utilizado scoring='recall' porque el objetivo es predecir con precisión qué clientes abandonarán la compañía para tomar medidas preventivas. En este caso, los falsos positivos no son tan problemáticos, ya que son clientes que se espera que se vayan de la compañía pero finalmente no lo hacen (lo que realmente queremos lograr).

Evaluación del modelo

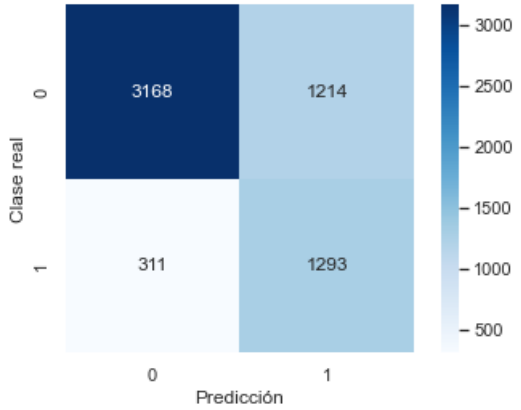
La función `show_results` recibe dos vectores de igual tamaño, denominados `y` e `y_pred` , con las salidas reales y predicciones del modelo respectivamente; dibuja la matriz de confusión; e imprime por pantalla las métricas de interés.

In [47]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

def show_results(y, y_pred):
    from sklearn.metrics import confusion_matrix
    c_mat = confusion_matrix(y, y_pred)
    sns.heatmap(c_mat, square=True, annot=True, fmt='d', cbar=True, cmap=plt.cm.Blues)
    plt.ylabel('Clase real')
    plt.xlabel('Predicción');
    plt.gca().set_ylim(2.0, 0)
    plt.show()
    print("Resultados: ")
    print(f'\taccuracy: {accuracy_score(y, y_pred):.3f}')
    print(f'\trecall: {recall_score(y, y_pred):.3f}')
    print(f'\tprecision: {precision_score(y, y_pred):.3f}')
    print(f'\tf1_score: {f1_score(y, y_pred):.3f}')
```

Ejercicio 15

Utilizar `show_results` para mostrar los resultados del modelo anterior y comentar los resultados

In [48]:
```
y_pred = churn_pipe_logr.predict(X)
show_results(y, y_pred)
```




```
Resultados:
        accuracy: 0.745
        recall: 0.806
        precision: 0.516
        f1_score: 0.629
```

Respuesta:

El objetivo principal es encontrar la mayor cantidad posible de personas que abandonen la compañía, lo que llamamos recall. Hemos logrado identificar correctamente alrededor del 80% de

☑

Umbral de clasificación

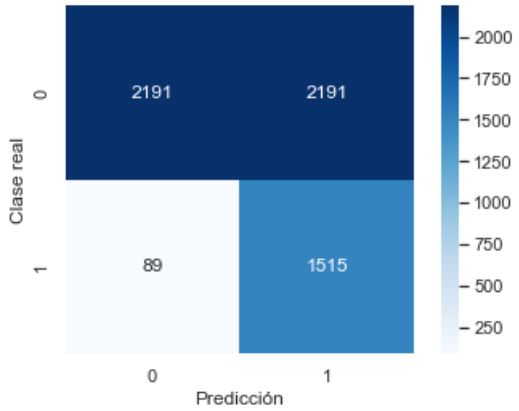
En el modelo de regresión logística, por ejemplo $c_\theta(x) = 1$ si $h_\theta(x) > 0.5$. Es posible cambiar este umbral (0.5), y con ello se modifican las propiedades del clasificador. Esto es aplicable a todos los modelos que devuelven probabilidad. En este apartado, veremos como afecta este cambio utilizando 3 umbrales: 0.25, 0.5, y 0.75.

> ❗ Por simplicidad, no incluiremos este proceso en el *Pipeline*.

In [49]:
```
y_prob = churn_pipe_logr.predict_proba(X)[:,1]
```

- $c_\theta(x) = 1$ si $h_\theta(x) >= 0.25$

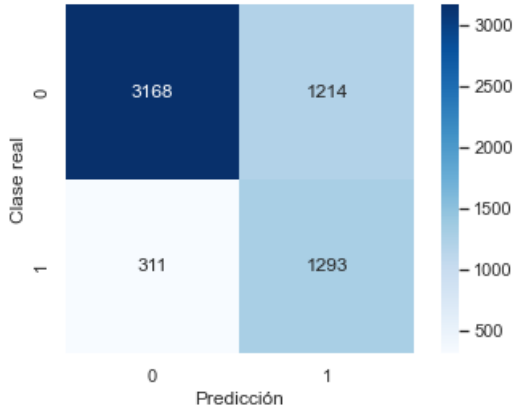
In [50]:
```
y_pred = y_prob >= 0.25
show_results(y, y_pred)
```




```
Resultados:
        accuracy: 0.619
        recall: 0.945
        precision: 0.409
        f1_score: 0.571
```

- $c_\theta(x) = 1$ si $h_\theta(x) >= 0.5$

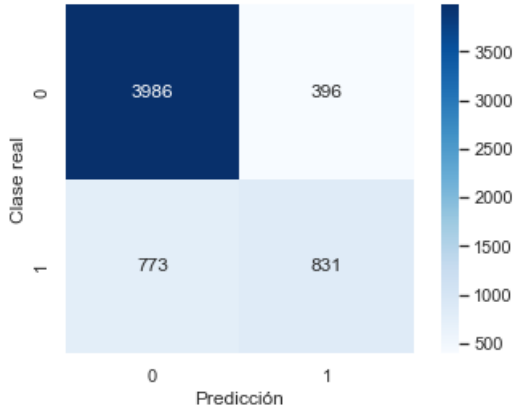
In [51]:
```
y_pred = y_prob >= 0.5
show_results(y, y_pred)
```




```
Resultados:
        accuracy: 0.745
        recall: 0.806
        precision: 0.516
        f1_score: 0.629
```

- $c_\theta(x) = 1$ si $h_\theta(x) >= 0.75$

In [52]:
```
y_pred = y_prob >= 0.75
show_results(y, y_pred)
```




```
Resultados:
        accuracy: 0.805
        recall: 0.518
        precision: 0.677
        f1_score: 0.587
```

Ejercicio 16

¿Cómo influye el cambio del umbral en las distintas métricas? ¿Por qué?

Respuesta:

El umbral de clasificación es el valor de probabilidad por encima del cual un modelo de clasificación considera que una instancia pertenece a una clase en particular. Dependiendo del umbral utilizado, se pueden obtener diferentes tasas de aciertos en la clasificación.
- Si el umbral es muy bajo ($h_0(x) = 0.25$), se pueden clasificar correctamente la mayoría de los positivos, pero se corre el riesgo de clasificar incorrectamente muchos negativos.
- Si es muy alto ($h_0(x) = 0.75$), se pueden clasificar correctamente muchos negativos, pero se falla en clasificar casi la mitad de los positivos (son los que más nos interesan).
- Con un umbral moderado ($h_0(x) = 0.5$) como el que utilizamos en nuestra predicción, logramos un equilibrio razonable entre la precisión en la clasificación de negativos y positivos.

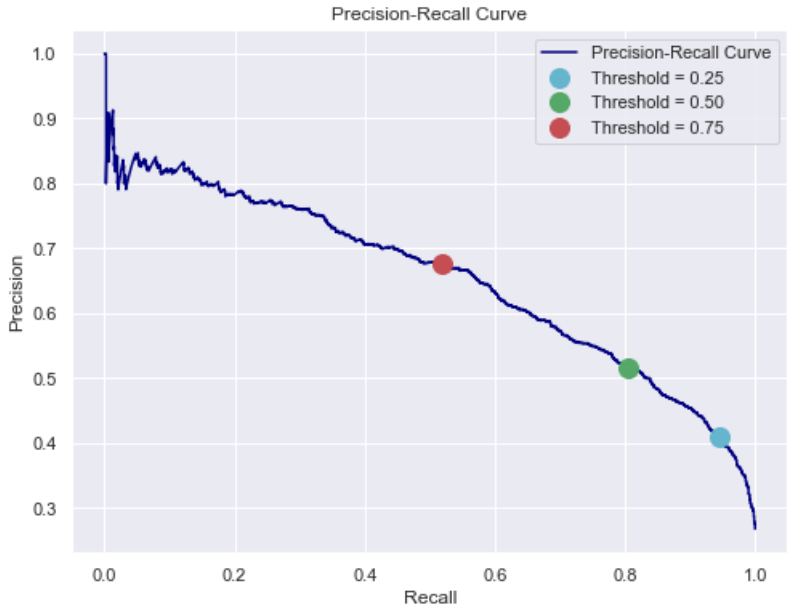
Es importante tener en cuenta que al aumentar el umbral, aumenta la precisión en la clasificación de la clase en cuestión, pero disminuye el recall, lo que puede ser un problema en problemas de clasificación desbalanceados donde es importante identificar correctamente los casos positivos.

Ejercicio 17

Dibujar la curva *precision/recall* utilizando `sklearn.metrics.precision_recall_curve` . ¿Hay un umbral más adecuado para este caso? ¿De qué factor dependería?

La función devuelve tres vectores, cada uno de ellos con los valores correspondientes a *precision* y *recall* d eun *threshold*, que es el tercer vector. Se trata de obtener estos valores y dibujar la curva correspondiente mediante, por ejemplo, `plt.plot()` .

In [53]:
```
from sklearn.metrics import precision_recall_curve
import matplotlib.pyplot as plt

