

Bernat Sort Rufat

Máster en Data Science

Artificial Intelligence for Data Science

# 1 Predicción del precio de venta de la vivienda en la ciudad de Barcelona

*Bagging & Boosting Project*

## 1.1 Contexto

Imaginemos que queremos comprar la vivienda en la que residimos de alquiler y nuestro casero nos hace una oferta para comprarle el piso. De cara a tomar esta decisión nos planteamos generar un modelo de machine learning que nos ayude a determinar cuánto debemos pagar por la vivienda.

## 1.2 Objetivo

El objetivo es generar un modelo que sea capaz de predecir el precio de venta de una vivienda en base al dataset de venta y posteriormente hacer uso de esa información en el dataset de alquiler.

## 1.3 Steps

### 1. Exploratory Data Analysis

- Estadística descriptiva y calidad general de los datos.
- Visualización y distribución de la variable objetivo.
- Distribución de las variables numéricas.
- Relación entre los atributos numéricos y la variable objetivo.

### 2. Data pre-processing & Feature engineering (1)

- Eliminación inicial de variables no relevantes.
- Tratamiento de missing values y creación de nuevas variables.
- Reducción de dimensionalidad de ciertas variables.

### 3. Correlation Analysis

- Análisis de correlación.
- Eliminación de variables altamente correlacionadas.
- Creación de nuevas variables.

### 3. Variables categóricas

- Distribución de las variables categóricas.
- Relación entre las variables categóricas y la variable objetivo.

### 4. Eliminación de variables no útiles (no predictoras)

### 5. Técnicas de reducción de dimensionalidad y Feature Importance

- PCA
- Mutual Information
- ANOVA / Kruskall - Wallis
- ExtraTreesRegressor

### 6. Test dataset pre-processing

### 7. Modeling

- Bagging: BaggingRegressor(DecisionTreeRegressor)
- Random Forest: RandomForestRegressor
- Boosting: CatBoostRegressor

### 8. SHAP Values Analysis

- Análisis de Shap values para el modelo de boosting (catboost).

### 9. Generación de predicciones y análisis de rentabilidad

- Entender qué tipos de viviendas son las más rentables para comprar.

### 10. Resultados finales, conclusiones y vías abiertas

- Resultado de R2 para el dataset de validación de nuestro mejor modelo.
- Limitaciones del modelo.
- Uso en la vida real.
- Propuestas de mejoras de los resultados obtenidos.

## TODO

Análisis exploratorio y limpieza de instancias:

- Entender bien el dataset y hacer las transformaciones necesarias para entender mejor los datos. Se valorará mediante:
  - Estadística descriptiva y calidad general de los datos.
  - Visualizaciones que ayuden a entender la distribución de las variables y categorías
  - Visualizaciones que ayuden a entender la relación entre los atributos y la variable objetivo precio\_de\_venta.
  - Estadísticos que nos ayuden a comprender mejor la relación (PCA, ANOVA, Correlación...).
  - Resumen de las conclusiones sacadas en este punto.

### Creación y comparación de modelos:

- Desarrollo de al menos 1 modelo de Bagging, 1 modelo de Random Forest y 1 modelo de Boosting a escoger entre CATBoost, XGBoost o LightGBM. Cada modelo debe seguir los siguientes pasos:
  - Preprocesamiento de los datos: hacer las transformaciones necesarias para que cada modelo pueda ser entrenado.
  - Split entre training y test.
  - Hiperoptimización de parámetros mediante grid search. En el caso de boosting debemos mostrar las visualizaciones del número de iteraciones con respecto a la variable objetivo para train y test.
  - Entrenamiento del modelo en training mediante 10 Fold Cross Validation.
  - Entrenamiento del modelo con todos los datos de training y validación con test.
  - Comparación de resultados obtenidos tanto en training como en test.

Como condiciones, el split entre train y test debe ser del 80% - 20% y la métrica de validación debe ser el R2.

### Análisis de Shap values para el modelo de boosting escogido:

- Desarrollar un análisis completo de Shap sobre el modelo de boosting desarrollado en el punto anterior.

### Generación de predicciones y análisis de rentabilidad:

- Entender qué tipos de viviendas son las más rentables para comprar. Para ello se realizarán los siguientes pasos:
  - Selección del mejor modelo de entre todos los calculados para predecir el precio de la vivienda.
  - Preparación del dataset de alquiler de pisos: el modelo que hemos generado admite datos que hemos transformado previamente (Cambio de variable para la variable objetivo, sustitución de null values, columnas no utilizadas...).
  - Cálculo de las predicciones sobre el dataset de alquiler.
  - Cálculo de la métrica de Break-even, resultante de dividir la predicción del precio de la vivienda por el precio de alquiler por mes.
  - Análisis exploratorio de esta métrica. ¿Qué tipo de viviendas son las más rentables para comprar?, ¿En qué barrios?, ¿Qué condiciones?

### Resultados finales, conclusiones y vías abiertas:

- Incluir el resultado de R2 para el dataset de validación de nuestro mejor modelo.
- Posteriormente, desarrollar los siguientes puntos: ¿Qué limitaciones hemos tenido con nuestro modelo?, ¿Lo utilizaríamos en la vida real?, ¿Cómo podemos mejorar los resultados obtenidos?

## 1.4 Carga de librerías

```
In [41]: import pandas as pd
import numpy as np
import math

# Plotting library
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import pylab
from matplotlib.colors import TwoSlopeNorm
plt.style.use('seaborn-v0_8')

# Funciones para hacer cálculo estadístico
import pingouin as pg
from scipy import stats
from scipy.stats import jarque_bera
import statsmodels.api as sm
from statsmodels.formula.api import ols
from sklearn.feature_selection import mutual_info_regression
from sklearn.decomposition import PCA

# Data pre-processing
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder

# Clustering
from sklearn.cluster import DBSCAN, KMeans
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster

# Type hint
from typing import Dict, List, Union, Tuple, Optional

# Pipelines
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import FunctionTransformer
from sklearn.base import BaseEstimator, TransformerMixin

# Modeling
from sklearn.model_selection import train_test_split
from sklearn import metrics # Calcula métricas para un modelo
from sklearn.metrics import r2_score
from sklearn.model_selection import GridSearchCV # Optimización de hipértulos
from sklearn.model_selection import cross_validate # trains model with
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import BaggingRegressor, RandomForestRegressor
import catboost
from catboost import CatBoostRegressor
```

```
# interactive widgets for Jupyter notebooks (catboost)
import ipywidgets
from ipywidgets import interact
!jupyter nbextension enable --py widgetsnbextension # for the interact

# Shap library
import shap
# print the JS visualization code to the notebook
shap.initjs()

# Define constants that we'll use throughout the notebook:
# All of the models in this notebook are fitted in a k-fold cross-
# manner. This constant represents the value of `k`.
# The target variable is 'price' for all the models.
# Random state = 42 for reproducibility
CROSS_VALIDATION_FOLDS = 10
TARGET_VARIABLE = 'price'
RANDOM_STATE = 42
Enabling notebook extension jupyter-js-widgets/extension...
- Validating: OK
```



```
In [42]: # Set display options to show all rows and columns
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

## 1.5 Tipo de problema

- Se trata de un problema de regresión ya que la variable objetivo es el precio de venta de la vivienda (variable numérica).
- Es supervisado ya que al algoritmo le estamos dando datos etiquetados: le proporcionamos muchas viviendas y le decimos cuánto cuesta cada una.

## 1.6 Carga de datos

```
In [14]: raw_data = pd.read_csv('sale_Barcelona.csv', delimiter = ',')
display(raw_data.head())
# dimensión del df (filas y columnas)
print(f" {raw_data.shape}\n")
# info de las columnas del df
print(raw_data.info())
```

	<b>id</b>	<b>price</b>	<b>currency</b>	<b>latitude</b>	<b>longitude</b>	<b>sq_meters</b>	<b>sq_meters_built</b>	<b>rooms</b>	<b>bathrooms</b>
<b>0</b>	320294	150000	€	41.459649	2.174793	63.0		67	3
<b>1</b>	1786997	150000	€	41.422081	2.155370	48.0		52	2
<b>2</b>	1787143	395000	€	41.402928	2.207851	84.0		91	2
<b>3</b>	1976767	540000	€	41.394692	2.144422	NaN		100	3
<b>4</b>	27972575	650000	€	41.398971	2.120754	NaN		141	3

(5847, 33)

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5847 entries, 0 to 5846
Data columns (total 33 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   id               5847 non-null   int64  
 1   price            5847 non-null   int64  
 2   currency         5847 non-null   object 
 3   latitude          5847 non-null   float64
 4   longitude         5847 non-null   float64
 5   sq_meters        3274 non-null   float64
 6   sq_meters_built  5847 non-null   int64  
 7   rooms             5847 non-null   int64  
 8   bathrooms         5847 non-null   int64  
 9   balcony           2210 non-null   float64
 10  terrace            1428 non-null   float64
 11  exterior          4770 non-null   float64
 12  orientation        3134 non-null   object  
 13  floor              3765 non-null   float64
 14  rooftop            407 non-null    float64
 15  elevator           5276 non-null   float64
 16  doorman            0 non-null     float64
 17  pool               170 non-null    float64
 18  ac                 2649 non-null   float64
 19  heating             3357 non-null   object  
 20  year_built         4082 non-null   float64
 21  quality            5847 non-null   int64  
 22  city                5847 non-null   object  
 23  neighborhood        5847 non-null   object 
```

```

24 dist_city_center      5847 non-null   float64
25 furniture            0 non-null    float64
26 garage               405 non-null   float64
27 property_type        5845 non-null   object
28 garden                61 non-null   float64
29 closest_station       5847 non-null   object
30 dist_closest_station 5847 non-null   float64
31 created_at           5847 non-null   object
32 last_seen             5847 non-null   object
dtypes: float64(18), int64(6), object(9)
memory usage: 1.5+ MB
None

```

## 1.7 Descripción del dataset

Columna	Descripción	Key
id	Identificador numérico de la vivienda	
price	Precio de mercado de la vivienda	
currency	Moneda	Euros
latitude	Latitud de las coordenadas geográficas de la vivienda	
longitude	Longitud de las coordenadas geográficas de la vivienda	
sq_meters	Metros cuadrados de la vivienda	
sq_meters_built	Metros cuadrados construidos de la vivienda	
rooms	Número de habitaciones	
bathrooms	Número de baños	
balcony	Indicador si la vivienda tiene balcón	1, 0
terrace	Indicador si la vivienda tiene terraza	1, 0
exterior	Indicador si la vivienda tiene una orientación exterior o interior en el edificio	1, 0
orientation	Orientación principal de la vivienda	norte, sur, este, oeste
floor	Piso de la vivienda	
rooftop	Indicador si la vivienda es un ático	1, 0
elevator	Indicador si el edificio de la vivienda tiene ascensor	1, 0
doorman	Indicador si el edificio tiene portero	1, 0
pool	Indicador si la vivienda cuenta con piscina o derecho de uso de piscina	1, 0
ac	Indicador si tiene aire acondicionado	1, 0
heating	Indicador si tiene calefacción	bomba, electric, gas, individual
year_builtin	Año de construcción	
quality	Indicador de calidad de la vivienda	2 - En buen estado

Columna	Descripción	Key
city	Ciudad de la vivienda	
neighborhood	Barrio de la vivienda	
dist_city_center	Distancia en kilómetros al centro de la ciudad	
furniture	Indicador si la vivienda cuenta con mobiliario	1: Sin Equipar; 2: Cocina Equipada; 3: Amueblado
garage	Indicador si la vivienda tiene garage	1, 0
property_type	Tipo de vivienda	
garden	Indicador si la vivienda cuenta con jardín	1,0
closest_station	Nombre de la estación de metro más cercana	
dist_closest_station	Distancia en kilómetros a la estación de metro más cercana	
created_at	Fecha de creación del anuncio	
last_seen	Fecha última en la que el anuncio fue publicado	

## 1.8 Separación de datasets en train y test

El primer paso debe ser separar los datos en conjuntos de train y test. Los separaremos en distintos archivos CSV, y el dataset de test de momento no lo trataremos (nos olvidamos de él por el momento). Esto lo hacemos de esta forma porque cada vez que se toca el conjunto de datos antes de dividirlo, se corre el riesgo de que se filtren datos ('data leakage'). El data leakage es hacer trampas, nuestro modelo parecerá mejor de lo que realmente es.

Por tanto, primero procesamos y transformamos el dataset de train. Luego, por separado, aplicamos el mismo procesamiento y transformación al dataset de test. Una vez procesados los datasets de train y test por separado, podremos aplicar los algoritmos de bagging, random forests y boosting.

```
In [4]: # Separar los datos en conjuntos de train y test (80-20)
train_data, test_data = train_test_split(raw_data, test_size=0.2, rand
```

```
# Guardar los conjuntos de datos en archivos CSV
train_data.to_csv('train_data.csv', index=False)
test_data.to_csv('test_data.csv', index=False)
```

Cargamos el train dataset:

```
In [8]: sale_df = pd.read_csv('train_data.csv', delimiter = ',')
print("Full train dataset shape is {}".format(sale_df.shape))
```

Full train dataset shape is (4677, 33)

## 1.9 Exploratory Data Analysis (EDA)

### 1.9.1 Exploración ràpida

```
In [4]: def explore_data(df: pd.DataFrame) -> None:  
    """  
        Takes in a pandas DataFrame and performs exploratory data analysis  
  
    Parameters:  
    -----  
    df: pandas DataFrame  
        The DataFrame to be analyzed.  
  
    Returns:  
    -----  
    None  
    """  
  
    # Print the dimensions of the DataFrame  
    print(f"The dataset includes {df.shape[0]} instances (rows) and {d  
  
    # Display the first few rows of the DataFrame  
    display(df.head())  
    print("\n")  
  
    # Print the column information for the DataFrame  
    print(df.info())  
    print("\n")  
  
    # Check missing values  
    print(f"Missing values: \n{df.isna().sum()}")  
    print("\n")  
  
    # Select only the numeric features  
    numeric_vars = df.select_dtypes(include=['int64', 'float64']).colu  
  
    # Calculate the descriptive statistics for the numeric variables  
    display(df[numeric_vars].describe())  
  
    # Check duplicates  
    print(f"\n Number of duplicates: {df.duplicated().sum()}")
```

In [40]: `explore_data(sale_df)`

The dataset includes 4677 instances (rows) and 33 variables (columns).

	<b>id</b>	<b>price</b>	<b>currency</b>	<b>latitude</b>	<b>longitude</b>	<b>sq_meters</b>	<b>sq_meters_built</b>	<b>rooms</b>	<b>balcony</b>
<b>0</b>	85122800	2290000	€	41.409071	2.099850	383.0		532	5
<b>1</b>	95206437	90000	€	41.434545	2.171110	42.0		50	1
<b>2</b>	94361131	145000	€	41.444518	2.175309	NaN		53	3
<b>3</b>	95156089	675000	€	41.392209	2.153368	93.0		120	4

Información de las columnas:

- Observamos que sq\_meters, balcony, terrace, exterior, orientation, floor, rooftop, elevator, pool, ac, heating, year\_built, garage, property\_type y garden tienen missing values.
- Las columnas doorman y furniture están vacías (todas las instancias están vacías).
- En realidad, balcony, terrace, exterior, rooftop, elevator, pool, ac, garage y garden son variables categóricas. Por lo tanto, las convertiremos de numéricas a categóricas una vez hayamos tratado los missing values.

El método df.describe() genera estadísticas descriptivas de un df. Obtenemos el recuento, la media, la desviación estándar, el mínimo, el primer cuartil (25%), la mediana (50%), el tercer cuartil (75%) y el máximo de cada columna.

Esta información puede ser útil para comprender la distribución y el rango de valores de cada columna numérica del df.

Por ejemplo, podemos observar que:

- El precio medio de las viviendas es de 358139 €, con una desviación típica de 496594 €. El precio mínimo es de 28000 € y el máximo de 9500000 €. Esto indica que existe una gran variación en los precios de las viviendas del dataset: el conjunto de datos incluye tanto propiedades muy baratas como muy caras.
- La propiedad más antigua del conjunto de datos se construyó en 1777, y la más reciente, en 2021.
- La superficie media de las propiedades (en metros cuadrados) es de 70.81 metros cuadrados, con una desviación típica de 48.17 metros cuadrados. La superficie mínima es de 3 metros cuadrados (muy pequeña), y la máxima, de 689 metros cuadrados (muy

grande).

### 1.9.2 Variable objetivo (price)

La variable objetivo en este caso es la columna "price", que indica el precio de mercado de la vivienda. Es decir, la variable que queremos predecir con nuestros modelos de machine learning.

Definimos la variable objetivo como una constante en el apartado de carga de librerías:  
TARGET\_VARIABLE = 'price'

#### Histograma

Histograma para entender la distribución del precio.

```
In [360]: def plot_histogram(data: pd.DataFrame, variable: str) -> None:
    """
    Creates a histogram of a given variable in a given DataFrame.

    Parameters:
    -----
    data : pandas DataFrame
        The DataFrame containing the target variable to be plotted.
    target_variable : str
        The name of the target variable to be plotted.

    Returns:
    -----
    None
    """

    fig, axs = plt.subplots(figsize=(10, 5))
    sns.histplot(data[variable])
    plt.title(f'Distribution of {variable}')
    plt.xlabel(variable)
    plt.ylabel('Frequency')
    plt.show()
```

```
In [54]: # Histograma de la variable precio  
plot_histogram(sale_df, TARGET_VARIABLE)
```



### Boxplot

Boxplot para ver si existen valores atípicos (outliers).

```
In [80]: def plot_boxplot(data: pd.DataFrame, target_variable: str, show_outliers: bool = False):
    """
    Creates a boxplot of a given variable in a given DataFrame.

    Parameters:
    -----
    data : pandas DataFrame
        The DataFrame containing the target variable to be plotted.
    target_variable : str
        The name of the target variable to be plotted.
    show_outliers : bool, optional (default=False)
        Whether to show or hide the outliers in the boxplot.

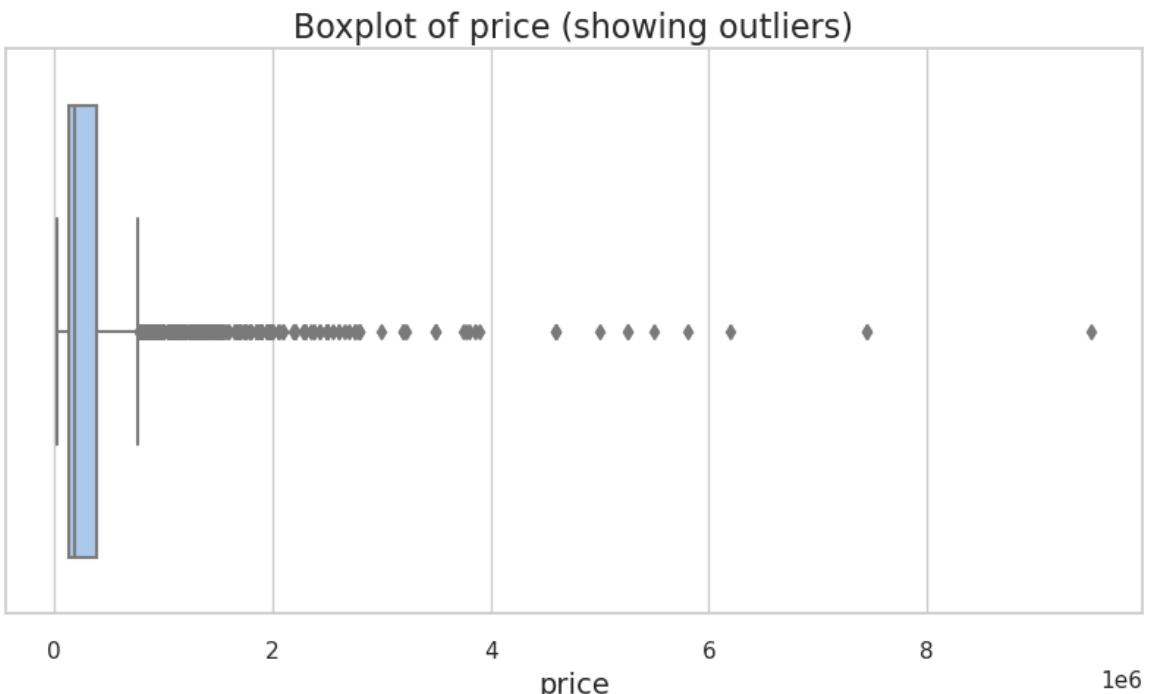
    Returns:
    -----
    None
    """

    sns.set(rc={"figure.figsize": (10, 5)}, style='whitegrid')
    if show_outliers:
        b = sns.boxplot(x=data[target_variable],
                         palette='pastel')
    else:
        b = sns.boxplot(x=data[target_variable],
                         palette='pastel',
                         showfliers=False)

    b.set_xlabel(target_variable, fontsize=14)
    b.set_title(f"Boxplot of {target_variable} {'showing outliers' if show_outliers else 'without outliers'}.")
    for patch in b.artists:
        r, g, b, a = patch.get_facecolor()
        patch.set_facecolor((r, g, b, .4))

    plt.show()
```

```
In [81]: # boxplot sin mostrar outliers  
plot_boxplot(sale_df, TARGET_VARIABLE, show_outliers = False)  
# boxplot mostrando outliers  
plot_boxplot(sale_df, TARGET_VARIABLE, show_outliers = True)
```



## Estadísticas descriptivas

```
In [82]: # Estadísticas descriptivas de la variable precio  
sale_df[TARGET_VARIABLE].describe()
```

```
Out[82]: count    4.677000e+03  
mean      3.581391e+05  
std       4.965949e+05  
min       2.800000e+04  
25%      1.350000e+05  
50%      1.870000e+05  
75%      3.900000e+05  
max       9.500000e+06  
Name: price, dtype: float64
```

- El precio medio de las viviendas es de 358139 €, con una desviación típica de 496594 €. El precio mínimo es de 28000 € y el máximo de 9500000 €. Esto indica que existe una gran variación en los precios de las viviendas del dataset: el conjunto de datos incluye tanto propiedades muy baratas como muy caras.
- El 50% de las propiedades tienen un precio entre 135000 y 390000 €, el 25% de las propiedades son más baratas que 135000 € y el 25% son más caras que 390000 €.
- También se puede observar que la distribución del precio es asimétrica (asimetría positiva) y que hay una gran diferencia entre el precio medio y el precio máximo, lo que sugiere que hay valores atípicos o outliers, como se observar en el boxplot y en el histograma.

### 1.9.3 Distribución de las variables numéricas

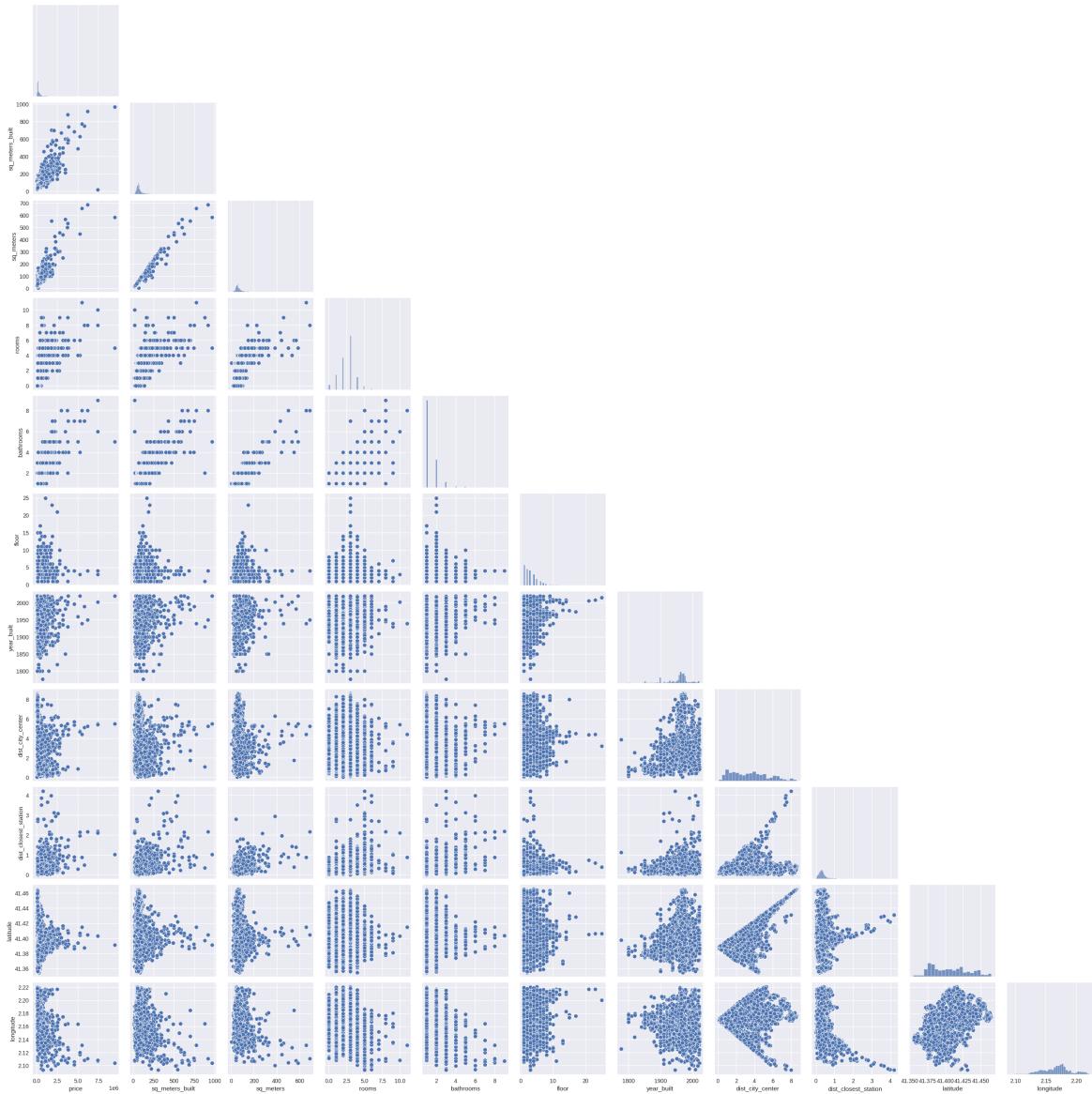
Exploramos la distribución de las variables numéricas. La distribución de las variables categóricas la exploraremos más adelante, una vez hayamos tratado los missing values, para poder ver la distribución correctamente.

```
In [126]: # variables numéricas
numeric_vars: List[str] = sale_df[['price', # target
                                    'sq_meters_built',
                                    'sq_meters',
                                    'rooms',
                                    'bathrooms',
                                    'floor',
                                    'year_built',
                                    'dist_city_center',
                                    'dist_closest_station',
                                    'latitude',
                                    'longitude']].columns

# variables numéricas (sin la variable objetivo price)
numeric_vars_no_target: List[str] = sale_df[['sq_meters_built',
                                              'sq_meters',
                                              'rooms',
                                              'bathrooms',
                                              'floor',
                                              'year_built',
                                              'dist_city_center',
                                              'dist_closest_station',
                                              'latitude',
                                              'longitude']].columns
```

In [97]: # gráfico de pares para todas las variables numéricas

```
plt.style.use('seaborn-v0_8')
sns.pairplot(sale_df[numeric_vars], corner=True)
plt.show()
```



- Cada punto en el gráfico representa una observación y la distribución conjunta de dos variables se muestra en los paneles de diagonal.
- Los paneles no diagonales muestran una dispersión bidimensional de dos variables.
- Parece que hay una relación lineal entre sq\_meters y sq\_meters\_build.
- También parece que cuantos más metros cuadrados habitables (y construidos) de la vivienda, más alto es el precio de la vivienda.

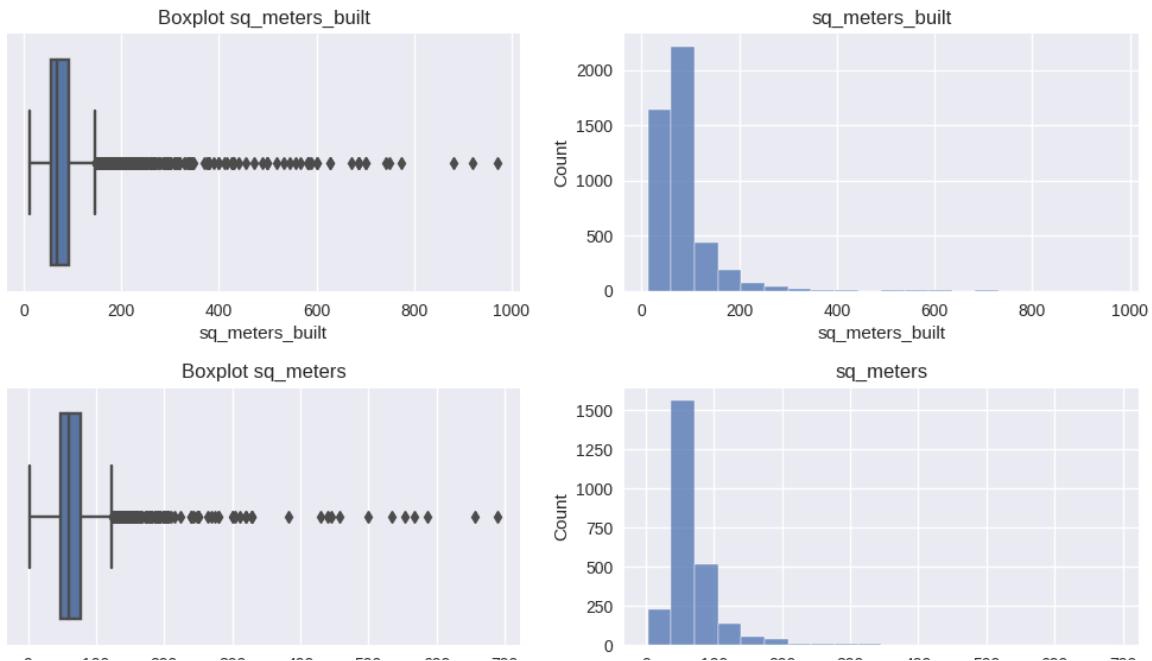
```
In [102]: # crear subplots con 5 filas y 2 columnas
plt.style.use('seaborn-v0_8')
fig, axes = plt.subplots(nrows=10, ncols=2, figsize=(10, 30))

# aplanar la matriz de subplots para iterar más fácilmente
axes = axes.flatten()

# iterar por cada variable y crear un histograma y boxplot en el subplot
for i, col in enumerate(sale_df[numeric_vars_no_target].columns):
    ax = axes[i*2+1]
    sns.histplot(ax=ax, data=sale_df[numeric_vars_no_target], x=col, bins=30)
    ax.set_title(col)

    ax = axes[i*2]
    sns.boxplot(ax=ax, data=sale_df[numeric_vars_no_target], x=col, showfliers=True)
    ax.set_title("Boxplot " + col)

# ajustar el espacio entre los subplots y mostrar el gráfico
plt.tight_layout()
plt.show()
```



- Observamos una asimetría positiva en las variables sq\_meters\_built, sq\_meters, floor y dist\_closest\_station.
- Rooms y longitude parecen seguir una distribución normal.
- Observamos outliers en la mayoría de variables numéricas.

#### 1.9.4 Relación entre los atributos numéricos y la variable objetivo price

Exploramos la distribución entre las variables numéricas y la variable objetivo price. La distribución entre las variables categóricas y la variable objetivo price la exploraremos más

Adelante, una vez hayamos tratado los missing values o hayamos hecho feature engineering.

## Scatter plots

```
In [156]: def scatter_plots(df: pd.DataFrame,
                      target: str,
                      numeric_vars: List[str],
                      figsize: Tuple[int, int]=(15,20)) -> None:
    """
    Creates scatter plots for each numeric variable in the given DataFrame.

    Args:
        - df: Pandas DataFrame.
        - target: Name of the target variable.
        - numeric_vars: List of names of the numeric variables to plot.
        - figsize: Tuple indicating the size of the figure to create.

    Returns: None.
    """

    # Create subplots with number of rows and columns based on number of numeric vars
    n_vars = len(numeric_vars)
    n_rows = (n_vars - 1) // 2 + 1
    fig, axs = plt.subplots(n_rows, 2, figsize=figsize)

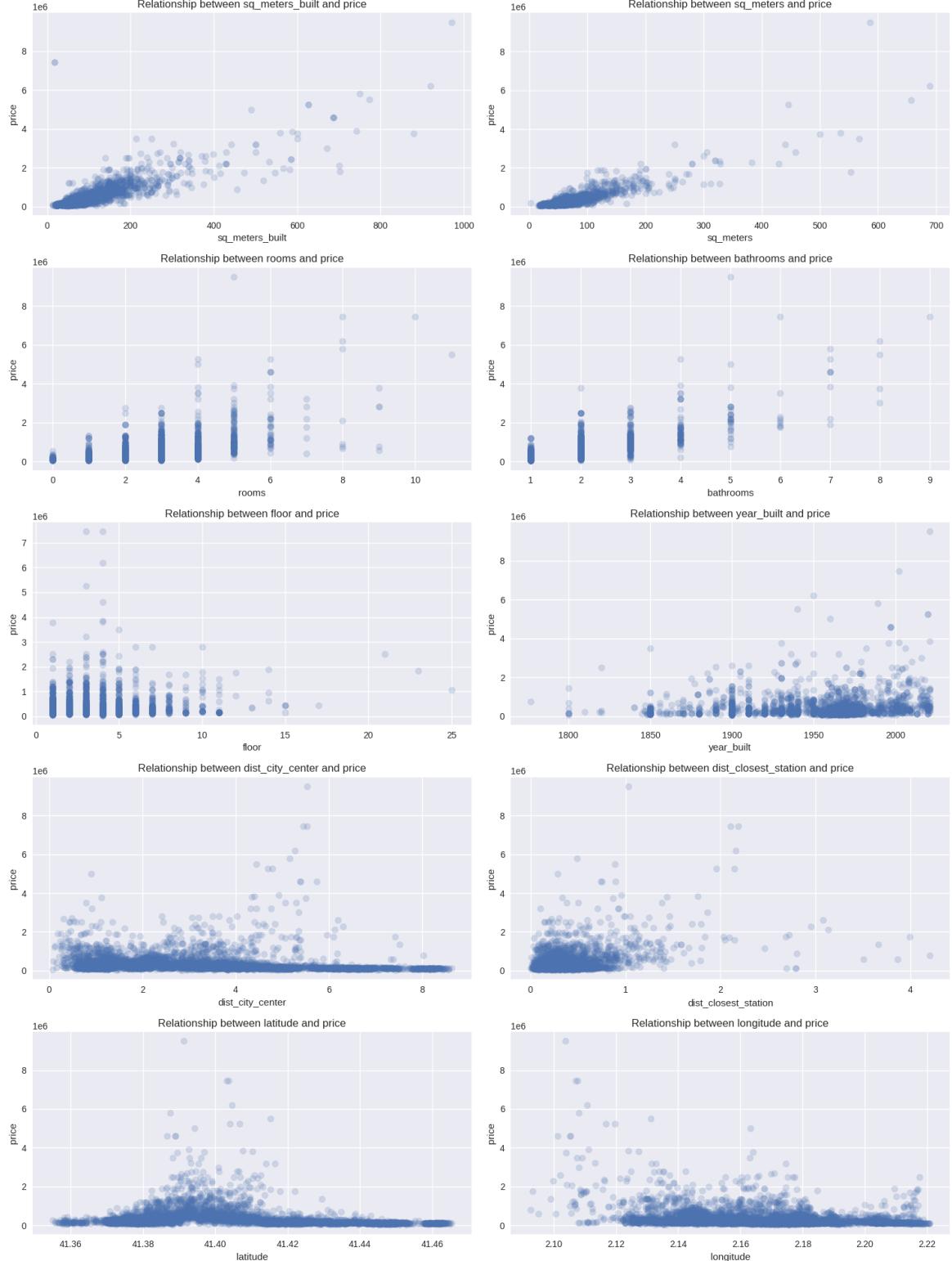
    # Flatten the matrix of subplots for easier iteration
    axs = axs.flatten()

    # Iterate over each variable and create a scatter plot on the corresponding subplot
    for i, col in enumerate(numeric_vars):
        ax = axs[i]
        ax.scatter(df[col], df[target], alpha=0.2)
        ax.set_title(f"Relationship between {col} and {target}")
        ax.set_xlabel(col)
        ax.set_ylabel(target)

    # Adjust the space between the subplots and remove empty plot if necessary
    plt.tight_layout()
    if n_vars % 2 != 0:
        axs[-1].remove()

    # Show the plot
    plt.show()
```

```
In [158]: scatter_plots(sale_df, TARGET_VARIABLE, numeric_vars_no_target)
```



- Parece ser que cuántos más metros cuadrados habitables de la vivienda (y construídos), más alto es el precio de la vivienda.

## 1.10 Data pre-processing & Feature engineering

## (1)

### 1.10.1 Eliminación inicial de variables no relevantes

Lo que nos interesa de este dataset es el precio y todo lo que nos ayude a predecir el precio. El resto, lo eliminaremos. Por tanto, retiramos las variables que no nos importan para el modelo.

Miramos las variables que tienen el mismo valor en todas las filas:

```
In [162]: def find_constant_variables(df: pd.DataFrame) -> list:  
    """  
        Returns a list of column names that have the same value in all rows  
  
    Parameters:  
        df (pandas dataframe): input dataframe  
  
    Returns:  
        list: list of column names that have the same value in all rows  
    """  
    constant_vars = []  
    for col in df.columns:  
        if df[col].nunique() == 1:  
            constant_vars.append(col)  
    return constant_vars
```

```
In [163]: find_constant_variables(sale_df)
```

```
Out[163]: ['currency',  
           'balcony',  
           'terrace',  
           'rooftop',  
           'pool',  
           'ac',  
           'quality',  
           'city',  
           'garage',  
           'garden']
```

Balcony, terrace, rooftop, pool, ac, garage y garden son binarias (0 y 1) y solo hay el valor 1 ya que los 0 no se han puesto debido a que la casa no disponía de estas características y simplemente los propietarios no lo especificaron. Por tanto, exploramos las demás:

```
In [159]: # checking 'quality'  
raw_data['quality'].value_counts()
```

```
Out[159]: 2      5847  
Name: quality, dtype: int64
```

```
In [160]: # checking 'city'  
raw_data['city'].value_counts()
```

```
Out[160]: Barcelona    5847  
Name: city, dtype: int64
```

```
In [161]: # checking 'currency'  
raw_data['currency'].value_counts()
```

```
Out[161]: €    5847  
Name: currency, dtype: int64
```

Inicialmente eliminamos:

- id: identificador único para cada propiedad y no aporta información relevante para el modelo.
- currency: dado que todos los precios están en euros, esta variable no aporta información relevante para el modelo.
- doorman: instancia vacía, podemos eliminarla.
- quality: todas las viviendas están en buen estado (quality = 2).
- city: todas las viviendas están en Barcelona.
- furniture: instancia vacía, podemos eliminarla.
- closest\_station: nos quedamos con dist\_closest\_station.
- created\_at y last\_seen: no aportan información relevante para el modelo de predicción del precio de la propiedad, ya que se refieren a la fecha de creación y última visualización del anuncio y no a las características de la propiedad en sí.

```
In [294]: # creamos un nuevo df:  
    # deep=True para asegurarnos de que los cambios realizados en el n  
    # no afecten al df original (objeto nuevo e independiente que no  
    # haga referencia a los mismos datos que el original)  
sale_clean_df = sale_df.copy(deep=True)  
  
# variables que no nos interesan  
not_useful_vars = ['id',  
                    'doorman',  
                    'quality',  
                    'city',  
                    'currency',  
                    'furniture',  
                    'closest_station',  
                    'created_at',  
                    'last_seen'  
]  
# eliminamos las variables que no nos interesan  
sale_clean_df.drop(not_useful_vars, axis=1, inplace=True)  
  
# Print the dimensions of the DataFrame  
print(f"The dataset includes {sale_clean_df.shape[0]} instances (rows)
```

The dataset includes 4677 instances (rows) and 24 variables (columns).

## 1.10.2 Tratamiento de missing values y creación de nuevas variables

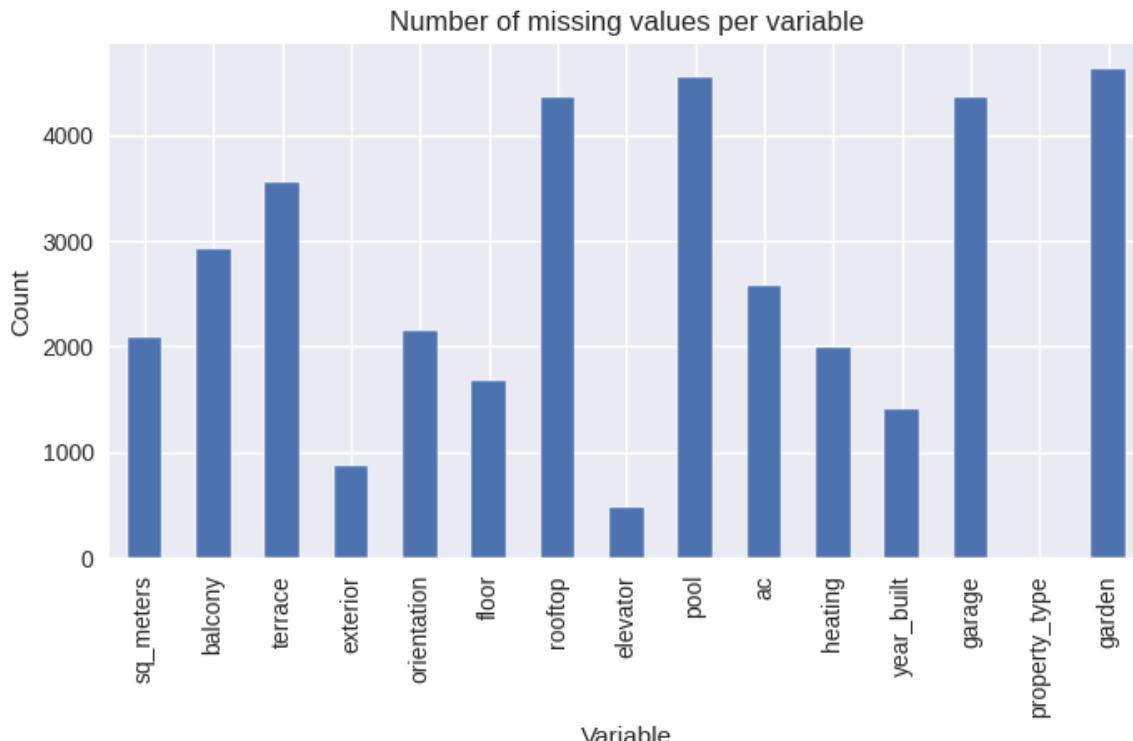
Miramos qué columnas tienen missing values:

```
In [295]: def count_null_values(df: pd.DataFrame) -> None:  
    """  
        Print the names of any columns in a Pandas DataFrame that contain  
        Also plots a bar chart showing the number of null values for each  
  
    Parameters:  
    -----  
    df: pd.DataFrame  
        The DataFrame to check for null values.  
    """  
    null_counts = df.isnull().sum()  
    null_vars = null_counts[null_counts != 0].index.tolist()  
  
    if null_vars:  
        print("Variables con valores nulos:\n")  
        for var in null_vars:  
            print(f"\t- '{var}': {null_counts[var]}")  
  
        # plot the number of null values for each variable  
        fig, ax = plt.subplots(figsize=(8, 4))  
        null_counts[null_vars].plot(kind='bar', ax=ax)  
        ax.set_title('Number of missing values per variable')  
        ax.set_xlabel('Variable')  
        ax.set_ylabel('Count')  
        plt.show()  
  
    else:  
        print("No se encontraron valores nulos en el dataset.\n")
```

```
In [296]: # columnas con missing values  
count_null_values(sale_clean_df)
```

Variables con valores nulos:

- 'sq\_meters': 2088
- 'balcony': 2918
- 'terrace': 3556
- 'exterior': 874
- 'orientation': 2153
- 'floor': 1680
- 'rooftop': 4353
- 'elevator': 470
- 'pool': 4553
- 'ac': 2569
- 'heating': 1986
- 'year\_built': 1415
- 'garage': 4355
- 'property\_type': 2
- 'garden': 4627



### 1.10.2.1 Binary variables

Los datos vienen de un portal inmobiliario y habrá campos obligatorios y otros opcionales.

Por ejemplo, en el caso de si la casa tiene piscina o no, los que tengan piscina van a poner que sí tienen piscina, pero los que no, tienen la opción de poner "NO" o no poner nada. Por tanto, el usuario no va a poner nada. En este caso, la transformación de los valores nulos a 0 es correcta para las variables numéricas porque van a ser vectores binarios (1 o 0).

Variables a las que se les aplicará la transformación de los valores nulos a 0:

- balcony
- terrace
- exterior
- rooftop
- elevator
- pool
- ac
- garage
- garden

```
In [149]: def get_binary_cols(df: pd.DataFrame) -> list[str]:  
    """  
        Returns a list with the names of the binary variables in a Pandas  
        DataFrame.  
        Parameters:  
            df: Pandas DataFrame to check for binary variables.  
  
        Returns:  
            A list with the names of the binary variables in the DataFrame  
    """  
    binary_cols = [col for col in df if (len(df[col].value_counts()) >  
                                         2)  
                  and (df[col].value_counts().min() == 1)]  
  
    return binary_cols
```

```
In [298]: get_binary_cols(sale_clean_df)
```

```
Out[298]: ['balcony',  
          'terrace',  
          'exterior',  
          'rooftop',  
          'elevator',  
          'pool',  
          'ac',  
          'garage',  
          'garden']
```

Sustituimos los valores nulos por 0:

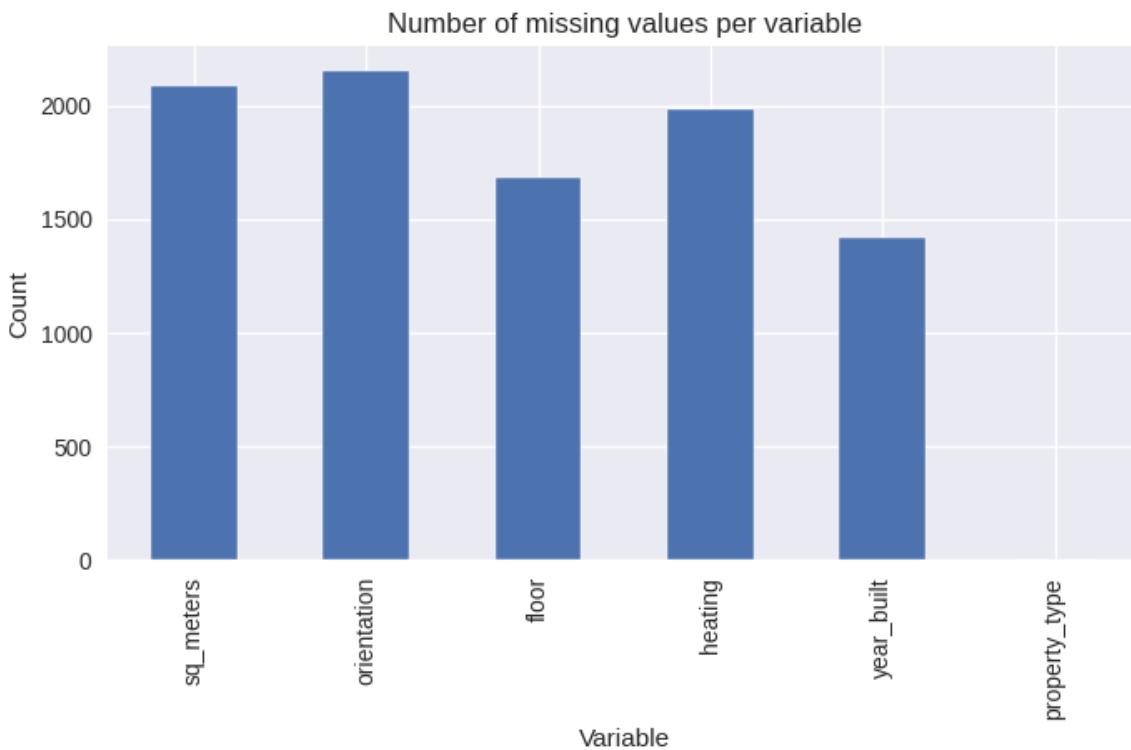
```
In [5]: def replace_binary_cols_nulls_by_zero(df: pd.DataFrame) -> None:  
    """  
        Replaces missing values in binary columns of a pandas DataFrame wi  
  
    Parameters:  
        df: Input pandas DataFrame  
  
    Returns:  
        None  
    """  
    binary_cols = get_binary_cols(df)  
    df.loc[:, binary_cols] = df.loc[:, binary_cols].fillna(0)
```

```
In [300]: # sustituimos los valores nulos por 0:  
replace_binary_cols_nulls_by_zero(sale_clean_df)
```

```
In [301]: # variables con missing values  
count_null_values(sale_clean_df)
```

Variables con valores nulos:

- 'sq\_meters': 2088
- 'orientation': 2153
- 'floor': 1680
- 'heating': 1986
- 'year\_built': 1415
- 'property\_type': 2



In [302]: `sale_clean_df.loc[:, get_binary_cols(sale_clean_df)].head()`

Out[302]:

	balcony	terrace	exterior	rooftop	elevator	pool	ac	garage	garden
0	1.0	1.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0
1	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0
2	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	1.0	0.0	1.0	1.0	0.0	1.0	0.0	0.0
4	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0

- Las variables binarias ya no tienen valores nulos: se han sustituido los missing values por 0.

