Clasificación - dataset Titanic

May 20, 2026

0.1 1. Carga del dataset

En esta práctica se trabajará con el dataset **Titanic**, orientado a un problema de **clasificación binaria**.

La variable objetivo será `Survived`, que indica si el pasajero sobrevivió (`1`) o no (`0`).

En este primer apartado se cargará el dataset y se mostrará una primera vista general de su contenido.

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

titanic = sns.load_dataset('titanic')
titanic.head()
```

[ ]:
```
   survived  pclass     sex   age  sibsp  parch     fare embarked  class  \
0         0       3    male  22.0      1      0   7.2500        S  Third
1         1       1  female  38.0      1      0  71.2833        C  First
2         1       3  female  26.0      0      0   7.9250        S  Third
3         1       1  female  35.0      1      0  53.1000        S  First
4         0       3    male  35.0      0      0   8.0500        S  Third

     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
1  woman       False    C    Cherbourg   yes  False
2  woman       False  NaN  Southampton   yes   True
3  woman       False    C  Southampton   yes  False
4    man        True  NaN  Southampton    no   True
```

[ ]:
```
print("Dimensiones del dataset:", titanic.shape)
```

```
Dimensiones del dataset: (891, 15)
```

0.1.1 Variables del dataset Titanic

El dataset **Titanic** contiene información sobre pasajeros del famoso transatlántico Titanic.
Cada fila representa a un pasajero y cada columna describe alguna característica personal, social o del viaje.

- **survived**
  Indica si el pasajero sobrevivió o no al naufragio.
  - `0` → no sobrevivió
  - `1` → sobrevivió

  Esta será la **variable objetivo** en el problema de **clasificación binaria**.
- **pclass**
  Clase del billete del pasajero.
  - `1` → primera clase
  - `2` → segunda clase
  - `3` → tercera clase

  Puede interpretarse como un indicador aproximado del nivel socioeconómico del pasajero.
- **sex**
  Sexo del pasajero (`male` o `female`).
- **age**
  Edad del pasajero en años.
  Puede contener valores faltantes.
- **sibsp**
  Número de hermanos/as o cónyuges del pasajero que también viajaban a bordo del Titanic.
- **parch**
  Número de padres, madres o hijos del pasajero que también viajaban a bordo.
- **fare**
  Precio pagado por el billete.
- **embarked**
  Puerto en el que embarcó el pasajero.
  - `C` → Cherbourg
  - `Q` → Queenstown
  - `S` → Southampton
- **class**
  Representación textual de la clase del pasajero (`First`, `Second`, `Third`).
  Contiene una información muy similar a la de `pclass`.
- **who**
  Clasificación general del pasajero en categorías como `man`, `woman` o `child`.
- **adult_male**
  Indica si el pasajero es un hombre adulto (`True` o `False`).

- **deck**
  Cubierta del barco en la que se encontraba la cabina del pasajero.
  Suele contener bastantes valores faltantes.
- **embark_town**
  Nombre completo de la ciudad o puerto de embarque.
  Contiene una información muy similar a la de `embarked`.
- **alive**
  Representación textual de la supervivencia (`yes` o `no`).
  Contiene la misma información que `survived`, por lo que no debe utilizarse como variable predictora.
- **alone**
  Indica si el pasajero viajaba solo (`True`) o acompañado (`False`).

0.2 2. Análisis exploratorio del dataset

Antes de entrenar los modelos de clasificación, conviene realizar una inspección mínima del dataset para identificar qué aspectos deben tratarse en el preprocesado.

En particular, interesa revisar:

- tipos de variables
- valores faltantes
- variables categóricas
- distribución de la variable objetivo

Además, esta inspección también ayudará a decidir si algunas columnas deben eliminarse antes del entrenamiento.

[ ]:
```
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    category
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object
```