# Calculo la curva de precision-recall
precision, recall, thresholds = precision_recall_curve(y_true=y, probas_pred=y_prob)

# Dibujo la curva de precision-recall
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot(recall, precision, color='navy')

# Defino los umbrales para encontrar los puntos de la curva más cercanos a ellos
thresholds_to_find = [0.25, 0.5, 0.75]
for th in thresholds_to_find:
    idx, nearest = min(enumerate(thresholds), key=lambda x: abs(x[1] - th))
    color = {0.25: 'oc', 0.5: 'og', 0.75: 'or'}[th]
    plt.plot(recall[idx], precision[idx], color, markersize=12)


# Agrego leyenda y etiquetas de los ejes
plt.legend(['Precision-Recall Curve', 'Threshold = 0.25', 'Threshold = 0.50', 'Threshold = 0.75'])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')

# Muestro el gráfico
plt.show()
```



Respuesta:
En el gráfico de Precision-Recall, la curva comienza en el extremo superior izquierdo, donde el Recall=0 y la Precision=1, lo que indica que no se ha identificado ningún verdadero positivo. A medida que el Recall aumenta, la Precision disminuye debido a la inclusión de falsos positivos. Cuando se llega al extremo inferior derecho del gráfico, con un Recall=1, significa que se han identificado todos los verdaderos positivos, pero la Precision ha disminuido significativamente debido a la inclusión de falsos positivos.
El umbral más adecuado en este caso es 0.5, ya que se tiene un alto Recall y una Precision no menor a 0.5.

1.3 Validación sobre nuevos datos

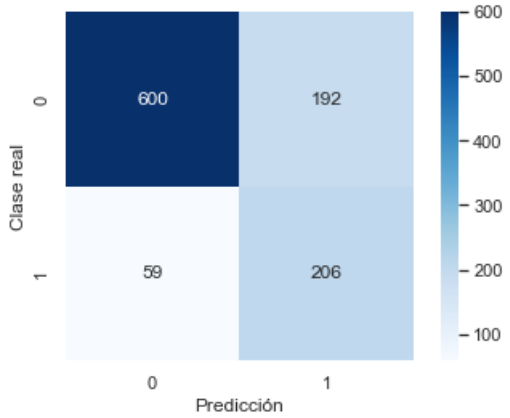
En este proceso se ha construído el modelo, y se dispone del flujo de trabajo completo, que se compone por una preparación de la base de datos original mediante `preprocess_data` , y del *Pipeline* `churn_pipe_logr` .

Ejercicio 18

Utilizar ambos para predecir los casos de abandono en `df_churn_new` . Puede utilizarse también un umbral de entre los seleccionados anteriormente. Mostrar los resultados.

> **Nota**: Ha de convertirse la clase a `int` porque ese paso se había sacado fuera del pipeline.

In [54]:
```
# Preproceso los datos de prueba
df_test = preprocess_data(df_churn_new, churn_data_prep_pipeline)

# Divido los datos
X_test = df_test.drop('Churn', axis = "columns")
y_test = (df_test['Churn']=='Yes').astype(int)

# Hago la predicción del conjunto de prueba utilizando el modelo de regresión logística
# Probabilidades de que el cliente abandone la compañía:
y_prob = churn_pipe_logr.predict_proba(X_test)[:,1]

# Clasificación de los clientes en función del umbral de clasificación de 0.5
y_pred = y_prob >= 0.5

# Muestro resultados
show_results(y_test, y_pred)
```




```
Resultados:
        accuracy: 0.763
        recall: 0.777
        precision: 0.518
        f1_score: 0.621
```

1.4 Comparación con otros clasificadores

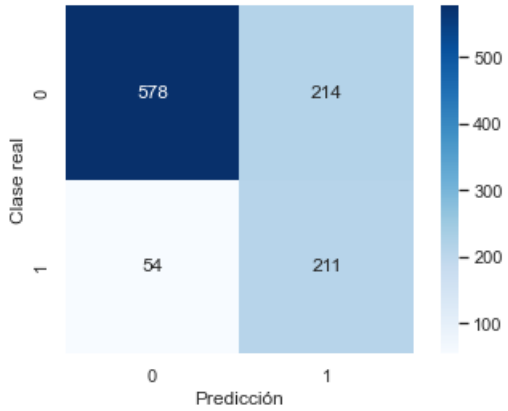
Una vez definido el proceso, se repetirá el entrenamiento con otros clasificadores: Un árbol y una máquina de soporte vectorial.

Ejercicio 19

Construir un *Pipeline* similar al anterior, denominado `churn_pipe_tree`, pero utilizando un árbol en lugar de un modelo de regresión logística. Determinar la mejor configuración de los parámetros con `GridSearchCV`, utilizando como `scoring` la medida que consideréis más adecuada. Guardar el modelo resultante en `churn_pipe_tree`. Mostrar los resultados con respecto al conjunto de nuevos datos `X_new, y_new`.

In [55]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Creo el modelo de árbol de decisión
dt_model = DecisionTreeClassifier()

# Creo el pipeline
churn_pipe_tree = Pipeline(steps=[('prep', churn_trans),
                                  ('clas', dt_model)])

# Defino los parámetros para la búsqueda
parameters = {
    'prep__num__imputer__strategy': ['mean', 'median'],
    'clas__max_depth': [None, 3, 5, 7, 9],
    'clas__class_weight': [None, 'balanced']
}

# Realizo la búsqueda de los mejores parámetros
GS = GridSearchCV(churn_pipe_tree, param_grid=parameters, cv=5, refit=True, scoring='f1')
GS.fit(X, y)

# Obtengo el mejor modelo
churn_pipe_tree = GS.best_estimator_

# Muestro los resultados de la búsqueda
print("Mejor score: ", GS.best_score_)
print("Mejor configuración de parámetros: ", GS.best_params_)

# Nuevo conjunto de datos
X_new = df_test.drop('Churn', axis = "columns")
y_new = (df_test['Churn']=='Yes').astype(int)

# Predigo la clase de los datos de prueba
y_prob = churn_pipe_tree.predict_proba(X_new)[:,1]
y_pred = y_prob >= 0.5

# Muestro los resultados de las métricas de evaluación
show_results(y_new, y_pred)
```

```
Mejor score:  0.6133507913283166
Mejor configuración de parámetros:  {'clas__class_weight': 'balanced', 'clas__max_depth': 5, 'prep__num__imputer__strategy': 'mean'}
```



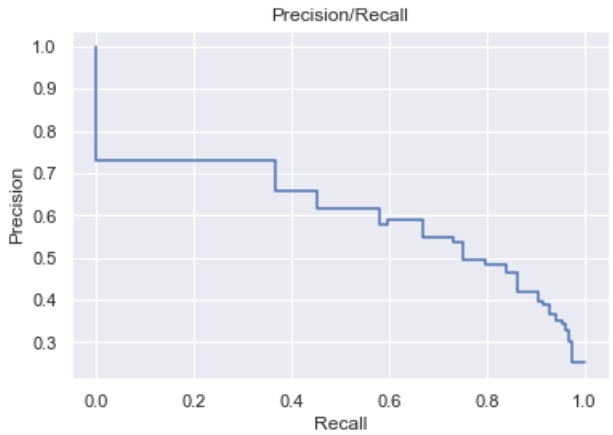

```
Resultados:
        accuracy: 0.746
        recall: 0.796
        precision: 0.496
        f1_score: 0.612
```

Ejercicio 20

Dibujar la curva *Precision/Recall* también con respecto a los nuevos datos.

