airbnb

February 22, 2024

[1]:
```
# Librerías
#pip install --upgrade pandas
#!pip install geopandas
#!pip install xgboost
#!pip install lightgbm
#!pip install shap
import geopandas as gpd
import pandas as pd
import numpy as np
import seaborn as sns
import shap
import matplotlib.pyplot as plt
import warnings
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.metrics import mean_absolute_error
warnings.filterwarnings("ignore", "use_inf_as_na")
warnings.simplefilter(action='ignore', category=FutureWarning)
```

```
C:\Users\valde\anaconda3\Lib\site-packages\pandas\core\arrays\masked.py:60:
UserWarning: Pandas requires version '1.3.6' or newer of 'bottleneck' (version
'1.3.5' currently installed).
  from pandas.core import (
```

1 Análisis exploratorio

1.0.1 Datos detallados del calendario para las publicaciones

A continuación se lee y se muestra la cabecera del `dataframe`que muestra los datos detallados del calendario para las publicaciones, y se inspecciona su estructura:

[2]:
```
df_calendar = pd.read_csv('./dataset_nuevo/calendar.csv.gz', sep=',',␣
 ↪parse_dates=['date'], index_col='listing_id')

# Renombra el nombre del índice
```

```
df_calendar.index.name = 'id'

df_calendar.head()
```

[2]:
```
            date available   price adjusted_price  minimum_nights  \
id
6369  2020-01-11         f  $80.00         $80.00             1.0
96072 2020-01-11         f  $25.00         $25.00             3.0
96072 2020-01-12         f  $25.00         $25.00             3.0
96072 2020-01-13         f  $25.00         $25.00             3.0
96072 2020-01-14         f  $25.00         $25.00             3.0

       maximum_nights
id
6369            365.0
96072           365.0
96072           365.0
96072           365.0
96072           365.0
```

[3]:
```
df_calendar.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 7845708 entries, 6369 to 41281944
Data columns (total 6 columns):
 #   Column          Dtype
---  ------          -----
 0   date            datetime64[ns]
 1   available       object
 2   price           object
 3   adjusted_price  object
 4   minimum_nights  float64
 5   maximum_nights  float64
dtypes: datetime64[ns](1), float64(2), object(3)
memory usage: 419.0+ MB
```

Las columnas de `price` y `adjusted_price` tienen el símbolo del dólar. Se va a eliminar este símbolo (al igual que las comas) y a convertir las columnas en numéricas para trabajar con ellas con más facilidad.

[4]:
```
df_calendar['price'] = df_calendar['price'].str.replace("$", '').str.
 ↪replace(',','').astype('float64')
df_calendar['adjusted_price'] = df_calendar['adjusted_price'].str.replace("$",␣
 ↪'').str.replace(',','').astype('float64')
df_calendar.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 7845708 entries, 6369 to 41281944
```

```
Data columns (total 6 columns):
 #   Column          Dtype
---  ------          -----
 0   date            datetime64[ns]
 1   available       object
 2   price           float64
 3   adjusted_price  float64
 4   minimum_nights  float64
 5   maximum_nights  float64
dtypes: datetime64[ns](1), float64(4), object(1)
memory usage: 419.0+ MB
```

Ahora los tipos de datos son correctos. Se inspecciona si el `dataframe` tiene valores nulos:

[5]:
```
df_calendar.isna().sum()
```

[5]:
```
date                0
available           0
price             171
adjusted_price    171
minimum_nights      4
maximum_nights      4
dtype: int64
```

Como el `dataframe` tiene mucha información y muchas filas, se opta por eliminar las filas con valores nulos, ya que no creo que tengan un impacto negativo en el análisis exploratorio:

[6]:
```
df_calendar.dropna(inplace=True)
```

Se va a proceder a obtener una descripción de las columnas numéricas de `df_calendar`. Se muestran los resultados de los cuantiles [0.25, 0.5, 0.75, 0.95, 0.99, 0.995]

[7]:
```
# Solo columnas numéricas
df_calendar.describe(percentiles = [0.25, 0.5, 0.75, 0.95, 0.99, 0.995],␣
 ↪include=np.number)
```

[7]:

	price	adjusted_price	minimum_nights	maximum_nights
count	7.845533e+06	7.845533e+06	7.845533e+06	7.845533e+06
mean	1.423431e+02	1.420381e+02	4.693651e+00	6.443547e+03
std	4.374274e+02	4.361963e+02	2.288716e+01	7.616210e+05
min	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00
25%	4.000000e+01	4.000000e+01	1.000000e+00	4.000000e+01
50%	7.000000e+01	7.000000e+01	2.000000e+00	1.125000e+03
75%	1.200000e+02	1.200000e+02	3.000000e+00	1.125000e+03
95%	4.140000e+02	4.140000e+02	1.200000e+01	1.125000e+03
99%	1.169000e+03	1.169000e+03	5.000000e+01	1.125000e+03
99.5%	1.500000e+03	1.500000e+03	9.000000e+01	1.125000e+03
max	1.023500e+04	1.023500e+04	1.125000e+03	1.111111e+08

Aquí se observa que el precio medio de los alquileres es de 142 dólares (con una desviación estándar de 437), lo que indica que hay una alta variabilidad de precios, y por tanto, una distribución anormal de los precios. Se aprecia omo aunque en el tercer cuartil los precios de los alquileres esté en 120 dólares, el cuantil 99 tiene un valor de 1169 dólares, por lo que hay un gran aumento de los precios a partir del tercer cuartil.

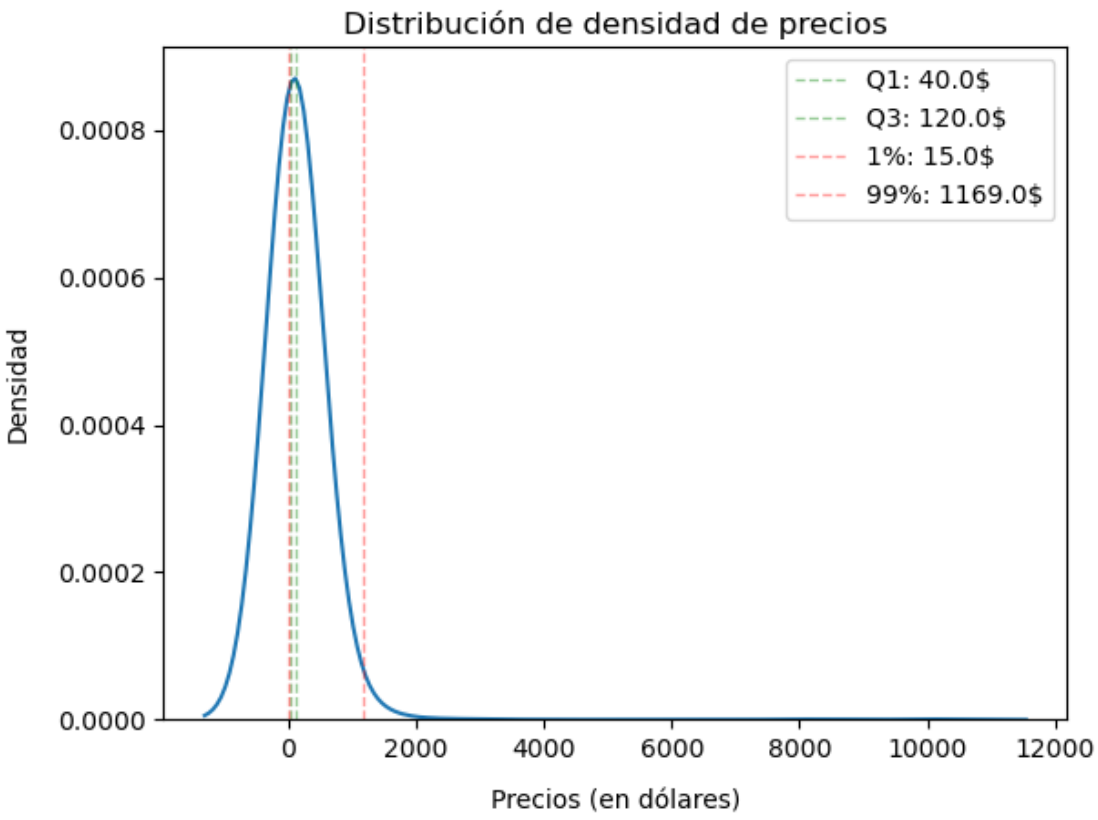
Podemos realizar una estimación de densidad de los precios para ver la distribución de los mismos:

[33]:
```
# Se establece la distribución de densidad de la columna 'price' y con 'bw' el␣
 ↪ancho de banda de suavizado que se usa
ax = sns.kdeplot(data=df_calendar, x="price", bw_method=1)

# Línea vertical para mostrar donde está Q1, Q3, y los percentiles 1% y 99%
plt.axvline(x=np.percentile(df_calendar.price, 25), color='green', ls='--',␣
 ↪lw=1, alpha=0.4, label='Q1: ' +
            str(np.percentile(df_calendar.price, 25)) + '$')
plt.axvline(x=np.percentile(df_calendar.price, 75), color='green', ls='--',␣
 ↪lw=1, alpha=0.4, label='Q3: ' +
            str(np.percentile(df_calendar.price, 75)) + '$')
plt.axvline(x=np.percentile(df_calendar.price, 1), color='red', ls='--', lw=1,␣
 ↪alpha=0.4, label='1%: ' +
            str(np.percentile(df_calendar.price, 1)) + '$')
plt.axvline(x=np.percentile(df_calendar.price, 99), color='red', ls='--', lw=1,␣
 ↪alpha=0.4, label='99%: ' +
            str(np.percentile(df_calendar.price, 99)) + '$')

# Se establece el título, los 'labels' de la gráfica y la leyenda
ax.set_title('Distribución de densidad de precios', pad=5, fontsize=12)
ax.set_xlabel('Precios (en dólares)', labelpad=10, fontsize=10)
ax.set_ylabel('Densidad', labelpad=10, fontsize=10)
plt.legend(loc='best')
```

[33]:
```
<matplotlib.legend.Legend at 0x284214a1f50>
```



En esta gráfica se aprecia que la mayoría de los precios está en torno al rango de los 40-120 dólares por día (primer y tercer cuartil). A precios menores o mayores a este rango, la densidad va disminuyendo, ya que la cantidad de alquileres a precios más bajos que Q1 o a precios más alto que Q3 es menor.

Se comprueban ahora las correlaciones numéricas de las columnas mediante el coeficiente de `Pearson`:

[9]:
```
calendar_target = ['price', 'adjusted_price', 'minimum_nights',␣
 ↪'maximum_nights']

# Se muestra la correlación entre estas cuatro columnas y se colorea cada una␣
 ↪de ellas según su importancia
# con 'DataFrame.style.background_gradient()'
df_calendar[calendar_target].corr(method='pearson').style.background_gradient()
```

[9]:
```
<pandas.io.formats.style.Styler at 0x23a2f9226d0>
```

Se aprecia una correlación positiva muy fuerte entre el precio y el precio ajustado, como se podía preveer al mostrar el `dataframe`. También vemos como no hay correlación ninguna entre los precios y los noches máximas o mínimas de alquiler.

Por último, para este dataframe creo que sería útil ver como evolucionan los precios a medida que pasan las fechas.