#### 1.10.2.1.1 Conversión de numéricas a categóricas binarias

En realidad, balcony, terrace, exterior, rooftop, elevator, pool, ac, garage y garden son variables categóricas. Por lo tanto, las convertiremos de numéricas a categóricas:

In [404]: `# variables numéricas que son 0 y 1 y queremos convertir a categóricas  
get_binary_cols(sale_clean_df)`

Out[404]: `['balcony',  
'terrace',  
'exterior',  
'rooftop',  
'elevator',  
'pool',  
'ac',  
'garage',  
'garden']`

In [405]: `# data types  
sale_clean_df[get_binary_cols(sale_clean_df)].dtypes`

Out[405]: `balcony float64  
terrace float64  
exterior float64  
rooftop float64  
elevator float64  
pool float64  
ac float64  
garage float64  
garden float64  
dtype: object`

- Todas son float64 (0.0 y 1.0)

```
In [406]: # primero las convertimos a int64 para eliminar los decimales: 0 y 1  
sale_clean_df[get_binary_cols(sale_clean_df)] = sale_clean_df[get_binary_cols(sale_clean_df)].astype(int)  
# luego las convertimos a 'category': '0' y '1'  
sale_clean_df[get_binary_cols(sale_clean_df)] = sale_clean_df[get_binary_cols(sale_clean_df)].map({0: '0', 1: '1'})  
# data types  
sale_clean_df[get_binary_cols(sale_clean_df)].dtypes
```

```
Out[406]: balcony      category  
terrace       category  
exterior      category  
rooftop        category  
elevator      category  
pool           category  
ac             category  
garage         category  
garden         category  
dtype: object
```

- Ya tenemos las variables categóricas binarias como 'category'.

### 1.10.2.2 Edad de las viviendas

Tenemos el año de construcción de la vivienda (year\_built), pero en realidad, lo que nos interesa no es el año de construcción, sino la edad relativa del edificio, es decir, como de viejo o nuevo es el edificio.

Definimos una nueva variable que será cuál es la edad de la vivienda (property\_age).

- Como hemos visto anteriormente, tenemos una gran cantidad de nulos para la variable year\_built (1415). Para solucionar esto, emplearemos una técnica que funciona bien en variables numéricas para poder sacar información: vamos a hacer una transformación categórica, de forma que los valores nulos les ponemos 'Unknown' y en el resto tratamos de hacer categorías con cierto criterio.

Creamos la edad (años) de la vivienda:

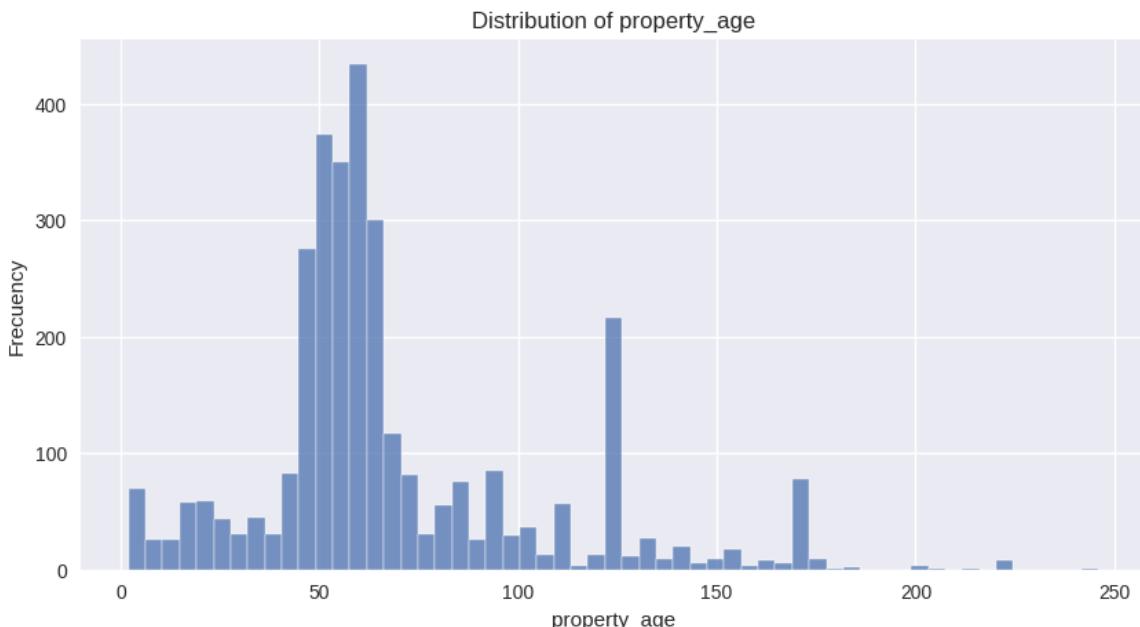
```
In [303]: sale_clean_df['property_age'] = 2023 - sale_clean_df['year_built']
display(sale_clean_df.head())
print(sale_clean_df.shape)
```

	price	latitude	longitude	sq_meters	sq_meters_built	rooms	bathrooms	balcony	ter
0	2290000	41.409071	2.099850	383.0		532	5	6	1.0
1	90000	41.434545	2.171110	42.0		50	1	1	0.0
2	145000	41.444518	2.175309	NaN		53	3	1	1.0
3	675000	41.392209	2.153368	93.0		120	4	2	0.0
4	319000	41.413385	2.162246	NaN		69	3	1	0.0

(4677, 25)

Hacemos un histograma de la edad de las viviendas:

```
In [304]: # Histograma de la variable precio
plot_histogram(sale_clean_df, 'property_age')
```



- Observamos que la mayoría de las viviendas tienen entre 50 y 80 años (desarrollo de Barcelona de los años 50 y 60). También tenemos un pico entorno a los 120 años.
- Apreciamos pequeños "clusters de viviendas, los cuales podemos agrupar en categorías según la edad de la vivienda.

### **1.10.2.2.1 Selección de rangos de edad mediante clustering**

Primero utilizamos DBSCAN (que determina automáticamente el número de clusters en función de la densidad de los datos) para determinar el número óptimo de clusters y, a continuación, utilizamos la agrupación de K-means para agrupar la variable `property_age` en función del número óptimo de clusters determinado. Por último, visualizamos los clusters mediante un gráfico de dispersión.

```
In [305]: # Create a copy
df = sale_clean_df.dropna(subset=['property_age']).copy()

# Scale the property_age column using StandardScaler
X = StandardScaler().fit_transform(df['property_age'].values.reshape(-1, 1))

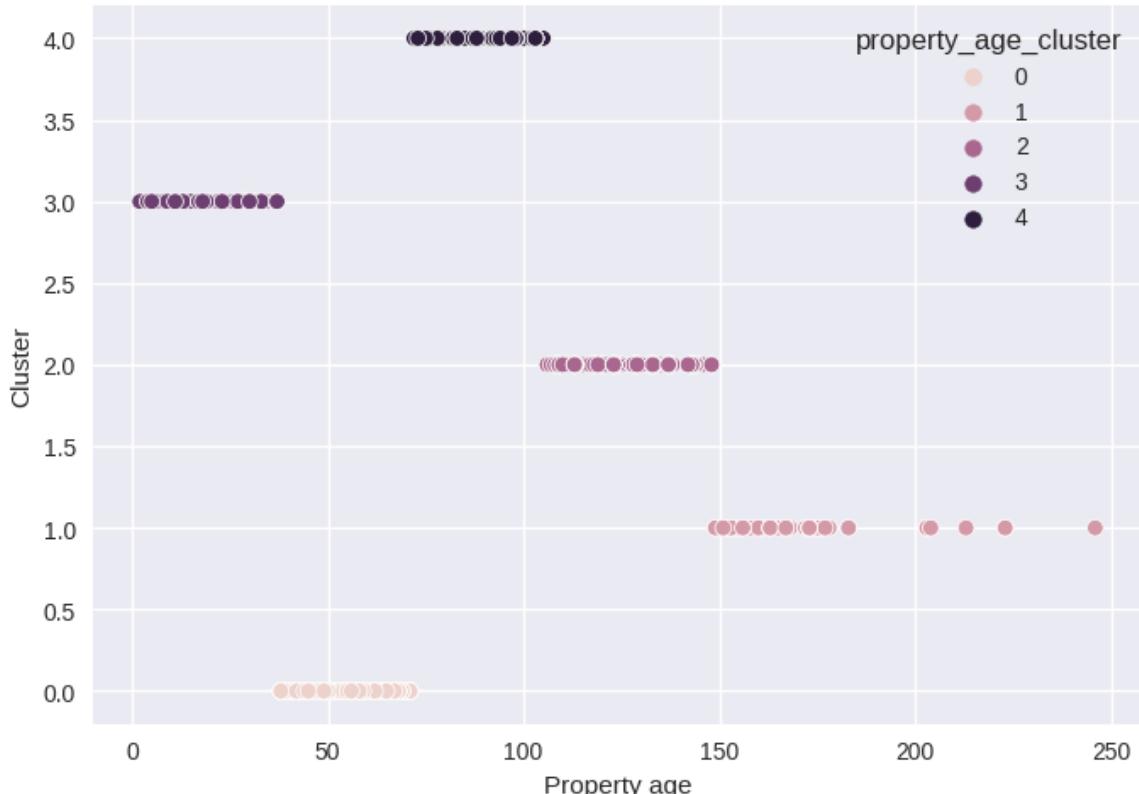
# Evaluate DBSCAN for range of eps values and select best_eps
eps_range = np.arange(0.1, 1.1, 0.1)
scores = [len(set(DBSCAN(eps=eps, min_samples=5).fit(X).labels_)) - 1 for eps in eps_range]
best_eps = eps_range[np.argmax(scores)]
# optimal number of clusters
k = len(set(DBSCAN(eps=best_eps, min_samples=5).fit(X).labels_)) - 1
print("The optimal number of clusters is:", k)

# Perform K-means clustering on the property_age variable with the opt
kmeans = KMeans(n_clusters=k, random_state=0).fit(X)

# Assign each property to a cluster based on its property age
df['property_age_cluster'] = kmeans.labels_

# Visualize the clusters using a scatter plot
sns.scatterplot(data=df, x='property_age', y='property_age_cluster', h
plt.xlabel('Property age')
plt.ylabel('Cluster')
plt.show()
```

The optimal number of clusters is: 5



```
In [306]: # ordenamos por el mínimo en orden descendente:  
def q25(x): return x.quantile(0.25)  
def q75(x): return x.quantile(0.75)  
  
df.groupby('property_age_cluster', as_index=False)\  
    .agg({'property_age': ['min', 'max', 'mean', 'median', q  
    .sort_values(('property_age', 'min')), ascending=False)
```

Out [306]:

	property_age_cluster	property_age						
		min	max	mean	median	q25	q75	
1		1	149.0	246.0	173.531469	173.0	167.0	173.0
2		2	106.0	148.0	123.880319	123.0	123.0	123.0
4		4	72.0	105.0	86.973105	87.0	82.0	93.0
0		0	38.0	71.0	55.919980	56.0	52.0	61.0
3		3	2.0	37.0	19.220430	21.0	10.0	27.0

- Tenemos 5 clústers.

La edad de la vivienda (property\_age) podemos transformarla a una variable categórica incluyendo los valores nulos como una categoría de 'Unknown'.

Definimos los grupos de edad en base al clustering anterior:

```
In [307]: def property_age_conditions(x: float) -> str:  
    """  
        Assigns a property age range label to a property age value.  
  
    Parameters:  
        x (float): The property age value.  
  
    Returns:  
        str: The label for the property age range that the value falls  
    """  
    if pd.isna(x): # check for NaN values  
        return "Unknown"  
    elif x < 0: # viviendas dónde no tengo valores (missing values)  
        return "Unknown"  
    elif 0 <= x <= 40: # entre 0 y 40 años  
        return "0 - 40"  
    elif 40 < x <= 70: # entre 41 y 70 años  
        return "40 - 70"  
    elif 70 < x <= 120: # entre 71 y 120 años  
        return "70 - 120"  
    elif 120 < x <= 150: # entre 121 y 150 años  
        return "120 - 150"  
    elif x > 150: # más de 150 años  
        return "+150"  
    else:  
        return "Unknown"
```

```
In [308]: # aplicamos la función a la columna 'property_age' del DataFrame
sale_clean_df['property_age_cat'] = sale_clean_df['property_age'].apply(lambda x: property_age_conditions(x))
sale_clean_df.head(10)
```

Out[308]:

	price	latitude	longitude	sq_meters	sq_meters_built	rooms	bathrooms	balcony	terrace
0	2290000	41.409071	2.099850	383.0		532	5	6	1.0
1	90000	41.434545	2.171110	42.0		50	1	1	0.0
2	145000	41.444518	2.175309	NaN		53	3	1	1.0
3	675000	41.392209	2.153368	93.0		120	4	2	0.0
4	319000	41.413385	2.162246	NaN		69	3	1	0.0
5	88000	41.423051	2.155127	61.0		64	2	1	0.0
6	262000	41.411514	2.209901	74.0		80	3	1	1.0
7	345000	41.402153	2.200675	78.0		87	3	1	1.0
8	210000	41.405843	2.174957	50.0		55	2	1	0.0
9	359000	41.399902	2.179506	53.0		61	0	1	0.0

```
In [309]: # prueba para comprobar que funciona bien
edades = [40, 39, 70, 71, 69, 149, 150]
for edad in edades:
    print(f"La edad {edad} se clasifica como {property_age_conditions(edad)}")
```

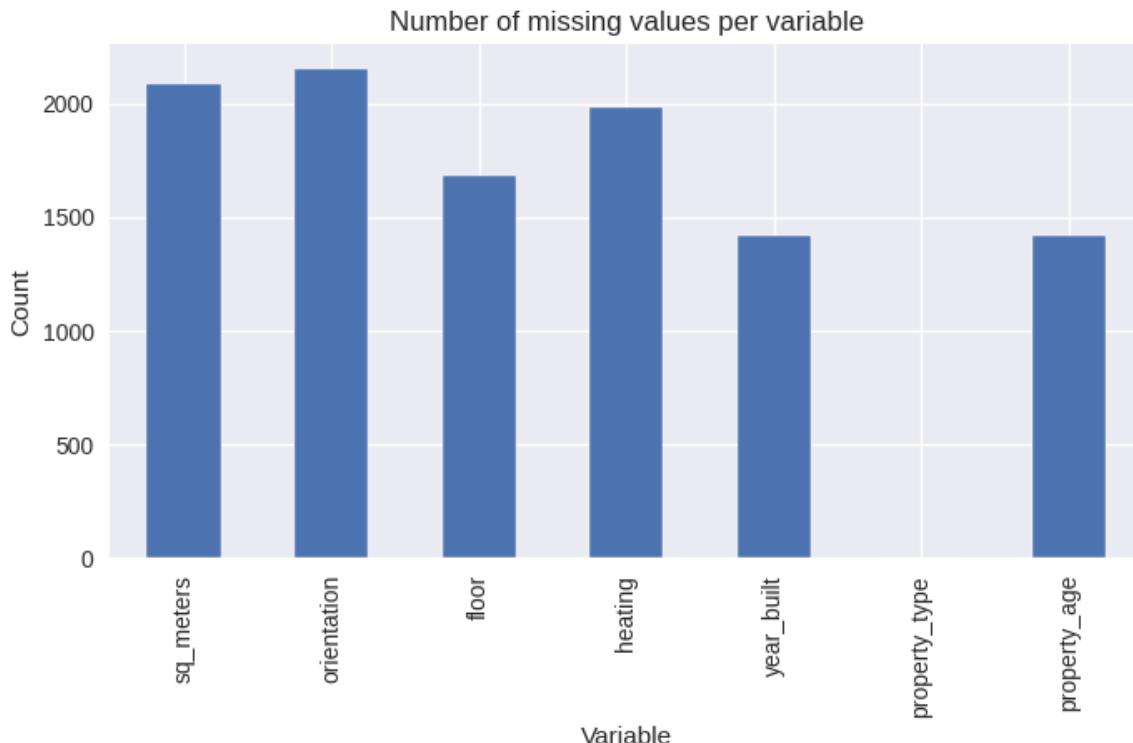
La edad 40 se clasifica como 0 – 40  
 La edad 39 se clasifica como 0 – 40  
 La edad 70 se clasifica como 40 – 70  
 La edad 71 se clasifica como 70 – 120  
 La edad 69 se clasifica como 40 – 70  
 La edad 149 se clasifica como 120 – 150  
 La edad 150 se clasifica como 120 – 150

- Hemos pasado de una variable numérica a una variable categórica.
- Es cierto que lo hemos hecho perdiendo información, pero perdemos más con los missing values que con decir que no sabemos lo que es (Unknown).
- Es probable que la predicción de este modelo sea buena en estas categorías pero la predicción será un poco peor en las categorías ‘Unknown’, ya que no sabemos cuál es la edad de la vivienda.

```
In [310]: # variables con missing values  
count_null_values(sale_clean_df)
```

Variables con valores nulos:

- 'sq\_meters': 2088
- 'orientation': 2153
- 'floor': 1680
- 'heating': 1986
- 'year\_built': 1415
- 'property\_type': 2
- 'property\_age': 1415



### 1.10.2.3 Property\_type, orientation y heating

En las variables categóricas property\_type, orientation y heating tenemos Null values: los transformamos a 'Unknown'. Ahora tenemos una categoría que es 'Unknown'.

Esto implica que podemos seguir utilizando estas instancias en el modelo y la precisión del modelo con estas instancias va a ser un poco peor, porque por ejemplo, te dice tengo una casa, pero no sabemos si es un piso, casa, chalet o dúplex.

No obstante, en este caso es mejor tener estas instancias y perder un poco de información que no eliminar todas estas instancias que no tienen null values.

```
In [311]: def check_value_counts(df: pd.DataFrame, cols: List[str]) -> None:  
    """  
        Print the value counts of the specified columns in a DataFrame.  
  
    Parameters:  
        df (pd.DataFrame): The DataFrame to check value counts for.  
        cols (List[str]): A list of column names to check value counts for  
  
    Returns:  
    None  
    """  
    for col in cols:  
        print(f"Value counts for column '{col}':")  
        print(df[col].value_counts())  
        print()
```

```
In [312]: # variables categóricas con missing values que queremos convertir a cat  
cat_features_na_to_unknown = ['property_type',  
                               'orientation',  
                               'heating'  
                             ]
```

```
In [313]: check_value_counts(sale_clean_df, cat_features_na_to_unknown)
```

```
Value counts for column 'property_type':  
piso          4086  
atico         235  
estudio       177  
duplex        75  
chalet        36  
chalet_adosado  28  
chalet_independiente 26  
chalet_pareado   12  
Name: property_type, dtype: int64
```

```
Value counts for column 'orientation':  
sur          1149  
norte        588  
este          553  
oeste         234  
Name: orientation, dtype: int64
```

```
Value counts for column 'heating':  
gas           1104  
na            772  
individual    350  
bomba         266  
electric      199  
Name: heating, dtype: int64
```

- Nos damos cuenta que 'heating' debería tener solo los valores bomba, electric, gas e individual, pero tiene 'na', así que consideramos este 'na' como un missing value.

```
In [314]: # Fill missing values with 'Unknown' for all columns
for col in cat_features_na_to_unknown:
    if col == 'heating':
        sale_clean_df[col].fillna('Unknown', inplace=True)
        sale_clean_df[col].replace('na', 'Unknown', inplace=True)
    else:
        sale_clean_df[col].fillna('Unknown', inplace=True)

sale_clean_df[cat_features_na_to_unknown].head(10)
```

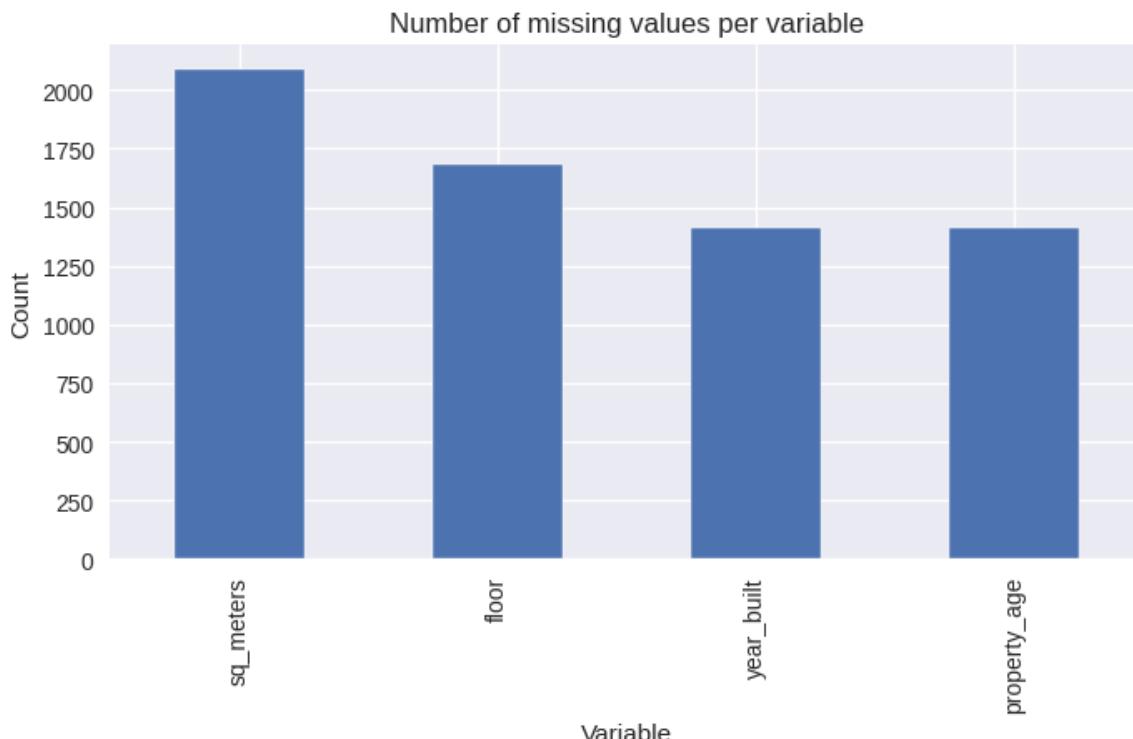
Out[314]:

	property_type	orientation	heating
0	chalet_independiente	sur	individual
1	piso	norte	bomba
2	piso	Unknown	Unknown
3	atico	norte	individual
4	piso	Unknown	Unknown
5	piso	este	Unknown
6	piso	norte	gas
7	piso	norte	Unknown
8	piso	oeste	Unknown
9	estudio	Unknown	Unknown

```
In [315]: # variables con missing values
count_null_values(sale_clean_df)
```

Variables con valores nulos:

- 'sq\_meters': 2088
- 'floor': 1680
- 'year\_built': 1415
- 'property\_age': 1415



- Realmente year\_built y property\_age no nos interesan ya que hemos creado la variable property\_age\_cat, así que las eliminaremos posteriormente.
- sq\_meters antes de tratarla miraremos, posteriormente, si puede estar correlacionada con sq\_meters\_built: si están correlacionadas, la eliminamos y nos quedamos con sq\_meters\_built ya que no tiene valores nulos.
- Nos queda por tratar los missing values de la variable floor, que es numérica.

#### 1.10.2.4 Floor

Estudiamos la categoría floor.

Calculamos varias estadísticas relacionadas con la variable "price" para cada valor único de la variable "floor".

- En primer lugar, seleccionamos las columnas "floor" y "price" del conjunto de datos. A continuación, agrupamos el conjunto de datos por los valores únicos de "floor" utilizando el método groupby con el argumento as\_index=False, lo que significa que el df resultante no tendrá los valores únicos de "floor" como índice.

- Tras la agrupación, aplicamos el método agg() para calcular diversas estadísticas de la variable "price". Estas estadísticas incluyen:
  - category\_count: el número de instancias de cada valor único de "floor" en el conjunto de datos, calculado mediante el método "count".
  - mean\_price: la media de la variable "price" para cada valor único de "floor", calculada mediante el método "mean".
  - median\_price: la mediana de la variable "price" para cada valor único de "floor", calculada mediante el método "median".
  - min\_price: valor mínimo de la variable "price" para cada valor único de "floor", calculado mediante el método "min".
  - max\_price: valor máximo de la variable "price" para cada valor único de "floor", calculado mediante el método "max".
  - standard\_deviation: la desviación estándar de la variable "precio" para cada valor único de "floor", calculada mediante el método "std".
- Por último, ordenamos el df resultante por la variable "floor" en orden ascendente y restablecemos el índice utilizando los métodos sort\_values y reset\_index, respectivamente. El df resultante tendrá una fila para cada valor único de "floor" en el dataset original, con columnas para cada una de las estadísticas calculadas.

```
In [391]: df_floor = sale_clean_df.copy(deep=True)
```

```
In [392]: floor = df_floor[['floor', 'price']].groupby('floor', as_index=False).  
    # Number of instances per category  
    category_count=('price', "count"),  
    # Mean price  
    mean_price=('price', "mean"),  
    # Median price  
    median_price=('price', "median"),  
    # Min price  
    min_price=('price', min),  
    # Max price  
    max_price=('price', max),  
    # Standard deviation  
    standard_deviation=('price', "std")).sort_values('floor', ascending=True)  
  
floor
```

Out [392]:

	index	floor	category_count	mean_price	median_price	min_price	max_price	standard deviation
0	0	1.0	770	3.077965e+05	199900.0	57000	3775000	300790.0
1	1	2.0	611	3.118708e+05	192000.0	69900	2200000	298530.0
2	2	3.0	565	3.708888e+05	215000.0	80000	7450000	521180.0
3	3	4.0	413	4.201416e+05	200000.0	35000	7450000	682510.0
4	4	5.0	251	3.946239e+05	250000.0	75000	3500000	424150.0
5	5	6.0	156	3.993942e+05	330000.0	83000	2800000	363960.0
6	6	7.0	89	4.118202e+05	320000.0	75000	2800000	369600.0
7	7	8.0	58	4.253619e+05	289495.0	90000	1690000	317720.0
8	8	9.0	23	4.245174e+05	250000.0	132000	1690000	408230.0
9	9	10.0	24	6.807292e+05	402500.0	158000	2800000	660670.0
10	10	11.0	22	3.681818e+05	159000.0	159000	1500000	408590.0
11	11	12.0	2	1.285000e+06	1285000.0	820000	1750000	657600.0
12	12	13.0	2	3.575000e+05	357500.0	345000	370000	176700.0
13	13	14.0	3	1.163333e+06	960000.0	630000	1900000	658960.0
14	14	15.0	4	3.617500e+05	435000.0	142000	435000	146500.0
15	15	17.0	1	4.350000e+05	435000.0	435000	435000	
16	16	21.0	1	2.500000e+06	2500000.0	2500000	2500000	
17	17	23.0	1	1.850000e+06	1850000.0	1850000	1850000	
18	18	25.0	1	1.060000e+06	1060000.0	1060000	1060000	

Precio mediano de la vivienda en BCN en base al piso (floor):

```
In [393]: plt.bar(x='floor', height='median_price', data=floor)
plt.rcParams["figure.figsize"] = (50,20)
plt.xticks(fontsize=40) # tamaño de las etiquetas del eje x
plt.yticks(fontsize=40) # tamaño de las etiquetas del eje y
plt.xlabel('Piso (floor)', fontsize=40) # etiqueta del eje x y su tamaño
plt.ylabel('Precio mediano', fontsize=40) # etiqueta del eje y y su tamaño
plt.title('Precio mediano de la vivienda en BCN por piso', fontsize=50)
plt.show()
```

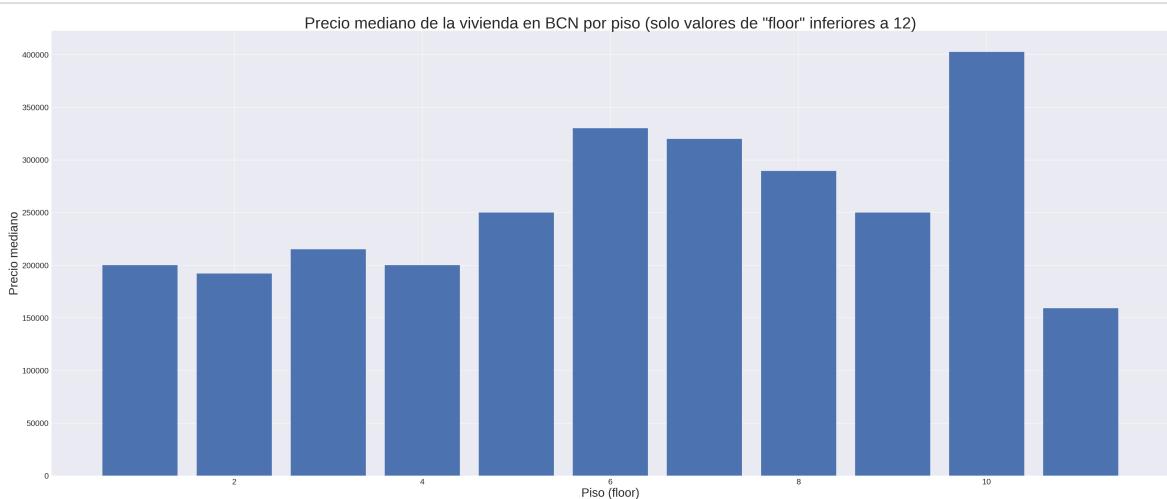


- Vemos que para pisos altos el precio aumenta mucho, pero tenemos muy pocas instancias (en category\_count) (Menos de 10) y además tenemos null values.

Valores de floor inferiores a 12:

```
In [394]: fig, ax = plt.subplots(figsize=(50, 20))

ax.bar(x='floor', height='median_price', data=floor[floor['floor'] < 12])
ax.set_xlabel('Piso (floor)', fontsize=30)
ax.set_ylabel('Precio mediano', fontsize=30)
ax.set_title('Precio mediano de la vivienda en BCN por piso (solo valores de "floor" inferiores a 12)', fontsize=30)
ax.tick_params(axis='both', labelsize=20) # aumenta el tamaño de las etiquetas
plt.show()
```



Decidimos hacer una transformación categórica con pisos de hasta 4 plantas, hasta 11 plantas y superiores.

- Perdemos el número pero por lo general tampoco importa tanto con respecto al precio mediano de una vivienda.

Lo ponemos en una variable categórica (floor\_cat).

```
In [397]: def conditions_floor(x: float) -> str:  
    """  
        Categorize floors of a property  
  
    Parameters:  
        x (Union[float, int]): Number of floors of a property  
  
    Returns:  
        str: Categorized floor of the property  
    """  
    if pd.isna(x): # check for NaN values  
        return "Unknown"  
    elif x < 0: # viviendas dónde no tengo valores (missing values)  
        return "Unknown"  
    elif ((x >= 0) & (x <= 4)): # viviendas entre 0 y 4 pisos  
        return "0 - 4"  
    elif ((x >= 5) & (x <= 11)): # viviendas entre 5 y 11 pisos  
        return "5 - 11"  
    elif ((x > 11)): # viviendas con más de 11 pisos  
        return "+11"  
    else:  
        return "Unknown"
```

In [402]: # aplicamos la función a la columna 'floor' del DataFrame  
sale\_clean\_df['floor\_cat'] = sale\_clean\_df['floor'].apply(conditions\_  
sale\_clean\_df.head(10)

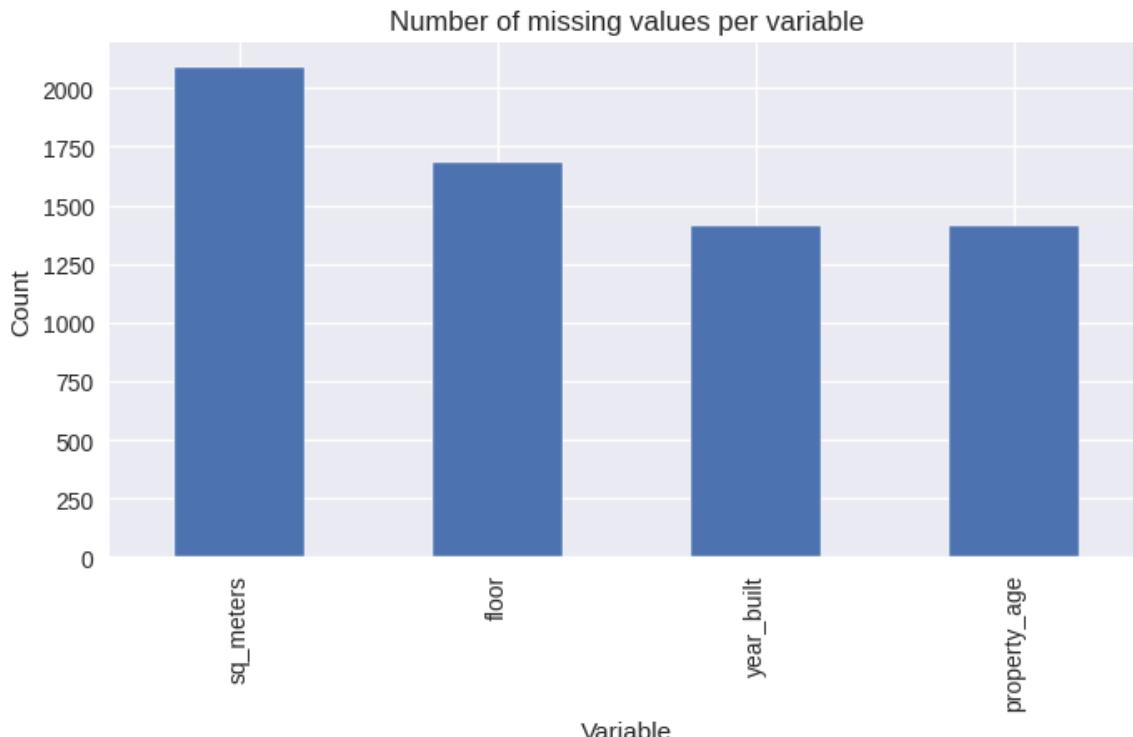
Out[402]:

	price	latitude	longitude	sq_meters	sq_meters_built	rooms	bathrooms	balcony	ter
0	2290000	41.409071	2.099850	383.0	532	5	6	1.0	
1	90000	41.434545	2.171110	42.0	50	1	1	0.0	
2	145000	41.444518	2.175309	NaN	53	3	1	1.0	
3	675000	41.392209	2.153368	93.0	120	4	2	0.0	
4	319000	41.413385	2.162246	NaN	69	3	1	0.0	
5	88000	41.423051	2.155127	61.0	64	2	1	0.0	
6	262000	41.411514	2.209901	74.0	80	3	1	1.0	
7	345000	41.402153	2.200675	78.0	87	3	1	1.0	
8	210000	41.405843	2.174957	50.0	55	2	1	0.0	
9	359000	41.399902	2.179506	53.0	61	0	1	0.0	

```
In [403]: # variables con missing values  
count_null_values(sale_clean_df)
```

Variables con valores nulos:

- 'sq\_meters': 2088
- 'floor': 1680
- 'year\_built': 1415
- 'property\_age': 1415



- Hemos eliminado los missing values de nuestro dataset.
- Nos quedan los valores nulos de las variables floor, year\_built y property\_age, pero hemos creado a partir de estas variables otras variables categóricas (floor\_cat y property\_age\_cat) sin missing values.
- Como hemos mencionado anteriormente, sq\_meters antes de tratarla miraremos, posteriormente, si puede estar correlacionada con sq\_meters\_built: si están correlacionadas, la eliminamos y nos quedamos con sq\_meters\_built ya que no tiene valores nulos.

### 1.10.3 Reducción de dimensionalidad de los barrios y categorización según su precio por metro cuadrado

Barrios de Barcelona en el dataset:

In [408]: # número de barrios en BCN  
`sale_clean_df['neighborhood'].nunique()`

Out[408]: 75

Tenemos muchas categorías de barrios (neighborhood), concretamente 75 barrios diferentes.

Tenemos que hacer una reducción de la dimensionalidad: una forma de hacer esta reducción de dimensionalidad podría ser mediante la creación de agrupaciones de barrios similares en términos de precios por metro cuadrado de vivienda (reducir el número de categorías con viviendas que tengan más o menos el mismo precio por metro cuadrado.). Para ello, haremos lo siguiente:

In [410]: `neighborhood_df = sale_clean_df[['neighborhood', 'price', 'sq_meters_built']]`  
`# Number of instances per category`  
`category_count = ('price', 'count'),`  
`# Mean price`  
`mean_price = ('price', 'mean'),`  
`# Median price`  
`mean_sq_meters_built = ('sq_meters_built', 'mean'),`  
`# Median price`  
`median_price = ('price', 'median'),`  
`# Median price`  
`median_sq_meters_built = ('sq_meters_built', 'median'))`  
  
`neighborhood_df['mean_price_per_sq_meters_built'] = neighborhood_df['mean_price'] / neighborhood_df['sq_meters_built']`  
`neighborhood_df['median_price_per_sq_meters_built'] = neighborhood_df['median_price'] / neighborhood_df['sq_meters_built']`  
`neighborhood_df.head(60)`

Out[410]:

	neighborhood	category_count	mean_price	mean_sq_meters_built	median_price	median_sq_meters_built
0	Baró de Viver	4	1.392500e+05	83.750000	150000.0	150000.0
1	Can Baró	20	1.055600e+05	49.150000	109750.0	109750.0
2	Can Peguera - El Turó de la Peira	68	1.248074e+05	61.279412	126000.0	126000.0
3	Canyelles	4	2.294750e+05	87.500000	194450.0	194450.0
4	Ciutat Meridiana - Torre Baró - Vallbona	98	1.141255e+05	66.571429	110000.0	110000.0
5	Ciutat Vella	219	3.862843e+05	74.666667	225000.0	225000.0
6	Diagonal Mar i el Front Marítim del	31	9.275484e+05	134.258065	850000.0	850000.0

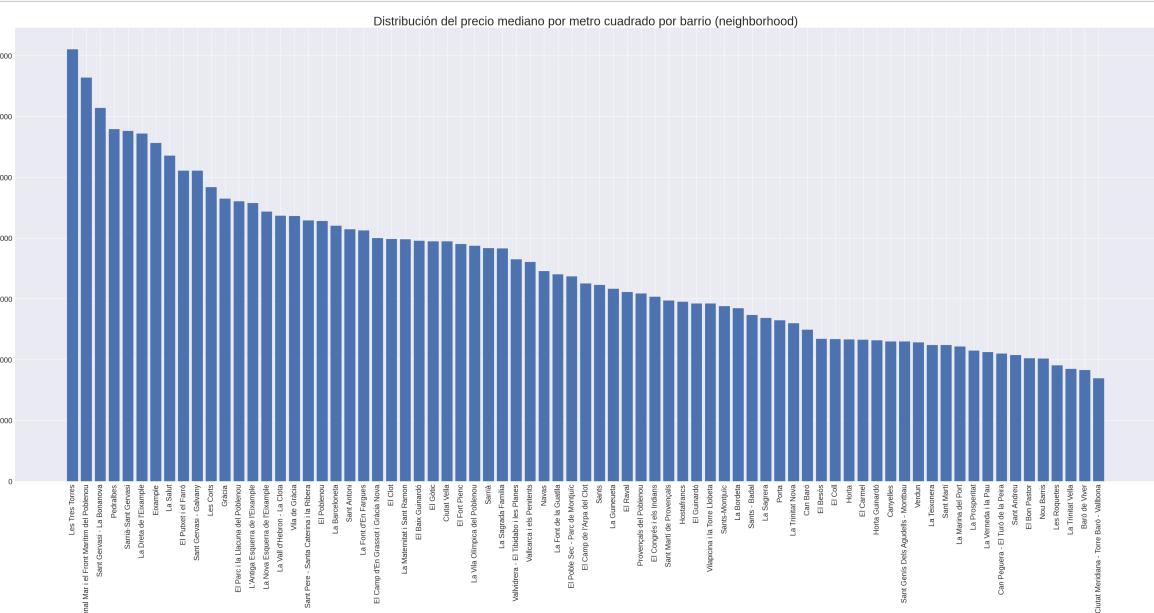
Distribución del precio mediano por metro cuadrado por barrio (neighborhood):

In [411]: # ordenamos los barrios por precio mediano por metro cuadrado de const  
neighborhood\_df[['neighborhood', 'median\_price\_per\_sq\_meters\_built']].s

Out[411]:

	index	neighborhood	median_price_per_sq_meters_built
0	51	Les Tres Torres	7105.263158
1	6	Diagonal Mar i el Front Marítim del Poblenou	6640.625000
2	61	Sant Gervasi - La Bonanova	6142.857143
3	54	Pedralbes	5789.473684
4	69	Sarrià-Sant Gervasi	5764.705882
5	32	La Dreta de l'Eixample	5720.524017
6	7	Eixample	5566.037736
7	42	La Salut	5357.142857
8	23	El Putxet i el Farró	5110.294118
9	60	Sant Gervasi - Galvany	5110.294118
10	49	Les Corts	4838.709677

In [412]: plt.bar(x = 'neighborhood',  
height = 'median\_price\_per\_sq\_meters\_built',  
data = neighborhood\_df.sort\_values('median\_price\_per\_sq\_meters\_built'))  
plt.rcParams["figure.figsize"] = (50,20)  
plt.xticks(fontsize=18, rotation=90)  
plt.yticks(fontsize=18)  
plt.title("Distribución del precio mediano por metro cuadrado por barrio")  
plt.show()



- Observamos una amplia variedad de precios en los diferentes barrios de Barcelona. El barrio más caro es Les Tres Torres con un precio mediano de 7105 €/m<sup>2</sup>, mientras que

el barrio más barato es la Ciutat Meridiana - Torre Baró - Vallbona con un precio mediano de 1692 €/m<sup>2</sup>.

Barrios que tienen el mayor precio mediano por metro cuadrado en Barcelona:

- Les Tres Torres
- Diagonal Mar i el Front Marítim del Poblenou
- Sant Gervasi - La Bonanova
- Pedralbes
- Sarrià - Sant Gervasi

Ahora miramos el número de viviendas en cada barrio:

```
In [413]: sale_clean_df['neighborhood'].value_counts()
```

```
Out[413]: Nou Barris          300
El Raval             273
Ciutat Vella        219
Sant Martí           212
Eixample            193
Sarrià-Sant Gervasi 143
Horta Guinardó      126
Les Corts            118
La Nova Esquerra de l'Eixample 115
El Carmel           113
El Poble Sec – Parc de Montjuïc 112
Sant Andreu          109
La Dreta de l'Eixample    104
Sants-Montjuïc        99
Ciutat Meridiana – Torre Baró – Vallbona 98
El Besòs              95
La Prosperitat         94
Les Roquetes          86
Sant Pere – Santa Caterina i la Ribera   85
Sant Gervasi – Galvany       85
La Sagrada Família       81
La Marina del Port       78
L'Antiga Esquerra de l'Eixample    72
Verdun                 71
La Barceloneta          70
Sants                  70
Can Peguera – El Turó de la Peira       68
Gràcia                62
El Gòtic               61
La Verneda i la Pau        60
Vila de Gràcia          59
Sants – Badal           57
Vilapicina i la Torre Llobeta     54
La Bordeta              47
Sant Antoni             46
El Camp d'En Grassot i Gràcia Nova   45
El Guinardó             44
El Poblenou              44
El Camp de l'Arpa del Clot        43
La Trinitat Nova          42
Sant Martí de Provençals      41
La Trinitat Vella          39
La Maternitat i Sant Ramon     37
El Bon Pastor             37
El Fort Pienc              35
La Sagrera                34
Porta                   32
La Teixonera              31
El Putxet i el Farró        31
Diagonal Mar i el Front Marítim del Poblenou 31
El Baix Guinardó          30
Provençals del Poblenou      28
El Clot                  25
Hostafrancs              25
```

Pedralbes	23
El Coll	23
Horta	21
Sant Genís Dels Agudells – Montbau	20
Can Baró	20
Sant Gervasi – La Bonanova	19
El Congrés i els Indians	18
El Parc i la Llacuna del Poblenou	17
Sarrià	16
La Guineueta	15
Navas	14
Les Tres Torres	13
Vallcarca i els Penitents	10
La Font de la Guatlla	10
La Font d'En Fargues	7
La Vila Olímpica del Poblenou	7
Vallvidrera – El Tibidabo i les Planes	4
Baró de Viver	4
Canyelles	4
La Vall d'Hebron – La Clota	2
La Salut	1
Name: neighborhood, dtype: int64	

- En algunos barrios tenemos muy pocas instancias (viviendas), por ejemplo en Canyelles, Vallvidrera - El Tibidabo i les Planes, Baró de Viver, La Salut y La Vall d'Hebron - La Clota.

Por tanto, cogemos categorías de barrios para asegurarnos que no tengamos barrios sin instancias y que el modelo no sepa qué hacer.

Definimos tiers de precios según la zona, dado que nos encontraremos con viviendas con pocas instancias.

#### **1.10.3.0.1 Método 1: Quantiles**

Una opción sería dividir los barrios en grupos en base a los quantiles del precio mediano por metro cuadrado construido. Por ejemplo, podríamos dividirlos en 3 grupos: bajo, medio y alto.

```
In [416]: # Definir los límites de cada rango de precios
bajo = neighborhood_df['median_price_per_sq_meters_built'].quantile(0.25)
medio = neighborhood_df['median_price_per_sq_meters_built'].quantile(0.5)
alto = neighborhood_df['median_price_per_sq_meters_built'].quantile(0.75)

print(f"bajo: {bajo}")
print(f"medio: {medio}")
print(f"alto: {alto}")
```

```
bajo: 2336.021505376344
medio: 3253.333333333335
alto: 4242.371103117506
```

```
In [418]: # Función para asignar tier
def assign_price_tier(x: float) -> str:
    if x < 0:
        return "Unknown"
    elif ((x >= 0) & (x < bajo)):
        return "0 - 2336"
    elif ((x >= bajo) & (x < medio)):
        return "2336 - 3253"
    elif ((x >= medio)):
        return "+3253"
    else:
        return "Unknown"
```

```
In [419]: # asignamos tiers
neighborhood_df['neighborhood_rent_index'] = neighborhood_df['median_p
neighborhood_df.head()
```

Out[419]:

	neighborhood	category_count	mean_price	mean_sq_meters_built	median_price	median
0	Baró de Viver	4	139250.000000	83.750000	150000.0	
1	Can Baró	20	105560.000000	49.150000	109750.0	
2	Can Peguera - El Turó de la Peira	68	124807.352941	61.279412	126000.0	
3	Canyelles	4	229475.000000	87.500000	194450.0	
4	Ciutat Meridiana - Torre Baró - Vallbona	98	114125.510204	66.571429	110000.0	

- El índice de renta de cada barrio (neighborhood\_rent\_index) se calcula como el precio mediano por metro cuadrado dividido por la mediana del tamaño en metros cuadrados construidos, se agrupan las viviendas por barrio y luego se asigna un tier de precios para cada barrio.

El objetivo era reducir la dimensión de la variable "neighborhood" agrupando los barrios que tengan precios similares. Para ello, hemos creado una nueva variable en el dataframe

"neighborhood\_df" que indica a qué categoría de precios pertenece cada barrio.

Para crear estas categorías, hemos utilizado la variable "median\_price\_per\_sq\_meters\_built", que es el precio mediano por metro cuadrado de cada barrio. Hemos creado 3 categorías de precios (tiers):

- Tier 1: Barrios con un precio mediano por metro cuadrado entre 0 y 2336.
- Tier 2: Barrios con un precio mediano por metro cuadrado entre 2336 y 3253.
- Tier 3: Barrios con un precio mediano por metro cuadrado mayor o igual a 3253.

Finalmente, hemos creado una nueva variable en el dataframe "neighborhood\_df" llamada "neighborhood\_rent\_index", que indica a qué tier de precios pertenece cada barrio.

#### **1.10.3.0.2 Método 2: Clustering**

Otra técnica comúnmente utilizada para dividir los datos en grupos es la segmentación de clústeres. Esta técnica agrupa los datos en diferentes clústeres según sus características similares.

En este caso, agrupamos los barrios según su precio mediano por metro cuadrado construido.