> **Nota**: La curva se puede dibujar con `sklearn.metrics.PrecisionRecallDisplay.from_estimator` . Puede verse que cambia el aspecto, ya que, en realidad, la curva se debe hacer escalonada.

In [56]:
```
from sklearn.metrics import PrecisionRecallDisplay

# Calculo la precisión, el recall y el umbral
precision, recall, thresholds = precision_recall_curve(y_true=y_new, probas_pred = y_prob)

# Creo el gráfico de precisión/recall
prd = PrecisionRecallDisplay(precision=precision, recall=recall)
prd.plot()

# Pongo un título al gráfico
prd.ax_.set_title('Precision/Recall')

# Muestro el gráfico
plt.show()
```



☑

Ejercicio 21

¿Cómo aumentaríamos al precisión hasta un 0.75 (con respecto a los datos obtenidos en el ejercicio anterior, con umbral 0.5)? ¿Compensaría hacerlo?

Respuesta:

Para alcanzar una precisión del 0.75 con respecto a los datos obtenidos anteriormente, tendríamos que aumentar el umbral utilizado en el análisis. Sin embargo, al hacerlo, es probable que obtengamos valores de Recall por debajo del 0.4.
Esto no sería beneficioso para este problema en particular, ya que el objetivo principal es detectar todos los clientes que podrían abandonar la compañía en el futuro.

☑

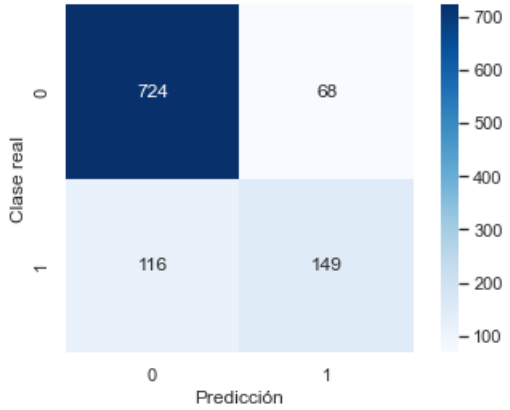
Ejercicio 22

Construir un *Pipeline* similar al anterior, denominado `churn_pipe_svc`, y que utilice una máquina de soporte vectorial. Determinar la mejor configuración de los parámetros con `GridSearchCV`, utilizando como `scoring` la medida que consideréis más adecuada. Guardar el modelo resultante en `churn_pipe_svc`. Mostrar los resultados con respecto al conjunto de nuevos datos `X_new, y_new`.

In [57]:
```
import os
import joblib
from sklearn.svm import SVC

# Creo una instancia de la clase SVC
svm_model = SVC(probability=True);

# Hago un pipeline con el preprocesado de los datos y nuestro modelo SVM
churn_pipe_svc = Pipeline(steps=[('prep', churn_trans),
                                 ('clas', svm_model)])

# Creo un diccionario de parámetros para la búsqueda de hiperparámetros
parameters = {}
parameters['prep__num__imputer__strategy'] = ['mean', 'median']
parameters['clas__C'] = [10e-2, 1, 100]
parameters['clas__kernel'] = ['linear', 'rbf']

# Guardo el modelo
filename = 'churn_pipe_svc_model.joblib'

if not os.path.exists(filename):
    # Se entrena el modelo
    GS = GridSearchCV(churn_pipe_svc, param_grid=parameters, scoring='recall', cv=5, refit=True)
    GS.fit(X, y)
    print(GS.best_params_)

    # Guardo el modelo entrenado
    joblib.dump(GS, filename)

else:
    # Carga el modelo guardado
    GS = joblib.load(filename)
    print(GS.best_params_)

# Obtengo el mejor modelo
churn_pipe_svc = GS.best_estimator_

# Muestro los resultados de la búsqueda
print("Mejor score: ", GS.best_score_)
print("Mejor configuración de parámetros: ", GS.best_params_)

# Predigo la clase de los datos de prueba
#y_prob = churn_pipe_svc.predict_proba(X_test)[:,1]
#y_pred = y_prob >= 0.25

# Muestro los resultados
#show_results(y_new, y_pred)

################################

# Nuevo conjunto de datos
X_new = df_test.drop('Churn', axis = "columns")
```

```
y_new = (df_test['Churn']=='Yes').astype(int)

# Predigo la clase de los datos de prueba
y_prob = churn_pipe_svc.predict_proba(X_new)[:,1]
y_pred = y_prob >= 0.5

# Muestro los resultados
show_results(y_new, y_pred)
```

```
{'clas__C': 100, 'clas__kernel': 'linear', 'prep__num__imputer__strategy': 'median'}
Mejor score:  0.54116238317757
Mejor configuración de parámetros:  {'clas__C': 100, 'clas__kernel': 'linear', 'prep__num__imputer__strategy': 'median'}
```

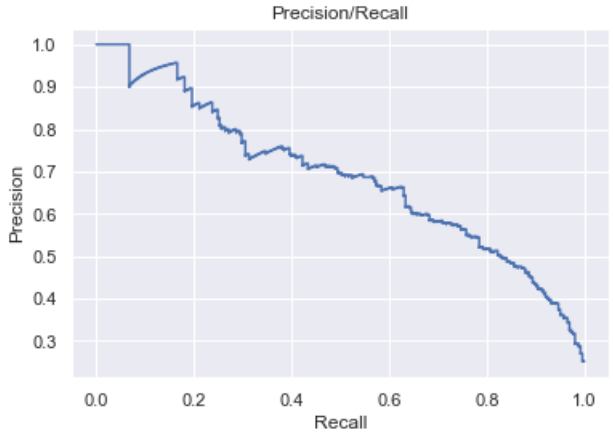



```
Resultados:
        accuracy: 0.826
        recall: 0.562
        precision: 0.687
        f1_score: 0.618
```

Ejercicio 23

Dibujar la curva *Precision/Recall* también con respecto a los nuevos datos.

In [58]:
```
# Calculo la precisión, el recall y el umbral
precision, recall, thresholds = precision_recall_curve(y_true=y_new, probas_pred = y_prob)

# Creo el gráfico de precisión/recall
prd = PrecisionRecallDisplay(precision=precision, recall=recall)
prd.plot()

# Pongo un título al gráfico
prd.ax_.set_title('Precision/Recall')

# Muestro el gráfico
plt.show()
```



Comparación de los modelos

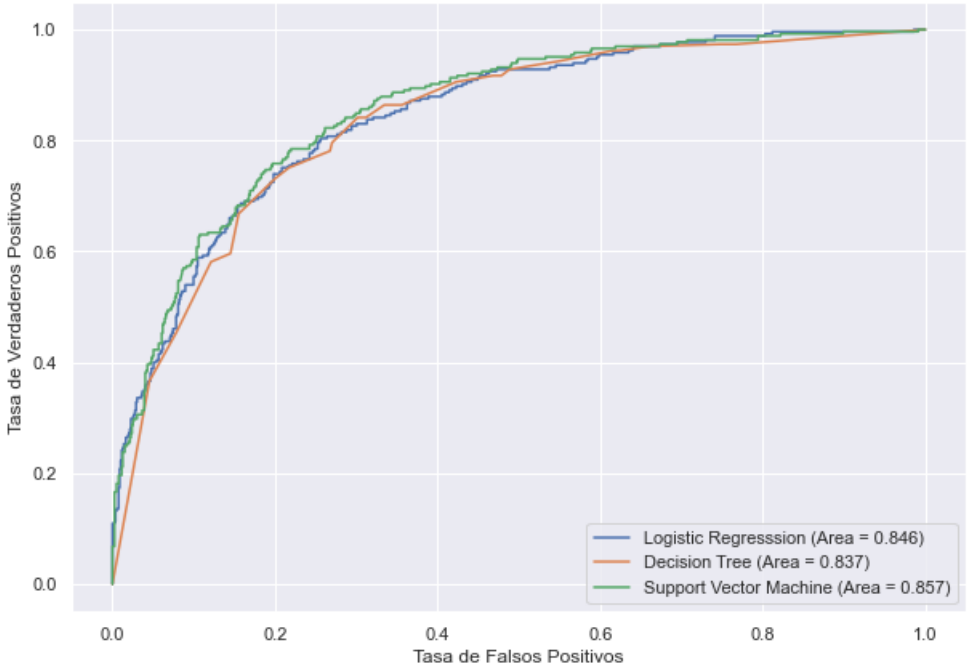
Como se ha visto anteriormente, el rendimiento de cada modelo de clasificación depende del umbral a partir del cual se considere una predicción como positiva. A la hora de comparar se podría determinar el umbral más conveniente para cada uno, y después utilizar los resultados obtenidos.

