

Informe

Análisis de Datos de Airbnb en la ciudad de Madrid

Anahi P. Valenzuela
Carolina G. Siqueira
Julieta Abbiendi
Liseth Apollini
María Victoria Ferrando
Renata Yumi Namie

ÍNDICE

ENUNCIADO DEL PROYECTO	2
INTRODUCCIÓN	4
REALIZACIÓN DEL PROYECTO FINAL	5
1. Definir Data Set	5
2. Arquitectura y validación de los datos	5
3. Análisis Exploratorio	11
4. Visualización de las métricas	22
5. Pre-procesamiento y Modelado	44
CONCLUSIONES	60

ENUNCIADO DEL PROYECTO

1. Definir Data Set

2. Arquitectura y validación de los datos

- a. Muestreo y exploración inicial de los datos
- b. Definir e implementar el Datawarehouse
- c. (Opcional) Ingesta de datos (ETL) y validación de que se ha cargado correctamente

3. Análisis Exploratorio

Hacer un estudio estadístico con R o Python, según preferencia personal, y averiguar cuáles son las métricas adecuadas para el dataset.

No olvidemos:

- a. Revisión de la calidad de los datos
- b. Detección outliers (rango de variables), imputación valores nulos.
- c. Boxplots, histogramas, etc.
- d. Normalización de los valores de las tablas (quitar tildes, “dobles espacios”, etc.)

4. Visualización de las métricas

A partir de los datos de Airbnb, obtén los KPIs que puedan ser de relevancia y contesta a través de un dashboard a una pregunta relevante que hagas sobre los datos.

- a. Se valorará el diseño final del dashboard.
- b. El uso de buenas prácticas.
- c. El cálculo de KPIs adecuados y el uso de campos calculados avanzados.
- d. El uso de vistas interactivas.

5. Preprocesamiento y Modelado

La tarea asignada es hacer un algoritmo de regresión lineal que prediga el precio de un inmueble en función de las características que elijáis.

INTRODUCCIÓN

Airbnb es conocido en la actualidad como una de las mejores y más utilizadas plataformas para la búsqueda de propiedades de alquiler vacacional alrededor del mundo.

No solo brinda la opción a los usuarios de encontrar la propiedad idónea para sus vacaciones si no que, permite introducir su propiedad al mercado para que alguien más la pueda alquilar, completando así, **el ciclo de experiencia de alquiler de Airbnb**.

Creemos que, uno de los principales retos que pueden tener los usuarios que buscan poner en alquiler una propiedad, es el de determinar el precio apropiado de alquiler por noche de esta. Sabemos que hay varios factores que pueden influir en el precio, pero más allá de utilizar la intuición y el compararse con el resto de las propiedades: ¿Qué herramientas tienen los usuarios? ¿Cómo podríamos hacerlo más fácil?

Pensamos que el desarrollo de un producto digital que mejore la experiencia del usuario, que pueda mantenerse en el tiempo y que sea capaz de adaptarse a las fluctuaciones del mercado a través de parámetros dinámicos fáciles de actualizar, podría ser una solución.

Es por eso que iniciamos este proyecto de análisis de datos cuyo objetivo principal es el de poder predecir a través de distintas herramientas y una base de datos sólida, el rango de precio que pudiese tener una propiedad según sus características.

REALIZACIÓN DEL PROYECTO FINAL

1. Definir Data Set

El Data Set fue dado y consistía en una base de datos de alquileres de la empresa Airbnb.

2. Arquitectura y validación de los datos

Al iniciar la exploración de datos el primer obstáculo que nos encontramos fue que la suscripción gratuita “Tiny Turtle” de PostgreSQL tiene como límite 20 mb y el dataset proporcionado supera esa cantidad.

Por tanto, decidimos hacer la exploración inicial en Python para así quedarnos con una versión más liviana del dataset. En la misma dejamos de lado columnas que no consideramos relevantes para el modelo y entre otras cosas tuvimos en cuenta por ejemplo sólo la ciudad de Madrid. Más detalles de la limpieza son detallados en el punto 3 - “Limpieza de Datos”.

Tuvimos que crear una Base de Datos que nos ayudara a dar respuesta al siguiente requisito:

realizar un modelo que predijera **el precio de un inmueble en función de ciertas características seleccionadas.**

Hemos realizado la normalización del .csv para poder diseñar la BBDD y luego representarla a través del modelo de Entidad-Relación.

Lo primero que se hizo fue separar los valores de los atributos “amenities” y “features” e identificar con valores booleanos si el mismo estaba presente o no en la propiedad, luego creamos una tabla de relación de dichos atributos con sus respectivas entidades:

- Amenities para la Property
- Features para el Host

Luego, fuimos identificando del .csv, cuáles eran las principales entidades que debíamos crear y cómo se relacionaban entre ellas evitando así la redundancia y dependencia de los datos.

De esta forma cumplimos con la segunda forma normal, haciendo que los datos que correspondían a distintas entidades estuviesen en tablas separadas con su propia clave de identificación. Como fue el caso de:

- Property Type
- Room Type
- Bed Type, etc.

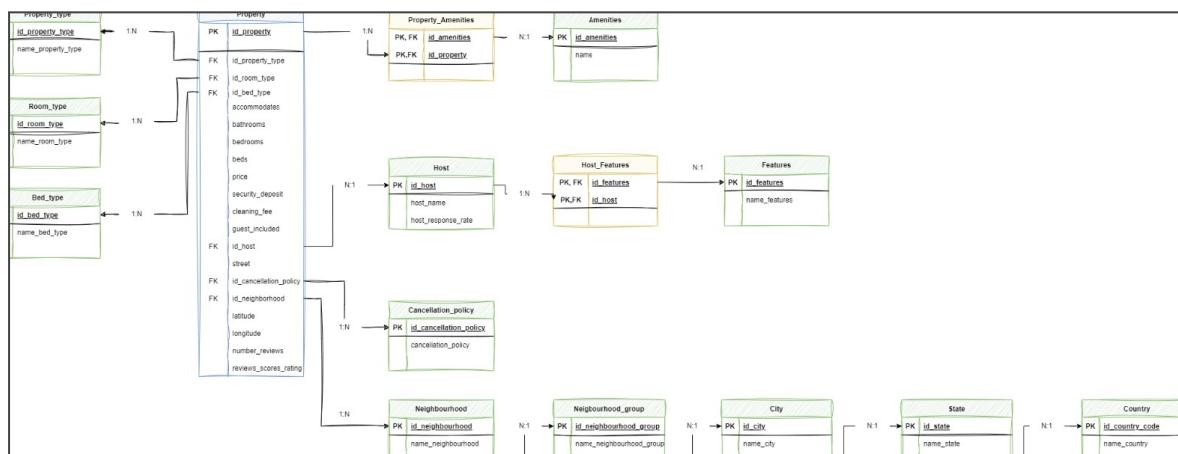
Para cumplir con la tercera forma normal, buscamos que las columnas de la tabla no dependieran de forma transitiva de la clave primaria.

Es por ello por lo que todas las relaciones con dichas tablas se extrajeron, lo cual nos da la posibilidad de aprovechar la relación para otra entidad (si se quisiera en un futuro) como podría ser el neighborhood para el Host.



De esta manera también se cumple la cuarta forma normal al no existir registros duplicados en las entidades, se tienen datos únicos mediante el uso de claves foráneas.

Finalizada la normalización, diseñamos el diagrama Entidad-Relación en el aplicativo diagrama.net.



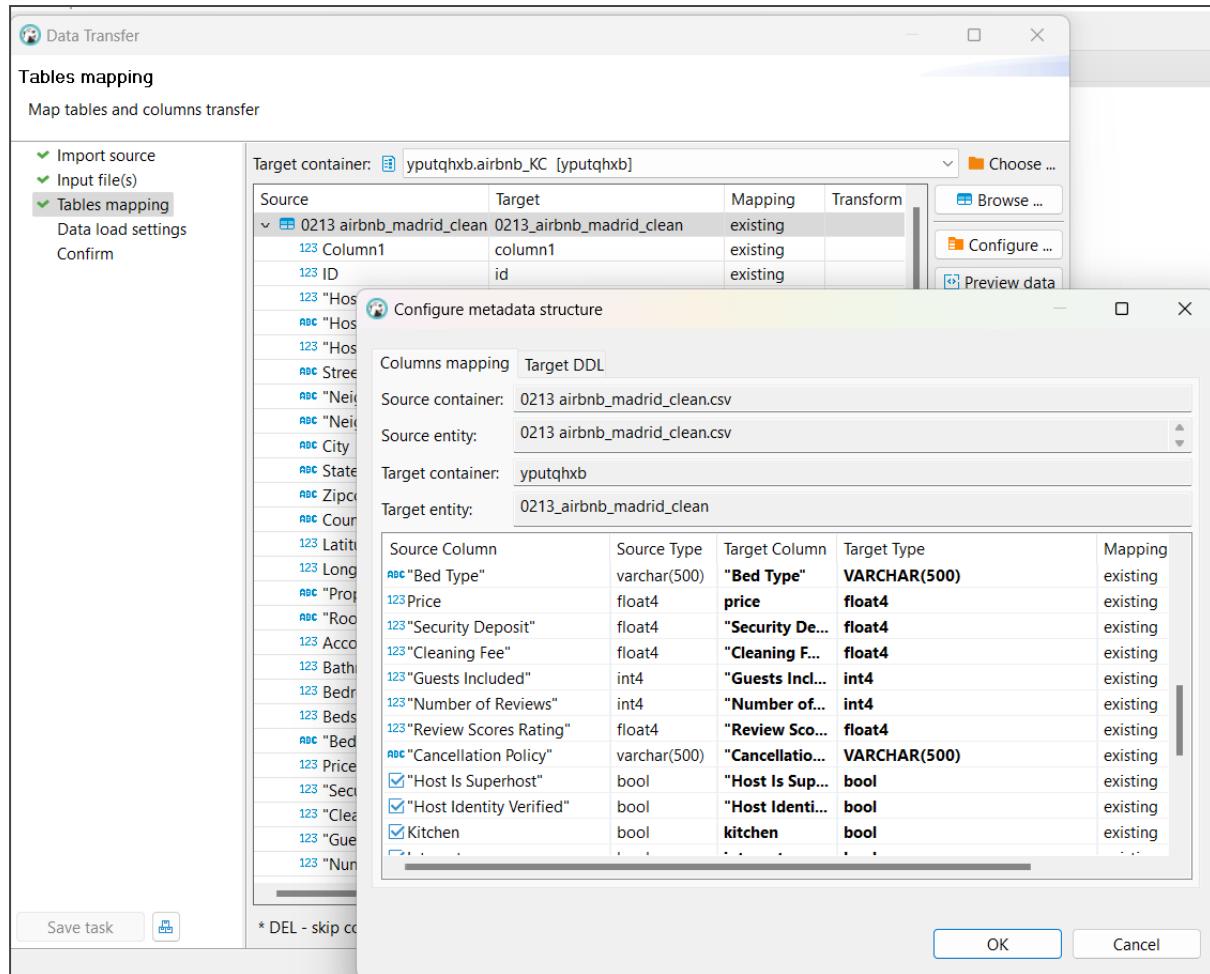
Habiendo modelado y verificado que nuestro modelo estuviese normalizado y que cumpliera con los requisitos funcionales procedimos a la creación de la estructura del modelo (tablas, campos, tipos de datos, claves) con SQL utilizando las herramientas DBeaver-PostgreSQL.

