# TP_IAA_Grupo_2_Entrega_Final

November 29, 2024

## 1 Trabajo Práctico Final - 2024 2C

**Carrera**: Licenciatura en Ciencia de Datos

**Materia**: Introducción al Aprendizaje Automático

**Profesores/as**: * Esteban Roitberg * Luna Schteingart * Homero Lozza * Francisco González Bianco

**Integrantes**: * Federico Menicillo, fedemeni02@gmail.com * Lucas Golchtein, lucas-golchtein@gmail.com * Marcos Achaval, marcos.achavalr@gmail.com

[217]:
```
!jupyter nbconvert --to pdf /content/TP_IAA_Grupo_2_Entrega_Final.ipynb
```

```
[NbConvertApp] Converting notebook /content/TP_IAA_Grupo_2_Entrega_Final.ipynb
to pdf
[NbConvertApp] Writing 119220 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 113320 bytes to /content/TP_IAA_Grupo_2_Entrega_Final.pdf
```

## 2 Indice:



## 3 Descripción del conjunto de datos

Este dataset contiene un historial de llamados telefónicos de campañas de marketing de un banco portugués. Estas campañas están basadas en llamados a los clientes y cada instancia del dataset representa a un cliente con algunas de sus características e información sobre el llamado telefónico. Como variable target, está definida una variable que contiene la información sobre si un cliente

se suscribió o no a un plazo fijo del banco. El período del mismo es desde mayo del 2008 hasta noviembre del 2010.
El dataset fue creado por: *Sérgio Moro (ISCTE-IUL), Paulo Cortez (Univ. Minho) y Paulo Rita (ISCTE-IUL).*

- Cantidad de filas = 45211
- Cantidad de columnas = 17
- Valores faltantes = 0
- Tipos de datos por columna:

1. **age**: numérica discreta (edad del cliente)
2. **job**: categórica nominal (trabajo del cliente)
3. **marital**: categórica nominal (estado civil del cliente)
4. **education**: categórica ordinal (educación del cliente)
5. **default**: categórica nominal (¿pago la deuda?)
6. **balance**: numérica contínua (balance promedio anual del cliente en euros)
7. **housing**: categórica nominal (¿tiene préstamo de vivienda?)
8. **loan**: categórica nominal (¿tiene un préstamo activo?)
9. **contact**: categórica nominal (tipo de llamada: "unknown", "telephone", "cellular")
10. **day**: numérica discreta (último día en el que el cliente fue contactado)
11. **month**: categórica ordinal (último mes en el que el cliente fue contactado)
12. **duration**: numérica contínua (duración del llamado)
13. **campaign**: numérica contínua (cantidad de llamados de la campaña hacia el mismo cliente)
14. **pdays**: numérica contínua (cantidad de días desde el último llamado al cliente de la campañia anterior)
15. **previous**: numérica contínua (cantidad de llamados realizados al cliente previos a la campaña actual)
16. **poutcome**: categórica nominal (resultado de la campaña anterior)
17. **y**: categórica binaria (¿el cliente se suscribió a un plazo fijo?)

[174]:
```
import pandas as pd
import seaborn as sns
import numpy as np
from matplotlib import pyplot as plt
```

[216]:
```
bank = pd.read_csv("bank-full.csv", sep=";")
print(bank.info())
print("\n---------------------------------------\n")
print(bank.describe())
```

```
print("\n---------------------------------------\n")
bank.head(5)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45211 entries, 0 to 45210
Data columns (total 17 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   age        45211 non-null  int64 
 1   job        45211 non-null  object
 2   marital    45211 non-null  object
 3   education  45211 non-null  object
 4   default    45211 non-null  object
 5   balance    45211 non-null  int64 
 6   housing    45211 non-null  object
 7   loan       45211 non-null  object
 8   contact    45211 non-null  object
 9   day        45211 non-null  int64 
 10  month      45211 non-null  object
 11  duration   45211 non-null  int64 
 12  campaign   45211 non-null  int64 
 13  pdays      45211 non-null  int64 
 14  previous   45211 non-null  int64 
 15  poutcome   45211 non-null  object
 16  y          45211 non-null  object
dtypes: int64(7), object(10)
memory usage: 5.9+ MB
None

---------------------------------------

                age        balance           day      duration      campaign  \
count  45211.000000   45211.000000  45211.000000  45211.000000  45211.000000   
mean      40.936210    1362.272058     15.806419    258.163080      2.763841   
std       10.618762    3044.765829      8.322476    257.527812      3.098021   
min       18.000000   -8019.000000      1.000000      0.000000      1.000000   
25%       33.000000      72.000000      8.000000    103.000000      1.000000   
50%       39.000000     448.000000     16.000000    180.000000      2.000000   
75%       48.000000    1428.000000     21.000000    319.000000      3.000000   
max       95.000000  102127.000000     31.000000   4918.000000     63.000000   

              pdays      previous  
count  45211.000000  45211.000000  
mean      40.197828      0.580323  
std      100.128746      2.303441  
min       -1.000000      0.000000  
25%       -1.000000      0.000000  
```

```
50%       -1.000000      0.000000
75%       -1.000000      0.000000
max      871.000000    275.000000

----------------------------------------
```

[216]:
```
   age           job  marital  education default  balance housing loan  \
0   58    management  married   tertiary      no     2143     yes   no
1   44    technician   single  secondary      no       29     yes   no
2   33  entrepreneur  married  secondary      no        2     yes  yes
3   47   blue-collar  married    unknown      no     1506     yes   no
4   33       unknown   single    unknown      no        1      no   no

   contact  day month  duration  campaign  pdays  previous poutcome   y
0  unknown    5   may       261         1     -1         0  unknown  no
1  unknown    5   may       151         1     -1         0  unknown  no
2  unknown    5   may        76         1     -1         0  unknown  no
3  unknown    5   may        92         1     -1         0  unknown  no
4  unknown    5   may       198         1     -1         0  unknown  no
```

Antes que nada, eliminamos la variable `duration`, debido a que dicha variable es un leak de nuestro target.

[176]:
```
bank.drop("duration", axis=1, inplace=True, errors="ignore")
```

## 4 Descripción del problema a resolver

Como objetivo de este trabajo práctico final, nos proponemos predecir si un cliente del banco se va a suscribir o no a un plazo fijo (target `y`). Esta predicción la vamos a llevar a cabo con un modelo de clasificación y este será evaluado con la métrica F1 Score. Decidimos no usar la exactitud, ya que nuestra muestra de datos está muy desbalanceada y sabemos que esta métrica no es precisa en estos casos.

- Dado nuestro target propuesto, queremos encontrar dos o más atributos que tengan alta correlación con respecto al target y baja entre ellos. Para eso realizamos un mapa de calor (*heatmap*) para visualizar la matriz de correlación entre variables numéricas de nuestro dataset.
- Para realizar un *heatmap* es necesario cambiar los valores de las variables categóricas de tipo `string`, como por ejemplo `job`, `marital`, `education`, etc, a valores de tipo `int`. Pero antes, en el atributo `contact`, podemos reemplazar los valores `telephone` por `cellular` ya que este atributo tiene únicamente el 6.5% de sus observaciones con un contacto `telephone`. Haciendo esto, logramos simplificar este atributo y convertirlo en uno binario a costa de perder algo de información (aunque muy poca). Además, los atributos categóricos ordinales, como lo son `education` y `month`, tienen que respetar su orden a la hora de ser reemplazados por números.

[177]:
```
pd.crosstab(bank["contact"], bank["y"])
```

[177]:
```
y             no   yes
contact
cellular   24916  4369
telephone   2516   390
unknown    12490   530
```

[178]:
```
bank["contact"] = bank["contact"].replace("telephone", "cellular")
```

[179]:
```
categorical_cols = bank.select_dtypes(include="object").columns

for col in categorical_cols:
    if col == "education":
        education_map = {"unknown": 0, "primary": 1, "secondary": 2, "tertiary":
 ↪ 3}
        bank[col] = bank[col].map(education_map)
        print(f"{col}: {education_map}")
        continue

    if col == "month":
        months_map = {"jan": 0, "feb": 1, "mar": 2, "apr": 3, "may": 4, "jun":␣
↪5, "jul": 6, "aug": 7, "sep": 8, "oct": 9, "nov": 10, "dec": 11}
        bank[col] = bank[col].map(months_map)
        print(f"{col}: {months_map}")
        continue

    if len(bank[col].unique()) == 2 and bank[col].isin(["yes", "no"]).all():
        binary_map = {"yes": 1, "no": 0}
        bank[col] = bank[col].map(binary_map)
        print(f"{col}: {binary_map}")
        continue

    cols_unique = bank[col].unique()
    key_value_pairs = {valor: i for i, valor in enumerate(cols_unique)}
    print(f"{col}: {key_value_pairs}")
    bank[col] = bank[col].map(key_value_pairs)
```

```
job: {'management': 0, 'technician': 1, 'entrepreneur': 2, 'blue-collar': 3,
'unknown': 4, 'retired': 5, 'admin.': 6, 'services': 7, 'self-employed': 8,
'unemployed': 9, 'housemaid': 10, 'student': 11}
marital: {'married': 0, 'single': 1, 'divorced': 2}
education: {'unknown': 0, 'primary': 1, 'secondary': 2, 'tertiary': 3}
default: {'yes': 1, 'no': 0}
housing: {'yes': 1, 'no': 0}
loan: {'yes': 1, 'no': 0}
contact: {'unknown': 0, 'cellular': 1}
month: {'jan': 0, 'feb': 1, 'mar': 2, 'apr': 3, 'may': 4, 'jun': 5, 'jul': 6,
'aug': 7, 'sep': 8, 'oct': 9, 'nov': 10, 'dec': 11}
```

```
poutcome: {'unknown': 0, 'failure': 1, 'other': 2, 'success': 3}
y: {'yes': 1, 'no': 0}
```