El área bajo la curva ROC (AUC) proporciona una medida del rendimiento considerando todos los umbrales posibles. Representa la probabilidad de que el modelo asigne un *score* mayor a un modelo positivo que a un negativo, y es invariable con respecto a este factor. Por eso se utiliza para la comparación de modelos.

Ejercicio 24

Dibujar la curva ROC para los tres modelos (utilizar los datos nuevos). ¿Qué clasificador es mejor?

In [60]:
```
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score

# Predigo las probabilidades con cada modelo
y_prob_logr = churn_pipe_logr.predict_proba(X_new)[:,1]
y_prob_tree = churn_pipe_tree.predict_proba(X_new)[:,1]
y_prob_svc = churn_pipe_svc.predict_proba(X_new)[:,1]

# Calculo las tasas de cada modelo
tasa_falsos_positivos_logr, tasa_verdaderos_positivos_logr, thresholds_logr = roc_curve(y_new, y_prob_logr)
tasa_falsos_positivos_tree, tasa_verdaderos_positivos_tree, thresholds_tree = roc_curve(y_new, y_prob_tree)
tasa_falsos_positivos_svc,  tasa_verdaderos_positivos_svc,  thresholds_svc = roc_curve(y_new, y_prob_svc)

# Calculo del área bajo la curva ROC
logr_roc_area = roc_auc_score(y_test, y_prob_logr)
tree_roc_area = roc_auc_score(y_test, y_prob_tree)
svc_roc_area = roc_auc_score(y_test, y_prob_svc)

# Creo la figura
fig,ax = plt.subplots(figsize=(10, 7))
plt.plot(tasa_falsos_positivos_logr, tasa_verdaderos_positivos_logr, label='Logistic Regresssion (Area = {:.3f})'.format(logr_roc_area))
plt.plot(tasa_falsos_positivos_tree, tasa_verdaderos_positivos_tree, label='Decision Tree (Area = {:.3f})'.format(tree_roc_area))
plt.plot(tasa_falsos_positivos_svc,  tasa_verdaderos_positivos_svc,  label='Support Vector Machine (Area = {:.3f})'.format(svc_roc_area))
ax.set_ylabel("Tasa de Verdaderos Positivos")
ax.set_xlabel("Tasa de Falsos Positivos")
plt.legend()
plt.show()
```



Respuesta:

Los resultados muestran que los clasificadores tienen un rendimiento bastante bueno en términos generales, con áreas bajo la curva (AUC) que rondan el 0.8. Si se compara con el clasificador ideal (AUC=1), se puede decir que estos modelos todavía tienen margen de mejora pero que en general están funcionando bien.
La Máquina de Vectores de Soporte es el clasificador con el mejor rendimiento. Sin embargo, la diferencia entre los tres clasificadores es relativamente pequeña, por lo que en este problema en particular, la elección de un clasificador específico puede depender de otros factores, como la interpretabilidad del modelo o la eficiencia computacional.

☑

Ejercicio 25

Mostrar el AUC para cada uno de los clasificadores. ¿Se corresponde con lo esperado?

In [61]:
```
from sklearn.metrics import roc_auc_score
print("Regresión logística: ", roc_auc_score(y_new, y_prob_logr))
print("Árbol de decisión: ", roc_auc_score(y_new, y_prob_tree))
print("Máquina de vectores de soporte: ", roc_auc_score(y_new, y_prob_svc))
```

```
Regresión logística:  0.845926243567753
Árbol de decisión:  0.83732132647227
Máquina de vectores de soporte:  0.8565990089574994
```

2. Predicción del valor potencial de un cliente (*Customer Lifetime Value*)

El valor potencial de un cliente (*Customer Lifetime Value* o *CLV*) permite determinar el beneficio que un cliente puede proporcionar a lo largo de un periodo de tiempo. En muchos casos, este valor es función de otros, por lo que es posible elaborar modelos predictivos para llevar a cabo una estimación. En este ejercicio se parte de un conjunto de datos denominado `Marketing-Customer-Value-Analysis.csv`, obtenido también del sitio de IBM analytics (https://www.ibm.com/analytics) con datos sobre CLV.

> A diferencia de la tarea anterior, en este caso el preprocesamiento se hará de modo más simple (no es necesario hacer más), y el trabajo se centrará en el desarrollo y análisis de un modelo de regresión.

In [62]:
```
df_clv = pd.read_csv('data/clv/Marketing-Customer-Value-Analysis.csv', index_col=0)

print("Tamaño del conjunto de datos:  %d" % df_clv.shape[0])
print("Número de variables: %d" % df_clv.shape[1])
if df_clv.index.is_unique:
    print('El índice es único.')
else:
    print('Los índices están duplicados.')

# Visualiza las primeras instancias
df_clv.head()
```

```
Tamaño del conjunto de datos:  9134
Número de variables: 23
El índice es único.
```

Out[62]:

Customer	State	Customer Lifetime Value	Response	Coverage	Education	Effective To Date	EmploymentStatus	Gender	Income	Location Code	...	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Policy Type	Policy	Renew Offer Type
BU79786	Washington	2763.519279	No	Basic	Bachelor	2/24/11	Employed	F	56274	Suburban	...	5	0	1	Corporate Auto	Corporate L3	Offer1
QZ44356	Arizona	6979.535903	No	Extended	Bachelor	1/31/11	Unemployed	F	0	Suburban	...	42	0	8	Personal Auto	Personal L3	Offer3
AI49188	Nevada	12887.431650	No	Premium	Bachelor	2/19/11	Employed	F	48767	Suburban	...	38	0	2	Personal Auto	Personal L3	Offer1
WW63253	California	7645.861827	No	Basic	Bachelor	1/20/11	Unemployed	M	0	Suburban	...	65	0	7	Corporate Auto	Corporate L2	Offer1
HB64268	Washington	2813.692575	No	Basic	Bachelor	2/3/11	Employed	M	43836	Rural	...	44	0	1	Personal Auto	Personal L1	Offer1

5 rows × 23 columns

Los nombres de las columnas numéricas del *DataFrame* se almacenan en una lista denominada `num_df_columns` , y el resto en otra denominada `dis_df_columns` .

In [63]:
```
dis_df_columns = df_clv.select_dtypes(exclude=np.number).columns.to_list()
num_df_columns = df_clv.select_dtypes(include=np.number).columns.to_list()

print('Discretas: ',dis_df_columns)
print('\nNuméricas: ',num_df_columns)
```

```
Discretas:  ['State', 'Response', 'Coverage', 'Education', 'Effective To Date', 'EmploymentStatus', 'Gender', 'Location Code', 'Marital Status', 'Policy Ty
pe', 'Policy', 'Renew Offer Type', 'Sales Channel', 'Vehicle Class', 'Vehicle Size']

Numéricas:  ['Customer Lifetime Value', 'Income', 'Monthly Premium Auto', 'Months Since Last Claim', 'Months Since Policy Inception', 'Number of Open Compl
aints', 'Number of Policies', 'Total Claim Amount']
```

En primer lugar, se procederá con las columnas que contienen valores discretos. Igual que anteriormente, conviene ver el número de valores que toma cada una para comprobar que, efectivamente, la representación corresponde a los datos. A continuación, se obtiene el número de valores para cada una de las variables discretas y se almacena en una lista denominada `num_values_dis_df_col`. Cada elemento de la lista ha de ser una tupla con el nombre de la columna y el número de variables.