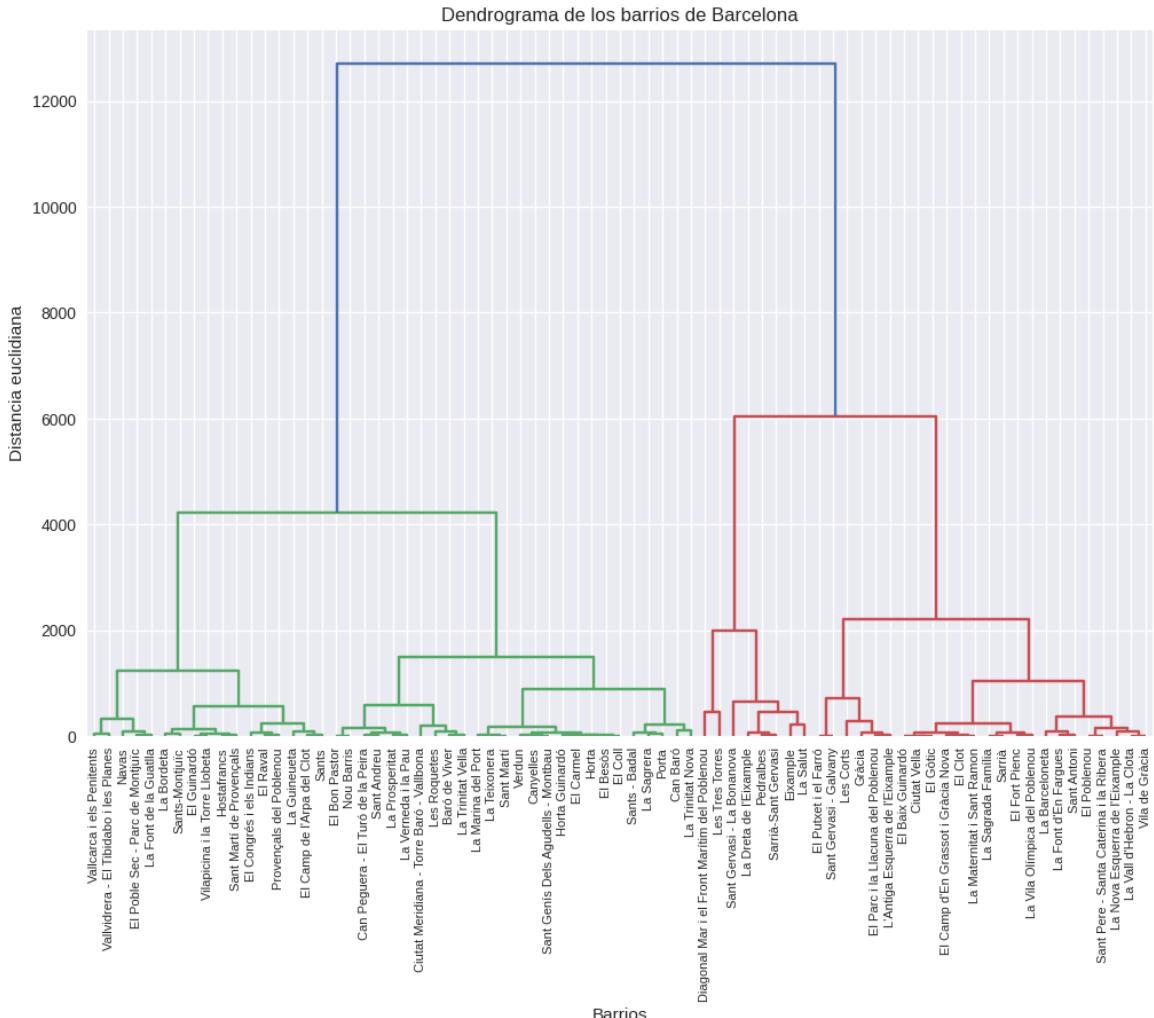
La segmentación de clústeres podría proporcionar una visión más detallada y precisa de cómo se distribuyen los barrios según el precio mediano por metro cuadrado construido.

```
In [422]: # Aplicar clustering jerárquico
X = neighborhood_df[['median_price_per_sq_meters_built']].values
Z = linkage(X, 'ward')

# Dibujar el dendrograma
plt.figure(figsize=(12, 8))
plt.title('Dendrograma de los barrios de Barcelona')
plt.xlabel('Barrios')
plt.ylabel('Distancia euclíadiana') # distancia euclidiana para calcular
dendrogram(Z, labels=neighborhood_df['neighborhood'].values, leaf_rotation=True)
plt.show()

# Seleccionar el número óptimo de clusters
max_d = 1500 # valor arbitrario para que los clusters estén bien diferenciados
clusters = fcluster(Z, max_d, criterion='distance')
n_clusters = len(np.unique(clusters))

# Asignar los clusters a cada barrio
neighborhood_df['price_cluster'] = clusters
neighborhood_df = neighborhood_df.rename(columns={'price_cluster': 'cluster'})
# Imprimir los resultados
print(f"El número óptimo de clusters es {n_clusters}")
display(neighborhood_df[['neighborhood', 'median_price_per_sq_meters_built', 'cluster']])
```



El número óptimo de clusters es 6

	neighborhood	median_price_per_sq_meters_built	cluster_median_price_sqm_built
0	Baró de Viver	1829.268293	2
1	Can Baró	2494.318182	2
2	Can Peguera - El Turó de la Peira	2100.000000	2
3	Canyelles	2301.183432	2
4	Ciutat Meridiana - Torre Baró - Vallbona	1692.307692	2
5	Ciutat Vella	3947.368421	6
6	Diagonal Mar i el Front Marítim del Poblenou	6640.625000	3
7	Eixample	5566.037736	4
8	El Baix Guinardó	3956.834532	6
9	El Besòs	2344.262295	2
10	El Bon Pastor	2023.437500	2
11	El Camp d'En Grassot i Gràcia Nova	4000.000000	6
12	El Camp de l'Arpa del Clot	3253.333333	1
13	El Carmel	2328.125000	2
14	El Clot	3984.375000	6

- Observamos que el número óptimo de clusters es 6 y que cada barrio ha sido asignado a un cluster específico según su precio mediano por metro cuadrado construido.
- Podemos ver, por ejemplo, que los barrios Baró de Viver, Can Peguera - El Turó de la Peira, Canyelles y Ciutat Meridiana - Torre Baró - Vallbona han sido asignados al cluster 2. Estos barrios tienen un precio mediano por metro cuadrado construido relativamente bajo en comparación con los otros clusters, lo que significa que son zonas más baratas para vivir.
- Por ejemplo, el cluster 3 incluye los barrios de Diagonal Mar i el Front Marítim del Poblenou, entre otros. Estos barrios tienen un precio mediano por metro cuadrado construido relativamente alto en comparación con los otros clusters, lo que sugiere que son zonas más costosas para vivir. Además, el hecho de que estén en un mismo cluster indica que tienen características similares en términos de precios de la vivienda.

Ajustamos el modelo KMeans con 6 clusters y obtenemos las etiquetas de los clusters para cada barrio en base a su precio mediano por metro cuadrado construido:

```
In [423]: # Instanciar el modelo con 6 clusters
kmeans = KMeans(n_clusters=6, random_state=42)

# Ajustar el modelo con los datos de los barrios
kmeans.fit(X)

# Obtener las etiquetas de los clusters para cada observación
labels = kmeans.labels_
```

```
In [424]: print(labels)
print(len(labels))

[0 0 0 0 0 1 4 2 1 0 0 1 3 0 1 0 3 1 3 1 5 3 1 5 3 5 0 0 3 5 1 3 2 1
3 3 0
1 5 0 1 3 2 0 3 0 1 0 1 5 0 4 3 0 2 3 3 0 1 0 5 2 0 3 1 3 3 3 1 2 1
1 0 1
3]
75
```

Visualizamos los clústers:

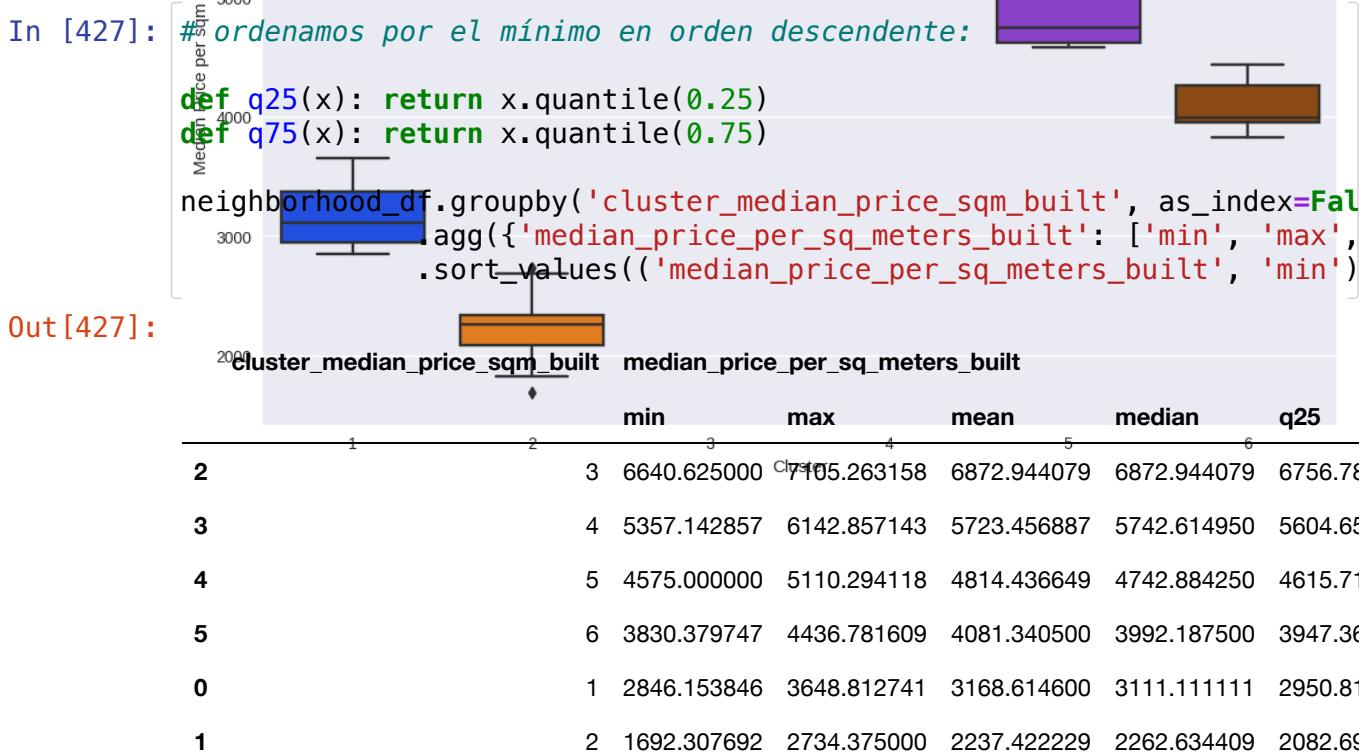
```
In [425]: # Scatter plot
plt.figure(figsize=(12,8))

sns.scatterplot(data=neighborhood_df,
                 x='median_price_per_sq_meters_built',
                 y='category_count',
                 hue='cluster_median_price_sqm_built',
                 palette='bright')

plt.title('Clustering of Neighborhoods by Median Price and Number of Instances')
plt.xlabel('Median Price per sqm built')
plt.ylabel('Number of Instances')
plt.show()

# Boxplot
plt.figure(figsize=(12,8))
sns.boxplot(data=neighborhood_df, x='cluster_median_price_sqm_built',
            plt.title('Clustering of Neighborhoods by Median Price per Sq. Meter Built')
            plt.xlabel('Cluster')
            plt.ylabel('Median Price per sqm built')
            plt.show()
```





- Tenemos 6 clusters.
- Por orden de menor a mayor precio mediano por metro cuadrado:
  - cluster 3
  - cluster 4
  - cluster 5
  - cluster 6
  - cluster 1
  - cluster 2
- Observamos lo siguiente:
  - El precio mediano por metro cuadrado construido del cluster 1 oscila entre 2846.15 y 3648.81 euros. El precio mediano es de 3168.61 euros, y el rango intercuartílico va desde los 2950.82 a los 3371.21 euros.
  - El precio mediano por metro cuadrado construido del cluster 2 oscila entre 1692.31 y 2734.38 euros. El precio mediano es de 2262.63 euros, y el rango intercuartílico va desde los 2082.69 a los 2337.37 euros.
  - El precio mediano por metro cuadrado construido del cluster 3 oscila entre 6640.62 y 7105.26 euros. El precio mediano es de 6872.94 euros, y el rango intercuartílico va desde los 6756.78 a los 6989.10 euros.
  - El precio mediano por metro cuadrado construido del cluster 4 oscila entre 5357.14 y 6142.86 euros. El precio mediano es de 5742.61 euros, y el rango intercuartílico va desde los 5604.66 a los 5783.28 euros.
  - El precio mediano por metro cuadrado construido del cluster 5 oscila entre 4575.00 y 5110.29 euros. El precio mediano es de 4742.88 euros, y el rango intercuartílico va desde los 4615.71 a los 5042.40 euros.
  - El precio mediano por metro cuadrado construido del cluster 6 oscila entre 3830.38 y 4436.78 euros. El precio mediano es de 3992.19 euros, y el rango intercuartílico

va desde los 3947.37 a los 4261.47 euros.

Definimos los grupos de precios:

```
In [428]: # Función para asignar tier
    # < 2337
    # >= 2337
    # >= 3371
    # >= 5042
    # + 5042

def assign_price_tier_cluster(x: float) -> str:
    if x < 0:
        return "Unknown"
    elif ((x >= 0) & (x < 2337)): # bajo
        return "0 - 2337" # cluster 2

    elif ((x >= 2337) & (x < 3371)): # medio - bajo
        return "2337 - 3371" # cluster 1

    elif ((x >= 3371) & (x < 5042)): # medio - alto
        return "3371 - 5042" # cluster 6 y 5

    elif ((x >= 5042)): # alto
        return "+5042" # cluster 4 y 3
    else:
        return "Unknown"
```

```
In [429]: # asignamos tiers
neighborhood_df['neighborhood_rent_index_cluster'] = neighborhood_df['neighborhood'].head()
```

Out[429]:

	neighborhood	category_count	mean_price	mean_sq_meters_built	median_price	median_
0	Baró de Viver	4	139250.000000	83.750000	150000.0	
1	Can Baró	20	105560.000000	49.150000	109750.0	
2	Can Peguera - El Turó de la Peira	68	124807.352941	61.279412	126000.0	
3	Canyelles	4	229475.000000	87.500000	194450.0	
4	Ciutat Meridiana - Torre Baró - Vallbona	98	114125.510204	66.571429	110000.0	

Al agrupar los barrios en clusters según su precio mediano por metro cuadrado construido y luego calcular los estadísticos de cada cluster (mínimo, máximo, media, mediana, primer cuartil y tercer cuartil), hemos logrado definir tiers de precios según la zona. Esto nos permitirá tener una mejor comprensión de los precios de las viviendas en cada cluster.

Finalmente agregamos esta información el modelo de datos:

```
In [436]: neighborhood_mapping = neighborhood_df[['neighborhood', 'neighborhood']
sale_clean_df = pd.merge(sale_clean_df, neighborhood_mapping, on='neig
sale_clean_df.head()
```

Out[436]:

	price	latitude	longitude	sq_meters	sq_meters_built	rooms	bathrooms	balcony	ter
0	2290000	41.409071	2.099850	383.0	532	5	6	1	
1	90000	41.434545	2.171110	42.0	50	1	1	0	
2	145000	41.444518	2.175309	NaN	53	3	1	1	
3	675000	41.392209	2.153368	93.0	120	4	2	0	
4	319000	41.413385	2.162246	NaN	69	3	1	0	

```
In [439]: # instancias en cada grupo:
sale_clean_df['neighborhood_rent_index_cluster'].value_counts()
```

Out[439]: 0 - 2337        1571  
3371 - 5042        1403  
2337 - 3371        1060  
+5042                643  
Name: neighborhood\_rent\_index\_cluster, dtype: int64

In [440]: `sale_clean_df.info()`

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4677 entries, 0 to 4676
Data columns (total 29 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   price            4677 non-null    int64  
 1   latitude          4677 non-null    float64 
 2   longitude         4677 non-null    float64 
 3   sq_meters         2589 non-null    float64 
 4   sq_meters_built  4677 non-null    int64  
 5   rooms             4677 non-null    int64  
 6   bathrooms          4677 non-null    int64  
 7   balcony            4677 non-null    category
 8   terrace            4677 non-null    category
 9   exterior           4677 non-null    category
 10  orientation        4677 non-null    object  
 11  floor              2997 non-null    float64 
 12  rooftop            4677 non-null    category
 13  elevator           4677 non-null    category
 14  pool               4677 non-null    category
 15  ac                 4677 non-null    category
 16  heating             4677 non-null    object  
 17  year_built         3262 non-null    float64 
 18  neighborhood        4677 non-null    object  
 19  dist_city_center   4677 non-null    float64 
 20  garage              4677 non-null    category
 21  property_type      4677 non-null    object  
 22  garden              4677 non-null    category
 23  dist_closest_station 4677 non-null    float64 
 24  property_age        3262 non-null    float64 
 25  property_age_cat   4677 non-null    object  
 26  floor_cat           4677 non-null    object  
 27  neighborhood_rent_index 4677 non-null    object  
 28  neighborhood_rent_index_cluster 4677 non-null    object  
dtypes: category(9), float64(8), int64(4), object(8)
memory usage: 809.5+ KB
```

Convertimos las categóricas de object a category:

```
In [5]: # convertimos las categóricas de object a category:  
def convert_object_to_category(df: pd.DataFrame) -> pd.DataFrame:  
    """  
        Convert object columns in a pandas dataframe to category data type  
  
    Parameters:  
    df (pd.DataFrame): Input pandas dataframe  
  
    Returns:  
    pd.DataFrame: A copy of the input dataframe with object columns converted to categories  
    """  
    # loop through columns and convert object columns to category  
    for col in df.columns:  
        if df[col].dtype == 'object':  
            df[col] = df[col].astype('category')  
  
    return df
```

```
In [442]: sale_clean_df = convert_object_to_category(sale_clean_df)
```

```
In [444]: sale_clean_df.dtypes # orient, neighb, proptype, heating, propagecat,
```

```
Out[444]: price                      int64  
latitude                     float64  
longitude                     float64  
sq_meters                     float64  
sq_meters_built                  int64  
rooms                        int64  
bathrooms                     int64  
balcony                       category  
terrace                        category  
exterior                       category  
orientation                     category  
floor                          float64  
rooftop                        category  
elevator                       category  
pool                           category  
ac                            category  
heating                         category  
year_built                      float64  
neighborhood                     category  
dist_city_center                  float64  
garage                          category  
property_type                     category  
garden                          category  
dist_closest_station                 float64  
property_age                      float64  
property_age_cat                   category  
floor_cat                        category  
neighborhood_rent_index                 category  
neighborhood_rent_index_cluster           category  
dtype: object
```

## 1.11 Correlation Analysis

Hacemos un análisis de correlaciones para saber como conviven las variables.

Realizar un análisis de correlación es una técnica útil para comprender cómo las diferentes variables interactúan entre sí. Este análisis nos permite identificar las variables que están significativamente asociadas con la variable objetivo y evaluar el grado de correlación entre las diferentes variables predictoras.

El análisis de correlación puede ayudarnos a descubrir relaciones lineales entre variables, lo que puede ser útil para seleccionar las variables más relevantes y eliminar aquellas que no contribuyen significativamente al modelo. También puede ayudarnos a detectar variables altamente correlacionadas que podrían estar duplicando información y afectar negativamente la calidad de las predicciones.

```
In [508]: def plot_corr_barchart(df1: pd.DataFrame, target: str, n: int = 10) -  
    """  
        Plots a color-gradient bar chart showing top n correlations between  
        Args:  
            df1 (pd.DataFrame): the dataframe to plot  
            n (int): number of top n correlations to plot  
        Returns:  
            None  
        Sources:  
            https://typefully.com/levikul09/j6qzwR0  
            https://stackoverflow.com/questions/17778394/list-highest-correlat  
    """  
        print(f"Correlation between numeric features (excluding the target  
        # drop target  
        df1 = df1.drop(columns=target)  
  
        # select only the numeric features  
        useful_columns = df1.select_dtypes(include=['int64', 'float64']).  
  
        def get_redundant_pairs(df):  
            pairs_to_drop = set()  
            cols = df.columns  
            for i in range(0,df.shape[1]):  
                for j in range(0,i+1):  
                    pairs_to_drop.add((cols[i],cols[j]))  
            return pairs_to_drop  
  
        def get_correlations(df,n=n):  
            au_corr = df.corr(method = 'spearman').unstack() # spearman us  
            labels_to_drop = get_redundant_pairs(df)  
            au_corr = au_corr.drop(labels = labels_to_drop).sort_values(as  
            top_n = au_corr[0:n]  
            bottom_n = au_corr[-n:]  
            top_corr = pd.concat([top_n, bottom_n])  
            return top_corr  
  
        corrplot = get_correlations(df1[useful_columns])  
  
        fig, ax = plt.subplots(figsize=(20,15))  
        norm = TwoSlopeNorm(vmin=-1, vcenter=0, vmax =1)  
        colors = [plt.cm.RdYlGn(norm(c)) for c in corrplot.values]  
  
        # n*(n-1)/2--> maximum value of "n" for which there are no repeat  
        # n = number of numerical variables = len(useful_columns)  
        num_corr = (len(useful_columns)*(len(useful_columns)-1)/2)  
        print(f"Max number of correlations (n): {num_corr/2}\n")  
  
        print(corrplot)  
  
        corrplot.plot.barh(color=colors)
```

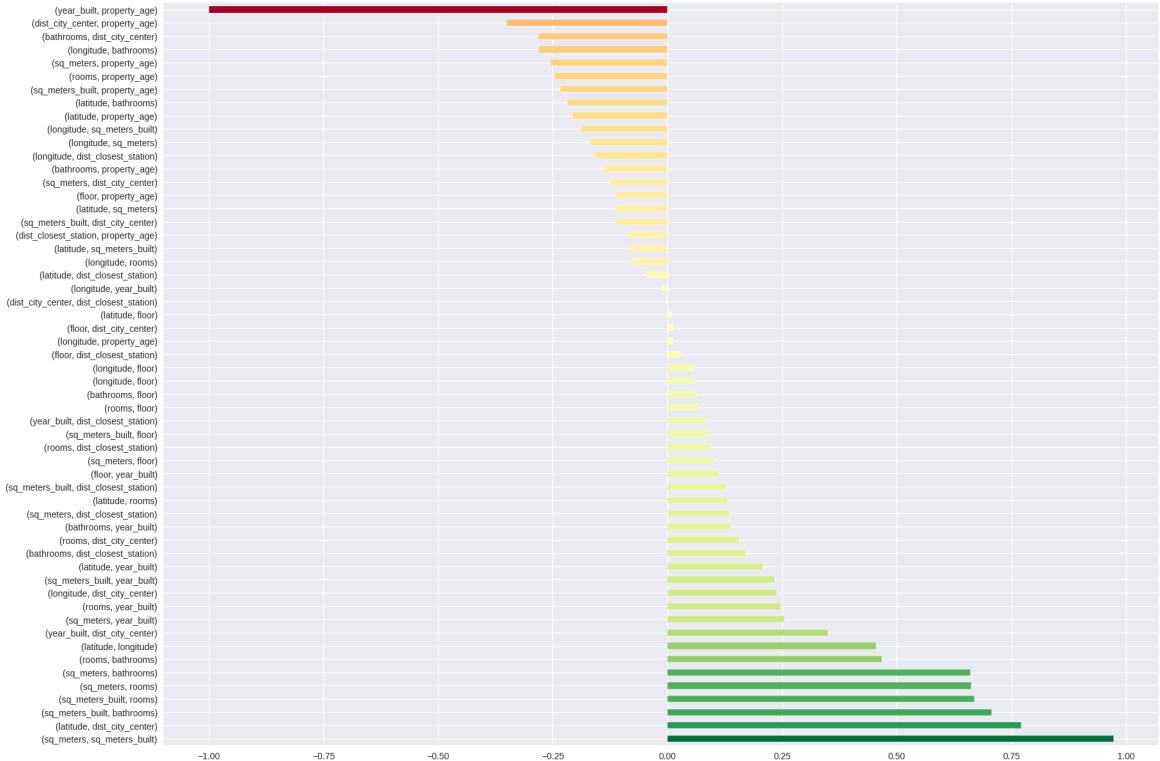
```
In [506]: # correlación entre las variables numéricas (excluyendo el precio)
plot_corr_barchart(sale_clean_df, target = TARGET_VARIABLE, n=28) # sp
```

Correlation between numeric features (excluding the target: price)

Max number of correlations (n): 27.5

sq_meters	sq_meters_built	0.973256
latitude	dist_city_center	0.770960
sq_meters_built	bathrooms	0.707718
	rooms	0.669373
sq_meters	rooms	0.661800
	bathrooms	0.660900
rooms	bathrooms	0.467502
latitude	longitude	0.454752
year_built	dist_city_center	0.350263
sq_meters	year_built	0.254260
rooms	year_built	0.246641
longitude	dist_city_center	0.237770
sq_meters_built	year_built	0.234041
latitude	year_built	0.207303
bathrooms	dist_closest_station	0.171199
rooms	dist_city_center	0.157014
bathrooms	year_built	0.138837
sq_meters	dist_closest_station	0.135769
latitude	rooms	0.131575
sq_meters_built	dist_closest_station	0.128468
floor	year_built	0.112753
sq_meters	floor	0.100506
rooms	dist_closest_station	0.094520
sq_meters_built	floor	0.091488
year_built	dist_closest_station	0.085339
rooms	floor	0.068118
bathrooms	floor	0.063666
longitude	floor	0.058956
	floor	0.058956
floor	dist_closest_station	0.029524
longitude	property_age	0.013011
floor	dist_city_center	0.012349
latitude	floor	0.008817
dist_city_center	dist_closest_station	-0.003384
longitude	year_built	-0.013011
latitude	dist_closest_station	-0.046087
longitude	rooms	-0.077787
latitude	sq_meters_built	-0.079186
dist_closest_station	property_age	-0.085339
sq_meters_built	dist_city_center	-0.112460
latitude	sq_meters	-0.112577
floor	property_age	-0.112753
sq_meters	dist_city_center	-0.124698
bathrooms	property_age	-0.138837
longitude	dist_closest_station	-0.157681
	sq_meters	-0.167688
latitude	sq_meters_built	-0.188934
	property_age	-0.207303
	bathrooms	-0.218992

```
sq_meters_built          property_age      -0.234041
rooms                   property_age      -0.246641
sq_meters                property_age      -0.254260
longitude                 bathrooms       -0.280341
bathrooms                dist_city_center -0.281263
dist_city_center           property_age      -0.350263
year_built                property_age     -1.000000
dtype: float64
```



```
In [511]: def plot_corr_vs_target(df1: pd.DataFrame, target: str, n: int) -> None
.....
    Plots a color-gradient bar chart showing top n correlations between
    Args:
        target (str): the name of the target column
        df1 (pd.DataFrame): the dataframe to plot
        n (int): number of top n correlations to plot
    Returns:
        None
.....
    print(f"Correlation between numeric features and the target variab

    # select the number of numeric features
    useful_columns = len(df1.select_dtypes(include=['int64', 'float64'])
    print(f"Max number of correlations (n): {useful_columns/2}\n")

    target_series = df1[target]

    x = df1.corrwith(target_series, method = 'spearman', numeric_only=True)
    top_n = x[0:n]
    bottom_n = x[-n:]
    top_corr = pd.concat([top_n, bottom_n])
    x = top_corr
    print(x)

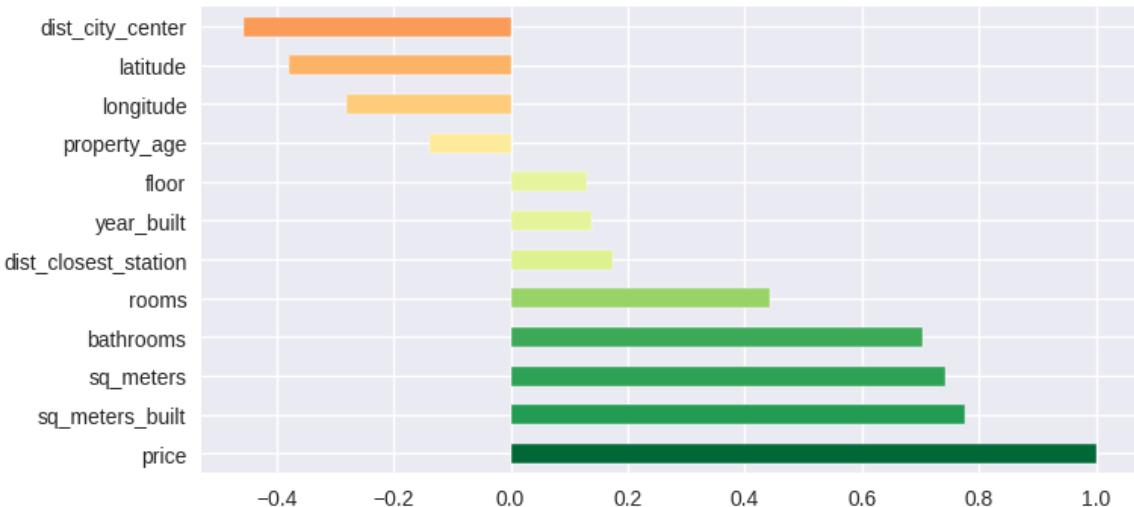
    fig, ax = plt.subplots(figsize=(8,4))
    norm = TwoSlopeNorm(vmin=-1, vcenter=0, vmax =1)
    colors = [plt.cm.RdYlGn(norm(c)) for c in x.values]
    x.plot.barh(color=colors)
```

```
In [512]: # Correlación entre las variables numéricas y la variable objetivo  
plot_corr_vs_target(sale_clean_df, target = TARGET_VARIABLE, n=6) # sp
```

Correlation between numeric features and the target variable price

Max number of correlations (n): 6.0

```
price                      1.000000  
sq_meters_built            0.774952  
sq_meters                  0.742625  
bathrooms                 0.704410  
rooms                      0.441490  
dist_closest_station       0.174439  
year_built                 0.138385  
floor                      0.128992  
property_age                -0.138385  
longitude                  -0.280358  
latitude                   -0.377873  
dist_city_center            -0.454986  
dtype: float64
```



```
In [450]: def calculate_correlation(df: pd.DataFrame, method: str, figsize: tuple):
    """
    Calculates the correlation matrix between numeric features in a pandas DataFrame.

    Parameters:
    df (pd.DataFrame): The DataFrame to calculate the correlation matrix for.
    method (str): The correlation method to use. Valid options are 'pearson', 'spearman', or 'kendall'.
    figsize (tuple): The size of the heatmap plot. Default is (6, 4).

    Returns:
    pd.DataFrame: The correlation matrix between numeric features in the DataFrame.
    """
    # select only the numeric features
    numeric_features = df.select_dtypes(include=['int64', 'float64'])
    X = numeric_features

    # scale the data
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

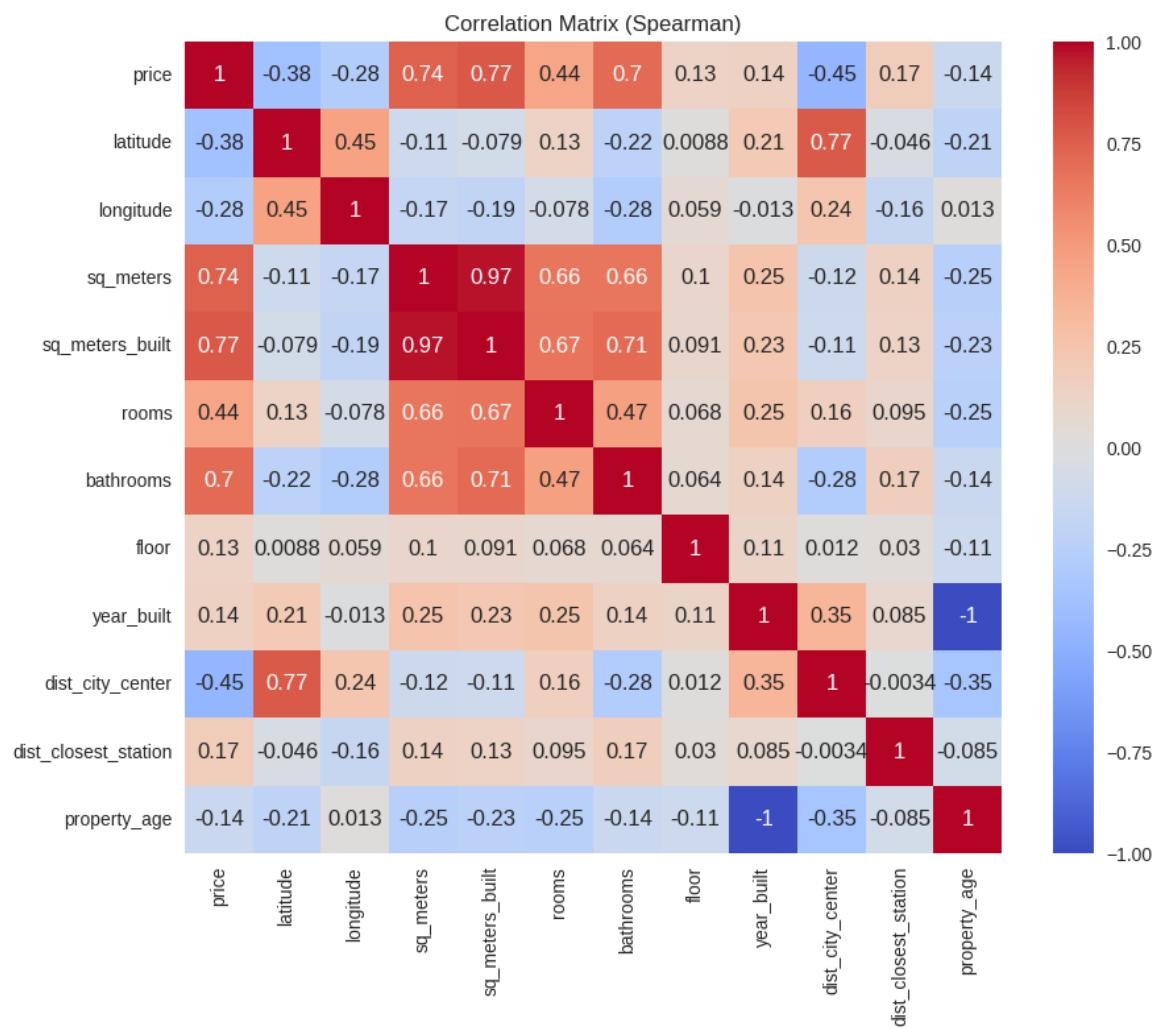
    # calculate correlation matrix
    if method == 'pearson':
        corr_matrix = pd.DataFrame(X_scaled).corr(method='pearson')
    elif method == 'spearman':
        corr_matrix = pd.DataFrame(X_scaled).corr(method='spearman')
    elif method == 'kendall':
        corr_matrix = pd.DataFrame(X_scaled).corr(method='kendall')
    else:
        raise ValueError("Invalid correlation method specified. Valid methods are 'pearson', 'spearman', or 'kendall'.")

    # set column names of correlation matrix
    corr_matrix.columns = numeric_features.columns
    corr_matrix.index = numeric_features.columns

    # plot heatmap
    plt.figure(figsize=figsize)
    sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
    plt.title(f"Correlation Matrix ({method.capitalize()})")
    plt.show()

    return corr_matrix
```

In [513]: `calculate_correlation(sale_clean_df, method="spearman", figsize= (10,8)`



Out[513]:

	price	latitude	longitude	sq_meters	sq_meters_built	rooms	bathrooms	floor	year_builtin	dist_city_center	dist_closest_station	property_age
price	1.000000	-0.377873	-0.280358	0.742625	0.774952	0.441490	0.704410	0.128992	0.138385	-0.454986	0.174439	-0.138385
latitude	-0.377873	1.000000	0.454752	-0.112577	-0.079186	0.131575	-0.218992	0.008817	0.207303	0.770960	-0.046087	-0.207303
longitude	-0.280358	0.454752	1.000000	-0.167688	-0.188934	0.661800	-0.280341	0.058956	0.254260	0.237770	-0.157681	0.013011
sq_meters	0.742625	-0.112577	-0.167688	1.000000	0.973256	0.669373	-0.124698	0.100506	0.254260	-0.124698	-0.112460	0.254260
sq_meters_built	0.774952	-0.079186	-0.188934	0.973256	1.000000	0.669373	-0.091488	0.068118	0.234041	-0.091488	-0.112460	0.234041
rooms	0.441490	0.131575	-0.077787	0.661800	0.669373	1.000000	-0.068118	0.068118	0.246641	-0.068118	-0.157014	0.246641
bathrooms	0.704410	-0.218992	-0.280341	0.660900	0.707718	0.467502	1.000000	-0.068118	0.246641	-0.068118	-0.157014	0.246641
floor	0.128992	0.008817	0.058956	0.100506	0.091488	0.068118	-0.068118	1.000000	0.246641	-0.068118	-0.157014	0.246641
year_builtin	0.138385	0.207303	-0.013011	0.254260	0.234041	0.246641	0.234041	0.246641	1.000000	-0.013011	-0.013011	0.246641
dist_city_center	-0.454986	0.770960	0.237770	-0.124698	-0.112460	0.157014	-0.112460	-0.157014	-0.013011	1.000000	-0.013011	-0.013011
dist_closest_station	0.174439	-0.046087	-0.157681	0.135769	0.128468	0.094520	-0.013011	-0.013011	0.246641	-0.013011	1.000000	-0.013011
property_age	-0.138385	-0.207303	0.013011	-0.254260	-0.234041	-0.246641	-0.013011	-0.013011	-0.013011	-0.013011	-0.013011	1.000000

Del análisis de correlación observamos lo siguiente:

- year\_built y property\_age en realidad son la misma variable pero expresada de distinta manera, por eso tenemos una correlación de -1. Deberemos eliminar year\_built ya que no nos proporciona tanta información como property\_age.
- sq\_meters y sq\_meters\_built tienen un coeficiente de correlación de 0.97, lo que indica que están casi perfectamente correlacionados de forma positiva. Deberemos elegir una y eliminar la otra.
- latitude y dist\_city\_center también están fuertemente correlacionadas positivamente.
- sq\_meters\_built está fuertemente correlacionada con bathrooms y rooms.
- Con la variable objetivo price, están correlacionadas positivamente sq\_meters\_built, sq\_meters, y bathrooms, lo que indica que podrían ser buenas predictoras de la variable objetivo price.

#### 1.11.0.1 Elección entre sq\_meters y sq\_meters\_built

Tenemos sq\_meters\_built (metros cuadrados construidos de la vivienda) y sq\_meters (metros cuadrados de la vivienda (habitables)).

El que nos interesa en realidad son los metros cuadrados habitables de la vivienda, no el construido (que siempre es mayor que el habitable).

No obstante, sq\_meters tiene valores nulos mientras que sq\_meters\_built no . Por tanto, podría ser una buena idea ver si sq\_meters y sq\_meters\_built están correlacionadas, y en el caso que lo estén, quedarnos con sq\_meters\_built. Como hemos visto en el análisis de correlación, sí que lo están. Volvamos a verlo:

#### Normality Tests:

- Jarque-Bera test
  - funciona bien en muestras grandes (normalmente superiores a 2000 observaciones).
- Quantile-Quantile Plot

```
In [514]: # Data preparation
sq_meters = sale_clean_df['sq_meters'].values
sq_meters_clean = sq_meters[~np.isnan(sq_meters)]
sq_meters_built = sale_clean_df['sq_meters_built'].values
sq_meters_built_clean = sq_meters_built[~np.isnan(sq_meters_built)]

# Check length of the data
print(len(sq_meters_clean))
print(len(sq_meters_built_clean))
```

2589

4677

```
In [515]: # Jarque-Bera test for Normality

# Function to check normality
def check_normality(pvalue):
    if pvalue > 0.05:
        print('Since the p-value > 0.05, we fail to reject the null hypothesis')
    else:
        print('Since p-value ≤ 0.05, we reject the null hypothesis i.e. the distribution is not normal/gaussian')

# Perform Jarque-Bera test
print(stats.jarque_bera(sq_meters_clean))
statistic, pvalue = jarque_bera(sq_meters_clean)
check_normality(pvalue)
print("\n")
print(stats.jarque_bera(sq_meters_built_clean))
statistic, pvalue = jarque_bera(sq_meters_built_clean)
check_normality(pvalue)
```

```
Jarque_beraResult(statistic=240077.538719649, pvalue=0.0)
Since p-value ≤ 0.05, we reject the null hypothesis i.e. we assume the distribution of our variable is not normal/gaussian.
```

```
Jarque_beraResult(statistic=328176.0017057933, pvalue=0.0)
Since p-value ≤ 0.05, we reject the null hypothesis i.e. we assume the distribution of our variable is not normal/gaussian.
```

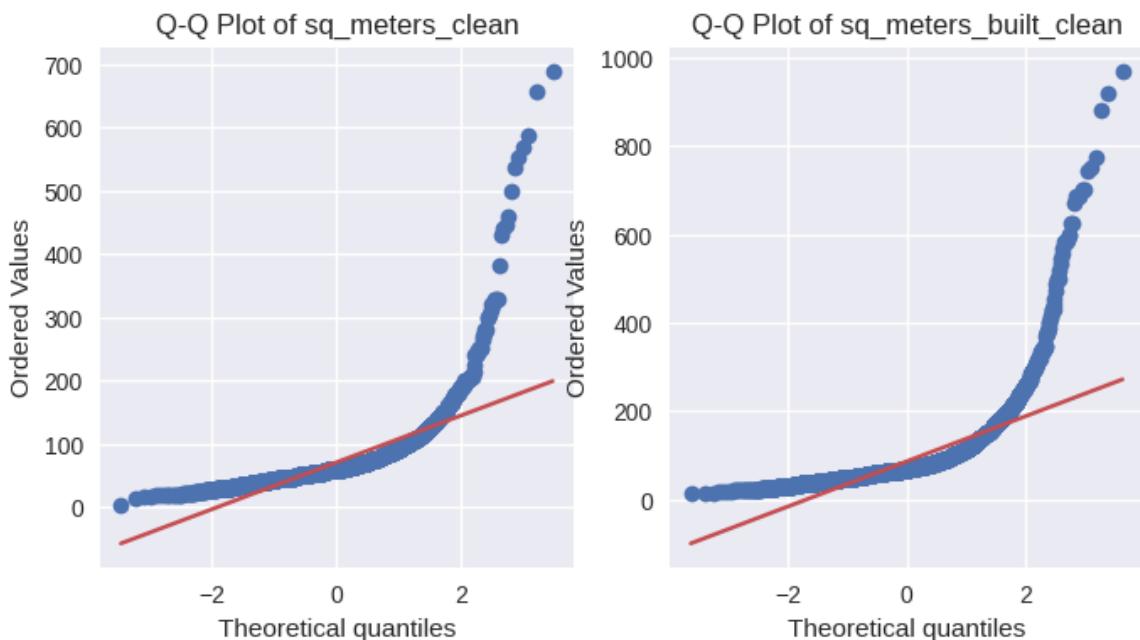
```
In [516]: # figure with two subplots
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4))

# first Q-Q plot on the left subplot
stats.probplot(sq_meters_clean, dist='norm', plot=ax1)
ax1.set_title('Q-Q Plot of sq_meters_clean')

# second Q-Q plot on the right subplot
stats.probplot(sq_meters_built_clean, dist='norm', plot=ax2)
ax2.set_title('Q-Q Plot of sq_meters_built_clean')

# titles
ax1.set_title('Q-Q Plot of sq_meters_clean')
ax2.set_title('Q-Q Plot of sq_meters_built_clean')

plt.show()
```



- Ni `sq_meters` ni `sq_meters_built` siguen una distribución normal.
- Debemos utilizar métodos no paramétricos de correlación de rangos: prueba rho de Spearman o prueba tau de Kendall.

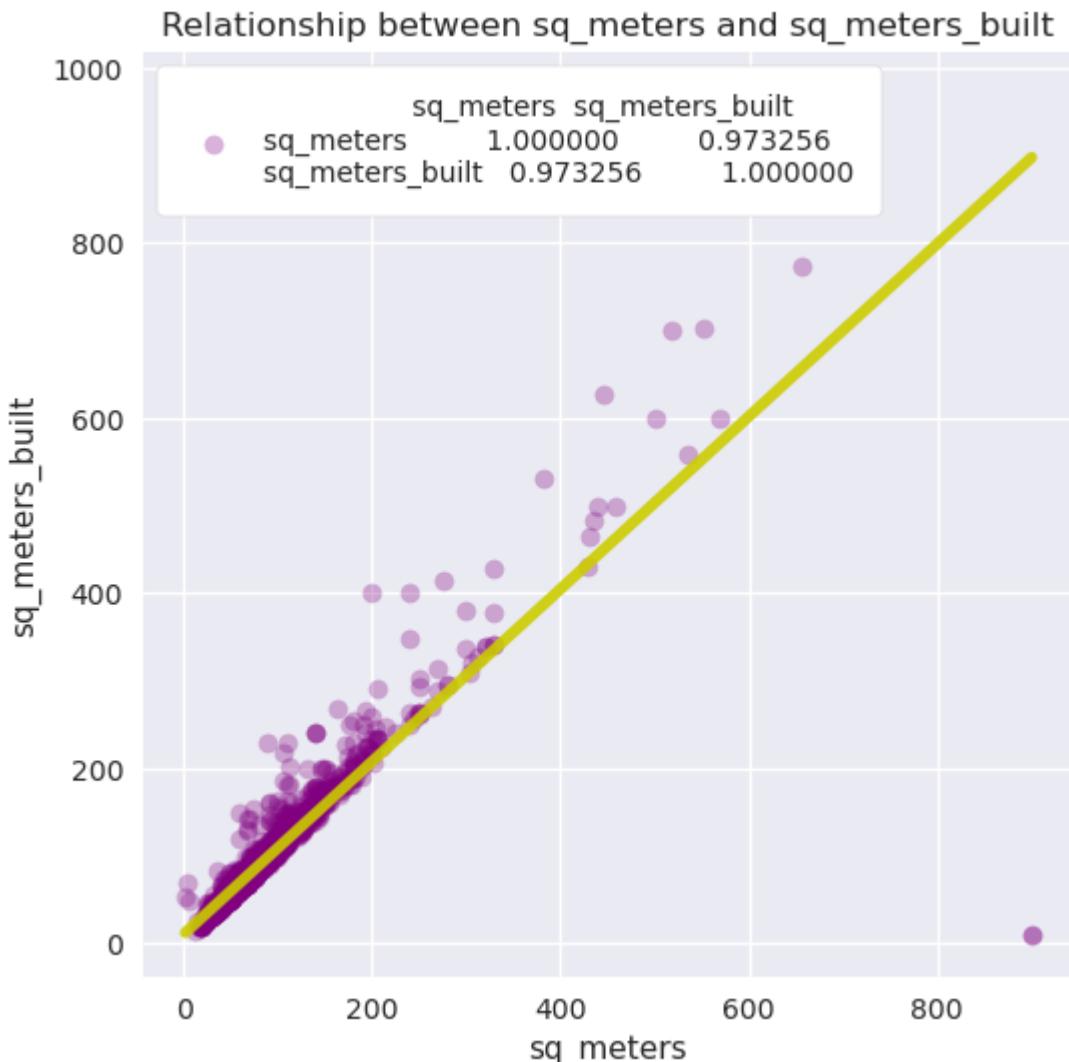
### Spearman's rho test

```
In [517]: # Spearman's rho test: correlación de Spearman entre sq_meters y sq_me
pg.corr(sale_clean_df['sq_meters'], sale_clean_df['sq_meters_built'],
```

Out[517]:

	n	r	CI95%	p-val	power
<b>spearman</b>	2589	0.973256	[0.97, 0.98]	0.0	1.0

```
In [518]: # Correlation scatter plot with the linear regression fit line
plt.figure(figsize=(6, 6))
fig = sns.regplot(x="sq_meters", y="sq_meters_built", data=raw_data, c
                   label="{}".format(sale_clean_df[['sq_meters', 'sq_m
                   fit_reg=True, scatter_kws = {'color': 'purple', 'alp
fig.set(xlabel='sq_meters', ylabel='sq_meters_built', title = "Relatio
sns.set_style('darkgrid')
plt.legend(facecolor='white', frameon=True, fancybox=True, framealpha=1
    # facecolor: The legend's background color
    # frameon: Whether the legend should be drawn on a patch (frame).
    # fancybox: Whether round edges should be enabled around the FancyB
    # framealpha: The alpha transparency of the legend's background
    # borderpad: The fractional whitespace inside the legend border, in
plt.show()
```



- El coeficiente de correlación de Spearman ( $\rho$ ) es de 0.97, lo que indica que hay una correlación muy fuerte y positiva entre las variables sq\_meters y sq\_meters\_built. Además, el p-value es 0, lo que significa que esta correlación es muy significativa estadísticamente.
- Por tanto, podemos decir que estas dos variables están altamente correlacionadas y que podría ser una buena idea quedarnos con sq\_meters\_built ya que no tiene valores

nulos y nos da la misma información que sq\_meters.

#### **1.11.0.2 Tratamiento de sq\_meters\_built en relación al número de habitaciones y de baños**

Queremos evitar tener variables autocorrelacionadas entre sí en nuestro modelo.

Como hemos visto en el análisis de correlación, sq\_meters\_built está correlacionada con bathrooms y rooms:

```
In [519]: # Creamos una matriz de correlación con los valores del dataset
corr_matrix = sale_clean_df[['sq_meters_built', 'bathrooms', 'rooms']]

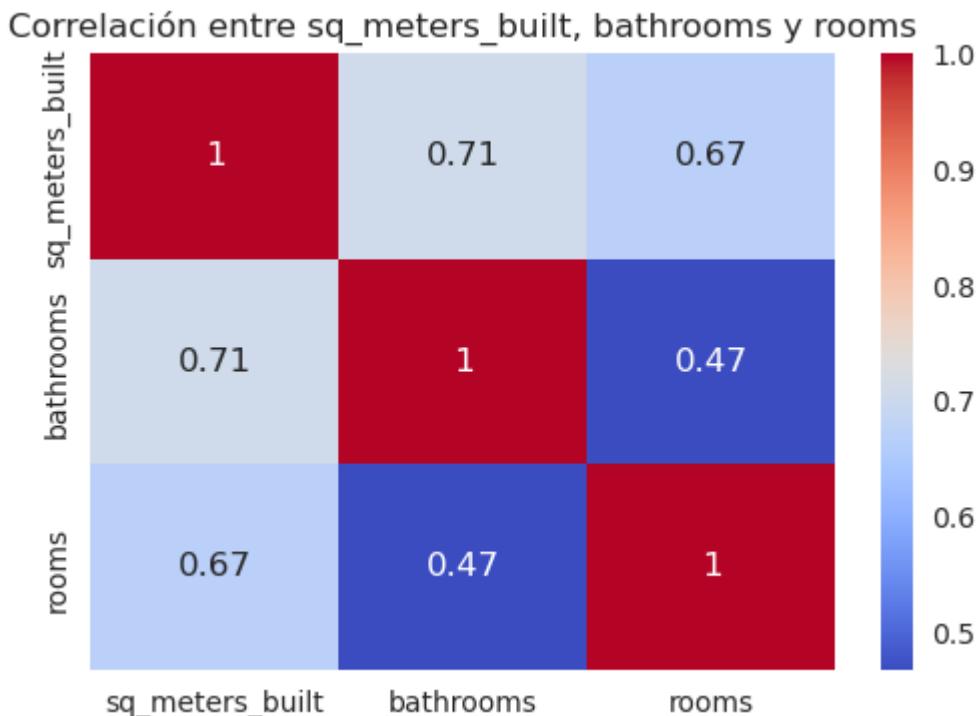
# Configuramos el tamaño de la figura
plt.figure(figsize=(6, 4))

# Generamos el heatmap
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')

# Añadimos el título
plt.title('Correlación entre sq_meters_built, bathrooms y rooms')

# Mostramos la figura
plt.show()

print(corr_matrix)
```



	sq_meters_built	bathrooms	rooms
sq_meters_built	1.000000	0.707718	0.669373
bathrooms	0.707718	1.000000	0.467502
rooms	0.669373	0.467502	1.000000

- En este caso, Spearman es más adecuado ya que no podemos asumir que las variables siguen una distribución normal.
- Como se puede observar en la matriz de correlación, la variable sq\_meters\_built está correlacionada con bathrooms y rooms.
- Podemos observar una correlación positiva fuerte entre sq\_meters\_built y bathrooms (0.71) y también una correlación moderada entre sq\_meters\_built y rooms (0.66).
- Es común que los inmuebles con más metros cuadrados construidos tengan más baños y habitaciones. A medida que aumenta el número de metros cuadrados construidos, también es probable que aumente el número de baños y habitaciones en el inmueble.

Sin embargo, debemos recordar que la correlación no implica causalidad, es decir, no podemos afirmar que el tamaño del inmueble es la única razón detrás del aumento en el número de baños y habitaciones.

Podemos decorrelar variables haciendo uso de las redes Bayesianas siempre y cuando encontremos una tercera variable que sea la causante del crecimiento o decrecimiento de las dos.

¿Qué causa qué una vivienda tenga más baños o habitaciones? Los metros cuadrados. En este caso, el tamaño de la vivienda nos servirá para decorrelar el número de habitaciones y el número de baños.

Por tanto, el número de habitaciones y el número de baños tienen relación con el número de metros cuadrados, entonces calculamos la proporción de habitaciones por metro cuadrado y baños por metro cuadrado. Es decir, vamos a poner habitaciones por metro cuadrado y baños por metro cuadrado. De esta forma estamos decorrelandolas de la que causa esto y le podemos dar más información al modelo.

Multiplicamos por 100 el número de habitaciones y de baños para que tenga más sentido: habitaciones y baños por cada 100 metros cuadrados.

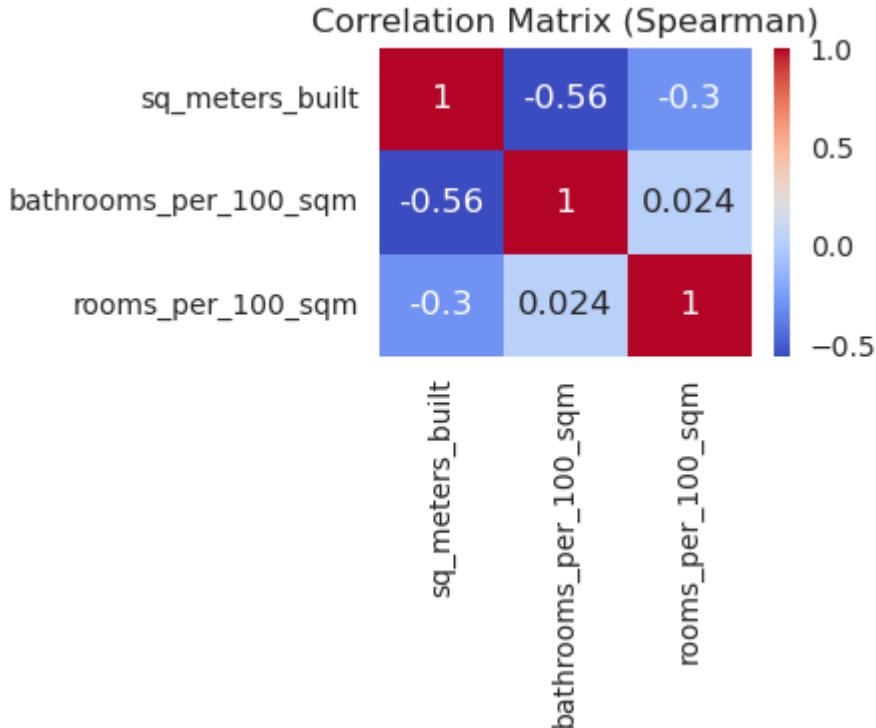
```
In [520]: sale_clean_df['rooms_per_100_sqm'] = sale_clean_df['rooms'] * 100 / sale_clean_df['sq_meters']  
sale_clean_df['bathrooms_per_100_sqm'] = sale_clean_df['bathrooms'] *
```

```
In [523]: display(sale_clean_df.head(3))  
print(sale_clean_df.shape)
```

	price	latitude	longitude	sq_meters	sq_meters_built	rooms	bathrooms	balcony	terrace
0	2290000	41.409071	2.099850	383.0		532	5	6	1
1	90000	41.434545	2.171110	42.0		50	1	1	0
2	145000	41.444518	2.175309	NaN		53	3	1	1
(4677, 31)									

- Ahora tenemos dos nuevas variables que expresan el número de habitaciones y baños por cada 100 metros cuadrados. Estas variables pueden ser más útiles para predecir el precio de venta, ya que proporcionan información más detallada sobre la relación entre la superficie y las habitaciones/baños.