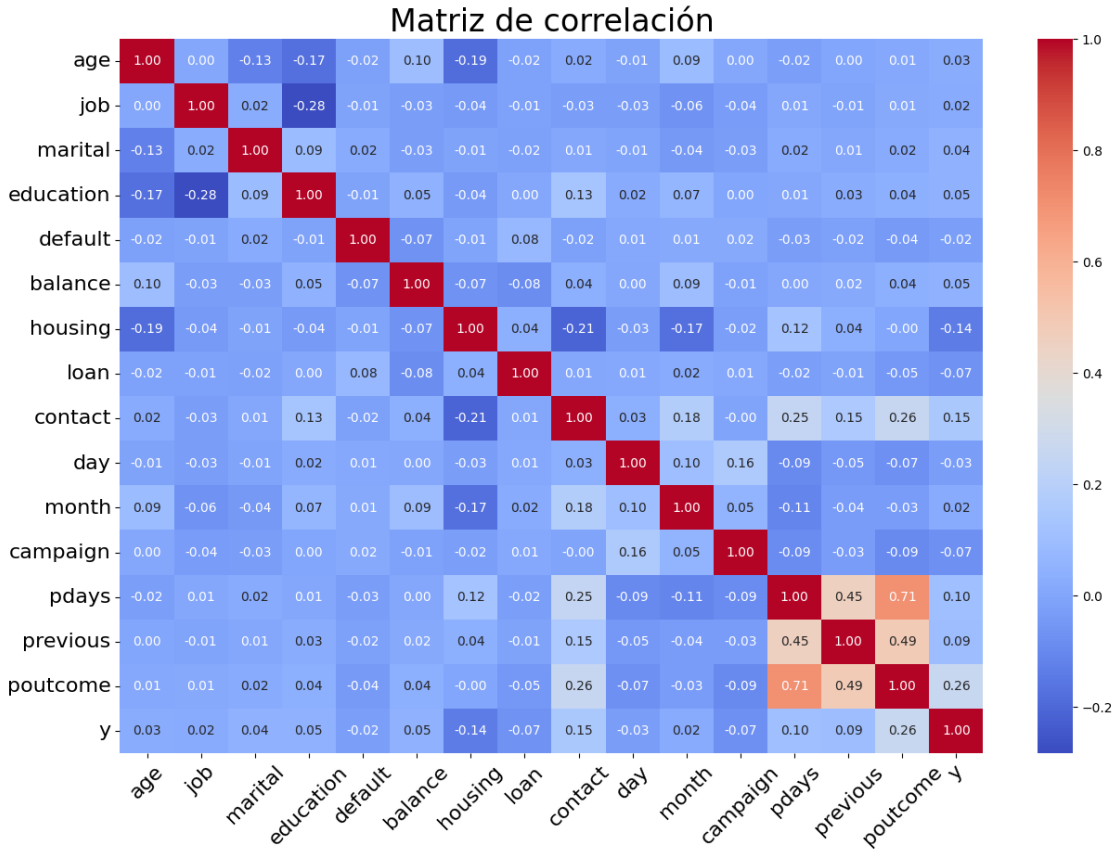
[180]:
```
bank.head(5)
```

[180]:

| | age | job | marital | education | default | balance | housing | loan | contact | \ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 58 | 0 | 0 | 3 | 0 | 2143 | 1 | 0 | 0 | |
| 1 | 44 | 1 | 1 | 2 | 0 | 29 | 1 | 0 | 0 | |
| 2 | 33 | 2 | 0 | 2 | 0 | 2 | 1 | 1 | 0 | |
| 3 | 47 | 3 | 0 | 0 | 0 | 1506 | 1 | 0 | 0 | |
| 4 | 33 | 4 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | |

| | day | month | campaign | pdays | previous | poutcome | y |
|---|---|---|---|---|---|---|---|
| 0 | 5 | 4 | 1 | -1 | 0 | 0 | 0 |
| 1 | 5 | 4 | 1 | -1 | 0 | 0 | 0 |
| 2 | 5 | 4 | 1 | -1 | 0 | 0 | 0 |
| 3 | 5 | 4 | 1 | -1 | 0 | 0 | 0 |
| 4 | 5 | 4 | 1 | -1 | 0 | 0 | 0 |

[181]:
```
data = bank.select_dtypes(include=np.number)
plt.figure(figsize=(15, 10))
sns.heatmap(data.corr(), annot= True, cmap='coolwarm', fmt=".2f")
plt.title("Matriz de correlación", fontsize=24)
plt.xticks(rotation=45, fontsize=16)
plt.yticks(fontsize=16)
plt.show()
```




Luego de hacer el mapa de calor, podemos deducir que las 3 variables que tienen mayor correlación con la variable target `y` son `poutcome`, `contact` y `housing`.

En el siguiente gráfico mostramos la relación entre `contact` y `poutcome` mediante un gráfico de dispersión coloreado por la variable target y aplicándole un ruido a las variables para una mejor visualización.

[182]:
```
pd.DataFrame(bank["poutcome"].value_counts())
```

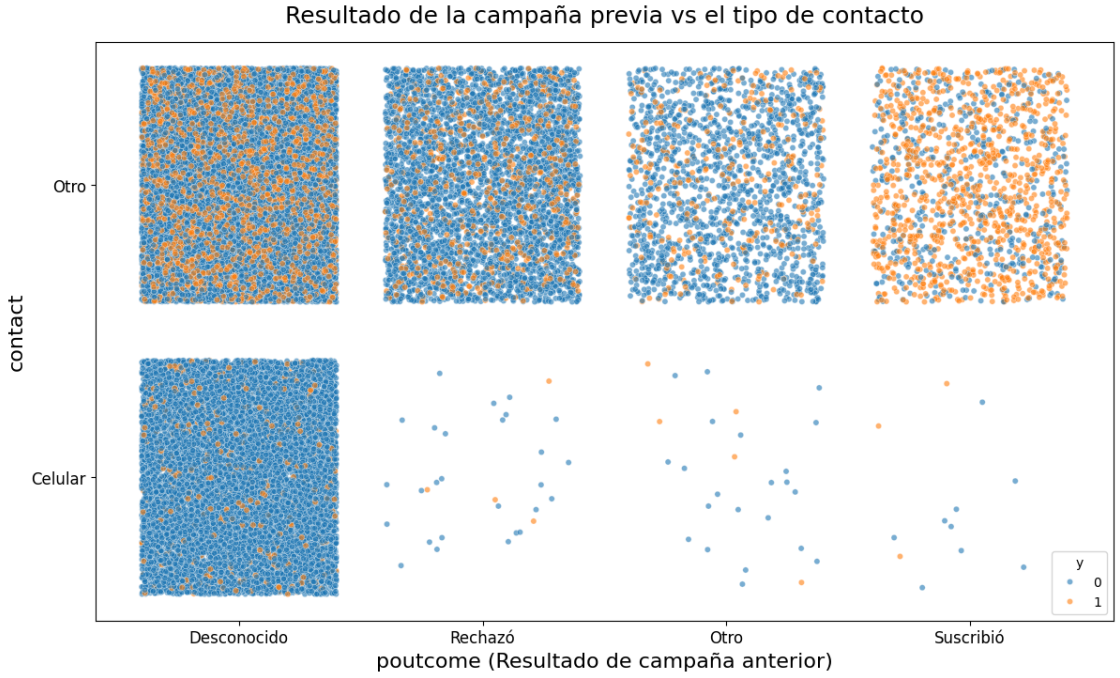
[182]:

| poutcome | count |
|---|---|
| 0 | 36959 |
| 1 | 4901 |
| 2 | 1840 |
| 3 | 1511 |

[183]:
```
bank["poutcome"] = bank["poutcome"].replace({0: 'Desconocido'})
bank["poutcome"] = bank["poutcome"].replace({1: 'Rechazó'})
bank["poutcome"] = bank["poutcome"].replace({2: 'Otro'})
bank["poutcome"] = bank["poutcome"].replace({3: 'Suscribió'})
bank["contact"] = bank["contact"].replace({0: 'Celular'})
```

```
bank["contact"] = bank["contact"].replace({1: 'Otro'})
bank["poutcome"] = pd.Categorical(bank["poutcome"], categories=["Desconocido",␣
 ↪"Rechazó", "Otro", "Suscribió"])
bank["contact"] = pd.Categorical(bank["contact"], categories=["Celular",␣
 ↪"Otro"])
```

[184]:
```
bank["jpoutcome"] = bank["poutcome"].cat.codes + np.random.uniform(-0.4, 0.4,␣
 ↪len(bank))
bank["jcontact"] = bank["contact"].cat.codes + np.random.uniform(-0.4, 0.4,␣
 ↪len(bank))


plt.figure(figsize=(14, 8))
sns.scatterplot(bank, x="jpoutcome", y="jcontact", s=20, alpha=0.6, marker="o",␣
 ↪hue="y")

plt.xticks(range(len(bank["poutcome"].cat.categories)), bank["poutcome"].cat.
 ↪categories, fontsize=12)
plt.yticks(range(len(bank["contact"].cat.categories)), bank["contact"].cat.
 ↪categories, fontsize=12)

plt.title("Resultado de la campaña previa vs el tipo de contacto", fontsize=18,␣
 ↪y=1.02)
plt.xlabel("poutcome (Resultado de campaña anterior)", fontsize=16)
plt.ylabel("contact", fontsize=16)
plt.show()
```