In [64]:
```
num_values_dis_df_col = list(map(lambda col: (col,len(df_clv[col].value_counts())), dis_df_columns))
num_values_dis_df_col
```

Out[64]:
```
[('State', 5),
 ('Response', 2),
 ('Coverage', 3),
 ('Education', 5),
 ('Effective To Date', 59),
 ('EmploymentStatus', 5),
 ('Gender', 2),
 ('Location Code', 3),
 ('Marital Status', 3),
 ('Policy Type', 3),
 ('Policy', 9),
 ('Renew Offer Type', 4),
 ('Sales Channel', 4),
 ('Vehicle Class', 6),
 ('Vehicle Size', 3)]
```

La columna `Effective To Date` contiene datos relativos a fechas. Estos no pueden manejarse directamente en `scikit-learn`. Sin embargo, es posible convertirlos a enteros que representen días transcurridos desde una fecha determinada. Por ejemplo, la más temprana que aparezca en la columna.

En la siguiente celda, se convierte la columna `Effective To Date` a formato `DateTime`. Posteriormente se calcula la diferencia con la primera fecha y se convierte a entera (con `TimeDelta.dt.days`). Por último, se elimina la columna de `num_df_columns` y se añade a `dis_df_columns`.

In [65]:
```
df_clv['Effective To Date'] = pd.to_datetime(df_clv['Effective To Date'])

min_date = min(df_clv['Effective To Date'])
df_clv['Effective To Date'] = df_clv['Effective To Date'].apply(lambda date: date - min_date)
df_clv['Effective To Date'] = df_clv['Effective To Date'].dt.days

num_df_columns.append('Effective To Date')
dis_df_columns.remove('Effective To Date')
```

Debido a que solamente dos variables son binarias, y con el ánimo de simplificar, se tratarán todas como categóricas.

A continuación se explorarán las variables numéricas para encontrar outliers.

In [66]:
```
df_clv[num_df_columns].describe(percentiles=[0.01,0.05,0.25,0.5,0.75,0.95,0.99])
```

Out[66]:

	Customer Lifetime Value	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Total Claim Amount	Effective To Date
count	9134.000000	9134.000000	9134.000000	9134.000000	9134.000000	9134.000000	9134.000000	9134.000000	9134.000000
mean	8004.940475	37657.380009	93.219291	15.097000	48.064594	0.384388	2.966170	434.088794	28.837749
std	6870.967608	30379.904734	34.407967	10.073257	27.905991	0.910384	2.390182	290.500092	16.942769
min	1898.007675	0.000000	61.000000	0.000000	0.000000	0.000000	1.000000	0.099007	0.000000
1%	2230.433731	0.000000	61.000000	0.000000	1.000000	0.000000	1.000000	10.402835	0.000000
5%	2475.109047	0.000000	62.000000	1.000000	4.000000	0.000000	1.000000	52.261227	2.000000
25%	3994.251794	0.000000	68.000000	6.000000	24.000000	0.000000	1.000000	272.258244	14.000000
50%	5780.182197	33889.500000	83.000000	14.000000	48.000000	0.000000	2.000000	383.945434	28.000000
75%	8962.167041	62320.000000	109.000000	23.000000	71.000000	0.000000	4.000000	547.514839	43.000000
95%	22064.361267	90374.350000	163.350000	33.000000	93.000000	3.000000	8.000000	960.115399	56.000000
99%	35971.104520	97831.340000	228.670000	35.000000	98.000000	4.000000	9.000000	1408.560051	58.000000
max	83325.381190	99981.000000	298.000000	35.000000	99.000000	5.000000	9.000000	2893.239678	58.000000

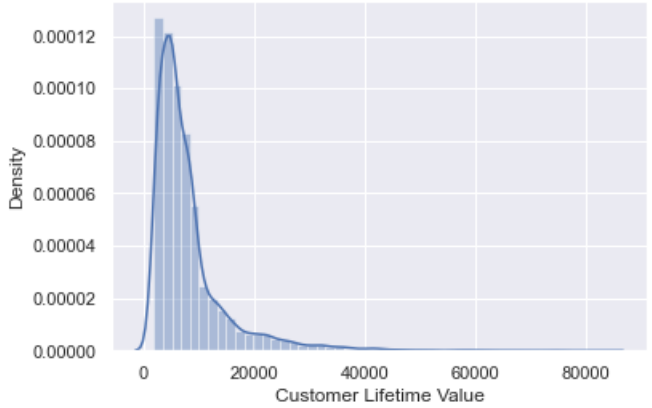
Parece que solamente la variable de clase, `Customer Lifetime Value` podría presentar *outliers*. Para comprobar si esos valores extremos corresponden a un error o son parte del "fenómeno" que representa la variable, puede dibujarse la distribución de la misma.

Ejercicio 26

Dibujar la distribución de la variable `Customer Lifetime Value` con una gráfica `sns.distplot()` / `sns.kdeplot()` .

In [67]:
```
import seaborn as sns
sns.distplot(df_clv["Customer Lifetime Value"]);
```

```
C:\Users\mbc98\AppData\Local\Temp\ipykernel_776\815063694.py:2: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751 (https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751)

  sns.distplot(df_clv["Customer Lifetime Value"]);
```



Se aprecia que los valores extremos no son anomalías o errores. Sin embargo, la diferencia con la mayoría de valores es tan grande, y su proporción en la base de datos tan reducida, que conviene no incluirlos en el modelo. Por tanto, se van a considerar los clientes con un $CLV < 22000$, es decir, aproximadamente el 95%.

Una vez determinados los tipos de las características, y filtrados los casos extremos, se crearán los conjuntos de entrenamiento y test.

In [68]:
```
from sklearn.model_selection import train_test_split

num_df_columns.remove('Customer Lifetime Value')
X = df_clv[df_clv['Customer Lifetime Value']<22000].drop('Customer Lifetime Value', axis=1)
y = df_clv.loc[df_clv['Customer Lifetime Value']<22000,'Customer Lifetime Value']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=0)
```

☑

Para llevar a cabo la transformación se utilizará el mismo procedimiento que en la tarea anterior(los *Pipelines* `num_transformer` y `cat_transformer`).

Ejercicio 27

Crear un objeto `ColumnTransformer` que aplique el *Pipeline* de transformación `num_transformer` a las columnas en `num_df_colums` , el *Pipeline* `cat_transformer` a las columnas en `dis_df_columns` . Denominarlo `clf_trans` . (Ambos fueron creados en la sección anterior).

In [69]:
```
clv_trans = ColumnTransformer(transformers=[('num', num_transformer, num_df_columns),
                                            ('cat', cat_transformer, dis_df_columns)]
                              )
```

Con respecto a la predicción, se utilizará inicialmente un modelo de regresión lineal.

Ejercicio 28

Crear un objeto *Pipeline*, denominado `churn_pipe_linr` a partir de `clv_trans` y un modelo de regresión, que estará almacenado en la variable `linr_model` . Entrenarlo con los datos de entrenamiento. Imprimir el *score* del modelo para entrenamiento y test. ¿Qué representa este *score*? ¿Qué valoración podría hacerse del mismo?

In [70]:
```
from sklearn.linear_model import LinearRegression

# Creo una instancia del modelo
linr_model = LinearRegression();

# Creo un pipeline
churn_pipe_linr =   Pipeline([('prep', clv_trans ),
                              ('clas', linr_model)])
# Entreno el modelo
churn_pipe_linr.fit(X_train, y_train);

# Imprimo los resultados
print("Entrenamiento:", churn_pipe_linr.score(X_train, y_train))
print("Test:", churn_pipe_linr.score(X_test, y_test))
```

```
Entrenamiento: 0.20364801290343515
Test: 0.2009558298286468
```

Respuesta:

El score del modelo es una medida de qué tan bien se ajusta el modelo a los datos de entrenamiento y test. Es un valor entre 0 y 1, donde un valor más cercano a 1 indica que el modelo se ajusta mejor a los datos. En este caso, el score del modelo para el entrenamiento y el test es bajo, lo que indica que el modelo no se ajusta muy bien a los datos y no es muy preciso en la predicción de los valores de respuesta. Por lo tanto, se puede decir que este modelo no es muy adecuado para predecir la variable objetivo en cuestión.