```
In [542]: # correlación entre sq_meters_built y bathrooms_per_100_sqm y rooms_per_100_sqm
calculate_correlation(sale_clean_df[['sq_meters_built',
                                      'bathrooms_per_100_sqm',
                                      'rooms_per_100_sqm']],
                      method="spearman",
                      figsize= (3,2)) # spearman used because not all
```



Out [542]:

	sq_meters_built	bathrooms_per_100_sqm	rooms_per_100_sqm
sq_meters_built	1.000000	-0.563598	-0.304545
bathrooms_per_100_sqm	-0.563598	1.000000	0.023556
rooms_per_100_sqm	-0.304545	0.023556	1.000000

- Estas 3 variables (habitaciones por cada 100 metros cuadrados, baños por cada 100 metros cuadrados y número de metros cuadrados) no están correlacionadas entre sí.
  - probablemente tengan un poco de relación (las casas con muchos metros cuadrados a lo mejor tendrán más número de habitaciones por cada 100 metros cuadrados) pero no será como antes que teníamos una relación lineal.
  - como ahora estas 3 variables no están correlacionadas entre sí, podemos incluirlas en el modelo.

## 1.12 Distribución de las variables categóricas

Ahora que tenemos el dataset con las modificaciones adecuadas (tratamiento de missing values, eliminación de variables no relevantes, feature engineering y creación de nuevas variables), podemos visualizar la distribución de las variables categóricas.

Primero convertimos todas las variables que sean categóricas a 'category':

In [544]: `sale_clean_df.info()`

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4677 entries, 0 to 4676
Data columns (total 31 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   price            4677 non-null    int64  
 1   latitude          4677 non-null    float64 
 2   longitude         4677 non-null    float64 
 3   sq_meters         2589 non-null    float64 
 4   sq_meters_built  4677 non-null    int64  
 5   rooms             4677 non-null    int64  
 6   bathrooms          4677 non-null    int64  
 7   balcony            4677 non-null    category
 8   terrace            4677 non-null    category
 9   exterior           4677 non-null    category
 10  orientation        4677 non-null    category
 11  floor              2997 non-null    float64 
 12  rooftop            4677 non-null    category
 13  elevator           4677 non-null    category
 14  pool               4677 non-null    category
 15  ac                 4677 non-null    category
 16  heating             4677 non-null    category
 17  year_built         3262 non-null    float64 
 18  neighborhood        4677 non-null    category
 19  dist_city_center   4677 non-null    float64 
 20  garage              4677 non-null    category
 21  property_type      4677 non-null    category
 22  garden              4677 non-null    category
 23  dist_closest_station 4677 non-null    float64 
 24  property_age        3262 non-null    float64 
 25  property_age_cat   4677 non-null    category
 26  floor_cat           4677 non-null    category
 27  neighborhood_rent_index 4677 non-null    category
 28  neighborhood_rent_index_cluster 4677 non-null    category
 29  rooms_per_100_sqm   4677 non-null    float64 
 30  bathrooms_per_100_sqm 4677 non-null    float64 

dtypes: category(17), float64(10), int64(4)
memory usage: 631.0 KB
```

- Observamos que todas las variables categóricas ya están como 'category'.

Si alguna no estuviera como 'category' podríamos usar el siguiente pipeline para convertirlas al tipo de dato 'category' (no lo haremos ahora ya que ya están en el formato correcto, pero lo dejamos a modo de ejemplo o para futuros usos):

```
In [546]: # variables numéricas que son 0 y 1 y queremos convertir a categóricas

binary_features = [ 'balcony',
                     'terrace',
                     'exterior',
                     'rooftop',
                     'elevator',
                     'pool',
                     'ac',
                     'garage',
                     'garden'
                 ]
# data types
sale_clean_df[binary_features].dtypes
```

```
In [7]: # convertimos las categóricas de object a category:
def convert_object_to_category(df: pd.DataFrame) -> pd.DataFrame:
    """
    Convert object columns in a pandas dataframe to category data type

    Parameters:
    df (pd.DataFrame): Input pandas dataframe

    Returns:
    pd.DataFrame: A copy of the input dataframe with object columns converted to category
    """
    # loop through columns and convert object columns to category
    for col in df.columns:
        if df[col].dtype == 'object':
            df[col] = df[col].astype('category')

    return df

# convertimos las variables numéricas que son 0 y 1 a categóricas (binarias)
def convert_var_to_category(df: pd.DataFrame, cols: List[str]) -> pd.DataFrame:
    """
    Convert specified columns in a pandas dataframe to category data type

    Parameters:
    df (pd.DataFrame): Input pandas dataframe
    cols (List[str]): List of column names to convert

    Returns:
    pd.DataFrame: A copy of the input dataframe with specified columns converted to category
    """
    # loop through specified columns and convert columns to category
    # convertimos las variables numéricas que son 0 y 1 a categóricas
    for i in range(0, len(cols)):
        # primero las convertimos a int64 para eliminar los decimales:
        df[cols[i]] = df[cols[i]].astype("int64")
        # luego las convertimos a 'category': '0' y '1'
        df[cols[i]] = df[cols[i]].astype("category")

    return df
```

Definimos un pipeline para convertir las variables categóricas binarias a 'category', los tipos de dato 'object' a 'category' y obtener una lista con los nombres de las variables categóricas:

```
In [550]: # define the pipeline to obtain category variables
category_pipeline = Pipeline(steps=[
    ('convert_object_to_category', FunctionTransformer(convert_object),
     ('convert_var_to_category', FunctionTransformer(convert_var_to_cat),
      ('column_transformer', ColumnTransformer(transformers=[
          ('select_category', 'passthrough', sale_clean_df.select_dtypes
           )))
    )))
]

# define the transformer to obtain the category variable names
get_category_names = FunctionTransformer(
    lambda X, ct=category_pipeline.named_steps['column_transformer']:
        [ct.get_feature_names_out()[i][len('select_category_'):] for
         i
        ])

# add the transformer to the pipeline
category_pipeline.steps.append(('get_category_names', get_category_names))

# apply the pipeline to the sample dataframe
category_vars = category_pipeline.fit_transform(sale_clean_df)

# print the resulting category variable names and types
print(f'Data types:\n\n{sale_clean_df.dtypes}')
print('\n')
print(f'Categorical variables:\n\n{category_vars}')
print('\n')
print(f'Number of categorical variables: {len(category_vars)}')
```

Data types:

price	int64
latitude	float64
longitude	float64
sq_meters	float64
sq_meters_built	int64
rooms	int64
bathrooms	int64
balcony	category
terrace	category
exterior	category
orientation	category
floor	float64
rooftop	category
elevator	category
pool	category
ac	category
heating	category
year_built	float64
neighborhood	category
dist_city_center	float64
garage	category
property_type	category
garden	category
dist_closest_station	float64

```
property_age           float64
property_age_cat       category
floor_cat              category
neighborhood_rent_index category
neighborhood_rent_index_cluster category
rooms_per_100_sqm      float64
bathrooms_per_100_sqm   float64
dtype: object
```

Categorical variables:

```
['balcony', 'terrace', 'exterior', 'orientation', 'rooftop', 'elevator', 'pool', 'ac', 'heating', 'neighborhood', 'garage', 'property_type', 'garden', 'property_age_cat', 'floor_cat', 'neighborhood_rent_index', 'neighborhood_rent_index_cluster']
```

Number of categorical variables: 17

```
In [553]: # variables categóricas
category_vars
```

```
Out[553]: ['balcony',
'terrace',
'exterior',
'orientation',
'rooftop',
'elevator',
'pool',
'ac',
'heating',
'neighborhood',
'garage',
'property_type',
'garden',
'property_age_cat',
'floor_cat',
'neighborhood_rent_index',
'neighborhood_rent_index_cluster']
```

Binary variables:

```
In [554]: # Define the number of rows and columns for the subplot grid
num_plots = len(binary_features)
num_rows = 5
num_cols = 2

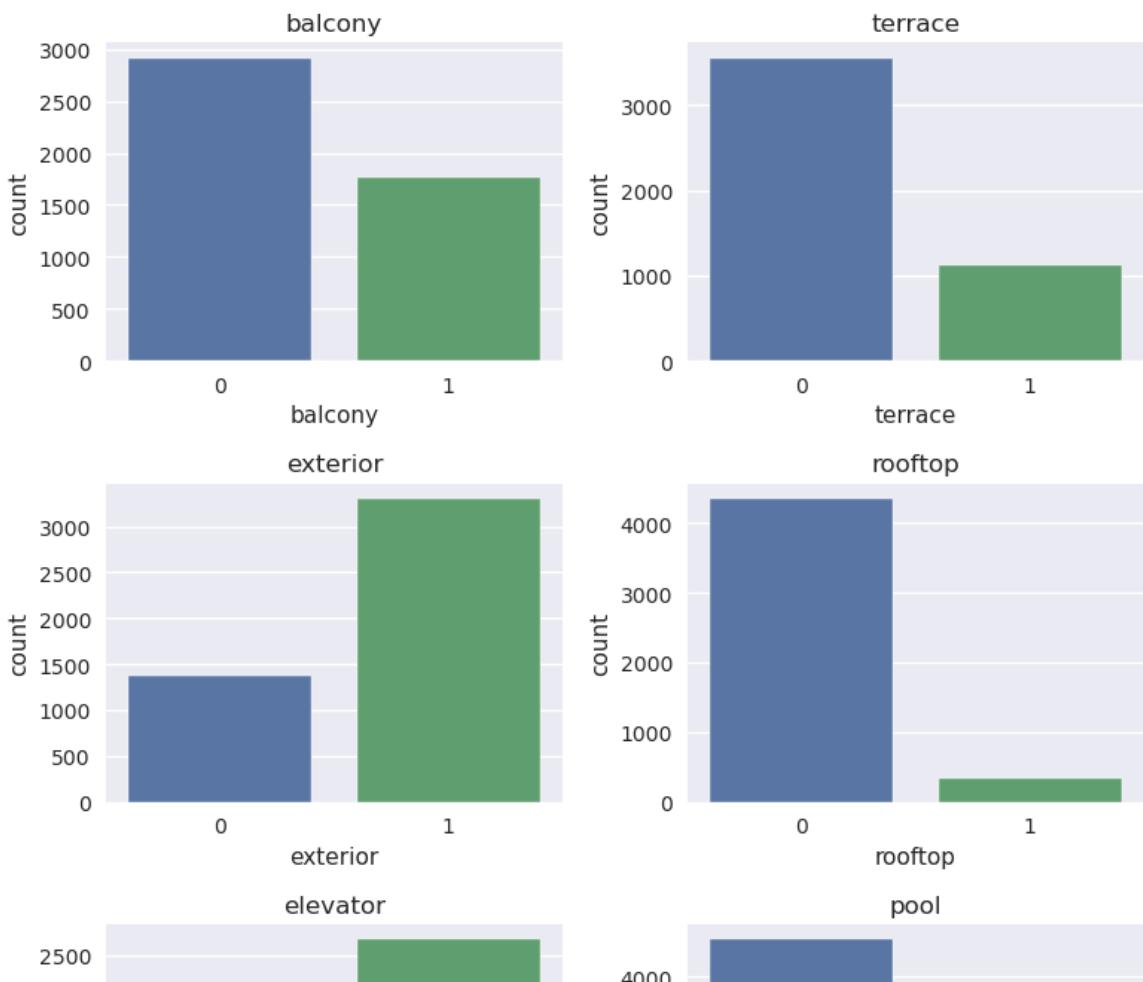
# Calculate the number of empty subplots to remove
num_empty_plots = num_rows * num_cols - num_plots

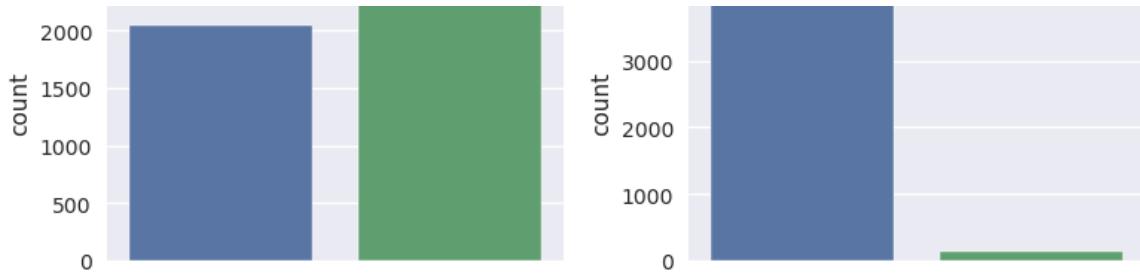
# Create the subplot grid and flatten it into a 1D array
fig, axs = plt.subplots(nrows=num_rows, ncols=num_cols, figsize=(8, 15))
axs = axs.flatten()

# Plot the data on each subplot
for i, feature in enumerate(binary_features):
    sns.countplot(data=sale_clean_df, x=feature, ax=axs[i])
    axs[i].set_title(feature)

# Remove the extra empty subplots
for i in range(num_plots, num_plots + num_empty_plots):
    fig.delaxes(axs[i])

# Adjust the spacing between the subplots and display the figure
plt.tight_layout()
plt.show()
```



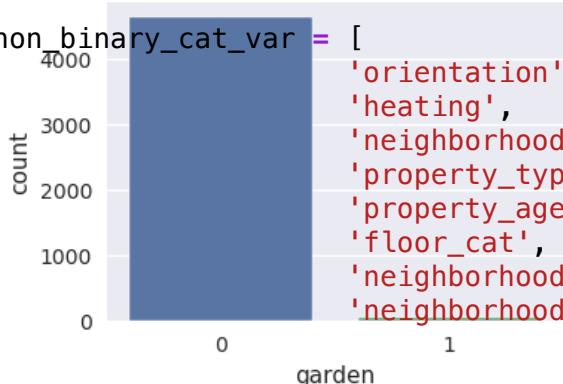


```
In [555]: sale_clean_df.columns
```

```
Out[555]: Index(['price', 'latitude', 'longitude', 'sq_meters', 'sq_meters_built', 'rooms', 'bathrooms', 'balcony', 'terrace', 'exterior', 'orientation', 'floor', 'rooftop', 'elevator', 'pool', 'ac', 'heating', 'year_built', 'neighborhood', 'dist_city_center', 'garage', 'property_type', 'garden', 'dist_closest_station', 'property_age', 'property_age_cat', 'floor_cat', 'neighborhood_rent_index', 'neighborhood_rent_index_cluster', 'rooms_per_100_sqm', 'bathrooms_per_100_sqm'], dtype='object')
```

```
In [556]: non_binary_cat_var = [
```

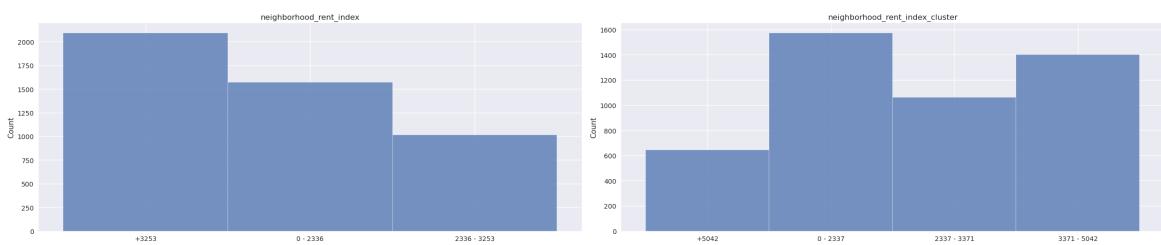
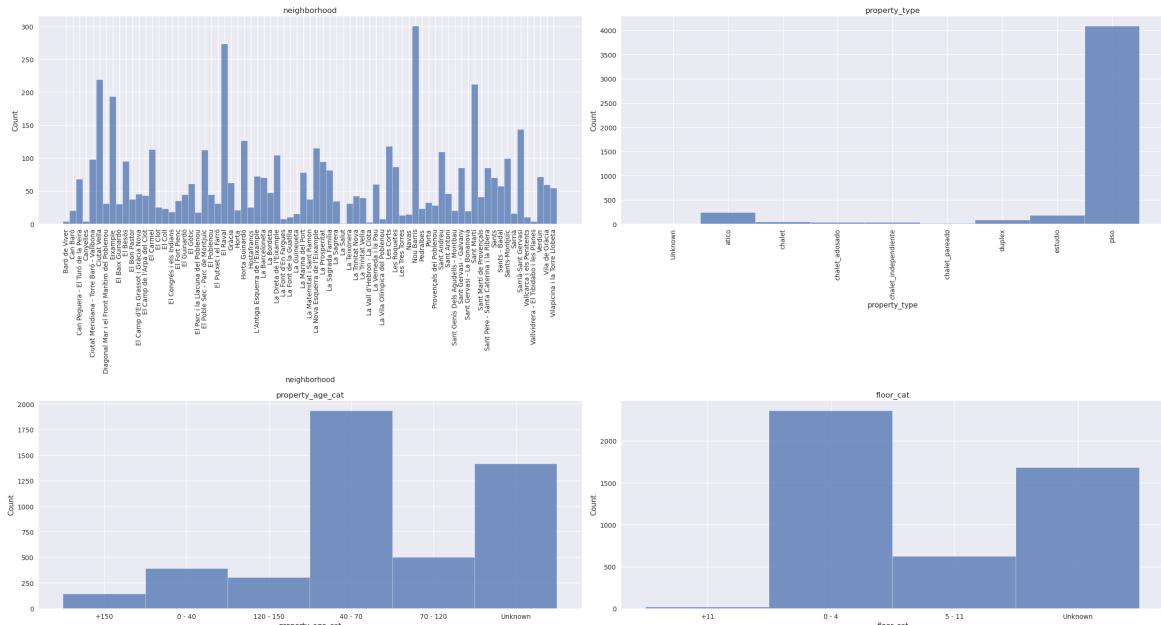
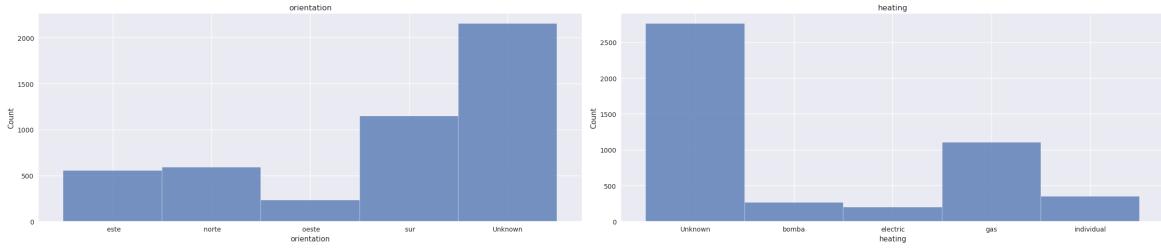
```
'orientation',
'heating',
'neighborhood',
'property_type',
'property_age_cat',
'floor_cat',
'neighborhood_rent_index',
'neighborhood_rent_index_cluster']
```



```
In [558]: fig, axs = plt.subplots(nrows=4, ncols=2, figsize=(25, 30))
axs = axs.flatten()

for i, feature in enumerate(non_binary_cat_var):
    sns.histplot(data=sale_clean_df, x=feature, ax=axs[i], kde=False)
    axs[i].set_title(feature)
    if feature in ['neighborhood', 'property_type']:
        axs[i].tick_params(axis='x', labelrotation=90)

plt.tight_layout()
plt.show()
```



- Observamos que la vivienda más común en BCN son los pisos.
- La mayoría de viviendas tienen entre 40 y 70 años.
- Neighborhood:
  - parece que el 'neighborhood\_rent\_index\_cluster' refleja mejor la realidad que 'neighborhood\_rent\_index', ya que agrupa en 4 rangos en lugar de 3 y parece más equilibrado, ya que, por ejemplo, indica que la mayoría de viviendas tienen un precio por metro cuadrado de hasta 2337 euros y que las que tienen un precio por metro cuadrado de más de 5042 euros son pocas, lo que tiene sentido. Por tanto, nos quedaremos con 'neighborhood\_rent\_index\_cluster'.

## 1.13 Relación entre las variables categóricas y la variable objetivo price

In [559]: `# variables categóricas`  
category\_vars

Out[559]: `['balcony',  
'terrace',  
'exterior',  
'orientation',  
'rooftop',  
'elevator',  
'pool',  
'ac',  
'heating',  
'neighborhood',  
'garage',  
'property_type',  
'garden',  
'property_age_cat',  
'floor_cat',  
'neighborhood_rent_index',  
'neighborhood_rent_index_cluster']`

### Violinplots

```
In [562]: def create_violinplots(df: pd.DataFrame,
                                cat_cols: List[str],
                                target_var: str,
                                plot_width: int = 8,
                                plot_height: int = 6) -> None:
    """
    Creates violin plots for the target variable across categorical variables.

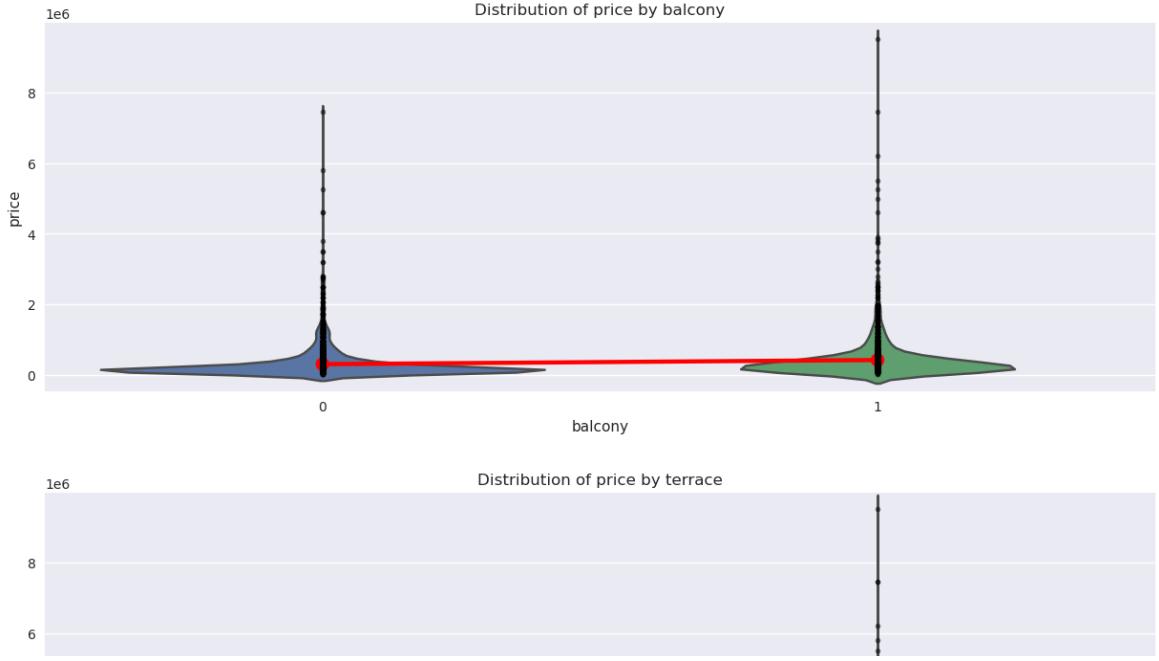
    Args:
        df (pd.DataFrame): The pandas DataFrame containing the data.
        cat_cols (List[str]): A list of the names of the categorical columns.
        target_var (str): The name of the target variable.
        plot_width (int): The width of the plot in inches. Default is 8.
        plot_height (int): The height of the plot in inches. Default is 6.

    Returns:
        None.
    """
    for col in cat_cols:
        # if num_categories is greater than 10, the x-axis label rotates
        num_categories = len(df[col].unique())
        if num_categories > 10:
            x_label_rotation = 90
        else:
            x_label_rotation = 0

        fig, ax = plt.subplots(figsize=(plot_width, plot_height))
        sns.violinplot(x=col, y=target_var, data=df, ax=ax)
        # show individual points
        sns.stripplot(x=col, y=target_var, data=df, jitter=False, color='black')
        # visualize the mean value on the violin plot
        sns.pointplot(x=col, y=target_var, data=df, color='red', ax=ax)
        plt.title(f"Distribution of {target_var} by {col}")
        plt.xlabel(col)
        plt.ylabel(target_var)

        if x_label_rotation != 0:
            ax.set_xticklabels(ax.get_xticklabels(), rotation=x_label_rotation)
        sns.despine()
        plt.show()
```

```
In [563]: create_violinplots(sale_clean_df, category_vars, TARGET_VARIABLE, 15,
```



## Boxplots

```
In [564]: def create_boxplots(df: pd.DataFrame,
                           cat_cols: List[str],
                           target_var: str,
                           plot_width: int = 8,
                           plot_height: int = 6) -> None:
    """
    Creates boxplots for the target variable across categorical variables.

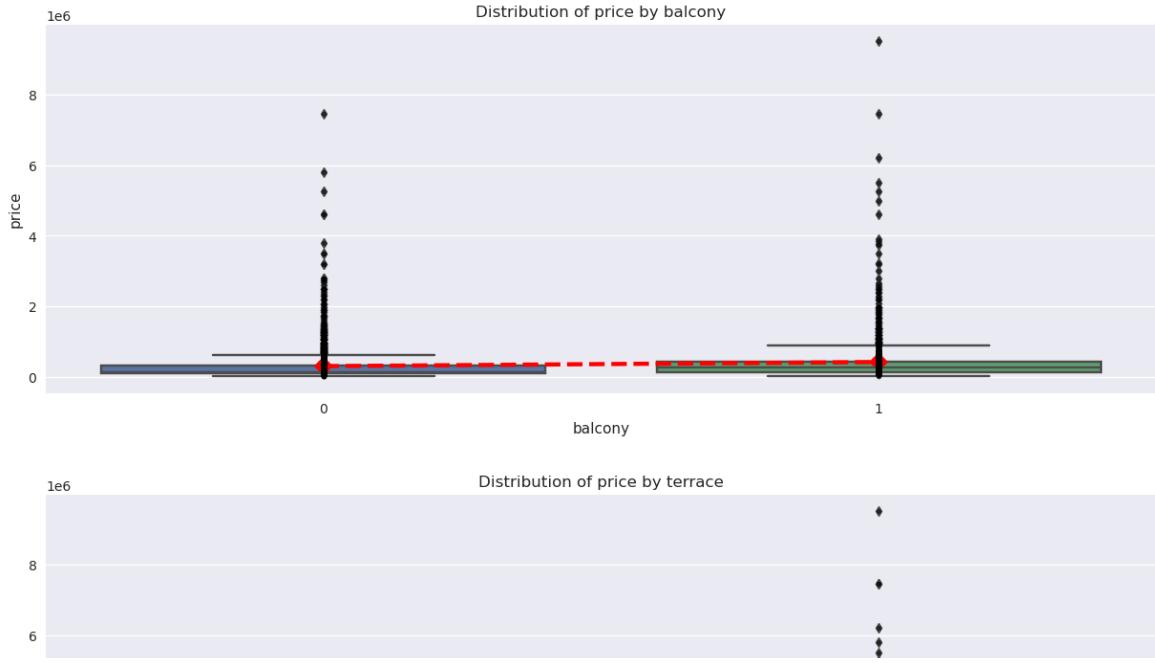
    Args:
        df (pd.DataFrame): The pandas DataFrame containing the data.
        cat_cols (List[str]): A list of the names of the categorical columns.
        target_var (str): The name of the target variable.
        plot_width (int): The width of the plot in inches. Default is 8.
        plot_height (int): The height of the plot in inches. Default is 6.

    Returns:
        None.
    """
    for col in cat_cols:
        # if num_categories is greater than 10, the x-axis label rotation is 90 degrees
        num_categories = len(df[col].unique())
        if num_categories > 10:
            x_label_rotation = 90
        else:
            x_label_rotation = 0

        fig, ax = plt.subplots(figsize=(plot_width, plot_height))
        sns.boxplot(x=col, y=target_var, data=df, ax=ax)
        # show individual points
        sns.stripplot(x=col, y=target_var, data=df, jitter=False, color='black')
        # visualize the mean value on the box plot
        sns.pointplot(x=col, y=target_var, data=df, color='red', ax=ax)
        # visualize the median value on the box plot
        median_val = df.groupby(col)[target_var].median().sort_values()
        #for i, val in enumerate(median_val):
        #    ax.text(i, val, f"{val:.2f}", horizontalalignment='center')
        plt.title(f"Distribution of {target_var} by {col}")
        plt.xlabel(col)
        plt.ylabel(target_var)

        if x_label_rotation != 0:
            ax.set_xticklabels(ax.get_xticklabels(), rotation=x_label_rotation)
        sns.despine()
        plt.show()
```

In [566]: `create_boxplots(sale_clean_df, category_vars, TARGET_VARIABLE, 15, 5)`



## 1.14 Eliminación de variables no útiles

Como hemos visto y justificado anteriormente, hay ciertas variables redundantes que podemos eliminar. De momento eliminaremos las siguientes:

In [599]: `# hacemos una copia  
sales_simplified_df = sale_clean_df.copy(deep=True)`

In [600]: `# variables a eliminar  
not_useful_vars = ['year_built',  
 'sq_meters',  
 'neighborhood_rent_index'  
 #'bathrooms',  
 #'rooms'  
 ]  
# eliminamos not_useful_vars  
sales_simplified_df.drop(not_useful_vars, axis=1, inplace=True)`

In [601]: `sales_simplified_df.shape`

Out[601]: (4677, 28)

## 1.15 PCA

El análisis de componentes principales es una herramienta valiosa para seleccionar las variables más importantes y comprender mejor la estructura de los datos en modelos de machine learning.

Los resultados del análisis de componentes principales (PCA) pueden proporcionar información valiosa sobre la estructura del conjunto de datos y las variables más importantes. Los loadings son una medida de la contribución de cada variable al componente principal, y pueden ser útiles para seleccionar las variables más importantes para los modelos de machine learning.

La selección de variables puede ayudar a reducir la dimensionalidad del conjunto de datos, lo que puede mejorar la eficiencia del modelo y reducir el riesgo de sobreajuste. Al seleccionar las variables más importantes, podemos enfocar nuestros recursos en las variables más relevantes para el modelo.

Además, el PCA puede ayudarnos a entender mejor la relación entre las diferentes variables y cómo se relacionan con los componentes principales. Esto puede ser útil para identificar patrones y tendencias que podrían ser importantes para nuestros modelos de machine learning.

```
In [594]: def perform_pca(df: pd.DataFrame, target_variable: str, n_components: int) -> Tuple[pd.DataFrame, pd.DataFrame]:  
    """  
    Perform principal component analysis (PCA) on a DataFrame with numeric features.  
    The function removes the target variable column from the DataFrame and performs PCA on the remaining columns.  
    Parameters:  
    df : pd.DataFrame  
        DataFrame with numeric features.  
    target_variable : str  
        Name of the target variable column to be removed from the DataFrame.  
    n_components : int, optional  
        Number of principal components to compute. Default is 3.  
  
    Returns:  
    Tuple[pd.DataFrame, pd.DataFrame]  
        Tuple containing the loadings for each component and the transformed data.  
  
    Example:  
    -----  
    # perform PCA on the DataFrame  
    loadings, transformed_data = perform_pca(df, 'target_variable', 2)  
    ....  
  
    # remove the target variable from the DataFrame  
    df = df.drop(columns=target_variable)  
  
    # select only the numeric features and drop na values  
    numeric_features = df.select_dtypes(include=['int64', 'float64']).columns  
    X = numeric_features  
  
    # scale the data  
    scaler = StandardScaler()  
    X_scaled = scaler.fit_transform(X)  
  
    # apply PCA  
    pca = PCA(n_components=n_components)  
    pca.fit(X_scaled)  
  
    # calculate the variance explained by each component  
    variance = pd.DataFrame({'variance': pca.explained_variance_ratio_,  
                            'PC': ['PC{}'.format(i) for i in range(1, n_components + 1)]})  
    variance = variance.sort_values('variance', ascending=False)  
  
    # plot the variance explained by each component  
    sns.barplot(x='PC', y='variance', data=variance, color='blue')  
    plt.title('Variance explained by each principal component')  
    plt.xlabel('Principal Component')  
    plt.ylabel('Percentage of explained variances')  
    plt.show()  
  
    # get the loadings for each component  
    loadings = pd.DataFrame(pca.components_.T, columns=['PC{}'.format(i) for i in range(1, n_components + 1)])
```

```

# plot the loadings for each principal component
n_rows = (n_components + 1) // 2
fig, axs = plt.subplots(n_rows, 2, figsize=(15, 6 * n_rows), sharex=True)

for i in range(n_components):
    row = i // 2
    col = i % 2
    loadings_pc = loadings[f'PC{i+1}'].abs().sort_values(ascending=False)
    sns.barplot(x=loadings_pc.values, y=loadings_pc.index, ax=axs[row, col])
    axs[row, col].set_title(f'Loadings for PC{i+1}')
    axs[row, col].set_xlabel('Loading Value')
    axs[row, col].set_ylabel('Variable')

# adjust the subplots if n_components is odd to remove an empty subplot
if n_components % 2 != 0:
    axs[n_rows-1, 1].remove()

plt.show()

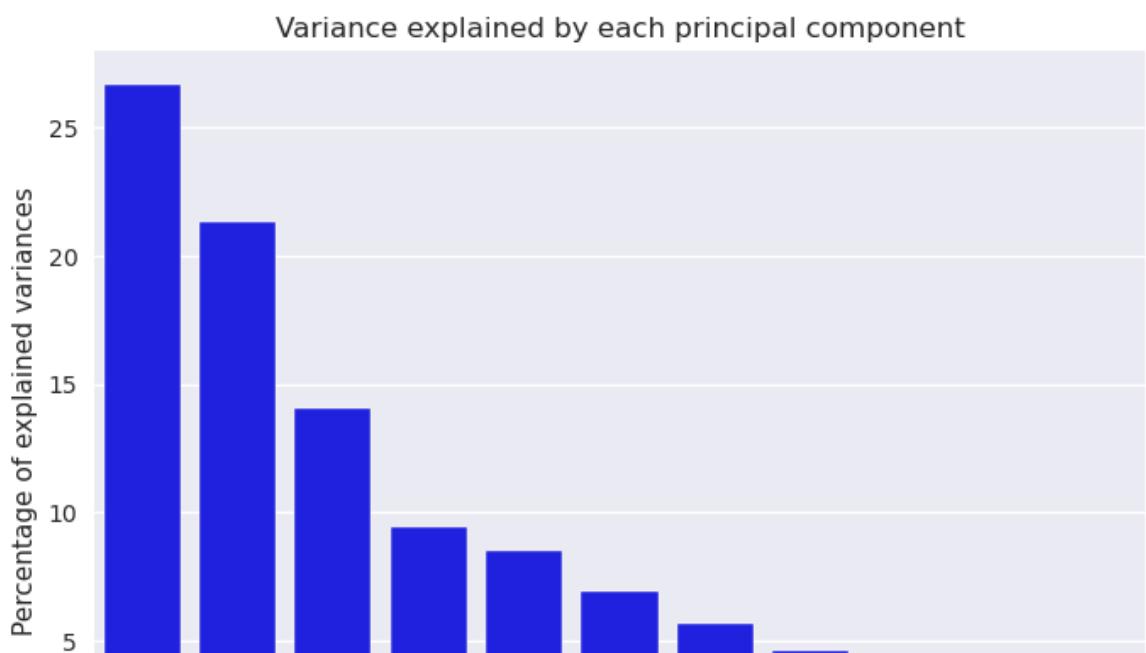
# return the transformed data and the loadings
transformed_data = pd.DataFrame(pca.transform(X_scaled), columns=[f'PC{i+1}' for i in range(n_components)])
return loadings, transformed_data

```

In [602]: # number of numeric variables  
len(sales\_simplified\_df.select\_dtypes(include=['int64', 'float64']).columns)

Out[602]: 12

In [604]: loadings, transdformed\_data = perform\_pca(sales\_simplified\_df, TARGET\_VARIABLE)  
display(loadings)



- Obtenemos un total de 11 componentes (tantas como variables tenemos sin contar las categóricas ni la variable objetivo).
- La 1<sup>a</sup> componente es la que más varianza nos va a explicar: ya lo resume, ya hace la

combinación lineal entre nuestras variables que explique todas esas variables que le hemos metido.

- Observando las varianzas explicadas por cada componente principal (PC), vemos que entre las 3 primeras componentes ya vemos que casi todo queda explicado, las últimas son residuales.
- Miramos la medida de la cantidad de variación en los datos originales que es capturada por cada uno de los componentes principales de la componente 1, 2 y 3:  $27 + 22 + 14 = 63$ . Es decir, la comp1, comp2 y comp3 nos explican el 63% de la variación de los datos. Con un 63%, para empezar a generar un modelo y comenzar a limpiar variables ya nos sirve.
- Loadings son los pesos: nos dice qué variables son las que realmente están aportando mayor información en esta componente. Los pesos son los coeficientes que describen la combinación lineal de las variables originales que componen cada componente principal. (Se miran tanto las positivas como las negativas, las que tengan más peso ya sea en + o en -). Los pesos de los componentes muestran las relaciones lineales entre cada variable original y los componentes. Los valores positivos indican una relación positiva entre la variable original y el componente, mientras que los valores negativos indican una relación negativa.
  - Podemos observar que varias variables tienen pesos elevados en el primer componente (Comp.1), como sq\_meters\_built, bathrooms y longitude (peso negativo). Esto sugiere que estas variables impulsan el primer componente, que podría captar algún aspecto importante de la variación de los datos.
  - El segundo componente (Comp.2) tiene pesos positivos para la variable dist\_city\_center y latitude, y pesos elevados negativos para property\_age. Esto sugiere que este componente podría estar captando alguna relación entre estas variables en los datos.
  - El tercer componente (Comp.3) tiene pesos positivos elevados para bathrooms\_per\_100\_sqm y rooms\_per\_100\_sqm. Esto sugiere que este componente podría estar capturando alguna relación entre estas variables en los datos.

Del análisis de componentes principales (PCA), se puede observar que las variables sq\_meters\_built, bathrooms, longitude, dist\_city\_center, latitude, bathrooms\_per\_100\_sqm y rooms\_per\_100\_sqm tienen los mayores pesos en los primeros 3 componentes principales, lo que indica que estas variables contribuyen significativamente a la variación total de los datos. Sin embargo, se requiere un análisis adicional para determinar si estas variables tienen una relación significativa con el precio de la vivienda y cómo se relacionan con él.

## 1.16 Mutual Information

La información mutua es una medida de la dependencia entre dos variables, y puede utilizarse para identificar qué predictores tienen la relación más estrecha con la variable objetivo.

Los valores más altos de MI indican una mayor dependencia entre dos variables, es decir, una puntuación más alta significa que el atributo es más informativo para predecir la variable objetivo.

```
In [669]: def mutual_info_regression_analysis(df: pd.DataFrame,
                                             target_col: str,
                                             figsize: tuple = (10, 15),
                                             encoder: str = None,
                                             ohe_cols: List[str] = None,
                                             le_cols: List[str] = None) -> pd.
.....
    Performs mutual information analysis between the predictors and target variable.
    It returns a sorted dataframe containing feature names and their mutual information scores.

    Parameters:
        df: A pandas dataframe containing the predictors and target variable.
        target_col: A string indicating the name of the target variable.
        figsize: A tuple indicating the size of the output plot (default: (10, 15)).
        encoder: An optional string indicating which encoder to use, e.g., "ohe" or "le".
        ohe_cols: An optional list of column names to be one-hot encoded.
        le_cols: An optional list of column names to be label encoded.

    Returns:
        A pandas dataframe containing feature names and their mutual information scores.
.....
# drop rows with na
df = df.dropna(axis=1)

# Separate the predictors and target variable
X = df.drop(target_col, axis=1)
y = df[target_col]

# Apply one-hot encoding if specified
if ohe_cols:
    categorical_cols = list(set(ohe_cols) & set(X.columns))
    if categorical_cols:
        ohe = OneHotEncoder()
        ohe.fit(X[categorical_cols])
        encoded_cols = ohe.get_feature_names_out(categorical_cols)
        X_encoded = pd.concat([X.drop(categorical_cols, axis=1),
                               pd.DataFrame(ohe.transform(X[categorical_cols]),
                                             columns=encoded_cols)])
    X = X_encoded
# Apply encoding if specified
elif encoder == "ohe":
    # One-hot encode categorical variables
    categorical_cols = X.select_dtypes(include=['category', 'object'])
    if categorical_cols:
        ohe = OneHotEncoder()
        ohe.fit(X[categorical_cols])
        encoded_cols = ohe.get_feature_names_out(categorical_cols)
        X_encoded = pd.concat([X.drop(categorical_cols, axis=1),
                               pd.DataFrame(ohe.transform(X[categorical_cols]),
                                             columns=encoded_cols)])
    X = X_encoded

# Apply label encoding if specified
if le_cols:
    categorical_cols = list(set(le_cols) & set(X.columns))
```

```
if categorical_cols:
    le = LabelEncoder()
    X[categorical_cols] = X[categorical_cols].apply(lambda col:
elif encoder == "le":
    # Label encode categorical variables
    categorical_cols = X.select_dtypes(include=['category', 'object'])
    if categorical_cols:
        le = LabelEncoder()
        X[categorical_cols] = X[categorical_cols].apply(lambda col:
# Delete categorical vars not specified to encode
cat_vars = X.select_dtypes(include=['category', 'object']).columns
if cat_vars: # if not empty
    # Remove columns with categorical variables
    categorical_cols_in_data = set(cat_vars).intersection(set(X.co
X = X.drop(categorical_cols_in_data, axis=1)

# Determine which predictors are discrete: all discrete features s
discrete_features = X.dtypes == int

# Calculate mutual information
mi_scores = mutual_info_regression(X, y, discrete_features=discret
mi_scores = pd.Series(mi_scores, name="MI Scores", index=X.columns)
mi_scores = mi_scores.sort_values(ascending=False)

# Plot mutual information scores
fig, ax = plt.subplots(figsize=figsize)
mi_scores.sort_values().plot.barh(ax=ax)
ax.set(title="Mutual Information Scores", xlabel="MI Score", ylab
plt.show()

return mi_scores
```

Debemos codificar las variables categóricas para realizar el análisis de información mutua. Para ello utilizaremos one-hot encoding para las variables que no siguen un orden o jerarquía, y ordinal encoder para las que hay un orden o jerarquía, como floor\_cat, property\_age\_cat y neighborhood\_rent\_index\_cluster. Esto garantiza que los valores codificados reflejen el orden inherente de las categorías y puedan utilizarse para el análisis.

Codificaremos la categoría "Unknown" como una etiqueta separada: sustituirla por -1 antes de utilizar OrdinalEncoder. Esta es una práctica común cuando se trata de datos desconocidos o faltantes.