[185]:
```
pd.DataFrame(bank[bank["poutcome"] == "Suscribió"].value_counts(["contact",␣
 ↪"y"])).sort_values("contact")
```

[185]:

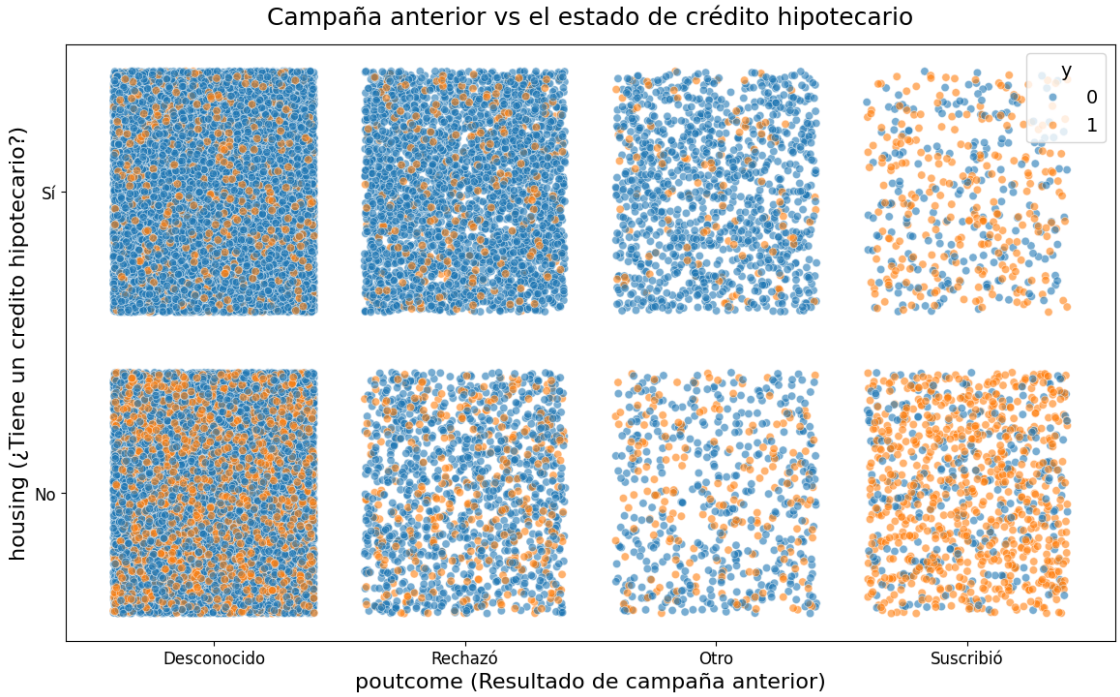
| | | count |
|---|---|---|
| contact | y | |
| Celular | 0 | 9 |
| | 1 | 3 |
| Otro | 1 | 975 |
| | 0 | 524 |

En el gráfico de dispersión anterior podemos notar que los clientes que se suscribieron a un plazo fijo en la campaña anterior, la mayoría de ellos se suelen suscribir nuevamente en la campaña actual. También se puede ver en la tabla de arriba que muestra la diferencia entre la clase positiva y negativa agrupada por tipo de contacto. En este caso no podemos destacar ningún patrón interesante con el atributo `contact`. Únicamente, vemos que las instancias que no se sabe su tipo de contacto son la minoría.

Además, generamos otro gráfico de dispersión para ver la relación entre la variable `housing` y `poutcome` coloreado por `y`. Al igual que en el gráfico anterior, se les agregó ruido a las variables para visualizarlas mejor.

[186]:
```
bank["housing"] = bank["housing"].replace({0: 'No'})
bank["housing"] = bank["housing"].replace({1: 'Sí'})
bank["housing"] = pd.Categorical(bank["housing"], categories=["No", "Sí"])
```

[187]:
```
plt.figure(figsize=(14,8))

bank["jhousing"] = bank["housing"].cat.codes + np.random.uniform(-0.4, 0.4,␣
 ↪len(bank))

sns.scatterplot(bank, x = "jpoutcome", y = "jhousing", hue = "y", alpha=0.6)
plt.title("Campaña anterior vs el estado de crédito hipotecario", fontsize=18,␣
 ↪y=1.02)
plt.xlabel("poutcome (Resultado de campaña anterior)", fontsize=16)
plt.ylabel("housing (¿Tiene un credito hipotecario?)", fontsize=16)
plt.xticks(range(len(bank["poutcome"].cat.categories)), bank["poutcome"].cat.
 ↪categories, fontsize=12)
plt.yticks(range(len(bank["housing"].cat.categories)), bank["housing"].cat.
 ↪categories, fontsize=12)
plt.legend(title="y", title_fontsize=14, fontsize=14)
plt.show()
```




[188]:
```
pd.DataFrame(bank[bank["poutcome"] == "Suscribió"].value_counts(["housing",␣
↪"y"])).sort_values("housing"), pd.DataFrame(bank[bank["poutcome"] ==␣
↪"Rechazó"].value_counts(["housing", "y"])).sort_values("housing")
```

[188]:
```
(           count
housing y
No      1     729
        0     311
Sí      1     249
        0     222,
           count
housing y
No      0   1101
        1    330
Sí      0   3182
        1    288)
```

De los que se suscribieron a un plazo fijo en la campaña anterior, si miramos únicamente a los que no tienen un crédito hipotecario, más del 50% de ellos se vuelve a suscribir nuevamente a un plazo fijo en la campaña actual. Mientras que en los que tienen un crédito hipotecario no hay mucha diferencia en el atributo y, aproximadamente de todos ellos un 50% se suscribe nuevamente y el otro 50% lo contrario. En el caso de los que no se suscribieron en la campaña anterior, en las dos categorías de housing predominan los que no se suscribieron a un plazo fijo. En otras palabras, indiferentemente de si el cliente tiene un crédito hipotecario o no, la mayoría de esos clientes no se suscribe a un plazo fijo en la actual campaña (mirar tablas de arriba).

[189]:
```
bank["poutcome"] = bank["poutcome"].cat.rename_categories({'Desconocido': 0,␣
↪'Rechazó': 1,'Otro':2,'Suscribió':3})
bank["contact"] = bank["contact"].cat.rename_categories({'Celular':0,'Otro':1})
bank["housing"] = bank["housing"].cat.rename_categories({'No':0,'Sí':1})
```

[190]:
```
bank.drop(["jpoutcome","jcontact","jhousing"], axis=1, inplace=True,␣
↪errors="ignore")
```

### 4.0.1 Balance de la muestra

Veamos la proporción de cada clase en la variable target `y`.

[191]:
```
proporciones = {
    "Target": bank["y"].value_counts().index,
    "Cantidad": bank["y"].value_counts().values,
    "%": (bank["y"].value_counts(normalize=True) * 100).round(0).astype(int).
↪values
}

pd.DataFrame(proporciones).reset_index(drop=True)
```

[191]:

| | Target | Cantidad | % |
|---|---|---|---|
| 0 | 0 | 39922 | 88 |
| 1 | 1 | 5289 | 12 |

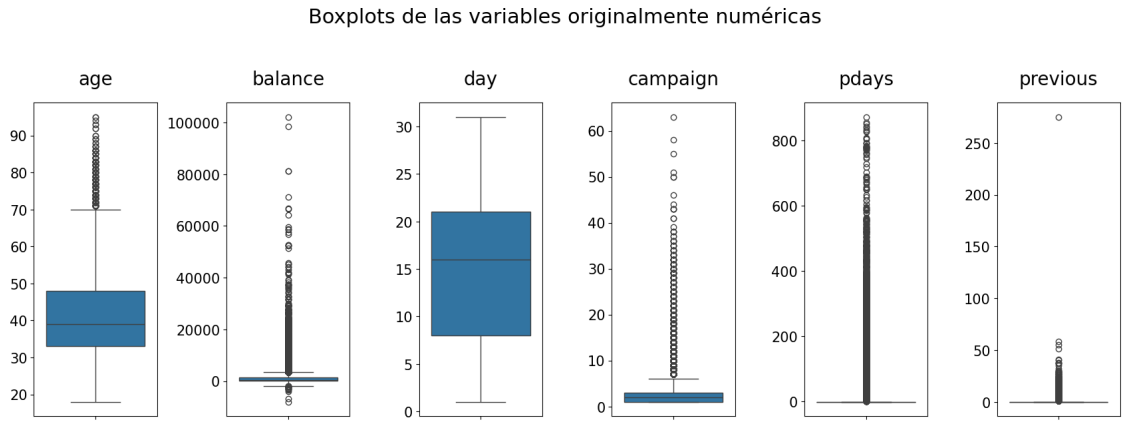
Podemos notar que la muestra está considerablemente desbalanceada. Aproximadamente 90 % de los datos de la muestra pertenecen a la clase *negativa*, mientras que el 10 % restante pertenece a la clase *positiva*. Esto es importante tenerlo en cuenta para el resto del trabajo práctico, ya que es información clave para la evaluación de los modelos y para el ajuste de sus diferentes hiperparámetros.