```
 13  alive        891 non-null    object
 14  alone        891 non-null    bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
```

[ ]:
```
titanic.isnull().sum()
```

[ ]:
```
survived         0
pclass           0
sex              0
age            177
sibsp            0
parch            0
fare             0
embarked         2
class            0
who              0
adult_male       0
deck           688
embark_town      2
alive            0
alone            0
dtype: int64
```

[ ]:
```
titanic.select_dtypes(include=["object"]).columns.tolist()
```

[ ]:
```
['sex', 'embarked', 'who', 'embark_town', 'alive']
```

[ ]:
```
titanic.describe()
```

[ ]:

	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

0.2.1 Distribución de la variable objetivo

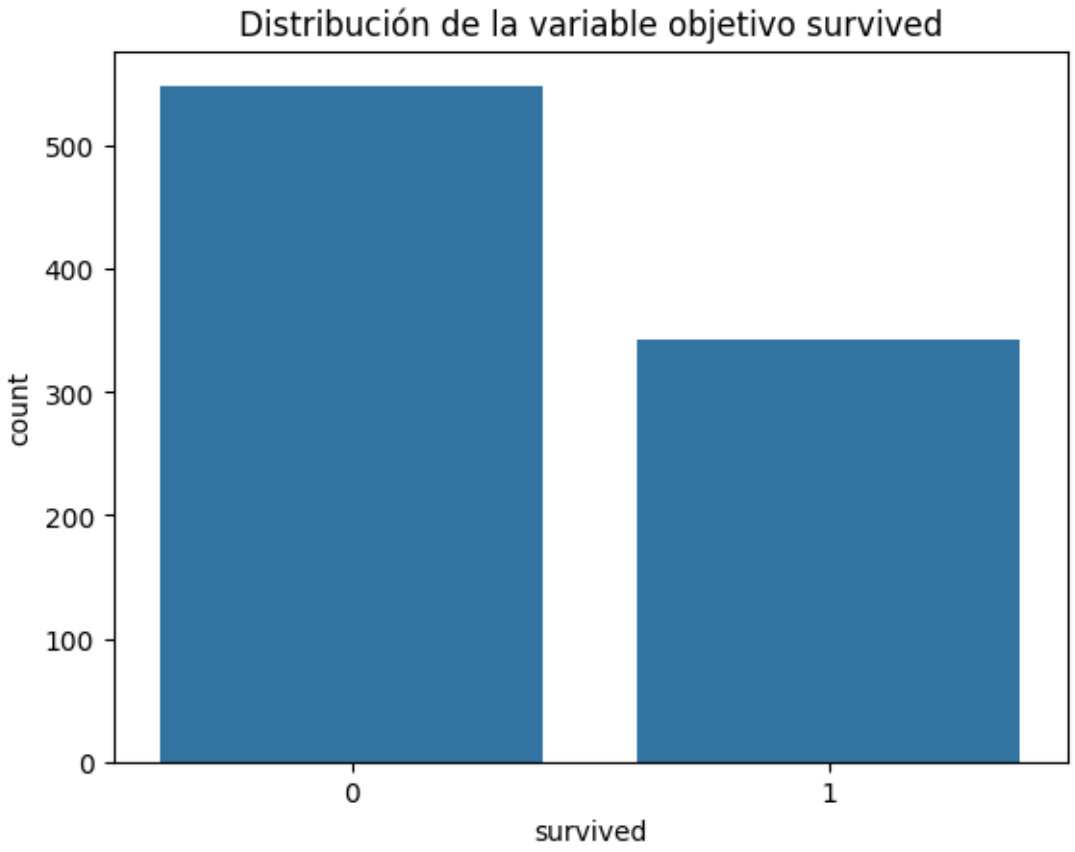
La variable `survived` indica si el pasajero sobrevivió (`1`) o no (`0`).

Antes de entrenar los modelos, conviene comprobar cómo se distribuyen ambas clases para detectar si existe o no desbalanceo entre clases .

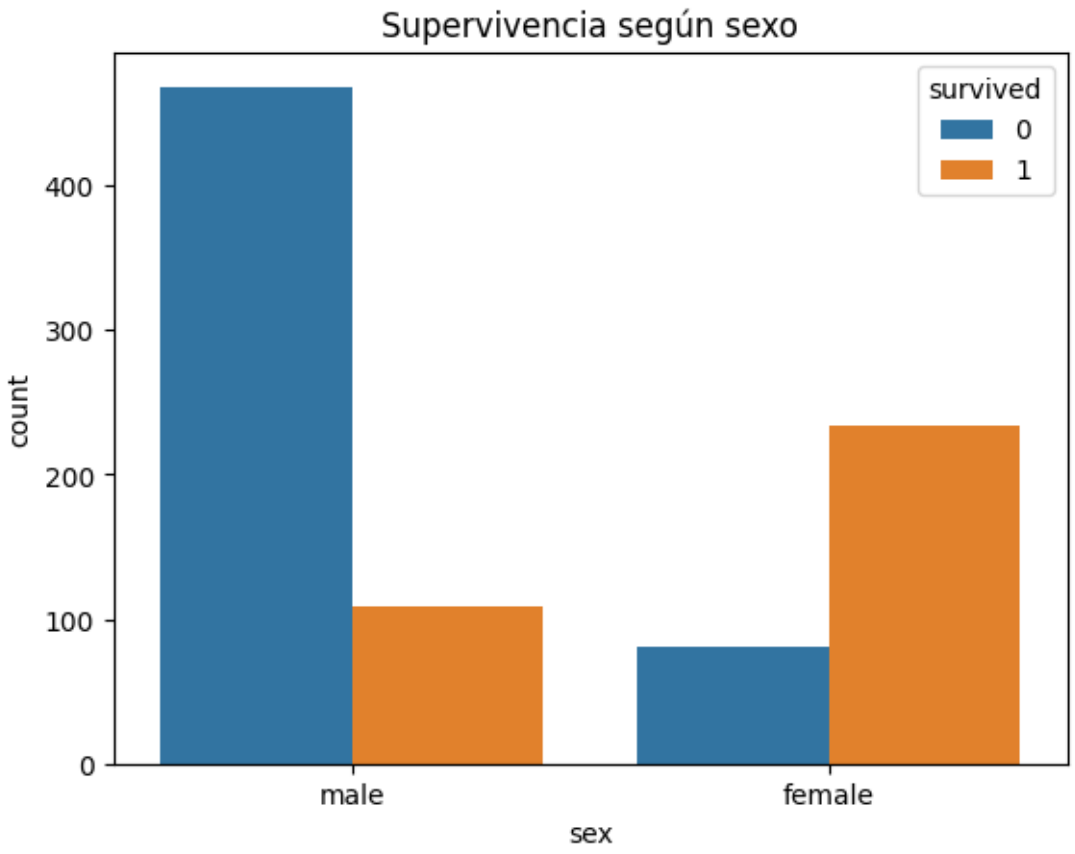
[ ]:
```
titanic["survived"].value_counts()
```

[ ]:
```
survived
0    549
1    342
Name: count, dtype: int64
```

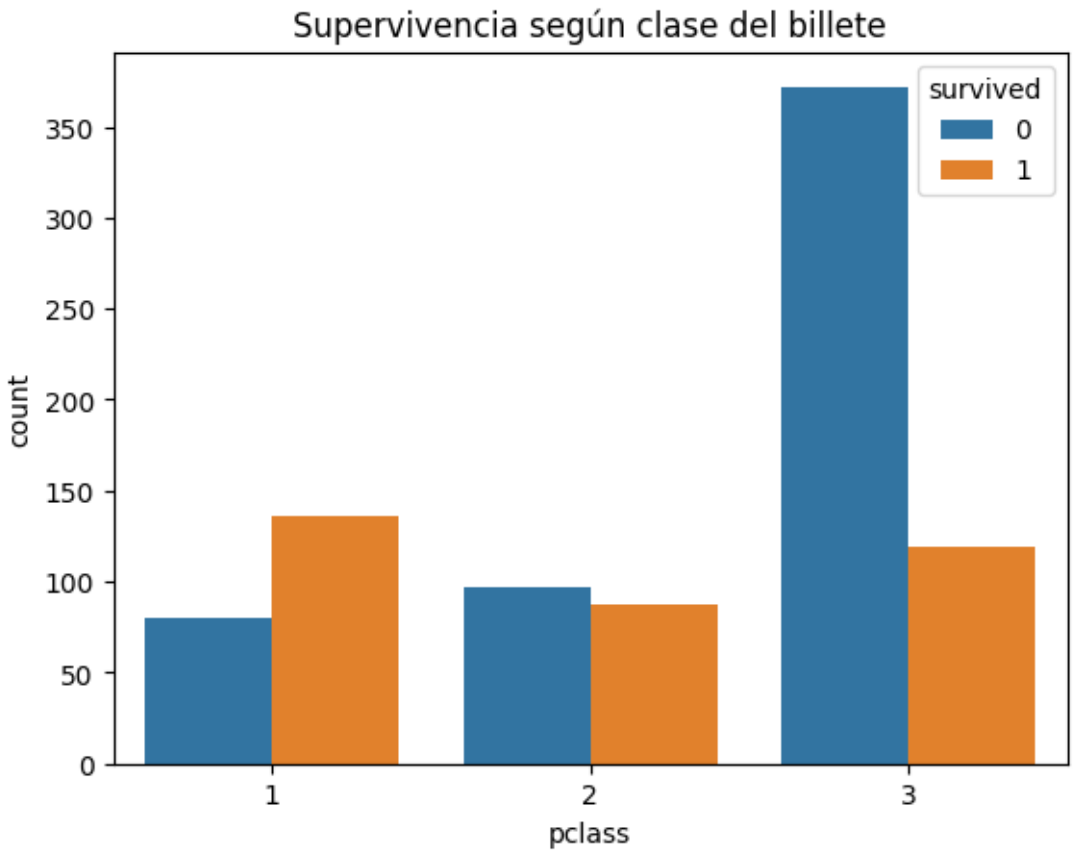
[ ]:
```
sns.countplot(data=titanic, x="survived")
plt.title("Distribución de la variable objetivo survived")
plt.show()
```



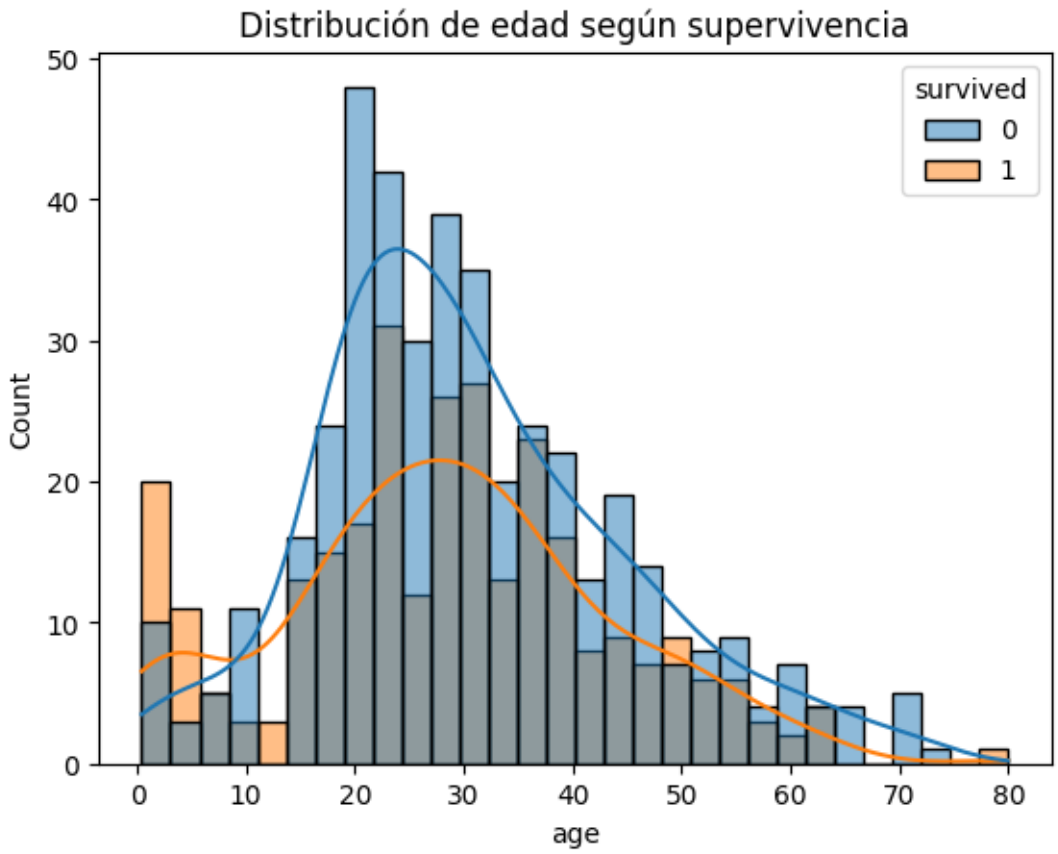
[ ]:
```
sns.countplot(data=titanic, x="sex", hue="survived")
plt.title("Supervivencia según sexo")
plt.show()
```



[ ]:
```
sns.countplot(data=titanic, x="pclass", hue="survived")
plt.title("Supervivencia según clase del billete")
plt.show()
```



[ ]:
```
sns.histplot(data=titanic, x="age", hue="survived", kde=True, bins=30)
plt.title("Distribución de edad según supervivencia")
plt.show()
```



0.2.2 Revisión básica de columnas potencialmente problemáticas

Además de los valores faltantes y de la variable objetivo, conviene revisar algunas columnas que podrían no resultar adecuadas para el entrenamiento del modelo.

En este dataset, interesará prestar especial atención a variables como:

- `alive`, porque contiene la misma información que `survived`
- `pclass` y `class`, porque representan la misma información con formatos distintos
- `embarked` y `embark_town`, porque también contienen una información muy similar
- `deck`, porque suele contener una cantidad importante de valores faltantes

[ ]:
```
titanic[["alive", "pclass", "class", "embarked", "embark_town", "deck"]].
↪head(10)
```

[ ]:

	alive	pclass	class	embarked	embark_town	deck
0	no	3	Third	S	Southampton	NaN
1	yes	1	First	C	Cherbourg	C
2	yes	3	Third	S	Southampton	NaN
3	yes	1	First	S	Southampton	C
4	no	3	Third	S	Southampton	NaN

```
5    no       3   Third        Q   Queenstown  NaN
6    no       1   First        S  Southampton    E
7    no       3   Third        S  Southampton  NaN
8   yes       3   Third        S  Southampton  NaN
9   yes       2  Second        C    Cherbourg  NaN
```

[ ]:
```
titanic[["alive", "pclass", "class", "embarked", "embark_town", "deck"]].
↪isnull().sum()
```

[ ]:
```
alive            0
pclass           0
class            0
embarked         2
embark_town      2
deck           688
dtype: int64
```

A partir de esta inspección inicial, ya se pueden identificar algunos aspectos importantes para el preprocesado y el entrenamiento de modelos:

- existen columnas con valores faltantes, como `age`, `embarked`, `embark_town` o `deck`
- hay variables categóricas que deberán transformarse
- algunas columnas pueden no resultar adecuadas para el entrenamiento y convendrá eliminarlas
- algunas variables pueden contener información redundante o incluso fuga de información

En particular:

- `alive` no debe utilizarse como predictora, ya que contiene la misma información que la variable objetivo `survived`
- `pclass` y `class` son redundantes; en esta práctica se mantendrá `class` y se eliminará `pclass`. Eliminamos `pclass` porque puede confundir a los clasificadores al tener valores numéricos que implícitamente definan un orden.
- `embarked` y `embark_town` también son redundantes; en esta práctica se mantendrá `embarked` y se eliminará `embark_town`
- `deck` presenta muchos valores faltantes, por lo que también se eliminará para simplificar el preprocesado

En la siguiente sección se definirá la variable objetivo y se seleccionarán las variables predictoras que se utilizarán en la práctica.

0.2.3 Eliminación de columnas no utilizadas

Una vez revisadas las columnas problemáticas, se eliminarán del dataset aquellas que no se utilizarán en el entrenamiento de los modelos.

[ ]:
```
titanic = titanic.drop(columns=["alive", "pclass", "embark_town", "deck"])
```

[ ]:
```
titanic.head()
```

[ ]:

	survived	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	alone
0	0	male	22.0	1	0	7.2500	S	Third	man	True	False
1	1	female	38.0	1	0	71.2833	C	First	woman	False	False
2	1	female	26.0	0	0	7.9250	S	Third	woman	False	True
3	1	female	35.0	1	0	53.1000	S	First	woman	False	False
4	0	male	35.0	0	0	8.0500	S	Third	man	True	True

0.3 3. Separación entre variables predictoras y variable objetivo

En esta práctica, la variable objetivo será `survived`, que indica si el pasajero sobrevivió (`1`) o no (`0`).

Una vez eliminadas en la sección anterior algunas columnas problemáticas o redundantes, el siguiente paso consiste en separar:

- las **variables predictoras** (`X`)
- la **variable objetivo** (`y`)

[ ]:
```
X = titanic.drop(columns=["survived"])
y = titanic["survived"].copy()
```

[ ]:
```
print("Dimensiones de X:", X.shape)
print("Dimensiones de y:", y.shape)
```