In [680]: sales\_simplified\_df.floor\_cat.value\_counts()

Out[680]:

0 - 4	2359
Unknown	1680
5 - 11	623
+11	15

Name: floor\_cat, dtype: int64

```
In [681]: sales_simplified_df.property_age_cat.value_counts()
```

```
Out[681]: 40 - 70      1933  
Unknown      1415  
70 - 120      500  
0 - 40        389  
120 - 150     300  
+150         140  
Name: property_age_cat, dtype: int64
```

```
In [682]: sales_simplified_df.neighborhood_rent_index_cluster.value_counts()
```

```
Out[682]: 0 - 2337      1571  
3371 - 5042      1403  
2337 - 3371      1060  
+5042          643  
Name: neighborhood_rent_index_cluster, dtype: int64
```

```
In [724]: # variables que queremos codificar usando ordinal encoder
oe_cols = oe_cols = ['floor_cat', # unknown
                     'property_age_cat', # unknown
                     'neighborhood_rent_index_cluster']
# creamos nuevo dataset para no modificar el original
oe_data = sales_simplified_df.copy(deep=True)

# Replace 'Unknown' values with -1
#oe_data[oe_cols] = oe_data[oe_cols].replace('Unknown', -1)

# Define the order of the categories for each column
floor_cat_order = ['Unknown', '0 - 4', '5 - 11', '+11']
property_age_cat_order = ['Unknown', '0 - 40', '40 - 70', '70 - 120',
                           neighborhood_rent_index_cluster_order = ['Unknown', '0 - 2337', '2337']

# Create an instance of the ordinal encoder
encoder = OrdinalEncoder(categories=[floor_cat_order, property_age_cat_order])
# Fit and transform the selected columns
# oe_data[new_col] = encoder.fit_transform(oe_data[[col]])

for col in oe_cols:
    # Create a new column name for the encoded data
    new_col = col + '_encoded'
    # Create an instance of the ordinal encoder
    encoder = OrdinalEncoder(categories=[eval(col+'_order')])
    # Fit and transform the selected columns
    oe_data[new_col] = encoder.fit_transform(oe_data[[col]])
    # replace 0 by -1 ('Unknown' by -1)
    oe_data[new_col] = oe_data[new_col].replace(0, -1)

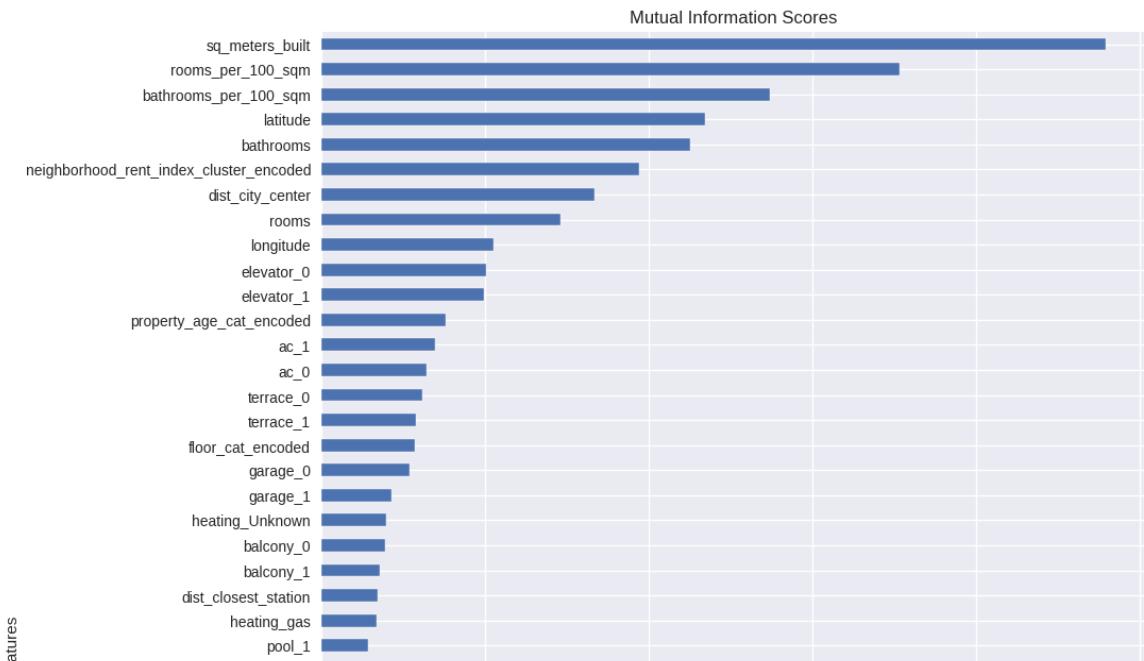
# Print the encoded data
display(oe_data[['floor_cat','floor_cat_encoded','property_age_cat','property_age_cat_encoded','neighborhood_ren']])
```

	floor_cat	floor_cat_encoded	property_age_cat	property_age_cat_encoded	neighborhood_ren
0	0 - 4	1.0	0 - 40	-1.0	1.0
1	Unknown	-1.0	40 - 70	2.0	
2	0 - 4	1.0	Unknown	-1.0	
3	Unknown	-1.0	40 - 70	2.0	
4	5 - 11	2.0	70 - 120	3.0	

- Ya tenemos las variables codificadas usando ordinal encoding.

```
In [726]: # variables que queremos codificar usando OHE
ohe_cols = get_binary_cols(sales_simplified_df) + ['orientation', 'heatin
# ['balcony',
# 'terrace',
# 'exterior',
# 'rooftop',
# 'elevator',
# 'pool',
# 'ac',
# 'garage',
# 'garden']
```

```
In [725]: # calculamos la información mutua
mutual_info_regression_analysis(oe_data, TARGET_VARIABLE, ohe_cols=ohe
```



Recordemos que una puntuación más alta significa que la característica es más informativa para predecir la variable objetivo.

Resultados:

- parece que "sq\_meters\_built" tiene la puntuación de información mutua más alta, lo que significa que puede ser el atributo más informativo para predecir el precio de la vivienda. Del mismo modo, "rooms\_per\_100\_sqm" y "bathrooms\_per\_100\_sqm" también tienen puntuaciones de información mutua relativamente altas, lo que indica que también son atributos informativos.
- Otros atributos que tienen puntuaciones de información mutua relativamente altas son "latitude", "bathrooms" y "neighborhood\_rent\_index\_cluster\_encoded", mientras que atributos como "property\_type\_duplex", "property\_type\_chalet\_adosado" y "heating\_electric" tienen puntuaciones muy bajas, lo que indica que son menos informativos a la hora predecir el precio de la vivienda.

No obstante, debemos tener en cuenta que la información mutua es sólo una métrica para

evaluar la relevancia de los atributos, y no es necesariamente la única ni la mejor métrica para ello. Por eso, probaremos otros métodos de selección de características y compararemos los resultados.

## 1.17 ANOVA

- ANOVA (Analysis of Variance) es una técnica estadística que se utiliza para dos o más grupos para comprobar si existe una diferencia entre sus valores medios.
- La prueba ANOVA nos dice cómo cuantificar la pertenencia a una de las categorías: mide la variación entre grupos comparada con la variación dentro de los grupos.
- La hipótesis nula es que las medias de todos los grupos son iguales.
- La prueba ANOVA calcula el estadístico F y el p-value.
  - Si el p-value es inferior a un nivel de significación (normalmente 0.05), se rechaza la hipótesis nula, lo que indica que existe una diferencia significativa entre las medias de al menos dos grupos.

```
In [744]: def perform_anova_kruskall(df: pd.DataFrame, target_col: str, normal_data: bool):
    """
    Performs ANOVA or Kruskal-Wallis analysis for each categorical variable.

    Parameters:
        df: A pandas dataframe containing the predictors and target variable.
        target_col: A string indicating the name of the target variable.
        normal_data: A boolean indicating whether the data is normally distributed.

    Returns:
        A pandas dataframe containing ANOVA or Kruskal-Wallis results.
    """

    # Select categorical columns
    cat_cols = df.select_dtypes(include=['category', 'object']).columns

    # Perform ANOVA or Kruskal-Wallis for each categorical variable
    result_list = []
    for col in cat_cols:
        if normal_data:
            model = ols(f'{target_col} ~ {col}', data=df).fit()
            anova_table = sm.stats.anova_lm(model, typ=2)
            result_list.append(anova_table)
        else:
            groups = [group[target_col].values for name, group in df.groupby(col)]
            groups = [g for g in groups if len(g) > 0] # remove empty groups
            if len(groups) > 1:
                stat, p = stats.kruskal(*groups)
                table = pd.DataFrame({'Group': [col], 'Statistic': [stat]})
                result_list.append(table)

    # Combine the results into a single dataframe
    result_df = pd.concat(result_list, axis=0)

    # Filter out the residual rows
    result_df = result_df[result_df.index != "Residual"]

    return result_df
```

In [745]: `perform_anova_kruskall(sales_simplified_df, TARGET_VARIABLE, normal_da`

Out[745]:

		sum_sq	df	F	PR(>F)
	<b>balcony</b>	1.492191e+13	1.0	61.289159	6.052713e-15
	<b>terrace</b>	1.586301e+14	1.0	745.695503	1.811869e-152
	<b>exterior</b>	1.218611e+11	1.0	0.494098	4.821397e-01
	<b>orientation</b>	1.208281e+13	4.0	12.368195	5.245114e-10
	<b>rooftop</b>	1.322870e+13	1.0	54.253887	2.071564e-13
	<b>elevator</b>	3.951209e+13	1.0	165.872606	2.526426e-37
	<b>pool</b>	2.309010e+14	1.0	1170.490286	3.858810e-229
	<b>ac</b>	9.147716e+13	1.0	402.819978	5.129500e-86
	<b>heating</b>	5.498184e+13	4.0	58.479069	3.069629e-48
	<b>neighborhood</b>	4.209760e+14	74.0	35.757621	0.000000e+00
	<b>garage</b>	2.376710e+14	1.0	1213.718826	1.260328e-236
	<b>property_type</b>	2.904986e+14	8.0	196.498278	2.116041e-287
	<b>garden</b>	2.758690e+14	1.0	1470.126584	6.324437e-280
	<b>property_age_cat</b>	1.013818e+14	5.0	90.050776	1.015491e-90
	<b>floor_cat</b>	7.406342e+12	3.0	10.069263	1.306050e-06
	<b>neighborhood_rent_index_cluster</b>	3.254277e+14	3.0	612.426281	0.000000e+00

Explicación de las columnas del resultado de ANOVA:

- La columna "sum\_sq" representa la suma de cuadrados, que es una medida de la variación entre grupos.
- La columna "df" representa los grados de libertad, que es el número de categorías menos uno.
- La columna "F" representa el estadístico F, que es la relación entre el cuadrado medio entre grupos y el cuadrado medio dentro de los grupos.
- La columna "PR(>F)" representa el p-value, que indica la probabilidad de obtener un estadístico F tan extremo como el observado, suponiendo que la hipótesis nula sea cierta.

Resultados:

- La variable "exterior" tiene un p-value de 0.48, que es superior al nivel de significación típico de 0.05. Esto significa que no hay una diferencia significativa en el precio de venta entre los inmuebles con o sin una orientación exterior o interior en el edificio. Por tanto, no existe una diferencia significativa en el precio de venta entre los inmuebles con o sin una orientación exterior o interior en el edificio.
- Las demás variables tienen un efecto significativo sobre la variable dependiente "price", ya que los p-values son todos menores que el nivel de significación típico de 0.05. Esto significa que cada una de las variables categóricas (excepto "exterior") está

significativamente relacionada con el precio de venta de un inmueble.

- En particular, podemos ver que las variables neighborhood\_rent\_index\_cluster, property\_type, garage, pool, terrace y garden tienen los valores F más altos y los p-values más bajos, lo que indica que tienen un fuerte efecto sobre el precio de venta de un inmueble.

Variables con mayor efecto sobre el precio de venta de la vivienda:

- neighborhood\_rent\_index\_cluster
- garden
- property\_type
- pool
- garage
- terrace

No obstante, debemos comprobar las condiciones para poder aplicar el test ANOVA.

Las suposiciones de ANOVA son:

- La distribución de los datos es normal
- La varianza es constante entre los grupos
- Las observaciones son independientes entre sí

### Normality Tests:

- Jarque–Bera test
  - funciona bien en muestras grandes (normalmente superiores a 2000 observaciones).
- Quantile-Quantile Plot

```
In [738]: # Data preparation  
price = sales_simplified_df[target].values  
print(len(price))
```

4677

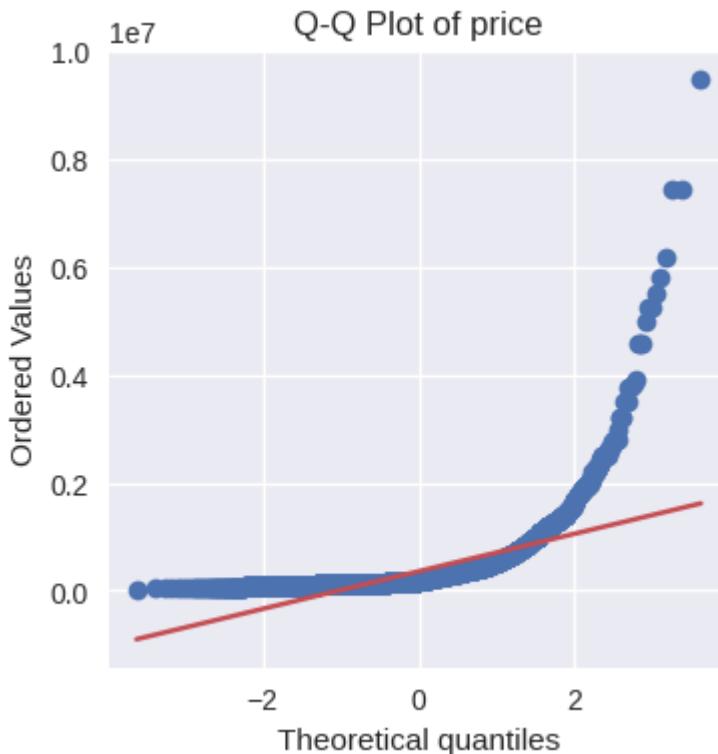
```
In [739]: # Jarque-Bera test for Normality  
# Perform Jarque-Bera test  
print(stats.jarque_bera(price))  
statistic, pvalue = jarque_bera(price)  
check_normality(pvalue)
```

```
Jarque_beraResult(statistic=900023.8239598732, pvalue=0.0)  
Since p-value ≤ 0.05, we reject the null hypothesis i.e. we assume th  
e distribution of our variable is not normal/gaussian.
```

```
In [743]: # figure with two subplots
fig, (ax) = plt.subplots(ncols=1, figsize=(4, 4))

# first Q-Q plot on the left subplot
stats.probplot(price, dist='norm', plot=ax)
ax.set_title('Q-Q Plot of price')

plt.show()
```



Realmente, no se cumple el supuesto de normalidad en la variable objetivo 'price' así que tendríamos que usar una prueba no paramétrica como Kruskal-Wallis (prueba no paramétrica que no requiere el supuesto de normalidad).

Aplicamos la prueba de Kruskal-Wallis a las distintas variables categóricas del conjunto de datos para determinar si tienen una relación estadísticamente significativa con la variable objetivo, que es el precio de la vivienda.

In [746]: # Kruskall-Wallis  
 perform\_anova\_kruskall(sales\_simplified\_df, TARGET\_VARIABLE, normal\_da

Out[746]:

Group		Statistic	p-value
0	balcony	284.056992	9.806851e-64
0	terrace	723.267358	2.607240e-159
0	exterior	202.605342	5.640349e-46
0	orientation	95.873648	7.429123e-20
0	rooftop	65.136135	6.989852e-16
0	elevator	1258.579108	1.134161e-275
0	pool	267.584833	3.813229e-60
0	ac	816.517434	1.383309e-179
0	heating	582.897101	7.790265e-125
0	neighborhood	2666.000979	0.000000e+00
0	garage	463.219475	9.578186e-103
0	property_type	433.876370	1.051525e-88
0	garden	126.820576	2.033516e-29
0	property_age_cat	536.810249	9.018004e-114
0	floor_cat	200.328966	3.581637e-43
0	neighborhood_rent_index_cluster	2131.195854	0.000000e+00

- Podemos ver que todas las variables tienen p-values muy bajos, siendo el más bajo 0.0. Esto significa que podemos rechazar la hipótesis nula de que no hay diferencias entre los grupos y concluir que existe una relación estadísticamente significativa entre cada variable categórica y la variable objetivo, el precio.
- Por lo tanto, podemos concluir que cada una de las variables categóricas analizadas tiene un impacto significativo en el precio de la vivienda. Por ejemplo, la variable neighborhood\_rent\_index\_cluster tiene el p-value más bajo, lo que indica que está fuertemente asociada con el precio del inmueble. Variables como terrace, heating, elevator, ac, property\_age\_cat y garage también tienen p-values muy bajos, lo que indica una relación significativa con el precio.

Variables con mayor efecto sobre el precio de venta de la vivienda:

- neighborhood\_rent\_index\_cluster
- elevator
- ac
- heating
- property\_age\_cat
- garage
- terrace

## 1.18 Feature Importance: ExtraTreesRegressor method

El algoritmo ExtraTreesRegressor es un método popular para la selección de atributos en ML.

La idea es que el ExtraTreesRegressor utiliza un subconjunto aleatorio de atributos y un subconjunto aleatorio de data points para construir cada árbol de decisión, lo que conduce a un conjunto diverso de árboles con baja correlación. Esto hace que el modelo sea menos propenso al sobreajuste (overfitting) y proporciona una estimación más precisa de la importancia de los atributos.

Mediante el análisis de las puntuaciones de importancia de las características obtenidas, podemos identificar los atributos más importantes del dataset. Estos atributos pueden utilizarse para construir un modelo más simple e interpretable o para reducir la dimensionalidad de los datos para tareas posteriores.

ExtraTreesRegressor es una herramienta útil para problemas de regresión porque puede manejar características numéricas y categóricas, puede detectar relaciones no lineales y es relativamente fácil de implementar y ajustar.

```
In [765]: def perform_extra_trees_regressor(df: pd.DataFrame,
                                         target_col: str,
                                         figsize: tuple = (10, 15),
                                         encoder: str = None,
                                         ohe_cols: List[str] = None,
                                         le_cols: List[str] = None,
                                         n_estimators: int = 100,
                                         criterion: str ='squared_error',
                                         max_features: float = 1.0,
                                         k: int = 10
                                         ) -> pd.DataFrame:
    """
    Trains an ExtraTreesRegressor model on the given dataframe and ret

    Parameters:
    -----
    df : pd.DataFrame
        The input dataframe with predictor and target variables.
    target_col : str
        The name of the target variable.
    figsize : tuple, optional
        The size of the matplotlib figure to display the feature impor
    encoder : str, optional
        The type of encoder to use on the categorical variables. Can b
    ohe_cols : list, optional
        The names of the columns to apply OneHotEncoder to. If not spe
    le_cols : list, optional
        The names of the columns to apply LabelEncoder to. If not spec
    n_estimators : int, optional
        The number of trees in the forest (default is 100).
    criterion : str, optional
        The function to measure the quality of a split. Can be either
    max_features : float, optional
        The maximum number of features each tree is allowed to use (de
    k : int, optional
        The number of top features to return (default is 10).

    Returns:
    -----
    pd.DataFrame
        A dataframe with the k most important features and their corre
    """
    # drop rows with na
    df = df.dropna(axis=1)

    # Separate the predictors and target variable
    X = df.drop(target_col, axis=1)
    y = df[target_col]

    # Apply one-hot encoding if specified
    if ohe_cols:
        categorical_cols = list(set(ohe_cols) & set(X.columns))
        if categorical_cols:
            ohe = OneHotEncoder()
```

```
        ohe.fit(X[categorical_cols])
        encoded_cols = ohe.get_feature_names_out(categorical_cols)
        X_encoded = pd.concat([X.drop(categorical_cols, axis=1),
                               pd.DataFrame(ohe.transform(X[categorical_cols]),
                                             columns=encoded_cols)])
        X = X_encoded
    # Apply encoding if specified
    elif encoder == "ohe":
        # One-hot encode categorical variables
        categorical_cols = X.select_dtypes(include=['category', 'object'])
        if categorical_cols:
            ohe = OneHotEncoder()
            ohe.fit(X[categorical_cols])
            encoded_cols = ohe.get_feature_names_out(categorical_cols)
            X_encoded = pd.concat([X.drop(categorical_cols, axis=1),
                                   pd.DataFrame(ohe.transform(X[categorical_cols]),
                                                 columns=encoded_cols)])
            X = X_encoded

    # Apply label encoding if specified
    if le_cols:
        categorical_cols = list(set(le_cols) & set(X.columns))
        if categorical_cols:
            le = LabelEncoder()
            X[categorical_cols] = X[categorical_cols].apply(lambda col:
                le.fit_transform(col))
    elif encoder == "le":
        # Label encode categorical variables
        categorical_cols = X.select_dtypes(include=['category', 'object'])
        if categorical_cols:
            le = LabelEncoder()
            X[categorical_cols] = X[categorical_cols].apply(lambda col:
                le.fit_transform(col))

    # Delete categorical vars not specified to encode
    cat_vars = X.select_dtypes(include=['category', 'object']).columns
    if cat_vars: # if not empty
        # Remove columns with categorical variables
        categorical_cols_in_data = set(cat_vars).intersection(set(X.columns))
        X = X.drop(categorical_cols_in_data, axis=1)

    model = ExtraTreesRegressor(n_estimators = n_estimators, criterion=criterion)
    model.fit(X,y)
    # use inbuilt class feature importances of tree based classifiers
    #print(model.feature_importances_)

    # perform feature important using the extra_tree_forest.feature_impor
    # plot graph of feature importances for better visualization

    k = k # k most important features
    feat_importances = pd.Series(model.feature_importances_, index=X.columns)
    feat_importances.nlargest(k).plot(kind='barh')
    plt.show()

    return feat_importances.nlargest(k).sort_values(ascending=False)
```

```
In [761]: oe_data.columns
```

```
Out[761]: Index(['price', 'latitude', 'longitude', 'sq_meters_built', 'rooms',
       'bathrooms', 'balcony', 'terrace', 'exterior', 'orientation',
       'floor',
       'rooftop', 'elevator', 'pool', 'ac', 'heating', 'neighborhood',
       'dist_city_center', 'garage', 'property_type', 'garden',
       'dist_closest_station', 'property_age', 'property_age_cat', 'f
loor_cat',
       'neighborhood_rent_index_cluster', 'rooms_per_100_sqm',
       'bathrooms_per_100_sqm', 'floor_cat_encoded',
       'property_age_cat_encoded', 'neighborhood_rent_index_cluster_e
ncoded'],
       dtype='object')
```

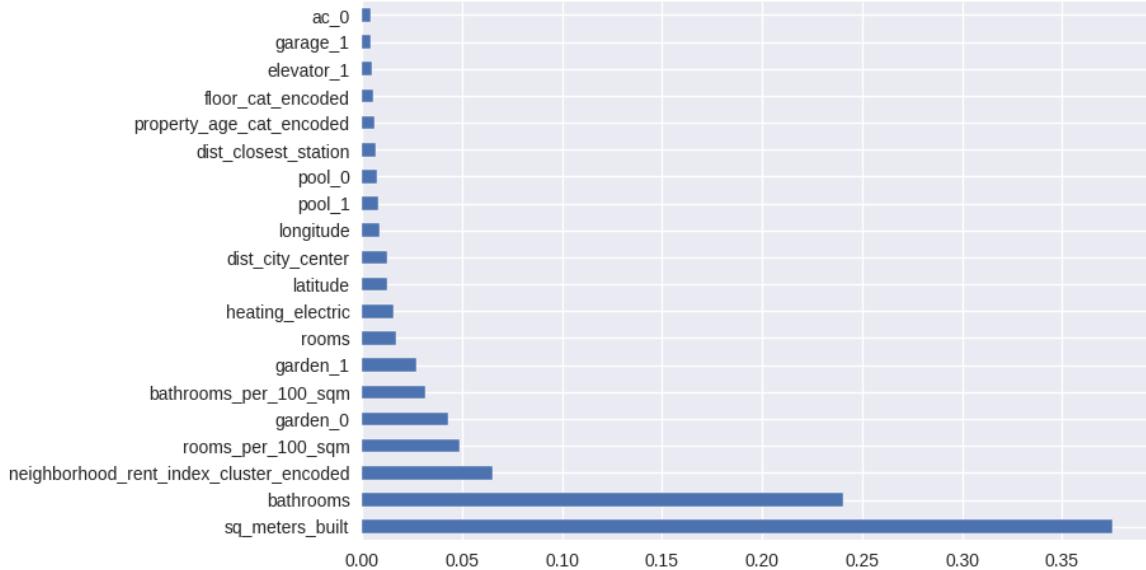
```
In [759]: # variables que queremos codificar usando OHE
```

```
ohe_cols = get_binary_cols(sales_simplified_df) + ['orientation', 'hea
# ['balcony',
#  'terrace',
#  'exterior',
#  'rooftop',
#  'elevator',
#  'pool',
#  'ac',
#  'garage',
#  'garden']
ohe_cols
```

```
Out[759]: ['balcony',
       'terrace',
       'exterior',
       'rooftop',
       'elevator',
       'pool',
       'ac',
       'garage',
       'garden',
       'orientation',
       'heating',
       'property_type']
```

In [768]: # aplicamos ExtraTreesRegressor

```
perform_extra_trees_regressor(df = oe_data,
                               target_col= TARGET_VARIABLE,
                               figsize = (10, 15),
                               #encoder,
                               ohe_cols = ohe_cols,
                               #le_cols,
                               n_estimators = 100,
                               criterion = 'squared_error',
                               max_features = 1.0,
                               k = 20)
```



Out[768]:

Atributo	Importancia
sq_meters_built	0.374912
bathrooms	0.240639
neighborhood_rent_index_cluster_encoded	0.065236
rooms_per_100_sqm	0.048994
garden_0	0.043345
bathrooms_per_100_sqm	0.031479
garden_1	0.027108
rooms	0.017001
heating_electric	0.015657
latitude	0.012793
dist_city_center	0.012373
longitude	0.008754
pool_1	0.008303
pool_0	0.007456
dist_closest_station	0.007154
property_age_cat_encoded	0.006438
floor_cat_encoded	0.005612
elevator_1	0.004900
garage_1	0.004102
ac_0	0.004029

dtype: float64

Las puntuaciones de importancia de los atributos que obtiene del modelo nos indican qué atributos tienen un mayor impacto en el precio predicho:

- Observamos que según el modelo, sq\_meters\_built es el atributo más importante para predecir el precio de la vivienda, seguida de bathrooms y neighborhood\_rent\_index\_cluster\_encoded.
- Otras características como rooms\_per\_100\_sqm, garden y bathrooms\_per\_100\_sqm también tienen puntuaciones de importancia relativamente altas.
- Esta información nos puede ser útil para la selección de atributos en fases posteriores del proceso de modelado.

## 1.19 Feature selection y reducción de la dimensionalidad

En el apartado "Eliminación de variables no útiles" ya eliminamos algunas variables que no eran útiles para realizar predicciones, como 'sq\_meters', 'year\_built' y 'neighborhood\_rent\_index':

- Nos quedamos con 'sq\_meters\_built' y eliminamos 'sq\_meters'.
- Creamos la variables categórica 'property\_age\_cat' (sin valores nulos) a partir de 'year\_built' y 'property\_age'.
- A partir de 'neighborhood' creamos 'neighborhood\_rent\_index' y 'neighborhood\_rent\_index\_cluster', donde al final decidimos eliminar 'neighborhood\_rent\_index' y quedarnos con 'neighborhood\_rent\_index\_cluster'.

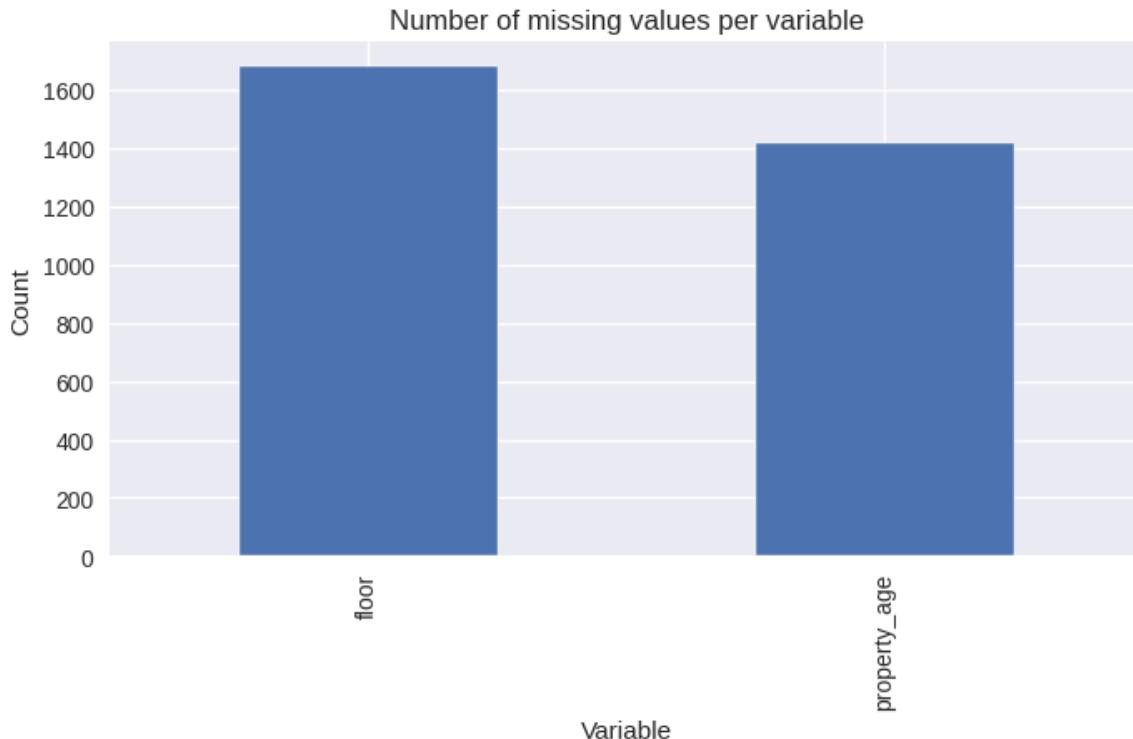
```
In [776]: # dimensión sales_simplified_df  
sales_simplified_df.shape
```

```
Out[776]: (4677, 28)
```

```
In [774]: # missing values  
count_null_values(sales_simplified_df)
```

Variables con valores nulos:

- 'floor': 1680
- 'property\_age': 1415



- Nos quedan los valores nulos de las variables floor y property\_age, pero hemos creado a partir de estas variables otras variables categóricas (floor\_cat y property\_age\_cat) sin missing values. Por tanto, podemos eliminarlas.

```
In [777]: # eliminamos floor y property_age  
cols_missing_val = ['floor', 'property_age']  
sales_simplified_df.drop(cols_missing_val, axis=1, inplace=True)  
sales_simplified_df.shape
```

Out[777]: (4677, 26)

```
In [778]: # missing values  
count_null_values(sales_simplified_df)
```

No se encontraron valores nulos en el dataset.

- Hemos eliminado los missing values de nuestro dataset.

In [780]: `explore_data(sales_simplified_df)`

The dataset includes 4677 instances (rows) and 26 variables (columns).

	price	latitude	longitude	sq_meters_built	rooms	bathrooms	balcony	terrace	exterie
0	2290000	41.409071	2.099850		532	5	6	1	1
1	90000	41.434545	2.171110		50	1	1	0	0
2	145000	41.444518	2.175309		53	3	1	1	0
3	675000	41.392209	2.153368		120	4	2	0	1
4	319000	41.413385	2.162246		69	3	1	0	0

Para dar por cerrado este apartado, nos hemos dado cuenta que para `property_type` sólo había 2 valores nulos en el dataset inicial, antes de dividirlo en train y test data. Como hemos visto, estos dos valores nulos se encuentran en el dataset de train, por lo que en el de test no habrá ninguno. Esto nos puede llevar problemas, ya que por ejemplo, a la hora de codificar la variable `property_type`, si usamos por ejemplo OHE, en el dataset de train habrá una columna más (`property_type_Unknown`) que en el de test.

Debido a esto y a que eliminar 2 filas de el dataset de train (que tiene 4677 instancias) no supone un problema de pérdida de información. Por tanto, eliminaremos estas 2 filas dónde se encuentran los valores nulos.

In [1040]: `sales_simplified_df.property_type.value_counts()`

Out [1040]:

piso	4086
atico	235
estudio	177
duplex	75
chalet	36
chalet_adosado	28
chalet_independiente	26
chalet_pareado	12
Unknown	2

Name: `property_type`, dtype: `int64`

```
In [1041]: # eliminamos las filas donde property_type es Unknown:  
sales_simplified_df.drop(sales_simplified_df[sales_simplified_df['prop  
sales_simplified_df.property_type.value_counts()
```

```
Out[1041]: piso          4086  
atico          235  
estudio         177  
duplex           75  
chalet            36  
chalet_adosado    28  
chalet_independiente 26  
chalet_pareado     12  
Unknown             0  
Name: property_type, dtype: int64
```

```
In [1045]: sales_simplified_df.shape
```

```
Out[1045]: (4675, 26)
```

```
In [1057]: # guardamos en csv  
sales_simplified_df.to_csv('train_data_preprocessed.csv', index=False)
```

```
In [427]: # cargamos el dataset de train  
sales_simplified_df = pd.read_csv('train_data_preprocessed.csv', delim  
# convertimos object a category  
sales_simplified_df = convert_object_to_category(sales_simplified_df)
```

## 1.20 Test dataset pre-processing

Es importante preprocesar el conjunto de datos de test del mismo modo que el conjunto de datos de entrenamiento. Esto garantiza que los datos de prueba se representen en el mismo formato y escala que los datos de entrenamiento, lo cual es necesario para obtener predicciones precisas.

Procesamos el conjunto de datos de prueba siguiendo los mismos pasos que utilizamos para preprocesar el conjunto de datos de entrenamiento.

```
In [240]: class DataPreprocessor:
    target=TARGET_VARIABLE
    not_useful_vars = ['id',
                        'doorman',
                        'quality',
                        'city',
                        'currency',
                        'furniture',
                        'closest_station',
                        'created_at',
                        'last_seen']
    ]
    def test_data_preprocessor(self, file_path):
        self.load_data(file_path)
        self.initial_feature_selection()
        self.replace_binary_cols_nulls_by_zero()
        self.convert_binary_to_category()
        self.property_age_cat()
        self.missing_values()
        self.floor_cat()
        self.dim_reduction_neighborhood()
        self.convert_object_to_category()
        self.bathrooms_roms_per_100_sqm()
        self.feature_selection_2()

    def load_data(self, file_path):
        self.df = pd.read_csv(file_path, delimiter=',').copy()

    def initial_feature_selection(self):
        self.df.drop(self.not_useful_vars, axis=1, inplace=True)

    def get_binary_cols(self) -> list[str]:
        """
        Returns a list with the names of the binary variables in a Pandas DataFrame.

        Parameters:
        None

        Returns:
        A list with the names of the binary variables in the DataFrame.
        """
        binary_cols = [col for col in self.df if (len(self.df[col].value_counts()) == 2) & (self.df[col].dtypes == 'category')]

        return binary_cols

    def replace_binary_cols_nulls_by_zero(self) -> None:
        """
        Replaces missing values in binary columns of a pandas DataFrame.

        Parameters:
        None
        """
```

```
Returns:  
    None  
....  
binary_cols = self.get_binary_cols()  
if binary_cols:  
    self.df.loc[:, binary_cols] = self.df.loc[:, binary_cols].  
  
def convert_binary_to_category(self) -> None:  
    # primero las convertimos a int64 para eliminar los decimales:  
    self.df[get_binary_cols(self.df)] = self.df[get_binary_cols(self.  
    # luego las convertimos a 'category': '0' y '1'  
    self.df[get_binary_cols(self.df)] = self.df[get_binary_cols(self.  
  
def property_age_conditions(self, x: float) -> str:  
....  
    Assigns a property age range label to a property age value.  
  
Parameters:  
    x (float): The property age value.  
  
Returns:  
    str: The label for the property age range that the value f  
....  
if pd.isna(x): # check for NaN values  
    return "Unknown"  
elif x < 0: # viviendas dónde no tengo valores (missing values  
    return "Unknown"  
elif 0 <= x <= 40: # entre 0 y 40 años  
    return "0 - 40"  
elif 40 < x <= 70: # entre 41 y 70 años  
    return "40 - 70"  
elif 70 < x <= 120: # entre 71 y 120 años  
    return "70 - 120"  
elif 120 < x <= 150: # entre 121 y 150 años  
    return "120 - 150"  
elif x > 150: # más de 150 años  
    return "+150"  
else:  
    return "Unknown"  
  
def property_age_cat(self) -> None:  
    # Creamos la edad (años) de la vivienda:  
    self.df['property_age'] = 2023 - self.df['year_built']  
    # aplicamos la función a la columna 'property_age' del DataFra  
    self.df['property_age_cat'] = self.df['property_age'].apply(pr  
  
def missing_values(self) -> None:  
    # variables categóricas con missing values que queremos conver  
    cat_features_na_to_unknown = ['property_type',  
                                  'orientation',  
                                  'heating'  
                                 ]  
    # Fill missing values with 'Unknown' for all columns  
    for col in cat_features_na_to_unknown:  
        if col == 'heating':
```

```
        self.df[col].fillna('Unknown', inplace=True)
        self.df[col].replace('na', 'Unknown', inplace=True)
    else:
        self.df[col].fillna('Unknown', inplace=True)

def conditions_floor(self, x: float) -> str:
    """
    Categorize floors of a property

    Parameters:
    x (Union[float, int]): Number of floors of a property

    Returns:
    str: Categorized floor of the property
    """
    if pd.isna(x): # check for NaN values
        return "Unknown"
    elif x < 0: # viviendas dónde no tengo valores (missing values
        return "Unknown"
    elif ((x >= 0) & (x <= 4)): # viviendas entre 0 y 4 pisos
        return "0 - 4"
    elif ((x >= 5) & (x <= 11)): # viviendas entre 5 y 11 pisos
        return "5 - 11"
    elif ((x > 11)): # viviendas con más de 11 pisos
        return "+11"
    else:
        return "Unknown"

def floor_cat(self) -> None:
    # aplicamos la función a la columna 'floor' del DataFrame
    self.df['floor_cat'] = self.df['floor'].apply(conditions_floo

# asignamos tiers
def assign_price_tier_cluster(self, x: float) -> str:
    if x < 0:
        return "Unknown"
    elif ((x >= 0) & (x < 2337)): # bajo
        return "0 - 2337" # cluster 2

    elif ((x >= 2337) & (x < 3371)): # medio - bajo
        return "2337 - 3371" # cluster 1

    elif ((x >= 3371) & (x < 5042)): # medio - alto
        return "3371 - 5042" # cluster 6 y 5

    elif ((x >= 5042)): # alto
        return "+5042" # cluster 4 y 3
    else:
        return "Unknown"

def dim_reduction_neighborhood(self) -> None:
    neighborhood_df = self.df[['neighborhood', 'price', 'sq_meters'
        # Number of instances per category
        category_count=('price', 'count'),
        # Mean price
```

```
mean_price=('price', "mean"),
# Median price
mean_sq_meters_built=('sq_meters_built', "mean"),
# Median price
median_price=('price', "median"),
# Median price
median_sq_meters_built=('sq_meters_built', "median"))

neighborhood_df['mean_price_per_sq_meters_built'] = neighborhood_df['mean_price']
neighborhood_df['median_price_per_sq_meters_built'] = neighborhood_df['median_price']

neighborhood_df['neighborhood_rent_index_cluster'] = neighborhood_df['neighborhood'].map(neighborhood_mapping)

# agregamos esta información el modelo de datos:
neighborhood_mapping = neighborhood_df[['neighborhood', 'neighborhood_rent_index_cluster']]
self.df = pd.merge(self.df, neighborhood_mapping, on='neighborhood')

##### Convertimos las categóricas de object a category #####
def convert_object_to_category(self) -> pd.DataFrame:
    """
    Convert object columns in a pandas dataframe to category data

    Parameters:
    df (pd.DataFrame): Input pandas dataframe

    Returns:
    pd.DataFrame: A copy of the input dataframe with object column converted
    """
    # loop through columns and convert object columns to category
    for col in self.df.columns:
        if self.df[col].dtype == 'object':
            self.df[col] = self.df[col].astype('category')

    return self.df

##### Tratamiento de sq_meters_built en relación al número de habitaciones #####
def bathrooms_rooms_per_100_sqm(self) -> None:
    self.df['rooms_per_100_sqm'] = self.df['rooms'] * 100 / self.df['sq_meters']
    self.df['bathrooms_per_100_sqm'] = self.df['bathrooms'] * 100 / self.df['sq_meters']

def feature_selection_2(self) -> None:
    ##### Feature Selection 2: Eliminación de variables no útiles #####
    # variables a eliminar
    not_useful_vars = ['year_built', 'sq_meters', 'floor', 'property_type']
    # eliminamos not_useful_vars
    self.df.drop(not_useful_vars, axis=1, inplace=True)

In [991]: file_path = 'test_data.csv'

data_preprocessor = DataPreprocessor()
data_preprocessor.test_data_preprocessor(file_path)

# guardamos en csv
data_preprocessor.df.to_csv('test_data_preprocessed.csv', index=False)
```

```
In [417]: # cargamos el dataset de test preprocesado
test_data_preprocessed = pd.read_csv('test_data_preprocessed.csv', delimiter=',')
# convertimos object a category
test_data_preprocessed = convert_object_to_category(test_data_preprocessed)

explore_data(test_data_preprocessed)
```

The dataset includes 1170 instances (rows) and 26 variables (columns).

	price	latitude	longitude	sq_meters_built	rooms	bathrooms	balcony	terrace	exterior
0	133000	41.421077	2.213120		76	3	1	0	0
1	485000	41.380788	2.152472		130	4	2	0	0
2	265000	41.406803	2.183533		77	3	1	0	0
3	154500	41.433890	2.172277		61	2	1	1	0
4	1200000	41.393411	2.145017		120	3	2	0	0

## 1.21 MODELING

Queremos predecir el precio de venta de la vivienda en base al dataset de venta.

En primer lugar generaremos un DataFrame para almacenar los resultados obtenidos de todos los modelos que generemos:

```
In [640]: models_results_df = pd.DataFrame(columns=['regressor', 'train_r2_score'])
```

- Almacenaremos los resultados para los distintos modelos en model\_results\_df.

### 1.21.1 Bagging & Random Forest Models

Haremos 3 aproximaciones:

- Modelo 1: BaggingRegressor y RandomForestRegressor con todas las variables del dataset sin seleccionar variables importantes, sin tratar outliers y sin escalar las variables numéricas.
- Modelo 2: BaggingRegressor y RandomForestRegressor con todas las variables del dataset sin seleccionar variables importantes, tratando outliers (método IQR) y escalando las variables numéricas (MinMaxScaler).

- Modelo 3: BaggingRegressor y RandomForestRegressor seleccionando las variables más importantes (mediante técnicas como PCA, Análisis de Correlación, Información Mutua, ANOVA y ExtraTreesRegressor), tratando outliers (método IQR) y escalando las variables numéricas (MinMaxScaler).
  - Modelo 3.1: BaggingRegressor y RandomForestRegressor seleccionando las variables más importantes (mediante técnicas como PCA, Análisis de Correlación, Información Mutua, ANOVA y ExtraTreesRegressor), tratando outliers (método IQR) pero NO escalando las variables numéricas.

#### 1.21.1.1 Models Pipeline

Creamos la Pipeline con los modelos de Bagging y RF:

```
In [614]: class PipelineRunner:  
    """A class that runs multiple regression models on given train and test dataframes.  
  
    Args:  
        cv_folds: Number of cross validation folds to be used in grid search.  
    """  
    self.cv_folds = cv_folds  
  
    def generate_train_test_sets(self,  
                                 df_train: pd.DataFrame,  
                                 df_test: pd.DataFrame,  
                                 target: str) -> Tuple[pd.DataFrame, pd.DataFrame]:  
        """  
        Generate training and test sets.  
  
        Args:  
            df_train (pd.DataFrame): Training dataset.  
            df_test (pd.DataFrame): Test dataset.  
            target (str): Target variable name.  
  
        Returns:  
            X_train (pd.DataFrame): Training features.  
            y_train (pd.Series): Training target variable.  
            X_test (pd.DataFrame): Test features.  
            y_test (pd.Series): Test target variable.  
        """  
        X_train = df_train.drop(columns=[target])  
        y_train = df_train[target]  
  
        X_test = df_test.drop(columns=[target])  
        y_test = df_test[target]  
  
        return X_train, y_train, X_test, y_test  
  
    def run_pipeline(self,  
                    df_train: pd.DataFrame,  
                    df_test: pd.DataFrame,  
                    target: str) -> GridSearchCV:  
        """  
        Run multiple regression models on given train and test dataframes.  
  
        Args:  
            df_train (pd.DataFrame): Training dataset.  
            df_test (pd.DataFrame): Test dataset.  
            target (str): Target variable name.  
  
        Returns:  
            A GridSearchCV object containing the fitted regression models.  
        """  
        X_train, y_train, X_test, y_test = self.generate_train_test_sets(df_train, df_test, target)  
  
        models = [  
            {
```

```
'name': 'BaggingRegressor',
'model': BaggingRegressor(DecisionTreeRegressor()),
'params': {
    'model__n_estimators': [10, 20, 50, 100, 200]
}
},
{
    'name': 'RandomForestRegressor',
'model': RandomForestRegressor(),
'params': {
    'model__n_estimators': [10, 20, 50, 100, 200],
    'model__max_depth': [2, 3, 4, 5, 6, 7, 9],
    'model__min_samples_split': [2, 5, 10]
}
}
]

results = []
for model_info in models:
    model = model_info['model']
    params = model_info['params']

    pipeline = Pipeline(steps=[
        ('model', model)
    ])

    grid_search = GridSearchCV(pipeline, params, cv=self.cv_fo
grid_search.fit(X_train, y_train)

y_train_pred = grid_search.predict(X_train)
train_r2_score = r2_score(y_train, y_train_pred)

y_test_pred = grid_search.predict(X_test)
test_r2_score = r2_score(y_test, y_test_pred)

results.append({
    'name': model_info['name'],
    'best_params': grid_search.best_params_,
    'best_score': grid_search.best_score_,
    'train_r2_score': train_r2_score,
    'test_r2_score': test_r2_score
})
print(f"Model: {model_info['name']}")
print(f"Best model parameters: {grid_search.best_params_}")

# median R^2 score for the training set across all folds i
print(f"Best cross-validation R^2 score: {grid_search.best

print(f"Train R^2 score: {train_r2_score:.4f}")
print(f"Test R^2 score: {test_r2_score:.4f}")
print()

best_model = max(results, key=lambda x: x['best_score'])
print(f"Best model: {best_model['name']}")
print(f"Best model parameters: {best_model['best_params']}")
# median R^2 score for the training set across all folds in th
```

```

        print(f"Best cross-validation R^2 score: {best_model['best_score']}")
        print(f"Train R^2 score: {best_model['train_r2_score']:.4f}")
        print(f"Test R^2 score: {best_model['test_r2_score']:.4f}")

    return results #grid_search

def predict(self, df: pd.DataFrame, model: GridSearchCV) -> np.ndarray:
    """
    Make predictions using a fitted regression model.

    Args:
        df (pd.DataFrame): Preprocessed dataset to be used for prediction.
        model (GridSearchCV): Fitted regression model.

    Returns:
        Predictions of the target variable.
    """
    X = df.drop(columns=[model.best_params_['model']])
    return model.predict(X)

```

### 1.21.1.2 Modelo 1

Modelo sin escalar los atributos ni hacer una selección de las variables predictoras más relevantes.

- Train dataset: bagging\_df1
- Test dataset: bagging\_df1\_test

#### 1.21.1.2.1 Train dataset (bagging\_df1) y Test dataset (bagging\_df1\_test)

Debemos codificar las variables categóricas para realizar los modelos de bagging (modelos de bagging solo aceptan variables numéricas).

Para ello utilizaremos:

- one-hot encoding: para las variables que no siguen un orden o jerarquía
- ordinal encoder: para las que hay un orden o jerarquía, como floor\_cat, property\_age\_cat y neighborhood\_rent\_index\_cluster. Esto garantiza que los valores codificados reflejen el orden inherente de las categorías y puedan utilizarse para el análisis.

In [615]: # creamos los datasets para el primer modelo de Bagging Y RF  
bagging\_df1 = sales\_simplified\_df.copy(deep=True)  
bagging\_df1\_test = test\_data\_preprocessed.copy(deep=True)

#### Ordinal Encoding

Codificamos las variables floor\_cat, property\_age\_cat y neighborhood\_rent\_index\_cluster utilizando ordinal encoding.

Codificaremos la categoría "Unknown" como una etiqueta separada: la sustituimos por -1. Esta es una práctica común cuando se trata de datos desconocidos o faltantes.

In [788]: `bagging_df1.floor_cat.value_counts()`

Out[788]:

0 – 4	2359
Unknown	1680
5 – 11	623
+11	15
Name: floor_cat, dtype: int64	

In [789]: `bagging_df1.property_age_cat.value_counts()`

Out[789]:

40 – 70	1933
Unknown	1415
70 – 120	500
0 – 40	389
120 – 150	300
+150	140
Name: property_age_cat, dtype: int64	

In [790]: `bagging_df1.neighborhood_rent_index_cluster.value_counts()`

Out[790]:

0 – 2337	1571
3371 – 5042	1403
2337 – 3371	1060
+5042	643
Name: neighborhood_rent_index_cluster, dtype: int64	

```
In [616]: # variables que queremos codificar usando ordinal encoder
oe_cols = oe_cols = ['floor_cat', # unknown
                     'property_age_cat', # unknown
                     'neighborhood_rent_index_cluster'
                    ]
# Define the order of the categories for each column
floor_cat_order = ['Unknown', '0 - 4', '5 - 11', '+11']
property_age_cat_order = ['Unknown', '0 - 40', '40 - 70', '70 - 120',
                           neighborhood_rent_index_cluster_order = ['Unknown', '0 - 2337', '2337

# Create an instance of the ordinal encoder
encoder = OrdinalEncoder(categories=[floor_cat_order,
                                      property_age_cat_order,
                                      neighborhood_rent_index_cluster_o

for col in oe_cols:
    # Create a new column name for the encoded data
    new_col = col + '_encoded'
    # Create an instance of the ordinal encoder
    encoder = OrdinalEncoder(categories=[eval(col+'_order')])
    # Fit and transform the selected columns
    bagging_df1[new_col] = encoder.fit_transform(bagging_df1[[col]])
    bagging_df1_test[new_col] = encoder.fit_transform(bagging_df1_test)
    # replace 0 by -1 ('Unknown' by -1)
    bagging_df1[new_col] = bagging_df1[new_col].replace(0, -1)
    bagging_df1_test[new_col] = bagging_df1_test[new_col].replace(0, -1

# Print the encoded data
display(bagging_df1[['floor_cat', 'floor_cat_encoded', 'property_age_cat',
display(bagging_df1_test[['floor_cat', 'floor_cat_encoded', 'property_ag
```

	floor_cat	floor_cat_encoded	property_age_cat	property_age_cat_encoded	neighborhood_ren
0	0 - 4	1.0	0 - 40		1.0
1	Unknown	-1.0	40 - 70		2.0
2	0 - 4	1.0	Unknown		-1.0
3	Unknown	-1.0	40 - 70		2.0
4	5 - 11	2.0	70 - 120		3.0

	floor_cat	floor_cat_encoded	property_age_cat	property_age_cat_encoded	neighborhood_ren
0	0 - 4	1.0	40 - 70		2.0
1	Unknown	-1.0	70 - 120		3.0
2	5 - 11	2.0	40 - 70		2.0
3	0 - 4	1.0	40 - 70		2.0
4	0 - 4	1.0	70 - 120		3.0

- Ya tenemos las variables codificadas usando ordinal encoding.