## 5 Preprocesamiento de Datos

Analicemos si hay valores atípicos ("*outliers*") en las variables que originalmente eran continuas de tipo `int`.

[192]:
```
fig, axs = plt.subplots(1, 6, figsize=(17, 6))

for i, col in enumerate(bank[["age", "balance", "day", "campaign", "pdays",␣
↪"previous"]]):
    sns.boxplot(data=bank[["age", "balance", "day", "campaign", "pdays",␣
↪"previous"]], y=col, ax=axs[i])
    axs[i].set_title(f"{col}", fontsize=20, y=1.04)
    axs[i].set_xlabel("")
    axs[i].set_ylabel("")
    axs[i].tick_params(axis="y", labelsize=15)
```

```
plt.suptitle("Boxplots de las variables originalmente numéricas", fontsize =␣
↪22, y=1.06)
plt.tight_layout()
plt.show()
```




Podemos observar que el atributo **previous** tiene un valor que se aleja significativamente de la distribución central (*outlier univariante*). Consideramos tan grande ese alejamiento que creemos que no hace falta analizar a ese dato en particular. Entonces directamente decidimos eliminar esa instancia completa.

[193]:
```
bank.drop(bank[bank["previous"] == 275].index, inplace=True)
bank.nlargest(10, "previous")
```

[193]:

| | age | job | marital | education | default | balance | housing | loan | contact | \ |
|---|---|---|---|---|---|---|---|---|---|---|
| 38326 | 46 | 3 | 0 | 1 | 0 | 1085 | 1 | 1 | 1 | |
| 44089 | 37 | 1 | 0 | 2 | 0 | 432 | 1 | 0 | 1 | |
| 28886 | 31 | 0 | 1 | 3 | 0 | 358 | 1 | 0 | 1 | |
| 44822 | 27 | 3 | 0 | 2 | 0 | 821 | 1 | 1 | 0 | |
| 42611 | 35 | 1 | 1 | 2 | 0 | 4645 | 1 | 0 | 1 | |
| 28498 | 49 | 0 | 1 | 3 | 0 | 145 | 1 | 0 | 1 | |
| 37567 | 39 | 0 | 0 | 3 | 0 | 0 | 1 | 0 | 1 | |
| 26668 | 51 | 2 | 0 | 2 | 0 | 653 | 1 | 0 | 1 | |
| 42422 | 27 | 11 | 1 | 2 | 0 | 91 | 0 | 0 | 1 | |
| 44484 | 28 | 0 | 1 | 3 | 0 | 6791 | 0 | 0 | 1 | |

| | day | month | campaign | pdays | previous | poutcome | y |
|---|---|---|---|---|---|---|---|
| 38326 | 15 | 4 | 2 | 353 | 58 | 2 | 1 |
| 44089 | 6 | 6 | 3 | 776 | 55 | 1 | 1 |
| 28886 | 30 | 0 | 3 | 256 | 51 | 1 | 0 |
| 44822 | 16 | 8 | 1 | 778 | 41 | 2 | 0 |
| 42611 | 11 | 0 | 3 | 270 | 40 | 2 | 0 |
| 28498 | 29 | 0 | 2 | 248 | 38 | 1 | 0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 37567 | 14 | 4 | 15 | 261 | 38 | 1 | 0 |
| 26668 | 20 | 10 | 9 | 112 | 37 | 2 | 0 |
| 42422 | 4 | 11 | 6 | 95 | 37 | 2 | 0 |
| 44484 | 9 | 7 | 1 | 46 | 35 | 1 | 0 |

El atributo `balance` tiene dos instancias candidatas a ser valores atípicos, pero al no ser tan significativo el alejamiento, necesitamos hacer un análisis específico a la fila completa de esos dos valores. Esto nos va a permitir observar otras características del cliente de esas instancias y determinar si tiene un balance atípico o no.

[194]:
```
bank.nlargest(10, "balance")
```

[194]:

| | age | job | marital | education | default | balance | housing | loan | contact | \ |
|---|---|---|---|---|---|---|---|---|---|---|
| 39989 | 51 | 0 | 1 | 3 | 0 | 102127 | 0 | 0 | 1 | |
| 26227 | 59 | 0 | 0 | 3 | 0 | 98417 | 0 | 0 | 1 | |
| 42558 | 84 | 5 | 0 | 2 | 0 | 81204 | 0 | 0 | 1 | |
| 43393 | 84 | 5 | 0 | 2 | 0 | 81204 | 0 | 0 | 1 | |
| 41693 | 60 | 5 | 0 | 1 | 0 | 71188 | 0 | 0 | 1 | |
| 19785 | 56 | 0 | 2 | 3 | 0 | 66721 | 0 | 0 | 1 | |
| 21192 | 52 | 3 | 0 | 1 | 0 | 66653 | 0 | 0 | 1 | |
| 19420 | 59 | 6 | 0 | 0 | 0 | 64343 | 0 | 0 | 1 | |
| 41374 | 32 | 2 | 1 | 3 | 0 | 59649 | 0 | 0 | 1 | |
| 12926 | 56 | 3 | 0 | 2 | 0 | 58932 | 0 | 0 | 1 | |

| | day | month | campaign | pdays | previous | poutcome | y |
|---|---|---|---|---|---|---|---|
| 39989 | 3 | 5 | 1 | -1 | 0 | 0 | 0 |
| 26227 | 20 | 10 | 5 | -1 | 0 | 0 | 0 |
| 42558 | 28 | 11 | 1 | 313 | 2 | 2 | 1 |
| 43393 | 1 | 3 | 1 | 94 | 3 | 3 | 1 |
| 41693 | 6 | 9 | 1 | -1 | 0 | 0 | 0 |
| 19785 | 8 | 7 | 2 | -1 | 0 | 0 | 0 |
| 21192 | 14 | 7 | 3 | -1 | 0 | 0 | 0 |
| 19420 | 6 | 7 | 4 | -1 | 0 | 0 | 0 |
| 41374 | 1 | 8 | 2 | -1 | 0 | 0 | 0 |
| 12926 | 7 | 6 | 2 | -1 | 0 | 0 | 0 |

Decidimos **no** clasificarlos como valores atípicos, ya que por sus edades, tipo de trabajo (`management`) y educación (título terciario) es posible que obtengan ese patrimonio.

Por último, nos fijamos si los atributos tienen valores negativos o mínimos que sean atípicos o ilógicos. El único atributo que nos llamó la atención es `balance`, ya que es raro que un cliente tenga un balance negativo. Sin embargo, puede ocurrir cuando el cliente está endeudado con el banco. El valor negativo en el atributo `pdays` es correcto, ya que ese valor significa que el cliente nunca fue contactado anteriormente, un hecho tranquilamente posible.

[195]:
```
pd.DataFrame(bank[['age', 'balance', 'day', 'campaign', 'pdays', 'previous']].
  ↪min())
```

[195]:
```
                0
age            18
balance     -8019
day             1
campaign        1
pdays          -1
previous        0
```

# 6 Modelado

Esta etapa la decidimos llevar a cabo con un **árbol de decisión**. Decidimos utilizar este tipo de modelo, ya que, a diferencia de la **regresión logística**, los árboles de decisión son más robustos ante muestras desbalanceadas y con atributos que no están linealmente relacionados. Para muestras desbalanceadas, DecisionTreeClassifier incluye un parámetro `class_weight` que le asigna un mayor peso (importancia) a la clase minoritaria. En nuestro caso particular, se le asigna más peso a la clase positiva. Además, este tipo de modelo ofrece una función `feature_importance` que determina el poder predictivo de cada atributo y esto ayuda mucho a la hora de decidir qué atributos usar en el modelo y a seleccionar únicamente a los más predictores para evitar un sobre ajuste. Cabe destacar que para evaluar cada modelo que hagamos durante toda esta etapa, nuestra clase de interés va a ser la clase *positiva* (la minoritaria). Vamos a concentrarnos en mejorar nuestros modelos a base de las predicciones de la clase de interés y su rendimiento respecto del **F1 Score** para obtener un modelo robusto para predecir si un cliente se va a suscribir o no a un plazo fijo. Hemos elegido basarnos en esta métrica, ya que combina la `exhaustividad` (**recall**) y la `presicion`, dos métricas apropiadas para la evaluación de nuestros modelos. Además, sabiendo que nuestra muestra está altamente desbalanceada, el `F1-Score` es una buena opción para estos casos.

