Preprocesado de características por lotes

En clase hemos visto como construir un pipeline de preprocesado de datos mediante la clase de Scikit-learn `ColumnTransformer`. En dicho ejemplo, definíamos la transformación a aplicar a cada variable de forma manual. Sin embargo, esto no es práctico cuando trabajamos con conjuntos de datos que tienen decenas o cientos de características.

En este ejemplo veremos, por un lado, como aplicar operaciones de preprocesado a grupos de variables, según su tipo (numéricas o categóricas) y si tienen o no valores perdidos. También veremos como recuperar los nombres de columnas tras aplicar la codificación one-hot.

En primer lugar, vamos a recuperar el ejemplo visto en clase.

Ejemplo práctica 7

In [1]:
```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import KBinsDiscretizer, MinMaxScaler, OneHotEncoder, O
```

In [2]:
```
df = pd.read_csv('./titanic_feat_eng.csv')
df
```

Out[2]:

	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	emb
0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	
1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.5500	C22 C26	
2	1	0	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.5500	C22 C26	
3	1	0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1	2	113781	151.5500	C22 C26	
4	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1	2	113781	151.5500	C22 C26	
...	...	...	...	...	...	...	...	...	...	...	...
1304	3	0	Zabour, Miss. Hileni	female	14.5000	1	0	2665	14.4542	NaN	
1305	3	0	Zabour, Miss. Thamine	female	NaN	1	0	2665	14.4542	NaN	
1306	3	0	Zakarian, Mr. Mapriededer	male	26.5000	0	0	2656	7.2250	NaN	
1307	3	0	Zakarian, Mr. Ortin	male	27.0000	0	0	2670	7.2250	NaN	
1308	3	0	Zimmerman, Mr. Leo	male	29.0000	0	0	315082	7.8750	NaN	

1309 rows × 19 columns

In [3]:
```
impohe = Pipeline([('si3', SimpleImputer(strategy='most_frequent')),
                   ('onehot', OneHotEncoder())])

impdisc_age = Pipeline([("si1", SimpleImputer(strategy='mean')),
                   ('disc1', KBinsDiscretizer(8, strategy='uniform', encode='ordina

impdisc_fare = Pipeline([("si2", SimpleImputer(strategy='median')),
                   ('disc2', KBinsDiscretizer(6, strategy='quantile', encode='ordi

ct = ColumnTransformer([("ohe",OneHotEncoder(),['sex', 'deck', 'title']),
                        ("si1",SimpleImputer(strategy='mean'),['age']), # age
                        ("disc_age",impdisc_age,['age']), # age_range
                        ("si2",SimpleImputer(strategy='median'),['fare']), # fare
                        ("disc_fare",impdisc_fare,['fare']), # fare_range
                        ("impohe",impohe,['embarked']),
                        ("original",'passthrough',['pclass', 'sibsp', 'parch', 'is
```

```
npdata = ct.fit_transform(df)
npdata
```

Out[3]:
```
array([[1., 0., 0., ..., 0., 1., 1.],
       [0., 1., 0., ..., 3., 0., 1.],
       [1., 0., 0., ..., 3., 0., 0.],
       ...,
       [0., 1., 0., ..., 0., 1., 0.],
       [0., 1., 0., ..., 0., 1., 0.],
       [0., 1., 0., ..., 0., 1., 0.]])
```

In [4]:
```
X_titanic = npdata[:,:-1]
X_titanic[0,:]
```

Out[4]:
```
array([  1.    ,   0.    ,   0.    ,   1.    ,   0.    ,   0.    ,
         0.    ,   0.    ,   0.    ,   0.    ,   0.    ,   0.    ,
         0.    ,   0.    ,   0.    ,   0.    ,   0.    ,   0.    ,
         0.    ,   0.    ,   1.    ,   0.    ,   0.    ,   0.    ,
         0.    ,   0.    ,   0.    ,   0.    ,   0.    ,  29.    ,
         2.    , 211.3375,   5.    ,   0.    ,   0.    ,   1.    ,
         1.    ,   0.    ,   0.    ,   0.    ,   0.    ,   1.    ])
```

In [5]:
```
y_titanic = npdata[:,-1]
y_titanic
```

Out[5]:
```
array([1., 1., 0., ..., 0., 0., 0.])
```

In [6]:
```
scaler = MinMaxScaler(feature_range=(0, 1))
scaler.fit(X_titanic)
X_titanic_escalado = scaler.transform(X_titanic)
X_titanic_escalado
```