Lo primero que se tuvo que hacer fue crear una nueva instancia en Elephant SQL para tener mayor capacidades de memoria y rendimiento.

Instances				
Name	Host	Plan	Datacenter	Actions
Proyecto Final KC	manny	Tiny Turtle	Amazon Web Services EU-West-1 (Ireland)	<button>Edit</button>
keepCodingModuloSQL	lucky	Tiny Turtle	Amazon Web Services EU-West-1 (Ireland)	<button>Edit</button>

Una vez realizada la nueva conexión en DBeaver procedimos a importar el nuevo csv limpio "airbnb_madrid_clean". Tuvimos que correr el proceso algunas veces para ir depurando los errores que teníamos en la importación.

Por ejemplo asegurarse que el delimitador fuera coma y no punto y coma, que el tipo de dato aceptara la cantidad adecuada de caracteres, pusimos varchar(500) en su mayoría para evitar problemas, así como las columnas de amenities y features las trajera como booleanas.



Una vez importados los datos, los exploramos y corroboramos que la cantidad de líneas importadas coincidiera con las del csv.

```
--Visualizamos y exploramos tabla ppal

@SELECT *
from "airbnb_KC"."0213_airbnb_madrid_clean" amc
order by amc.id
limit 20

@SELECT count(*)
from "airbnb_KC"."0213_airbnb_madrid_clean" amc
```

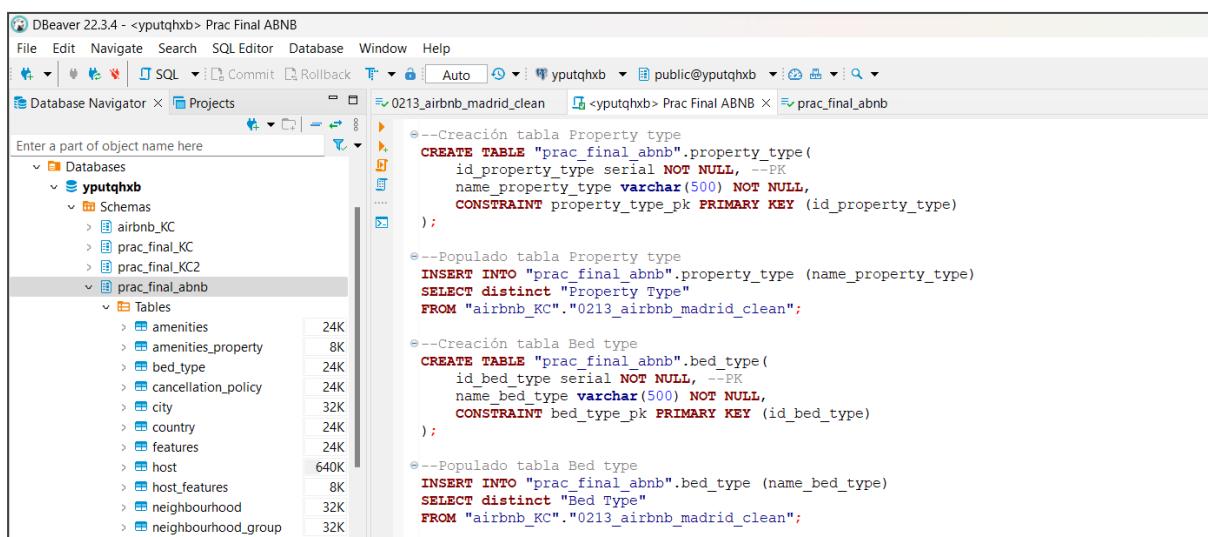
Al tener que usar una base de datos “limpia” y no el fichero original, el csv limpiado en python incluía dos columnas calculadas con objetivo de análisis y visualización que no fue considerada en la exploración de datos en SQL ya que no pertenecía a la base de datos por se. Así como tampoco se utilizó la columna zipcode tanto por algunas inconsistencias encontradas en la información que traía, así como tampoco la consideramos relevante frente a otras columnas que también referían a la locación de la propiedad.

Acto seguido creamos un esquema particular `pac_final_abnb` para luego avanzar en la creación de las tablas.

```
-- creamos schema particular para el DER

create schema pac_final_abnb authorization yputqhx;
```

Empezamos por aquellas que no tenían referencia con ninguna clave foránea (FK) en otra tabla. Paso siguiente se fueron llenando las tablas con los valores únicos que se encontraban en el .csv importado y con una nueva columna de identificación única “serial” que pasaría a ser su clave primaria.



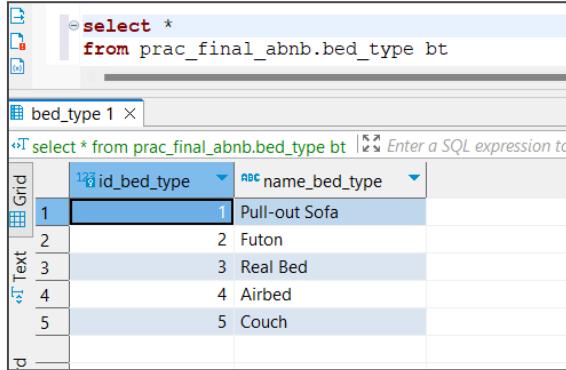
```
--Creación tabla Property type
CREATE TABLE "pac_final_abnb".property_type(
    id_property_type serial NOT NULL, --PK
    name_property_type varchar(500) NOT NULL,
    CONSTRAINT property_type_pk PRIMARY KEY (id_property_type)
);

--Populado tabla Property type
INSERT INTO "pac_final_abnb".property_type (name_property_type)
SELECT distinct "Property Type"
FROM "airbnb_KC"."0213_airbnb_madrid_clean";

--Creación tabla Bed type
CREATE TABLE "pac_final_abnb".bed_type(
    id_bed_type serial NOT NULL, --PK
    name_bed_type varchar(500) NOT NULL,
    CONSTRAINT bed_type_pk PRIMARY KEY (id_bed_type)
);

--Populado tabla Bed type
INSERT INTO "pac_final_abnb".bed_type (name_bed_type)
SELECT distinct "Bed Type"
FROM "airbnb_KC"."0213_airbnb_madrid_clean";
```

Debajo se puede visualizar como se fueron creando las tablas normalizadas con su id único generado de forma autonumérica.



	id_bed_type	name_bed_type
1	1	Pull-out Sofa
2	2	Futon
3	3	Real Bed
4	4	Airbed
5	5	Couch

Con dichas tablas completas, avanzamos en la creación de las tablas que poseían FK con las primeras tablas.

Para ello hicimos una consulta para que en función de los nombres que poseían esas tablas, traernos los Ids correspondientes del maestro y los valores únicos del .cvs. Se guardó la consulta en una nueva tabla. Cómo siguientes pasos:

- Cambiamos el nombre de las columnas para que siguiera lo definido en el Diagrama E-R
- Creamos una columna nueva para la clave primaria (PK)
- Modificamos la columna que sería la FK
- Definimos que la FK sería un valor no nulo

Debajo un ejemplo de los pasos mencionados con la creación de la tabla City:

```


-- Creación y poblado tabla City
CREATE TABLE "Práctica_Final".city
AS
SELECT DISTINCT amc.city, s.id_state
FROM "Práctica_Final".airbnb_madrid_clean amc , "Práctica_Final".state s
WHERE amc.state = s.name_state

--Se cambia el nombre de columna para coincidir con la definición en DER
ALTER table "Práctica_Final".city
RENAME COLUMN city TO name_city;

--Se agrega la PK con serial
ALTER TABLE "Práctica_Final".city
ADD COLUMN id_city serial PRIMARY KEY NOT NULL;

--Se modifica la columna a FK
ALTER TABLE "Práctica_Final".city
ADD CONSTRAINT city_fk FOREIGN KEY (id_state) REFERENCES state (id_state);

--Se define que la FK no puede ser null
ALTER TABLE "Práctica_Final".city
ALTER COLUMN id_state SET NOT NULL;


```

Finalmente, con todas las tablas creadas, avanzamos en la creación y poblado de la tabla Property que es la tabla principal de nuestra base de datos normalizada y todas sus claves.

```

-- Creación tabla property y Populación

CREATE TABLE "prac_final_abnb".property
AS

SELECT DISTINCT amc.id, amc."Host ID", pt.id_property_type ,
rt.id_room_type, bt.id_bed_type, cp.id_cancellation_policy,
n.id_neighbourhood ,amc.accommodates, amc.bathrooms , amc.bedrooms,
amc.beds , amc.price , amc."Security Deposit" , amc."Cleaning Fee" ,
amc."Guests Included" , amc."Review Scores Rating" , amc.latitude ,
amc.longitude , amc."Number of Reviews"

FROM "airbnb_KC"."0213_airbnb_madrid_clean" amc ,
"prac_final_abnb".property_type pt , "prac_final_abnb".room_type rt ,
"prac_final_abnb".bed_type bt , "prac_final_abnb".cancellation_policy cp ,
"prac_final_abnb".neighbourhood n

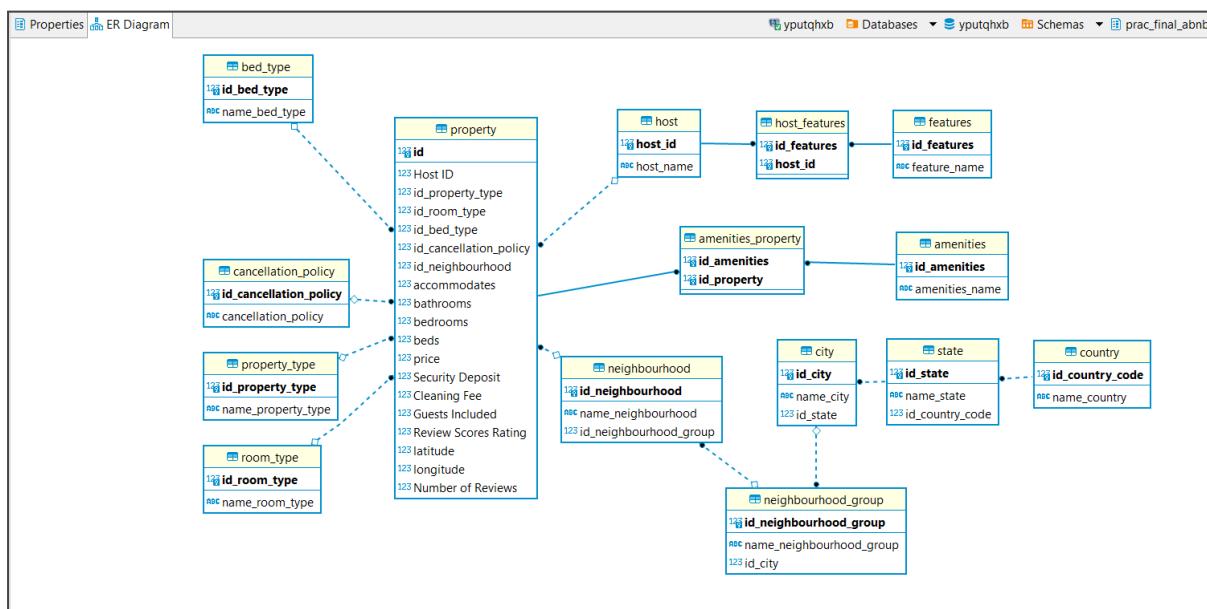
WHERE amc."Property Type" = pt.name_property_type and amc."Room Type" =
rt.name_room_type and amc."Bed Type" = bt.name_bed_type and
amc."Cancellation Policy" = cp.cancellation_policy and amc."Neighbourhood
Cleansed" = n.name_neighbourhood