La función `show_errors` muestra el error absoluto medio con respecto a los datos de entrenamiento y test, y también una medida del error relativo absoluto medio. Mostrar los errores relativos al modelo `churn_pipe_linr` .

El error relativo absoluto medio se calculará aquí como:

$$error_{rel} = \frac{\sum_{i=1}^{m} \frac{|\hat{y_i} - y_i|}{y_i}}{m}.$$

Hemos creado expresamente esta última medida para hacernos una idea de la magnitud del error con respecto al *CLV* de cada usuario.

In [71]:
```
from sklearn.metrics import mean_absolute_error

def show_errors(y, y_pred):
    mae = mean_absolute_error(y, y_pred)
    rmae = np.sum(np.abs(y-y_pred)/y)/len(y)
    print(f'\tEl error absoluto medio es: {mae:.2f}')
    print(f'\tEl error absoluto relativo medio es: {rmae:.2f}')

print('Entrenamiento')
y_pred = churn_pipe_linr.predict(X_train)
show_errors(y_train, y_pred)

print('Test')
y_pred = churn_pipe_linr.predict(X_test)
show_errors(y_test, y_pred)
```

```
Entrenamiento
        El error absoluto medio es: 2627.97
        El error absoluto relativo medio es: 0.46
Test
        El error absoluto medio es: 2573.59
        El error absoluto relativo medio es: 0.46
```

2.1 Construcción de un árbol de regresión

Puede apreciarse que, tal y como era de esperar, el error es muy alto. Por tanto se va a definir el procedimiento anterior, pero con un árbol de regresión.

Ejercicio 29

Crear un *Pipeline* denominado `churn_pipe_dtr` , similar al anterior, pero en el que el modelo predictivo sea un árbol de regresión. Entrenarlo e imprimir su *score* para entrenamiento y test.

In [72]:
```
from sklearn.tree import DecisionTreeRegressor


dtr_model = DecisionTreeRegressor(random_state=0);

churn_pipe_dtr = Pipeline([ ('prep', clv_trans ),
                            ('clas', dtr_model)])

churn_pipe_dtr.fit(X_train, y_train);

print("Entrenamiento:", churn_pipe_dtr.score(X_train,y_train))
print("Test:", churn_pipe_dtr.score(X_test,y_test))
```

```
Entrenamiento: 1.0
Test: 0.702649059852204
```

☑

La siguiente celda muestra el error absoluto medio (también el relativo) para entrenamiento y test:

In [73]:
```
print('Entrenamiento')
y_pred = churn_pipe_dtr.predict(X_train)
show_errors(y_train, y_pred)

print('Test')
y_pred = churn_pipe_dtr.predict(X_test)
show_errors(y_test, y_pred)
```

```
Entrenamiento
        El error absoluto medio es: 0.00
        El error absoluto relativo medio es: 0.00
Test
        El error absoluto medio es: 692.63
        El error absoluto relativo medio es: 0.07
```

Por otra parte, el tamaño del árbol obtenido es:

In [74]:
```
print(f"Profundidad: {churn_pipe_dtr.named_steps['clas'].get_depth()}")
print(f"Número de hojas: {churn_pipe_dtr.named_steps['clas'].get_n_leaves()}")
```

```
Profundidad: 30
Número de hojas: 5278
```

Ejercicio 30

¿Qué conclusión se puede sacar de los resultados?

Respuesta:

En el análisis del modelo de regresión, vemos que el árbol de regresión puede predecir con mayor precisión que otros modelos. Sin embargo, existe una diferencia significativa entre la precisión en los datos de entrenamiento (1.0) y en los datos de prueba (0.7), lo que sugiere que el modelo puede estar sobreajustado a los datos de entrenamiento y no generalizar bien a otros datos nuevos.

En el caso anterior, no se había limitado la profundidad, por lo que el tamaño del árbol resultante es muy grande.

Ejercicio 31

A continuación se determinará, mediante `GridSearchCV` la profundidad adecuada. Crear un *Pipeline* similar anterior, denominado `churn_pipe_dtr_p` . Utilizar validación cruzada de 5 particiones, y `neg_mean_absolute_error` como medida de *scoring*. Almacenar el *Pipeline* resultante en `churn_pipe_dtr_p` . Por último, imprimir los *score* (R^2) del modelo obtenido.

In [75]:
```
parameters = {}
parameters['clas__max_depth'] = [None, 2, 5, 10, 15, 20]

dtr_p_model = DecisionTreeRegressor(random_state=0);

churn_pipe_dtr_p = Pipeline(steps=[('prep', clv_trans),
                                   ('clas', dtr_p_model)])

GS_p = GridSearchCV(churn_pipe_dtr_p, param_grid=parameters, scoring='neg_mean_absolute_error', cv=5, refit=True)
GS_p.fit(X_train, y_train)

# Muestro los resultados de la búsqueda
print("Mejor score: ", GS_p.best_score_)
print("Mejor configuración de parámetros: ", GS_p.best_params_)

# Obtengo el mejor modelo
churn_pipe_dtr_p = GS_p.best_estimator_

print("\nEntrenamiento:", churn_pipe_dtr_p.score(X_train, y_train))
print("Test:", churn_pipe_dtr_p.score(X_test, y_test))
```

```
Mejor score:  -725.3055980428136
Mejor configuración de parámetros:  {'clas__max_depth': 10}

Entrenamiento: 0.8893117529464726
Test: 0.7836483671865917
```

Por otra parte, el tamaño del árbol obtenido es ahora:

In [76]:
```
print(f"Profundidad: {churn_pipe_dtr_p.named_steps['clas'].get_depth()}")
print(f"Número de hojas: {churn_pipe_dtr_p.named_steps['clas'].get_n_leaves()}")
```

```
Profundidad: 10
Número de hojas: 695
```

Error absoluto medio (también el relativo) para entrenamiento y test:

In [77]:
```
print('Entrenamiento')
y_pred = churn_pipe_dtr_p.predict(X_train)
show_errors(y_train, y_pred)

print('Test')
y_pred = churn_pipe_dtr_p.predict(X_test)
show_errors(y_test, y_pred)
```

```
Entrenamiento
        El error absoluto medio es: 466.04
        El error absoluto relativo medio es: 0.05
Test
        El error absoluto medio es: 673.70
        El error absoluto relativo medio es: 0.07
```

☑

Ejercicio 32

Compara y analiza los resultados obtenidos con respecto a los obtenidos anteriormente.

Respuesta:

Se observa que el primer árbol de regresión presentaba sobreajuste debido a su gran profundidad de 30 y una puntuación perfecta de 1 en entrenamiento, pero una puntuación menor de 0.7 en la prueba.
En este apartado se ha utilizado GridSearchCV para determinar la profundidad más apropiada del árbol, resultando ser 10. Esto ha mejorado la capacidad predictiva del modelo, reduciendo el sobreajuste y obteniendo una puntuación de 0.89 en entrenamiento y 0.78 en la prueba.

☑

---

3. Segmentación de clientes

En el manejo de la relación con el cliente (*CRM*) es necesario tener en cuenta varios factores. Por un lado, cada cliente tiene unas peculiaridades, y la relación con él se ha de definir de manera concreta. Por otro lado, el número de clientes es elevado, y no es posible personalizar completamente el modo en que se va a llevar la relación. Debido a esto, una de las tareas relacionadas con ciencia de datos que son más comunes en *CRM* es la segmentación, cuyo objetivo es crear grupos de clientes con el mismo perfil.

Para abordar la segmentación se utilizan algoritmos de *clustering*. Una de las particularidades de estos algoritmos es que pierden utilidad cuando se trata con un número muy elevado de variables. Por eso suelen utilizarse con conjuntos reducidos. Un tipo de análisis frecuente en este sentido es el RFM (*Recency*, *Frequency*, *Monetary value*), en el que se segmenta a los clientes en función de cuando hicieron su última transacción, con qué frecuencia han hecho transacciones, y cuanto dinero han gastado en total.

En esta tarea se partirá de un conjunto de datos de venta online obtenido en el repositorio de conjuntos de datos de la UCI (https://archive.ics.uci.edu/ml/datasets/online+retail) (un subconjunto de éste). Tras una serie de transformaciones, se utilizará `KMeans` para caracterizar los clusters.