Out[6]:
```
array([[1. , 0. , 0. , ..., 0. , 0. , 1. ],
       [0. , 1. , 0. , ..., 0. , 0.3, 0. ],
       [1. , 0. , 0. , ..., 0. , 0.3, 0. ],
       ...,
       [0. , 1. , 0. , ..., 0. , 0. , 1. ],
       [0. , 1. , 0. , ..., 0. , 0. , 1. ],
       [0. , 1. , 0. , ..., 0. , 0. , 1. ]])
```

Ejemplo de preprocesado de características por lotes

Para trabajar por lotes de variables lo más sencillo es usar varios `ColumnTransformer`, cada uno para realizar una operación concreta (por ejemplo, imputación), en vez de usar un solo `ColumnTransformer` en el que se realiza todo.

1) Carga de datos y eliminación de columnas sobrantes

En primer lugar, cargamos los datos y descartamos aquellas características que sabemos que no queremos incluir en el conjunto de datos.

In [7]:
```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import KBinsDiscretizer, MinMaxScaler, OneHotEncoder, O
```

In [8]:
```
df1 = pd.read_csv('./titanic_feat_eng.csv')
df1
```

Out[8]:

	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	emb
0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	
1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.5500	C22 C26	
2	1	0	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.5500	C22 C26	
3	1	0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1	2	113781	151.5500	C22 C26	
4	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1	2	113781	151.5500	C22 C26	
...	...	...	...	...	...	...	...	...	...	...	...
1304	3	0	Zabour, Miss. Hileni	female	14.5000	1	0	2665	14.4542	NaN	
1305	3	0	Zabour, Miss. Thamine	female	NaN	1	0	2665	14.4542	NaN	
1306	3	0	Zakarian, Mr. Mapriededer	male	26.5000	0	0	2656	7.2250	NaN	
1307	3	0	Zakarian, Mr. Ortin	male	27.0000	0	0	2670	7.2250	NaN	
1308	3	0	Zimmerman, Mr. Leo	male	29.0000	0	0	315082	7.8750	NaN	

1309 rows × 19 columns

In [9]:
```
cols_to_delete = ['name', 'ticket', 'boat', 'body', 'home.dest', 'cabin']
df2 = df.drop(cols_to_delete, axis = 1)
df2
```

Out[9]:

	pclass	survived	sex	age	sibsp	parch	fare	embarked	title	is_married	de
0	1	1	female	29.0000	0	0	211.3375	S	Miss	0	
1	1	1	male	0.9167	1	2	151.5500	S	Master	0	
2	1	0	female	2.0000	1	2	151.5500	S	Miss	0	
3	1	0	male	30.0000	1	2	151.5500	S	Mr	0	
4	1	0	female	25.0000	1	2	151.5500	S	Mrs	1	
...	...	...	...	...	...	...	...	...	...	...	...
1304	3	0	female	14.5000	1	0	14.4542	C	Miss	0	
1305	3	0	female	NaN	1	0	14.4542	C	Miss	0	
1306	3	0	male	26.5000	0	0	7.2250	C	Mr	0	
1307	3	0	male	27.0000	0	0	7.2250	C	Mr	0	
1308	3	0	male	29.0000	0	0	7.8750	S	Mr	0	

1309 rows × 13 columns

2) Creación de listas de nombres de columnas

El siguiente paso es agrupar las variables por lotes, según las operaciones de preprocesado que queramos aplicar.

Todas las columnas

In [10]:
```
all_cols = df2.columns
all_cols
```

Out[10]:
```
Index(['pclass', 'survived', 'sex', 'age', 'sibsp', 'parch', 'fare',
       'embarked', 'title', 'is_married', 'deck', 'relatives', 'alone'],
      dtype='object')
```

Columnas numéricas

In [11]:
```
numeric_cols = df2.select_dtypes(include='number').columns
numeric_cols
```

Out[11]:
```
Index(['pclass', 'survived', 'age', 'sibsp', 'parch', 'fare', 'is_married',
       'relatives', 'alone'],
      dtype='object')
```

Columnas categóricas

In [12]:
```
categorical_cols = df2.select_dtypes(exclude='number').columns
categorical_cols
```

Out[12]:
```
Index(['sex', 'embarked', 'title', 'deck'], dtype='object')
```

Columnas con valores perdidos

In [13]:
```
cols_to_impute = df2.columns[df2.isnull().sum() > 0]
cols_to_impute
```

Out[13]:
```
Index(['age', 'fare', 'embarked'], dtype='object')
```

3) Imputación de valores perdidos

Vamos a realizar el primer paso de tratamiento de datos, que en este caso será la imputación de valores perdidos. Para esto vamos a crear nuevos grupos de variables, según el tipo de imputación a aplicar:

- `cols_to_impute_mean` : Variables a imputar por la media, es decir, variables numéricas con valores perdidos.
- `cols_to_impute_mode` : Variables a imputar por la moda, es decir, variables categóricas con valores perdidos.
- `cols_to_passthrought` : Variables que no requieren imputación, es decir, variables numéricas o categóricas sin valores perdidos.