```
Dimensiones de X: (891, 10)
Dimensiones de y: (891,)
```

[ ]:
```
X
```

[ ]:

	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	alone
0	male	22.0	1	0	7.2500	S	Third	man	True	False
1	female	38.0	1	0	71.2833	C	First	woman	False	False
2	female	26.0	0	0	7.9250	S	Third	woman	False	True
3	female	35.0	1	0	53.1000	S	First	woman	False	False
4	male	35.0	0	0	8.0500	S	Third	man	True	True
..	…	…	…	…	…	…	…	…	…	…
886	male	27.0	0	0	13.0000	S	Second	man	True	True
887	female	19.0	0	0	30.0000	S	First	woman	False	True
888	female	NaN	1	2	23.4500	S	Third	woman	False	False
889	male	26.0	0	0	30.0000	C	First	man	True	True
890	male	32.0	0	0	7.7500	Q	Third	man	True	True

```
     alone
0    False
1    False
2     True
3    False
4     True
..     …
886   True
887   True
888  False
889   True
890   True

[891 rows x 10 columns]
```

[ ]:
```
y
```

[ ]:
```
0      0
1      1
2      1
3      1
4      0
      ..
886    0
887    1
888    0
889    1
890    0
Name: survived, Length: 891, dtype: int64
```

0.4 4. División del dataset en entrenamiento y prueba

Una vez definidas las variables predictoras y la variable objetivo, el siguiente paso consiste en dividir el dataset en dos partes:

- **train**: se utilizará para entrenar los modelos
- **test**: se reservará para evaluar el rendimiento final sobre datos no vistos

En problemas de clasificación, además, conviene que esta partición mantenga aproximadamente la misma proporción de clases en ambos subconjuntos. Para ello se utilizará el parámetro `stratify=y`.

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

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42,
    stratify=y
```

```
)
```

[ ]:
```
print("Dimensiones de X_train:", X_train.shape)
print("Dimensiones de X_test:", X_test.shape)
print("Dimensiones de y_train:", y_train.shape)
print("Dimensiones de y_test:", y_test.shape)
```

```
Dimensiones de X_train: (712, 10)
Dimensiones de X_test: (179, 10)
Dimensiones de y_train: (712,)
Dimensiones de y_test: (179,)
```

[ ]:
```
print("Distribución de survived en y_train:")
print(y_train.value_counts(normalize=True))

print()

print("Distribución de survived en y_test:")
print(y_test.value_counts(normalize=True))
```

```
Distribución de survived en y_train:
survived
0    0.616573
1    0.383427
Name: proportion, dtype: float64

Distribución de survived en y_test:
survived
0    0.614525
1    0.385475
Name: proportion, dtype: float64
```

0.5 5. Preprocesado específico para modelado ajustado solo con train

Una vez realizada la partición entre entrenamiento y prueba, el siguiente paso consiste en definir el preprocesado que se aplicará a los datos antes de entrenar los modelos de clasificación.

Este preprocesado debe ajustarse únicamente con `X_train`. Después, ese mismo procesamiento se aplicará a `X_test`.

En esta práctica se considerarán dos tipos de variables:

- **variables numéricas**, sobre las que se aplicará imputación y escalado
- **variables categóricas**, sobre las que se aplicará imputación y codificación

De este modo, el dataset quedará preparado para entrenar distintos modelos de clasificación dentro de un flujo común basado en pipelines.

[ ]:
```
num_attribs = X_train.select_dtypes(include=[np.number]).columns.tolist()
cat_attribs = X_train.select_dtypes(exclude=[np.number]).columns.tolist()
```

```
print("Columnas numéricas:", num_attribs)
print("Columnas categóricas:", cat_attribs)
```

```
Columnas numéricas: ['age', 'sibsp', 'parch', 'fare']
Columnas categóricas: ['sex', 'embarked', 'class', 'who', 'adult_male', 'alone']
```

0.5.1 Definición del pipeline numérico

En las variables numéricas se aplicarán dos transformaciones:

- **imputación por la mediana**, para tratar posibles valores faltantes
- **escalado con `StandardScaler`**, para dejar las variables numéricas en una escala más homogénea

[ ]:
```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

[ ]:
```
num_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler())
])
```

0.5.2 Definición del pipeline categórico

En las variables categóricas se aplicarán también dos transformaciones:

- **imputación por el valor más frecuente**, para tratar posibles valores faltantes
- **One Hot Encoding**, para convertir las categorías a una representación numérica adecuada para los modelos

[ ]:
```
cat_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])
```

0.5.3 Combinación del preprocesado numérico y categórico

Una vez definidos ambos pipelines, se combinan en una única estructura de preprocesado mediante `ColumnTransformer`.

De este modo:

- el pipeline numérico se aplicará solo a las columnas numéricas
- el pipeline categórico se aplicará solo a las columnas categóricas

[ ]:
```
preprocessor = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs)
```

```
])
```

0.5.4 Ajuste del preprocesado con `X_train`

A continuación, el preprocesado se ajustará utilizando únicamente `X_train`.

Después, ese mismo preprocesado ya ajustado se aplicará a `X_test`.

[ ]:
```
X_train_prepared = preprocessor.fit_transform(X_train)
X_test_prepared = preprocessor.transform(X_test)
```

[ ]:
```
print("Dimensiones originales de X_train:", X_train.shape)
print("Dimensiones transformadas de X_train:", X_train_prepared.shape)
print("Dimensiones originales de X_test:", X_test.shape)
print("Dimensiones transformadas de X_test:", X_test_prepared.shape)
```

```
Dimensiones originales de X_train: (712, 10)
Dimensiones transformadas de X_train: (712, 19)
Dimensiones originales de X_test: (179, 10)
Dimensiones transformadas de X_test: (179, 19)
```

[ ]:
```
# Este código nos vuelca a un df el resultado de la transformación
# X_train_prepared es una matriz de valores en la que es difícil ver que el␣
 ↪resultado es el esperado

feature_names = preprocessor.get_feature_names_out()

X_train_prepared_df = pd.DataFrame(
    X_train_prepared,
    columns=feature_names,
    index=X_train.index
)

X_train_prepared_df
```

[ ]:
```
     num__age  num__sibsp  num__parch  num__fare  cat__sex_female  \
692 -0.081135   -0.465084   -0.466183   0.513812              0.0
481 -0.081135   -0.465084   -0.466183  -0.662563              0.0
527 -0.081135   -0.465084   -0.466183   3.955399              0.0
855 -0.887827   -0.465084    0.727782  -0.467874              1.0
801  0.110934    0.478335    0.727782  -0.115977              1.0
..         …           …           …          …                …
359 -0.081135   -0.465084   -0.466183  -0.498500              1.0
258  0.418245   -0.465084   -0.466183  10.005329              1.0
736  1.417007    0.478335    3.115713   0.053205              1.0
462  1.340179   -0.465084   -0.466183   0.139097              0.0
507 -0.081135   -0.465084   -0.466183  -0.109730              0.0

     cat__sex_male  cat__embarked_C  cat__embarked_Q  cat__embarked_S  \
```

```
692           1.0             0.0             0.0             1.0
481           1.0             0.0             0.0             1.0
527           1.0             0.0             0.0             1.0
855           0.0             0.0             0.0             1.0
801           0.0             0.0             0.0             1.0
..            …               …               …               …
359           0.0             0.0             1.0             0.0
258           0.0             1.0             0.0             0.0
736           0.0             0.0             0.0             1.0
462           1.0             0.0             0.0             1.0
507           1.0             0.0             0.0             1.0

     cat__class_First  cat__class_Second  cat__class_Third  cat__who_child  \
692               0.0                0.0               1.0             0.0
481               0.0                1.0               0.0             0.0
527               1.0                0.0               0.0             0.0
855               0.0                0.0               1.0             0.0
801               0.0                1.0               0.0             0.0
..                …                  …                 …               …
359               0.0                0.0               1.0             0.0
258               1.0                0.0               0.0             0.0
736               0.0                0.0               1.0             0.0
462               1.0                0.0               0.0             0.0
507               1.0                0.0               0.0             0.0

     cat__who_man  cat__who_woman  cat__adult_male_False  \
692           1.0             0.0                    0.0
481           1.0             0.0                    0.0
527           1.0             0.0                    0.0
855           0.0             1.0                    1.0
801           0.0             1.0                    1.0
..            …               …                      …
359           0.0             1.0                    1.0
258           0.0             1.0                    1.0
736           0.0             1.0                    1.0
462           1.0             0.0                    0.0
507           1.0             0.0                    0.0

     cat__adult_male_True  cat__alone_False  cat__alone_True
692                   1.0               0.0              1.0
481                   1.0               0.0              1.0
527                   1.0               0.0              1.0
855                   0.0               1.0              0.0
801                   0.0               1.0              0.0
..                    …                 …                …
359                   0.0               0.0              1.0
258                   0.0               0.0              1.0
```

```
736                0.0               1.0               0.0
462                1.0               0.0               1.0
507                1.0               0.0               1.0