[196]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn import tree
from sklearn.model_selection import train_test_split, cross_validate,␣
 ↪GridSearchCV, RandomizedSearchCV
from sklearn.metrics import f1_score
from sklearn.dummy import DummyClassifier
from sklearn.feature_selection import RFE
```

**Train-Test Split**

[197]:
```
X = bank.drop("y", axis=1)
y = bank["y"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42, stratify=y)
```

**Benchmark** Comenzamos con un modelo que asigna valores al azar respetando la proporción original de la muestra. Por ejemplo, nuestra muestra que contiene el 90 % de los datos como clase **negativa** y el 10 % restante como clase **positiva**, el asigna valores de clase (etiquetas) al azar y luego, la proporción de clases de la predicción va a ser similar a la original (90/10).

[198]:
```
dummy_clf = DummyClassifier(strategy="stratified")
dummy_clf.fit(X_train, y_train)

y_test_pred_bench = dummy_clf.predict(X_test)
f1_score_bench = f1_score(y_test, y_test_pred_bench)

print(f"F1 Score = {round(f1_score_bench,2)}")
```

```
F1 Score = 0.12
```

Podemos ver que el desempeño del benchmark es bajo, algo totalmente esperado. El resto de los modelos van a ser evaluados y comparados respecto de este.

**Primer Modelo** Como siguiente paso, nos proponemos realizar un primer modelo simple con un **árbol de decisión** de profundidad 2. Elegimos las 2 variables que tienen más correlación con el , pero cero correlación entre ellas. Las variables son: `poutcome` y `housing`

[199]:
```
clf = DecisionTreeClassifier(max_depth=2,random_state=42,␣
↪class_weight="balanced")

cv = cross_validate(clf, X_train[["poutcome", "housing"]], y_train, cv=5,␣
↪scoring="f1", return_train_score=True)

print("F1-Score (train):", {round(cv["train_score"].mean(), 2)}, "±", {round(np.
↪std(cv["train_score"]), 2)})
print("F1-Score (test):", {round(cv["test_score"].mean(), 2)}, "±", {round(np.
↪std(cv["test_score"]), 2)})
```

```
F1-Score (train): {0.28} ± {0.0}
F1-Score (test): {0.28} ± {0.0}
```

Podemos observar una mejora significativa en el `F1-Score` con respecto del benchmark. Además, como este modelo es simple, mirando la igualdad de rendimiento en entrenamiento y evaluación de la validación cruzada, podemos afirmar que este modelo está sub ajustando y tiene un alto sesgo.

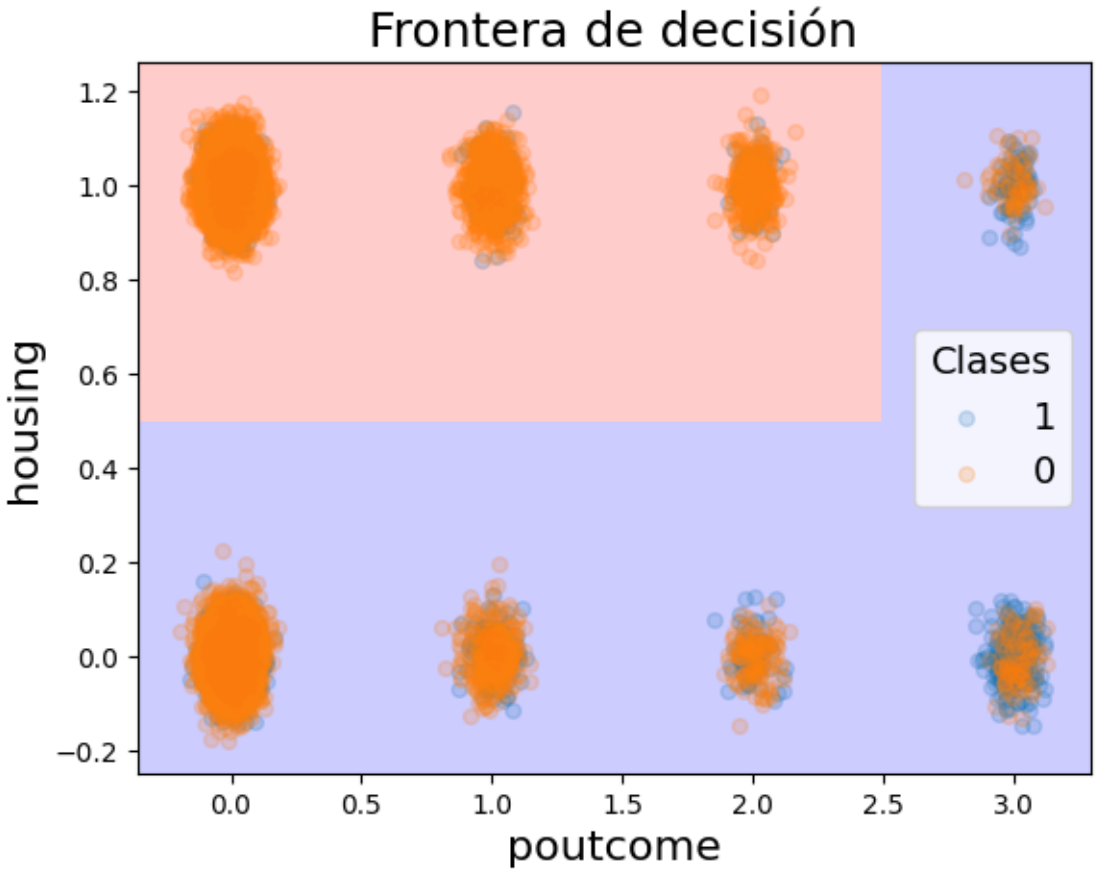
A continuación presentamos una gráfica de la frontera de decisión de este árbol. Sin embargo, como se trata de dos atributos categóricos, el gráfico no brinda una clara interpretación.

[200]:
```
def visualize_classifier(model, X, y, ax=None, proba=False, jitter_amount=0.05):
    if isinstance(X, pd.DataFrame):
        X = X.values
    if isinstance(y, pd.Series):
        y = y.values
    ax = ax or plt.gca()

    colors_tab10 = plt.cm.tab10.colors

    np.random.seed(42)
    X_jittered = X + jitter_amount * np.random.normal(size=X.shape)
```

```
    unique_classes = np.unique(y)
    for i, y_value in enumerate(reversed(unique_classes)):
        ax.scatter(X_jittered[y == y_value, 0], X_jittered[y == y_value, 1],␣
↪s=30,
                   zorder=3, alpha=0.2, color=colors_tab10[i], label=y_value)

    ax.legend(title="Clases", loc="best", fontsize=14, title_fontsize=14)
    ax.axis("tight")
    plt.title("Frontera de decisión", fontsize=18)
    ax.set_xlabel("poutcome", fontsize=16)
    ax.set_ylabel("housing", fontsize=16)

    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                         np.linspace(*ylim, num=200))

    if proba:
        Z = model.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1].reshape(xx.
↪shape)
    else:
        Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    Z = -Z + 1
    ax.pcolormesh(xx, yy, Z, cmap="bwr", vmin=0, vmax=1, alpha=0.2)

    ax.set(xlim=xlim, ylim=ylim)
```

[201]:
```
visualize_classifier(clf.fit(X_train[["poutcome", "housing"]], y_train),␣
↪X_test[["poutcome", "housing"]], y_test)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:493: UserWarning: X does
not have valid feature names, but DecisionTreeClassifier was fitted with feature
names
  warnings.warn(
```




**Segundo modelo** Como segundo modelo, decidimos entrenar otro **árbol de decisión,** pero en este caso lo entrenamos con los 5 atributos más importantes usando **Recursive Feature Elimination (RFE)**. Esta técnica entrena múltiples árboles de decisión y en cada iteración elimina el atributo menos importante (mediante el nivel de reducción de impurezas del árbol) hasta llegar a la cantidad de atributos deseada. Esto nos va a ayudar a reducir la complejidad del modelo, excluyendo información que no aporta a las predicciones y manteniendo solo la más valiosa y con alto poder predictivo.

[202]:
```
clf2 = DecisionTreeClassifier(random_state=42, class_weight="balanced")
rfe = RFE(estimator=clf2, n_features_to_select=5)
rfe.fit(X_train, y_train)

feature_names = X_train.columns
selected_features = feature_names[rfe.support_]
pd.DataFrame({"Atributo": selected_features})
```

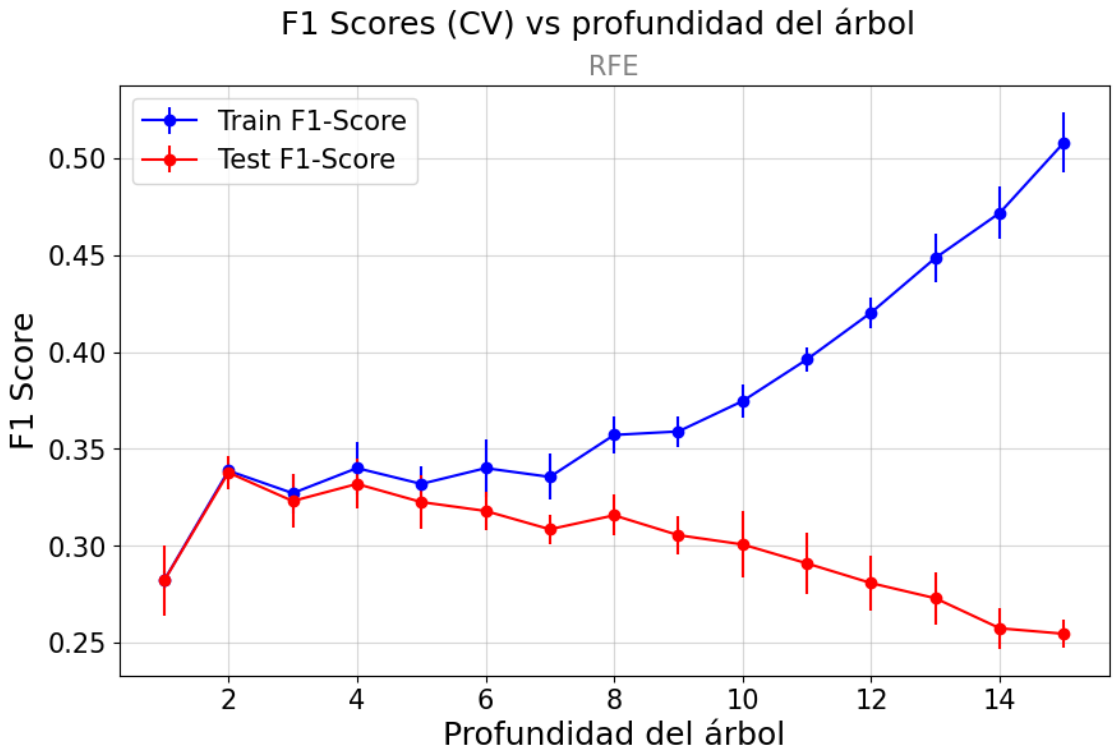
[202]:

| | Atributo |
|---|---|
| 0 | age |
| 1 | job |