In [14]:
```
cols_to_impute_mean = []
cols_to_impute_mode = []
cols_to_passthrought = []
for col in df2.columns:
    if col in cols_to_impute:
        if col in numeric_cols:
            cols_to_impute_mean.append(col)
        else:
            cols_to_impute_mode.append(col)
    else:
        cols_to_passthrought.append(col)
```

In [15]:
```
cols_to_impute_mean
```

Out[15]:
```
['age', 'fare']
```

In [16]:
```
cols_to_impute_mode
```

Out[16]:
```
['embarked']
```

In [17]:
```
cols_to_passthrought
```

Out[17]:
```
['pclass',
 'survived',
 'sex',
 'sibsp',
 'parch',
 'title',
 'is_married',
 'deck',
 'relatives',
 'alone']
```

A continuación definimos un `ColumnTransformer` que aplica la operación de imputación correspondiente a cada uno de los tres grupos de variables definidos.

Nótese que para realizar un tratamiento más especializado, por ejemplo, separando las variables numéricas en dos grupos, uno para ser imputado por la mediana y otro por la

media, debemos hacerlo a mano.

In [18]:
```
ct_impute = ColumnTransformer([("si_mean".format(col),SimpleImputer(strategy='mean
                                ("si_mode".format(col),SimpleImputer(strategy='most_
                                ("original",'passthrough',cols_to_passthrought)
                               ])
npdata1 = ct_impute.fit_transform(df2)
npdata1
```

Out[18]:
```
array([[29.0, 211.3375, 'S', ..., 'B', 0, 1],
       [0.9167, 151.55, 'S', ..., 'C', 3, 0],
       [2.0, 151.55, 'S', ..., 'C', 3, 0],
       ...,
       [26.5, 7.225, 'C', ..., 'U', 0, 1],
       [27.0, 7.225, 'C', ..., 'U', 0, 1],
       [29.0, 7.875, 'S', ..., 'U', 0, 1]], dtype=object)
```

Por último, vamos a reconstruir el `DataFrame` de Pandas. Para esto necesitamos obtener la lista de nombres de columnas en el mismo orden en que se encuentran en `npdata1`. Dicho orden es en el que se han definido las operaciones en el `ColumnTransformer`.

In [19]:
```
new_column_order = cols_to_impute_mean + cols_to_impute_mode + cols_to_passthrough
new_column_order
```

Out[19]:
```
['age',
 'fare',
 'embarked',
 'pclass',
 'survived',
 'sex',
 'sibsp',
 'parch',
 'title',
 'is_married',
 'deck',
 'relatives',
 'alone']
```

In [20]:
```
df3 = pd.DataFrame(npdata1, columns=new_column_order)
df3
```

Out[20]:

	age	fare	embarked	pclass	survived	sex	sibsp	parch	title	is_married
0	29.0	211.3375	S	1	1	female	0	0	Miss	0
1	0.9167	151.55	S	1	1	male	1	2	Master	0
2	2.0	151.55	S	1	0	female	1	2	Miss	0
3	30.0	151.55	S	1	0	male	1	2	Mr	0
4	25.0	151.55	S	1	0	female	1	2	Mrs	1
...	...	...	...	...	...	...	...	...	...	...
1304	14.5	14.4542	C	3	0	female	1	0	Miss	0
1305	29.881135	14.4542	C	3	0	female	1	0	Miss	0
1306	26.5	7.225	C	3	0	male	0	0	Mr	0
1307	27.0	7.225	C	3	0	male	0	0	Mr	0
1308	29.0	7.875	S	3	0	male	0	0	Mr	0

1309 rows × 13 columns

Vemos que ya no hay valores perdidos en nuestro conjunto de datos.