```

Visualizamos los datos de la tabla `Property` y confirmamos su armado y población satisfactoria al coincidir la cantidad de registros con el csv original y la relación de los datos originales con los normalizados de forma numérica con sus respectivos IDs.

	123 id	123 Host ID	123 id_property_type	123 id_room_type	123 id_bed_type	123 id_cancellation_policy	123 id_neighbourhood	123 accommodates	123 bathrooms	123 b
1	18,628	71,597	3	1	1	5	124	2	1	
2	19,864	74,966	3	1	1	5	66	2	1	
3	21,512	82,175	3	1	1	3	113	2	1	
4	21,548	82,175	3	1	3	3	75	4	1	
5	21,853	83,531	3	3	3	5	36	1	1	
6	23,021	82,175	3	1	3	1	113	10	3	
7	24,805	101,471	3	1	3	5	110	3	1	
8	24,836	101,653	3	1	3	1	124	4	1	
9	26,571	112,866	2	3	3	5	118	1	1	
10	26,816	114,340	3	1	3	5	121	3	2	
11	26,823	114,340	1	3	3	3	121	2	1	
12	26,825	114,340	1	3	3	5	121	1	1	
13	30,320	130,907	3	1	3	1	49	2	1	
14	30,478	130,502	3	3	3	5	66	2	1	
15	30,711	132,184	1	3	3	5	14	1	0	
16	30,924	132,736	3	3	3	1	81	2	2	
17	32,134	138,713	3	3	3	3	64	3	[NULL]	
18	35,146	141,962	3	1	3	5	59	5	2	

Al finalizar la creación de toda la estructura del modelo ejecutamos el Diagrama E-R desde DBeaver y gratamente pudimos verificar la coincidencia con el modelo original diseñado.



3. Análisis Exploratorio

Limpieza de datos:

Empezamos la limpieza de los datos observando las columnas y cuáles podrían ser, como primera aproximación, útiles para el modelo. Una vez esto decidido, eliminamos las columnas innecesarias y nos quedamos con las filas correspondientes a la ciudad de Madrid.

```

df = pd.read_csv('airbnb-listings.csv', sep=';')
rows, columns = df.shape

print(f'We have {rows} rows and {columns} columns')

column_names = list(df)

print(column_names)

columns_kept = ['ID', 'Host ID', 'Host Name', 'Host Response Rate', 'Street',
'Neighbourhood Cleansed', 'Neighbourhood Group Cleansed', 'City', 'State', 'Zipcode',
'Country', 'Latitude', 'Longitude', 'Property Type', 'Room Type', 'Accommodates', 'Bathrooms', 'Bedrooms',
'Beds', 'Bed Type', 'Amenities', 'Price', 'Security Deposit', 'Cleaning Fee', 'Guests Included',
'Number of Reviews', 'Review Scores Rating', 'Cancellation Policy', 'Features']

df_less_columns = df.filter(columns_kept, axis=1)
df_with_city = df_less_columns[df_less_columns.City.notnull()]

#remove airbnbs not in Madrid
df_madrid = df_with_city[df_with_city['City'].str.contains('Mad', case=False)]

```

Como segunda instancia creamos un set de amenities para poder visualizar cuales eran todas las disponibles en los diferentes apartamentos y, en base a eso, seleccionar cuales creíamos importantes. Lo mismo con las features referentes al host.

Con la columna de *Property Types* decidimos modificar su clasificación agrupando todos los que no eran '*Apartment*' o '*House*' en '*Other*'.

```

amenities = set()
list_amenities = list(df_madrid['Amenities'])
for item in list_amenities:
    if type(item) == str:
        new_list = item.split(',')
        for amenity in new_list:
            amenities.add(amenity)

important_features = ['Host Is Superhost', 'Host Identity Verified']
important_amenities = ['Kitchen', 'Internet', 'Wireless', 'Air conditioning', 'Heating', 'Washer',
'Dryer', 'Elevator', 'Wheelchair accessible', 'TV', 'Pool', '24-hour check-in']

##we group the data in property types like apartment, house or other
relevant_types = ['Apartment', 'House']
for index in range(len(df_madrid)):
    if df_madrid['Property Type'].iat[index] not in relevant_types:
        df_madrid['Property Type'].iat[index] = 'Other'

```

Con las amenities y features seleccionadas, creamos una columna para cada una de ellas con datos booleanos.

```

for feature in important_features:
    df_madrid_2[feature] = False
    booleans = list(df_madrid_2["Features"].str.contains(feature))
    for i in range(len(df_madrid)):
        df_madrid_2[feature].iat[i] = booleans[i]

print(df_madrid_2.head())

amenities = set()
list_amenities = list(df_madrid_2["Amenities"])
for item in list_amenities:
    if type(item) == str:
        new_list = item.split(",")
        for amenity in new_list:
            amenities.add(amenity)

```

Como primera prueba hicimos los cambios con bucles `for` pero, después, para modularizar el proceso, convertimos los bucles en una función. Esto, además de un código más limpio, lo hizo más eficiente al ejecutar el archivo.

```

##adding columns for important features and amenities
df_madrid = create_and_populate_columns(important_features, 'Features', df_madrid)

df_madrid = create_and_populate_columns(important_amenities, 'Amenities', df_madrid)

```

```

def create_and_populate_columns(list_columns, column_reference, df):
    ...
    Creates columns from list_columns and populates them with booleans if the new column exists in column_refence
    ...
    for item in list_columns:
        df[item] = None
        booleans = list(df[column_reference].str.contains(item, case=False))
        df[item] = booleans
    return df

```

Unimos la columna `Wireless` con la columna `Internet`. En la columna `Zipcode`, reemplazamos los datos ‘-’ o Nulls por un string vacío.

```

##join columns 'Internet' and 'Wireless' under the internet column
for i in range(len(df_madrid)):
    df_madrid['Internet'].iat[i] = df_madrid['Internet'].iat[i] and df_madrid['Wireless'].iat[i]

##removing the original columns for Features, Amenities and Wireless
df_madrid = remove_columns(['Features', 'Amenities', 'Wireless'], df_madrid)

important_amenities.remove('Wireless')

#cleaning column 'Zipcode'
df_madrid.loc[(df_madrid['Zipcode'].isnull()) | (df_madrid['Zipcode'] == '-'), 'Zipcode'] = ''

```

Para continuar decidimos crear las columnas *Amenities Score* y *Amenities Rating*. *Amenities Score* está calculado en base a la suma del peso en la tabla de correlaciones.

```

df_madrid_copy = df_madrid[['Price', 'Accommodates', 'Bathrooms', 'Bedrooms', 'Beds', 'Cleaning Fee', 'Security Deposit',
                             'Kitchen', 'Internet', 'Air conditioning', 'Heating', 'Washer', 'Dryer', 'Elevator', 'Wheelchair accessible',
                             'TV', 'Pool', '24-hour check-in']]

##we will create a new 'Amenities Score' column. The data will be obtained by adding the weight in the correlation for each amenity in the row
corr = df_madrid_copy.corr(numeric_only=False)

dict_weights = {}
print(corr["Price"])
for amenity in important_amenities:
    weight = corr['Price'][amenity]
    dict_weights[amenity] = weight

scores = []
for index in range(len(df_madrid)):
    score = 0
    for amenity in important_amenities:
        if df_madrid[amenity].iat[index]:
            score += dict_weights[amenity]
    scores.append(score)

df_madrid['Amenities Score'] = scores

```

Amenities Rating clasifica en ‘A’, ‘B’ y ‘C’ a cada apartamento según el *Amenities Score* obtenido.

```

##based in the amenities score's quantiles, we create a new classification in the scale 'A', 'B', 'C'
q3 = df_madrid['Amenities Score'].quantile(0.33)
q6 = df_madrid['Amenities Score'].quantile(0.67)

ratings = []
for index in range(len(df_madrid)):
    if df_madrid['Amenities Score'].iat[index] < q3:
        ratings.append('C')
    elif df_madrid['Amenities Score'].iat[index] < q6:
        ratings.append('B')
    else:
        ratings.append('A')
df_madrid['Amenities Rating'] = ratings

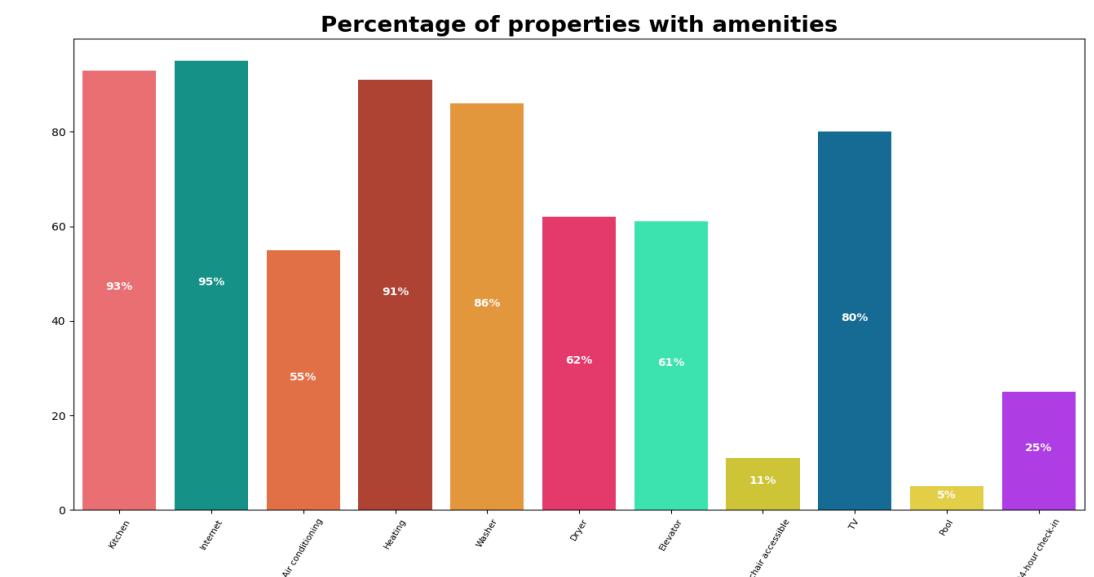
```

Con esta primera limpieza de datos, exportamos un nuevo csv para realizar el paso 4 de visualización con Tableau y, por otra parte, hacer una exploración con Matplotlib en Python.