```
2   balance
3       day
4  poutcome
```

Luego queremos encontrar la profundidad óptima del **árbol de decisión** evaluando el `F1 Score` con **Validación Cruzada** tanto en su conjunto de entrenamiento como en el de validación, a medida que aumenta la complejidad del modelo (profundidad). Vamos a prestar atención a las zonas de alto sesgo y varianza y determinar la mejor profundidad en donde el modelo presenta un equilibro entre estos dos factores.

[203]:
```
max_depths = range(1, 16)
f1_scores_train = []
std_error_train = []
f1_scores_test = []
std_error_test = []

for depth in max_depths:
    clf = DecisionTreeClassifier(max_depth=depth, random_state=42,␣
 ↪class_weight="balanced")

    cv_results = cross_validate(clf, X_train[selected_features], y_train, cv=5,␣
 ↪scoring="f1", return_train_score=True)

    f1_scores_train.append(np.mean(cv_results["train_score"]))
    f1_scores_test.append(np.mean(cv_results["test_score"]))

    std_error_train.append(np.std(cv_results["train_score"]))
    std_error_test.append(np.std(cv_results["test_score"]))

plt.figure(figsize=(10, 6))
plt.errorbar(max_depths, f1_scores_train, yerr=std_error_train, fmt="o-",␣
 ↪color="b", label="Train F1-Score")
plt.errorbar(max_depths, f1_scores_test, yerr=std_error_test, fmt="o-",␣
 ↪color="r", label="Test F1-Score")

plt.suptitle("F1 Scores (CV) vs profundidad del árbol", fontsize=18)
plt.title("RFE", fontsize=15, color="grey")
plt.xlabel("Profundidad del árbol", fontsize=18)
plt.ylabel("F1 Score", fontsize=18)
plt.legend(fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.grid(True, alpha=0.5)
plt.show()
```

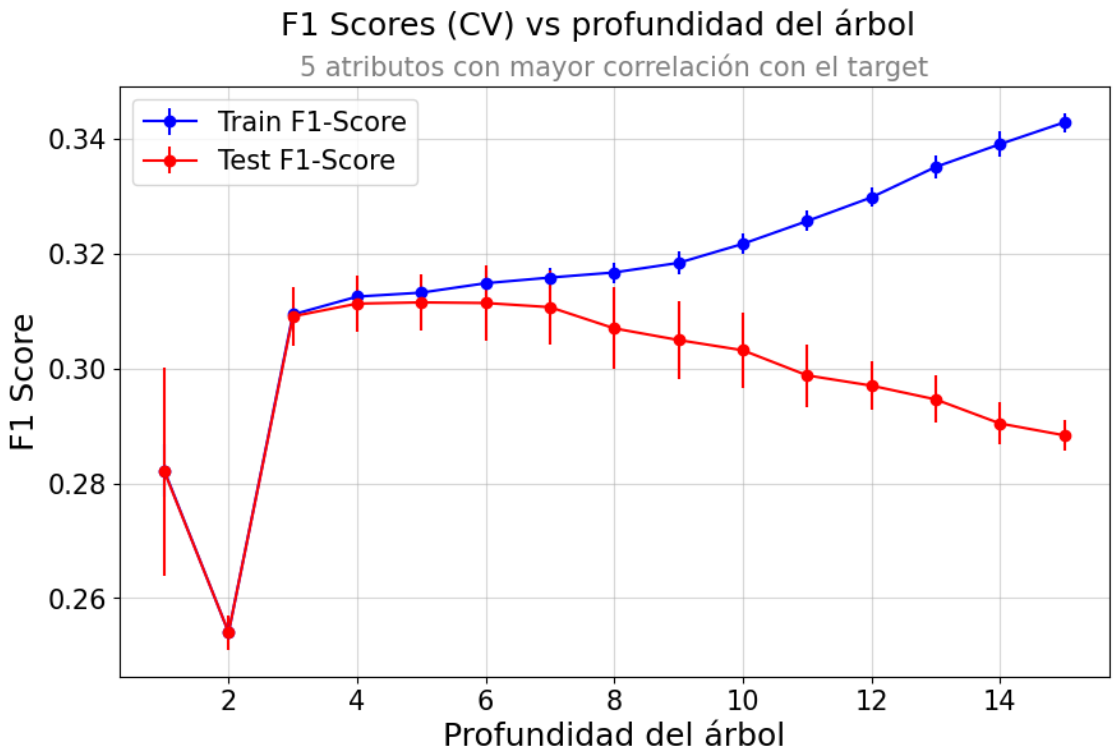



Consideramos que la mejor profundidad es 3, ya que es la que obtiene un F1 Score más alto en validación y un similar rendimiento en entrenamiento. A partir de profundidad 8 vemos que el rendimiento de entrenamiento empieza a aumentar, mientras que el de validación empieza a disminuir. Por lo tanto, en este punto el árbol empieza a sobre ajustar y a aumentar su varianza. Es muy probable que si entrenamos este árbol con profundidad 12 (por ejemplo), su frontera de decisión va a cambiar mucho a medida que se entrena con diferentes conjuntos de datos (submuestras).

Luego queremos analizar cuánto cambia el rendimiento del modelo si usamos los 5 atributos que tienen mayor correlación con el target sin importar la correlación entre ellos. Veamos qué sucede.

[204]:
```
max_depths = range(1, 16)
f1_scores_train = []
std_error_train = []
f1_scores_test = []
std_error_test = []

for depth in max_depths:
    clf = DecisionTreeClassifier(max_depth=depth, random_state=42,␣
 ↪class_weight="balanced")

    cv_results = cross_validate(clf, X_train[["poutcome", "contact", "housing",␣
 ↪"pdays", "previous"]], y_train, cv=5, scoring="f1", return_train_score=True)
```

```
    f1_scores_train.append(np.mean(cv_results["train_score"]))
    f1_scores_test.append(np.mean(cv_results["test_score"]))

    std_error_train.append(np.std(cv_results["train_score"]))
    std_error_test.append(np.std(cv_results["test_score"]))

plt.figure(figsize=(10, 6))
plt.errorbar(max_depths, f1_scores_train, yerr=std_error_train, fmt="o-",␣
 ↪color="b", label="Train F1-Score")
plt.errorbar(max_depths, f1_scores_test, yerr=std_error_test, fmt="o-",␣
 ↪color="r", label="Test F1-Score")

plt.suptitle("F1 Scores (CV) vs profundidad del árbol", fontsize=18)
plt.title("5 atributos con mayor correlación con el target", fontsize=15,␣
 ↪color="grey")
plt.xlabel("Profundidad del árbol", fontsize=18)
plt.ylabel("F1 Score", fontsize=18)
plt.legend(fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.grid(True, alpha=0.5)
plt.show()
```



Observamos que los rendimientos obtenidos con este modelo son más bajos que los del modelo anterior. Analizando las correlaciones entre las variables utilizadas en este modelo y en el anterior, notamos que los atributos elegidos con RFE (feature importance) tienen correlaciones entre ellos de entre 0 y 0.1 (poca correlación). Mientras que los atributos que tienen mayor correlación con el target, presentan correlaciones entre ellos de entre 0 y 0.71 con la mayoría por encima de 0.2. Creemos que los atributos que menos se correlacionan entre sí, son mejores predictores porque le brindan más variedad de información al modelo. En otras palabras, si dos atributos están altamente correlacionados, brindan casi la misma información.

*Estos resultados se pueden visualizar en la matriz de correlación del comienzo.*

A continuación trataremos de encontrar mejores **hiperparámetros** (inclusive la profundidad) para optimizar el modelo que utiliza los 5 atributos más importantes y obtener un mejor desempeño. Primero vamos a llevar a cabo un Randomized Search con validación cruzada para obtener un rango de valores acotado de cada hiperparámetro y luego evaluarlos con Grid Search.

[205]:
```
params_tree_rand = {
    "max_depth": [1,2,3,4,5,6,7,8,9,10],
    "min_samples_split": [1,2,3,4,5],
    "min_samples_leaf": [1,2,3,4,5],
    "max_features": ["auto", "sqrt", "log2", None],
    "splitter": ["best", "random"],
    "max_leaf_nodes": [None,10,20],
    "min_weight_fraction_leaf": [0.0,0.01,0.05,0.1],
    "min_impurity_decrease": [0.0,0.001,0.01,0.1]
}

random_search_tree = RandomizedSearchCV(clf2, params_tree_rand, n_iter=1000,␣
 ↪cv=5, random_state=42, scoring="f1", return_train_score=True)
random_search_tree.fit(X_train[selected_features], y_train)

print(f"Mejores parámetros: {random_search_tree.best_params_}")

max_index_train = np.nanargmax(random_search_tree.
 ↪cv_results_["mean_train_score"])
print("Mejor F1 Score (train):", {round(random_search_tree.
 ↪cv_results_["mean_train_score"][max_index_train], 3)}, "±",␣
 ↪{round(random_search_tree.cv_results_["std_train_score"][max_index_train],␣
 ↪3)})

max_index_test = np.nanargmax(random_search_tree.cv_results_["mean_test_score"])
print("Mejor F1 Score (test):", {round(random_search_tree.
 ↪cv_results_["mean_test_score"][max_index_test], 3)}, "±",␣
 ↪{round(random_search_tree.cv_results_["std_test_score"][max_index_test], 3)})
```