[712 rows x 19 columns]
```

0.6 6. Logistic Regression + GridSearchCV

El primer modelo que se entrenará será una **regresión logística**.

En este caso, el preprocesado y el modelo se integrarán en un único pipeline completo. Después, se utilizará `GridSearchCV` para probar distintas configuraciones del modelo y seleccionar automáticamente la mejor mediante validación cruzada.

La métrica principal que se utilizará en esta práctica será **F1-score**, ya que combina *precision* y *recall* y resulta especialmente útil en problemas de clasificación binaria.

[ ]:
```python
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
```

0.6.1 Definición del pipeline completo

A continuación, se construirá un pipeline que agrupe:

- el preprocesado definido en la sección anterior
- el modelo de regresión logística

[ ]:
```python
full_pipeline_logreg = Pipeline([
    ("preprocessor", preprocessor),
    ("model", LogisticRegression())
])
```

0.6.2 Definición de la búsqueda de hiperparámetros

Se probarán distintas configuraciones del modelo modificando algunos hiperparámetros habituales de la regresión logística:

- `C`, que controla el nivel de regularización
- `solver`, que indica el método de optimización utilizado
- `max_iter`, para asegurar convergencia del entrenamiento

[ ]:
```python
param_grid_logreg = {
    "model__C": [0.01, 0.1, 1, 10],
    "model__solver": ["liblinear", "lbfgs"],
    "model__max_iter": [500, 1000]
}
```

0.6.3 Entrenamiento con GridSearchCV

A continuación, se aplicará `GridSearchCV` sobre el conjunto `train` para seleccionar automáticamente la mejor configuración del modelo mediante validación cruzada.

[ ]:
```
grid_search_logreg = GridSearchCV(
    estimator=full_pipeline_logreg,
    param_grid=param_grid_logreg,
    scoring="f1",
    cv=5
)
```

[ ]:
```
grid_search_logreg.fit(X_train, y_train)
```

[ ]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('preprocessor',
                                        ColumnTransformer(transformers=[('num',
Pipeline(steps=[('imputer',
                 SimpleImputer(strategy='median')),
                ('scaler',
                 StandardScaler())]),
                                                                        ['age',
'sibsp',
'parch',
'fare']),
                                                                       ('cat',
Pipeline(steps=[('imputer',
                 SimpleImputer(strategy='most_frequent')),
                ('onehot',
                 OneHotEncoder(handle_unknown='ignore'))]),
                                                                        ['sex',
'embarked',
'class',
                                                                         'who',
'adult_male',
'alone'])])),
                                       ('model', LogisticRegression())]),
             param_grid={'model__C': [0.01, 0.1, 1, 10],
                         'model__max_iter': [500, 1000],
                         'model__solver': ['liblinear', 'lbfgs']},
             scoring='f1')
```

0.6.4 Resultados de la búsqueda

Una vez finalizado el proceso, se pueden consultar:

- los mejores hiperparámetros encontrados
- el mejor valor medio de F1-score obtenido en validación cruzada

[ ]:
```
print("Mejores parámetros:", grid_search_logreg.best_params_)
print("Mejor F1-score en validación cruzada:", round(grid_search_logreg.
 ↪best_score_, 4))
```

```
Mejores parámetros: {'model__C': 10, 'model__max_iter': 500, 'model__solver':
'liblinear'}
Mejor F1-score en validación cruzada: 0.7541
```

0.6.5 Evaluación final sobre test

Después de seleccionar la mejor configuración, se utilizará el mejor modelo encontrado para generar predicciones sobre el conjunto `test`.

A continuación, se calcularán varias métricas de clasificación para evaluar su rendimiento.

[ ]:
```
best_model_logreg = grid_search_logreg.best_estimator_
y_test_predicted_logreg = best_model_logreg.predict(X_test)
```

[ ]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, confusion_matrix

accuracy_logreg = accuracy_score(y_test, y_test_predicted_logreg)
precision_logreg = precision_score(y_test, y_test_predicted_logreg)
recall_logreg = recall_score(y_test, y_test_predicted_logreg)
f1_logreg = f1_score(y_test, y_test_predicted_logreg)
cm_logreg = confusion_matrix(y_test, y_test_predicted_logreg)
```

[ ]:
```
print("Accuracy:", round(accuracy_logreg, 4))
print("Precision:", round(precision_logreg, 4))
print("Recall:", round(recall_logreg, 4))
print("F1-score:", round(f1_logreg, 4))
```

```
Accuracy: 0.8324
Precision: 0.8095
Recall: 0.7391
F1-score: 0.7727
```

0.6.6 Classification report

Además de las métricas anteriores, también puede mostrarse un resumen completo del rendimiento del modelo mediante `classification_report`.

Este informe recoge, para cada clase, los valores de **precision**, **recall**, **f1-score** y **support**.

[ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_test_predicted_logreg))
```

```
              precision    recall  f1-score   support

           0       0.84      0.89      0.87       110
```

```
           1       0.81      0.74      0.77        69

    accuracy                           0.83       179
   macro avg       0.83      0.82      0.82       179
weighted avg       0.83      0.83      0.83       179
```

0.6.7 Interpretación de los resultados del classification report

A partir del `classification_report`, puede observarse que el modelo ofrece un rendimiento general bastante razonable, con una **accuracy de 0.83**, lo que significa que clasifica correctamente aproximadamente el **83%** de los pasajeros del conjunto de prueba.

Analizando cada clase por separado:

- Para la clase **0** (pasajeros que **no sobrevivieron**), el modelo obtiene:
  - **precision = 0.84** → cuando el modelo predice que un pasajero no sobrevivió, acierta en un 84% de los casos
  - **recall = 0.89** → detecta correctamente el 89% de los pasajeros que realmente no sobrevivieron
  - **f1-score = 0.87** → combina precision y recall en una única medida, mostrando un comportamiento bastante sólido para esta clase
- Para la clase **1** (pasajeros que **sí sobrevivieron**), el modelo obtiene:
  - **precision = 0.81** → cuando el modelo predice que un pasajero sobrevivió, acierta en un 81% de los casos
  - **recall = 0.74** → detecta correctamente el 74% de los pasajeros que realmente sobrevivieron
  - **f1-score = 0.77** → el rendimiento sigue siendo razonable, aunque algo inferior al de la clase 0

Esto indica que el modelo clasifica mejor la clase **0** que la clase **1**. En particular, el valor de **recall** para la clase 1 es más bajo, lo que sugiere que el modelo está dejando sin identificar correctamente a una parte de los pasajeros que sí sobrevivieron.

Además:

- **macro avg** calcula la media simple entre ambas clases, dando el mismo peso a cada una
- **weighted avg** calcula una media ponderada según el número de ejemplos de cada clase

Como ambas medias son parecidas y cercanas a **0.83**, puede decirse que el modelo presenta un comportamiento general bastante equilibrado, aunque con algo más de dificultad para detectar correctamente la clase de los supervivientes.

En conjunto, estos resultados pueden considerarse **buenos como primera aproximación**, aunque todavía existe margen de mejora, especialmente en la capacidad del modelo para identificar correctamente a los pasajeros que sí sobrevivieron.

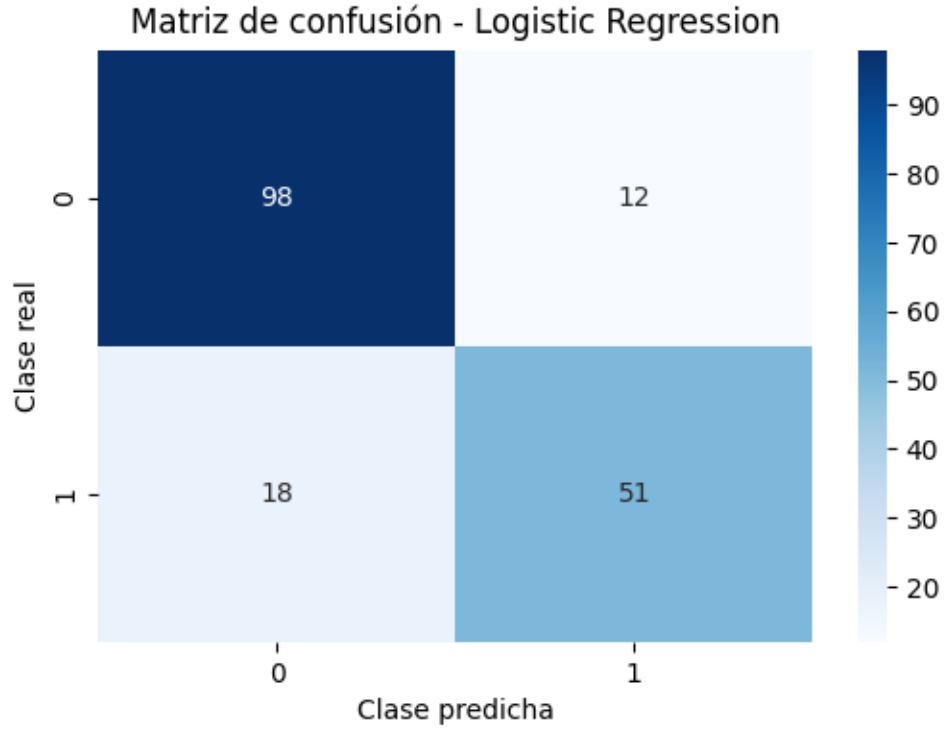
0.6.8 Matriz de confusión

La matriz de confusión permite observar cuántos ejemplos de cada clase se han clasificado correctamente y cuántos se han clasificado de forma incorrecta.

[ ]:
```
print("Matriz de confusión:")
print(cm_logreg)
```

```
Matriz de confusión:
[[98 12]
 [18 51]]
```

[ ]:
```
plt.figure(figsize=(6, 4))
sns.heatmap(
    cm_logreg,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=["0", "1"],
    yticklabels=["0", "1"]
)
plt.title("Matriz de confusión - Logistic Regression")
plt.xlabel("Clase predicha")
plt.ylabel("Clase real")
plt.show()
```



0.6.9 Interpretación básica del modelo

En el caso de la regresión logística, una forma sencilla de interpretar el modelo consiste en observar los **coeficientes** asociados a cada variable transformada.