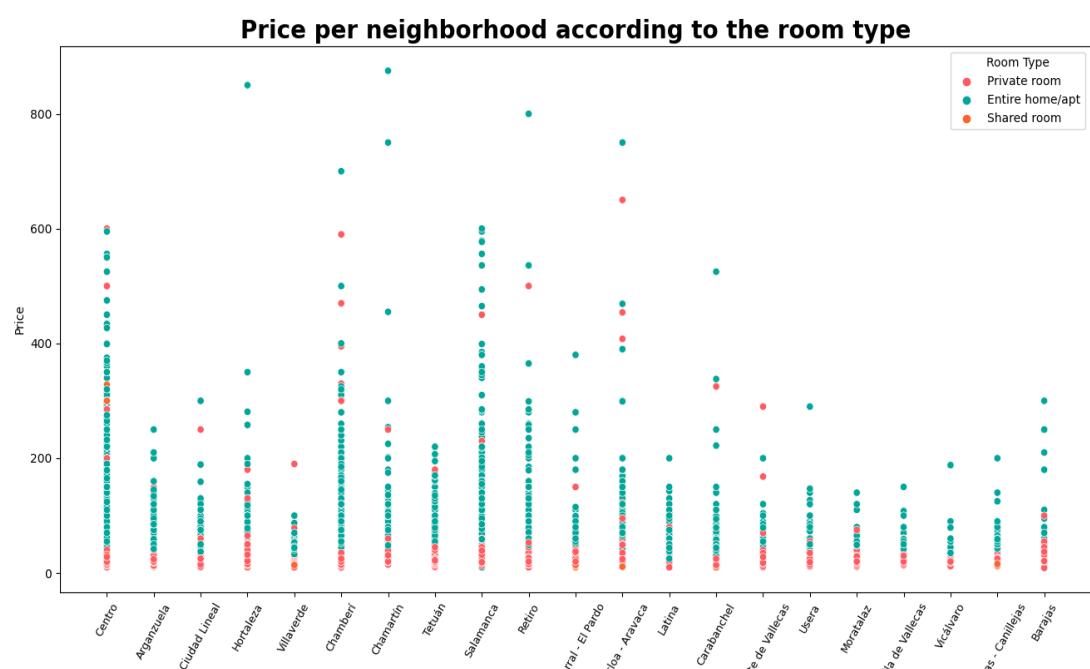
Exploración y visualización con Matplotlib en Python:

Con el fin de continuar entendiendo como estaban compuestos nuestros datos, hicimos los siguientes gráficos:

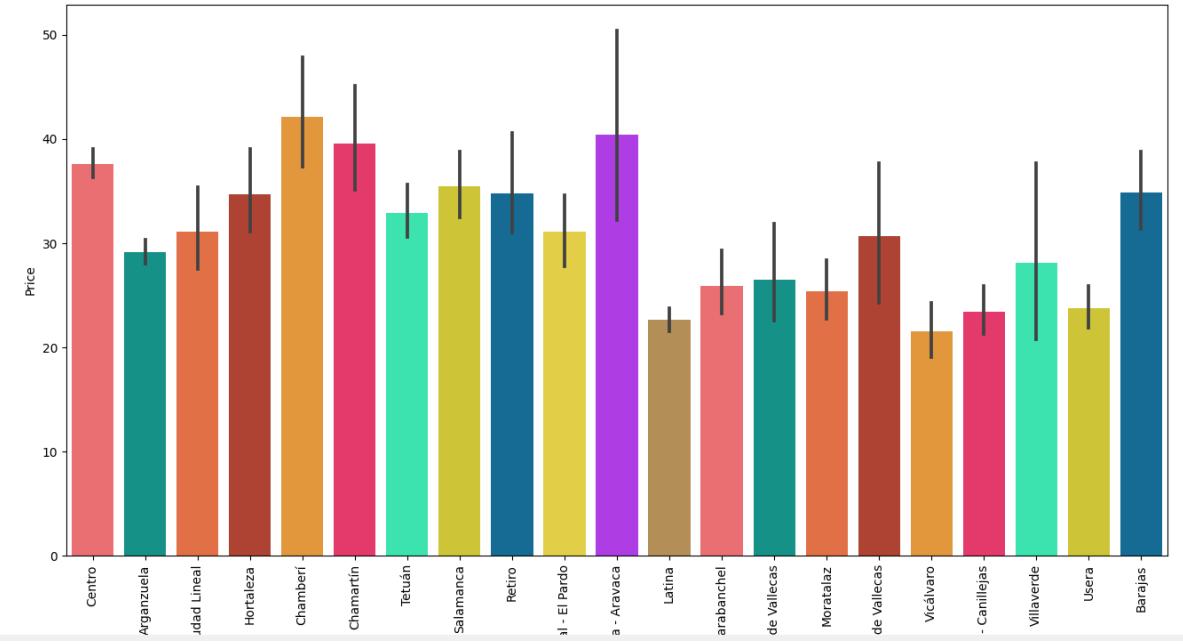
1. ¿En qué porcentaje aparece cada amenity? ¿Esto se ve reflejado en el precio del apartamento?



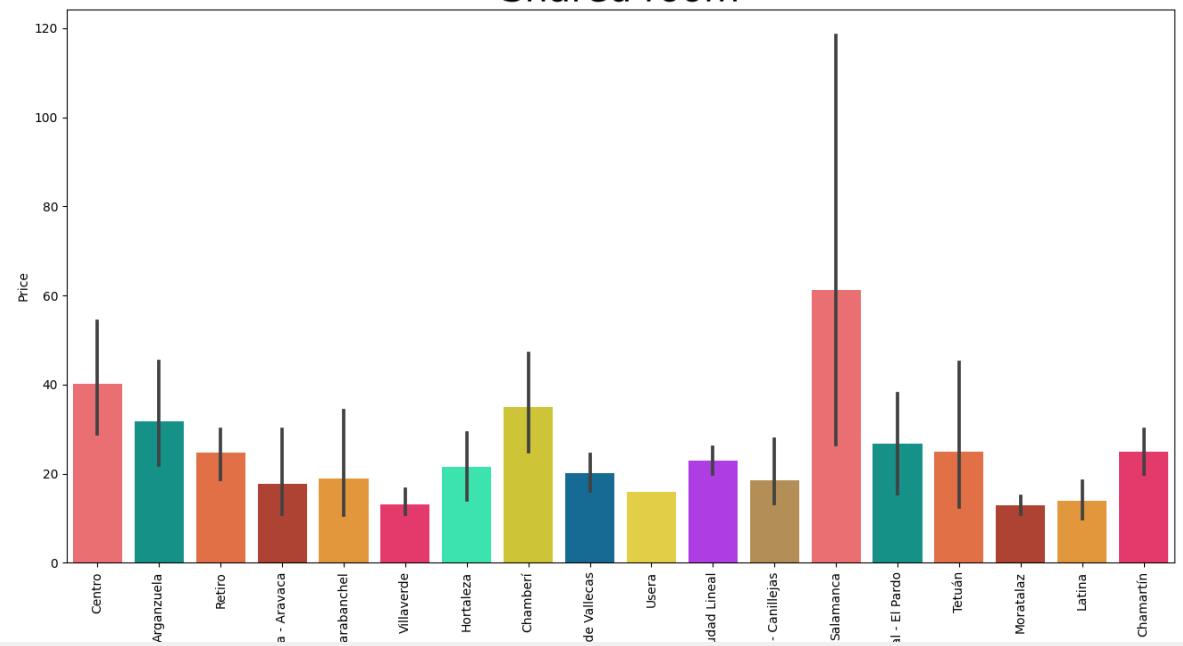
2. Según cada barrio: ¿qué rango de precios maneja y cómo esto se relaciona con el tipo de habitación?