```
Mejores parámetros: {'splitter': 'best', 'min_weight_fraction_leaf': 0.0,
'min_samples_split': 3, 'min_samples_leaf': 5, 'min_impurity_decrease': 0.001,
'max_leaf_nodes': None, 'max_features': 'log2', 'max_depth': 3}
Mejor F1 Score (train): {0.354} ± {0.005}
```

```
Mejor F1 Score (test): {0.354} ± {0.02}

/usr/local/lib/python3.10/dist-
packages/sklearn/model_selection/_validation.py:540: FitFailedWarning:
2070 fits failed out of a total of 5000.
The score on these train-test partitions for these parameters will be set to
nan.
If these failures are not expected, you can try to debug them by setting
error_score='raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
830 fits failed with the following error:
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-
packages/sklearn/model_selection/_validation.py", line 888, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 1466, in
wrapper
    estimator._validate_params()
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 666, in
_validate_params
    validate_parameter_constraints(
  File "/usr/local/lib/python3.10/dist-
packages/sklearn/utils/_param_validation.py", line 95, in
validate_parameter_constraints
    raise InvalidParameterError(
sklearn.utils._param_validation.InvalidParameterError: The 'min_samples_split'
parameter of DecisionTreeClassifier must be an int in the range [2, inf) or a
float in the range (0.0, 1.0]. Got 1 instead.

--------------------------------------------------------------------------------
1240 fits failed with the following error:
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-
packages/sklearn/model_selection/_validation.py", line 888, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 1466, in
wrapper
    estimator._validate_params()
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 666, in
_validate_params
    validate_parameter_constraints(
  File "/usr/local/lib/python3.10/dist-
packages/sklearn/utils/_param_validation.py", line 95, in
validate_parameter_constraints
    raise InvalidParameterError(
sklearn.utils._param_validation.InvalidParameterError: The 'max_features'
```

```
0.2094543  0.2094543  0.2094543  0.2094543  0.25285485        nan
       nan        nan 0.20443127 0.20443127        nan 0.33405456
0.2094543  0.27004322        nan        nan 0.28221399        nan
0.30878396 0.2094543  0.2094543  0.2094543  0.2094543  0.28221399
0.33545348 0.27468132 0.09298482 0.32022495 0.20443127 0.2094543
0.2094543         nan        nan 0.2094543  0.2980411  0.2094543
0.2094543         nan        nan 0.2094543  0.2094543         nan
       nan 0.33876946 0.34025997        nan]
 warnings.warn(
```

Evaluamos un rango de valores más acotado de cada hiperparámetro (proveniente del Randomized Search) mediante Grid Search con validación cruzada.

[206]:
```
params_tree_grid = {
    "max_depth": [2,3,4],
    "min_samples_split": [3,4,5],
    "min_samples_leaf": [3,4,5],
    "max_features": ["auto", "sqrt", "log2", None],
    "splitter": ["best", "random"],
    "max_leaf_nodes": [None,10,20],
    "min_weight_fraction_leaf": [0.0,0.01,0.05],
    "min_impurity_decrease": [0.0,0.001,0.01]
}

grid_search_tree = GridSearchCV(clf2, params_tree_grid, cv=5, scoring="f1",␣
 ↪n_jobs=-1, return_train_score=True)
grid_search_tree.fit(X_train[selected_features], y_train)

print(f"Mejores parámetros: {grid_search_tree.best_params_}")

max_index_train = np.nanargmax(grid_search_tree.cv_results_["mean_train_score"])
print("Mejor F1 Score (train):", {round(grid_search_tree.
 ↪cv_results_["mean_train_score"][max_index_train], 3)}, "±",␣
 ↪{round(grid_search_tree.cv_results_["std_train_score"][max_index_train], 3)})

max_index_test = np.nanargmax(grid_search_tree.cv_results_["mean_test_score"])
print("Mejor F1 Score (test):", {round(grid_search_tree.
 ↪cv_results_["mean_test_score"][max_index_test], 3)}, "±",␣
 ↪{round(grid_search_tree.cv_results_["std_test_score"][max_index_test], 3)})
```

```
/usr/local/lib/python3.10/dist-
packages/sklearn/model_selection/_validation.py:540: FitFailedWarning:
7290 fits failed out of a total of 29160.
The score on these train-test partitions for these parameters will be set to
nan.
If these failures are not expected, you can try to debug them by setting
error_score='raise'.
```

```
Below are more details about the failures:
--------------------------------------------------------------------------------
3625 fits failed with the following error:
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-
packages/sklearn/model_selection/_validation.py", line 888, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 1466, in
wrapper
    estimator._validate_params()
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 666, in
_validate_params
    validate_parameter_constraints(
  File "/usr/local/lib/python3.10/dist-
packages/sklearn/utils/_param_validation.py", line 95, in
validate_parameter_constraints
    raise InvalidParameterError(
sklearn.utils._param_validation.InvalidParameterError: The 'max_features'
parameter of DecisionTreeClassifier must be an int in the range [1, inf), a
float in the range (0.0, 1.0], a str among {'log2', 'sqrt'} or None. Got 'auto'
instead.

--------------------------------------------------------------------------------
3665 fits failed with the following error:
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-
packages/sklearn/model_selection/_validation.py", line 888, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 1466, in
wrapper
    estimator._validate_params()
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 666, in
_validate_params
    validate_parameter_constraints(
  File "/usr/local/lib/python3.10/dist-
packages/sklearn/utils/_param_validation.py", line 95, in
validate_parameter_constraints
    raise InvalidParameterError(
sklearn.utils._param_validation.InvalidParameterError: The 'max_features'
parameter of DecisionTreeClassifier must be an int in the range [1, inf), a
float in the range (0.0, 1.0], a str among {'sqrt', 'log2'} or None. Got 'auto'
instead.

  warnings.warn(some_fits_failed_message, FitFailedWarning)

Mejores parámetros: {'max_depth': 3, 'max_features': 'sqrt', 'max_leaf_nodes':
None, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 3, 'min_samples_split':
3, 'min_weight_fraction_leaf': 0.01, 'splitter': 'best'}
```

```
Mejor F1 Score (train): {0.355} ± {0.005}
Mejor F1 Score (test): {0.354} ± {0.02}

/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_search.py:1103:
UserWarning: One or more of the test scores are non-finite: [       nan
nan        nan … 0.28206229 0.30814008 0.28206229]
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_search.py:1103:
UserWarning: One or more of the train scores are non-finite: [       nan
nan        nan … 0.28221399 0.31447005 0.28221399]
  warnings.warn(
```

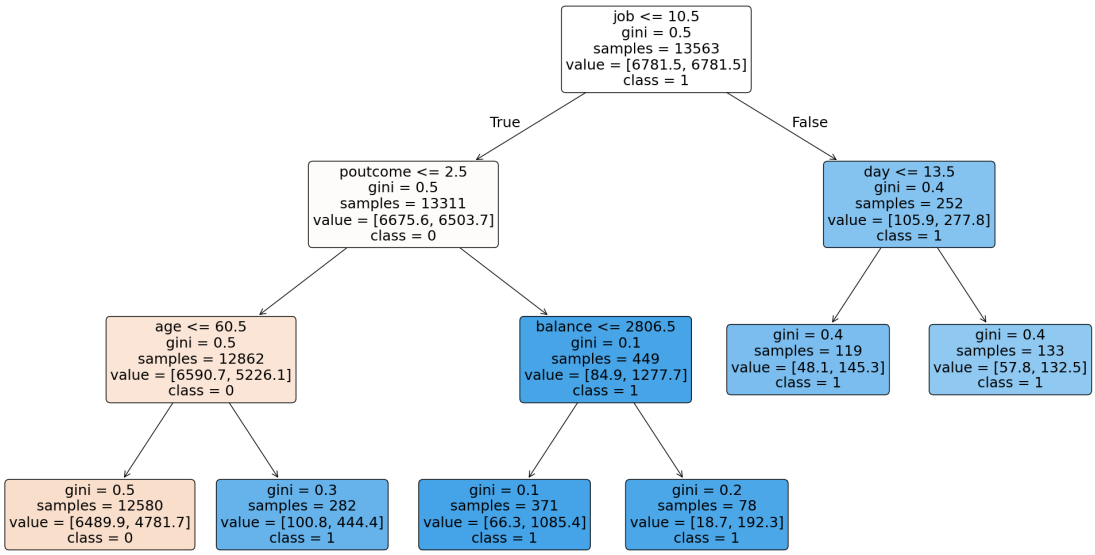
El modelo optimizado tiene `F1 Scores` muy parecidos tanto en los conjuntos de entrenamiento como de validación, demostrando que es consistente y que no está sobre ajustando.

Finalmente, evaluamos al modelo optimizado en la muestra de validación para observar cómo se comporta con una muestra de datos totalmente desconocida.

[207]:
```
clf2 = DecisionTreeClassifier(max_depth=3, max_features="sqrt",␣
 ↪max_leaf_nodes=None, min_impurity_decrease=0.0,
                               min_samples_leaf=3, min_samples_split=3,␣
 ↪min_weight_fraction_leaf=0.01, splitter="best",
                               random_state=42, class_weight="balanced")
clf2.fit(X_test[selected_features], y_test)
y_preds = clf2.predict(X_test[selected_features])