In [21]:
```
df3.isnull().sum()
```

Out[21]:
```
age           0
fare          0
embarked      0
pclass        0
survived      0
sex           0
sibsp         0
parch         0
title         0
is_married    0
deck          0
relatives     0
alone         0
dtype: int64
```

Importante: Nuestro conjunto de datos todavía contiene variables de tipo `string` (tipo `object` en Pandas y Numpy). Por otro lado, los objetos de Numpy (`npdata1` en nuestro caso) solo tienen un tipo de datos, al contrario que los `DataFrame` de Pandas que tienen un tipo de datos por columna. El resultado de esto es que el tipo de datos de `npdata1` vendrá dado por el tipo de datos más general de las columnas del DataFrame de origen (que en este caso será `object`).

Veamos primero el tipo de datos de `df2` :

In [22]:
```
df2.dtypes
```

Out[22]:
```
pclass          int64
survived        int64
sex            object
age           float64
sibsp           int64
parch           int64
fare          float64
embarked       object
title          object
is_married      int64
deck           object
relatives       int64
alone           int64
dtype: object
```

Veamos ahora el tipo de datos de `df3`, el cual hemos construido a partir de `npdata1`.

In [23]:
```
df3.dtypes
```

Out[23]:
```
age           object
fare          object
embarked      object
pclass        object
survived      object
sex           object
sibsp         object
parch         object
title         object
is_married    object
deck          object
relatives     object
alone         object
dtype: object
```

Vamos a volver a convertir a tipo numérico las columnas que originalmente lo eran. Para evitar conflictos entre enteros y decimales, usaremos el tipo numérico más general, es decir, `float`.

In [24]:
```
df3[numeric_cols] = df3[numeric_cols].astype(float)
df3
```

Out[24]:

	age	fare	embarked	pclass	survived	sex	sibsp	parch	title	is_married
0	29.000000	211.3375	S	1.0	1.0	female	0.0	0.0	Miss	0.0
1	0.916700	151.5500	S	1.0	1.0	male	1.0	2.0	Master	0.0
2	2.000000	151.5500	S	1.0	0.0	female	1.0	2.0	Miss	0.0
3	30.000000	151.5500	S	1.0	0.0	male	1.0	2.0	Mr	0.0
4	25.000000	151.5500	S	1.0	0.0	female	1.0	2.0	Mrs	1.0
...	...	...	...	...	...	...	...	...	...	...
1304	14.500000	14.4542	C	3.0	0.0	female	1.0	0.0	Miss	0.0
1305	29.881135	14.4542	C	3.0	0.0	female	1.0	0.0	Miss	0.0
1306	26.500000	7.2250	C	3.0	0.0	male	0.0	0.0	Mr	0.0
1307	27.000000	7.2250	C	3.0	0.0	male	0.0	0.0	Mr	0.0
1308	29.000000	7.8750	S	3.0	0.0	male	0.0	0.0	Mr	0.0

1309 rows × 13 columns

In [25]:
```
df3.dtypes
```

Out[25]:
```
age           float64
fare          float64
embarked       object
pclass        float64
survived      float64
sex            object
sibsp         float64
parch         float64
title          object
is_married    float64
deck           object
relatives     float64
alone         float64
dtype: object
```

3) Tratamiento de variables numéricas: Discretización

Supongamos además que queremos discretizar algunas variables. Dado que normalmente vamos a discretizar solo algunas variables y que los parámetros de discretización suelen ser diferentes (número de bins y estrategia) este paso lo vamos a hacer a mano. Sin embargo, si se desea discretizar un gran número de variables numéricas, aplicando el mismo criterio a todas, también podemos pasar una lista.

En este ejemplo, tal y como hicimos en la práctica 7, vamos a crear copias discretizadas de las variables `age` y `fare`, es decir, mantenemos las originales.

In [26]:
```
ct_discretize = ColumnTransformer([("original",'passthrough',df3.columns),
                                   ('disc1', KBinsDiscretizer(8, strategy='uniform
                                   ('disc2', KBinsDiscretizer(6, strategy='quantil
                                  ])
npdata2 = ct_discretize.fit_transform(df3)
npdata2
```

Out[26]:
```
array([[29.0, 211.3375, 'S', ..., 1.0, 2.0, 5.0],
       [0.9167, 151.55, 'S', ..., 0.0, 0.0, 5.0],
       [2.0, 151.55, 'S', ..., 0.0, 0.0, 5.0],
       ...,
       [26.5, 7.225, 'C', ..., 1.0, 2.0, 0.0],
       [27.0, 7.225, 'C', ..., 1.0, 2.0, 0.0],
       [29.0, 7.875, 'S', ..., 1.0, 2.0, 1.0]], dtype=object)
```

Volvemos a generar la lista de nombres de columnas, en el orden en que han sido procesadas, para reconstruir el DataFrame de Pandas. Nótese que hemos mantenido todas las columnas anteriores y creado dos nuevas.