Private room

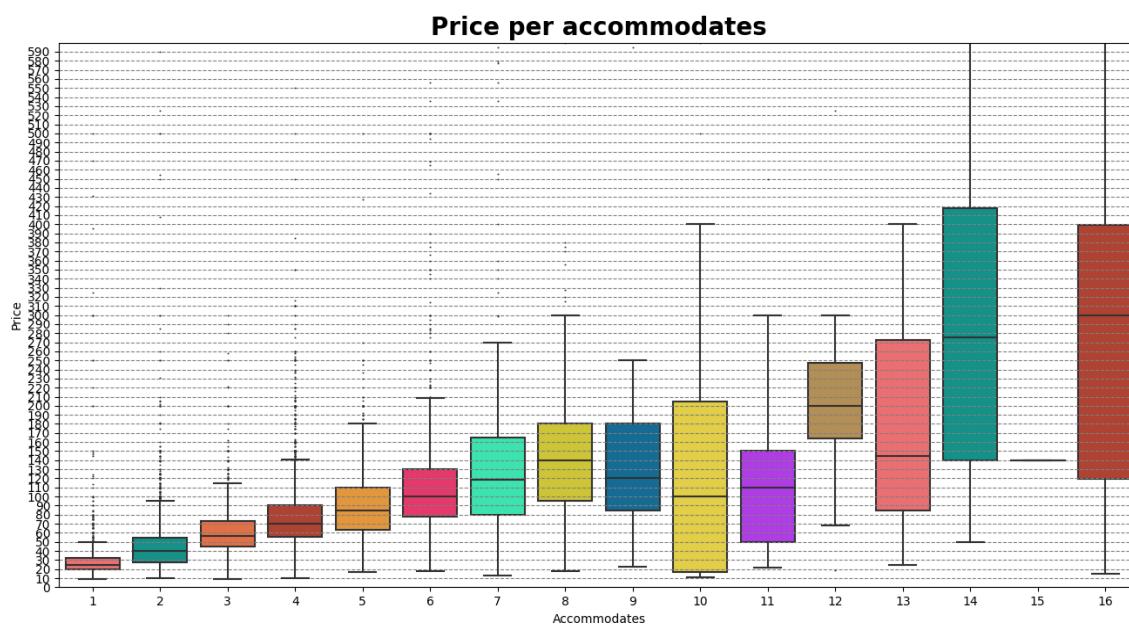
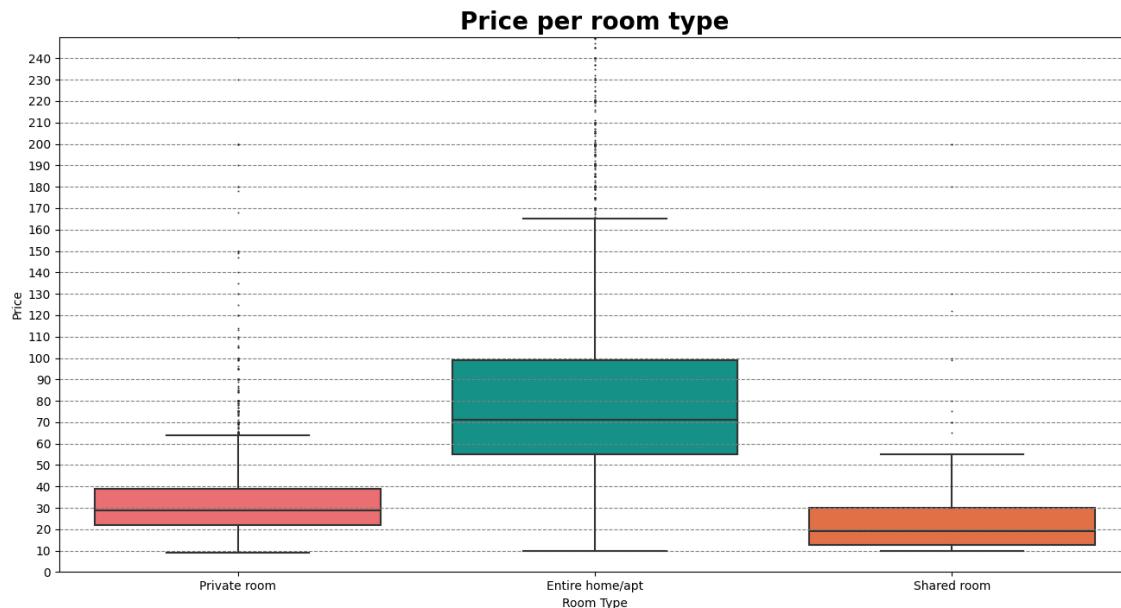


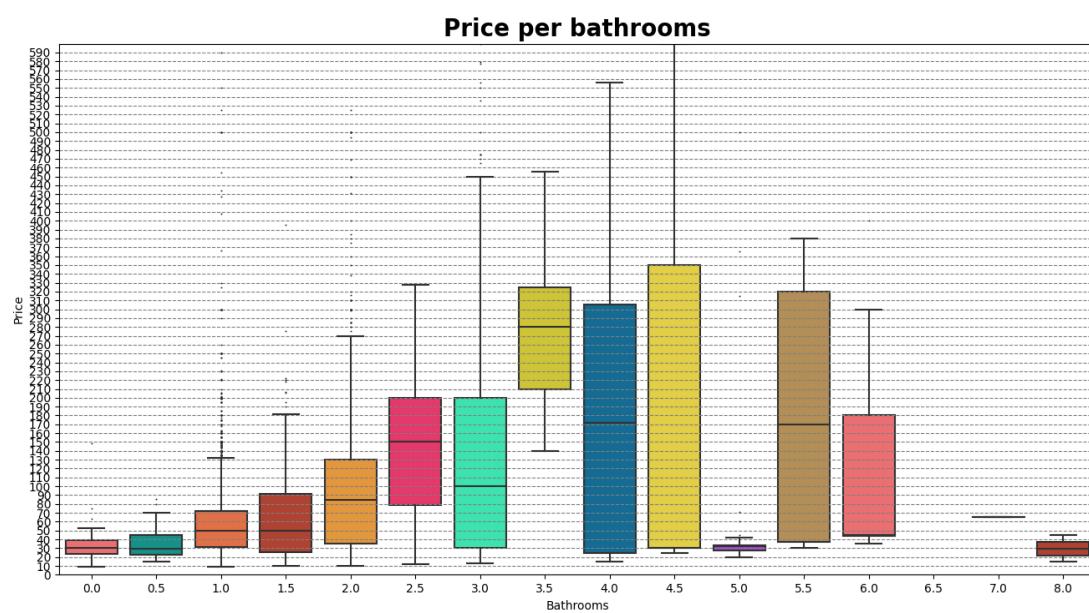
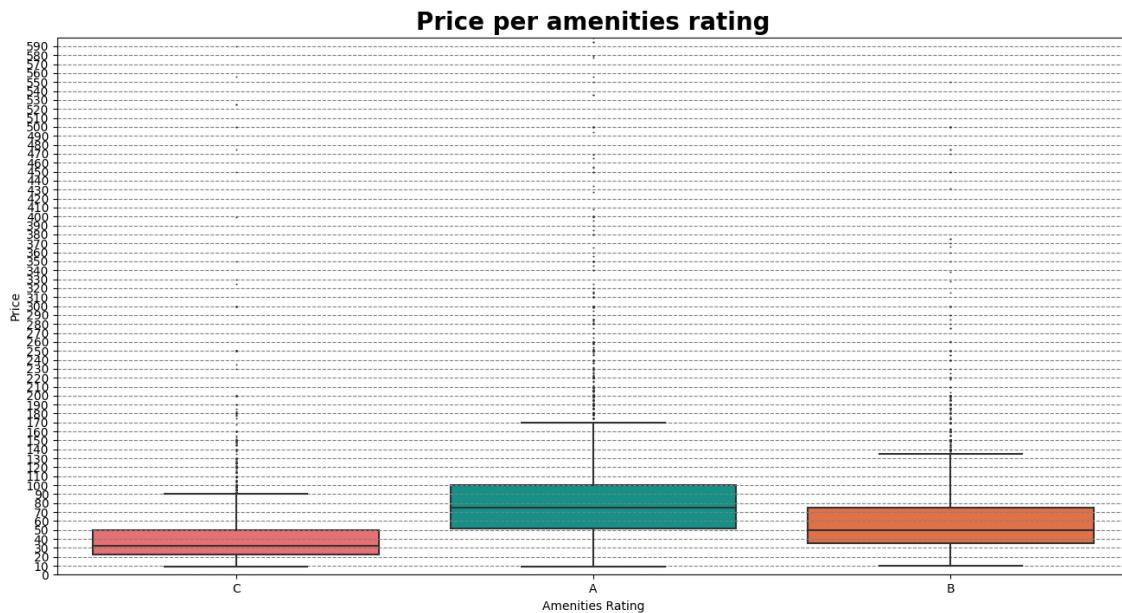
Shared room

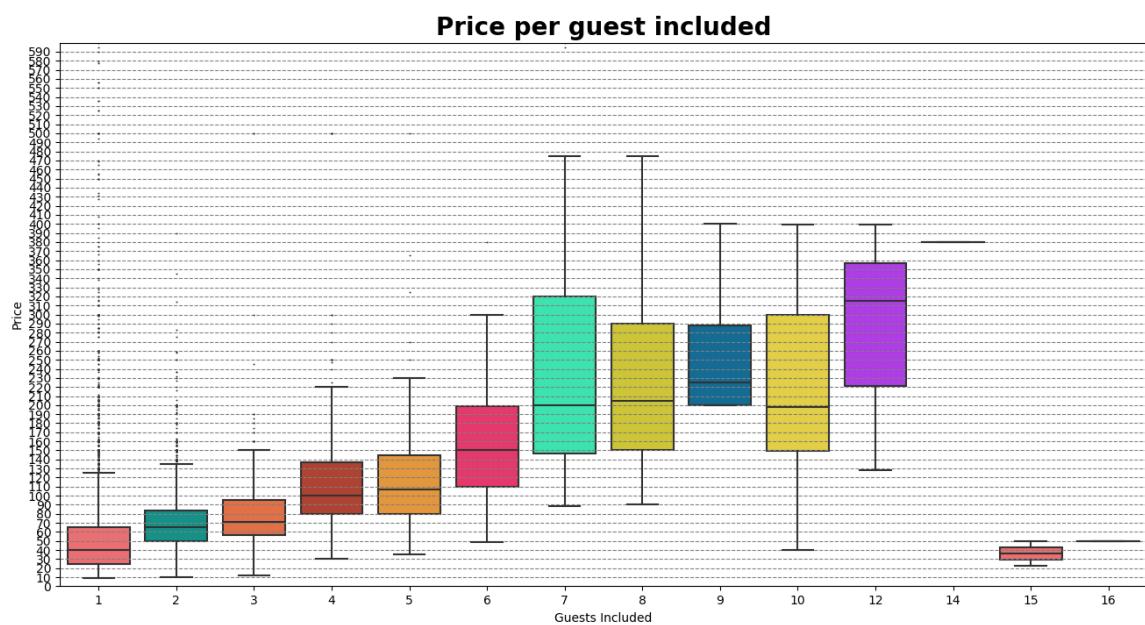
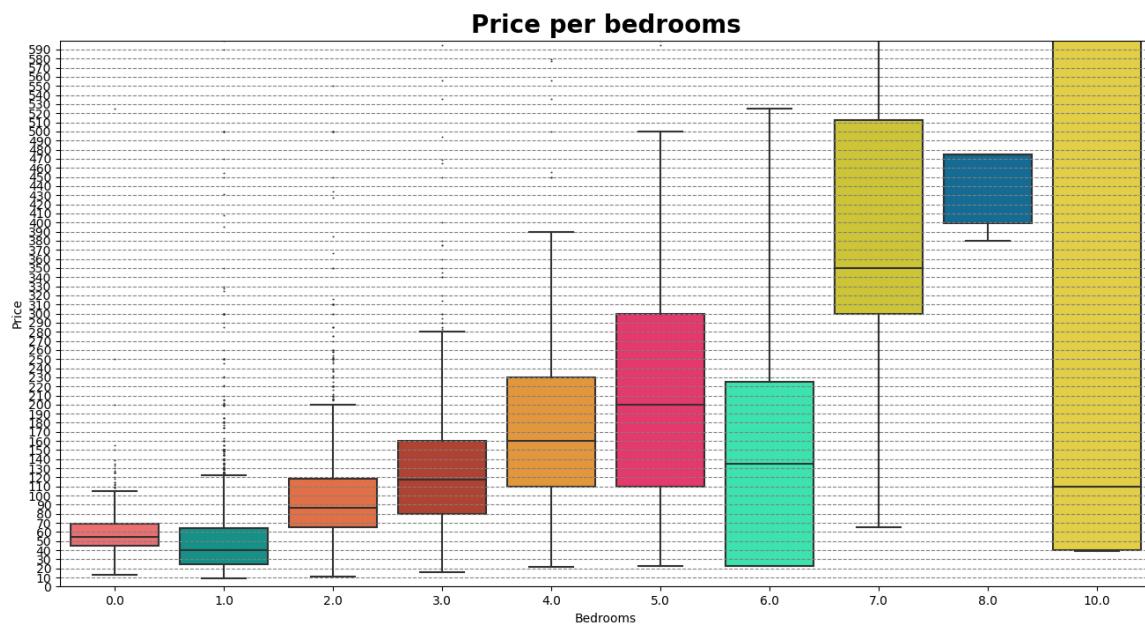


3. ¿Cuáles son nuestros outliers?