A continuación, se lee el conjunto de datos `data/segmentation/online12M.csv` y se guarda en el `DataFrame` `df_ol`. La columna `InvoiceDate` se trata como marca de tiempo.

In [78]:
```
# Carga los datos
df_ol = pd.read_csv('data/segmentation/online12M.csv', index_col=0, parse_dates=['InvoiceDate']);

print("Tamaño del conjunto de datos:  %d" % df_ol.shape[0])
print("Número de variables: %d" % df_ol.shape[1])
if df_ol.index.is_unique:
    print('El índice es único.')
else:
    print('Los índices están duplicados.')

# Visualiza las primeras instancias
df_ol.head()
```

```
Tamaño del conjunto de datos:  68176
Número de variables: 8
El índice es único.
```

Out[78]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
416792	572558	22745	POPPY'S PLAYHOUSE BEDROOM	6	2011-10-25	2.10	14286	United Kingdom
482904	577485	23196	VINTAGE LEAF MAGNETIC NOTEPAD	1	2011-11-20	1.45	16360	United Kingdom
263743	560034	23299	FOOD COVER WITH BEADS SET 2	6	2011-07-14	3.75	13933	United Kingdom
495549	578307	72349B	SET/6 PURPLE BUTTERFLY T-LIGHTS	1	2011-11-23	2.10	17290	United Kingdom
204384	554656	21756	BATH BUILDING BLOCK WORD	3	2011-05-25	5.95	17663	United Kingdom

Una de las columnas que vamos a utilizar (*Monetary Value*) hace referencia al precio total gastado por cliente. La columna `Total` almacena el resultado de multiplicar el precio por unidad de cada compra (`UnitPrice`) por la cantidad de unidades (`Quantity`).

In [79]:
```
df_ol['Total'] = df_ol['Quantity']*df_ol['UnitPrice']
df_ol.head()
```

Out[79]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	Total
416792	572558	22745	POPPY'S PLAYHOUSE BEDROOM	6	2011-10-25	2.10	14286	United Kingdom	12.60
482904	577485	23196	VINTAGE LEAF MAGNETIC NOTEPAD	1	2011-11-20	1.45	16360	United Kingdom	1.45
263743	560034	23299	FOOD COVER WITH BEADS SET 2	6	2011-07-14	3.75	13933	United Kingdom	22.50
495549	578307	72349B	SET/6 PURPLE BUTTERFLY T-LIGHTS	1	2011-11-23	2.10	17290	United Kingdom	2.10
204384	554656	21756	BATH BUILDING BLOCK WORD	3	2011-05-25	5.95	17663	United Kingdom	17.85

A continuación se crearán las columnas `Recency`, `Frequency` y `MonetaryValue`. Para ello, se han de agrupar los datos por cliente, y después hacer una agregación a partir del grupo sobre las columnas:

- `InvoiceDat`. Hay que crear un día de referencia (el posterior al último de la lista) y restarle el último día (`max`) en que compró cada cliente. El resultado, un `TimeDelta` se devuelve en días (`.days`).
- `InvoiceNo`. Se cuenta el número de facturas.
- `MonitaryValue`. Se suma, para cada grupo, la columna `Total`.

A continuación se obtiene la información según el procedimiento descrito, y se almacena en el `DataFrame` `df_rfm`.

In [80]:
```
import datetime
dia_ref = max(df_ol['InvoiceDate']) + datetime.timedelta(days=1)
```

In [81]:
```
df_rfm = df_ol.groupby(['CustomerID']).agg({
    'InvoiceDate': lambda x: (dia_ref - x.max()).days,
    'InvoiceNo': 'count',
    'Total': 'sum'})

df_rfm.rename(columns = {'InvoiceDate': 'Recency', 'InvoiceNo': 'Frequency', 'Total': 'MonetaryValue'}, inplace=True)

df_rfm.head()
```

Out[81]:

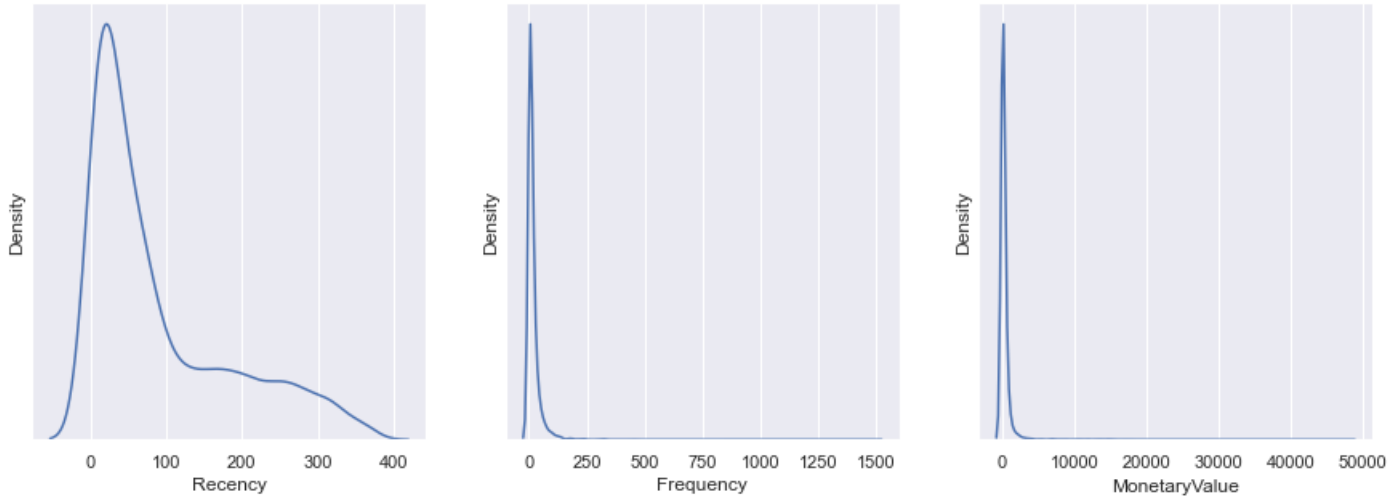
CustomerID	Recency	Frequency	MonetaryValue
12747	3	25	948.70
12748	1	888	7046.16
12749	4	37	813.45
12820	4	17	268.02
12822	71	9	146.15

Un valor más alto de `Recency` implica que la última compra del cliente ocurrió hace más tiempo.

Ejercicio 33

Dibujar la distribución de cada una de las variables mediante `sns.kdeplot()` en una figura de 1×3 gráficas.

In [82]:
```
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
features = ['Recency', 'Frequency','MonetaryValue']
for col, ax in enumerate(axs.flatten()):
    col_name = features[col]
    sns.kdeplot(df_rfm[features[col]], ax=ax)
    ax.set_yticks([])
```



Puede apreciarse que las distribuciones están muy sesgadas. Esto supone un problema para el algoritmo `KMeans` que construye los clusters por distancias. Una forma de solucionarlo es utilizar el logaritmo de las variables. El resultado se almacena en el `DataFrame` `dt_rfm_log` .

In [83]:
```
df_rfm_log = np.log(df_rfm)
```

La diferencia de escalas también supone un problema, ya que intrínsecamente, hace que el cálculo de las distancias de más importancia a las variables con mayor rango. Debido a esto, es necesario estandarizar los datos. El resultado se almacena en el *DataFrame* `df_rfm_norm` .

In [84]:
```
df_rfm_norm = df_rfm_log - df_rfm_log.mean()
df_rfm_norm = df_rfm_norm / df_rfm_log.std()
df_rfm_norm.describe()
```

Out[84]:

	Recency	Frequency	MonetaryValue
count	3.643000e+03	3.643000e+03	3.643000e+03
mean	-7.604344e-15	-1.081413e-14	-1.258779e-14
std	1.000000e+00	1.000000e+00	1.000000e+00
min	-2.814131e+00	-1.794485e+00	-4.092988e+00
25%	-6.373074e-01	-6.490904e-01	-6.578277e-01
50%	9.266763e-02	2.092218e-02	-1.489601e-02
75%	8.339256e-01	7.209827e-01	6.692029e-01
max	1.547663e+00	4.246243e+00	4.458854e+00

Una vez preparados los datos, se puede llevar a cabo el procedimiento de agrupación. El algoritmo `KMeans` toma como parametro más importante `k` que corresponde al número de clusters. Cuando no se tiene información a priori, se dibuja la curva del coste en función de `k`. Debido a que se busca un compromiso entre un bajo coste, y un número reducido de clusters, se toma aquel valor de `k` a partir del cual el descenso es menor. Este método se conoce como *método del codo*.