En términos generales:

- coeficientes **positivos** favorecen la predicción de la clase `1`
- coeficientes **negativos** favorecen la predicción de la clase `0`
- cuanto mayor sea el valor absoluto del coeficiente, mayor será la influencia de esa variable en la decisión del modelo

[ ]:
```
feature_names_logreg = best_model_logreg.named_steps["preprocessor"].
 ↪get_feature_names_out()
coefficients_logreg = best_model_logreg.named_steps["model"].coef_[0]

coef_df_logreg = pd.DataFrame({
    "feature": feature_names_logreg,
    "coefficient": coefficients_logreg
})

coef_df_logreg["abs_coefficient"] = coef_df_logreg["coefficient"].abs()
coef_df_logreg = coef_df_logreg.sort_values(by="abs_coefficient",␣
 ↪ascending=False)

coef_df_logreg
```

[ ]:

	feature	coefficient	abs_coefficient
9	cat__class_First	1.073183	1.073183
11	cat__class_Third	-1.000457	1.000457
15	cat__adult_male_False	0.873135	0.873135
16	cat__adult_male_True	-0.760178	0.760178
13	cat__who_man	-0.760178	0.760178
1	num__sibsp	-0.601040	0.601040
12	cat__who_child	0.442484	0.442484
14	cat__who_woman	0.430651	0.430651
17	cat__alone_False	0.323565	0.323565
7	cat__embarked_Q	0.317304	0.317304
2	num__parch	-0.277040	0.277040
0	num__age	-0.272703	0.272703
8	cat__embarked_S	-0.263130	0.263130
18	cat__alone_True	-0.210608	0.210608
4	cat__sex_female	0.148831	0.148831
3	num__fare	0.117945	0.117945
6	cat__embarked_C	0.058783	0.058783
10	cat__class_Second	0.040231	0.040231
5	cat__sex_male	-0.035874	0.035874

A partir de los resultados obtenidos, pueden destacarse algunas ideas:

- `cat__class_First` presenta un coeficiente claramente positivo, lo que sugiere que pertenecer a **primera clase** aumenta la probabilidad de supervivencia.
- `cat__class_Third` presenta un coeficiente claramente negativo, lo que indica que pertenecer a **tercera clase** reduce la probabilidad de supervivencia.
- `cat__adult_male_False` tiene un coeficiente positivo, mientras que `cat__adult_male_True` presenta un coeficiente negativo. Esto sugiere que **no ser un varón adulto** favorece la supervivencia, mientras que **ser un varón adulto** la reduce.
- La variable `cat__who_man` también aparece con coeficiente negativo, mientras que `cat__who_child` y `cat__who_woman` presentan coeficientes positivos. Esto refuerza la idea de que el modelo asocia una mayor probabilidad de supervivencia a **mujeres y niños**, y una menor probabilidad a los **hombres adultos**.
- `num__sibsp` tiene un coeficiente negativo relativamente alto, lo que sugiere que viajar con más hermanos/as o cónyuges se asocia con una menor probabilidad de supervivencia en este modelo.
- `num__age` también presenta un coeficiente negativo, por lo que una mayor edad parece asociarse con una menor probabilidad de supervivencia.
- `num__fare` tiene un coeficiente positivo, aunque menor que otras variables, lo que sugiere que pagar una tarifa más alta se relaciona con una mayor probabilidad de supervivencia.
- En las variables de embarque, `cat__embarked_Q` aparece con coeficiente positivo y `cat__embarked_S` con coeficiente negativo, lo que indica que el puerto de embarque también influye en la decisión del modelo.

En conjunto, los coeficientes reflejan patrones bastante coherentes con el contexto histórico del Titanic: la clase del pasajero, el sexo, la edad y algunas características familiares parecen influir de forma importante en la probabilidad de supervivencia.

Aun así, conviene recordar que estos coeficientes se interpretan dentro del modelo y después del preprocesado aplicado. Por tanto, ofrecen una guía útil sobre las variables más influyentes, pero no deben entenderse como una relación causal directa.

0.7 7. Decision Tree + GridSearchCV

El segundo modelo que se entrenará será un **árbol de decisión**.

Al igual que en la sección anterior, el preprocesado y el modelo se integrarán en un único pipeline completo. Después, se utilizará `GridSearchCV` para probar distintas configuraciones del árbol y seleccionar automáticamente la mejor mediante validación cruzada.

De nuevo, la métrica principal que se utilizará será **F1-score**.

0.7.1 Definición del pipeline completo

A continuación, se construirá un pipeline que agrupe:

- el preprocesado definido en la sección 5
- el modelo de árbol de decisión

[ ]:
```
from sklearn.tree import DecisionTreeClassifier
```

[ ]:
```
full_pipeline_tree = Pipeline([
    ("preprocessor", preprocessor),
    ("model", DecisionTreeClassifier(random_state=42))
])
```

0.7.2 Definición de la búsqueda de hiperparámetros

Se probarán distintas configuraciones del árbol modificando algunos hiperparámetros habituales:

- `max_depth`, que controla la profundidad máxima del árbol
- `min_samples_split`, que indica el número mínimo de muestras necesarias para dividir un nodo
- `min_samples_leaf`, que indica el número mínimo de muestras necesarias en una hoja

[ ]:
```
param_grid_tree = {
    "model__max_depth": [3, 5, 7, 10, None],
    "model__min_samples_split": [2, 5, 10],
    "model__min_samples_leaf": [1, 2, 4]
}
```

0.7.3 Entrenamiento con GridSearchCV

A continuación, se aplicará `GridSearchCV` sobre el conjunto `train` para seleccionar automáticamente la mejor configuración del árbol mediante validación cruzada.

[ ]:
```
grid_search_tree = GridSearchCV(
    estimator=full_pipeline_tree,
    param_grid=param_grid_tree,
    scoring="f1",
    cv=5
)
```

[ ]:
```
grid_search_tree.fit(X_train, y_train)
```

[ ]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('preprocessor',
                                        ColumnTransformer(transformers=[('num',
Pipeline(steps=[('imputer',
                 SimpleImputer(strategy='median')),
                ('scaler',
                 StandardScaler())]),
                                                                        ['age',
'sibsp',
'parch',
'fare']),
                                                                       ('cat',
Pipeline(steps=[('imputer',
                 SimpleImputer(strategy='most_frequent')),
                ('onehot',
```

```
          OneHotEncoder(handle_unknown='ignore'))]),
                                                                    ['sex',
'embarked',
'class',
                                                                     'who',
'adult_male',
'alone'])])),
                                        ('model',
DecisionTreeClassifier(random_state=42))]),
             param_grid={'model__max_depth': [3, 5, 7, 10, None],
                         'model__min_samples_leaf': [1, 2, 4],
                         'model__min_samples_split': [2, 5, 10]},
             scoring='f1')
```

0.7.4 Resultados de la búsqueda

Una vez finalizado el proceso, se pueden consultar:

- los mejores hiperparámetros encontrados
- el mejor valor medio de F1-score obtenido en validación cruzada

[ ]:
```
print("Mejores parámetros:", grid_search_tree.best_params_)
print("Mejor F1-score en validación cruzada:", round(grid_search_tree.
↪best_score_, 4))
```

```
Mejores parámetros: {'model__max_depth': 3, 'model__min_samples_leaf': 4,
'model__min_samples_split': 2}
Mejor F1-score en validación cruzada: 0.7312
```

0.7.5 Evaluación final sobre test

Después de seleccionar la mejor configuración, se utilizará el mejor modelo encontrado para generar predicciones sobre el conjunto `test`.

A continuación, se calcularán varias métricas de clasificación para evaluar su rendimiento.

[ ]:
```
best_model_tree = grid_search_tree.best_estimator_
y_test_predicted_tree = best_model_tree.predict(X_test)
```

[ ]:
```
accuracy_tree = accuracy_score(y_test, y_test_predicted_tree)
precision_tree = precision_score(y_test, y_test_predicted_tree)
recall_tree = recall_score(y_test, y_test_predicted_tree)
f1_tree = f1_score(y_test, y_test_predicted_tree)
cm_tree = confusion_matrix(y_test, y_test_predicted_tree)
```

[ ]:
```
print("Accuracy:", round(accuracy_tree, 4))
print("Precision:", round(precision_tree, 4))
print("Recall:", round(recall_tree, 4))
print("F1-score:", round(f1_tree, 4))
```

```
Accuracy: 0.8268
Precision: 0.7879
Recall: 0.7536
F1-score: 0.7704
```

0.7.6 Classification report

Además de las métricas anteriores, también puede mostrarse un resumen completo del rendimiento del modelo mediante `classification_report`.

Este informe recoge, para cada clase, los valores de **precision**, **recall**, **f1-score** y **support**.

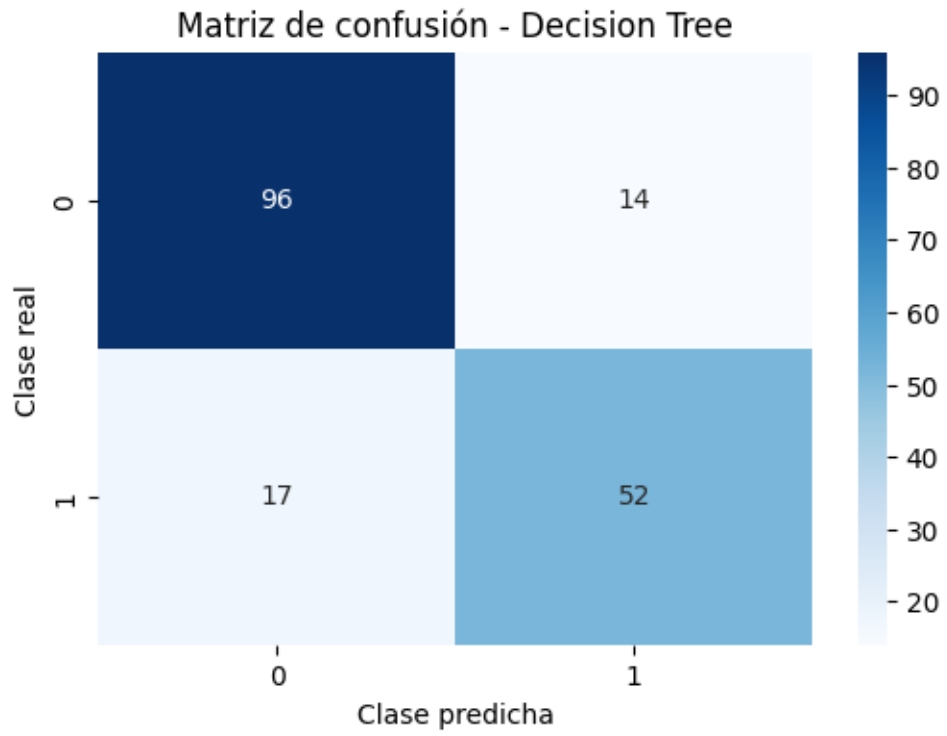
[ ]:
```
print(classification_report(y_test, y_test_predicted_tree))
```