[10]:
```
# Se lee de nuevo el dataframe especificando como índice la columna 'date' y␣
 ↪quedándonos solamente con la columna 'price',
# eliminando los símbolos ',' y '$' como se ha realizado arriba
df_calendar_ts = pd.read_csv('./dataset_nuevo/calendar.csv.gz', sep=',',␣
 ↪parse_dates=['date'], index_col='date')[['price']]
df_calendar_ts['price'] = df_calendar_ts['price'].str.replace("$", '').str.
 ↪replace(',','').astype('float64')
df_calendar_ts.head()
```

[10]:

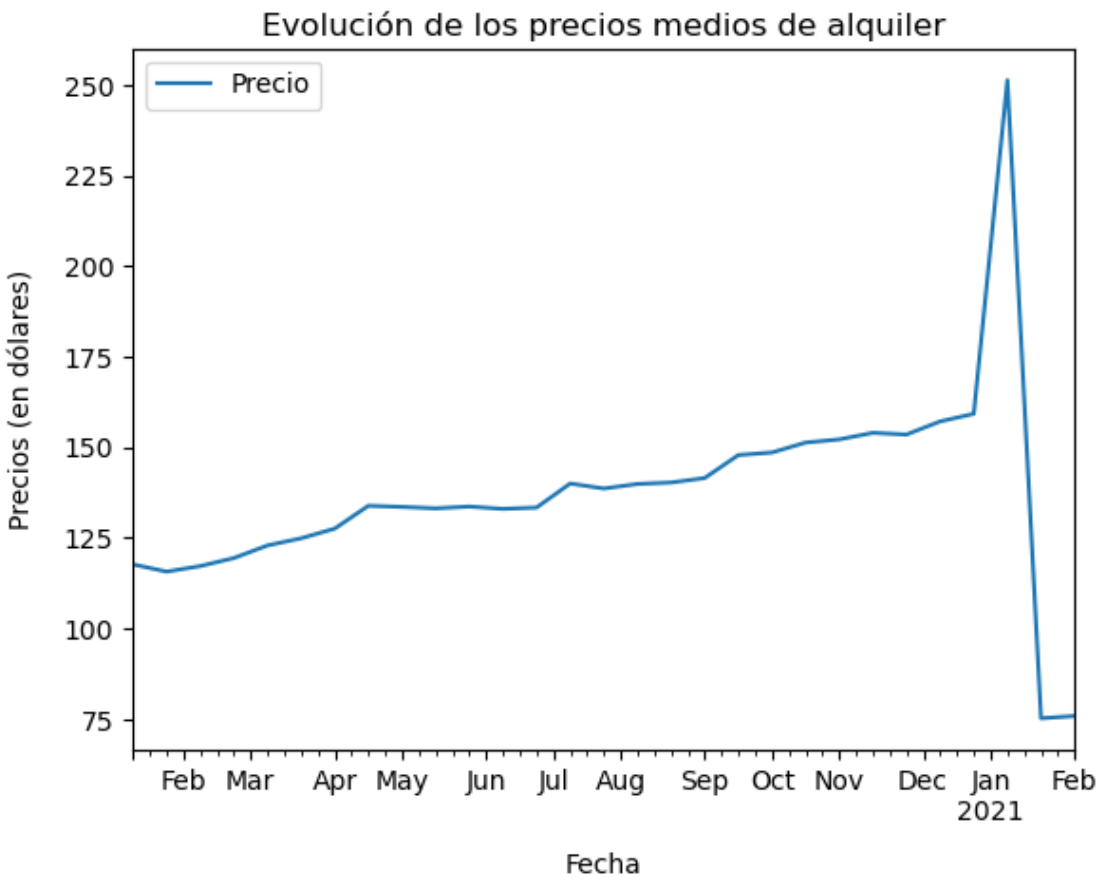
date	price
2020-01-11	80.0
2020-01-11	25.0
2020-01-12	25.0
2020-01-13	25.0
2020-01-14	25.0

Si se agrupa el `dataframe` por fechas (días) y se realiza la media de los precios en cada una de ellas, se puede ver como evolucionan los precios cada día. El resultado sería una serie temporal con muchas variaciones. Para solucionar esto, se hace `downsampling` para tener una menor frecuencia (cada dos semanas en este caso):

[11]:
```
ax = df_calendar_ts.groupby('date').mean().resample(rule='2W').mean().plot()

# Se establece el título y los 'labels' de la gráfica
ax.set_title('Evolución de los precios medios de alquiler', pad=5, fontsize=12)
ax.set_xlabel('Fecha', labelpad=10, fontsize=10)
ax.set_ylabel('Precios (en dólares)', labelpad=10, fontsize=10)
plt.legend(["Precio"])
```

[11]:
```
<matplotlib.legend.Legend at 0x23a1ebe02d0>
```



Como se puede apreciar, los precios han tenido a lo largo de 2020 una tendencia ascendente, con una fuerte subida en diciembre, en épocas navideñas y con ligeros repuntes en julio y septiembre. Con el comienzo del nuevo año y el fin de las épocas festivas, los precios cayeron a un mínimo global en torno a finales de enero de 2021.

1.0.2 Resumen de información y métricas para las publicaciones

A continuación se lee y se muestra la cabecera del `dataframe` que muestra el resumen de información y métricas para las publicaciones y se inspecciona su estructura:

[43]:
```
df_listing_summary = pd.read_csv('./dataset_nuevo/listings.csv', sep=',',␣
↪index_col='id', parse_dates=['last_review'])
df_listing_summary.head()
```

[43]:

id	name	host_id	host_name
6369	Rooftop terrace room , ensuite bathroom	13660	Simon
21853	Bright and airy room	83531	Abdel
23001	Apartmento Arganzuela- Madrid Rio	82175	Jesus
24805	Gran Via Studio Madrid	101471	Iraido

```
24836            Select the Madrid more "cool".   101653     Tenty

      neighbourhood_group   neighbourhood  latitude  longitude  \
id
6369            Chamartín  Hispanoamérica  40.45628   -3.67763
21853              Latina        Cármenes  40.40341   -3.74084
23001          Arganzuela         Legazpi  40.38695   -3.69304
24805              Centro     Universidad  40.42202   -3.70395
24836              Centro        Justicia  40.41995   -3.69764

             room_type  price  minimum_nights  number_of_reviews last_review  \
id
6369      Private room     70               1                 73  2019-12-13
21853     Private room     17               4                 33  2018-07-15
23001  Entire home/apt     50              15                  0         NaT
24805  Entire home/apt     80               5                  9  2020-01-03
24836  Entire home/apt    115               3                 67  2019-12-08

       reviews_per_month  calculated_host_listings_count  availability_365
id
6369                0.61                               1               82
21853               0.52                               2              162
23001                NaN                               6              213
24805               0.14                               1              362
24836               0.64                               1              342
```

[44]:
```
df_listing_summary.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 21495 entries, 6369 to 41452557
Data columns (total 15 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   name                            21492 non-null  object
 1   host_id                         21495 non-null  int64
 2   host_name                       21470 non-null  object
 3   neighbourhood_group             21495 non-null  object
 4   neighbourhood                   21495 non-null  object
 5   latitude                        21495 non-null  float64
 6   longitude                       21495 non-null  float64
 7   room_type                       21495 non-null  object
 8   price                           21495 non-null  int64
 9   minimum_nights                  21495 non-null  int64
 10  number_of_reviews               21495 non-null  int64
 11  last_review                     17204 non-null  datetime64[ns]
 12  reviews_per_month               17204 non-null  float64
 13  calculated_host_listings_count  21495 non-null  int64
```

```
 14  availability_365                21495 non-null  int64
dtypes: datetime64[ns](1), float64(3), int64(6), object(5)
memory usage: 2.6+ MB
```

Todas las columnas tienen correctamente su tipo establecido. Se pasa a mostrar la correlación entre todas las variables numéricas:

[45]:
```
target = ['latitude', 'longitude', 'price', 'minimum_nights',␣
 ↪'number_of_reviews', 'last_review', 'reviews_per_month',
        'calculated_host_listings_count', 'availability_365']

# Se muestra la correlación entre estas cuatro columnas y se colorea cada una␣
 ↪de ellas según su importancia
# con 'DataFrame.style.background_gradient()'
df_listing_summary[target].corr(method='pearson').style.background_gradient()
```

[45]:
```
<pandas.io.formats.style.Styler at 0x169d85fe590>
```

Se puede ver como hay una correlación positiva alta entre el número de `reviews` y las `reviews` por mes. También hay correlación positiva entre la última reseña y las resñas por mes, y entre la última reseña y el número de `reviews`realizadas. Por otro lado, se puede ver que hay cierta correlación positiva entre la longitud y la latitud. En cuánto al precio, no parece que haya ninguna variable numérica que influya en ella positiva o negativamente de forma significativa.

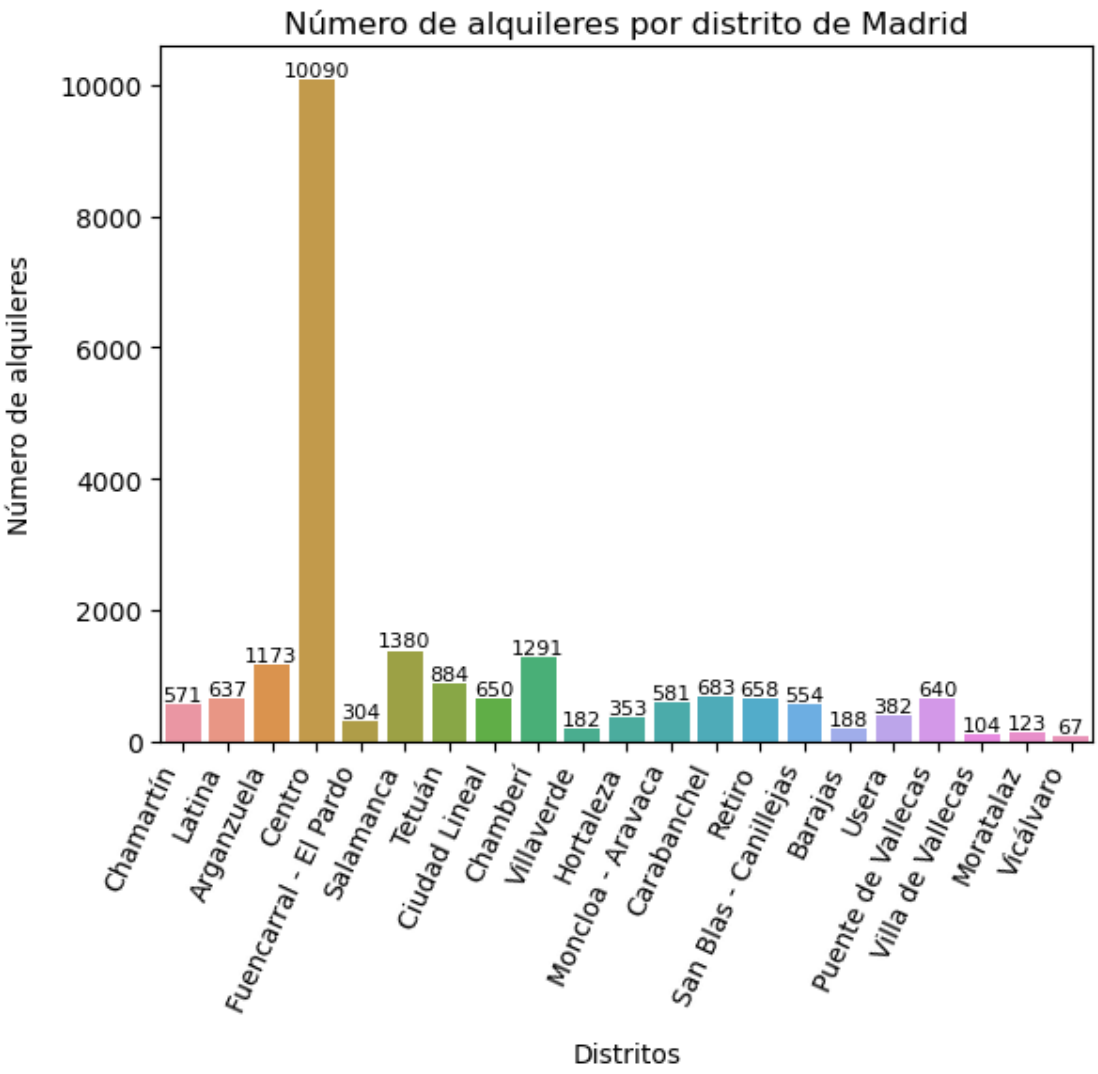
Ahora nos centramos en los distritos de la ciudad, y es que en este `dataframe` aparece información sobre los distritos de Madrid. Primero vemos el número de alquileres que hay por distrito:

[59]:
```
df_listing_summary['neighbourhood_group'].value_counts()

# Cuenta de observaciones
ax = sns.countplot(data=df_listing_summary, x='neighbourhood_group')

# Se muestra la cuenta arriba de cada barra
ax.bar_label(ax.containers[0], fontsize=8)

# Se establece el título, los 'ticks' en el eje X y los 'labels' de la gráfica
plt.title('Número de alquileres por distrito de Madrid', pad=5, fontsize=12)
plt.xlabel('Distritos', labelpad=10, fontsize=10)
plt.ylabel('Número de alquileres', labelpad=10, fontsize=10)
plt.xticks(rotation = 65, ha='right');
```



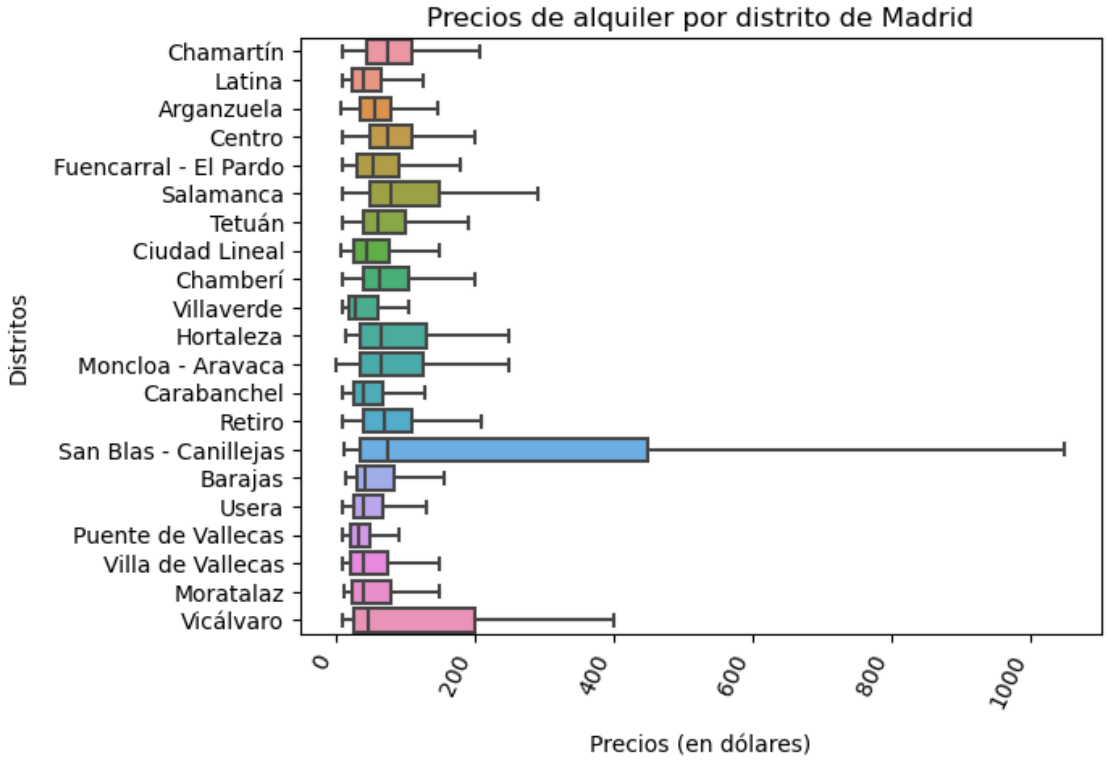
Centro, Salamanca, Chamberí y Arganzuela son los distritos donde más alquileres hay (principalmente en Centro, donde hay más de 10000 alquileres). Por el contrario, en Vicálvaro, Villa de Vallecas o Moratalaz son los distritos donde menos alquileres hay (menos de 150 alquileres en todos estos distritos).

Para profundizar más en este aspecto, se realiza un diagrama de cajas para ver como se distribuye el precio en cada uno de estos distritos (sin valores atípicos, para ver la gráfica con más claridad):

[19]:
```
# Diagrama de cajas sin outliers (showfliers es False)
sns.boxplot(data=df_listing_summary, x='price', y='neighbourhood_group',␣
 ↪showfliers=False)

# Se establece el título, los 'ticks' en el eje X y los 'labels' de la gráfica
plt.title('Precios de alquiler por distrito de Madrid', pad=5, fontsize=12)
```

```
plt.xlabel('Precios (en dólares)', labelpad=10, fontsize=10)
plt.ylabel('Distritos', labelpad=10, fontsize=10)
plt.xticks(rotation = 65, ha='right');
```



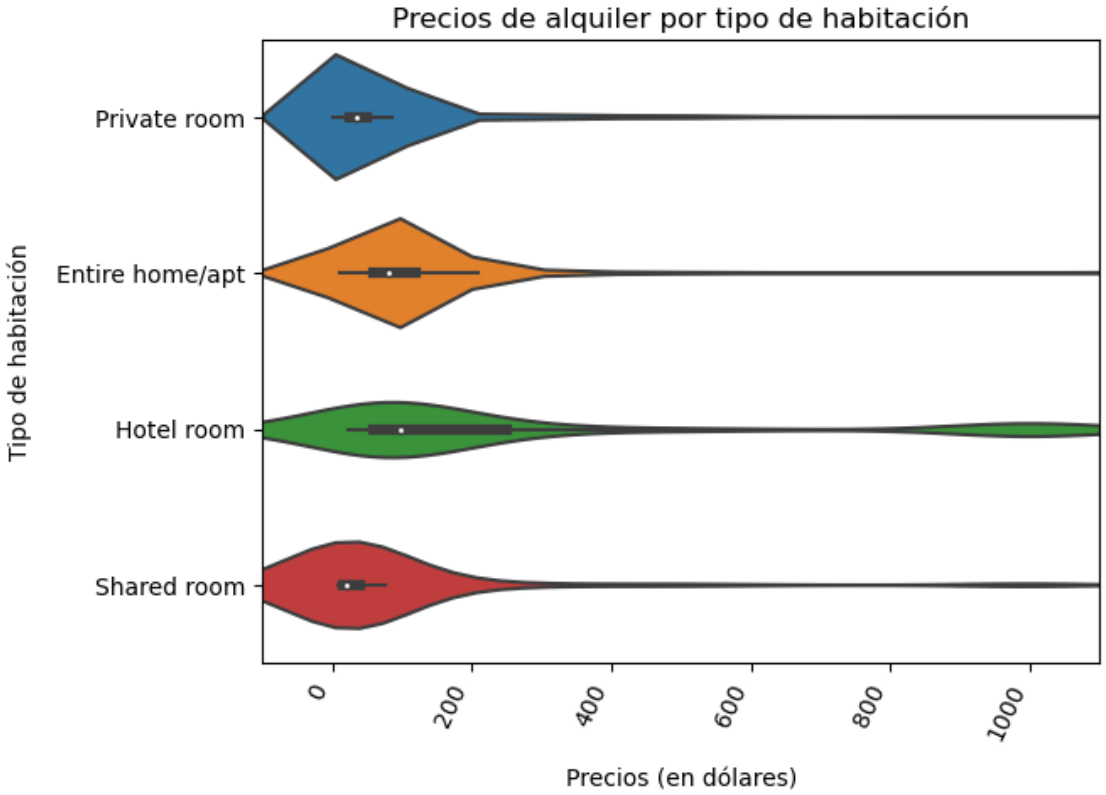
Como se puede ver en el diagrama de cajas, según las medianas los precios más altos están en distritos como los de San Blas - Canillejas, Salamanca o Chamartín. En cambio, los más bajos están en distritos como Carabanchel, Villaverde o Puente de Vallecas. Por otra parte, se aprecia claramente que el precio máximo de alquiler (sin valores atípicos) está en el distrito de San Blas - Canillejas, con un precio de más de 1000 dólares. Por último, hay que destacar los rangos intercuartílicos del propio distrito de San Blas - Canillejas y del distrito de Vicálvaro. Son rangos muy grandes, lo que hace indicar que la variabilidad de precios en esos distritos es muy alta.

También puede ser interesante ver la distribución de los precios en cada uno de los tipos de habitaciones ofertados. Ya que dependiendo del tipo de habitación que se quiere alquilar, ésta se puede poner en alquiler a mayor o menor precio. Para verlo gráficamente realizamos un diagrama de violines:

[20]:
```
# Diagrama de violines (limitando el eje X para ver mejor la gráfica)
ax = sns.violinplot(data=df_listing_summary, x='price', y='room_type',␣
  ↪inner='box')
ax.set(xlim=(-100, 1100))

# Se establece el título, los 'ticks' en el eje X y los 'labels' de la gráfica
```

```
plt.title('Precios de alquiler por tipo de habitación', pad=5, fontsize=12)
plt.xlabel('Precios (en dólares)', labelpad=10, fontsize=10)
plt.ylabel('Tipo de habitación', labelpad=10, fontsize=10)
plt.xticks(rotation = 65, ha='right');
```

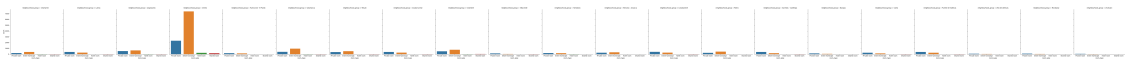


En esta ocasión se puede ver como los precios más caros son de habitaciones de hoteles y de casas/apartamentos enteros, mientras que los precios más bajos son de habitaciones privadas y habitaciones compartidas. También se aprecia como la mayor variabilidad de precios se da en habitaciones de hotel, ya que su rango intercuartílico es el mayor de todos.

Otro análisis interesante para ver como se encuentra actualmente el mercado de alquileres en Madrid, sería ver una distribución del tipo de alquileres según el distrito en el que nos encontremos.

[21]:
```
sns.catplot(data=df_listing_summary, x='room_type', col =␣
↪'neighbourhood_group', kind='count')
```

[21]: <seaborn.axisgrid.FacetGrid at 0x1f815258890>



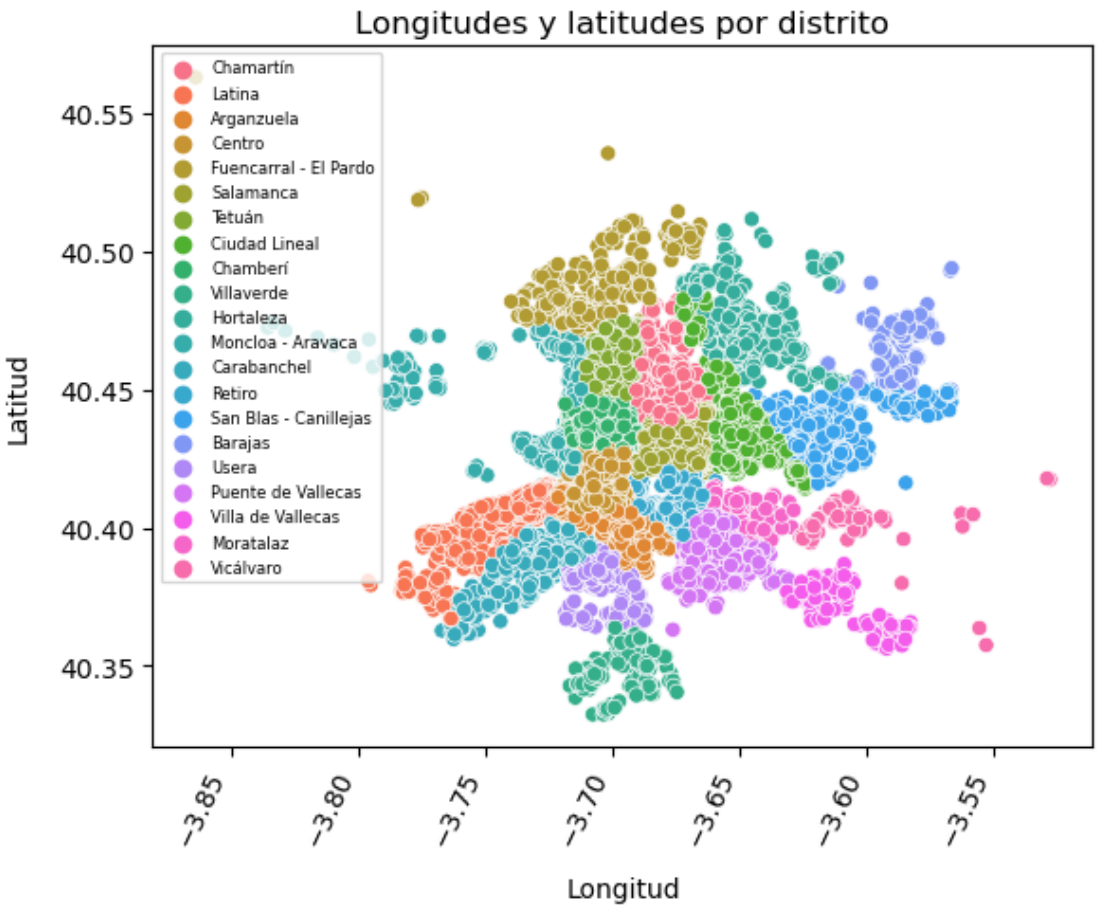
De forma general, lo que más se ofrece son casas/apartamentos completos o habitaciones privadas. En la mayoría de ellos, la mayor oferta es de casas/apartamentos completos. Solo hay unos pocos distritos donde se ofrecen mayor cantidad de habitaciones privadas que apartamentos enteros, como en Latina, El Pardo, Ciudad Lineal, Carabanchel o Moratalaz. Las habitaciones de hotel y las habitaciones compartidas realmente tienen poquísimas ofertas en todo Madrid.