In [27]:
```
new_column_order = list(df3.columns) + ['age_range', 'fare_range']
new_column_order
```

Out[27]:
```
['age',
 'fare',
 'embarked',
 'pclass',
 'survived',
 'sex',
 'sibsp',
 'parch',
 'title',
 'is_married',
 'deck',
 'relatives',
 'alone',
 'age_range',
 'fare_range']
```

In [28]:
```
df4 = pd.DataFrame(npdata2, columns=new_column_order)
df4
```

Out[28]:

	age	fare	embarked	pclass	survived	sex	sibsp	parch	title	is_married
0	29.0	211.3375	S	1.0	1.0	female	0.0	0.0	Miss	0.0
1	0.9167	151.55	S	1.0	1.0	male	1.0	2.0	Master	0.0
2	2.0	151.55	S	1.0	0.0	female	1.0	2.0	Miss	0.0
3	30.0	151.55	S	1.0	0.0	male	1.0	2.0	Mr	0.0
4	25.0	151.55	S	1.0	0.0	female	1.0	2.0	Mrs	1.0
...	...	...	...	...	...	...	...	...	...	...
1304	14.5	14.4542	C	3.0	0.0	female	1.0	0.0	Miss	0.0
1305	29.881135	14.4542	C	3.0	0.0	female	1.0	0.0	Miss	0.0
1306	26.5	7.225	C	3.0	0.0	male	0.0	0.0	Mr	0.0
1307	27.0	7.225	C	3.0	0.0	male	0.0	0.0	Mr	0.0
1308	29.0	7.875	S	3.0	0.0	male	0.0	0.0	Mr	0.0

1309 rows × 15 columns

In [29]:
```
df4.dtypes
```

Out[29]:
```
age           object
fare          object
embarked      object
pclass        object
survived      object
sex           object
sibsp         object
parch         object
title         object
is_married    object
deck          object
relatives     object
alone         object
age_range     object
fare_range    object
dtype: object
```

Hemos vuelto a perder el tipo de las variables numéricas, por lo que vamos a recuperarlo. Añadimos las dos nuevas columnas a la lista de variables numéricas.

In [30]:
```
numeric_cols = list(numeric_cols) + ['age_range', 'fare_range']
numeric_cols
```

Out[30]:
```
['pclass',
 'survived',
 'age',
 'sibsp',
 'parch',
 'fare',
 'is_married',
 'relatives',
 'alone',
 'age_range',
 'fare_range']
```

In [31]:
```
df4[numeric_cols] = df4[numeric_cols].astype(float)
df4
```

Out[31]:

	age	fare	embarked	pclass	survived	sex	sibsp	parch	title	is_married
0	29.000000	211.3375	S	1.0	1.0	female	0.0	0.0	Miss	0.0
1	0.916700	151.5500	S	1.0	1.0	male	1.0	2.0	Master	0.0
2	2.000000	151.5500	S	1.0	0.0	female	1.0	2.0	Miss	0.0
3	30.000000	151.5500	S	1.0	0.0	male	1.0	2.0	Mr	0.0
4	25.000000	151.5500	S	1.0	0.0	female	1.0	2.0	Mrs	1.0
...	...	...	...	...	...	...	...	...	...	...
1304	14.500000	14.4542	C	3.0	0.0	female	1.0	0.0	Miss	0.0
1305	29.881135	14.4542	C	3.0	0.0	female	1.0	0.0	Miss	0.0
1306	26.500000	7.2250	C	3.0	0.0	male	0.0	0.0	Mr	0.0
1307	27.000000	7.2250	C	3.0	0.0	male	0.0	0.0	Mr	0.0
1308	29.000000	7.8750	S	3.0	0.0	male	0.0	0.0	Mr	0.0

1309 rows × 15 columns

In [32]:
```
df4.dtypes
```

Out[32]:
```
age           float64
fare          float64
embarked       object
pclass        float64
survived      float64
sex            object
sibsp         float64
parch         float64
title          object
is_married    float64
deck           object
relatives     float64
alone         float64
age_range     float64
fare_range    float64
dtype: object
```

4) Tratamiento de variables categóricas

Recordemos que las variables categóricas, expresadas como texto, deben ser convertidas a tipo numérico. En clase hemos visto dos operaciones diferentes para alcanzar este objetivo:

- OrdinalEncoder: A partir de una columna con diferentes valores o categorías, devuelve una columna en la que cada categoría ha sido sustituida por un número.
- OneHotEncoder: A partir de una columna con N diferentes valores o categorías, devuelve N columnas binarias. A esta operación también la hemos denominado binarización.