```
              precision    recall  f1-score   support

           0       0.85      0.87      0.86       110
           1       0.79      0.75      0.77        69

    accuracy                           0.83       179
   macro avg       0.82      0.81      0.82       179
weighted avg       0.83      0.83      0.83       179
```

0.7.7 Matriz de confusión

La matriz de confusión permite observar cuántos ejemplos de cada clase se han clasificado correctamente y cuántos se han clasificado de forma incorrecta.

[ ]:
```
plt.figure(figsize=(6, 4))
sns.heatmap(
    cm_tree,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=["0", "1"],
    yticklabels=["0", "1"]
)
plt.title("Matriz de confusión - Decision Tree")
plt.xlabel("Clase predicha")
plt.ylabel("Clase real")
plt.show()
```



0.7.8 Interpretación básica del modelo

En los árboles de decisión, una forma habitual de interpretar el modelo consiste en observar la **importancia de las variables**.

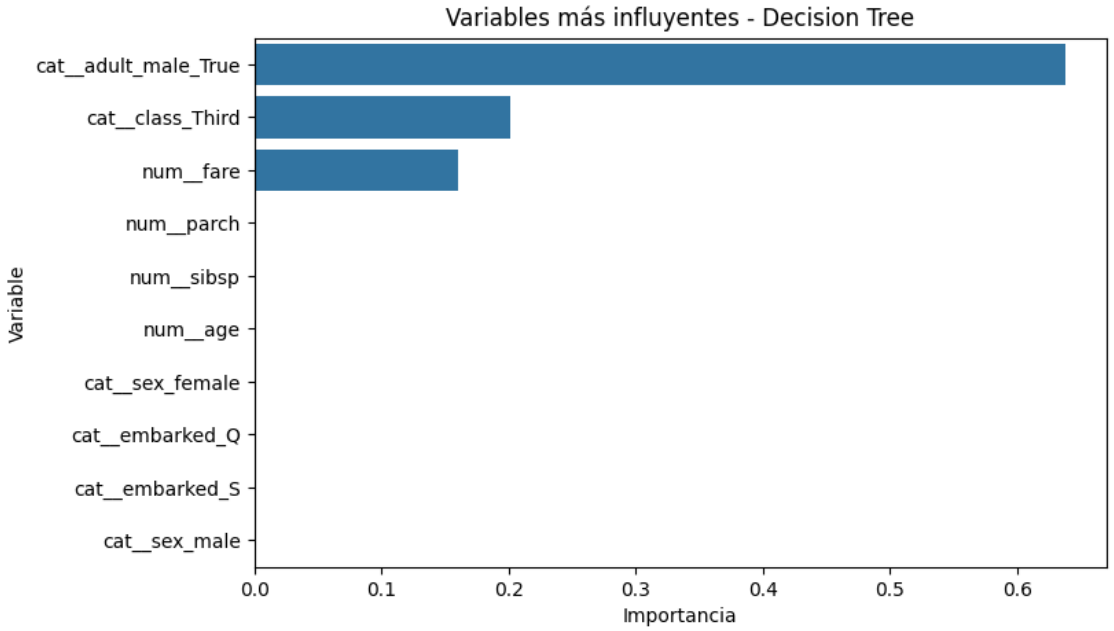
Esta medida indica cuánto contribuye cada variable a las divisiones del árbol y, por tanto, a la toma de decisiones del modelo.

[ ]:
```
feature_names_tree = best_model_tree.named_steps["preprocessor"].
 ↪get_feature_names_out()
importances_tree = best_model_tree.named_steps["model"].feature_importances_

importance_df_tree = pd.DataFrame({
    "feature": feature_names_tree,
    "importance": importances_tree
})

importance_df_tree = importance_df_tree.sort_values(by="importance",␣
 ↪ascending=False)

importance_df_tree
```

[ ]:

	feature	importance
16	cat__adult_male_True	0.638219
11	cat__class_Third	0.201113
3	num__fare	0.160668
2	num__parch	0.000000
1	num__sibsp	0.000000
0	num__age	0.000000
4	cat__sex_female	0.000000
7	cat__embarked_Q	0.000000
8	cat__embarked_S	0.000000
5	cat__sex_male	0.000000
6	cat__embarked_C	0.000000
10	cat__class_Second	0.000000
9	cat__class_First	0.000000
13	cat__who_man	0.000000
12	cat__who_child	0.000000
14	cat__who_woman	0.000000
15	cat__adult_male_False	0.000000
17	cat__alone_False	0.000000
18	cat__alone_True	0.000000

[ ]:
```
top_importance_tree = importance_df_tree.head(10)

plt.figure(figsize=(8, 5))
sns.barplot(data=top_importance_tree, x="importance", y="feature")
plt.title("Variables más influyentes - Decision Tree")
plt.xlabel("Importancia")
plt.ylabel("Variable")
plt.show()
```



0.8 8. Random Forest + GridSearchCV

El tercer modelo que se entrenará será un **random forest**.

Este algoritmo combina múltiples árboles de decisión y suele ofrecer un comportamiento más robusto que un único árbol, ya que reduce en parte el riesgo de sobreajuste y suele generalizar mejor.

Al igual que en las secciones anteriores, el preprocesado y el modelo se integrarán en un único pipeline completo. Después, se utilizará `GridSearchCV` para probar distintas configuraciones del modelo y seleccionar automáticamente la mejor mediante validación cruzada.

De nuevo, la métrica principal que se utilizará será **F1-score**.

[ ]:
```
from sklearn.ensemble import RandomForestClassifier
```

0.8.1 Definición del pipeline completo

A continuación, se construirá un pipeline que agrupe:

- el preprocesado definido en la sección 5
- el modelo de random forest

[ ]:
```
full_pipeline_rf = Pipeline([
    ("preprocessor", preprocessor),
    ("model", RandomForestClassifier(random_state=42))
])
```

0.8.2 Definición de la búsqueda de hiperparámetros

Se probarán distintas configuraciones del random forest modificando algunos hiperparámetros habituales:

- `n_estimators`, que indica el número de árboles del bosque
- `max_depth`, que controla la profundidad máxima de cada árbol
- `min_samples_split`, que indica el número mínimo de muestras necesarias para dividir un nodo

[ ]:
```
param_grid_rf = {
    "model__n_estimators": [50, 100, 200],
    "model__max_depth": [3, 5, 10, None],
    "model__min_samples_split": [2, 5, 10]
}
```

0.8.3 Entrenamiento con GridSearchCV

A continuación, se aplicará `GridSearchCV` sobre el conjunto `train` para seleccionar automáticamente la mejor configuración del random forest mediante validación cruzada.

[ ]:
```
grid_search_rf = GridSearchCV(
    estimator=full_pipeline_rf,
    param_grid=param_grid_rf,
    scoring="f1",
    cv=5
)
```

[ ]:
```
grid_search_rf.fit(X_train, y_train)
```

[ ]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('preprocessor',
                                        ColumnTransformer(transformers=[('num',
Pipeline(steps=[('imputer',
                 SimpleImputer(strategy='median')),
                ('scaler',
                 StandardScaler())]),
                                                                        ['age',
'sibsp',
'parch',
'fare']),
                                                                       ('cat',
Pipeline(steps=[('imputer',
                 SimpleImputer(strategy='most_frequent')),
                ('onehot',
                 OneHotEncoder(handle_unknown='ignore'))]),
                                                                        ['sex',
'embarked',
'class',
                                                                         'who',
'adult_male',
'alone'])])),
                                       ('model',
RandomForestClassifier(random_state=42))]),
             param_grid={'model__max_depth': [3, 5, 10, None],
                         'model__min_samples_split': [2, 5, 10],
                         'model__n_estimators': [50, 100, 200]},
             scoring='f1')
```

0.8.4 Resultados de la búsqueda

Una vez finalizado el proceso, se pueden consultar:

- los mejores hiperparámetros encontrados
- el mejor valor medio de F1-score obtenido en validación cruzada

[ ]:
```
print("Mejores parámetros:", grid_search_rf.best_params_)
print("Mejor F1-score en validación cruzada:", round(grid_search_rf.
 ↪best_score_, 4))
```

```
Mejores parámetros: {'model__max_depth': 10, 'model__min_samples_split': 10,
'model__n_estimators': 100}
Mejor F1-score en validación cruzada: 0.7596
```

0.8.5 Evaluación final sobre test

Después de seleccionar la mejor configuración, se utilizará el mejor modelo encontrado para generar predicciones sobre el conjunto `test`.

A continuación, se calcularán varias métricas de clasificación para evaluar su rendimiento.

[ ]:
```
best_model_rf = grid_search_rf.best_estimator_
y_test_predicted_rf = best_model_rf.predict(X_test)
```

[ ]:
```
accuracy_rf = accuracy_score(y_test, y_test_predicted_rf)
precision_rf = precision_score(y_test, y_test_predicted_rf)
recall_rf = recall_score(y_test, y_test_predicted_rf)
f1_rf = f1_score(y_test, y_test_predicted_rf)
cm_rf = confusion_matrix(y_test, y_test_predicted_rf)
```

[ ]:
```
print("Accuracy:", round(accuracy_rf, 4))
print("Precision:", round(precision_rf, 4))
print("Recall:", round(recall_rf, 4))
print("F1-score:", round(f1_rf, 4))
```

```
Accuracy: 0.8212
Precision: 0.8136
Recall: 0.6957
F1-score: 0.75
```

0.8.6 Classification report

Además de las métricas anteriores, también puede mostrarse un resumen completo del rendimiento del modelo mediante `classification_report`.

Este informe recoge, para cada clase, los valores de **precision**, **recall**, **f1-score** y **support**.

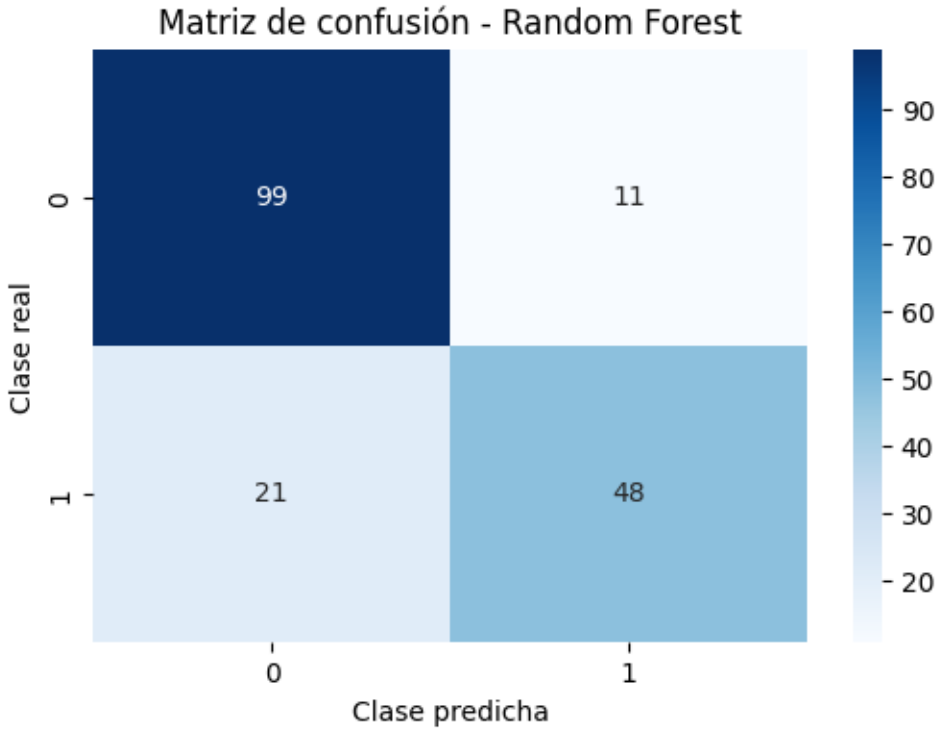
[ ]:
```
print(classification_report(y_test, y_test_predicted_rf))
```

```
              precision    recall  f1-score   support

           0       0.82      0.90      0.86       110
           1       0.81      0.70      0.75        69

    accuracy                           0.82       179
   macro avg       0.82      0.80      0.81       179
weighted avg       0.82      0.82      0.82       179
```

0.8.7 Matriz de confusión

La matriz de confusión permite observar cuántos ejemplos de cada clase se han clasificado correctamente y cuántos se han clasificado de forma incorrecta.