También hay disponibles en el `dataframe` datos sobre la longitud y la latitud de los alquileres. Sería interesante ver un gráfico de dispersión de estas dos variables según el distrito al que pertenecen para ver si latitudes y longitudes parecidas pertenecen al mismo distrito:

[22]:
```
# Diagrama de dispersión
ax = sns.scatterplot(data=df_listing_summary, x="longitude", y="latitude",␣
 ↪hue="neighbourhood_group")

# Se establece el título. los 'ticks' en el eje X, los 'labels' y la leyenda de␣
 ↪la gráfica (reduciendo su tamaño)
plt.title('Longitudes y latitudes por distrito', pad=5, fontsize=12)
plt.xlabel('Longitud', labelpad=10, fontsize=10)
plt.ylabel('Latitud', labelpad=10, fontsize=10)
plt.xticks(rotation = 65, ha='right');
plt.legend(handles=ax.get_legend_handles_labels()[0],
           fontsize="6",
           loc='best')
```

[22]:
```
<matplotlib.legend.Legend at 0x1f815b620d0>
```



Efectivamente, longitudes y latitudes cercanas pertenecen al mismo distrito. Esto tiene sentido, ya que, evidentemente, dependiendo de la localización geográfica tendremos unas longitudes y latitudes distintos, y por cada distrito, estas longitudes y latitudes deben ser muy parecidas o próximas entre ellas.

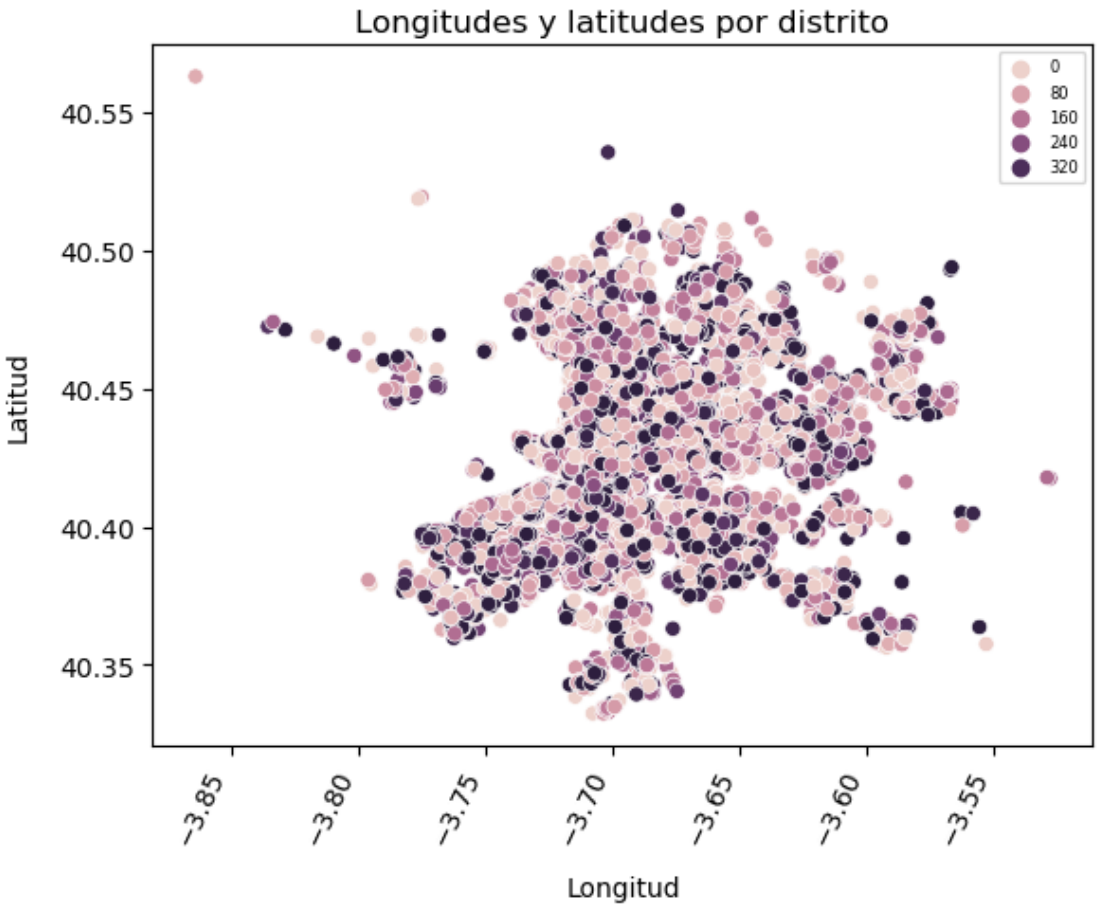
Se puede hacer otro gráfico de dispersión con la longitud y la latitud, pero esta vez agrupando por la disponibilidad de los alquileres al día durante un periodo anual:

[25]:
```
# Diagrama de dispersión
ax = sns.scatterplot(data=df_listing_summary, x="longitude", y="latitude",␣
 ↪hue="availability_365")

# Se establece el título. los 'ticks' en el eje X, los 'labels' y la leyenda de␣
 ↪la gráfica (reduciendo su tamaño)
plt.title('Longitudes y latitudes por distrito', pad=5, fontsize=12)
plt.xlabel('Longitud', labelpad=10, fontsize=10)
plt.ylabel('Latitud', labelpad=10, fontsize=10)
plt.xticks(rotation = 65, ha='right');
plt.legend(handles=ax.get_legend_handles_labels()[0],
```

```
            fontsize="6",
            loc='best')
```

[25]: <matplotlib.legend.Legend at 0x1f81c6c3a10>

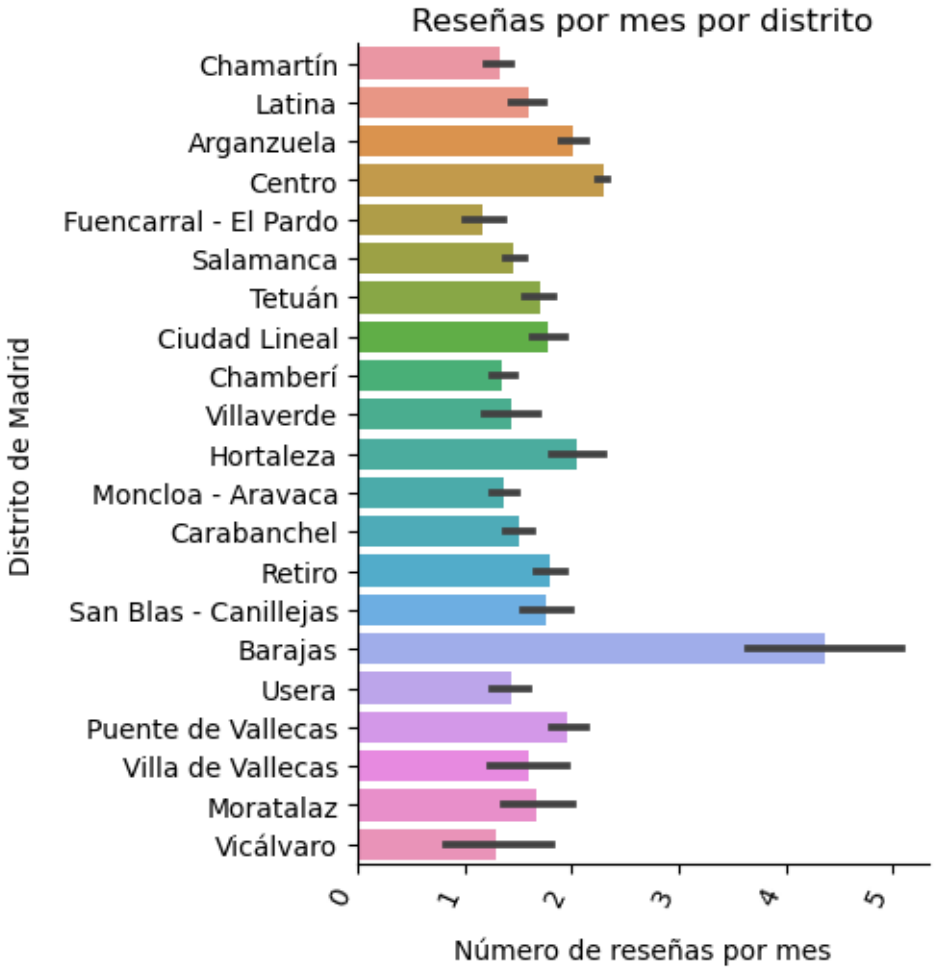


Se aprecia que no hay correlación entre las variables de longitud y latitud y la disponibilidad anual de los alquileres. Los puntos de varios colores se reparten a lo largo de todo el gráfico, por lo que la disponibilidad de los alquileres es muy diversa, pero se reparte por todas las longitudes y latitudes de la ciudad de Madrid.