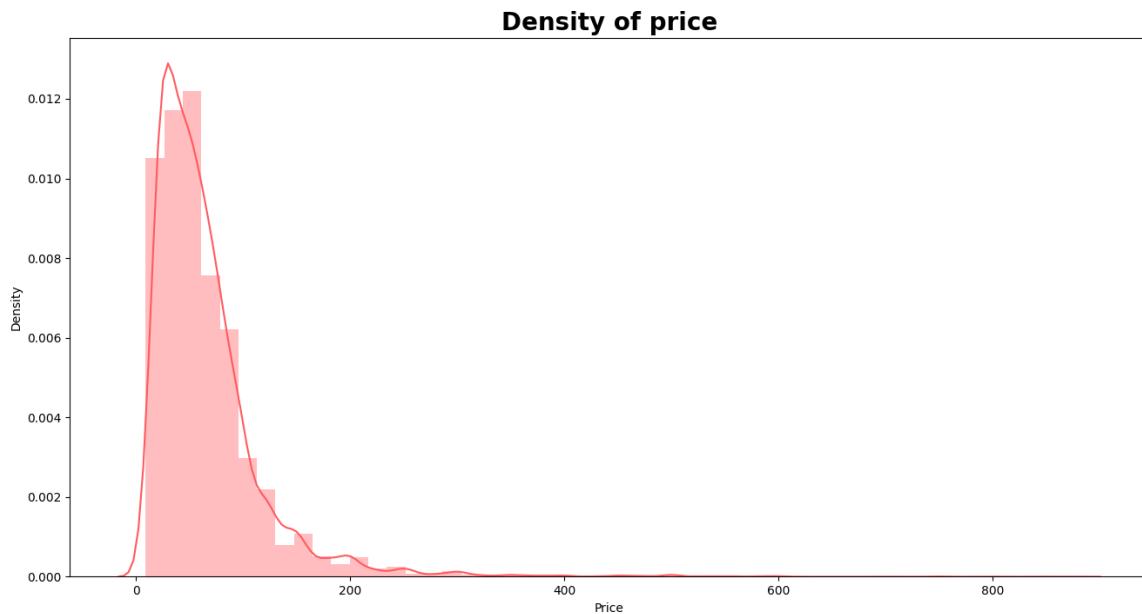
Para eso creamos diferentes boxplot en relación a *Price* como eje Y para poder detectar outliers y eliminarlos del train set antes de entrenar el modelo.







4. Una vez eliminados nuestros outliers, ¿estos se corresponden en general con la densidad de nuestra columna Price?



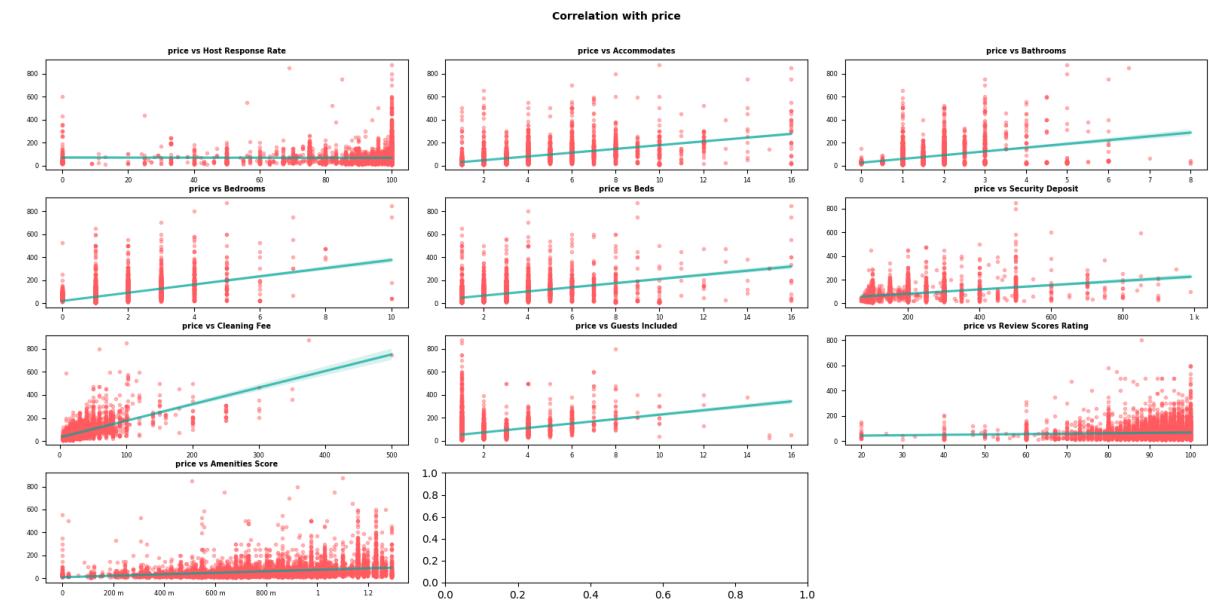
5. ¿Cuál es la correlación de las variables categóricas y numéricas?

Según la matriz de correlación podemos observar que las variables que más influyen sobre nuestro precio (*Price* por nombre de columna) son *Cleaning Fee*, *Accommodates* y *Bedrooms*. Si bien están apenas superando el 0.50, nos hace notar que no es un único factor que pesa más que otros para poder predecir el precio del apartamento, sino que son todas ellas en conjunto, junto con otras variables quienes pueden llegar a darnos un valor aproximado.



Host Response Rate	1	0.062	-0.037	0.012	0.035	-0.008	-0.027	0.023	0.066	0.098	0.046
Accommodates	0.062	1	0.33	0.68	0.82	0.58	0.17	0.38	0.58	-0.061	0.31
Bathrooms	-0.037	0.33	1	0.42	0.39	0.35	0.19	0.32	0.19	0.01	0.1
Bedrooms	0.012	0.68	0.42	1	0.69	0.53	0.21	0.38	0.44	0.01	0.16
Beds	0.035	0.82	0.39	0.69	1	0.49	0.17	0.35	0.46	-0.053	0.2
Price	-0.008	0.58	0.35	0.53	0.49	1	0.36	0.66	0.37	0.056	0.34
Security Deposit	-0.027	0.17	0.19	0.21	0.17	0.36	1	0.48	0.075	0.071	0.11
Cleaning Fee	0.023	0.38	0.32	0.38	0.35	0.66	0.48	1	0.2	0.0048	0.26
Guests Included	0.066	0.58	0.19	0.44	0.46	0.37	0.075	0.2	1	0.0087	0.23
Review Scores Rating	0.098	-0.061	0.01	0.01	-0.053	0.056	0.071	0.0048	0.0087	1	0.15
Amenities Score	0.046	0.31	0.1	0.16	0.2	0.34	0.11	0.26	0.23	0.15	1

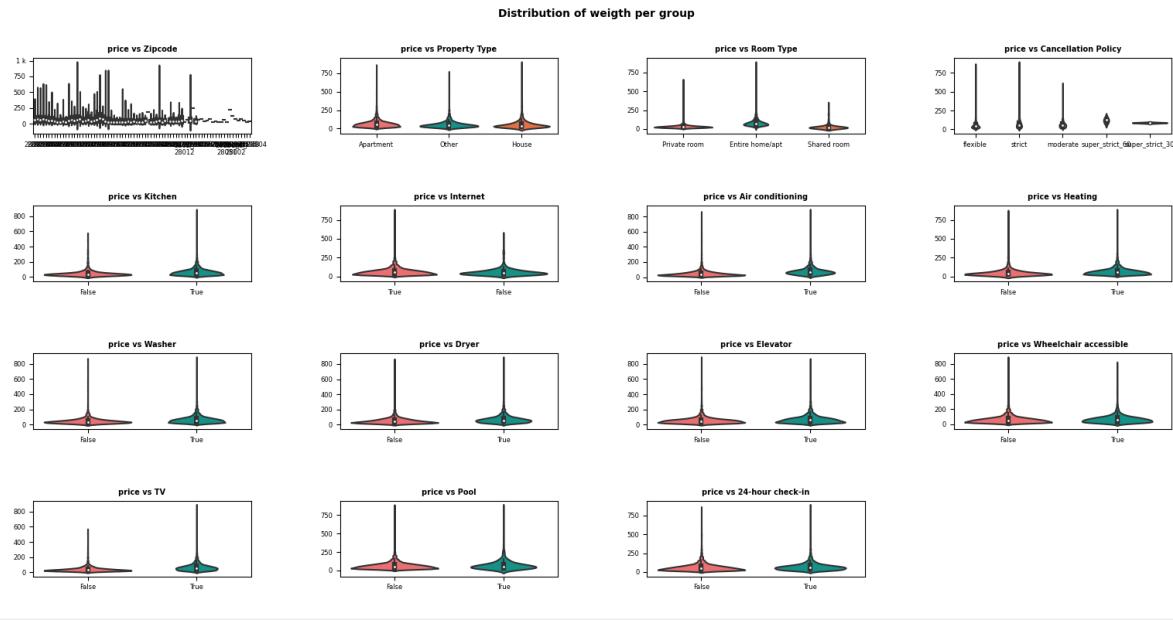
En estos gráficos que representan las correlaciones de las diferentes columnas con *Price*, podemos notar cómo, si bien en tendencia forman una línea creciente entre X e Y, los datos están desparramados en todo el eje Y dificultando que la correlación sea completamente lineal.