## One-hot Encoding

Codificamos las variables binarias junto con orientation, heating y property\_type utilizando OHE.

```
In [617]: def one_hot_encode(df: pd.DataFrame, cat_features: List[str]) -> pd.DataFrame:
    """
    Encodes categorical features in a dataframe using scikit-learn's OneHotEncoder.

    Args:
        df (pandas.DataFrame): input dataframe
        cat_features (list): list of categorical feature column names

    Returns:
        pandas.DataFrame: dataframe with categorical features one-hot encoded
    """
    # create OneHotEncoder object and fit on categorical features
    encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
    encoder.fit(df[cat_features])

    # get category names from encoder
    cat_names = encoder.get_feature_names_out(cat_features)

    # transform categorical features to one-hot encoded features and concat them
    encoded_cat_features = encoder.transform(df[cat_features])
    df_encoded = pd.DataFrame(encoded_cat_features, columns=cat_names)

    # add original numerical features to encoded features dataframe
    num_features = df.drop(cat_features, axis=1)
    df_encoded = pd.concat([num_features.reset_index(drop=True), df_encoded], axis=1)

    return df_encoded
```

```
In [618]: # variables que queremos codificar usando OHE
ohe_cols = get_binary_cols(sales_simplified_df) + ['orientation', 'heating', 'property_type']
# binary variables:
# ['balcony',
#  'terrace',
#  'exterior',
#  'rooftop',
#  'elevator',
#  'pool',
#  'ac',
#  'garage',
#  'garden']
```

```
In [619]: # aplicamos one - hot encoding al dataset
bagging_df1 = one_hot_encode(bagging_df1, ohe_cols)
bagging_df1_test = one_hot_encode(bagging_df1_test, ohe_cols)

display(bagging_df1.head())
display(bagging_df1.shape)

display(bagging_df1_test.head())
display(bagging_df1_test.shape)
```

`sparse` was renamed to `sparse\_output` in version 1.2 and will be removed in 1.4. `sparse\_output` is ignored unless you leave `sparse` to its default value.

`sparse` was renamed to `sparse\_output` in version 1.2 and will be removed in 1.4. `sparse\_output` is ignored unless you leave `sparse` to its default value.

	price	latitude	longitude	sq_meters_built	rooms	bathrooms	neighborhood	dist_city_
0	2290000	41.409071	2.099850		532	5	6	Vallvidrera - El Tibidabo i les Planes
1	90000	41.434545	2.171110		50	1	1	Can Peguera - El Turó de la Peira
2	145000	41.444518	2.175309		53	3	1	Verdun
3	675000	41.392209	2.153368		120	4	2	L'Antiga Esquerra de l'Eixample
4	319000	41.413385	2.162246		69	3	1	El Baix Guinardó

(4675, 53)

	price	latitude	longitude	sq_meters_built	rooms	bathrooms	neighborhood	dist_city_
0	133000	41.421077	2.213120		76	3	1	Sant Martí
1	485000	41.380788	2.152472		130	4	2	La Nova Esquerra de l'Eixample
2	265000	41.406803	2.183533		77	3	1	El Camp de l'Arpa del Clot
3	154500	41.433890	2.172277		61	2	1	Porta
4	1200000	41.393411	2.145017		120	3	2	Sant Gervasi - Galvany

(1170, 53)

Eliminamos las variables categóricas no codificadas:

```
In [620]: no_encoded_cat_vars = bagging_df1.select_dtypes(include=['category']).  
no_encoded_cat_vars
```

```
Out[620]: ['neighborhood',  
          'property_age_cat',  
          'floor_cat',  
          'neighborhood_rent_index_cluster']
```

```
In [621]: no_encoded_cat_vars_test = bagging_df1_test.select_dtypes(include=['ca  
no_encoded_cat_vars_test
```

```
Out[621]: ['neighborhood',  
          'property_age_cat',  
          'floor_cat',  
          'neighborhood_rent_index_cluster']
```

```
In [622]: bagging_df1.drop(no_encoded_cat_vars, axis=1, inplace=True)  
bagging_df1_test.drop(no_encoded_cat_vars_test, axis=1, inplace=True)
```

```
In [623]: # exploración rápida del df  
print("TRAIN: ")  
explore_data(bagging_df1)  
print("\n\n")  
print("TEST: ")  
explore_data(bagging_df1_test)
```

TRAIN:

The dataset includes 4675 instances (rows) and 49 variables (columns).

	price	latitude	longitude	sq_meters_built	rooms	bathrooms	dist_city_center	dist_clo
0	2290000	41.409071	2.099850	532	5	6	6.302847	
1	90000	41.434545	2.171110	50	1	1	5.189265	
2	145000	41.444518	2.175309	53	3	1	6.313668	
3	675000	41.392209	2.153368	120	4	2	1.462060	
4	319000	41.413385	2.162246	69	3	1	2.906147	

<class 'pandas.core.frame.DataFrame'>

- No tenemos valores nulos y todas las variables son numéricas.

### 1.21.1.2.2 Model 1: Bagging & RF

```
In [624]: # create an instance of PipelineRunner
pipeline_runner = PipelineRunner()
# call the run_pipeline() method, passing in the dataframes and target
results = pipeline_runner.run_pipeline(df_train=bagging_df1,
                                         df_test=bagging_df1_test,
                                         target=TARGET_VARIABLE)
# examine the results for each model
results
```

Model: BaggingRegressor  
Best model parameters: {'model\_\_n\_estimators': 100}  
Best cross-validation R<sup>2</sup> score: 0.8413  
Train R<sup>2</sup> score: 0.9767  
Test R<sup>2</sup> score: 0.7393

Model: RandomForestRegressor  
Best model parameters: {'model\_\_max\_depth': 9, 'model\_\_min\_samples\_split': 10, 'model\_\_n\_estimators': 20}  
Best cross-validation R<sup>2</sup> score: 0.8376  
Train R<sup>2</sup> score: 0.9392  
Test R<sup>2</sup> score: 0.7231

Best model: BaggingRegressor  
Best model parameters: {'model\_\_n\_estimators': 100}  
Best cross-validation R<sup>2</sup> score: 0.8413  
Train R<sup>2</sup> score: 0.9767  
Test R<sup>2</sup> score: 0.7393

```
Out[624]: [{"name": "BaggingRegressor",
  "best_params": {"model__n_estimators": 100},
  "best_score": 0.8413459550193905,
  "train_r2_score": 0.9766845778640442,
  "test_r2_score": 0.739290218306422},
 {"name": "RandomForestRegressor",
  "best_params": {"model__max_depth": 9,
    "model__min_samples_split": 10,
    "model__n_estimators": 20},
  "best_score": 0.8376489876860275,
  "train_r2_score": 0.9392475407127426,
  "test_r2_score": 0.7231028416313489}]
```

```
In [655]: # Loop through each model in results
for model in results:
    model_name = model['name']+ '_1'
    train_r2_score = model['train_r2_score']
    test_r2_score = model['test_r2_score']

    # Append the results to the DataFrame
    models_results_df = pd.concat([models_results_df, pd.DataFrame({'r':model_name, 'tr':train_r2_score, 'te':test_r2_score})], ignore_index=True)

# Display the DataFrame
display(models_results_df)
```

	regressor	train_r2_score	test_r2_score
0	BaggingRegressor_1	0.976685	0.739290
1	RandomForestRegressor_1	0.939248	0.723103

### 1.21.1.3 Modelo 2

En los modelos anteriores (BaggingRegressor y RandomForestRegressor) nos se han escalado las variables, no se han quitado los outliers y no se ha hecho una selección de variables.

Por tanto, ahora escalaremos las variables numéricas y trataremos los outliers. La selección de variables predictoras más importantes la realizaremos en el tercer modelo.

#### 1.21.1.3.1 Train dataset (bagging\_df2) y Test dataset (bagging\_df2\_test)

Primero, realizamos el mismo preprocesamiento que en el primer modelo:

```
In [626]: # creamos los datasets para el primer modelo de Bagging
bagging_df2 = sales_simplified_df.copy(deep=True)
bagging_df2_test = test_data_preprocessed.copy(deep=True)

# variables que queremos codificar usando ordinal encoder
oe_cols = oe_cols = ['floor_cat', # unknown
                     'property_age_cat', # unknown
                     'neighborhood_rent_index_cluster'
                    ]
# Define the order of the categories for each column
floor_cat_order = ['Unknown', '0 - 4', '5 - 11', '+11']
property_age_cat_order = ['Unknown', '0 - 40', '40 - 70', '70 - 120',
                           '120 - 2337', '2337']
neighborhood_rent_index_cluster_order = ['Unknown', '0 - 2337', '2337']

# Create an instance of the ordinal encoder
encoder = OrdinalEncoder(categories=[floor_cat_order,
                                      property_age_cat_order,
                                      neighborhood_rent_index_cluster_order])

for col in oe_cols:
    # Create a new column name for the encoded data
    new_col = col + '_encoded'
    # Create an instance of the ordinal encoder
    encoder = OrdinalEncoder(categories=[eval(col+'_order')])
    # Fit and transform the selected columns
    bagging_df2[new_col] = encoder.fit_transform(bagging_df2[[col]])
    bagging_df2_test[new_col] = encoder.fit_transform(bagging_df2_test)
    # replace 0 by -1 ('Unknown' by -1)
    bagging_df2[new_col] = bagging_df2[new_col].replace(0, -1)
    bagging_df2_test[new_col] = bagging_df2_test[new_col].replace(0, -1)

# variables que queremos codificar usando OHE
ohe_cols = get_binary_cols(sales_simplified_df) + ['orientation', 'heat']
# binary variables:
# ['balcony',
#  'terrace',
#  'exterior',
#  'rooftop',
#  'elevator',
#  'pool',
#  'ac',
#  'garage',
#  'garden']

# aplicamos one - hot encoding al dataset
bagging_df2 = one_hot_encode(bagging_df2, ohe_cols)
bagging_df2_test = one_hot_encode(bagging_df2_test, ohe_cols)

# eliminamos variables categóricas no codificadas
no_encoded_cat_vars = bagging_df2.select_dtypes(include=['category'])
bagging_df2.drop(no_encoded_cat_vars, axis=1, inplace=True)

no_encoded_cat_vars_test = bagging_df2_test.select_dtypes(include=['category'])
bagging_df2_test.drop(no_encoded_cat_vars_test, axis=1, inplace=True)
```

```
# exploración rápida del df
print("TRAIN: ")
explore_data(bagging_df2)
print("\n\n")
print("TEST: ")
explore_data(bagging_df2_test)
TRAIN:
The dataset includes 4675 instances (rows) and 49 variables (columns).
```

`sparse` was renamed to `sparse\_output` in version 1.2 and will be removed in 1.4. `sparse\_output` is ignored unless you leave `sparse` to its default value.

`sparse` was renamed to `sparse\_output` in version 1.2 and will be removed in 1.4. `sparse\_output` is ignored unless you leave `sparse` to its default value.

	price	latitude	longitude	sq_meters_built	rooms	bathrooms	dist_city_center	dist_clo
0	2290000	41.409071	2.099850	532	5	6	6.302847	
1	90000	41.434545	2.171110	50	1	1	5.189265	
2	145000	41.444518	2.175309	53	3	1	6.313668	

## Tratamiento de Outliers

Empezaremos por la eliminación de outliers antes de escalar las variables, ya que los outliers pueden afectar la distribución de los datos y hacer que los resultados del escalado no sean los esperados.

Dado que la mayoría de las variables numéricas tienen muchos valores atípicos, usaremos el método IQR para tratar los outliers.

El método IQR es un método robusto para la detección y eliminación de valores atípicos (puede tratar un gran número de valores atípicos en los datos). Además, es menos sensible a los valores extremos que el método Z-score, lo que lo hace más adecuado para un dataset con muchos outliers.

Para utilizar el método IQR, primero calcularemos el IQR para cada variable numérica. A continuación, identificaremos cualquier data point que caiga por debajo de  $Q1 - 1.5 * IQR$  o por encima de  $Q3 + 1.5 * IQR$  como un valor atípico y lo eliminaremos del conjunto de datos.

```
In [627]: def remove_outliers_iqr(df: pd.DataFrame, cols: List[str], threshold: float):
    """
    Remove outliers from a pandas DataFrame using the interquartile range method.

    Args:
        df: A pandas DataFrame containing the data.
        cols: A list of columns to remove outliers from.
        threshold: The number of IQRs beyond which a data point is considered an outlier.
                    Defaults to 1.5.

    Returns:
        A pandas DataFrame with the outliers removed.
    """
    # Calculate the IQR for each specified column
    Q1 = df[cols].quantile(0.25)
    Q3 = df[cols].quantile(0.75)
    IQR = Q3 - Q1

    # Determine the threshold values for each specified column
    thresholds = (Q1 - threshold * IQR, Q3 + threshold * IQR)

    # Identify and remove the outliers
    is_outlier = ((df[cols] < thresholds[0]) | (df[cols] > thresholds[1]))
    df_clean = df.loc[~is_outlier, :]

    return df_clean
```

```
In [628]: # variables numéricas
num_vars = ['price', 'latitude', 'longitude',
            'sq_meters_built', 'rooms', 'bathrooms',
            'dist_city_center', 'dist_closest_station',
            'rooms_per_100_sqm', 'bathrooms_per_100_sqm']
```

```
In [629]: # Remove outliers using the IQR method
bagging_df2 = remove_outliers_iqr(bagging_df2, num_vars)
bagging_df2_test = remove_outliers_iqr(bagging_df2_test, num_vars)
```

```
In [630]: # exploración rápida del df
print("TRAIN: ")
print(bagging_df2.shape)
print("TEST: ")
print(bagging_df2_test.shape)
```

```
TRAIN:
(3669, 49)
TEST:
(898, 49)
```

## Escalar los atributos

Escalamos los atributos ya que, por ejemplo, los metros cuadrados o el número de habitaciones, están en una escala diferente a otras características, como la ubicación (longitud y latitud) o el año de construcción. Escalar las características de entrada a una

escala común puede ayudar a garantizar que todas estas características reciben el mismo tratamiento en el modelo.

Como los datos no siguen una distribución normal y ya hemos tratado los valores atípicos utilizando la técnica IQR, entonces utilizar el MinMaxScaler puede ser una buena opción para escalar las variables numéricas.

El MinMaxScaler escala los atributos a un rango fijo entre 0 y 1, por lo que es apropiado cuando los atributos no tienen una distribución normal o tienen un rango limitado.

```
In [631]: def scale_columns(df: pd.DataFrame, columns: List[str]) -> pd.DataFrame:
    """
    Applies MinMaxScaler to the specified columns of a pandas DataFrame.

    Parameters:
    -----
    df : pd.DataFrame
        The pandas DataFrame to scale.
    columns : List[str]
        The list of column names to scale.

    Returns:
    -----
    pd.DataFrame
        The scaled pandas DataFrame.
    """

    # Create a copy of the DataFrame
    df_scaled = df.copy()

    # Create a MinMaxScaler object
    scaler = MinMaxScaler()

    # Scale the specified columns
    df_scaled[columns] = scaler.fit_transform(df_scaled[columns])

    return df_scaled
```

```
In [632]: # Scale the columns
bagging_df2 = scale_columns(bagging_df2, num_vars)
bagging_df2_test = scale_columns(bagging_df2_test, num_vars)
```

El siguiente código hace lo mismo pero usando un Pipeline:

```
In [633]: class OutlierRemover(BaseEstimator, TransformerMixin):
    def __init__(self, cols=None, threshold=1.5):
        self.cols = cols
        self.threshold = threshold

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        X_clean = X.copy()
        # Calculate the IQR for each specified column
        Q1 = X_clean[self.cols].quantile(0.25)
        Q3 = X_clean[self.cols].quantile(0.75)
        IQR = Q3 - Q1

        # Determine the threshold values for each specified column
        thresholds = (Q1 - self.threshold * IQR, Q3 + self.threshold *

        # Identify and remove the outliers
        is_outlier = ((X_clean[self.cols] < thresholds[0]) | (X_clean[
        X_clean = X_clean.loc[~is_outlier, :]

    return X_clean

class MinMaxScalerTransformer(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.scaler = MinMaxScaler()

    def fit(self, X, y=None):
        self.scaler.fit(X)
        return self

    def transform(self, X):
        X_scaled = self.scaler.transform(X)
        feature_names = [f'{col}' for col in X.columns]
        return pd.DataFrame(X_scaled, columns=feature_names)
```

```
In [1248]: # define the preprocessing pipeline
preprocessing_pipeline = Pipeline([
    ('outlier_removal', OutlierRemover(cols=num_vars)),
    ('scaler', ColumnTransformer([('num_scaler', MinMaxScaler(), num_v
                                remainder='passthrough'))
])

# Apply the preprocessing pipeline to the training and testing data
# Fit and transform the training data
bagging_df2 = pd.DataFrame(preprocessing_pipeline.fit_transform(baggin
bagging_df2_test = pd.DataFrame(preprocessing_pipeline.fit_transform(b
```

### 1.21.1.3.2 Model 2: Bagging & RF

Una vez hemos tratado los outliers y hemos escalado los atributos numéricos, vemos cómo se comportan los modelos:

```
In [635]: # create an instance of PipelineRunner
pipeline_runner = PipelineRunner()
# call the run_pipeline() method, passing in the dataframes and target
results_2 = pipeline_runner.run_pipeline(df_train=bagging_df2,
                                         df_test=bagging_df2_test,
                                         target=TARGET_VARIABLE)
# examine the results for each model
results_2
```

Model: BaggingRegressor

Best model parameters: {'model\_\_n\_estimators': 200}

Best cross-validation R<sup>2</sup> score: 0.7708

Train R<sup>2</sup> score: 0.9701

Test R<sup>2</sup> score: 0.6908

Model: RandomForestRegressor

Best model parameters: {'model\_\_max\_depth': 9, 'model\_\_min\_samples\_split': 2, 'model\_\_n\_estimators': 200}

Best cross-validation R<sup>2</sup> score: 0.7601

Train R<sup>2</sup> score: 0.9245

Test R<sup>2</sup> score: 0.6744

Best model: BaggingRegressor

Best model parameters: {'model\_\_n\_estimators': 200}

Best cross-validation R<sup>2</sup> score: 0.7708

Train R<sup>2</sup> score: 0.9701

Test R<sup>2</sup> score: 0.6908

```
Out[635]: [{"name": "BaggingRegressor",
  "best_params": {"model__n_estimators": 200},
  "best_score": 0.7707652188812547,
  "train_r2_score": 0.97011092170689,
  "test_r2_score": 0.6907957655585963},
 {"name": "RandomForestRegressor",
  "best_params": {"model__max_depth": 9,
    "model__min_samples_split": 2,
    "model__n_estimators": 200},
  "best_score": 0.7600922045988325,
  "train_r2_score": 0.9245244491203484,
  "test_r2_score": 0.6744430145609775}]
```

```
In [656]: # Loop through each model in results
for model in results_2:
    model_name = model['name']+ '_2'
    train_r2_score = model['train_r2_score']
    test_r2_score = model['test_r2_score']

    # Append the results to the DataFrame
    models_results_df = pd.concat([models_results_df, pd.DataFrame({'r':train_r2_score, 't':test_r2_score, 'model':model_name})], ignore_index=True)

# Display the DataFrame
display(models_results_df)
```

	regressor	train_r2_score	test_r2_score
0	BaggingRegressor_1	0.976685	0.739290
1	RandomForestRegressor_1	0.939248	0.723103
2	BaggingRegressor_2	0.970111	0.690796
3	RandomForestRegressor_2	0.924524	0.674443

- Parece que de momento, el tratamiento de outliers y el escalado de las variables numéricas no han mejorado el rendimiento de los modelos, sino todo lo contrario.

#### 1.21.1.4 Modelo 3

Recordemos lo que obtuvimos de cada técnica (para seleccionar las mejores variables predictoras del precio de la vivienda):

- Del análisis de componentes principales (PCA), observamos que las variables sq\_meters\_built, bathrooms, longitude, dist\_city\_center, latitude, bathrooms\_per\_100\_sqm y rooms\_per\_100\_sqm tenían los mayores pesos en los primeros 3 componentes principales, lo que indica que estas variables contribuyen significativamente a la variación total de los datos. Sin embargo, se requiere un análisis adicional para determinar si estas variables tienen una relación significativa con el precio de la vivienda y cómo se relacionan con él.
- Del análisis de información mutua (recordemos que una puntuación más alta significa que la característica es más informativa para predecir la variable objetivo) vimos que "sq\_meters\_built" tenía la puntuación de información mutua más alta, lo que significa que puede ser el atributo más informativo para predecir el precio de la vivienda. Del mismo modo, "rooms\_per\_100\_sqm" y "bathrooms\_per\_100\_sqm" también tenían puntuaciones de información mutua relativamente altas, lo que indica que también son atributos informativos. Otros atributos que tienen puntuaciones de información mutua relativamente altas eran "latitude", "bathrooms" y "neighborhood\_rent\_index\_cluster", mientras que atributos como "property\_type\_duplex", "property\_type\_chalet\_adosado" y "heating\_electric" tenían puntuaciones muy bajas, lo que indica que son menos informativos a la hora predecir el precio de la vivienda.

- Del análisis ANOVA (y Kruskall-Wallis) vimos que las variables con mayor efecto sobre el precio de venta de la vivienda eran neighborhood\_rent\_index\_cluster, garage, garden, pool, terrace, property\_type, property\_age\_cat, elevator, ac, heating y property\_age\_cat.
- Al realizar el método ExtraTreesRegressor vimos que que según el modelo, sq\_meters\_built es el atributo más importante para predecir el precio de la vivienda, seguida de bathrooms y neighborhood\_rent\_index\_cluster\_encoded. Otras características como rooms\_per\_100\_sqm, garden y bathrooms\_per\_100\_sqm también tenían puntuaciones de importancia relativamente altas.

Al considerar los resultados de los diferentes métodos, parece que "sq\_meters\_built", "bathrooms", "longitude", "dist\_city\_center", "latitude", "bathrooms\_per\_100\_sqm", "rooms\_per\_100\_sqm", "neighborhood\_rent\_index\_cluster", "garage", "garden", "pool", "terrace", "property\_type", "property\_age\_cat", "elevator", "ac", "heating" y "property\_age\_cat" son las variables más importantes o informativas para predecir el precio de la vivienda en Barcelona. Recordemos que "bathrooms" estaba fuertemente correlacionado con "sq\_meters\_built" y por eso creamos "bathrooms\_per\_100\_sqm". Por tanto, nos quedamos con "bathrooms\_per\_100\_sqm".

Por tanto, realizaremos los modelos de Bagging y Random Forest con estas variables. Como los resultados han mejorado después de tratar los outliers y escalar los atributos numéricos, relizaremos el mismo proceso con las variables seleccionadas.

#### **1.21.1.4.1 Train dataset (*bagging\_df3*) y Test dataset (*bagging\_df3\_test*)**

```
In [636]: # creamos los datasets para el tercer modelo de Bagging y RF
bagging_df3 = sales_simplified_df.copy(deep=True)
bagging_df3_test = test_data_preprocessed.copy(deep=True)

# lista contiene las variables que se consideran más importantes o inf
# para predecir el precio de la vivienda en Barcelona, según los diferentes predictores
good_predictors = ["price", # target variable
                    "sq_meters_built", "longitude",
                    "dist_city_center", "latitude", "bathrooms_per_100_rooms",
                    "rooms_per_100_sqm", "neighborhood_rent_index_cluster",
                    "garage", "garden", "pool", "terrace", "property_type",
                    "property_age_cat", "elevator", "ac", "heating"]

# nos quedamos sólo con las columnas indicadas en la lista good_predictors
bagging_df3 = bagging_df3[good_predictors]
bagging_df3_test = bagging_df3_test[good_predictors]

# variables que queremos codificar usando ordinal encoder
oe_cols = oe_cols = ['property_age_cat', 'neighborhood_rent_index_cluster']
# Define the order of the categories for each column
property_age_cat_order = ['Unknown', '0 - 40', '40 - 70', '70 - 120',
                           '120 - 2337', '2337 +']
neighborhood_rent_index_cluster_order = ['Unknown', '0 - 2337', '2337 +']
# Create an instance of the ordinal encoder
encoder = OrdinalEncoder(categories=[property_age_cat_order,
                                      neighborhood_rent_index_cluster_order])

for col in oe_cols:
    # Create a new column name for the encoded data
    new_col = col + '_encoded'
    # Create an instance of the ordinal encoder
    encoder = OrdinalEncoder(categories=[eval(col+'_order')])
    # Fit and transform the selected columns
    bagging_df3[new_col] = encoder.fit_transform(bagging_df3[[col]])
    bagging_df3_test[new_col] = encoder.fit_transform(bagging_df3_test)
    # replace 0 by -1 ('Unknown' by -1)
    bagging_df3[new_col] = bagging_df3[new_col].replace(0, -1)
    bagging_df3_test[new_col] = bagging_df3_test[new_col].replace(0, -1)

# variables que queremos codificar usando OHE
ohe_cols = ["garage", "garden", "pool",
            "terrace", "property_type",
            "elevator", "ac", "heating"]

# aplicamos one - hot encoding al dataset
bagging_df3 = one_hot_encode(bagging_df3, ohe_cols)
bagging_df3_test = one_hot_encode(bagging_df3_test, ohe_cols)

# eliminamos variables categóricas no codificadas
no_encoded_cat_vars = bagging_df3.select_dtypes(include=['category'])
bagging_df3.drop(no_encoded_cat_vars, axis=1, inplace=True)
no_encoded_cat_vars_test = bagging_df3_test.select_dtypes(include=['category'])
bagging_df3_test.drop(no_encoded_cat_vars_test, axis=1, inplace=True)

##### Tratamos los outliers y escalamos los atributos numéricos
num_vars = ["price", "sq_meters_built", "longitude", "dist_city_center"]
```

```

# define the preprocessing pipeline
preprocessing_pipeline = Pipeline([
    ('outlier_removal', OutlierRemover(cols=num_vars)),
    ('scaler', ColumnTransformer([('num_scaler', MinMaxScaler(), num_vars),
                                 ('cat_scaler', OneHotEncoder(), cat_vars),
                                 ('remainder', 'passthrough')])
])

# Apply the preprocessing pipeline to the training and testing data
# Fit and transform the training data
bagging_df3 = pd.DataFrame(preprocessing_pipeline.fit_transform(bagging_df))
bagging_df3_test = pd.DataFrame(preprocessing_pipeline.fit_transform(bagging_df3))

# exploración rápida del df
print("TRAIN: ")
explore_data(bagging_df3)
print("\n\n")
print("TEST: ")
explore_data(bagging_df3_test)
TRAIN:
The dataset includes 3914 instances (rows) and 34 variables (columns).

```

`sparse` was renamed to `sparse\_output` in version 1.2 and will be removed in 1.4. `sparse\_output` is ignored unless you leave `sparse` to its default value.  
`sparse` was renamed to `sparse\_output` in version 1.2 and will be removed in 1.4. `sparse\_output` is ignored unless you leave `sparse` to its default value.

	price	sq_meters_built	longitude	dist_city_center	latitude	bathrooms_per_100_sqm	rooms
0	0.083558	0.149123	0.539966	0.598009	0.721144	0.555140	0.217251
1	0.157682	0.175439	0.578853	0.729485	0.811563	0.506260	0.217251
2	0.871968	0.763158	0.375652	0.162188	0.337326	0.411215	0.217251
3	0.200192	0.215780	0.457071	0.221045	0.500210	0.506260	0.217251

### 1.21.1.4.2 Model 3: Bagging & RF

Una vez hemos seleccionado las variables predictoras más importantes, hemos tratado los outliers y hemos escalado los atributos numéricos, vemos cómo se comportan los modelos:

```
In [1294]: # create an instance of PipelineRunner
pipeline_runner = PipelineRunner()
# call the run_pipeline() method, passing in the dataframes and target
results_3 = pipeline_runner.run_pipeline(df_train=bagging_df3,
                                         df_test=bagging_df3_test,
                                         target=TARGET_VARIABLE)
# examine the results for each model
results_3
```

```
Out[1294]: [{'model_name': 'BaggingRegressor',
  'best_params': {'model_n_estimators': 200},
  'train_r2_score': 0.982214048578808,
  'test_r2_score': 0.8512093886379279},
 {'model_name': 'RandomForestRegressor',
  'best_params': {'model_max_depth': 9,
                  'model_min_samples_split': 2,
                  'model_n_estimators': 200},
  'train_r2_score': 0.9479322101359778,
  'test_r2_score': 0.8462529129408495}]
```

```
In [637]: # create an instance of PipelineRunner
pipeline_runner = PipelineRunner()
# call the run_pipeline() method, passing in the dataframes and target
results_3 = pipeline_runner.run_pipeline(df_train=bagging_df3,
                                         df_test=bagging_df3_test,
                                         target=TARGET_VARIABLE)
# examine the results for each model
results_3
```

```
Model: BaggingRegressor
Best model parameters: {'model__n_estimators': 200}
Best cross-validation R^2 score: 0.8708
Train R^2 score: 0.9823
Test R^2 score: 0.8525

Model: RandomForestRegressor
Best model parameters: {'model__max_depth': 9, 'model__min_samples_split': 2, 'model__n_estimators': 100}
Best cross-validation R^2 score: 0.8618
Train R^2 score: 0.9469
Test R^2 score: 0.8451

Best model: BaggingRegressor
Best model parameters: {'model__n_estimators': 200}
Best cross-validation R^2 score: 0.8708
Train R^2 score: 0.9823
Test R^2 score: 0.8525
```

```
Out[637]: [{"name": "BaggingRegressor",
  "best_params": {"model__n_estimators": 200},
  "best_score": 0.8707953896715244,
  "train_r2_score": 0.9822737024925415,
  "test_r2_score": 0.8525076454575563},
 {"name": "RandomForestRegressor",
  "best_params": {"model__max_depth": 9,
    "model__min_samples_split": 2,
    "model__n_estimators": 100},
  "best_score": 0.8618490056134946,
  "train_r2_score": 0.9469431030838147,
  "test_r2_score": 0.8451119639621244}]
```

```
In [657]: # Loop through each model in results
for model in results_3:
    model_name = model['name']+ '_3'
    train_r2_score = model['train_r2_score']
    test_r2_score = model['test_r2_score']

    # Append the results to the DataFrame
    models_results_df = pd.concat([models_results_df, pd.DataFrame({'r':model_name, 'tr':train_r2_score, 'te':test_r2_score})], ignore_index=True)

# Display the DataFrame
display(models_results_df)
```

	regressor	train_r2_score	test_r2_score
0	BaggingRegressor_1	0.976685	0.739290
1	RandomForestRegressor_1	0.939248	0.723103
2	BaggingRegressor_2	0.970111	0.690796
3	RandomForestRegressor_2	0.924524	0.674443
4	BaggingRegressor_3	0.982274	0.852508
5	RandomForestRegressor_3	0.946943	0.845112

- Seleccionando las variables predictoras más importantes los modelos han funcionado mucho mejor.

#### 1.21.1.4.3 Model 3.1: Bagging & RF (NO SCALING)

Escalar los atributos numéricos puede dificultar la interpretación de los resultados. Por tanto, probaremos el modelo anterior (Modelo 3) sin escalar los atributos numéricos. Si nos dá un resultado parecido en validación (o la diferencia no es significativa), nos quedaremos con el modelo sin estandarizar las variables, ya que será más fácil de interpretar.

```
In [428]: # creamos los datasets para el tercer modelo de Bagging y RF
bagging_df3 = sales_simplified_df.copy(deep=True)
bagging_df3_test = test_data_preprocessed.copy(deep=True)

# lista contiene las variables que se consideran más importantes o inf
# para predecir el precio de la vivienda en Barcelona, según los diferentes predictores
good_predictors = ["price", # target variable
                    "sq_meters_built", "longitude",
                    "dist_city_center", "latitude", "bathrooms_per_100_rooms",
                    "rooms_per_100_sqm", "neighborhood_rent_index_cluster",
                    "garage", "garden", "pool", "terrace", "property_type",
                    "property_age_cat", "elevator", "ac", "heating"]

# nos quedamos sólo con las columnas indicadas en la lista good_predictors
bagging_df3 = bagging_df3[good_predictors]
bagging_df3_test = bagging_df3_test[good_predictors]

# variables que queremos codificar usando ordinal encoder
oe_cols = oe_cols = ['property_age_cat', 'neighborhood_rent_index_cluster']
# Define the order of the categories for each column
property_age_cat_order = ['Unknown', '0 - 40', '40 - 70', '70 - 120',
                           '120 - 2337', '2337 +']
neighborhood_rent_index_cluster_order = ['Unknown', '0 - 2337', '2337 +']
# Create an instance of the ordinal encoder
encoder = OrdinalEncoder(categories=[property_age_cat_order,
                                      neighborhood_rent_index_cluster_order])

for col in oe_cols:
    # Create a new column name for the encoded data
    new_col = col + '_encoded'
    # Create an instance of the ordinal encoder
    encoder = OrdinalEncoder(categories=[eval(col+'_order')])
    # Fit and transform the selected columns
    bagging_df3[new_col] = encoder.fit_transform(bagging_df3[[col]])
    bagging_df3_test[new_col] = encoder.fit_transform(bagging_df3_test)
    # replace 0 by -1 ('Unknown' by -1)
    bagging_df3[new_col] = bagging_df3[new_col].replace(0, -1)
    bagging_df3_test[new_col] = bagging_df3_test[new_col].replace(0, -1)

# variables que queremos codificar usando OHE
ohe_cols = ["garage", "garden", "pool",
            "terrace", "property_type",
            "elevator", "ac", "heating"]

# aplicamos one - hot encoding al dataset
bagging_df3 = one_hot_encode(bagging_df3, ohe_cols)
bagging_df3_test = one_hot_encode(bagging_df3_test, ohe_cols)

# eliminamos variables categóricas no codificadas
no_encoded_cat_vars = bagging_df3.select_dtypes(include=['category'])
bagging_df3.drop(no_encoded_cat_vars, axis=1, inplace=True)
no_encoded_cat_vars_test = bagging_df3_test.select_dtypes(include=['category'])
bagging_df3_test.drop(no_encoded_cat_vars_test, axis=1, inplace=True)

##### Tratamos los outliers per NO escalamos los atributos numéricos
num_vars = ["price", "sq_meters_built", "longitude", "dist_city_center"]
```

```
# define the preprocessing pipeline
preprocessing_pipeline = Pipeline([
    ('outlier_removal', OutlierRemover(cols=num_vars)),
    #('scaler', ColumnTransformer([('num_scaler', MinMaxScaler(), num_
                                # remainder='passthrough'))
])

# Apply the preprocessing pipeline to the training and testing data
# Fit and transform the training data
bagging_df3 = pd.DataFrame(preprocessing_pipeline.fit_transform(baggin
bagging_df3_test = pd.DataFrame(preprocessing_pipeline.fit_transform(b

# exploración rápida del df
print("TRAIN: ")
explore_data(bagging_df3)
print("\n\n")
print("TEST: ")
explore_data(bagging_df3_test)
TRAIN:
The dataset includes 3914 instances (rows) and 34 variables (column
s).
```

`sparse` was renamed to `sparse\_output` in version 1.2 and will be removed in 1.4. `sparse\_output` is ignored unless you leave `sparse` to its default value.

`sparse` was renamed to `sparse\_output` in version 1.2 and will be removed in 1.4. `sparse\_output` is ignored unless you leave `sparse` to its default value.

	price	sq_meters_built	longitude	dist_city_center	latitude	bathrooms_per_100_sqm	roo
1	90000	50	2.171110	5.189265	41.434545	2.000000	
2	145000	53	2.175309	6.313668	41.444518	1.886792	
3	675000	120	2.153368	1.462060	41.392209	1.666667	
4	210000	60	2.162216	2.006147	41.412285	1.440075	

```
In [512]: # create an instance of PipelineRunner
pipeline_runner = PipelineRunner()
# call the run_pipeline() method, passing in the dataframes and target
results_4 = pipeline_runner.run_pipeline(df_train=bagging_df3,
                                         df_test=bagging_df3_test,
                                         target=TARGET_VARIABLE)
```

```
Model: BaggingRegressor
Best model parameters: {'model__n_estimators': 200}
Best cross-validation R^2 score: 0.8699
Train R^2 score: 0.9826
Test R^2 score: 0.8562

Model: RandomForestRegressor
Best model parameters: {'model__max_depth': 9, 'model__min_samples_split': 2, 'model__n_estimators': 100}
Best cross-validation R^2 score: 0.8615
Train R^2 score: 0.9473
Test R^2 score: 0.8466

Best model: BaggingRegressor
Best model parameters: {'model__n_estimators': 200}
Best cross-validation R^2 score: 0.8699
Train R^2 score: 0.9826
Test R^2 score: 0.8562
```

```
In [513]: results_4
```

```
Out[513]: [{"name": "BaggingRegressor",
  "best_params": {"model__n_estimators": 200},
  "best_score": 0.8699084485798494,
  "train_r2_score": 0.9826165282700704,
  "test_r2_score": 0.8562174003334514},
 {"name": "RandomForestRegressor",
  "best_params": {"model__max_depth": 9,
  "model__min_samples_split": 2,
  "model__n_estimators": 100},
  "best_score": 0.8614809042234631,
  "train_r2_score": 0.9472748317785578,
  "test_r2_score": 0.8466012770286642}]
```

```
In [658]: # Loop through each model in results
for model in results_4:
    model_name = model['name']+ '_4'
    train_r2_score = model['train_r2_score']
    test_r2_score = model['test_r2_score']

    # Append the results to the DataFrame
    models_results_df = pd.concat([models_results_df, pd.DataFrame({'r':model_name, 'tr':train_r2_score, 'te':test_r2_score})])

# Display the DataFrame
display(models_results_df)
```

	regressor	train_r2_score	test_r2_score
0	BaggingRegressor_1	0.976685	0.739290
1	RandomForestRegressor_1	0.939248	0.723103
2	BaggingRegressor_2	0.970111	0.690796
3	RandomForestRegressor_2	0.924524	0.674443
4	BaggingRegressor_3	0.982274	0.852508
5	RandomForestRegressor_3	0.946943	0.845112
6	BaggingRegressor_4	0.982617	0.856217
7	RandomForestRegressor_4	0.947275	0.846601

- Observamos que sin escalar los atributos numéricos, los resultados son muy similares a cuando los hemos escalado (entre 0 y 1), incluso ligeramente superiores en validación para los dos modelos.
- Por tanto, de momento, este ha sido nuestro mejor modelo: BaggingRegressor con test\_r2\_score = 0.856.

Podemos concluir que el modelo 3.1, que utilizó la selección de variables más importantes y también incluyó el tratamiento de outliers, pero no el escalado de variables numéricas, tuvo el mejor rendimiento en términos de R2, tanto en training como en test. En este caso, la selección de variables importantes y el tratamiento de outliers han sido factores importantes para mejorar la capacidad predictiva de los modelos de BaggingRegressor y RandomForestRegressor para predecir el precio de las viviendas en Barcelona.

## 1.21.2 Boosting Models: CatBoost

Haremos 2 aproximaciones:

- Modelo 1: CatBoostRegressor con todas las variables del dataset sin seleccionar variables importantes, sin tratar outliers y sin escalar las variables numéricas.
- Modelo 2: CatBoostRegressor seleccionando las variables más importantes (mediante

técnicas como PCA, Análisis de Correlación, Información Mutua, ANOVA y ExtraTreesRegressor), tratando outliers (método IQR) y escalando las variables numéricas (MinMaxScaler). El dataset va a ser el mismo que el dataset del modelo 3 de Bagging y Random Forest, pero sin codificar las variables categóricas.

```
In [163]: def generate_train_test_sets(df_train: pd.DataFrame,
                                    df_test: pd.DataFrame,
                                    target: str) -> Tuple[pd.DataFrame, pd.Se
.....
    Generate training and test sets.

    Args:
        df_train (pd.DataFrame): Training dataset.
        df_test (pd.DataFrame): Test dataset.
        target (str): Target variable name.

    Returns:
        X_train (pd.DataFrame): Training features.
        y_train (pd.Series): Training target variable.
        X_test (pd.DataFrame): Test features.
        y_test (pd.Series): Test target variable.
.....
    X_train = df_train.drop(columns=[target])
    y_train = df_train[target]

    X_test = df_test.drop(columns=[target])
    y_test = df_test[target]

    return X_train, y_train, X_test, y_test
```

#### 1.21.2.1 Modelo 1

- Dataset con mínimo preprocessado.
- Solo eliminamos las variables que seguro que no son predictoras (id, currency, etc.)
- Convertimos las variables que realmente sean categóricas a 'category' y los valores nulos de las categóricas a 'Unknown'.

Cargamos los datasets de train y test originales, sin procesar:

```
In [18]: # cargamos el dataset de train original
train_data_original = pd.read_csv('train_data.csv', delimiter = ',')
# convertimos object a category
train_data_original = convert_object_to_category(train_data_original)
# cargamos el dataset de test original
test_data_original = pd.read_csv('test_data.csv', delimiter=',')
# convertimos object a category
test_data_original = convert_object_to_category(test_data_original)

# exploración rápida del df
print("TRAIN: ")
explore_data(train_data_original)
print("\n\n")
print("TEST: ")
explore_data(test_data_original)
```

TRAIN:

The dataset includes 4677 instances (rows) and 33 variables (columns).

	<b>id</b>	<b>price</b>	<b>currency</b>	<b>latitude</b>	<b>longitude</b>	<b>sq_meters</b>	<b>sq_meters_built</b>	<b>rooms</b>	<b>bathrooms</b>
<b>0</b>	85122800	2290000	€	41.409071	2.099850	383.0		532	5
<b>1</b>	95206437	90000	€	41.434545	2.171110	42.0		50	1
<b>2</b>	94361131	145000	€	41.444518	2.175309	NaN		53	3
<b>3</b>	95156089	675000	€	41.392209	2.153368	93.0		120	4

Definimos una clase que nos va a hacer el mínimo preprocesado que hemos comentado anteriormente:

```
In [8]: class CatBoostDataPreprocessor:
    target=TARGET_VARIABLE
    not_useful_vars = ['id',
                        'doorman',
                        'quality',
                        'city',
                        'currency',
                        'furniture',
                        'closest_station',
                        'created_at',
                        'last_seen']
    ]
    def test_data_preprocessor(self, file_path):
        self.load_data(file_path)
        #self.initial_feature_selection()
        self.get_binary_cols()
        self.replace_binary_cols_nulls_by_zero()
        self.convert_binary_to_category()
        #self.convert_nan_to_unknown()

    def load_data(self, file_path):
        self.df = pd.read_csv(file_path, delimiter=',').copy()

    def initial_feature_selection(self):
        self.df.drop(self.not_useful_vars, axis=1, inplace=True)

    def get_binary_cols(self) -> list[str]:
        """
        Returns a list with the names of the binary variables in a Pandas DataFrame.

        Parameters:
        None

        Returns:
        A list with the names of the binary variables in the DataFrame.
        """
        binary_cols = [col for col in self.df if (len(self.df[col].value_counts()) == 2)]
        return binary_cols

    def replace_binary_cols_nulls_by_zero(self) -> None:
        """
        Replaces missing values in binary columns of a pandas DataFrame.

        Parameters:
        None

        Returns:
        None
        """
        binary_cols = self.get_binary_cols()
        if binary_cols:
```

```
        self.df.loc[:, binary_cols] = self.df.loc[:, binary_cols].  
  
    def convert_binary_to_category(self) -> None:  
        # get binary columns  
        binary_cols = self.get_binary_cols()  
  
        # check if binary_cols is not empty  
        if binary_cols:  
            # convert binary columns to category  
            for col in binary_cols:  
                # check if column has non-missing values  
                if self.df[col].notna().all():  
                    # convert to int64 to remove decimals: 0 and 1  
                    self.df[col] = self.df[col].astype("int64")  
                    # convert to category: '0' and '1'  
                    self.df[col] = self.df[col].astype("category")  
                else:  
                    print("No binary columns found.")  
  
    def convert_nan_to_unknown(self) -> None:  
        cat_features = self.df.select_dtypes(include=['object', 'categ  
        self.df[cat_features] = self.df[cat_features].fillna("unknown")
```

```
In [119]: ##### creamos los datasets para el primer modelo de catboost #####  
# create an instance of the CatBoostDataPreprocessor class  
data_preprocessor = CatBoostDataPreprocessor()  
# train  
data_preprocessor.test_data_preprocessor('train_data.csv')  
boosting_df1_train = data_preprocessor.df  
# test  
data_preprocessor.test_data_preprocessor('test_data.csv')  
boosting_df1_test = data_preprocessor.df
```

```
In [128]: # Define categorical features
cat_features = boosting_df1_train.select_dtypes(include=['object', 'ca'])

# Generate training and test sets
X_train, y_train, X_test, y_test = generate_train_test_sets(boosting_d
                                                               boosting_d
                                                               TARGET_VAR

d_train_cat = catboost.Pool(X_train, y_train, cat_features = cat_featu
d_test_cat = catboost.Pool(X_test, y_test, cat_features = cat_features

# Gridsearch nos ayudará a generar la mejor combinación:
model = catboost.CatBoostRegressor()

# Define the hyperparameters to tune
param_grid = {'iterations': [1000, 6000],
              'learning_rate': [0.01, 0.015, 0.05, 0.1],
              'depth': [4, 6, 8, 10],
              'l2_leaf_reg': [1, 3, 5, 9]}

grid_search_result = model.grid_search(param_grid,
                                         d_train_cat,
                                         plot=True) # cv = 3 by default
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

```
0:      learn: 609853.4296681    test: 597293.2619657    best: 597293.
2619657 (0)      total: 4.94ms  remaining: 4.93s
1:      learn: 605123.5039487    test: 592781.9622227    best: 592781.
9622227 (1)      total: 11ms   remaining: 5.5s
2:      learn: 600680.8055244    test: 588406.1136420    best: 588406.
1136420 (2)      total: 16.3ms  remaining: 5.42s
3:      learn: 596195.8328184    test: 583874.5390213    best: 583874.
5390213 (3)      total: 20.3ms  remaining: 5.04s
4:      learn: 591831.0440212    test: 579282.6410521    best: 579282.
6410521 (4)      total: 26.5ms  remaining: 5.26s
5:      learn: 587480.3543110    test: 574829.7899005    best: 574829.
7899005 (5)      total: 33.1ms  remaining: 5.48s
6:      learn: 583274.5564414    test: 570662.4759041    best: 570662.
4759041 (6)      total: 37.2ms  remaining: 5.27s
```

```
In [129]: best_params = grid_search_result['params']
print("Best hyperparameters found:\n", best_params)
```

```
Best hyperparameters found:
{'depth': 6, 'l2_leaf_reg': 5, 'iterations': 6000, 'learning_rate': 0.015}
```

```
In [130]: params_1 = {"iterations": 6000,
                    "depth": 6,
                    "loss_function": "RMSE",
                    "learning_rate": 0.015,
                    "verbose": False,
                    "subsample": 0.7,
                    "l2_leaf_reg": 5,
                    "eval_metric": "R2"}

scores = catboost.cv(d_train_cat,
                      params_1,
                      fold_count=CROSS_VALIDATION_FOLDS, # 10-fold CV
                      plot=True)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

Training on fold [0/10]

```
bestTest = 0.7094201837
bestIteration = 5243
```

Training on fold [1/10]

```
bestTest = 0.8922555613
bestIteration = 2518
```

Training on fold [2/10]

```
bestTest = 0.8941688943
bestIteration = 5823
```

Training on fold [3/10]

```
bestTest = 0.802021692
bestIteration = 3102
```

Training on fold [4/10]

```
bestTest = 0.9087740433
bestIteration = 5081
```

Training on fold [5/10]

```
bestTest = 0.9024225511
bestIteration = 5995
```

Training on fold [6/10]

```
bestTest = 0.7003341455
bestIteration = 5973
```

Training on fold [7/10]

```
bestTest = 0.9105764178  
bestIteration = 5901
```

Training on fold [9/10]

```
bestTest = 0.8932949294  
bestIteration = 5998
```

```
In [131]: # Modelo final
# estructura más optimizada
model_cat_1 = catboost.CatBoostRegressor(iterations = params_1['iterations'],
                                           learning_rate = params_1['learning_rate'],
                                           loss_function = params_1['loss_function'],
                                           random_seed = 2,
                                           depth = params_1['depth'],
                                           subsample=0.7 # controls the
                                                       # nstructuring C
                                           )

model_cat_1.fit(d_train_cat,
                 eval_set = d_test_cat,
                 verbose_eval=500, # the model will print out training
                 plot = True)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

```
0:      learn: 491919.7793372    test: 641313.0079469    best: 641313.
0079469 (0)      total: 5.29ms   remaining: 31.7s
500:     learn: 137877.4411986    test: 386497.7729555    best: 386497.
7729555 (500)    total: 1.63s   remaining: 17.9s
1000:    learn: 103610.8206636    test: 371269.7459556    best: 371258.
0177998 (996)    total: 3.13s   remaining: 15.6s
1500:    learn: 86226.6226437    test: 368107.5981796    best: 367984.
8569473 (1493)    total: 4.64s   remaining: 13.9s
2000:    learn: 74889.3257865    test: 366778.6782865    best: 366723.
9393334 (1974)    total: 6.36s   remaining: 12.7s
2500:    learn: 66023.9734343    test: 365736.9932047    best: 365724.
8357294 (2487)    total: 8.16s   remaining: 11.4s
3000:    learn: 59539.5832038    test: 364330.2406760    best: 364314.
9731143 (2999)    total: 9.72s   remaining: 9.72s
3500:    learn: 54153.7444823    test: 363706.8265439    best: 363704.
8799717 (3498)    total: 11.3s   remaining: 8.07s
4000:    learn: 49785.0305848    test: 363286.8435240    best: 363226.
8704452 (3904)    total: 12.9s   remaining: 6.45s
4500:    learn: 46226.8372480    test: 363030.9097705    best: 363024.
0554577 (4437)    total: 14.5s   remaining: 4.84s
5000:    learn: 43227.9621395    test: 362809.3261104    best: 362808.
1813925 (4999)    total: 16.5s   remaining: 3.29s
5500:    learn: 40892.7566187    test: 362625.6310278    best: 362625.
6310278 (5500)    total: 18.1s   remaining: 1.64s
5999:    learn: 38648.0230044    test: 362417.2838437    best: 362406.
0019464 (5959)    total: 19.7s   remaining: 0us

bestTest = 362406.0019
bestIteration = 5959

Shrink model to first 5960 iterations.
```

```
Out[131]: <catboost.core.CatBoostRegressor at 0x13d8db4f0>
```

```
In [132]: # Vamos a ver cómo de bueno es este modelo:
y_pred_train = model_cat_1.predict(X_train)
y_pred_test = model_cat_1.predict(X_test)

# error rate
print("R2 Training: " + str(metrics.r2_score(y_train, y_pred_train)))
print("R2 Test: " + str(metrics.r2_score(y_test, y_pred_test)))
```

R2 Training: 0.9907346903456915  
R2 Test: 0.6832622271562178

The R2 score for the training set is 0.9907, which indicates that the model fits the training data very well. However, the R2 score for the test set is 0.6833, which is lower than the training score and suggests that the model may be overfitting to the training data.

To improve the performance of the model on the test set, you can try adjusting the hyperparameters or regularizing the model. For example, you could try reducing the depth of the tree or increasing the regularization strength (e.g., by increasing the value of l2\_leaf\_reg). You could also try adding more training data or using a more complex model architecture.

It's also worth noting that the choice of evaluation metric can affect the interpretation of the model's performance. The R2 score is a commonly used metric for regression problems, but it has some limitations and may not always capture the full picture of how well the model is performing. You may want to try using other metrics, such as mean squared error (MSE) or mean absolute error (MAE), to get a more complete understanding of the model's performance.