Recordemos que para las variables categóricas nominales, la operación recomendada es `OneHotEncoder`. Si tratamos este tipo de variables mediante `OrdinalEncoder` obtendremos modelos predictivos con peor rendimiento, a excepción de los modelos basados en árboles, en los que podemos usar cualquiera de las dos operaciones.

4.1) OrdinalEncoder

Aunque no sea lo más adecuado, en este ejemplo, trataremos todas las variables categóricas con `OrdinalEncoder`.

Primero obtenemos las listas de nombres de columnas por tipos.

In [33]:
```
categorical_cols = df4.select_dtypes(exclude='number').columns
categorical_cols
```

Out[33]:
```
Index(['embarked', 'sex', 'title', 'deck'], dtype='object')
```

In [34]:
```
numeric_cols = df4.select_dtypes(include='number').columns
numeric_cols
```

Out[34]:
```
Index(['age', 'fare', 'pclass', 'survived', 'sibsp', 'parch', 'is_married',
       'relatives', 'alone', 'age_range', 'fare_range'],
      dtype='object')
```

A continuación definimos y aplicamos el `ColumnTransformer`.

In [35]:
```
ct_ordinal_encoder = ColumnTransformer([('ord_enc', OrdinalEncoder(), categorical_
                                        ("original",'passthrough', numeric_cols)
                                       ])
npdata3_ord_enc = ct_ordinal_encoder.fit_transform(df4)
npdata3_ord_enc
```

Out[35]:
```
array([[ 2.,  0.,  9., ...,  1.,  2.,  5.],
       [ 2.,  1.,  8., ...,  0.,  0.,  5.],
       [ 2.,  0.,  9., ...,  0.,  0.,  5.],
       ...,
       [ 0.,  1., 12., ...,  1.,  2.,  0.],
       [ 0.,  1., 12., ...,  1.,  2.,  0.],
       [ 2.,  1., 12., ...,  1.,  2.,  1.]])
```

Finalmente reconstruimos el `DataFrame` de pandas proporcionando los nombres de las columnas en el orden en que han sido procesadas.

In [36]:
```
new_column_order = list(categorical_cols) + list(numeric_cols)
new_column_order
```

Out[36]:
```
['embarked',
 'sex',
 'title',
 'deck',
 'age',
 'fare',
 'pclass',
 'survived',
 'sibsp',
 'parch',
 'is_married',
 'relatives',
 'alone',
 'age_range',
 'fare_range']
```

In [37]:
```
df5_ord_enc = pd.DataFrame(npdata3_ord_enc, columns=new_column_order)
df5_ord_enc
```

Out[37]:

	embarked	sex	title	deck	age	fare	pclass	survived	sibsp	parch	is_married
0	2.0	0.0	9.0	1.0	29.000000	211.3375	1.0	1.0	0.0	0.0	0.0
1	2.0	1.0	8.0	2.0	0.916700	151.5500	1.0	1.0	1.0	2.0	0.0
2	2.0	0.0	9.0	2.0	2.000000	151.5500	1.0	0.0	1.0	2.0	0.0
3	2.0	1.0	12.0	2.0	30.000000	151.5500	1.0	0.0	1.0	2.0	0.0
4	2.0	0.0	13.0	2.0	25.000000	151.5500	1.0	0.0	1.0	2.0	1.0
...	...	...	...	...	...	...	...	...	...	...	...
1304	0.0	0.0	9.0	8.0	14.500000	14.4542	3.0	0.0	1.0	0.0	0.0
1305	0.0	0.0	9.0	8.0	29.881135	14.4542	3.0	0.0	1.0	0.0	0.0
1306	0.0	1.0	12.0	8.0	26.500000	7.2250	3.0	0.0	0.0	0.0	0.0
1307	0.0	1.0	12.0	8.0	27.000000	7.2250	3.0	0.0	0.0	0.0	0.0
1308	2.0	1.0	12.0	8.0	29.000000	7.8750	3.0	0.0	0.0	0.0	0.0

1309 rows × 15 columns

Nótese que ya no hemos perdido el tipo de las variables numéricas, ya que al haber convertido las columnas de tipo `string` o `object` a numéricas, ahora el tipo de datos más general en nuestra tabla es `float`.