Ahora se va a analizar de forma general las reseñas realizadas en `Airbnb`. Una gráfica interesante sería ver las reseñas por mes que se han realizado por distrito, ya que en los distritos donde mayor ratio de reseñas se realiza nos indicaría que muy probablemente habrá más huéspedes que se hospenden por esas zobnas:

[20]:
```
# Diagrama de barras
sns.catplot(data=df_listing_summary, x='reviews_per_month',␣
 ↪y='neighbourhood_group', kind='bar')
```

```
# Se establece el título. los 'ticks' en el eje X y los 'labels' de la gráfica
plt.title('Reseñas por mes por distrito', pad=5, fontsize=12)
plt.xlabel('Número de reseñas por mes', labelpad=10, fontsize=10)
plt.ylabel('Distrito de Madrid', labelpad=10, fontsize=10)
plt.xticks(rotation = 65, ha='right');
```



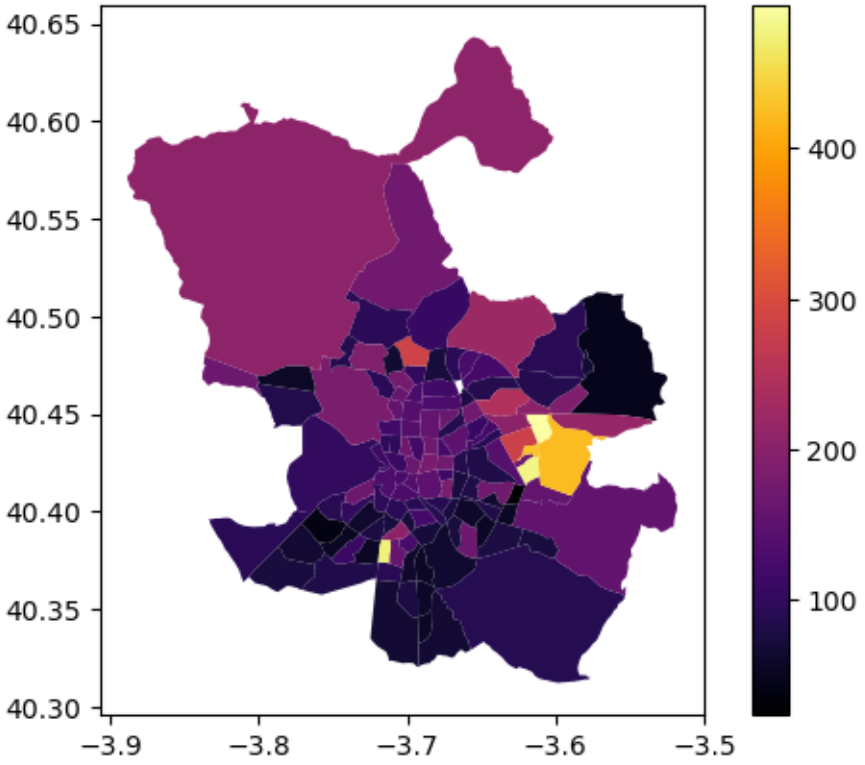
En el distrito de Barajas es dónde más reseñas por mes de media se realizan con bastante diferencia (más de 4 reseñas por mes) a pesar de que se ha visto anteriormente que es uno de los distritos donde menos alquileres ofertados hay. Esto indica que los huéspedes que se hospedan en este distrito es muy probable que escriban alguna reseña. Les siguen en este sentido distritos como Centro (el que más oferta de alquileres tiene) y Arganzuela, con más de 2 reseñas de media por mes. Por el lado contrario, tenemos distritos como Vicálvaro y Fuencarral - El Pardo, con menos de dos reseñas por mes.

Ahora pasamos a analizar los datos geoespaciales. Para empezar, se va a realizar un mapa coroplético que muestre el precio medio del alquiler en cada uno de los barrios de Madrid, así se tiene una idea de a cuánto puede estar el precio medio de alquiler por barrios:

[21]:
```
# Archivo GeoJSON con todos los barrios de Madrid
geojson = gpd.read_file('./dataset_nuevo/neighbourhoods.geojson')

# Se fusiona el archivo GeoJSON con las dos columnas de 'df_listing_summary',␣
 ↪agrupando el GeoDataframe
# por barrio, calculando la media de los precios. Por último se dibuja el mapa␣
 ↪coroplético
geojson.merge(df_listing_summary[['price', 'neighbourhood']], on =␣
 ↪'neighbourhood') \
                .dissolve(by='neighbourhood', aggfunc={"price": "mean"}) \
                .plot('price', cmap = 'inferno', legend=True)
```

[21]: <Axes: >

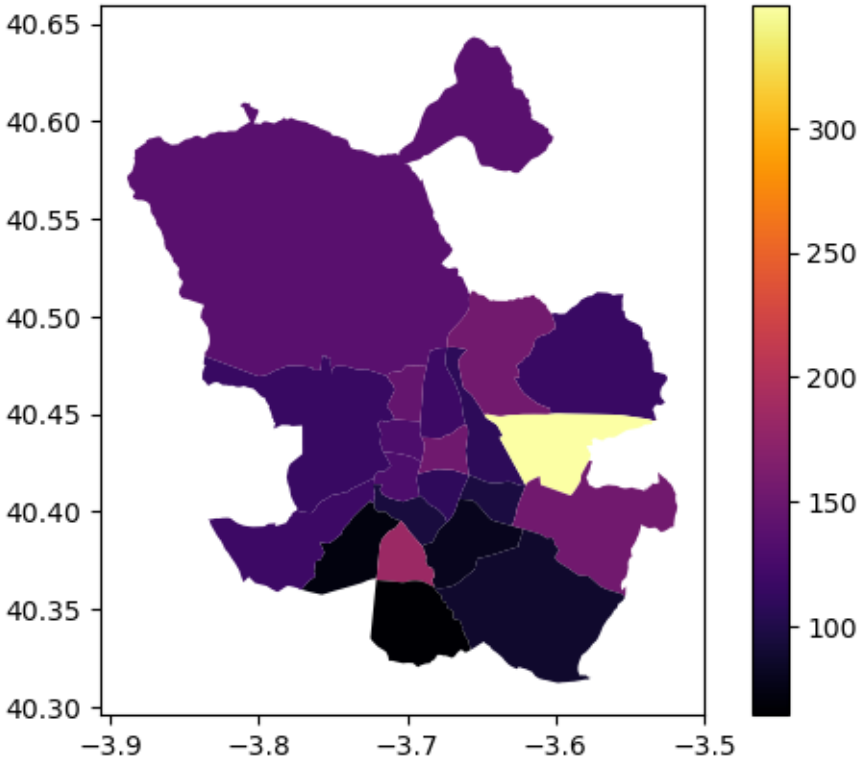


La información mostrada aquí muestra que los precios de alquiler medios más caros están en barrios como Canillejas, Arcos o Zofío, dónde se llegan a alcanzar los 400 dólares. Por otro lado, los precios medios más baratos se encuentran en barrios como Horcajo, Aluche o Vista Alegre, lugares dónde el alquiler medio está por debajo de los 100 dólares.

Ahora se procede a realizar lo mismo, pero esta vez con los distritos de Madrid. El resultado es el siguiente:

[22]:
```
# Se fusiona el archivo GeoJSON con las dos columnas de 'df_listing_summary',␣
 ↪agrupando el GeoDataframe
# por distrito, calculando la media de los precios. Por último se dibuja el␣
 ↪mapa coroplético
geojson.merge(df_listing_summary[['price', 'neighbourhood_group']], on =␣
 ↪'neighbourhood_group') \
                .dissolve(by='neighbourhood_group', aggfunc={"price": "mean"}) \
                .plot('price', cmap = 'inferno', legend=True)
```

[22]: <Axes: >



Como se puede apreciar, el precio medio más alto está en el distrito de San Blas - Canilleja, que está en torno a los 340 dólares. Por el contrario, los precios más bajos se encuentran en distritos como Villaverde o Carabanchel, dónde los alquileres están por debajo de los 100 dólares. Como se puede comprobar, la información obtenida en este mapa se parece bastante a la obtenida en el diagrama de cajas realizado anteriormente en los distritos de Madrid.

2 Predicción

En esta parte del `notebook` se va a intentar entrenar un modelo que sea capaz de predecir el precio de alquiler diario de una propiedad de `Airbnb`.

[8]:
```
df_listing = pd.read_csv('./dataset_nuevo/listings.csv.gz', sep=',',␣
↪index_col='id', low_memory=False)
```

Las columna objetiva price tiene el símbolo del dólar. Además también hay otras columnas monetarias con el mismo problema. Por lo tanto, se va a eliminar este símbolo (al igual que las comas) y a convertir las columnas en numéricas para trabajar con ellas con más facilidad.

[9]:
```
df_listing['price'] = df_listing['price'].str.replace("$", '').str.
↪replace(',','').astype('float64')
df_listing['cleaning_fee'] = df_listing['cleaning_fee'].str.replace("$", '').
↪str.replace(',','').astype('float64')
df_listing['security_deposit'] = df_listing['security_deposit'].str.
↪replace("$", '').str.replace(',','').astype('float64')
df_listing['extra_people'] = df_listing['extra_people'].str.replace("$", '').
↪str.replace(',','').astype('float64')
df_listing['monthly_price'] = df_listing['monthly_price'].str.replace("$", '').
↪str.replace(',','').astype('float64')
df_listing['weekly_price'] = df_listing['weekly_price'].str.replace("$", '').
↪str.replace(',','').astype('float64')
```

Este `dataframe` contiene demasiadas columnas. Se comprueba con el coeficiente de Pearson las variables numéricas que más pueden influir en el precio para correlaciones positivas (obviamente sin incluir las variable de precio):

[10]:
```
df_listing.corr(numeric_only=True, method='pearson')['price'].nlargest(n=21)[3:]
```

[10]:
```
square_feet                                   0.312792
accommodates                                  0.105941
cleaning_fee                                  0.098121
beds                                          0.094738
availability_30                               0.093308
bedrooms                                      0.092216
availability_60                               0.083132
availability_90                               0.076761
host_id                                       0.068562
availability_365                              0.061795
security_deposit                              0.061578
calculated_host_listings_count                0.057652
longitude                                     0.051922
calculated_host_listings_count_entire_homes   0.051521
calculated_host_listings_count_private_rooms  0.042715
latitude                                      0.037647
bathrooms                                     0.033799
guests_included                               0.031503
Name: price, dtype: float64
```

Del mismo modo, se comprueba con el coeficiente de Pearson las variables numéricas que más pueden influir en el precio para correlaciones negativas:

[11]:
```
df_listing.corr(numeric_only=True, method='pearson')['price'].nsmallest(n=10)
```

[11]:
```
number_of_reviews_ltm                          -0.081587
number_of_reviews                              -0.074793
reviews_per_month                              -0.041010
calculated_host_listings_count_shared_rooms    -0.016677
review_scores_communication                    -0.016062
maximum_minimum_nights                         -0.015253
minimum_nights_avg_ntm                         -0.015085
minimum_nights                                 -0.014658
review_scores_rating                           -0.013410
review_scores_checkin                          -0.013254
Name: price, dtype: float64
```

Como se puede apreciar, no hay muchas variables que tengan gran correlación con el precio de una casa: * Dentro de las correlaciones positivas variables, hay algunas que se deben eliminar, pues el cliente parte de una situación de partida a la hora de alquilar viviendas. Por tanto, se eliminará `host_id`, ya que es el identificador de los anfitriones de los alquileres; y las demás variables de `host`, ya que al principio no se tendría esa información. * En las correlaciones negativas, las variables de `reviews` no se pueden incluir puesto que al principio el cliente no tiene información de reseñas acerca de su casa. En cuánto a las demás variables, la correlación es pequeñisima, y por tanto, no se incluyen en la selección de características finales.