```
In [661]: # Add CatBoostRegressor1 data
models_results_df.loc[8] = ['CatBoostRegressor_1', 0.990734, 0.683262]
display(models_results_df)
```

	regressor	train_r2_score	test_r2_score
0	BaggingRegressor_1	0.976685	0.739290
1	RandomForestRegressor_1	0.939248	0.723103
2	BaggingRegressor_2	0.970111	0.690796
3	RandomForestRegressor_2	0.924524	0.674443
4	BaggingRegressor_3	0.982274	0.852508
5	RandomForestRegressor_3	0.946943	0.845112
6	BaggingRegressor_4	0.982617	0.856217
7	RandomForestRegressor_4	0.947275	0.846601
8	CatBoostRegressor_1	0.990734	0.683262

### 1.21.2.2 Modelo 2

En el modelo anterior (Modelo 1) no se ha hecho un preprocesado profundo del dataset: no

se han tratado los missing values, no se han escalado las variables, no se han quitado los outliers y no se ha hecho una selección de variables.

Para este modelo (Modelo 2), el dataset se han seleccionado las variables más importantes (mediante técnicas como PCA, Análisis de Correlación, Información Mutua, ANOVA y ExtraTreesRegressor), se han tratado outliers (método IQR) y escalado las variables numéricas (MinMaxScaler). Es decir, el dataset va a ser el mismo que el dataset del modelo 3 de Bagging y Random Forest, pero sin codificar las variables categóricas.

Creamos los datasets para el segundo modelo de catboost:

- Tratamos outliers y escalamos los atributos numéricos.

```
In [9]: ##### creamos los datasets para el segundo modelo de catboost #####
# create an instance of the CatBoostDataPreprocessor class
data_preprocessor = CatBoostDataPreprocessor()

# train
data_preprocessor.test_data_preprocessor('train_data_preprocessed.csv')
boosting_df2_train = data_preprocessor.df
boosting_df2_train = convert_object_to_category(boosting_df2_train) #

# test
data_preprocessor.test_data_preprocessor('test_data_preprocessed.csv')
boosting_df2_test = data_preprocessor.df
boosting_df2_test = convert_object_to_category(boosting_df2_test) # co

# Lista contiene las variables que se consideran más importantes o inf
# para predecir el precio de la vivienda en Barcelona, según los diferen
good_predictors = ["price", # target variable
                    "sq_meters_built", "longitude",
                    "dist_city_center", "latitude", "bathrooms_per_100_
                    "rooms_per_100_sqm", "neighborhood_rent_index_clust
                    "garage", "garden", "pool", "terrace", "property_ty
                    "property_age_cat", "elevator", "ac", "heating"]

# nos quedamos sólo con las columnas indicadas en la lista good_predicto
boosting_df2_train = boosting_df2_train[good_predictors]
boosting_df2_test = boosting_df2_test[good_predictors]

##### Tratamos los outliers y escalamos los atributos numérico
num_vars = ["price", "sq_meters_built", "longitude", "dist_city_center
            "latitude", "bathrooms_per_100_sqm", "rooms_per_100_sqm"]

cat_features = ["neighborhood_rent_index_cluster",
                "garage", "garden", "pool", "terrace", "property_type"
                "property_age_cat", "elevator", "ac", "heating"]

# Define the column transformer to apply the numeric pipeline to numer
# This ensures that the categorical variables are not processed by the
preprocessor = ColumnTransformer(
    transformers=[
        ('num', num_pipeline, num_vars)
    ])

# Define the pipeline steps for numeric variables
num_pipeline = Pipeline([
    ('outlier_remover', OutlierRemover(cols=num_vars)),
    ('scaler', MinMaxScalerTransformer())
])

# Define the column transformer to apply the numeric pipeline to numer
preprocessing_pipeline = ColumnTransformer(transformers=[
```

```
( 'num_pipeline', num_pipeline, num_vars)], remainder='passthrough'

# Apply the column transformer to the training and test data
boosting_df2_train_processed = preprocessor.fit_transform(boosting_df2)
boosting_df2_test_processed = preprocessor.transform(boosting_df2_test)

# Convert the processed data to a DataFrame
boosting_df2_train_processed = pd.DataFrame(boosting_df2_train_processed)
boosting_df2_test_processed = pd.DataFrame(boosting_df2_test_processed)

# Merge the processed numeric variables with the categorical variables
boosting_df2_train_processed = pd.merge(boosting_df2_train_processed,
boosting_df2_test_processed = pd.merge(boosting_df2_test_processed, bo

#-----
# exploración rápida del df
print("TRAIN: ")
explore_data(boosting_df2_train_processed)
print("\n\n")
print("TEST: ")
explore_data(boosting_df2_test_processed)
TRAIN:
The dataset includes 3914 instances (rows) and 17 variables (columns).
s).
```

	price	sq_meters_built	longitude	dist_city_center	latitude	bathrooms_per_100_sqm	ro
0	0.083558	0.149123	0.539966	0.598009	0.721144		0.555140
1	0.157682	0.175439	0.578853	0.729485	0.811563		0.506260
2	0.871968	0.763158	0.375652	0.162188	0.337326		0.411215
3	0.392183	0.315789	0.457874	0.331045	0.529310		0.317351
4	0.080863	0.271930	0.391947	0.470643	0.616941		0.366238

<class 'pandas.core.frame.DataFrame'>  
Tnt64Tndex: 3914 entries. 0 to 3913

## Modelo 2

Una vez hemos seleccionado las variables predictoras más importantes, hemos tratado los outliers y hemos escalado los atributos numéricos, vemos cómo se comporta el modelo de catboost:

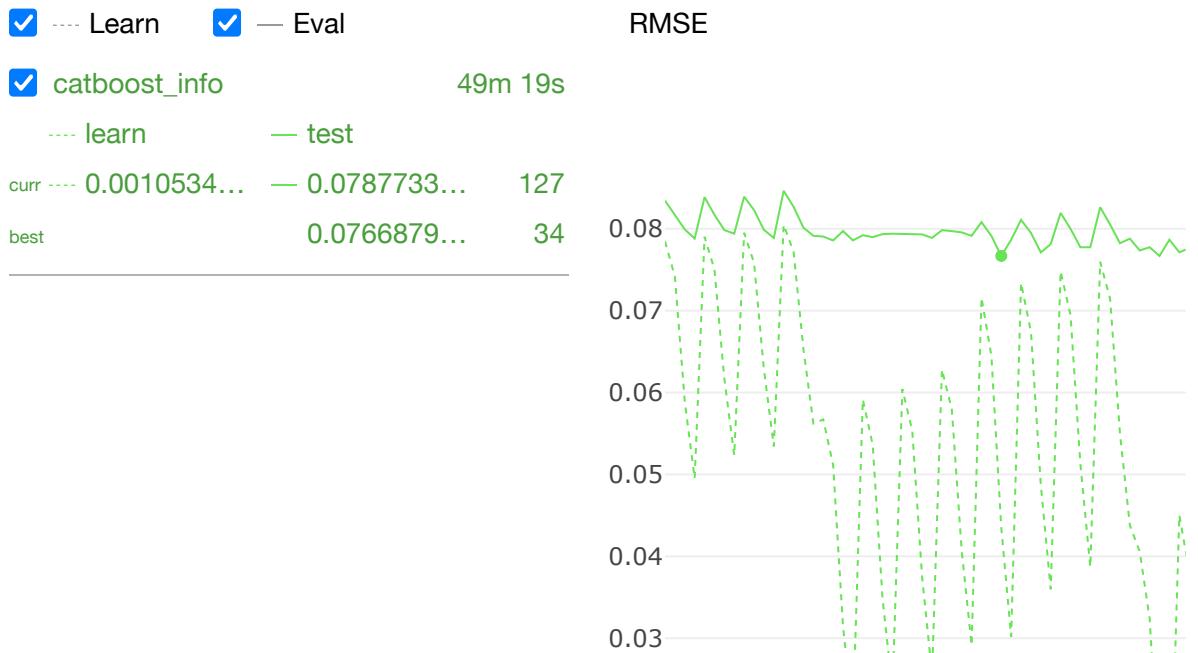
```
In [13]: # Generate training and test sets
X_train, y_train, X_test, y_test = generate_train_test_sets(boosting_d
                                                               boosting_d
                                                               TARGET_VAR

d_train_cat_2 = catboost.Pool(X_train, y_train, cat_features = cat_fea
d_test_cat_2 = catboost.Pool(X_test, y_test, cat_features = cat_featur

# Gridsearch nos ayudará a generar la mejor combinación:
model_2 = catboost.CatBoostRegressor()

# Define the hyperparameters to tune
param_grid = {'iterations': [1000, 6000],
              'learning_rate': [0.01, 0.015, 0.05, 0.1],
              'depth': [4, 6, 8, 10],
              'l2_leaf_reg': [1, 3, 5, 9]}

grid_search_result = model_2.grid_search(param_grid,
                                         d_train_cat_2, # cv = 3 by defa
                                         verbose = False,
                                         plot=True)
```



```
In [14]: best_params_2 = grid_search_result['params']
        print("Best hyperparameters found:\n", best_params_2)
```

```
Best hyperparameters found:  
{'depth': 6, 'l2_leaf_reg': 1, 'iterations': 1000, 'learning_rate':  
0.05}
```

```
In [15]: params_2 = {"iterations": 1000,
                  "depth": 6,
                  "loss_function": "RMSE",
                  "learning_rate": 0.05,
                  "verbose": False,
                  "subsample": 0.7,
                  "l2_leaf_reg": 1,
                  "eval_metric": "R2"}

scores = catboost.cv(d_train_cat_2,
                      params_2,
                      fold_count=CROSS_VALIDATION_FOLDS, # 10-fold CV
                      plot=True)
```



Training on fold [0/10]

```
bestTest = 0.8602784867
bestIteration = 978
```

Training on fold [1/10]

```
bestTest = 0.8311744542
bestIteration = 848
```

Training on fold [2/10]

```
bestTest = 0.8172953639
bestIteration = 535

Training on fold [3/10]

bestTest = 0.8398898689
bestIteration = 847

Training on fold [4/10]

bestTest = 0.8278121284
bestIteration = 997

Training on fold [5/10]

bestTest = 0.8654670856
bestIteration = 832

Training on fold [6/10]

bestTest = 0.8430954008
bestIteration = 992

Training on fold [7/10]

bestTest = 0.8178906495
bestIteration = 824

Training on fold [8/10]

bestTest = 0.8459529001
bestIteration = 923

Training on fold [9/10]

bestTest = 0.8541878361
bestIteration = 552
```

```
In [16]: # Modelo final
# estructura más optimizada
model_cat_2 = catboost.CatBoostRegressor(iterations = params_2['iterations'],
                                           learning_rate = params_2['learning_rate'],
                                           loss_function = params_2['loss_function'],
                                           random_seed = 2,
                                           depth = params_2['depth'],
                                           subsample=0.7 # controls the
                                                       # nstructuring C
                                           )

model_cat_2.fit(d_train_cat_2,
                 eval_set = d_test_cat_2,
                 verbose_eval=500, # the model will print out training
                 plot = True)
```



```
0:      learn: 0.1894636      test: 0.2134708 best: 0.2134708 (0)
       total: 4.84ms   remaining: 4.83s
500:     learn: 0.0623029      test: 0.0886654 best: 0.0886654 (500)
       total: 1.34s   remaining: 1.33s
999:     learn: 0.0513777      test: 0.0884745 best: 0.0882315 (637)
       total: 4.19s   remaining: 0us

bestTest = 0.08823148114
bestIteration = 637
```

```
    shrink model to fit next 620 iterations
```

Out[16]: <catboost.core.CatBoostRegressor at 0x14f72fc0>

In [17]:

```
# Vamos a ver cómo de bueno es este modelo:  
y_pred_train_2 = model_cat_2.predict(X_train)  
y_pred_test_2 = model_cat_2.predict(X_test)  
  
# error rate  
print("R2 Training: " + str(metrics.r2_score(y_train, y_pred_train_2)))  
print("R2 Test: " + str(metrics.r2_score(y_test, y_pred_test_2)))
```

R2 Training: 0.9044574141849953

R2 Test: 0.8379225284112017

In [662]:

```
# Add CatBoostRegressor2 data  
models_results_df.loc[9] = ['CatBoostRegressor2', 0.904457, 0.837922]  
display(models_results_df)
```

	regressor	train_r2_score	test_r2_score
0	BaggingRegressor_1	0.976685	0.739290
1	RandomForestRegressor_1	0.939248	0.723103
2	BaggingRegressor_2	0.970111	0.690796
3	RandomForestRegressor_2	0.924524	0.674443
4	BaggingRegressor_3	0.982274	0.852508
5	RandomForestRegressor_3	0.946943	0.845112
6	BaggingRegressor_4	0.982617	0.856217
7	RandomForestRegressor_4	0.947275	0.846601
8	CatBoostRegressor_1	0.990734	0.683262
9	CatBoostRegressor2	0.904457	0.837922

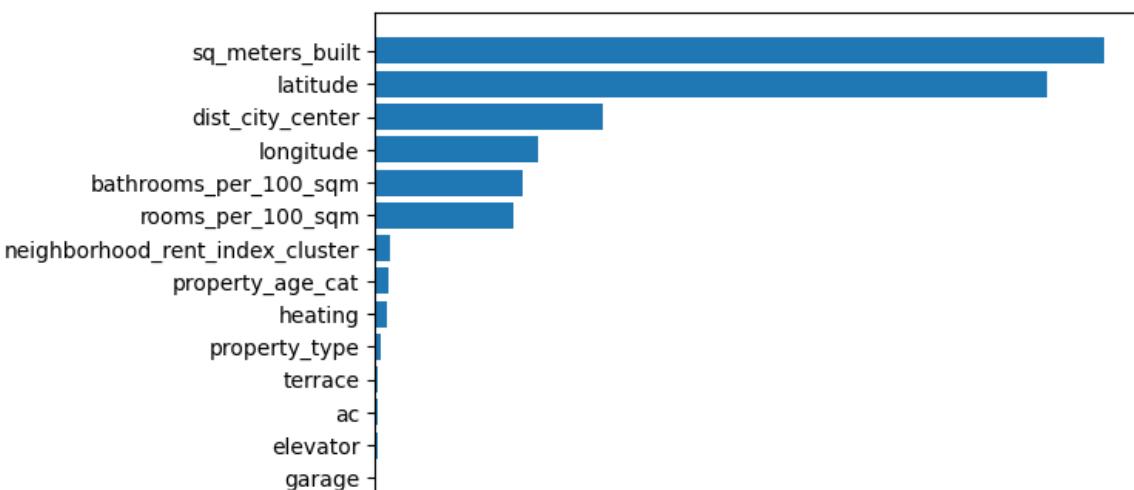
Ahora tenemos un modelo potente que nos da unas predicciones.

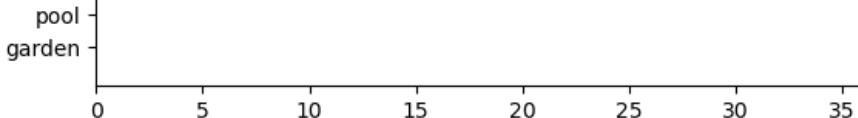
Pero debemos saber por qué nos dice, por ejemplo, que una casa cuesta 1000€ si no tiene garaje, está en un 1r piso y no tiene piscina?

En los Decision Trees, cada nodo se basa en el principio de entropía, y ahora lo que podemos decidir y ver es, en base a la importancia del nodo, cuáles son las variables más importantes.

```
In [31]: cat_importance = pd.DataFrame({"Features": model_cat_2.feature_names_,  
                                         "Importance": model_cat_2.feature_importances_}).sort_values('Importance', ascending=False)  
display(cat_importance)  
  
plt.rcParams()  
fig, ax = plt.subplots()  
  
ax.barh(y = 'Features',  
         width = 'Importance',  
         data = cat_importance)  
ax.invert_yaxis() # labels read top-to-bottom
```

	Features	Importance
0	sq_meters_built	34.176149
3	latitude	31.502604
2	dist_city_center	10.654744
1	longitude	7.647864
4	bathrooms_per_100_sqm	6.947486
5	rooms_per_100_sqm	6.513888
6	neighborhood_rent_index_cluster	0.684250
12	property_age_cat	0.627577
15	heating	0.522538
11	property_type	0.265222
10	terrace	0.142925
14	ac	0.141822
13	elevator	0.132893
7	garage	0.029843
9	pool	0.009766
8	garden	0.000429





de superficie construida (sq\_meters\_built) y la latitud de la propiedad. Estas dos características representan más del 65% de la importancia total.

La tercera característica más importante es la distancia al centro de la ciudad (dist\_city\_center), que representa el 10.65% de la importancia. La longitud, baños\_por\_100\_m<sup>2</sup> y habitaciones\_por\_100\_m<sup>2</sup> son las siguientes características más importantes.

El resto de atributos tienen una importancia muy baja: neighborhood\_rent\_index\_cluster, property\_age\_cat y heating son las únicas que representan más del 0.5% de la importancia de la característica. Las demás características tienen una importancia mínima.

De este análisis, podemos concluir que el tamaño y la ubicación de la propiedad son los factores más importantes a la hora de determinar el precio de una vivienda en Barcelona. Características como la distancia al centro de la ciudad, el número de baños y habitaciones por metro cuadrado, y la presencia de ciertas comodidades (garaje, piscina, jardín, ascensor, aire acondicionado) también tienen cierta importancia, pero en menor medida que el tamaño y la ubicación de la propiedad.

Es importante señalar que la importancia relativa de estos atributos puede variar en función del conjunto de datos y del modelo utilizado, por lo que siempre se recomienda realizar un análisis de la importancia de las características caso por caso para comprender mejor los factores que impulsan las predicciones del modelo.

#### 1.21.2.2.1 SHAP Values Analysis

##### SHAP Values Analysis para el segundo modelo de catboost

Creamos un objeto TreeExplainer en SHAP para el modelo.

Este objeto se utiliza para calcular los valores de Shapley de las predicciones del modelo y atribuir las contribuciones de cada característica a esas predicciones.

Utilizando este objeto TreeExplainer, podemos saber qué características son las más importantes en las predicciones del modelo, cómo interactúan entre sí y qué valores específicos de esas características conducen a predicciones más altas o más bajas. Esta información puede ser muy útil para comprender y explicar el comportamiento del modelo, así como para identificar posibles sesgos o incoherencias en el modelo.

```
In [45]: explainer_cat2 = shap.TreeExplainer(model_cat_2)  
explainer_cat2
```

```
Out[45]: <shap.explainers._tree.Tree at 0x15460c580>
```

```
In [46]: shap_values_cat2 = explainer_cat2.shap_values(X_train)
```

```
In [93]: # Primera instancia (primera vivienda)
i = 0
print("Attributes")
print("")
print(X_train.iloc[i,:])
print("")
print("Price")
print("")
print(y_train.iloc[i])
```

Attributes

sq_meters_built	0.149123
longitude	0.539966
dist_city_center	0.598009
latitude	0.721144
bathrooms_per_100_sqm	0.55514
rooms_per_100_sqm	0.28
neighborhood_rent_index_cluster	3371 – 5042
garage	1
garden	1
pool	1
terrace	1
property_type	chalet_independiente
property_age_cat	0 – 40
elevator	0
ac	1
heating	individual
Name: 0, dtype: object	

Price

0.08355795148247977

- Observamos la primera instancia (primera vivienda).
- Recordemos que los atributos numéricos están escalados a un rango fijo entre 0 y 1.
- Calculamos artesanalmente la inversa del escalado del precio para tener una mejor idea de que vale la casa y sus metros cuadrados construidos.

```
In [227]: ##### obtención de valores sin escalar #####
df_train = sales_simplified_df.copy(deep=True)
df_test = test_data_preprocessed.copy(deep=True)

# lista contiene las variables que se consideran más importantes o inf
# para predecir el precio de la vivienda en Barcelona, según los diferentes predictores
good_predictors = ["price", # target variable
                    "sq_meters_built", "longitude",
                    "dist_city_center", "latitude", "bathrooms_per_100_rooms",
                    "rooms_per_100_sqm", "neighborhood_rent_index_clust",
                    "garage", "garden", "pool", "terrace", "property_type",
                    "property_age_cat", "elevator", "ac", "heating"]

# nos quedamos sólo con las columnas indicadas en la lista good_predictors
df_train = df_train[good_predictors]
df_test = df_test[good_predictors]

##### Tratamos los outliers pero NO escalamos los atributos numéricos
num_vars = ["price", "sq_meters_built", "longitude", "dist_city_center"]

# define the preprocessing pipeline
preprocessing_pipeline = Pipeline([
    ('outlier_removal', OutlierRemover(cols=num_vars))
])

# Apply the preprocessing pipeline to the training and testing data
    # Fit and transform the training data
df_train = pd.DataFrame(preprocessing_pipeline.fit_transform(df_train))
df_test = pd.DataFrame(preprocessing_pipeline.fit_transform(df_test),

# Generate training and test sets
X_train_SHAP, y_train_SHAP, X_test_SHAP, y_test_SHAP = generate_train_and_test_sets()

##### PRECIO #####
# Calculamos el mínimo y el máximo del original
i = 0
price_max = y_train_SHAP[i:].max()
price_min = y_train_SHAP[i:].min()

# Inversa del escalado (entre 0 y 1)
original_price = y_pred_train_2 * (price_max - price_min) + price_min

# Print the price
print(f"Precio en €: {original_price[i]}")

##### sq_meters_built #####
# Calculamos el mínimo y el máximo del original
f_max = X_train_SHAP.iloc[:, 0].max()
f_min = X_train_SHAP.iloc[:, 0].min()

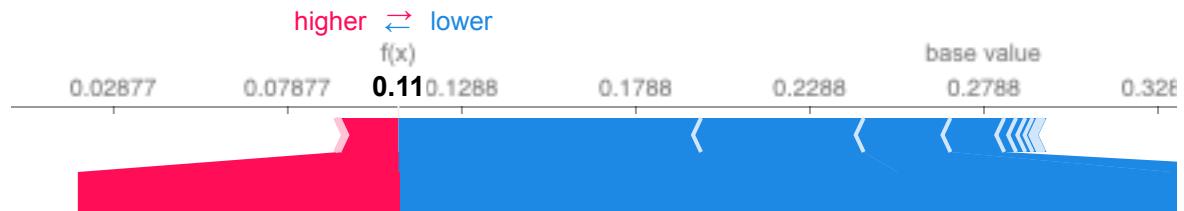
# Inversa del escalado (entre 0 y 1)
original_f = X_train_SHAP.iloc[:, 0] * (f_max - f_min) + f_min
```

```
# Print the price
print(f"Metros cuadrados construidos: {original_f}")
Precio en €: 110254.37992416498
Metros cuadrados construidos: 130.0
```

In [53]: `print("CatBoostRegressor (Model 2) Shap visualization for instance 0")  
shap.force_plot(explainer_cat2.expected_value, shap_values_cat2[0,:],`

CatBoostRegressor (Model 2) Shap visualization for instance 0

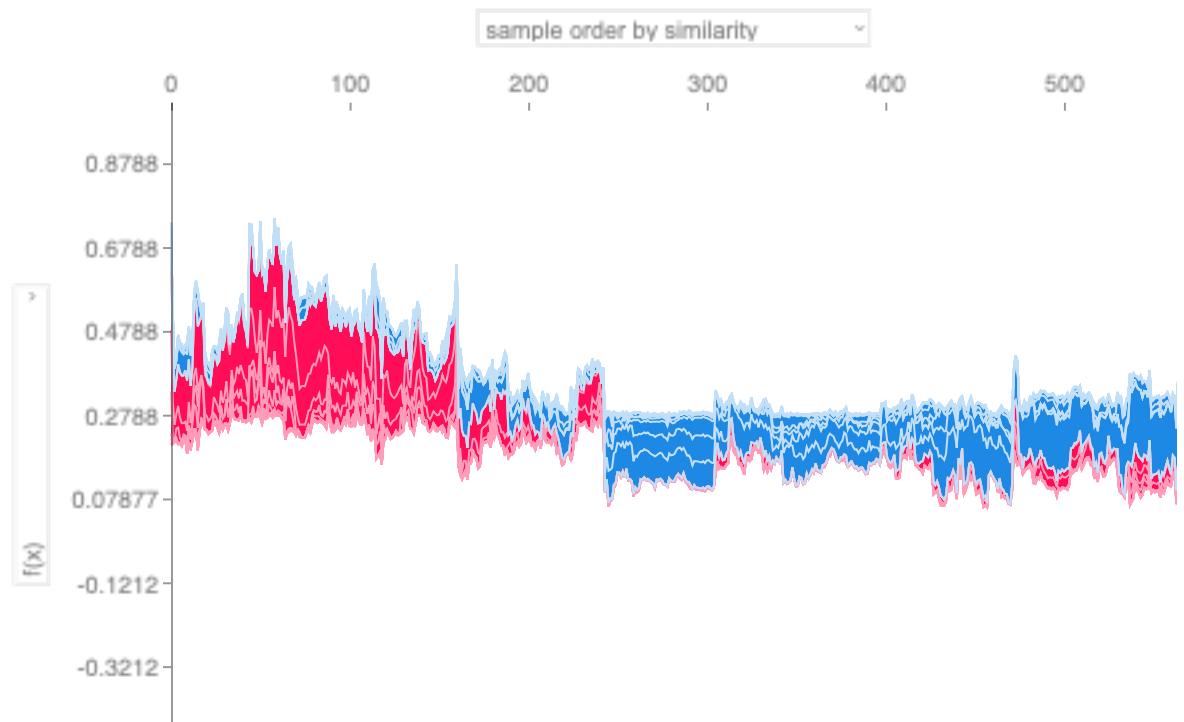
Out [53]:



- SHAP nos dice cuál es el valor medio del precio de las viviendas del dataset (base value): 0.2788 (valor escalado).
- La casa que hemos seleccionado cuesta 0.11 (valor escalado).
  - Azul: disminuye el precio.
  - Rojo: aumenta el precio.
- La que tiene más impacto es rooms\_per\_100\_sqm.

In [54]: `shap.force_plot(explainer_cat2.expected_value, shap_values_cat2[0:1000]`

Out [54]:

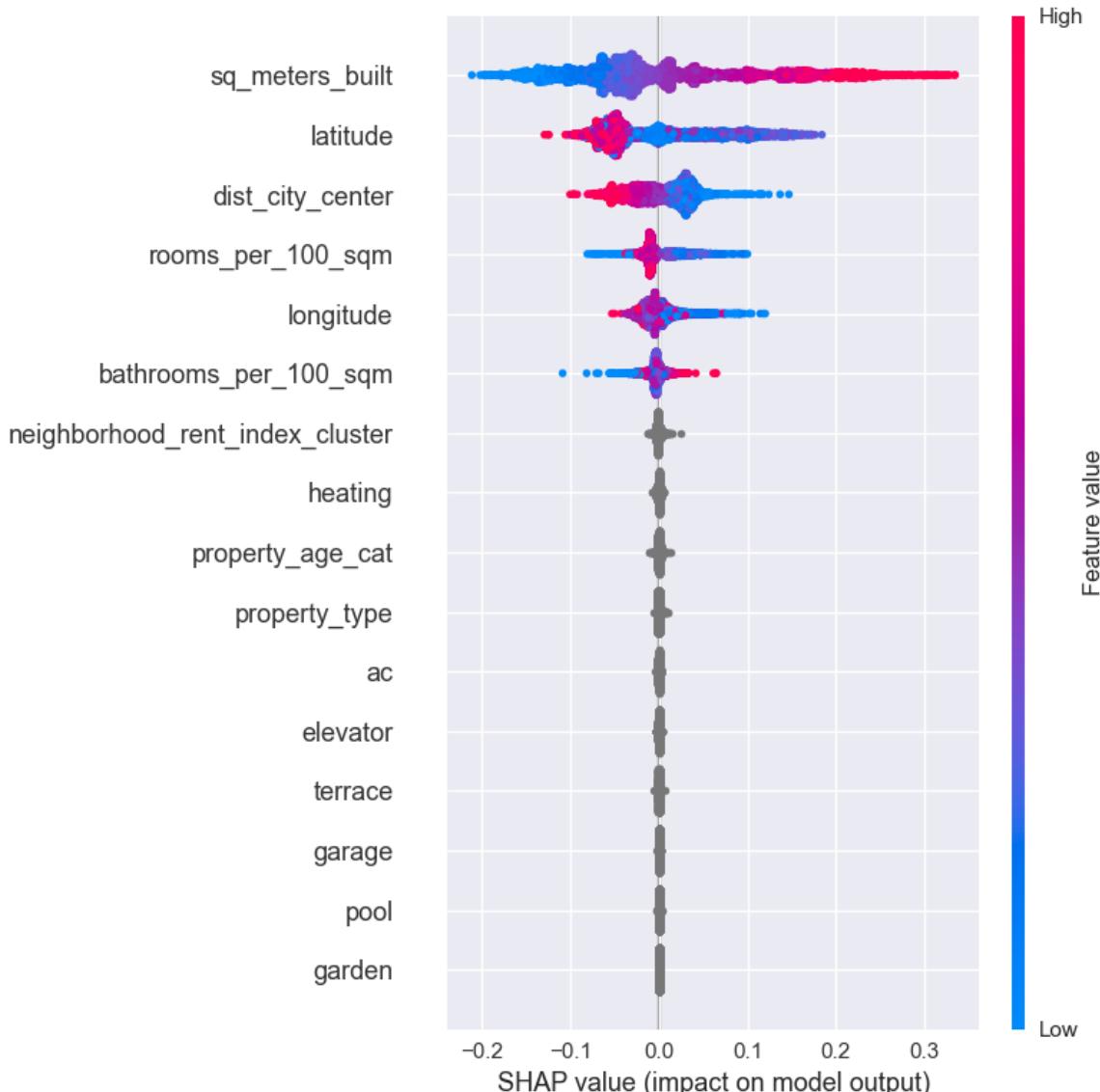


- La distancia al centro de la ciudad.
- Rojo: aumenta precio.

- Azul: baja precio.
- El límite está entorno a 0.4378 (valor escalado).
- A medida que nos alejamos del centro de la ciudad, el precio disminuye.

```
In [50]: shap.summary_plot(shap_values_cat2, X_train)
```

No data for colormapping provided via 'c'. Parameters 'vmin', 'vmax' will be ignored



### Importancia total de los SHAP values:

- Los metros cuadrados construidos es la más importante.
  - Nuestro modelo funciona bien para las viviendas que no son extremadamente caras. En el histograma ya vimos que la mayoría de viviendas estaban en la parte izquierda del gráfico, y había unas pocas muy caras (con una cantidad de metros cuadrados construidos muy grande).
  - En la parte azul tenemos más información.
- La distancia al centro de la ciudad nos dice que, como más cerca del centro, más cara

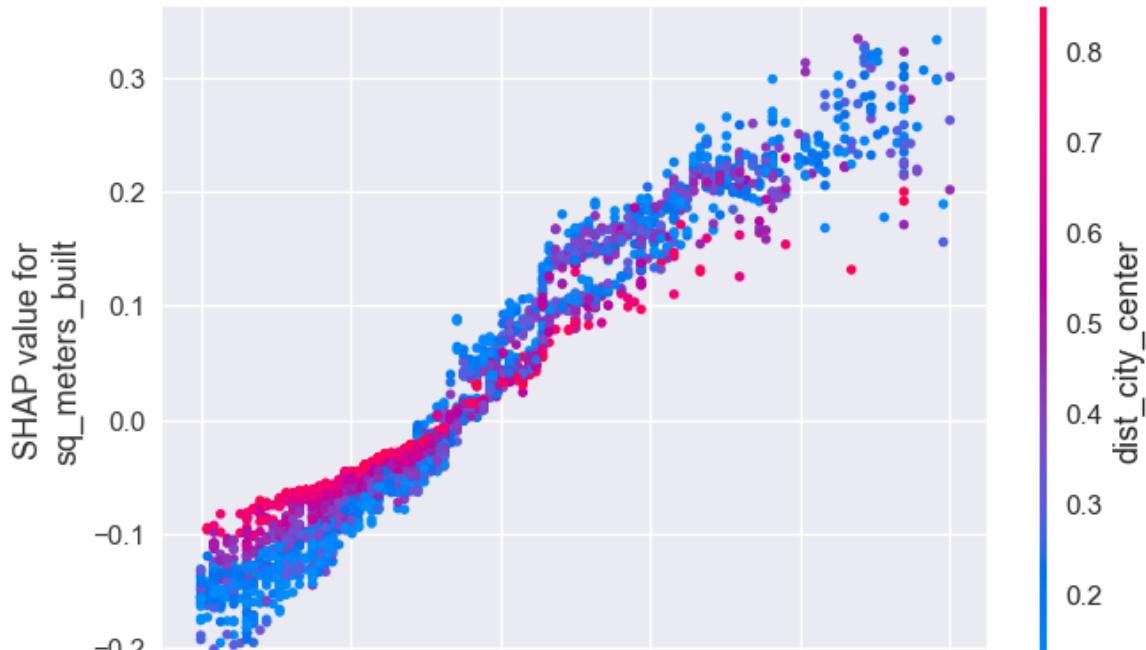
es la vivienda.

- Todo lo que está a la derecha del eje quiere decir que el precio aumenta, y a la izquierda significa que el precio disminuye.
- A la derecha están los puntos azules: los valores bajos de distancia al centro. Por tanto, si vives en el centro, la vivienda será más cara que si vives lejos del centro. Hay mucha concentración de vivienda cerca del centro de la ciudad de BCN.
- En gris son las variables categóricas. Por tanto, este gráfico no nos aporta mucho valor para variables categóricas.

Los siguientes gráficos están ordenados por importancia de variables:

```
In [55]: top_inds = np.argsort(-np.sum(np.abs(shap_values_cat2), 0))

for i in range(0, len(top_inds)):
    shap.dependence_plot(top_inds[i], shap_values_cat2, X_train)
```



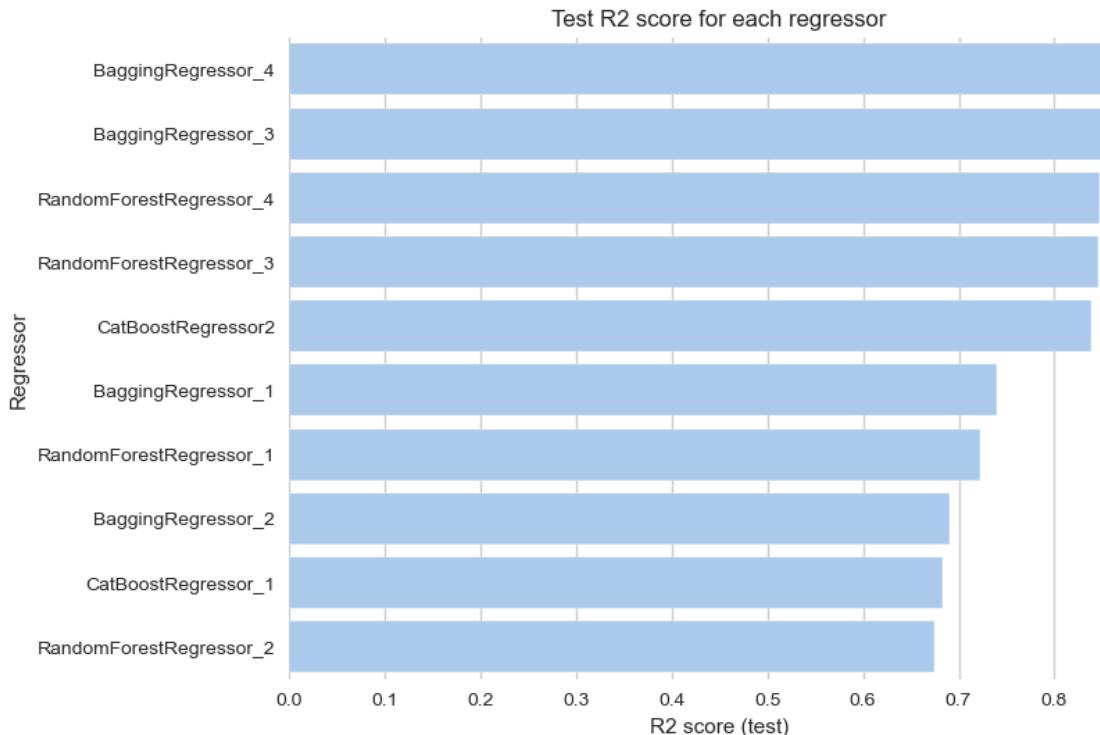
### 1.21.3 Comparación de modelos:

In [663]: # Models results

```
display(models_results_df.sort_values('test_r2_score', ascending=False))
```

	regressor	train_r2_score	test_r2_score
6	BaggingRegressor_4	0.982617	0.856217
4	BaggingRegressor_3	0.982274	0.852508
7	RandomForestRegressor_4	0.947275	0.846601
5	RandomForestRegressor_3	0.946943	0.845112
9	CatBoostRegressor2	0.904457	0.837922
0	BaggingRegressor_1	0.976685	0.739290
1	RandomForestRegressor_1	0.939248	0.723103
2	BaggingRegressor_2	0.970111	0.690796
8	CatBoostRegressor_1	0.990734	0.683262
3	RandomForestRegressor_2	0.924524	0.674443

```
In [664]: # horizontal bar chart
sns.set_style('whitegrid')
sns.set_color_codes('pastel')
plt.figure(figsize=(8, 6))
ax = sns.barplot(x='test_r2_score',
                  y='regressor',
                  data=models_results_df.sort_values('test_r2_score', ascending=False),
                  color='b')
sns.despine(left=True, bottom=True)
plt.title('Test R2 score for each regressor')
plt.xlabel('R2 score (test)')
plt.ylabel('Regressor')
plt.show()
```



Para predecir el precio de las casas de Barcelona, elegiría BaggingRegressor\_4 o BaggingRegressor\_3 porque tienen los valores test\_r2\_score más altos (0.856 y 0.852, respectivamente). Esto indica que estos modelos han funcionado bien durante las fases de entrenamiento y validación, y es probable que tengan un buen poder predictivo para datos nuevos no vistos.

Aunque CatBoostRegressor\_1 tiene el valor train\_r2\_score más alto (0.99), su puntuación test\_r2\_score (0.68) es relativamente más baja en comparación con los modelos BaggingRegressor\_3 y BaggingRegressor\_4. Puede ser que se esté sobreajustando a los datos de entrenamiento (overfitting) y, por tanto, no funciona tan bien con datos nuevos no vistos. Por otro lado, CatBoostRegressor\_2 tiene un valor train\_r2\_score más bajo (0.90) y una puntuación test\_r2\_score relativamente alta (0.83), lo que indica que el modelo ha generalizado relativamente bien a los nuevos datos.

## 1.22 Generación de predicciones y análisis de rentabilidad

Hemos generado 10 modelos distintos y los hemos entrenado con los datos de venta.

Ahora, generaremos predicciones del precio de venta de la vivienda sobre el dataset de alquiler. Por tanto, en el dataset de alquiler, tendremos el precio de alquiler (que ya viene dado) y el precio de venta predicho por nuestro modelo.

De esta forma, podremos hacer una comparativa. Podremos calcular el retorno de la inversión, es decir, el precio de venta dividido entre el precio mensual por alquiler. Esto nos va a dar meses. Por tanto, x meses significará que con el alquiler de x meses nos podemos pagar el piso (el precio de venta total).

Además, haremos un análisis del retorno de la inversión: qué viviendas son más rentables en Barcelona para comprar y luego alquilar, en qué barrios, qué tamaños tienen, si tienen aire acondicionado o no, etc.

1. Selección del mejor modelo de entre todos los calculados para predecir el precio de la vivienda: Modelo 3.1 Bagging.
2. Preparación del dataset de alquiler de pisos.
3. Cálculo de las predicciones sobre el dataset de alquiler.
4. Cálculo de la métrica de Break-even, resultante de dividir la predicción del precio de la vivienda por el precio de alquiler por mes.

### 1.22.1 Preparación del dataset de alquiler de pisos

Debemos preprocessar y transformar el dataset de alquiler de la misma manera que el de venta. Por tanto seguiremos los mismos pasos:

```
In [450]: # Cargamos los datos crudos del dataset de alquiler  
raw_data_renting = pd.read_csv('renting_Barcelona.csv', delimiter = ',',  
explore_data(raw_data_renting)
```

The dataset includes 8502 instances (rows) and 33 variables (columns).

	<b>id</b>	<b>price</b>	<b>currency</b>	<b>latitude</b>	<b>longitude</b>	<b>sq_meters</b>	<b>sq_meters_built</b>	<b>rooms</b>	<b>bathroom</b>
<b>0</b>	536625	850	€/mes	41.401708	2.154077	52.0		55	2
<b>1</b>	545910	725	€/mes	41.407221	2.135569	32.0		37	2
<b>2</b>	570697	950	€/mes	41.411508	2.164608	NaN		72	3
<b>3</b>	591588	750	€/mes	41.402256	2.140764	NaN		45	1
<b>4</b>	610243	990	€/mes	41.405327	2.146929	NaN		45	1

Primero, exploraremos el dataset de alquiler para poder definir, sobre todo, los diferentes grupos de las variables que hemos convertido a rangos anteriormente en el dataset de venta, como `property_age_cat`, `floor_cat` y `neighborhood_rent_index_cluster`.

Una vez creados los rangos, luego los definiremos dentro de la clase `DataPreprocessorRent()` para automatizar el preprocesado del dataset de alquiler.

```
In [451]: # creamos el que será el dataset de alquiler preprocesado  
rent_data_preprocessed = raw_data_renting.copy(deep=True)
```

## Edad Vivienda

```
In [439]: # creamos variable 'property_age'
rent_data_preprocessed['property_age'] = 2023 - rent_data_preprocessed

# Histograma de la variable precio
plot_histogram(rent_data_preprocessed, 'property_age')
# Create a copy
df = rent_data_preprocessed.dropna(subset=['property_age']).copy()

# Scale the property_age column using StandardScaler
X = StandardScaler().fit_transform(df['property_age'].values.reshape(-1, 1))

# Evaluate DBSCAN for range of eps values and select best_eps
eps_range = np.arange(0.1, 1.1, 0.1)
scores = [len(set(DBSCAN(eps=eps, min_samples=5).fit(X).labels_)) - 1 for eps in eps_range]
best_eps = eps_range[np.argmax(scores)]
# optimal number of clusters
k = len(set(DBSCAN(eps=best_eps, min_samples=5).fit(X).labels_)) - 1
print("The optimal number of clusters is:", k)

# Perform K-means clustering on the property_age variable with the optimal number of clusters
kmeans = KMeans(n_clusters=k, random_state=0).fit(X)

# Assign each property to a cluster based on its property age
df['property_age_cluster'] = kmeans.labels_

# Visualize the clusters using a scatter plot
fig, ax = plt.subplots(figsize=(15, 10))
sns.scatterplot(data=df, x='property_age', y='property_age_cluster', hue='property_age_cluster')
plt.xlabel('Property age')
plt.ylabel('Cluster')
plt.show()

# ordenamos por el mínimo en orden descendente:
def q25(x): return x.quantile(0.25)
def q75(x): return x.quantile(0.75)

display(df.groupby('property_age_cluster', as_index=False)\n        .agg({'property_age': ['min', 'max', 'mean', 'median', 'q25', 'q75']})\n        .sort_values(('property_age', 'min'), ascending=False))

def property_age_rent_conditions(x: float) -> str:
    """
    Assigns a property age range label to a property age value.

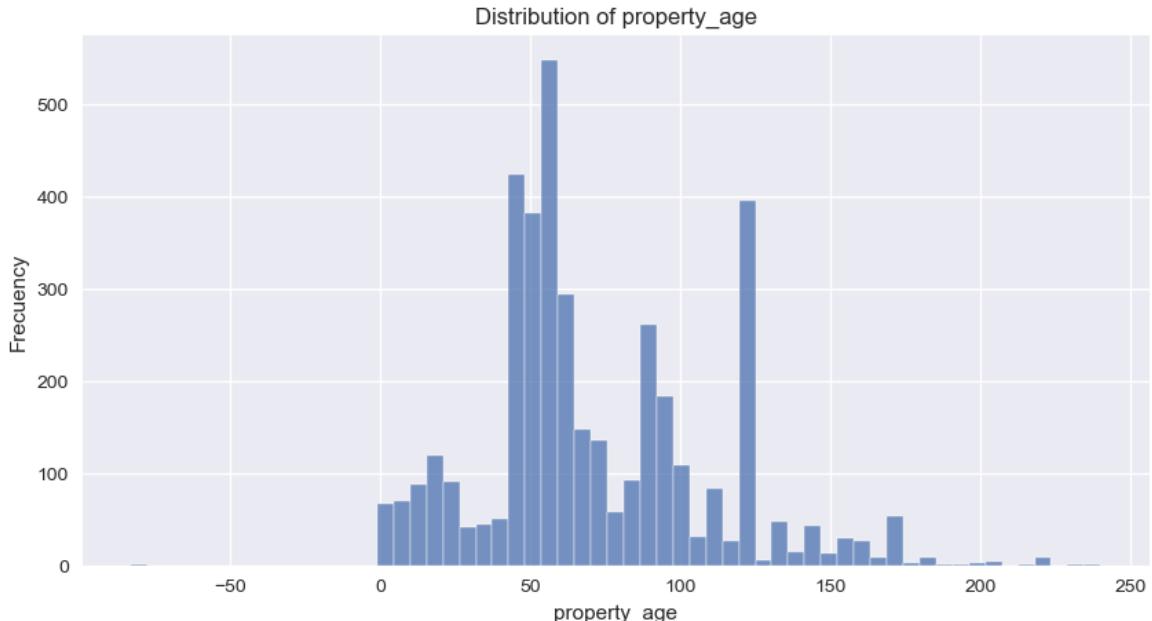
    Parameters:
        x (float): The property age value.