In [38]:
```
df5_ord_enc.dtypes
```

Out[38]:
```
embarked      float64
sex           float64
title         float64
deck          float64
age           float64
fare          float64
pclass        float64
survived      float64
sibsp         float64
parch         float64
is_married    float64
relatives     float64
alone         float64
age_range     float64
fare_range    float64
dtype: object
```

4.2) OneHotEncoder

En este caso vamos a aplicar `OneHotEncoder` a todas las variables categóricas, para esto, recuperamos el DataFrame anterior, es decir, `df4`.

Primero obtenemos las listas de nombres de columnas por tipos.

In [39]:
```
categorical_cols = df4.select_dtypes(exclude='number').columns
categorical_cols
```

Out[39]:
```
Index(['embarked', 'sex', 'title', 'deck'], dtype='object')
```

In [40]:
```
numeric_cols = df4.select_dtypes(include='number').columns
numeric_cols
```

Out[40]:
```
Index(['age', 'fare', 'pclass', 'survived', 'sibsp', 'parch', 'is_married',
       'relatives', 'alone', 'age_range', 'fare_range'],
      dtype='object')
```

A continuación definimos y aplicamos el `ColumnTransformer`.

In [41]:
```
ct_onehot_encoder = ColumnTransformer([('ohe', OneHotEncoder(), categorical_cols),
                                       ("original",'passthrough', numeric_cols)
                                      ])
npdata3_ohe = ct_onehot_encoder.fit_transform(df4)
npdata3_ohe
```

Out[41]:
```
array([[0., 0., 1., ..., 1., 2., 5.],
       [0., 0., 1., ..., 0., 0., 5.],
       [0., 0., 1., ..., 0., 0., 5.],
       ...,
       [1., 0., 0., ..., 1., 2., 0.],
       [1., 0., 0., ..., 1., 2., 0.],
       [0., 0., 1., ..., 1., 2., 1.]])
```

Finalmente, nos gustaría reconstruimos el `DataFrame` de pandas proporcionando los nombres de las columnas en el orden en que han sido procesadas. Sin embargo, la operación de binarización ha generado muchas columnas nuevas sin nombre.

Antes teníamos 15 columnas.

In [42]:
```
new_column_order = list(categorical_cols) + list(numeric_cols)
print(len(new_column_order))
new_column_order
```

Out[42]:
```
15
['embarked',
 'sex',
 'title',
 'deck',
 'age',
 'fare',
 'pclass',
 'survived',
 'sibsp',
 'parch',
 'is_married',
 'relatives',
 'alone',
 'age_range',
 'fare_range']
```

Ahora tenemos 43.

In [43]:
```
npdata3_ohe.shape
```

Out[43]:
```
(1309, 43)
```

Aunque es posible obtener, a partir del objeto `ColumnTransformer`, los nombres de las nuevas columnas generadas, es complejo y existe un método mucho más sencillo.

4.2) OneHotEncoder usando Pandas (get_dummies)

En este caso vamos a obtener una codificación `one-hot` de todas las variables categóricas, pero usando el método de Pandas `get_dummies` en lugar de la clase `OneHotEncoder` de Scikit-learn como veníamos haciéndolo hasta ahora. De nuevo, recuperamos el DataFrame anterior, es decir, `df4`.

Basta con llamar al método `get_dummies` pasándole el DataFrame y las columnas a transformar. El resultado, como podemos ver, es una tabla la que cada nueva variable que resulta de la binarización, tiene como nombre `NOMBRE-VARIABLE_NOMBRE-CATEGORÍA`.

In [44]:
```
df5_ohe = pd.get_dummies(df4, columns = categorical_cols)
df5_ohe
```

Out[44]:

	age	fare	pclass	survived	sibsp	parch	is_married	relatives	alone	age_range
0	29.000000	211.3375	1.0	1.0	0.0	0.0	0.0	0.0	1.0	2.0
1	0.916700	151.5500	1.0	1.0	1.0	2.0	0.0	3.0	0.0	0.0
2	2.000000	151.5500	1.0	0.0	1.0	2.0	0.0	3.0	0.0	0.0
3	30.000000	151.5500	1.0	0.0	1.0	2.0	0.0	3.0	0.0	2.0
4	25.000000	151.5500	1.0	0.0	1.0	2.0	1.0	3.0	0.0	2.0
...	...	...	...	...	...	...	...	...	...	...
1304	14.500000	14.4542	3.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0
1305	29.881135	14.4542	3.0	0.0	1.0	0.0	0.0	1.0	0.0	2.0
1306	26.500000	7.2250	3.0	0.0	0.0	0.0	0.0	0.0	1.0	2.0
1307	27.000000	7.2250	3.0	0.0	0.0	0.0	0.0	0.0	1.0	2.0
1308	29.000000	7.8750	3.0	0.0	0.0	0.0	0.0	0.0	1.0	2.0

1309 rows × 43 columns

Comprobamos que la tabla resultante tiene 43 columnas y que los tipos de datos son correctos.