Si, además, junto con estos dos gráficos anteriores de matriz y correlación directa con *Price*, sumamos al análisis como se distribuye el peso de nuestras variables categóricas en



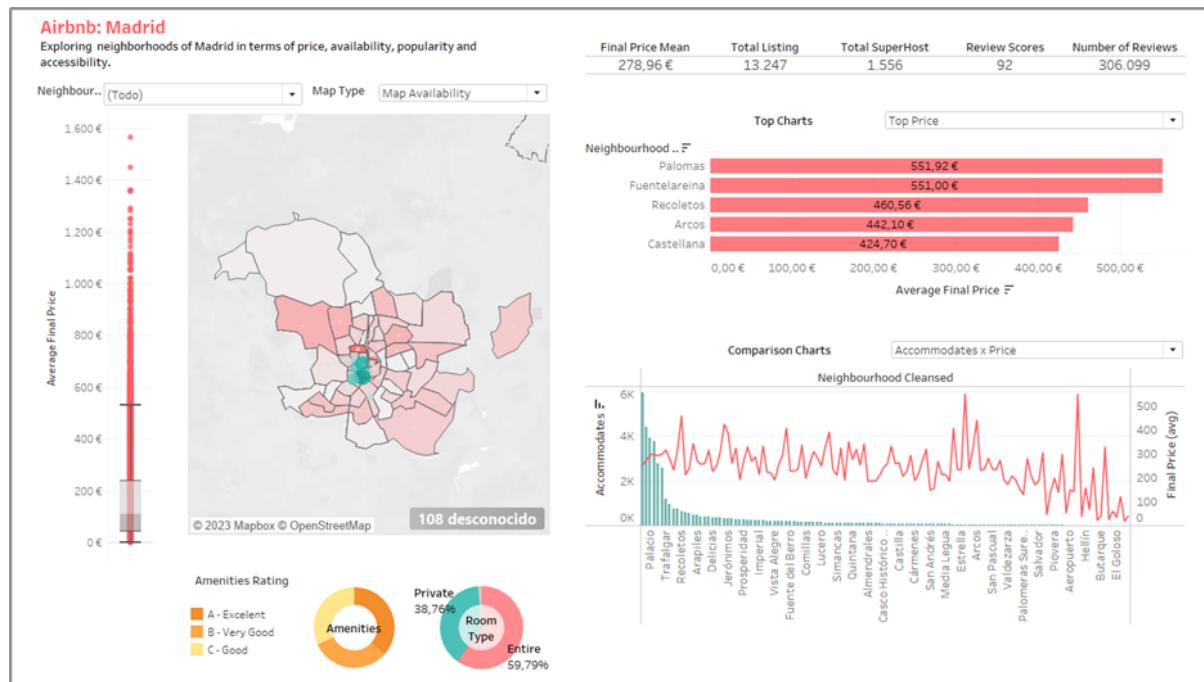
relación con el precio vemos que no hay una tendencia significativa que lo afecte en ninguno de los casos.



4. Visualización de las métricas

Dashboard “Airbnb: Madrid”

Explorando los barrios en Madrid según precio, disponibilidad, popularidad y accesibilidad.



Antes de empezar con el diseño y la selección de variables y KPIs relevantes, decidimos concentrarnos en qué tipo de gráficos queríamos representar en nuestro “dashboard”. En función de la base de datos que manejamos (base de datos que fue limpiada en pasos anteriores y cuyo enfoque sería la ciudad de Madrid, España), la calidad de los datos y lo que intentábamos comunicar, concluimos en que la representación exploratoria de datos sería la indicada ya que nos permitiría analizar los datos, visualizar posibles tendencias, identificar patrones o simplemente dar a conocer el comportamiento de las variables al relacionarse entre sí.

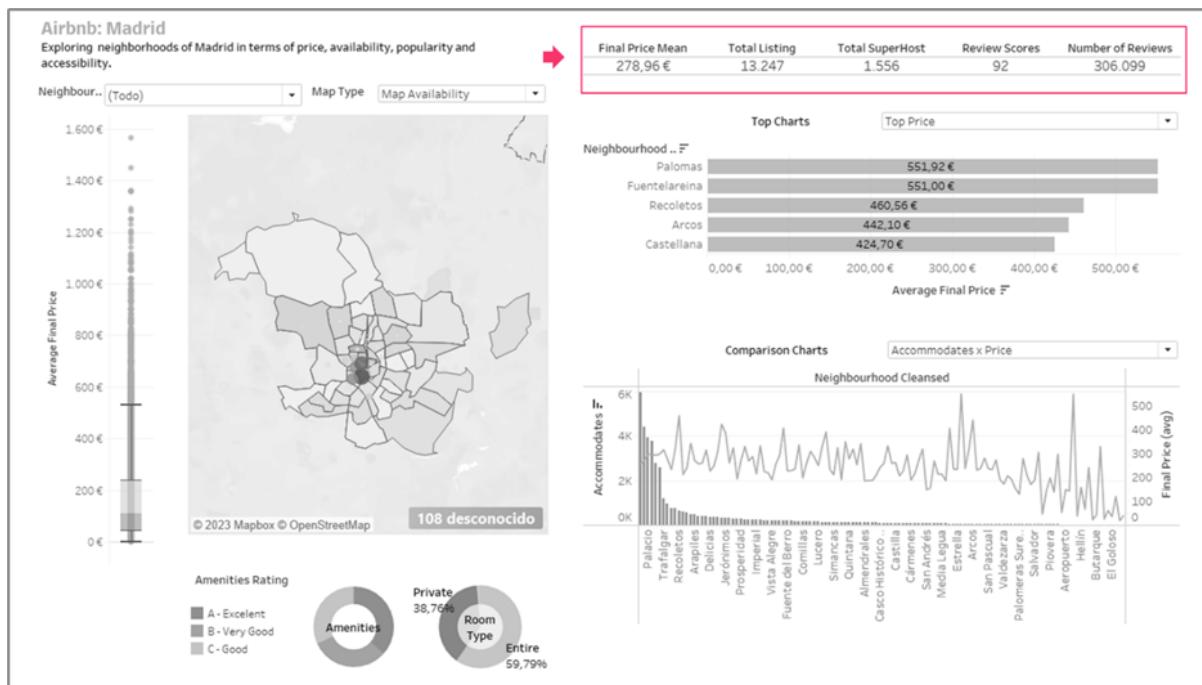
Nuestro primer acercamiento fue sobre el papel, intercambiando ideas, dudas e hipótesis lo que nos permitió llegar a la conclusión de que nuestros principales indicadores para explorar los barrios de Madrid serían:

- Precio
- Disponibilidad
- Popularidad
- Accesibilidad

Elaboración de gráficos:

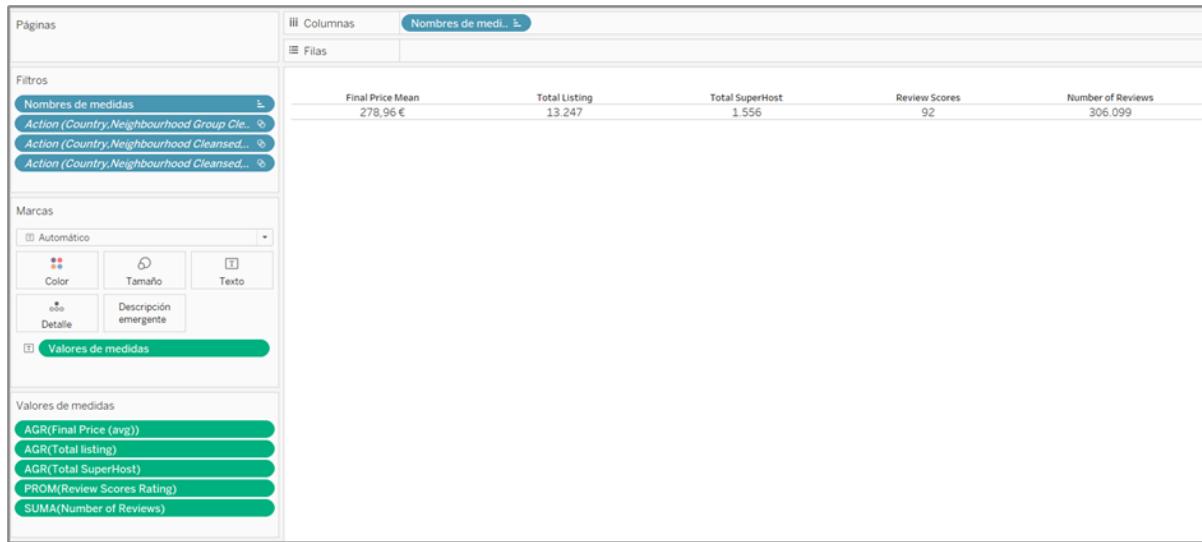
1. Resumen de datos:

Creímos conveniente mostrar un resumen de nuestra base de datos a través de una barra informativa dando a conocer el promedio de precio final, la cantidad total de propiedades encontradas, la cantidad de superhost, la calificación promedio de las propiedades y el número total de calificaciones.



Con esto, podemos dimensionar los valores que reflejan el resto de los gráficos y ser capaces de entender el peso de estos.