[ ]:
```
plt.figure(figsize=(6, 4))
sns.heatmap(
    cm_rf,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=["0", "1"],
    yticklabels=["0", "1"]
)
plt.title("Matriz de confusión - Random Forest")
plt.xlabel("Clase predicha")
plt.ylabel("Clase real")
plt.show()
```



0.8.8 Interpretación básica del modelo

En el caso de random forest, también puede analizarse la **importancia de las variables**.

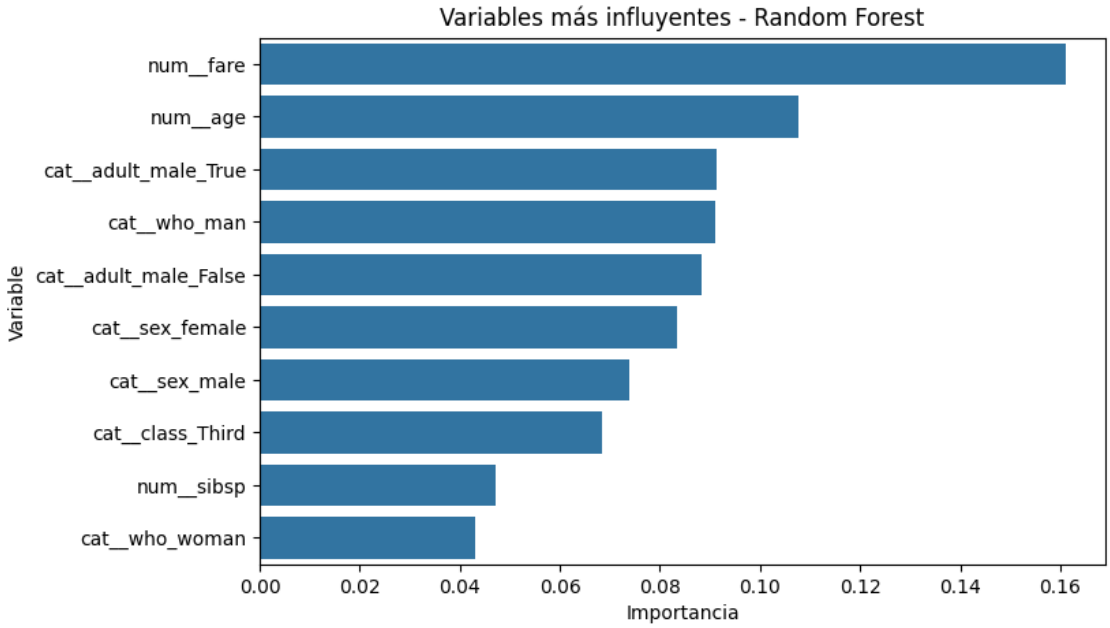
Como este modelo combina múltiples árboles de decisión, esta medida suele resultar más estable que en un único árbol y permite identificar qué variables han contribuido más al proceso de clasificación.

[ ]:
```
feature_names_rf = best_model_rf.named_steps["preprocessor"].
 ↪get_feature_names_out()
importances_rf = best_model_rf.named_steps["model"].feature_importances_

importance_df_rf = pd.DataFrame({
    "feature": feature_names_rf,
    "importance": importances_rf
})

importance_df_rf = importance_df_rf.sort_values(by="importance",␣
 ↪ascending=False)

importance_df_rf
```

[ ]:

	feature	importance
3	num__fare	0.160968
0	num__age	0.107546
16	cat__adult_male_True	0.091346
13	cat__who_man	0.091157
15	cat__adult_male_False	0.088426
4	cat__sex_female	0.083510
5	cat__sex_male	0.073890
11	cat__class_Third	0.068319
1	num__sibsp	0.047062
14	cat__who_woman	0.042980
9	cat__class_First	0.037935
10	cat__class_Second	0.026977
2	num__parch	0.020745
8	cat__embarked_S	0.015764
6	cat__embarked_C	0.011373
17	cat__alone_False	0.009708
7	cat__embarked_Q	0.009242
12	cat__who_child	0.006657
18	cat__alone_True	0.006395

[ ]:
```
top_importance_rf = importance_df_rf.head(10)

plt.figure(figsize=(8, 5))
sns.barplot(data=top_importance_rf, x="importance", y="feature")
plt.title("Variables más influyentes - Random Forest")
plt.xlabel("Importancia")
plt.ylabel("Variable")
plt.show()
```



0.9 9. k-NN + GridSearchCV

El cuarto modelo que se entrenará será **k-Nearest Neighbors (k-NN)**.

Este algoritmo clasifica cada ejemplo en función de los ejemplos más cercanos del conjunto de entrenamiento. Por ello, la escala de las variables resulta especialmente importante, lo que hace que el preprocesado definido anteriormente tenga un papel clave en este modelo.

Al igual que en las secciones anteriores, el preprocesado y el modelo se integrarán en un único pipeline completo. Después, se utilizará `GridSearchCV` para probar distintas configuraciones del modelo y seleccionar automáticamente la mejor mediante validación cruzada.

De nuevo, la métrica principal que se utilizará será **F1-score**.

[ ]:
```
from sklearn.neighbors import KNeighborsClassifier
```

0.9.1 Definición del pipeline completo

A continuación, se construirá un pipeline que agrupe:

- el preprocesado definido en la sección 5
- el modelo k-NN

[ ]:
```
full_pipeline_knn = Pipeline([
    ("preprocessor", preprocessor),
    ("model", KNeighborsClassifier())
])
```

0.9.2 Definición de la búsqueda de hiperparámetros

Se probarán distintas configuraciones del modelo modificando algunos hiperparámetros habituales:

- `n_neighbors`, que indica cuántos vecinos se tendrán en cuenta
- `weights`, que permite decidir si todos los vecinos pesan igual o si los más cercanos tienen mayor influencia
- `p`, que determina el tipo de distancia utilizada

[ ]:
```
param_grid_knn = {
    "model__n_neighbors": [3, 5, 7, 9, 11],
    "model__weights": ["uniform", "distance"],
    "model__p": [1, 2]
}
```

0.9.3 Entrenamiento con GridSearchCV

A continuación, se aplicará `GridSearchCV` sobre el conjunto `train` para seleccionar automáticamente la mejor configuración del modelo k-NN mediante validación cruzada.

[ ]:
```
grid_search_knn = GridSearchCV(
    estimator=full_pipeline_knn,
    param_grid=param_grid_knn,
    scoring="f1",
    cv=5
)
```

[ ]:
```
grid_search_knn.fit(X_train, y_train)
```

[ ]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('preprocessor',
                                        ColumnTransformer(transformers=[('num',
Pipeline(steps=[('imputer',
                 SimpleImputer(strategy='median')),
                ('scaler',
                 StandardScaler())]),
                                                                        ['age',
'sibsp',
'parch',
'fare']),
                                                                       ('cat',
Pipeline(steps=[('imputer',
                 SimpleImputer(strategy='most_frequent')),
                ('onehot',
                 OneHotEncoder(handle_unknown='ignore'))]),
                                                                        ['sex',
'embarked',
'class',
                                                                         'who',
```

```
'adult_male',
'alone'])])),
                                        ('model', KNeighborsClassifier())]),
             param_grid={'model__n_neighbors': [3, 5, 7, 9, 11],
                         'model__p': [1, 2],
                         'model__weights': ['uniform', 'distance']},
             scoring='f1')
```

0.9.4 Resultados de la búsqueda

Una vez finalizado el proceso, se pueden consultar:

- los mejores hiperparámetros encontrados
- el mejor valor medio de F1-score obtenido en validación cruzada

[ ]:
```
print("Mejores parámetros:", grid_search_knn.best_params_)
print("Mejor F1-score en validación cruzada:", round(grid_search_knn.
  ↪best_score_, 4))
```

```
Mejores parámetros: {'model__n_neighbors': 5, 'model__p': 1, 'model__weights':
'uniform'}
Mejor F1-score en validación cruzada: 0.7474
```

0.9.5 Evaluación final sobre test

Después de seleccionar la mejor configuración, se utilizará el mejor modelo encontrado para generar predicciones sobre el conjunto `test`.

A continuación, se calcularán varias métricas de clasificación para evaluar su rendimiento.

[ ]:
```
best_model_knn = grid_search_knn.best_estimator_
y_test_predicted_knn = best_model_knn.predict(X_test)
```

[ ]:
```
accuracy_knn = accuracy_score(y_test, y_test_predicted_knn)
precision_knn = precision_score(y_test, y_test_predicted_knn)
recall_knn = recall_score(y_test, y_test_predicted_knn)
f1_knn = f1_score(y_test, y_test_predicted_knn)
cm_knn = confusion_matrix(y_test, y_test_predicted_knn)
```

[ ]:
```
print("Accuracy:", round(accuracy_knn, 4))
print("Precision:", round(precision_knn, 4))
print("Recall:", round(recall_knn, 4))
print("F1-score:", round(f1_knn, 4))
```