In [45]:
```
df5_ohe.shape
```

Out[45]:
```
(1309, 43)
```

In [46]:
```
df5_ohe.dtypes
```

Out[46]:
```
age                    float64
fare                   float64
pclass                 float64
survived               float64
sibsp                  float64
parch                  float64
is_married             float64
relatives              float64
alone                  float64
age_range              float64
fare_range             float64
embarked_C               uint8
embarked_Q               uint8
embarked_S               uint8
sex_female               uint8
sex_male                 uint8
title_Capt               uint8
title_Col                uint8
title_Don                uint8
title_Dona               uint8
title_Dr                 uint8
title_Jonkheer           uint8
title_Lady               uint8
title_Major              uint8
title_Master             uint8
title_Miss               uint8
title_Mlle               uint8
title_Mme                uint8
title_Mr                 uint8
title_Mrs                uint8
title_Ms                 uint8
title_Rev                uint8
title_Sir                uint8
title_the Countess       uint8
deck_A                   uint8
deck_B                   uint8
deck_C                   uint8
deck_D                   uint8
deck_E                   uint8
deck_F                   uint8
deck_G                   uint8
deck_T                   uint8
deck_U                   uint8
dtype: object
```

Para terminar, separamos los predictores de la variable respuesta y normalizamos los datos.

In [47]:
```
y_titanic = df5_ohe['survived']
y_titanic
```

Out[47]:
```
0       1.0
1       1.0
2       0.0
3       0.0
4       0.0
       ...
1304    0.0
1305    0.0
1306    0.0
1307    0.0
1308    0.0
Name: survived, Length: 1309, dtype: float64
```

In [48]:
```
X_titanic = df5_ohe.drop('survived', axis=1)
X_titanic
```

Out[48]:

	age	fare	pclass	sibsp	parch	is_married	relatives	alone	age_range	fare_rang
0	29.000000	211.3375	1.0	0.0	0.0	0.0	0.0	1.0	2.0	5
1	0.916700	151.5500	1.0	1.0	2.0	0.0	3.0	0.0	0.0	5
2	2.000000	151.5500	1.0	1.0	2.0	0.0	3.0	0.0	0.0	5
3	30.000000	151.5500	1.0	1.0	2.0	0.0	3.0	0.0	2.0	5
4	25.000000	151.5500	1.0	1.0	2.0	1.0	3.0	0.0	2.0	5
...	...	...	...	...	...	...	...	...	...	
1304	14.500000	14.4542	3.0	1.0	0.0	0.0	1.0	0.0	1.0	3
1305	29.881135	14.4542	3.0	1.0	0.0	0.0	1.0	0.0	2.0	3
1306	26.500000	7.2250	3.0	0.0	0.0	0.0	0.0	1.0	2.0	0
1307	27.000000	7.2250	3.0	0.0	0.0	0.0	0.0	1.0	2.0	0
1308	29.000000	7.8750	3.0	0.0	0.0	0.0	0.0	1.0	2.0	1

1309 rows × 42 columns

In [49]:
```
X_titanic.shape
```

Out[49]:
```
(1309, 42)
```

In [50]:
```
scaler = MinMaxScaler(feature_range=(0, 1))
scaler.fit(X_titanic)
X_titanic_escalado = scaler.transform(X_titanic)
X_titanic_escalado
```

Out[50]:
```
array([[0.36116884, 0.41250333, 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.00939458, 0.2958059 , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.0229641 , 0.2958059 , 0.        , ..., 0.        , 0.        ,
        0.        ],
       ...,
       [0.32985358, 0.01410226, 1.        , ..., 0.        , 0.        ,
        1.        ],
       [0.33611663, 0.01410226, 1.        , ..., 0.        , 0.        ,
        1.        ],
       [0.36116884, 0.01537098, 1.        , ..., 0.        , 0.        ,
        1.        ]])
```

In [51]:
```
X_titanic_escalado.shape
```

Out[51]:
```
(1309, 42)
```