Se guardan los nombres de las variables numéricas elegidas en un `array`:

[12]:
```
# Por cada variable de las correlaciones más altas, guarda las que no tengan el␣
 ↪string 'host'
numerical_columns = [variable for variable in df_listing.
 ↪corr(numeric_only=True, method='pearson')['price'].nlargest(n=21)[3:].index
                     if 'host' not in variable]
numerical_columns
```

[12]:
```
['square_feet',
 'accommodates',
 'cleaning_fee',
 'beds',
 'availability_30',
 'bedrooms',
 'availability_60',
 'availability_90',
 'availability_365',
 'security_deposit',
 'longitude',
 'latitude',
 'bathrooms',
 'guests_included']
```

Una vez analizaas las columnas numéricas, se imprime en una lista las variables no numéricas para

ver cuáles se tienen:

[13]:
```
# Columnas con variables no numéricas
df_listing.select_dtypes(exclude=np.number).columns
```

[13]:
```
Index(['listing_url', 'last_scraped', 'name', 'summary', 'space',
       'description', 'experiences_offered', 'neighborhood_overview', 'notes',
       'transit', 'access', 'interaction', 'house_rules', 'picture_url',
       'host_url', 'host_name', 'host_since', 'host_location', 'host_about',
       'host_response_time', 'host_response_rate', 'host_is_superhost',
       'host_thumbnail_url', 'host_picture_url', 'host_neighbourhood',
       'host_verifications', 'host_has_profile_pic', 'host_identity_verified',
       'street', 'neighbourhood', 'neighbourhood_cleansed',
       'neighbourhood_group_cleansed', 'city', 'state', 'zipcode', 'market',
       'smart_location', 'country_code', 'country', 'is_location_exact',
       'property_type', 'room_type', 'bed_type', 'amenities',
       'calendar_updated', 'has_availability', 'calendar_last_scraped',
       'first_review', 'last_review', 'requires_license', 'license',
       'instant_bookable', 'is_business_travel_ready', 'cancellation_policy',
       'require_guest_profile_picture', 'require_guest_phone_verification'],
      dtype='object')
```

Aquí se eliminan columnas con `URL`, ya que no aportan información; las de ciudad (todo es Madrid); código postal, país, ya que no aportan información o ésta es redundante; y las de `host`, reseñas, o calendarios, ya que no tendríamos esa información al principio para los nuevos alquileres. Por lo tanto, se eligen las columnas de tipos de propiedad, barrios, distritos, comodidades, que ya se han visto en el análisis exploratorio de datos que pueden ser útiles para predecir el precio de alquileres, y la variable objetivo `price`. Al final, el dataframe resultante con las columnas elegidas (numéricas y categóricas) sería el siguiente:

[14]:
```
df_listing_reduced = df_listing[numerical_columns + ['neighbourhood',␣
↪'neighbourhood_group_cleansed',
                                                     'property_type',␣
↪'room_type', 'amenities', 'price']]
```

Se muestra el nuevo `dataframe` por pantalla:

[15]:
```
df_listing_reduced.head()
```

[15]:

id	square_feet	accommodates	cleaning_fee	beds	availability_30	\
6369	172.0	2	5.0	0.0	22	
21853	97.0	1	NaN	1.0	0	
23001	1184.0	6	30.0	5.0	2	
24805	0.0	3	30.0	1.0	27	
24836	NaN	4	0.0	3.0	24	

	bedrooms	availability_60	availability_90	availability_365	\

```
id
6369       1.0              52              82               82
21853      1.0               0               0              162
23001      3.0               2               2              213
24805      0.0              57              87              362
24836      2.0              54              77              342

       security_deposit  longitude  latitude  bathrooms  guests_included  \
id
6369                0.0   -3.67763  40.45628        1.0                2
21853               NaN   -3.74084  40.40341        1.0                1
23001             300.0   -3.69304  40.38695        2.0                1
24805             200.0   -3.70395  40.42202        1.0                2
24836             200.0   -3.69764  40.41995        1.5                4

      neighbourhood neighbourhood_group_cleansed property_type  \
id
6369      Chamartín                    Chamartín     Apartment
21853        Aluche                       Latina     Apartment
23001       Legazpi                    Arganzuela     Apartment
24805      Malasaña                       Centro     Apartment
24836      Justicia                       Centro     Apartment

            room_type                                          amenities  \
id
6369      Private room  {Wifi,"Air conditioning",Kitchen,Elevator,Heat…
21853     Private room  {TV,Internet,Wifi,"Air conditioning",Kitchen,"…
23001  Entire home/apt  {TV,Internet,Wifi,"Air conditioning","Wheelcha…
24805  Entire home/apt  {TV,Internet,Wifi,"Air conditioning",Kitchen,E…
24836  Entire home/apt  {TV,"Cable TV",Internet,Wifi,"Air conditioning…

       price
id
6369    70.0
21853   17.0
23001   50.0
24805   80.0
24836  115.0
```

Una vez elegidas las columnas, se van a repartir por un lado las columnas numéricas, guardándolas en una variable; y por otro lado las columnas categóricas, guardándolas en otra variable:

[16]:
```
# Variables categóricas (excluye tipos np.number)
categorical_df_columns = df_listing_reduced.select_dtypes(exclude=np.number).
 ↪columns

# Variables numéricas (incluye tipos np.number)
```

```
numerical_df_columns = df_listing_reduced.select_dtypes(include=np.number).
 ↪columns

print('Categóricas: ', categorical_df_columns)
print('\nNuméricas: ', numerical_df_columns)
```

```
Categóricas:  Index(['neighbourhood', 'neighbourhood_group_cleansed',
'property_type',
       'room_type', 'amenities'],
      dtype='object')

Numéricas:  Index(['square_feet', 'accommodates', 'cleaning_fee', 'beds',
       'availability_30', 'bedrooms', 'availability_60', 'availability_90',
       'availability_365', 'security_deposit', 'longitude', 'latitude',
       'bathrooms', 'guests_included', 'price'],
      dtype='object')
```

En relación a las columnas categóricas hay que tener en cuenta dos aspectos: el número de valores que puede tomar cada una; y si existe una relación de orden entre estos valores. Estos factores determinan el tipo de transformación que se ha de hacer:

- Si la columna tiene dos valores se puede binarizar directamente.
- Si por el contrario, tiene más de dos valores sin un orden entre ellos, se aplica la técnica `One Hot Encoding`, la cual crea una columna binaria para cada uno de dichos valores.
- Si la columna categórica tiene una relación de orden, los valores se transforman a numéricos, sustituyendo cada valor por su orden.

A continuación, se obtiene el número de valores para cada una de las variables categóricas:

[17]:
```
# Se imprime el nombre y el número de valores para cada columna de la lista de␣
 ↪variables categóricas
num_values_categ_df_col = list(map(lambda col: (col,␣
 ↪len(df_listing_reduced[col].value_counts())), categorical_df_columns))
num_values_categ_df_col
```

[17]:
```
[('neighbourhood', 67),
 ('neighbourhood_group_cleansed', 21),
 ('property_type', 31),
 ('room_type', 4),
 ('amenities', 19182)]
```

Como se puede ver, las columnas categóricas tienen todas más de dos valores, y además, no tienen ningún orden entre ellos. Por tanto, se debe aplicar la técnica de `One Hot Encoding` a todas ellas para crear una columna binaria para cada uno de los valores que pueden presentar estas columnas. Sin embargo, hay una columna que realmente debe tratarse distinta, y esta columna es `amenities`, ya que tiene una grandísima cantidad de valores.

Como se ha podido ver en el `dataframe`, la columna `amenities` es un conjunto con todas las comodidades de cada alquiler. Para transformar esta variable se puede usar `MultiLabelBinarizer`,

pero suceden dos cosas: me da problemas al insertarlo posteriormente en los `pipelines` posteriores; y segundo, se crearían demasiadas columna, ya que se crearía una para cada una de las palabras únicas que aparecen en toda la columna (por lo que se crearían más de 150 columnas binarias). Debido a estos dos motivos, se decide sustituir cada conjunto de la columna por un número entero que simbolice el número de comodides totales que tiene cada alquiler.

Para ello se eliminan los símbolos `{` y `}` de cada conjunto, y se convierte a una lista. Una vez construida la lista para cada fila, se eliminan los elementos que contienen el `substring` 'translation', ya que no se trata de comodidades y se calcula finalmente el número de elementos de la lista.

[18]:
```
# Reemplazo de valores y conversión a lista cada fila de la columna 'amenities'
df_listing_reduced.loc[:, 'amenities'] = df_listing_reduced['amenities'].str.
 ↪replace("{", "") \
                        .str.replace("}", "") \
                        .str.replace('"', "") \
                        .str.split(',')

# Para cada fila de la columna 'amenities' se aplica una función: de la lista␣
 ↪de comodidades, se eliminan las
# que tengan el subtring 'translation' porque estos elementos de la lista no␣
 ↪son comodidades
df_listing_reduced.loc[:, 'amenities'] = df_listing_reduced['amenities'].
 ↪apply(lambda x: [element for element in x if 'translation' not in element])

# Para cada fila de la columna 'amenities', se cuenta el número total de␣
 ↪comodidades
df_listing_reduced.loc[:, 'amenities'] = df_listing_reduced['amenities'].
 ↪apply(lambda x: len(x))
```

Una vez se han seleccionado las columnas y se ha transformado la columna `amenities`, se imprime por pantalla la información sobre cada tipo de las columnas del `dataframe`:

[19]:
```
df_listing_reduced.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 21495 entries, 6369 to 41452557
Data columns (total 20 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   square_feet                 301 non-null    float64
 1   accommodates                21495 non-null  int64
 2   cleaning_fee                16270 non-null  float64
 3   beds                        21356 non-null  float64
 4   availability_30             21495 non-null  int64
 5   bedrooms                    21479 non-null  float64
 6   availability_60             21495 non-null  int64
 7   availability_90             21495 non-null  int64
 8   availability_365            21495 non-null  int64
```

```
 9   security_deposit              14440 non-null  float64
 10  longitude                     21495 non-null  float64
 11  latitude                      21495 non-null  float64
 12  bathrooms                     21480 non-null  float64
 13  guests_included               21495 non-null  int64
 14  neighbourhood                 21361 non-null  object
 15  neighbourhood_group_cleansed  21495 non-null  object
 16  property_type                 21495 non-null  object
 17  room_type                     21495 non-null  object
 18  amenities                     21495 non-null  object
 19  price                         21495 non-null  float64
dtypes: float64(9), int64(6), object(5)
memory usage: 3.4+ MB
```

Como se ve en la tabla, prácticamente todas las columnas tienen el tipo correcto. Solo habría que cambiar el tipo de la variable `amenities`, que tras las transformaciones realizadas arribas, ya no es una variable de tipo objeto, sino una variable de tipo entero:

[20]:
```
df_listing_reduced = df_listing_reduced.astype({'amenities': np.int64})
```

Una vez transformado al tipo adecuado, se elimina la variable `amenities` de la lista de variables categóricas:

[21]:
```
# Se elimina la columna 'amenities' de la lista de variables categóricas
categorical_columns = categorical_df_columns.to_list()
categorical_columns.remove('amenities')
categorical_columns
```

[21]:
```
['neighbourhood', 'neighbourhood_group_cleansed', 'property_type', 'room_type']
```

Y se une a la lista de variables numéricas:

[22]:
```
# Se añade la columna 'amenities' a la lista de variables numéricas
numerical_columns = numerical_df_columns.to_list()
numerical_columns.append('amenities')
numerical_columns
```

[22]:
```
['square_feet',
 'accommodates',
 'cleaning_fee',
 'beds',
 'availability_30',
 'bedrooms',
 'availability_60',
 'availability_90',
 'availability_365',
 'security_deposit',
 'longitude',
 'latitude',
```

```
 'bathrooms',
 'guests_included',
 'price',
 'amenities']
```

Ahora que ya se ha tratado esta columna aparte, a las demás variables categóricas se les aplica la técnica `One Hot Encoding` como se ha comentado anteriormente. Para ello se crea un `pipeline` (una secuencia de transformaciones de datos) para que de esta forma los datos puedan ser procesados posteriormente en el modelo durante el proceso de entrenamiento. Así, para este `pipeline`se van a definir dos pasos: * Un objeto `SimpleImputer` (que reemplace los valores perdidos por la etiqueta `missing`) * Un objeto `OneHotEncoder` que transforme las variables categóricas a etiquetas binarias. Se fija el parámetro `handle_unknown='ignore'` para que si encuentrora alguna categoría desconocida después del entrenamiento, las columnas resultantes sean todas cero.

[23]:
```
# Se crea un 'pipeline' para variables categóricas que imputa valores perdidos␣
 ↪usando el valor 'missing' y
# que realiza 'One Hot Encoding', creando una columna binaria para cada uno de␣
 ↪los valores que puede tomar cada variable
# El 'pipeline' es una lista de tuplas (nombre, transformación)
categorical_transformer = Pipeline([('imputer',␣
 ↪SimpleImputer(strategy='constant',
                                                                    ␣
 ↪fill_value='missing')),
                                     ('onehot',␣
 ↪OneHotEncoder(handle_unknown='ignore'))])
categorical_transformer
```

[23]:
```
Pipeline(steps=[('imputer',
                 SimpleImputer(fill_value='missing', strategy='constant')),
                ('onehot', OneHotEncoder(handle_unknown='ignore'))])
```

Una vez definido el `pipeline`que se va a usar para las variables categóricas, se pasa a inspeccionar las variables numéricas. Para empezar, se comprueba si hay valores perdidos en cada una de ellas, y el número de ellas:

[24]:
```
df_listing_reduced[numerical_columns].isna().sum()
```

[24]:
```
square_feet         21194
accommodates            0
cleaning_fee         5225
beds                  139
availability_30         0
bedrooms               16
availability_60         0
availability_90         0
availability_365        0
security_deposit     7055
```

```
longitude              0
latitude               0
bathrooms             15
guests_included        0
price                  0
amenities              0
dtype: int64
```

Puede apreciarse que hay muchísimos valores perdidos en `square_feet`, además de `security_deposit` y `cleaning_fee`. En menor medida, hay valores perdidos en `beds`, `bedrooms` y `bathrooms`. Esto hay que tratarlo de alguna forma.