    Returns:
        str: The label for the property age range that the value falls into.
    """
    if pd.isna(x): # check for NaN values
        return "Unknown"
    elif x < 0: # viviendas dónde no tengo valores (missing values)
        return "Missing Values"
    else:
        if x <= 10:
            return "0-10 years old"
        elif x <= 20:
            return "10-20 years old"
        elif x <= 30:
            return "20-30 years old"
        elif x <= 40:
            return "30-40 years old"
        elif x <= 50:
            return "40-50 years old"
        elif x <= 60:
            return "50-60 years old"
        elif x <= 70:
            return "60-70 years old"
        elif x <= 80:
            return "70-80 years old"
        elif x <= 90:
            return "80-90 years old"
        else:
            return "90+ years old"
```

```

    return "Unknown"
elif 0 <= x <= 35: # entre 0 y 35 años
    return "0 - 35"
elif 35 < x <= 71: # entre 35 y 71 años
    return "35 - 71"
elif 71 < x <= 111: # entre 71 y 111 años
    return "71 - 111"
elif x > 111: # más de 111 años
    return "+111"
else:
    return "Unknown"

```



The optimal number of clusters is: 4

The default value of `n\_init` will change from 10 to 'auto' in 1.4. Set the value of `n\_init` explicitly to suppress the warning



**property\_age\_cluster** **property\_age**

		min	max	mean	median	q25	q75	
<b>1</b>		1	112.0	240.0	134.015769	123.0	123.0	143.0
<b>3</b>		3	72.0	111.0	88.756410	87.0	83.0	93.0
<b>0</b>		0	36.0	71.0	54.425916	54.0	48.0	59.0
<b>2</b>		2	-83.0	35.0	16.400794	17.0	8.0	23.0

**Floor**

```
In [442]: df_floor = rent_data_preprocessed.copy()

floor = df_floor[['floor', 'price']].groupby('floor', as_index=False).
    # Number of instances per category
category_count=('price', "count"),
    # Mean price
mean_price=('price', "mean"),
    # Median price
median_price=('price', "median"),
    # Min price
min_price=('price', min),
    # Max price
max_price=('price', max),
    # Standard deviation
standard_deviation=('price', "std")).sort_values('floor', ascending=True)
display(floor)

plt.bar(x='floor', height='median_price', data=floor)
plt.rcParams["figure.figsize"] = (50,20)
plt.xticks(fontsize=40) # tamaño de las etiquetas del eje x
plt.yticks(fontsize=40) # tamaño de las etiquetas del eje y
plt.xlabel('Piso (floor)', fontsize=40) # etiqueta del eje x y su tamaño
plt.ylabel('Precio mediano', fontsize=40) # etiqueta del eje y y su tamaño
plt.title('Precio mediano de alquiler en BCN por piso', fontsize=50)
plt.show()

fig, ax = plt.subplots(figsize=(50, 20))
ax.bar(x='floor', height='median_price', data=floor[floor['floor'] < 1])
ax.set_xlabel('Piso (floor)', fontsize=30)
ax.set_ylabel('Precio mediano', fontsize=30)
ax.set_title('Precio mediano del alquiler en BCN por piso (solo valores < 1 piso)')
ax.tick_params(axis='both', labelsize=20) # aumenta el tamaño de las etiquetas
plt.show()

def conditions_floor_rent(x: float) -> str:
    """
    Categorize floors of a property

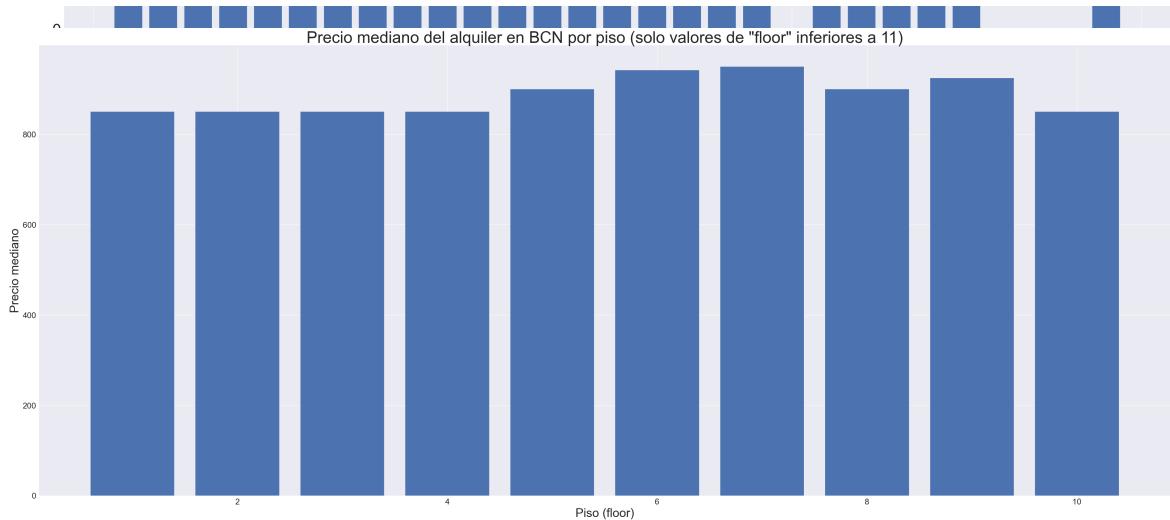
    Parameters:
    x (Union[float, int]): Number of floors of a property

    Returns:
    str: Categorized floor of the property
    """
    if pd.isna(x): # check for NaN values
        return "Unknown"
    elif x < 0: # viviendas dónde no tengo valores (missing values)
        return "Unknown"
    elif ((x >= 0) & (x <= 5)): # viviendas entre 0 y 5 pisos
        return "0 - 5"
    elif ((x >= 5) & (x <= 10)): # viviendas entre 5 y 10 pisos
        return "5 - 10"
    elif ((x > 10)): # viviendas con más de 10 pisos
        return "+10"
    else:
        return "Unknown"
```

```
return "Unknown"
```

	<b>index</b>	<b>floor</b>	<b>category_count</b>	<b>mean_price</b>	<b>median_price</b>	<b>min_price</b>	<b>max_price</b>	<b>standard_de</b>
0	0	1.0	1621	1058.713140	850.0	375	15000	872.
1	1	2.0	1444	1034.605956	850.0	450	8000	657.
2	2	3.0	1247	1076.694467	850.0	365	11000	728.
3	3	4.0	967	1109.232678	850.0	480	14000	919.
4	4	5.0	583	1124.686106	900.0	419	7250	620.
5	5	6.0	298	1227.724832	942.5	550	7200	820.
6	6	7.0	192	1345.104167	950.0	500	8000	1027.
7	7	8.0	87	1079.298851	900.0	460	3500	562.
8	8	9.0	41	1371.560976	925.0	700	3400	746.
9	9	10.0	21	1122.285714	850.0	675	2500	572.
10	10	11.0	4	1650.000000	1600.0	1000	2400	704.
11	11	12.0	6	1573.833333	1200.0	823	3000	884.
12	12	13.0	3	1224.666667	1049.0	925	1700	416.
13	13	14.0	7	1702.142857	1200.0	760	3900	1169.
14	14	15.0	1	710.000000	710.0	710	710	
15	15	16.0	1	895.000000	895.0	895	895	
16	16	17.0	1	1800.000000	1800.0	1800	1800	
17	17	18.0	1	1795.000000	1795.0	1795	1795	
18	18	19.0	1	1100.000000	1100.0	1100	1100	
19	19	21.0	2	7750.000000	7750.0	6500	9000	1767.
20	20	22.0	1	6500.000000	6500.0	6500	6500	
21	21	23.0	1	6000.000000	6000.0	6000	6000	
22	22	24.0	2	6500.000000	6500.0	6000	7000	707.
23	23	25.0	3	6266.666667	6000.0	2800	10000	3607.
24	24	29.0	1	3800.000000	3800.0	3800	3800	





**Reducción de dimensionalidad de los barrios y categorización según su precio por metro cuadrado**

```
In [445]: # número de barrios en BCN
print(f"Hay {rent_data_preprocessed['neighborhood'].nunique()} barrios")

neighborhood_df = rent_data_preprocessed[['neighborhood', 'price', 'sq_meters_built']]
# Number of instances per category
category_count = ('price', "count"),
# Mean price
mean_price = ('price', "mean"),
# Median price
mean_sq_meters_built = ('sq_meters_built', "mean"),
# Median price
median_price = ('price', "median"),
# Median price
median_sq_meters_built = ('sq_meters_built', "median"))

neighborhood_df['mean_price_per_sq_meters_built'] = neighborhood_df['median_price_per_sq_meters_built']
neighborhood_df['median_price_per_sq_meters_built'] = neighborhood_df['mean_price_per_sq_meters_built']
neighborhood_df.head()

plt.bar(x = 'neighborhood',
        height = 'median_price_per_sq_meters_built',
        data = neighborhood_df.sort_values('median_price_per_sq_meters_built'))
plt.rcParams["figure.figsize"] = (50,20)
plt.xticks(fontsize=18, rotation=90)
plt.yticks(fontsize=18)
plt.title("Distribución del precio mediano por metro cuadrado por barrio")
plt.show()

# Definir los límites de cada rango de precios mediante quantiles
bajo = neighborhood_df['median_price_per_sq_meters_built'].quantile(0.0)
medio = neighborhood_df['median_price_per_sq_meters_built'].quantile(0.5)
alto = neighborhood_df['median_price_per_sq_meters_built'].quantile(1.0)

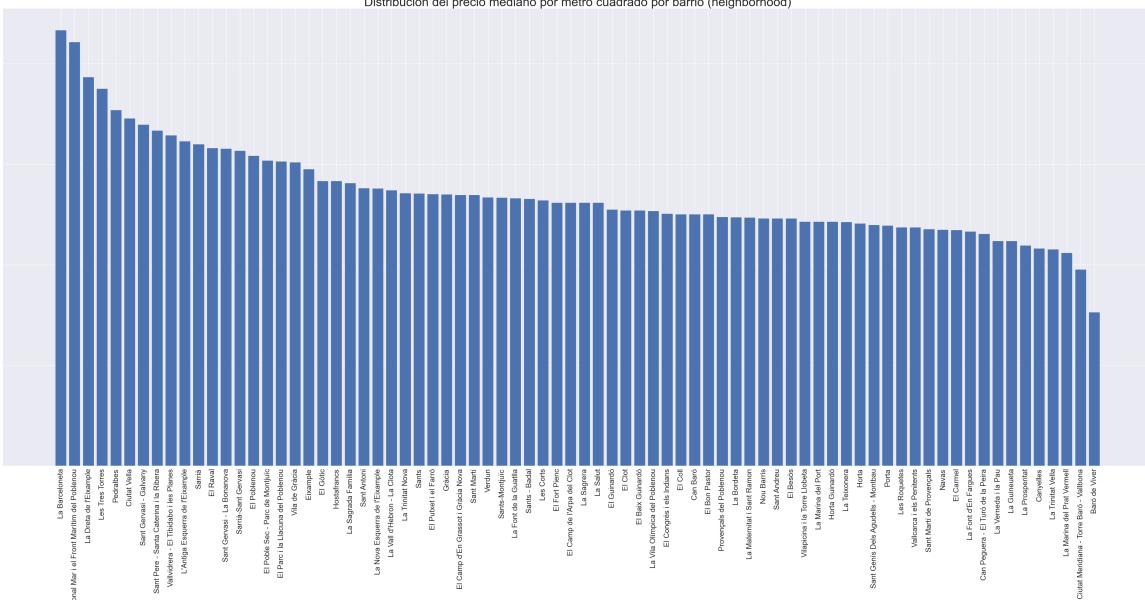
print(f"bajo: {bajo}")
print(f"medio: {medio}")
print(f"alto: {alto}")

# Función para asignar tier
def assign_price_tier_cluster_rent(x: float) -> str:
    if x < 0:
        return "Unknown"
    elif ((x >= 0) & (x < 11.5)): # bajo
        return "0 - 11.5"

    elif ((x >= 11.5) & (x < 15)): # medio
        return "11.5 - 15"

    elif ((x >= 15)): # alto
        return "+15"
    else:
        return "Unknown"
```

## Hay 76 barrios en RCN



bajo: 12.134853905784137

medio: 13.076923076923077

alto: 14.3125

### 1.22.1.1 Rent dataset pre-processing pipeline:

Ahora ya podemos actualizar, para el dataset de alquiler, las funciones que calculan los rangos de edad de la vivienda, piso y precio mediano por metro cuadrado construido:

```
In [453]: class DataPreprocessorRent:
    target=TARGET_VARIABLE
    not_useful_vars = ['id',
                        'doorman',
                        'quality',
                        'city',
                        'currency',
                        'furniture',
                        'closest_station',
                        'created_at',
                        'last_seen']
    ]
    def dataset_preprocessor(self, file_path):
        self.load_data(file_path)
        self.initial_feature_selection()
        self.replace_binary_cols_nulls_by_zero()
        self.convert_binary_to_category()

        self.property_age_cat()
        self.property_age_rent_conditions(self.df['property_age'])

        self.missing_values()

        self.floor_cat()

        self.dim_reduction_neighborhood()

        self.convert_object_to_category()
        self.bathrooms_roms_per_100_sqm()
        self.feature_selection_2()

    def load_data(self, file_path):
        self.df = pd.read_csv(file_path, delimiter=',').copy()

    def initial_feature_selection(self):
        self.df.drop(self.not_useful_vars, axis=1, inplace=True)

    def get_binary_cols(self) -> list[str]:
        """
        Returns a list with the names of the binary variables in a Pandas DataFrame.
        Parameters:
            None
        Returns:
            A list with the names of the binary variables in the DataFrame.
        """
        binary_cols = [col for col in self.df if (len(self.df[col].value_counts()) == 2)]
        return binary_cols

    def replace_binary_cols_nulls_by_zero(self) -> None:
```

```
"""
    Replaces missing values in binary columns of a pandas DataFrame

Parameters:
    None

Returns:
    None
"""

binary_cols = self.get_binary_cols()
if binary_cols:
    self.df.loc[:, binary_cols] = self.df.loc[:, binary_cols].fillna(0)

def convert_binary_to_category(self) -> None:
    # primero las convertimos a int64 para eliminar los decimales:
    self.df[self.get_binary_cols()] = self.df[self.get_binary_cols()]
    # luego las convertimos a 'category': '0' y '1'
    self.df[self.get_binary_cols()] = self.df[self.get_binary_cols()].astype('category')

def property_age_cat(self) -> None:
    # Creamos la edad (años) de la vivienda:
    self.df['property_age'] = 2023 - self.df['year_built']
    # aplicamos la función a la columna 'property_age' del DataFrame
    self.df['property_age_cat'] = self.property_age_rent_condition

def property_age_rent_conditions(self, x: pd.Series) -> pd.Series:
    """
        Assigns a property age range label to a series of property age

    Args:
        x (pd.Series): A series of property age values.

    Returns:
        pd.Series: A series of property age range labels.
    """

    conditions = [
        pd.isna(x),
        x < 0,
        x.between(0, 35),
        x.between(36, 71),
        x.between(72, 111),
        x > 111,
    ]
    choices = [
        "Unknown",
        "Unknown",
        "0 - 35",
        "35 - 71",
        "71 - 111",
        "+111",
    ]
    return np.select(conditions, choices)
```

```
def missing_values(self) -> None:
    # variables categóricas con missing values que queremos convertir
    cat_features_na_to_unknown = ['property_type',
                                    'orientation',
                                    'heating']
    ]
    # Fill missing values with 'Unknown' for all columns
    for col in cat_features_na_to_unknown:
        if col == 'heating':
            self.df[col].fillna('Unknown', inplace=True)
            self.df[col].replace('na', 'Unknown', inplace=True)
        else:
            self.df[col].fillna('Unknown', inplace=True)

def conditions_floor_rent(self, x: pd.Series) -> pd.Series:
    """
    Categorize floors of a property and assign to a new column 'floor'
    """
    conditions = [
        pd.isna(x),
        x < 0,
        x.between(0, 5),
        x.between(6, 10),
        x > 10,
    ]
    choices = [
        "Unknown",
        "Unknown",
        "0 - 5",
        "5 - 10",
        "+10",
    ]
    return np.select(conditions, choices)

def floor_cat(self) -> None:
    # aplicamos la función a la columna 'floor' del DataFrame
    #self.df['floor_cat'] = self.df['floor'].apply(conditions_floor_rent)
    self.df['floor_cat'] = self.conditions_floor_rent(self.df['floor'])
    #self.df['floor_cat'] = self.conditions_floor(self.df['floor'])

def dim_reduction_neighborhood(self) -> None:
    neighborhood_df = self.df[['neighborhood', 'price', 'sq_meters']]
    # Number of instances per category
    category_count = ('price', 'count'),
    # Mean price
    mean_price = ('price', 'mean'),
    # Median price
    mean_sq_meters_built = ('sq_meters_built', 'mean'),
    # Median price
    median_price = ('price', 'median'),
```

```
# Median price
median_sq_meters_built = ('sq_meters_built', "median"))

neighborhood_df['mean_price_per_sq_meters_built'] = neighborhood_df['sq_meters_built'].mean()
neighborhood_df['median_price_per_sq_meters_built'] = neighborhood_df['sq_meters_built'].median()

neighborhood_rent_index_cluster = self.assign_price_tier_cluster(neighborhood_df)

neighborhood_df = neighborhood_df.assign(neighborhood_rent_index_cluster)

# agregamos esta información el modelo de datos:
neighborhood_mapping = neighborhood_df[['neighborhood', 'neighborhood_rent_index_cluster']]
self.df = pd.merge(self.df, neighborhood_mapping, on='neighborhood')

# asignamos tiers
def assign_price_tier_cluster_rent(self, x: Optional[pd.Series] = None) -> str:
    if x is None:
        x = self.df['median_price_per_sq_meters_built']

    conditions = [
        pd.isna(x),
        x < 0,
        x < 11.5, # bajo
        x < 15, # medio
        x >= 15 # alto
    ]
    choices = [
        "Unknown",
        "Unknown",
        "0 - 11.5",
        "11.5 - 15",
        "+15"
    ]
    return np.select(conditions, choices)

#####
##### Convertimos las categóricas de object a category #####
def convert_object_to_category(self) -> pd.DataFrame:
    """
    Convert object columns in a pandas dataframe to category data

    Parameters:
    df (pd.DataFrame): Input pandas dataframe

    Returns:
    pd.DataFrame: A copy of the input dataframe with object column converted to category
    """
    # loop through columns and convert object columns to category
    for col in self.df.columns:
        if self.df[col].dtype == 'object':
            self.df[col] = self.df[col].astype('category')

    return self.df
```

```
##### Tratamiento de sq_meters_built en relación al número de habitaciones
def bathrooms_rooms_per_100_sqm(self) -> None:
    self.df['rooms_per_100_sqm'] = self.df['rooms'] * 100 / self.df['sq_meters']
    self.df['bathrooms_per_100_sqm'] = self.df['bathrooms'] * 100 / self.df['sq_meters']

def feature_selection_2(self) -> None:
    ##### Feature Selection 2: Eliminación de variables no útiles
    # variables a eliminar
    not_useful_vars = ['year_built', 'sq_meters', 'floor', 'property_type']
    # eliminamos not_useful_vars
    self.df.drop(not_useful_vars, axis=1, inplace=True)

# RENT DATASET PREPROCESSING
file_path_rent = 'renting_Barcelona.csv'

data_preprocessor = DataPreprocessorRent()
data_preprocessor.dataset_preprocessor(file_path_rent)

# guardamos en csv
data_preprocessor.df.to_csv('rent_data_preprocessed.csv', index=False)
```

```
# cargamos el dataset de rent preprocesado
rent_data_preprocessed = pd.read_csv('rent_data_preprocessed.csv', delimiter=';')
# convertimos object a category
rent_data_preprocessed = convert_object_to_category(rent_data_preprocessed)
explore_data(rent_data_preprocessed)
```

The dataset includes 8502 instances (rows) and 26 variables (columns).

	price	latitude	longitude	sq_meters_built	rooms	bathrooms	balcony	terrace	exterior
0	850	41.401708	2.154077	55	2	1	0	0	1
1	725	41.407221	2.135569	37	2	1	1	0	1
2	950	41.411508	2.164608	72	3	1	0	0	1
3	750	41.402256	2.140764	45	1	1	1	0	1
4	990	41.405327	2.146929	45	1	1	0	0	1

## 1.22.2 Cálculo de predicciones del precio de venta de la vivienda sobre el dataset de alquiler

Recordemos que el modelo que mejor predijo el precio de la vivienda fué el modelo 3.1 de bagging, es decir, el modelo de bagging en el dataset de venta donde se había seleccionando las variables más importantes (mediante técnicas como PCA, Análisis de Correlación, Información Mutua, ANOVA y ExtraTreesRegressor), tratando outliers (método

IQR) pero sin escalar las variables numéricas.

Por tanto, aplicamos otra capa de procesamiento a los datasets de alquiler para tenerlos en el formato óptimo para hacer predicciones.

```
In [496]: # creamos los datasets para el tercer modelo de Bagging y RF
bagging_rent_preprocessed = rent_data_preprocessed.copy(deep=True)

# lista contiene las variables que se consideran más importantes o info
# para predecir el precio de la vivienda en Barcelona, según los diferentes predictores
good_predictors = ["price", # target variable
                    "sq_meters_built", "longitude",
                    "dist_city_center", "latitude", "bathrooms_per_100_rooms",
                    "rooms_per_100_sqm", "neighborhood_rent_index_cluster",
                    "garage", "garden", "pool", "terrace", "property_type",
                    "property_age_cat", "elevator", "ac", "heating"]
# nos quedamos sólo con las columnas indicadas en la lista good_predictors
bagging_rent_preprocessed = bagging_rent_preprocessed[good_predictors]

# Debemos tener las mismas columnas que en el dataset de venta:
# eliminamos la columna property_type = 'Unknown' (hay 2 'Unknown')
bagging_rent_preprocessed.drop(bagging_rent_preprocessed[bagging_rent_]

# variables que queremos codificar usando ordinal encoder
oe_cols = oe_cols = ['property_age_cat', 'neighborhood_rent_index_cluster']
# Define the order of the categories for each column
property_age_cat_order = ['Unknown', '0 - 35', '35 - 71', '71 - 111',
                           neighborhood_rent_index_cluster_order = ['Unknown', '0 - 11.5', '11.5'
# Create an instance of the ordinal encoder
encoder = OrdinalEncoder(categories=[property_age_cat_order,
                                      neighborhood_rent_index_cluster_o

for col in oe_cols:
    # Create a new column name for the encoded data
    new_col = col + '_encoded'
    # Create an instance of the ordinal encoder
    encoder = OrdinalEncoder(categories=[eval(col+'_order')])
    # Fit and transform the selected columns
    bagging_rent_preprocessed[new_col] = encoder.fit_transform(bagging_rent_
    # replace 0 by -1 ('Unknown' by -1)
    bagging_rent_preprocessed[new_col] = bagging_rent_preprocessed[new_col].replace(0, -1)

# variables que queremos codificar usando OHE
ohe_cols = ["garage", "garden", "pool",
            "terrace", "property_type",
            "elevator", "ac", "heating"]

# aplicamos one - hot encoding al dataset
bagging_rent_preprocessed = one_hot_encode(bagging_rent_preprocessed, ohe_cols)

# eliminamos variables categóricas no codificadas
no_encoded_cat_vars = bagging_rent_preprocessed.select_dtypes(include='category')
bagging_rent_preprocessed.drop(no_encoded_cat_vars, axis=1, inplace=True)

##### Tratamos los outliers pero NO escalamos los atributos numéricos
num_vars = ["price", "sq_meters_built", "longitude", "dist_city_center",
            "latitude", "bathrooms_per_100_sqm", "rooms_per_100_sqm"]
```

```
# define the preprocessing pipeline
preprocessing_pipeline = Pipeline([
    ('outlier_removal', OutlierRemover(cols=num_vars))
    #,('scaler', ColumnTransformer([('num_scaler', MinMaxScaler(), num
                                    #remainder='passthrough')])
])

# Apply the preprocessing pipeline to the training and testing data
# Fit and transform the training data
bagging_rent_preprocessed = pd.DataFrame(preprocessing_pipeline.fit_transform(
    bagging_rent,
    columns=num_vars + list(bagging_rent)[len(num_vars):]
))

# exploración rápida del df
print("RENTING DATASET PRE-PROCESSED FOR BAGGING: ")
explore_data(bagging_rent_preprocessed)
RENTING DATASET PRE-PROCESSED FOR BAGGING:
The dataset includes 7176 instances (rows) and 34 variables (columns).
```

`sparse` was renamed to `sparse\_output` in version 1.2 and will be removed in 1.4. `sparse\_output` is ignored unless you leave `sparse` to its default value.

	price	sq_meters_built	longitude	dist_city_center	latitude	bathrooms_per_100_sqm	room
0	850	55	2.154077	2.026455	41.401708	1.818182	
1	725	37	2.135569	3.582409	41.407221	2.702703	
2	950	72	2.164608	2.663025	41.411508	1.388889	
3	750	45	2.140764	2.910067	41.402256	2.222222	
4	990	45	2.146929	2.727149	41.405327	2.222222	

Ya tenemos el dataset de alquiler procesado.

Hacemos las predicciones del precio de venta de las viviendas sobre el dataset de alquiler con el Modelo 3.1 de Bagging:

```
In [505]: # When we have already trained and obtained the 'results_4' object using
          # eliminamos el precio del dataset de alquiler para que tenga las mismas
          # características que el dataset de venta
data_pred = bagging_rent_preprocessed.drop([TARGET_VARIABLE], axis=1)

# make predictions using the trained model
y_pred = results_4.predict(data_pred)

# Add the predictions to the preprocessed dataset
pred_price_sale_df = data_pred.copy(deep=True)
pred_price_sale_df['pred_selling_price'] = y_pred

# Save the updated dataset with predictions
pred_price_sale_df.to_csv('pred_price_sale_df.csv', index=False)
```

In [507]: # resultados predicción  
pred\_price\_sale\_df.head()

Out[507]:

	sq_meters_built	longitude	dist_city_center	latitude	bathrooms_per_100_sqm	rooms_per_sqm
0	55	2.154077	2.026455	41.401708	1.818182	1.222222
1	37	2.135569	3.582409	41.407221	2.702703	1.222222
2	72	2.164608	2.663025	41.411508	1.388889	1.222222
3	45	2.140764	2.910067	41.402256	2.222222	1.222222
4	45	2.146929	2.727149	41.405327	2.222222	1.222222

In [511]: # Merge (por índice, ya que son los mismos) de los dos datasets:  
# bagging\_rent\_preprocessed y pred\_price\_sale\_df (que contiene el precio de venta)  
rent\_pred\_selling\_price\_df = bagging\_rent\_preprocessed.merge(pred\_price\_sale\_df, on='id')  
# Save the merge  
rent\_pred\_selling\_price\_df.to\_csv('rent\_pred\_selling\_price\_df.csv', index=False)  
rent\_pred\_selling\_price\_df.head()

Out[511]:

	price	sq_meters_built	longitude	dist_city_center	latitude	bathrooms_per_100_sqm	rooms_per_sqm
0	850	55	2.154077	2.026455	41.401708	1.818182	1.222222
1	725	37	2.135569	3.582409	41.407221	2.702703	1.222222
2	950	72	2.164608	2.663025	41.411508	1.388889	1.222222
3	750	45	2.140764	2.910067	41.402256	2.222222	1.222222
4	990	45	2.146929	2.727149	41.405327	2.222222	1.222222

### 1.22.3 Cálculo de la métrica de Break-even

Hemos generado 10 modelos distintos y los hemos entrenado con los datos de venta. Ahora, hemos generado predicciones del precio de venta de la vivienda sobre el dataset de alquiler. Por tanto, ahora en el dataset de alquiler, tenemos el precio de alquiler (que ya viene dado) y el precio de venta predicho por nuestro modelo.

De esta forma, podremos hacer una comparativa: calcularemos el retorno de la inversión. Es decir, el precio de venta dividido entre el precio mensual por alquiler. Esto nos va a dar meses, por tanto, x meses significará que con el alquiler de x meses nos podemos pagar el piso (el precio de venta total).

Con esto, podremos hacer un análisis del retorno de la inversión: qué viviendas son más rentables en Barcelona para comprar y luego alquilar, en qué barrios, qué tamaños tienen, si tienen aire acondicionado o no, etc.

La métrica break-even, que en este caso se expresa en meses, se refiere al número de meses de alquiler necesarios para recuperar el costo de la propiedad. Es decir, cuantos

menor sea el valor de break-even, más rápido se recuperará la inversión en la propiedad a través del alquiler.

Por ejemplo, si la métrica break-even es de 240 meses, significa que se necesitan 240 meses de alquiler para recuperar el costo de la propiedad. Si el alquiler mensual es de 1000 euros y el precio de venta de la propiedad es de 240000 euros, entonces la métrica break-even sería de 240 meses.

La métrica break-even se calcula dividiendo el precio de venta predicho por el precio mensual de alquiler:

```
In [546]: ##### Ejemplo:  
precio_venta = 240000  
precio_alquiler_mensual = 1000  
break_even = precio_venta/precio_alquiler_mensual  
print(f"Se tardarán {break_even} meses de alquiler para recuperar el c
```

Se tardarán 240.0 meses de alquiler para recuperar el costo de la propiedad.

```
In [570]: rent_pred_selling_price_breakeven_df = rent_pred_selling_price_df.copy
```

Break-even en meses:

```
In [571]: # Calcular break-even point: break-even point = pred_selling_price / p  
rent_pred_selling_price_breakeven_df['break_even'] = rent_pred_selling
```

Expresamos el break-even en años para tener una idea más intuitiva de cuánto tiempo se tardaría en recuperar la inversión:

```
In [572]: rent_pred_selling_price_breakeven_df['break_even_years'] = rent_pred_s
```

Mostramos los resultados:

```
In [577]: # ordenamos por columnas que más nos interesan:  
excluded_columns = ['price', 'pred_selling_price', 'break_even', 'break_even_years']  
included_columns = list(set(rent_pred_selling_price_breakeven_df.columns) - set(excluded_columns))  
rent_pred_selling_price_breakeven_df = rent_pred_selling_price_breakeven_df[included_colum
```

```
In [578]: ### Rentables: la inversión en la propiedad a través del alquiler se recuperará relativamente rápido  

display(rent_pred_selling_price_breakeven_df.sort_values('break_even',  

### Poco rentables: la inversión en la propiedad a través del alquiler se recuperará al cabo de muchísimos años  

display(rent_pred_selling_price_breakeven_df.sort_values('break_even',
```

	price	pred_selling_price	break_even	break_even_years	sq_meters_built	dist_city_center
1864	1050	97366.291413	92.729801	7.727483	30	3.731620
4312	1500	167659.963490	111.773309	9.314442	60	0.970419
4132	1300	146400.530243	112.615792	9.384649	30	1.324574
700	1400	158255.759792	113.039828	9.419986	65	3.550204
4363	1300	148715.178773	114.396291	9.533024	30	1.182899

	price	pred_selling_price	break_even	break_even_years	sq_meters_built	dist_city_center
4884	850	473593.805399	557.169183	46.430765	110	1.148911
3821	769	427413.206981	555.803910	46.316993	62	0.299793
1344	700	380362.736780	543.375338	45.281278	80	3.151259
5477	900	473101.875227	525.668750	43.805729	86	1.301826
6523	750	389337.365227	519.116487	43.259707	93	0.380341

- La propiedad que tiene la métrica break-even más baja (en años) es de 7.72, lo que significa que la inversión en la propiedad a través del alquiler se recuperará relativamente rápido: en 93 meses o 8 años aproximadamente.
- La propiedad que tiene la métrica break-even más alta (en años) es de 46.43, lo que significa que la inversión en la propiedad a través del alquiler se recuperará al cabo de muchísimos años: en 557 meses o 46 años aproximadamente.

## 1.22.4 Análisis del retorno de la inversión

Hemos visto cuántos años tardaríamos en recuperar la inversión a través de los ingresos de alquiler.

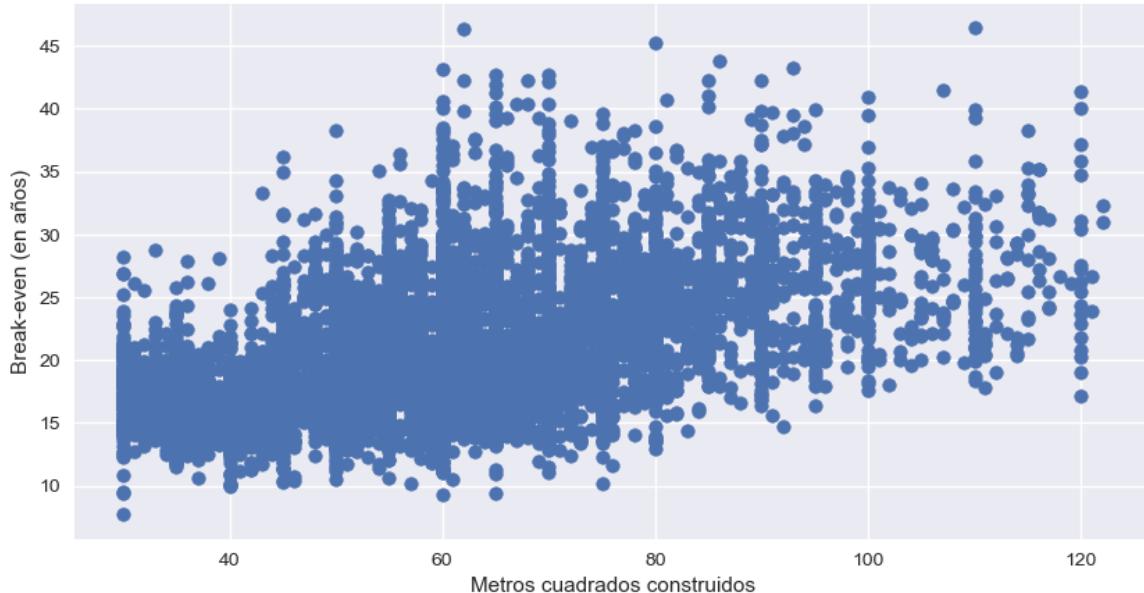
Ahora analizaremos qué tipos de viviendas son las más rentables en Barcelona para comprar y alquilar.

- Haremos un análisis del retorno de la inversión: qué viviendas son más rentables en Barcelona para comprar y luego alquilar, en qué barrios, qué tamaños tienen, si tienen aire acondicionado o no, etc.

### Metros cuadrados construidos

Podría haber una relación inversa entre los metros cuadrados construidos y la métrica de break-even, lo que significa que las viviendas más pequeñas podrían ser más rentables para comprar y alquilar.

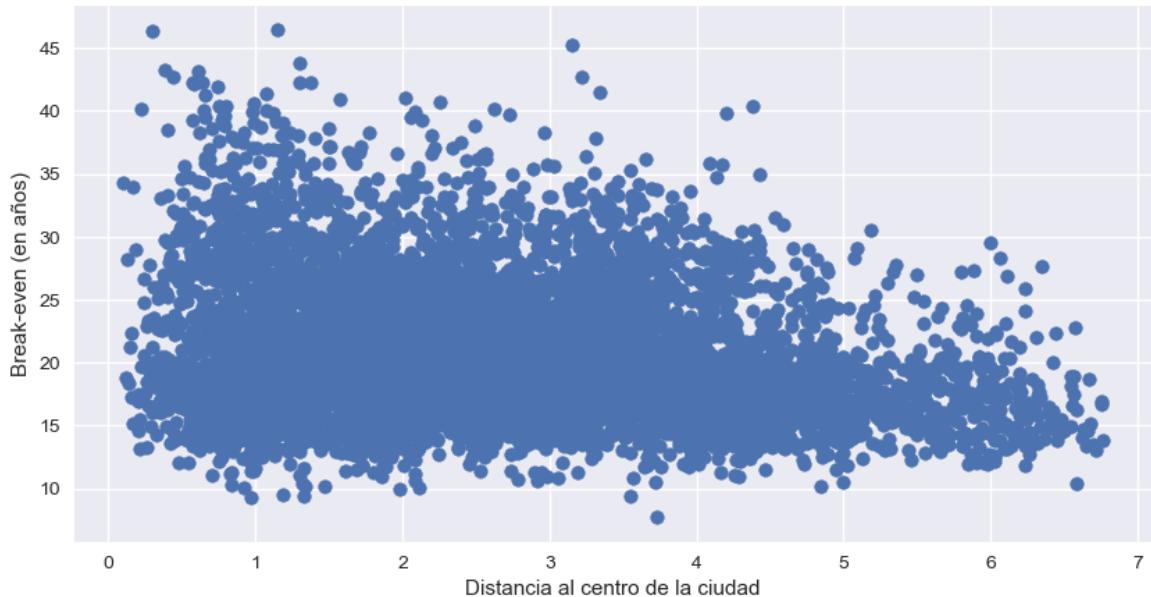
```
In [583]: fig, axs = plt.subplots(figsize=(10, 5))
plt.scatter(rent_pred_selling_price_breakeven_df['sq_meters_built'], r
plt.xlabel('Metros cuadrados construidos')
plt.ylabel('Break-even (en años)')
plt.show()
```



### Distancia al centro de la ciudad

Una propiedad más cercana al centro de la ciudad puede ser más costosa de comprar, pero también podría tener una tasa de alquiler más alta, lo que podría afectar la métrica de break-even y ser más rentable.

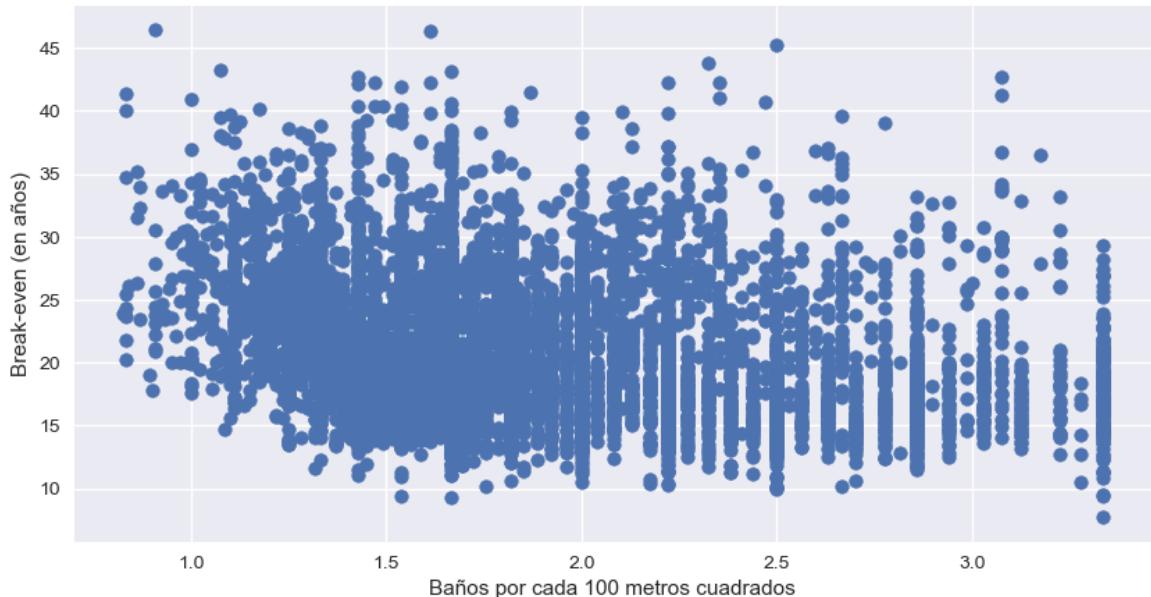
```
In [591]: fig, axs = plt.subplots(figsize=(10, 5))
plt.scatter(rent_pred_selling_price_breakeven_df['dist_city_center'],
            plt.xlabel('Distancia al centro de la ciudad')
            plt.ylabel('Break-even (en años)')
            plt.show()
```



### Baños por cada 100 metros cuadrados

Un mayor número de baños por cada 100 metros cuadrados podría indicar un mayor valor de la propiedad y, por lo tanto, una mayor rentabilidad potencial.

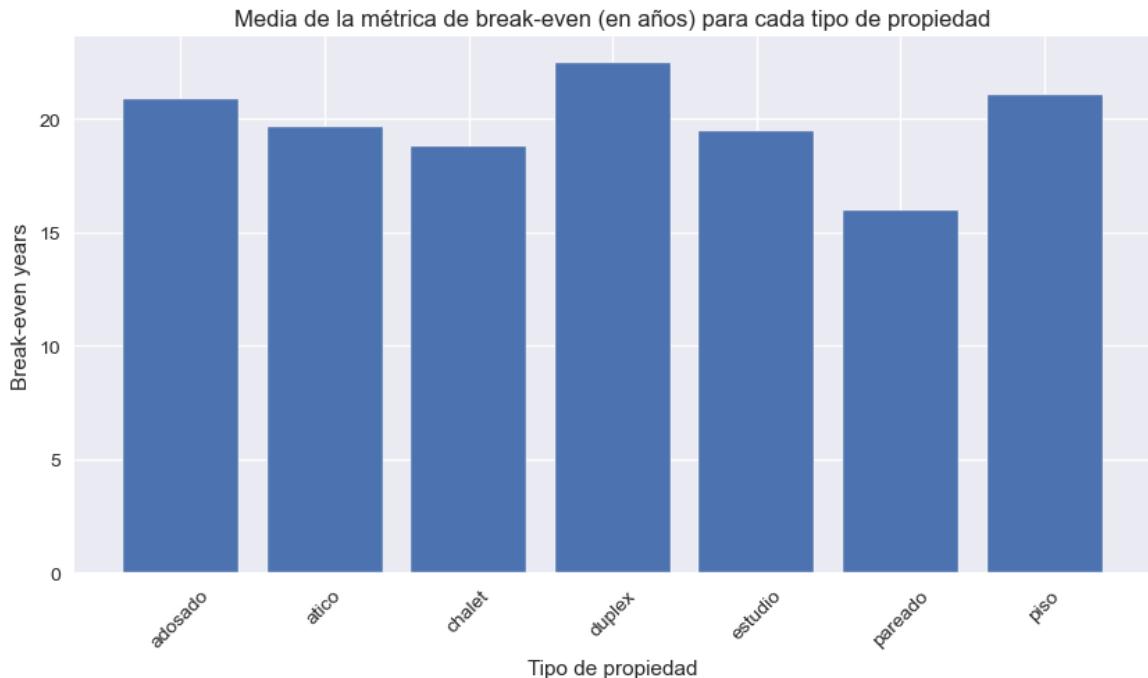
```
In [593]: fig, axs = plt.subplots(figsize=(10, 5))
plt.scatter(rent_pred_selling_price_breakeven_df['bathrooms_per_100_sqm'],
            rent_pred_selling_price_breakeven_df['Break-even (en años)'])
plt.xlabel('Baños por cada 100 metros cuadrados')
plt.ylabel('Break-even (en años)')
plt.show()
```



### Tipo de propiedad

Es posible que algunos tipos de propiedad sean más rentables que otros.

```
In [598]: # Obtener una lista de las columnas de one hot encoding que representa  
property_type_cols = [col for col in rent_pred_selling_price_breakeven  
  
# Crear una nueva columna en el DataFrame que contenga el tipo de prop  
def get_property_type(row):  
    for col in property_type_cols:  
        if row[col] == 1:  
            return col.split('_')[-1]  
    return None  
  
rent_pred_selling_price_breakeven_df['property_type'] = rent_pred_sell  
  
# Agrupar el DataFrame por tipo de propiedad y calcular la media de la  
property_type_means = rent_pred_selling_price_breakeven_df.groupby('pr  
  
# Graficar los resultados  
fig, axs = plt.subplots(figsize=(10, 5))  
plt.bar(property_type_means.index, property_type_means.values)  
plt.xticks(rotation=45)  
plt.title("Media de la métrica de break-even (en años) para cada tipo de propiedad")  
plt.ylabel('Break-even years')  
plt.xlabel('Tipo de propiedad')  
plt.show()
```



- Observamos que el pareado es el que tiene la métrica break-even más baja, seguido del chalet y del ático.
- La propiedad tipo pareado parece la opción más rentable para comprar y alquilar. La siguiente propiedad más rentable es chalet, seguida de estudio, ático, adosado, piso y dúplex.

## Aire acondicionado

Podría ser que las viviendas con aire acondicionado sean más rentables.

```
In [602]: ac_break_even = rent_pred_selling_price_breakeven_df.groupby('ac_1')[  
        print(ac_break_even)
```

```
ac_1  
0.0    248.572044  
1.0    253.220128  
Name: break_even, dtype: float64
```

- Observamos que las viviendas con aire acondicionado tienen una métrica de break-even ligeramente mayor que las viviendas sin aire acondicionado.
- Por tanto, en promedio, las viviendas con aire acondicionado tardan ligeramente un poco más en recuperar la inversión inicial a través del alquiler en comparación con las viviendas sin aire acondicionado.

### Piscina

```
In [603]: pool_break_even = rent_pred_selling_price_breakeven_df.groupby('pool_1')[  
        print(pool_break_even)
```

```
pool_1  
0.0    250.646640  
1.0    243.260668  
Name: break_even, dtype: float64
```

- Observamos que las viviendas con piscina tienen una métrica de break-even ligeramente menor que las viviendas sin piscina.
- Por tanto, en promedio, las viviendas con piscina tardan ligeramente un poco menos en recuperar la inversión inicial a través del alquiler en comparación con las viviendas sin piscina.

### Precio mediano por metro cuadrado por barrio.

```
In [606]: # ['Unknown', '0 - 11.5', '11.5 - 15', '+15']  
  
break_even_by_neighborhood = rent_pred_selling_price_breakeven_df.grou  
print(break_even_by_neighborhood)  
  
neighborhood_rent_index_cluster_encoded  
1.0    212.411137  
2.0    233.982943  
3.0    278.173376  
Name: break_even, dtype: float64
```

- Observamos que los barrios en la categoría 1 (con un precio mediano por metro cuadrado entre 0 y 11.5) tienen la métrica break-even media más baja, mientras que los barrios en la categoría 3 (con un precio mediano por metro cuadrado de más de 15)

tienen la métrica break-even media más alta. Esto podría indicar que las propiedades en los barrios más económicos son más rentables para comprar y alquilar que las propiedades en los barrios más caros.

### Ubicación de la propiedad

Creamos un mapa de calor que muestre la distribución de la métrica de break-even en función de la ubicación de la propiedad, utilizando la longitud y latitud.

In [613]:

```
import folium
from folium.plugins import HeatMap

# Creamos el mapa de calor
heat_map = folium.Map(location=[41.39, 2.15], zoom_start=12)

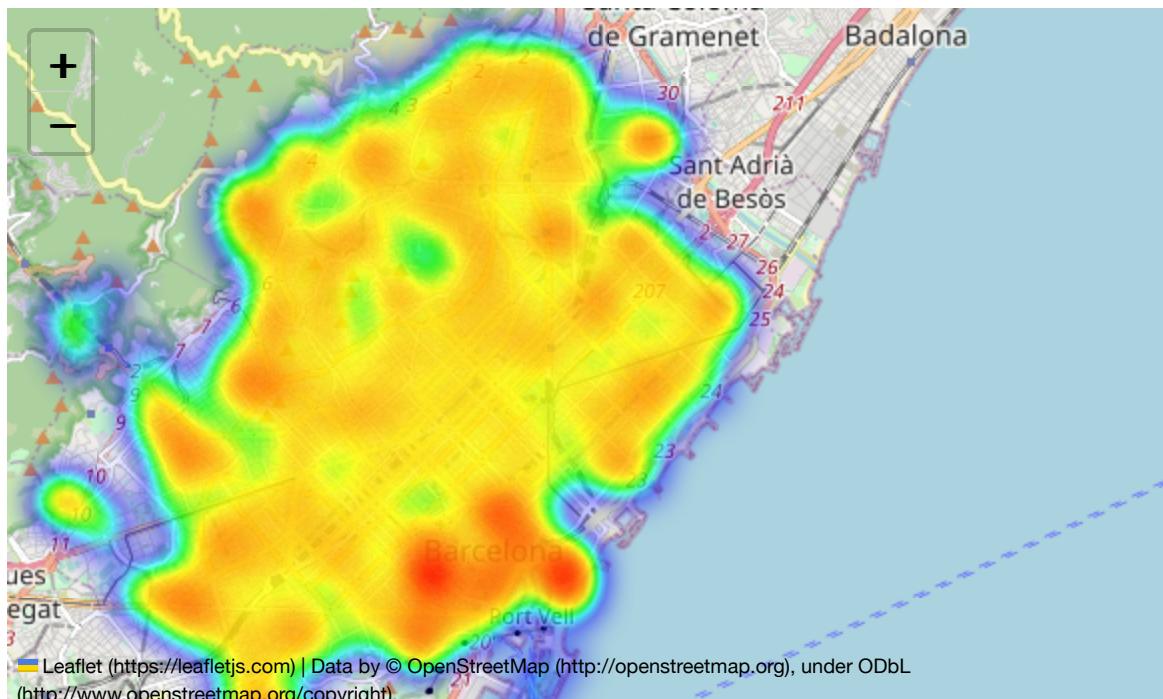
# Convertimos las coordenadas de latitud y longitud en una lista de tuplas
coordinates = list(zip(rent_pred_selling_price_breakeven_df['latitude'],
                       rent_pred_selling_price_breakeven_df['longitude']))

# Creamos una lista de pesos a partir de la métrica de break-even
weights = rent_pred_selling_price_breakeven_df['break_even'].tolist()

# Creamos el mapa de calor con las coordenadas y pesos
HeatMap(coordinates, weights=weights, radius=15).add_to(heat_map)

# Mostramos el mapa de calor
heat_map
```

Out[613]:



- Cada punto en el mapa representa una propiedad, y el color del punto indica la métrica de break-even. Los puntos más rojos indican una métrica de break-even más alta (viviendas menos rentables), mientras que los puntos más claros indican una métrica de

break-even más baja (viviendas más rentables).

## 1.23 Resultados finales, conclusiones y vías abiertas

En base al análisis realizado, podemos concluir que hemos implementado un modelo de bagging (BaggingRegressor) que nos permite predecir razonablemente bien (con un R2 en validación del 0.856) el precio de venta de una vivienda en Barcelona. Utilizando las predicciones de este modelo en el conjunto de datos de alquiler, hemos calculado el retorno de la inversión y hemos observado que parece que las propiedades más rentables para comprar y alquilar son aquellas de tamaño pequeño, tipo pareado y ubicadas en barrios económicos.

Sin embargo, hay algunas limitaciones a tener en cuenta en nuestro modelo. Primero, debemos tener en cuenta que el modelo solo puede predecir los precios de venta basados en la información disponible en el dataset de venta. Si se producen cambios significativos en el mercado inmobiliario, estos cambios no se tendrán en cuenta en nuestro modelo. Además, nuestro modelo no tiene en cuenta otros factores que pueden afectar la rentabilidad de una inversión inmobiliaria, como los costos de mantenimiento y reparación.

En cuanto a las vías abiertas, podemos considerar varias opciones para mejorar nuestro modelo. En primer lugar, podríamos ampliar nuestro conjunto de datos con información adicional, como su proximidad a lugares de interés como tiendas, escuelas y transporte público. Además, podríamos utilizar técnicas de aprendizaje automático avanzadas, como redes neuronales, para mejorar aún más nuestra capacidad para predecir los precios de venta de las propiedades.

Nuestro modelo podría ser utilizado por inversores inmobiliarios o personas interesadas en evaluar la rentabilidad de una inversión en una propiedad determinada. Eso sí, teniendo en cuenta que los resultados de nuestro modelo deben ser considerados junto con otros factores relevantes antes de tomar una decisión de inversión.

En conclusión, hemos desarrollado un modelo de machine learning que puede ayudarnos a predecir el precio de venta de una propiedad en Barcelona, y hemos utilizado este modelo para evaluar la rentabilidad de una inversión inmobiliaria. Si bien nuestro modelo tiene limitaciones, predice razonablemente bien el precio de venta de las viviendas y podría ser una herramienta útil para los interesados. En el futuro, se podría considerar la incorporación de información adicional para mejorar aún más nuestra capacidad para predecir los precios de venta de las propiedades en la ciudad de Barcelona.