```
Accuracy: 0.8268
Precision: 0.7969
Recall: 0.7391
F1-score: 0.7669
```

0.9.6 Classification report

Además de las métricas anteriores, también puede mostrarse un resumen completo del rendimiento del modelo mediante `classification_report`.

Este informe recoge, para cada clase, los valores de **precision**, **recall**, **f1-score** y **support**.

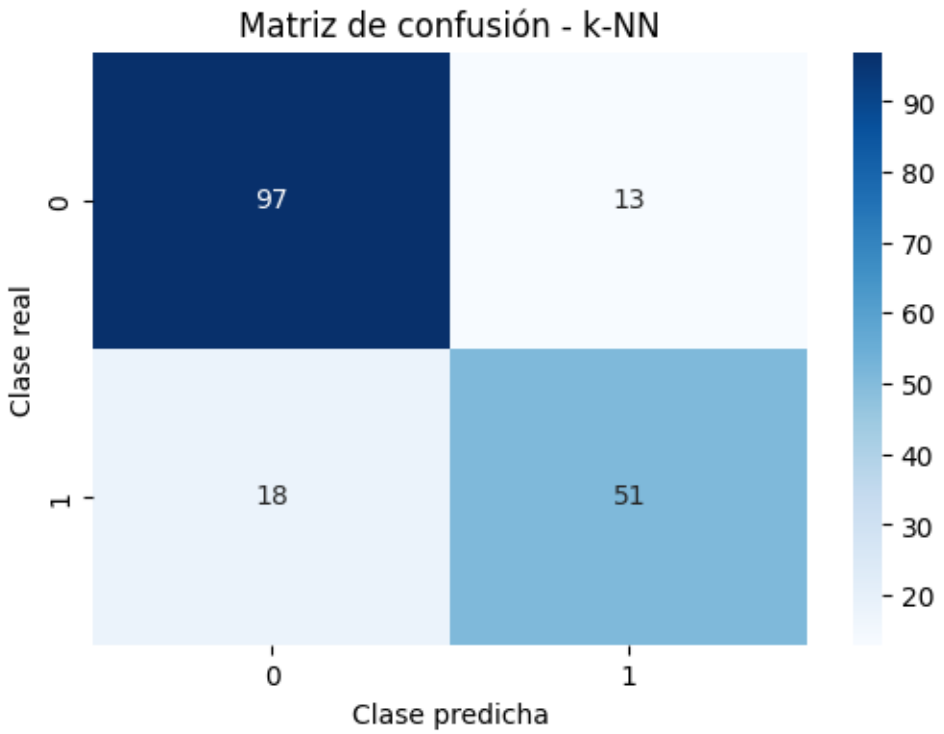
[ ]:
```
print(classification_report(y_test, y_test_predicted_knn))
```

```
              precision    recall  f1-score   support

           0       0.84      0.88      0.86       110
           1       0.80      0.74      0.77        69

    accuracy                           0.83       179
   macro avg       0.82      0.81      0.81       179
weighted avg       0.83      0.83      0.83       179
```

0.9.7 Matriz de confusión

La matriz de confusión permite observar cuántos ejemplos de cada clase se han clasificado correctamente y cuántos se han clasificado de forma incorrecta.

[ ]:
```
plt.figure(figsize=(6, 4))
sns.heatmap(
    cm_knn,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=["0", "1"],
    yticklabels=["0", "1"]
)
plt.title("Matriz de confusión - k-NN")
plt.xlabel("Clase predicha")
plt.ylabel("Clase real")
plt.show()
```



0.10 10. Comparación final de modelos

Una vez entrenados y evaluados los cuatro modelos de clasificación, conviene comparar sus resultados para identificar cuál ofrece el mejor comportamiento sobre el conjunto `test`.

En esta práctica se compararán las siguientes métricas:

- **accuracy**
- **precision**
- **recall**
- **f1-score**

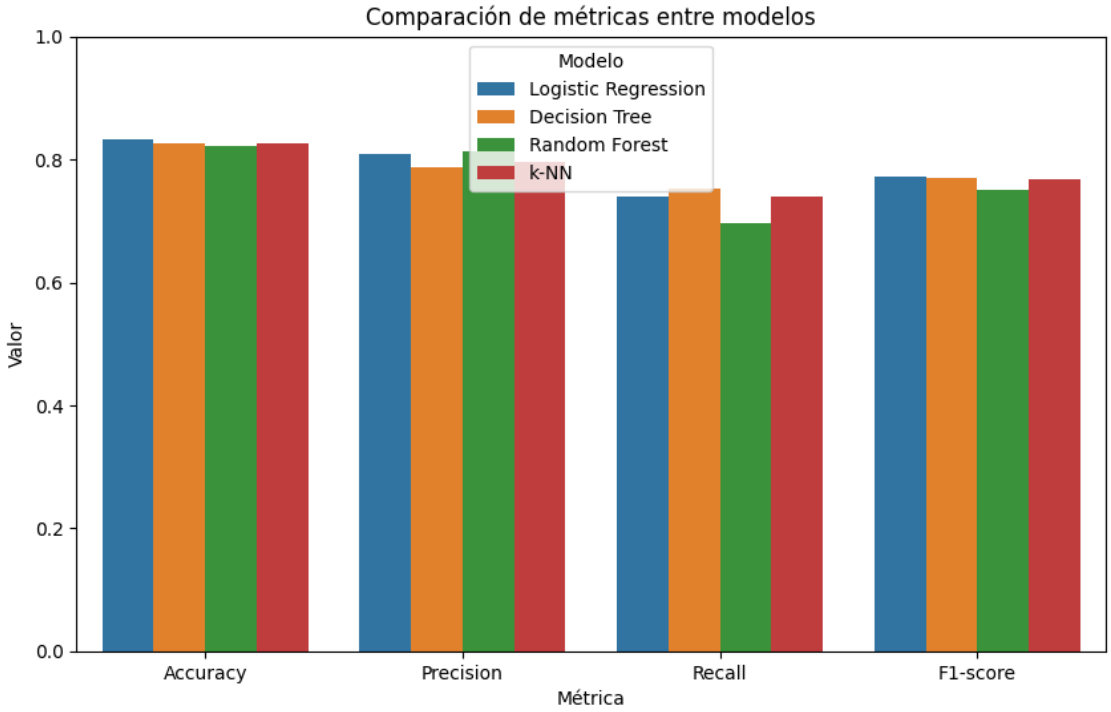
La comparación permitirá analizar no solo qué modelo obtiene mejores resultados globales, sino también si existen diferencias importantes entre ellos en función de la métrica considerada.

[ ]:
```
results_df = pd.DataFrame({
    "Modelo": ["Logistic Regression", "Decision Tree", "Random Forest", "k-NN"],
    "Accuracy": [accuracy_logreg, accuracy_tree, accuracy_rf, accuracy_knn],
    "Precision": [precision_logreg, precision_tree, precision_rf,␣
 ↪precision_knn],
    "Recall": [recall_logreg, recall_tree, recall_rf, recall_knn],
    "F1-score": [f1_logreg, f1_tree, f1_rf, f1_knn]
}).round(4)

results_df
```

[ ]:

	Modelo	Accuracy	Precision	Recall	F1-score
0	Logistic Regression	0.8324	0.8095	0.7391	0.7727
1	Decision Tree	0.8268	0.7879	0.7536	0.7704
2	Random Forest	0.8212	0.8136	0.6957	0.7500
3	k-NN	0.8268	0.7969	0.7391	0.7669

[ ]:
```
results_df_melted = results_df.melt(id_vars="Modelo", var_name="Métrica",␣
↪value_name="Valor")

plt.figure(figsize=(10, 6))
sns.barplot(data=results_df_melted, x="Métrica", y="Valor", hue="Modelo")
plt.title("Comparación de métricas entre modelos")
plt.ylim(0, 1)
plt.show()
```



0.10.1 Interpretación comparativa de los modelos

Los cuatro modelos presentan un rendimiento bastante parecido en el conjunto de prueba. Todas las métricas se mueven en valores próximos, lo que indica que, para este problema y con este preprocesado, ninguno de los algoritmos ofrece una ventaja muy grande sobre los demás.

Logistic Regression Es el modelo con mejores resultados globales en esta comparación:

- **Accuracy = 0.8324**
- **Precision = 0.8095**

- **Recall = 0.7391**
- **F1-score = 0.7727**

Esto sugiere que la regresión logística ofrece el comportamiento más equilibrado entre precisión y recall, y además obtiene el mejor F1-score. Por tanto, puede considerarse una opción especialmente interesante si se busca un modelo sencillo, interpretable y con buen rendimiento general.

Decision Tree El árbol de decisión ofrece resultados también competitivos:

- **Accuracy = 0.8268**
- **Precision = 0.7879**
- **Recall = 0.7536**
- **F1-score = 0.7704**

Aunque sus métricas globales son ligeramente inferiores a las de la regresión logística, destaca por obtener el **mayor recall**, lo que significa que detecta algo mejor la clase positiva (`survived = 1`). A cambio, pierde algo de precisión.

Random Forest El random forest no mejora al árbol de decisión en este caso:

- **Accuracy = 0.8212**
- **Precision = 0.8136**
- **Recall = 0.6957**
- **F1-score = 0.7500**

Su **precision** es la más alta de los cuatro modelos, lo que significa que cuando predice supervivencia suele acertar con bastante frecuencia. Sin embargo, su **recall** es el más bajo, por lo que deja escapar más pasajeros supervivientes reales que los demás modelos. En conjunto, esto hace que su F1-score sea el menor de los cuatro.

k-NN El modelo k-NN también presenta un rendimiento razonable:

- **Accuracy = 0.8268**
- **Precision = 0.7969**
- **Recall = 0.7391**
- **F1-score = 0.7669**

Sus resultados quedan muy próximos a los de la regresión logística y el árbol de decisión, aunque ligeramente por debajo en F1-score.

0.10.2 Conclusión general

Los resultados muestran que los modelos se comportan de forma bastante parecida. Aun así, pueden extraerse algunas conclusiones:

- **Logistic Regression** parece la mejor opción global en esta práctica, ya que obtiene el mejor equilibrio entre métricas y además es un modelo fácil de interpretar.
- **Decision Tree** puede ser interesante si se prioriza algo más el **recall**.
- **Random Forest** ofrece la mejor **precision**, pero sacrifica recall.
- **k-NN** se mantiene competitivo, aunque no supera claramente a la regresión logística.

En conjunto, como las diferencias son pequeñas, la elección final también puede apoyarse en criterios como la **interpretabilidad**, la **simplicidad** del modelo o el tipo de error que interese más reducir.