Ejercicio 34

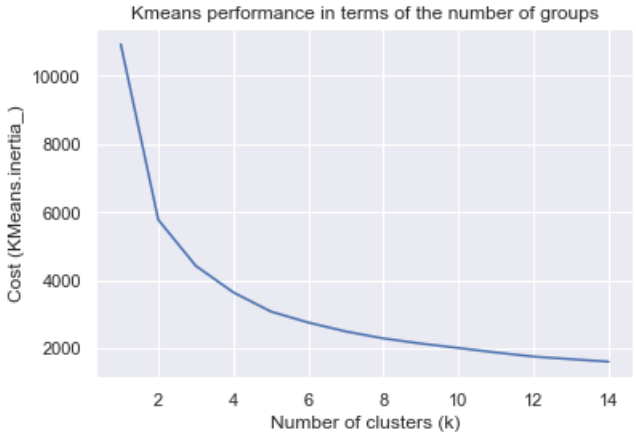
Ejecutar `KMeans` para valores del 1 al 15 sobre el conjunto de datos `df_rfm_norm`. Almacenando el coste resultante (`KMeans.inertia_`) para cada valor de `k` en el diccionario `cost`.

In [85]:
```
from sklearn.cluster import KMeans

cost = {}
for k in range(1, 15):
    kmeans = KMeans(n_clusters=k, random_state=0, n_init=10)
    kmeans.fit(df_rfm_norm)
    cost[k] = kmeans.inertia_
```

Ejercicio 35

Dibujar la función de coste (utilizar `cost.keys()` y `cost.values()`).

In [86]:
```
plt.plot(list(cost.keys()), list(cost.values()))
plt.title('Kmeans performance in terms of the number of groups')
plt.xlabel("Number of clusters (k)")
plt.ylabel("Cost (KMeans.inertia_)")
plt.show()
```



Parece que a partir de $k = 4$ el descenso es menor.

Ejercicio 36

Realizar el agrupamiento con `k=4` .

In [87]:
```
kmeans = KMeans(n_clusters=4, random_state=0, n_init=10)
kmeans.fit(df_rfm_norm)
```

Out[87]:
```
KMeans(n_clusters=4, n_init=10, random_state=0)
```

Con el fin de analizar cada grupo, se trabajará con los datos de `df_rfm` (podría trabajarse incluso con el conjunto inicial si fuese necesario). A continuación, se añade a cada entrada de `df_rfm` , una columna denominada `Group` con la etiqueta correspondiente al cluster (`KMeans.labels_`).

In [88]:
```
df_rfm = df_rfm.assign(Group = kmeans.labels_)
df_rfm.head()
```

Out[88]:

CustomerID	Recency	Frequency	MonetaryValue	Group
12747	3	25	948.70	0
12748	1	888	7046.16	0
12749	4	37	813.45	0
12820	4	17	268.02	0
12822	71	9	146.15	1

Ejercicio 37

Obtener la media por grupo para cada una de las columnas de `df_rfm` (excepto `Group`, obviamente). Almacenar el resultado en `df_clusters_mean`.

> Esto nos da los centroides, pero con respecto al conjunto original. `kmeans.cluster_centers_` nos daría esa información, pero con respecto al logaritmo de los datos normalizado. Lo que no es representativo de cara a describir los grupos.

In [89]:
```
df_clusters_mean = df_rfm.groupby(['Group']).mean()
display(df_clusters_mean)
```

Group	Recency	Frequency	MonetaryValue
0	19.475449	53.742515	1129.977401
1	129.379689	11.797804	231.814584
2	19.507171	10.348110	145.583978
3	165.422996	2.604430	44.169061

Un modo sencillo de visualizar los datos consiste dividir los valores medios de cada grupo por las medias de cada columna para el total de la información. Esto devolvería la importancia relativa de cada variable en cada cluster.

Ejercicio 38

Llevar a cabo esta operación y almacenar el resultado en `relative_imp`. Restar 1 al resultado para que un valor 0 corresponda con la media de cada columna (no hay diferencia entre el grupo y el total).

En el ejercicio anterior se han obtenido las medias para cada columna/grupo, y se han almacenado en `df_clusters_mean`. En este, se ha de calcular la media de cada columna de `df_rfm`, lo que da lugar a una *Serie* con tres elementos. La división del *DataFrame* por la *Serie* se hace por *Broadcasting*, por lo que se divide cada una de las filas. Al resultado hay que restarle uno.

In [90]:
```
df_rfm_mean = df_rfm[["Recency","Frequency","MonetaryValue"]].mean()
relative_imp = df_clusters_mean.div(df_rfm_mean, axis='columns')-1
relative_imp
display(relative_imp)
```

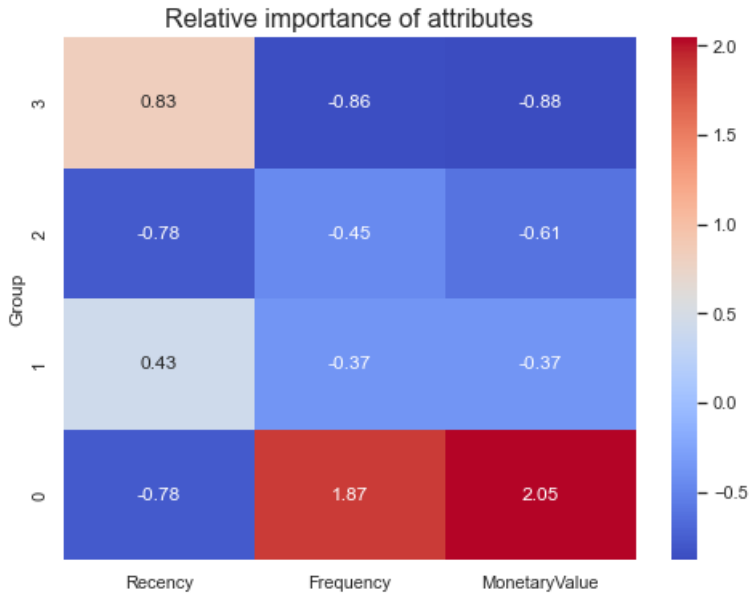
Group	Recency	Frequency	MonetaryValue
0	-0.784648	1.871743	2.048272
1	0.430627	-0.369582	-0.374648
2	-0.784298	-0.447046	-0.607267
3	0.829179	-0.860832	-0.880848

Ejercicio 39

Dibujar un mapa de calor con los datos de `relative_imp`. Utilizar anotaciones con formato `.2f` y el mapa de color `RdYlGn`.

In [91]:
```
plt.figure(figsize=(8, 6))
plt.title('Relative importance of attributes', fontdict={'fontsize': 16})
sns.heatmap(relative_imp, annot=True, fmt='.2f', cmap='coolwarm')
plt.gca().set_ylim(0, 4)
```

Out[91]: (0.0, 4.0)



☑

Ejercicio 40

¿Qué caracteriza, a cada grupo? Proporcionar una descripción de los mismos.

Respuesta:

- Grupo 0: Este grupo tiene una Recency relativamente baja, lo que significa que han realizado una compra recientemente. Además, tienen una alta frecuencia de compra y un alto valor monetario, lo que indica que son clientes leales y valiosos para la empresa.

- Grupo 1: Este grupo tiene una Recency alta, lo que sugiere que han pasado mucho tiempo desde su última compra. También tienen una frecuencia de compra relativamente baja y un bajo valor monetario, lo que indica que no son clientes muy activos o valiosos.

- Grupo 2: Este grupo tiene una Recency baja, similar al Grupo 0, pero su frecuencia de compra y valor monetario son relativamente bajos. Son clientes menos valiosos que los del Grupo 0, pero que pueden llegar a convertirse en futuros Grupo 0 si se los consigue fidelizar.

- Grupo 3: Este grupo tiene la Recency más alta, lo que indica que han pasado mucho tiempo desde su última compra. Además, tienen una frecuencia de compra muy baja y un valor