print("F1 Score:", {round(f1_score(y_test, y_preds).mean(), 3)})
```

```
F1 Score: {0.364}
```

[208]:
```
plt.figure(figsize=(25,14))
plot_tree(clf2, feature_names=X_test[selected_features].columns,␣
 ↪class_names=[str(cla) for cla in clf2.classes_], filled=True, impurity=True,␣
 ↪rounded=True, precision=1, fontsize=18)
plt.show()
```




El modelo mantuvo el rendimiento que tenía en la muestra de entrenamiento e incluso lo mejoró. Finalmente, decidimos predecir con este modelo, ya que a pesar del desbalance de la muestra y su error irreducible (ruido), este logra satisfacer nuestros requerimientos mínimos para predecir si un cliente se suscribirá o no a un plazo fijo.

**Random Forest** Para finalizar con nuestro trabajo práctico, decidimos entrenar un modelo Random Forest y comparar su rendimiento con el modelo final. Nos gustaría saber qué rendimiento obtiene un modelo de ensamble y más complejo que el que hicimos anteriormente y comparar sus resultados.

Primero creamos al clasificador y optimizamos sus hiperparámetros con Randomized Search y validación cruzada.

[209]:
```
params_rf_rand = {
    "max_depth": [1,2,3,4,5,6],
    "min_samples_split": [15,20,25,30,35,40],
    "min_samples_leaf": [5,10,15,20,25],
    "max_features": ["auto", "sqrt", "log2", None],
    "bootstrap": [True, False],
    "max_leaf_nodes": [None,10,20,30],
    "min_weight_fraction_leaf": [0.0,0.01,0.05,0.1],
    "min_impurity_decrease": [0.0,0.001,0.01,0.1],
    "n_estimators": [80,100,120,140,160,180,200,220,300]
}

clf3 = RandomForestClassifier(n_jobs=-1, random_state=42,␣
 ↪class_weight="balanced")
```

```
random_search_rf = RandomizedSearchCV(clf3, params_rf_rand, n_iter=100, cv=5,␣
 ↪random_state=42, scoring="f1", return_train_score=True)
random_search_rf.fit(X_train[selected_features], y_train)

print(f"Mejores parámetros: {random_search_rf.best_params_}")

max_index_train = np.nanargmax(random_search_rf.cv_results_["mean_train_score"])
print("Mejor F1-Score (train):", {round(random_search_rf.
 ↪cv_results_["mean_train_score"][max_index_train], 2)}, "±",␣
 ↪{round(random_search_rf.cv_results_["std_train_score"][max_index_train], 2)})

max_index_test = np.nanargmax(random_search_rf.cv_results_["mean_test_score"])
print("Mejor F1-Score (test):", {round(random_search_rf.
 ↪cv_results_["mean_test_score"][max_index_test], 2)}, "±",␣
 ↪{round(random_search_rf.cv_results_["std_test_score"][max_index_test], 2)})
```

```
/usr/local/lib/python3.10/dist-
packages/sklearn/model_selection/_validation.py:540: FitFailedWarning:
125 fits failed out of a total of 500.
The score on these train-test partitions for these parameters will be set to
nan.
If these failures are not expected, you can try to debug them by setting
error_score='raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
125 fits failed with the following error:
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-
packages/sklearn/model_selection/_validation.py", line 888, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 1466, in
wrapper
    estimator._validate_params()
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 666, in
_validate_params
    validate_parameter_constraints(
  File "/usr/local/lib/python3.10/dist-
packages/sklearn/utils/_param_validation.py", line 95, in
validate_parameter_constraints
    raise InvalidParameterError(
sklearn.utils._param_validation.InvalidParameterError: The 'max_features'
parameter of RandomForestClassifier must be an int in the range [1, inf), a
float in the range (0.0, 1.0], a str among {'sqrt', 'log2'} or None. Got 'auto'
instead.
```

```
  warnings.warn(some_fits_failed_message, FitFailedWarning)
/usr/local/lib/python3.10/dist-packages/numpy/ma/core.py:2820: RuntimeWarning:
invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_search.py:1103:
UserWarning: One or more of the test scores are non-finite: [0.33385393
0.34774102 0.29210341 0.33873466 0.12566362 0.33560896
 0.33873466 0.34727336 0.28206229 0.08379066 0.33873466 0.33863246
 0.29210341 0.30261763        nan 0.33873466 0.34165819 0.28206229
 0.331469   0.33873466 0.34900663 0.33873466        nan 0.30265511
 0.313622   0.04191769 0.29210341        nan 0.16758132        nan
        nan 0.16758132        nan        nan 0.35460362 0.33236908
        nan 0.33873466 0.20945428 0.31595195 0.33873466 0.28206229
 0.33385393        nan 0.20945428 0.04191769 0.34352595        nan
 0.34528686 0.33873466 0.33802727        nan        nan 0.35625783
 0.29210341        nan 0.34192035 0.33873466 0.29210341 0.35845463
 0.34102588        nan 0.31519418        nan 0.16758132        nan
 0.31579827 0.31249986 0.28739761 0.33873466 0.35895826 0.31141471
 0.33873466 0.20945428 0.32602457 0.33873466 0.20945428 0.29210341
 0.35383261        nan        nan 0.08379066 0.12566362 0.12570835
 0.16758132 0.33873466 0.04191769 0.29210341        nan        nan
        nan 0.27119072        nan        nan 0.34642065        nan
 0.16758132 0.31969503        nan 0.31737144]
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_search.py:1103:
UserWarning: One or more of the train scores are non-finite: [0.33390043
0.35634178 0.29229018 0.33876946 0.12567482 0.3456074
 0.33876946 0.35567985 0.28221399 0.08377948 0.33876946 0.34239601
 0.29229018 0.30486802        nan 0.33876946 0.3498684  0.28221399
 0.33995537 0.33876946 0.35635734 0.33876946        nan 0.30507456
 0.31600862 0.04188415 0.29229018        nan 0.16755897        nan
        nan 0.16755897        nan        nan 0.35894892 0.34060551
        nan 0.33876946 0.2094543  0.31815587 0.33876946 0.28221399
 0.33390043        nan 0.2094543  0.04188415 0.35195783        nan
 0.34542181 0.33876946 0.34448123        nan        nan 0.36425362
 0.29229018        nan 0.34803987 0.33876946 0.29229018 0.36498495
 0.3437364         nan 0.31783312        nan 0.16755897        nan
 0.31857216 0.3166356  0.29376607 0.33876946 0.36529811 0.31476636
 0.33876946 0.2094543  0.33655736 0.33876946 0.2094543  0.29229018
 0.35370309        nan        nan 0.08377948 0.12567482 0.12566363
 0.16755897 0.33876946 0.04188415 0.29229018        nan        nan
        nan 0.27817873        nan        nan 0.35614401        nan
 0.16755897 0.3237765         nan 0.31552827]
  warnings.warn(

Mejores parámetros: {'n_estimators': 300, 'min_weight_fraction_leaf': 0.0,
'min_samples_split': 30, 'min_samples_leaf': 15, 'min_impurity_decrease': 0.001,
'max_leaf_nodes': 30, 'max_features': 'log2', 'max_depth': 4, 'bootstrap': True}
```

```
Mejor F1-Score (train): {0.37} ± {0.0}
Mejor F1-Score (test): {0.36} ± {0.02}
```

**Comparación de modelos** Segundo modelo (árbol más complejo) y Benchmark: * El benchmark obtuvo un F1 Score de 0.11, mientras que el árbol más complejo obtuvo un F1 Score de 0.35 $\pm$ 0.02. Claramente, el modelo elegido rinde mejor que el benchmark y para nuestro caso de estudio es el más adecuado.

Segundo modelo (árbol más complejo) y Random Forest: * El random forest obtuvo un F1 Score igual a 0.36 $\pm$ 0.02. Este resultado es prácticamente igual al que obtuvo el modelo más complejo. Por lo tanto, creemos que en general el modelo más adecuado entre estos dos es el elegido, ya que es un modelo más simple, más fácil de interpretar y más barato computacionalmente respecto del random forest y logra predecir igual de bien.

**Conclusión** Finalmente, logramos obtener un modelo que es mejor que el resto de los que entrenamos y que mantiene un equilibrio entre el sesgo y la varianza sin presentar señales de sobre ajuste. Sin embargo, estamos dispuestos a desarrollar nuevas tareas para mejorar aún más el resultado de nuestras predicciones. Podemos nombrar tales como técnicas para balancear la muestra de datos (undersampling o oversampling), probar modelos como el XGBoost y desarrollar evaluaciones de modelos con el área bajo la curva de precisión-exhaustividad (PR-AUC).