Por otra parte, las variables numéricas de longitud y latitud, las cuales están expresadas en grados, tienen el problema de que representan un espacio tridimensional. Esto significa que los dos valores más extremos de la coordenada de longitud en la realidad están muy cerca uno de otro. Lo que se puede realizar para intentar solucionar esto es mapear estas dos coordenadas a coordenadas cartesianas (X, Y y Z). De esta forma, los puntos cercanos en estas tres dimensiones serán también puntos cercanos en la realidad. Así, se crean tres nuevas columnas para estas tres coordenadas y se eliminan las columnas de `latitude` y `longitude`:

[25]:
```
# Se crean las tres nuevas columnas (importante asignarlas al dataframe␣
 ↪existente)
df_listing_reduced = df_listing_reduced.assign(X = 6378137 * np.cos(np.
 ↪radians(df_listing_reduced['latitude'])) * np.cos(np.
 ↪radians(df_listing_reduced['longitude'])))
df_listing_reduced = df_listing_reduced.assign(Y = 6378137 * np.cos(np.
 ↪radians(df_listing_reduced['latitude'])) * np.sin(np.
 ↪radians(df_listing_reduced['longitude'])))
df_listing_reduced = df_listing_reduced.assign(Z = 6378137 * np.sin(np.
 ↪radians(df_listing_reduced['latitude'])))

# Se eliminan las columnas de longitud y latitud
df_listing_reduced = df_listing_reduced.drop(columns=['latitude', 'longitude'])
```

Una vez se tienen estas tres nuevas columnas, se añaden a la lista de columnas numéricas y se eliminan las columnas de latitud y longitud. Además, se elimina la variable `price`, ya que esta va a ser la etiqueta o variable objetivo:

[26]:
```
# List Comprehensión para eliminar las dos variables de la lista, además de la␣
 ↪variable objetivo 'price'
numerical_columns = [column for column in numerical_columns if 'latitude' not␣
 ↪in column if 'longitude' not in column
                     if 'price' not in column]

# Se extiende la lista con las nuevas columnas de las coordenadas cartesianas
numerical_columns.extend(['X', 'Y', 'Z'])

# Imprimer lista por pantalla para ver que está correcta
```

```
numerical_columns
```

[26]:
```
['square_feet',
 'accommodates',
 'cleaning_fee',
 'beds',
 'availability_30',
 'bedrooms',
 'availability_60',
 'availability_90',
 'availability_365',
 'security_deposit',
 'bathrooms',
 'guests_included',
 'amenities',
 'X',
 'Y',
 'Z']
```

Una vez creadas las nuevas columnas, se crea un `pipeline` que será utilizado posteriormente para las características numéricas. En este caso, también se va a definir un `pipeline` de dos pasos: * En el primero de ellos se imputan valores perdidos (de nuevo con `SimpleImputer`), con la media o mediana de cada columna (se definirá la estrategia más adelante). * En el segunda se usa `Standard Scaler` para escalar y redimensionar la distribución de los valores de cada columna, de forma que al final tengan media cero y desviación típica unidad. Así, ningun valor individual de ninguna columna influye desproporcionadamente en el algoritmo debido a diferencias de escala.

[27]:
```
# Se crea un 'pipeline' para variables numéricas que imputa valores
# perdidos y que escala los valores a media cero y desviación unidad
# El 'pipeline' es una lista de tuplas (nombre, transformación)
numerical_transformer = Pipeline([('imputer', SimpleImputer()),
                                  ('scaler', StandardScaler())])
numerical_transformer
```

[27]:
```
Pipeline(steps=[('imputer', SimpleImputer()), ('scaler', StandardScaler())])
```

Antes del entrenamiento, falta realizar una última cosa. Y es que puede fusionar `df_listing_reduced` con `df_calendar`, el cual se examinó en el apartado del análisis exploratorio de datos. De esta forma, sería posible extraer información de las fechas de las publicaciones, ya que el cliente pretende predecir el precio de alquiler diario de sus inmuebles, y claro, este precio puede ir variando dependiendo del mes o del periodo del año en el que uno se encuentre.

Como ya se ha transformado `df_listing_reduced`, es ahora cuando se hace la fusión. Una vez se realice, se eliminan las columnas sobrantes de `df_calendar` que no son útiles para la predicción de los precios y se renombra la columna de precios (simplemente por estética):

[28]:
```
# Fusión de 'dataframes', guardándolo en la variable con la que se estaba␣
↪trabajando hasta ahora
```

```
df_listing_reduced = pd.merge(df_listing_reduced, df_calendar, on='id')

# Elimina columnas que sobran tras la fusión y se renombra la columna de␣
 ↪precios del 'dataframe' de calendarios
# a 'price'
df_listing_reduced.drop(columns=['price_x', 'adjusted_price', 'minimum_nights',␣
 ↪'maximum_nights', 'available'],
                        inplace=True)
df_listing_reduced.rename(columns={"price_y": "price"}, inplace = True)
```

Una vez se realiza la fusión, se transforma la columna `date` para que muestre en vez del día, el mes y la fecha; el número de la semana anual a la que pertenece la observación (es decir, de la semana 1 a la semana 53):

[29]:
```
df_listing_reduced['date'] = df_listing_reduced['date'].dt.isocalendar().week
```

Una vez realizada esta transformación, se agrupa `df_listing_reduced` seleccionando todas las columnas (menos la columna objetivo de precio) sin descartar los valores nulos, puesto que más tarde se van a imputar. A cada uno de estos grupos se les calcula la media de los precios. De esta forma, si para un alquiler existen dos o más precios distintos en la misma semana en el `dataframe`, se calcula la media de los mismos. Con esto se consiguen dos cosas: * No se tiene un `dataframe` con múltiples columnas prácticamente iguales con varios precios para cada semana, sino que se tiene el precio medio del alquiler para cada semana. * Se reduce el coste computacional a la hora del entrenamiento.

[30]:
```
# df_listing_reduced.groupby(['id'] + df_listing_reduced.columns.tolist()[:-1],␣
 ↪as_index=False, dropna=False).mean()

# Se agrupa por todas las columnas menos el precio, y se calcula la media (sin␣
 ↪eliminar NaN)
df_listing_reduced = df_listing_reduced.groupby(['id'] + df_listing_reduced.
 ↪columns.tolist()[:-1], as_index=False, dropna=False).mean()
```

Por último, se crea un `pipeline` específico para la columna `date`, y que solamente va a imputar valores perdidos (aunque no tiene ninguno). Para esta columna no se escalan los valores numéricos porque se considera como columna categórica, ya que cada semana del año tiene un orden entre ellas. En este caso, si la variable no fuera numérica, habría que hacer `Label Encoding`, pero como la propia columna ya tiene la propia codificación realizada, se deja como está sin aplicar dicha técnica.

[31]:
```
date_transformer = Pipeline(steps=[('imputer', SimpleImputer())])
date_transformer
```

[31]:
```
Pipeline(steps=[('imputer', SimpleImputer())])
```

Tras la preparación de estos datos y la creación de los `pipelines` para columnas específicas, se crean los conjuntos de `entrenamiento` y `test`: * Por un lado, se guardan todas las columnas características en una variable. Del mismo modo, se extrae la columna objetivo `price`y se guarda

en otra variable. * A partir de aquí, se hace la repartición entre el conjunto de entrenamiento y test: el conjunto de entrenamiento tendrá el 80% de los datos, y el conjunto de test el 20%.

[32]:
```
# Se elimina la columna 'price', quedándonos con todas las demás variables, que␣
 ↪se guardarán en la variable X
X = df_listing_reduced.drop('price', axis=1)

# Se extrae la variable objetivo 'price'
y = df_listing_reduced.loc[:, 'price']

# Repartición entrenamiento-test (80%-20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
 ↪random_state=42)
```

Ya que se ha realizado la repartición de datos de entreno y test, se crea ahora un objeto de tipo `ColumnTransformer` que aplica los `pipelines` definidos anteriormente, y por tanto las transformaciones definidas, sobre las columnas que se especifiquen. De esta forma: * Se aplica el `pipeline` nombrado `numerical_transformer` a las columnas numéricas del `dataframe`. * Se aplica el `pipeline` nombrado `categorical_transformer` a las columnas categóricas del `dataframe` * Se aplica el `pipeline` nombrado `date_transformer` a la columna `date` del `dataframe`.

[33]:
```
# ColumnTransformer aplica los transformadores definidos a las columnas del␣
 ↪'dataframe'.
# ColumnTransformer es una lista de tuplas (nombre, transformación, nombre␣
 ↪columnas)
column_transformer = ColumnTransformer(transformers = [('numerical',␣
 ↪numerical_transformer, numerical_columns),
                                                        ('categorical',␣
 ↪categorical_transformer, categorical_columns),
                                                        ('date',␣
 ↪date_transformer, ['date'])])
column_transformer
```

[33]:
```
ColumnTransformer(transformers=[('numerical',
                                 Pipeline(steps=[('imputer', SimpleImputer()),
                                                 ('scaler', StandardScaler())]),
                                 ['square_feet', 'accommodates', 'cleaning_fee',
                                  'beds', 'availability_30', 'bedrooms',
                                  'availability_60', 'availability_90',
                                  'availability_365', 'security_deposit',
                                  'bathrooms', 'guests_included', 'amenities',
                                  'X', 'Y', 'Z']),
                                ('categorical',
                                 Pipeline(steps=[('imputer',
SimpleImputer(fill_value='missing',
strategy='constant')),
                                                 ('onehot',
```

```
OneHotEncoder(handle_unknown='ignore'))]),
                                 ['neighbourhood',
                                  'neighbourhood_group_cleansed',
                                  'property_type', 'room_type']),
                                ('date',
                                 Pipeline(steps=[('imputer', SimpleImputer())]),
                                 ['date'])])
```

Ahora sí, se procede a entrenar el modelo usando validación cruzada. Con este método, los datos de entrenamiento se van a dividir en cinco subconjuntos (en este caso), siendo cuatro de ellos utilizados como datos de entrenamiento y el quinto restante como datos de validación. Este proceso se repite 5 veces (como el número de subconjuntos establecidos), de modo que todos los subconjuntos rotan y ejercen como subconjunto de validación en alguna iteración.

Como métrica de evaluación se va a examinar el coeficiente de determinación y el error absoluto medio (aunque también se podría examinar el error cuadrático medio), que son las métricas más usadas en un problema de regresión como al que nos enfrentamos. El coeficiente de determinación determina la calidad del modelo para replicar resultados y la proporción de variación de los resultados que puede explicarse con él. Cuánto más cercano a la unidad sea el valor de este coeficiente, mejor será el modelo y mayor será la variabilidad explicada para la variable de respuesta (`price` en este caso). El error absoluto medio, por su parte, mide la media de las diferencias absolutas entre los valores reales y los predichos por el modelo. Esta métrica es útil para minimizar el error general del modelo.

Se va a optar para este primer entrenamiento por un modelo de regresión lineal. Para ello se va a usar `GridSearchCV` para realizar una búsqueda exhaustiva del mejor parámetro de imputación de valores entre los especificados: * Se va a buscar dos métodos de imputación para las variables numéricas (media y mediana).

Al final, una vez encontrado los mejores parámetros, el modelo se reentrena con todos los subconjuntos establecidos ya como datos de entrenamiento. Este modelo final se guarda y se calcula el error absoluto medio y el coeficiente de determinación en el conjunto de `test` con este mejor modelo.

[49]:
```
from sklearn.linear_model import LinearRegression

# Se crea un objeto de regresión lineal
lr_model = LinearRegression()

# Se crea un 'Pipeline' que encadena las transformaciones a las columnas␣
 ↪('prepro') y el modelo de regresión ('lr_model').
# 'Pipeline' es una lista de tuplas (nombre, transformación)
price_pipe_lr = Pipeline([('prepro', column_transformer), ('ridge', lr_model)])

# Se configuran los parámetros a buscar para el método GridSearchCV
parameters = {}
parameters['prepro__numerical__imputer__strategy'] = ['mean', 'median']
```

```
# 'GridSearchCV' realiza una búsqueda exhaustiva sobre los parámetros␣
 ↪especificados para el estimador 'price_pipe_lr' (que
# es el 'pipeline' que encadena las transformaciones a las columnas y el modelo␣
 ↪de regresión)
# 'param_grid' es el diccionario con las configuraciones de parámetros a probar
# Como 'scoring' se utiliza el coeficiente de determinación R2
search_lr = GridSearchCV(estimator=price_pipe_lr, param_grid=parameters,␣
 ↪scoring='r2', cv=5)

# Se entrena el modelo con los datos de entrenamiento todos los conjuntos de␣
 ↪parámetros especificados
search_lr = search_lr.fit(X_train, y_train)

# Se imprime por pantalla los mejores paráemtros encontrados por GridSearchCV
print("Mejor configuración de parámetros: ", search_lr.best_params_)

# Se guarda en 'search_ridge' el mejor estimador
search_lr = search_lr.best_estimator_

# 'Score' en el conjunto de entrenamiento y test
print("R2 Entrenamiento: ", search_lr.score(X_train, y_train))
print("R2 Test: ", search_lr.score(X_test, y_test))
print("MAE Test: ", mean_absolute_error((y_test), (search_lr.predict(X_test))))
```

```
Mejor configuración de parámetros:  {'prepro__numerical__imputer__strategy':
'mean'}
R2 Entrenamiento:  0.05001978194232126
R2 Test:  0.05032853170554297
MAE Test:  114.66670124013648
```

Como se puede apreciar, la mejor estrategia para imputar los valores numéricos ha sido usar la media. Por otra parte, el coeficiente de determinación es muy malo y el error absoluto medio también es bastante grande, de 114 dólares. Por tanto, se prueba a entrenar otro tipo de modelo, ya que está claro que un modelo lineal no es muy útil para este problema.

Ahora se va a entrenar un `árbol de decisión`. Se trata de un algoritmo de aprendizaje supervisado que posee una estructura parecida a la de un árbol de forma jerárquica, con nodo raíz, nodos internos y nodos hoja.

[40]:
```
# 30 min de entrenamiento casi con dos parámetros
from sklearn.tree import DecisionTreeRegressor

# Se crea un objeto del estimador
dtr_model = DecisionTreeRegressor()

# Se crea un 'Pipeline' que encadena las transformaciones a las columnas y el␣
 ↪propio modelo.
# 'Pipeline' es una lista de tuplas (nombre, transformación)
```

```
price_pipe_dtr = Pipeline([('prepro', column_transformer), ('decision_tree',␣
 ↪dtr_model)])

# Se configuran los parámetros a buscar para el método GridSearchCV
parameters = {}
parameters['prepro__numerical__imputer__strategy'] = ['mean']

# 'GridSearchCV' realiza una búsqueda exhaustiva sobre los parámetros␣
 ↪especificados para el estimador (que es el 'pipeline'
# que encadena las transformaciones a las columnas y el modelo)
# 'param_grid' es el diccionario con las configuraciones de parámetros a probar
# Como 'scoring' se utiliza el coeficiente de determinación R2
search_dtr = GridSearchCV(estimator=price_pipe_dtr, param_grid=parameters,
                          scoring='neg_mean_absolute_error', cv=5, n_jobs = -1)

# Se entrena el modelo con los datos de entrenamiento todos los conjuntos de␣
 ↪parámetros especificados
search_dtr = search_dtr.fit(X_train, y_train)

# Se imprime por pantalla los mejores paráemtros encontrados por GridSearchCV
print("Mejor configuración de parámetros: ", search_dtr.best_params_)

# Se guarda el mejor estimador
best_dtr = search_dtr.best_estimator_

# 'Score' en el conjunto de entrenamiento y test
print("R2 Entrenamiento: ", best_dtr.score(X_train, y_train))
print("R2 Test: ", best_dtr.score(X_test, y_test))
print("MAE Test: ", mean_absolute_error(y_test, best_dtr.predict(X_test)))
```

```
Mejor configuración de parámetros:  {'prepro__numerical__imputer__strategy':
'mean'}
R2 Entrenamiento:  1.0
R2 Test:  0.7364475779074708
MAE Test:  10.213793019127841
```

El árbol de decisión es un modelo que se adapta mucho mejor a este problema. Como se puede apreciar, el coeficiente de determinación en el conjunto de test es bastante bueno, lo que mejora muchísimo al modelo de regresión, y por tanto, el mode explica en gran parte la proporción de variación de los resultados. En cuanto al error absoluto medio, éste es tan solo de 10 dólares. Esto quiere decir que, de media en el conjunto de test, hay un error de tan solo 10 dólares a la hora de predecir el alquilar un inmueble (si por ejemplo la vivienda tiene un precio de 80 dólares al alquiler diario, el modelo como error medio predice 70 u 90 dólares).

Ahora se va a entrenar modelos `ensemble`, los cuales sobreajustan menos al fusionar predicciones de múltiples modelos. Para empezar se opta por entrenar un modelo de `XGBoost`, un algoritmo de `boosting` (un método en el que se realiza un entrenamiento secuencial de varios modelos para mejorar el resultado final) basado en árboles de decisión que combina las salidas de estos árboles para

alcanzar un solo resultado, mitigando errores o sesgos que puedan existir en modelos individuales. Se realiza el entrenamiento con 1000 estimadores base (número de árboles de decisión) y una profundidad máxima sin límites.

[37]:
```
from xgboost import XGBRegressor

# Se crea un objeto del estimador
xgb_model = XGBRegressor(n_estimators=1000, max_depth=None)

# Se crea un 'Pipeline' que encadena las transformaciones a las columnas y el␣
 ↪propio modelo.
# 'Pipeline' es una lista de tuplas (nombre, transformación)
price_pipe_xgb = Pipeline([('prepro', column_transformer), ('xgb', xgb_model)])

# Se configuran los parámetros a buscar para el método GridSearchCV
parameters = {}
parameters['prepro__numerical__imputer__strategy'] = ['mean', 'median']

# 'GridSearchCV' realiza una búsqueda exhaustiva sobre los parámetros␣
 ↪especificados para el estimador (que es el 'pipeline' que encadena
# las transformaciones a las columnas y el modelo)
# 'param_grid' es el diccionario con las configuraciones de parámetros a probar
search_xgb = GridSearchCV(estimator=price_pipe_xgb, param_grid=parameters,
                          scoring='neg_mean_absolute_error', cv=5, n_jobs=-1)

# Se entrena el modelo con los datos de entrenamiento todos los conjuntos de␣
 ↪parámetros especificados
search_xgb = search_xgb.fit(X_train, y_train)

# Se imprime por pantalla los mejores paráemtros encontrados por GridSearchCV
print("Mejor configuración de parámetros: ", search_xgb.best_params_)

# Se guarda el mejor estimador
search_xgb = search_xgb.best_estimator_

# 'Score' en el conjunto de entrenamiento y test
print("R2 Entrenamiento: ", search_xgb.score(X_train, y_train))
print("R2 Test: ", search_xgb.score(X_test, y_test))
print("MAE Test: ", mean_absolute_error((y_test), (search_xgb.predict(X_test))))
```

```
Mejor configuración de parámetros:  {'prepro__numerical__imputer__strategy':
'mean'}
R2 Entrenamiento:  0.9674725420768129
R2 Test:  0.8511368376200105
MAE Test:  32.80599475420433
```

Como se ve, con el modelo `XGBoost` se tiene un mayor coeficiente de determinación en el conjunto de `test` que con el árbol de decisión anterior. Esto quiere decir que la proporción de variación

de los resultados puede explicarse mejor con este modelo. Sin embargo, el error absoluto medio es mayor en este caso. Por lo tanto, las predicciones de los alquileres serán peores en este caso respecto al árbol de decisión.

Ahora se prueba otro modelo de `Boosting` llamado `LGBM`. Este modelo es más rápido que `XGBoost` y tiene un enfoque mucho mejor cuando trata grandes conjuntos de datos como éste. Se configura la misma configuración que en el entrenamiento anterior (1000 estimadores y profundidad sin límite).

[36]:
```
from lightgbm import LGBMRegressor

# Se crea un objeto del estimador
lgbm_model = LGBMRegressor(n_estimators=1000, max_depth=None)

# Se crea un 'Pipeline' que encadena las transformaciones a las columnas y el␣
 ↪propio modelo.
# 'Pipeline' es una lista de tuplas (nombre, transformación)
price_pipe_lgbm = Pipeline([('prepro', column_transformer), ('lgbm',␣
 ↪lgbm_model)])

# Se configuran los parámetros a buscar para el método GridSearchCV
parameters = {}
parameters['prepro__numerical__imputer__strategy'] = ['mean', 'median']

# 'GridSearchCV' realiza una búsqueda exhaustiva sobre los parámetros␣
 ↪especificados para el estimador (que es el 'pipeline' que encadena
# las transformaciones a las columnas y el modelo)
# 'param_grid' es el diccionario con las configuraciones de parámetros a probar
search_lgbm = GridSearchCV(estimator=price_pipe_lgbm, param_grid=parameters,
                           scoring='neg_mean_absolute_error', cv=5, n_jobs=-1)

# Se entrena el modelo con los datos de entrenamiento todos los conjuntos de␣
 ↪parámetros especificados
search_lgbm = search_lgbm.fit(X_train, y_train)

# Se imprime por pantalla los mejores paráemtros encontrados por GridSearchCV
print("Mejor configuración de parámetros: ", search_lgbm.best_params_)

# Se guarda el mejor estimador
search_lgbm = search_lgbm.best_estimator_

# 'Score' en el conjunto de entrenamiento y test
print("R2 Entrenamiento: ", search_lgbm.score(X_train, y_train))
print("R2 Test: ", search_lgbm.score(X_test, y_test))
print("MAE Test: ", mean_absolute_error(y_test, search_lgbm.predict(X_test)))
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.040716 seconds.
You can set `force_row_wise=true` to remove the overhead.
```

```
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 1989
[LightGBM] [Info] Number of data points in the train set: 911369, number of used
features: 141
[LightGBM] [Info] Start training from score 142.479084
Mejor configuración de parámetros:  {'prepro__numerical__imputer__strategy':
'mean'}
R2 Entrenamiento:  0.8986629874760689
R2 Test:  0.8346992219194962
MAE Test:  43.09882347331721
```

Con este modelo se tiene un menor coeficiente de determinación en el conjunto de `test` y un error absoluto medio mayor que con el modelo entrenado con `XGBoost`. Por lo tanto, es preferible usar el modelo anterior o el del árbol de regresión.

INFO:

Me hubiera gustado haber sacado métricas de otros algoritmos como `Random Forest` o `SVM`, pero como el `dataset` es tan grande, el tiempo de cómputo me superaba la hora y media y aún no se ejecutaba la celda. O directamente el ordenador se me quedaba congelado.

El objetivo del cliente es saber a que precio alquilar sus inmuebles diariamente. Entre todos los modelos entrenados se elige el modelo del árbol de regresión porque es el que menor error absoluto medio ha conseguido en el conjunto de `test`, y precisamente esto es lo que le interesa al cliente, reducir al máximo posible el error de predicción.