Para el filtro en columnas se utilizó el nombre de las distintas medias que fueron creadas bajo marca de texto:



The screenshot shows a data visualization interface with the following sections:

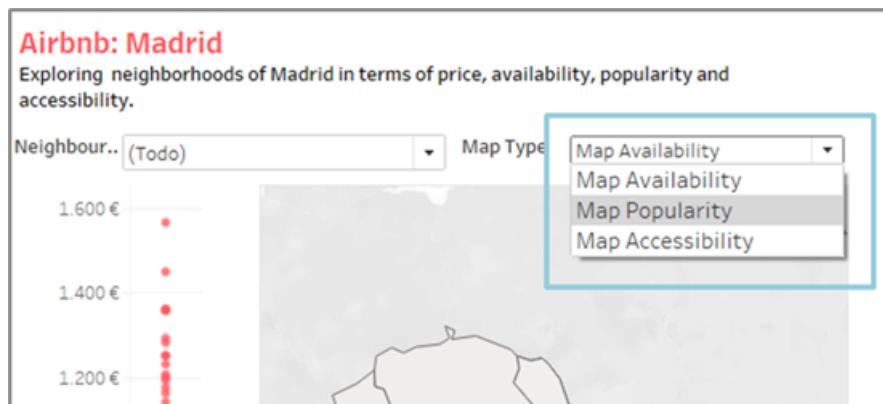
- Páginas:** Shows 'Nombres de medida' selected.
- Filtros:** Shows 'Nombres de medidas' expanded, listing 'Action (Country,Neighbourhood Group Cle...' and 'Action (Country,Neighbourhood Cleansed...)'.
- Valores de medidas:** Shows 'Valores de medidas' selected, listing 'AGR(Final Price (avg))', 'AGR(Total listing)', 'AGR(Total SuperHost)', 'PROM(Review Scores Rating)', and 'SUMA(Number of Reviews)'.
- Resumen:** Displays summary statistics:

	Final Price Mean	Total Listing	Total SuperHost	Review Scores	Number of Reviews
	278,96 €	13.247	1.556	92	306.099

1. Gráfico Espacial:

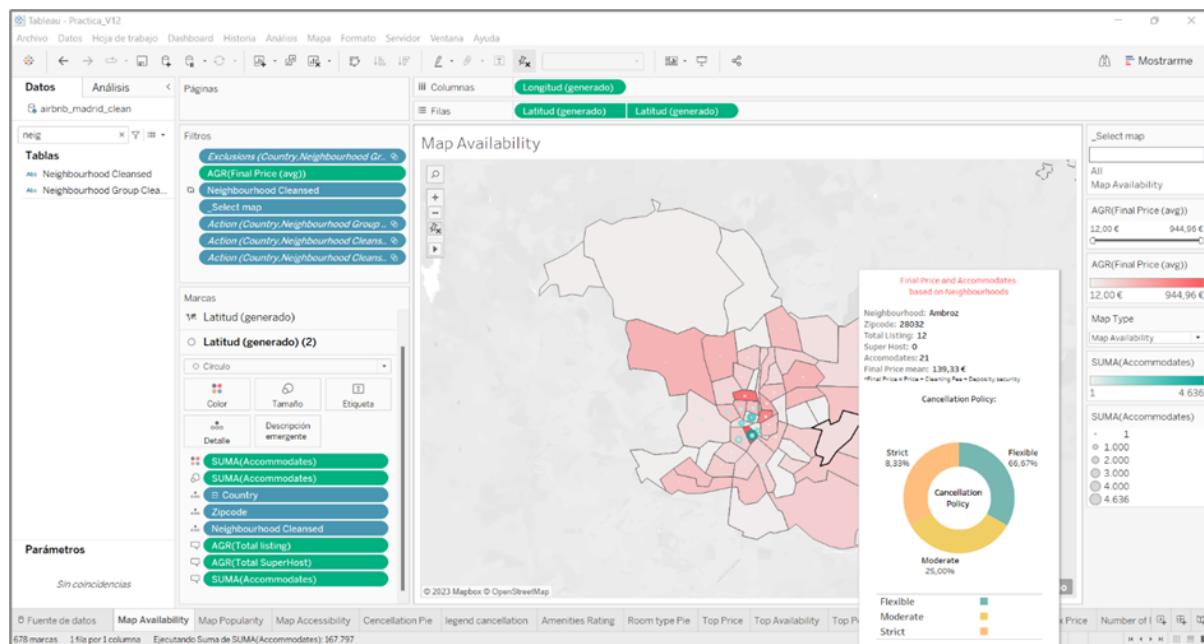
Nuestro objetivo principal y gráfico más importante es el espacial, dándole la posibilidad al usuario de ubicarse geográficamente y ver la magnitud de ciertas variables en relación con el espacio, permitiendo utilizarla como filtro clicando sobre un barrio en el mapa o simplemente seleccionando en el filtro “Neighbourhood”.

Elaboramos 3 distintos gráficos espaciales siguiendo la elección de nuestros KPIs. Se puede acceder a estos 3 gráficos dentro del dashboard principal a través de un desplegable de lista compacta:



Mapa de disponibilidad: Map Availability

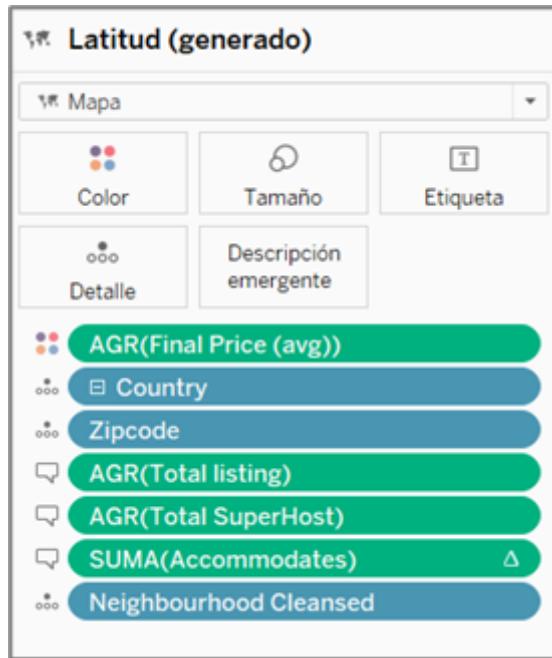
La finalidad de este gráfico es mostrar la cantidad total de personas que pueden ser ubicadas en los distintos barrios de Madrid y la variación de precio que existe entre ellos.



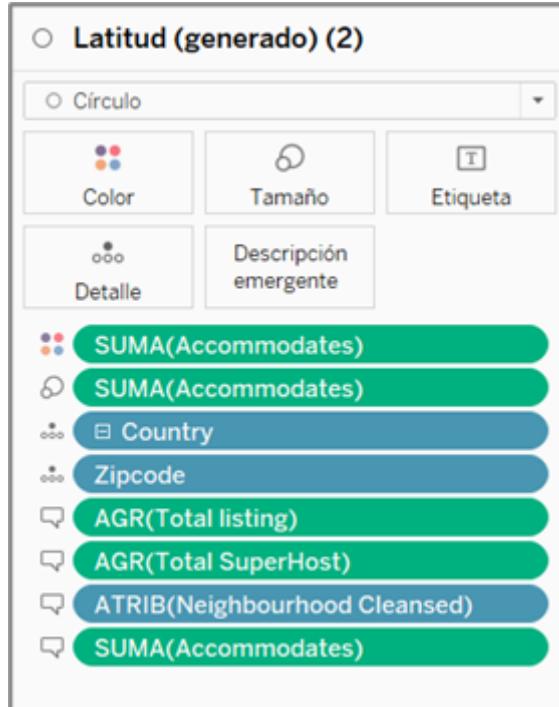
Como filtros de columnas utilizamos las variables geográficas de “Longitud (Generado)” y para las filas, “Latitud (Generado)” en eje doble para poder solapar mapas, uno sobre otro.



En el primer mapa, se consideró el promedio de variable calculada “Precio Final” (precio, recargo de limpieza y depósito de seguridad) dentro de marcas de color, el Zipcode de los barrios dentro de la jerarquía creada: geolocalización y la columna de barrios como parte del detalle.



Y para el segundo, la suma de las “accommodates” o cantidad de personas que pueden ser ubicadas por barrios a través de marcas de tamaño y color, el Zipcode de los barrios dentro de la jerarquía creada: geolocalización y la columna de barrios como parte del detalle.

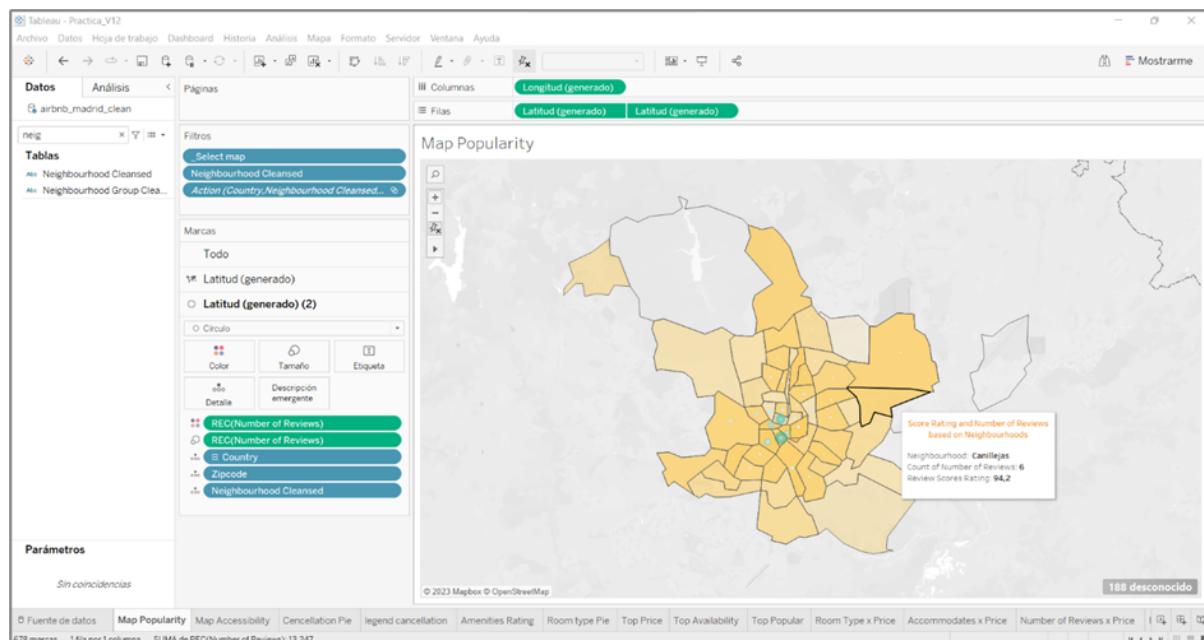


Además, aprovechamos la oportunidad de añadir dentro de la marca de descripción emergente, otras variables y gráficos que dieran información adicional al usuario:

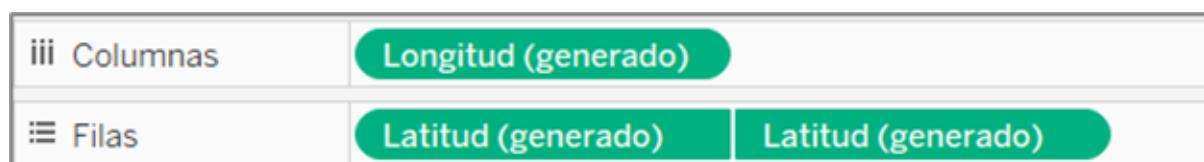
- Nombre del barrio
- Zip Code
- Cantidad total de “listing” dentro del barrio
- Cantidad de Superhost
- Cantidad de “accommodates”
- Promedio de precio final
- Gráfico de dona: Política de Cancelación

Mapa de popularidad: Map Popularity

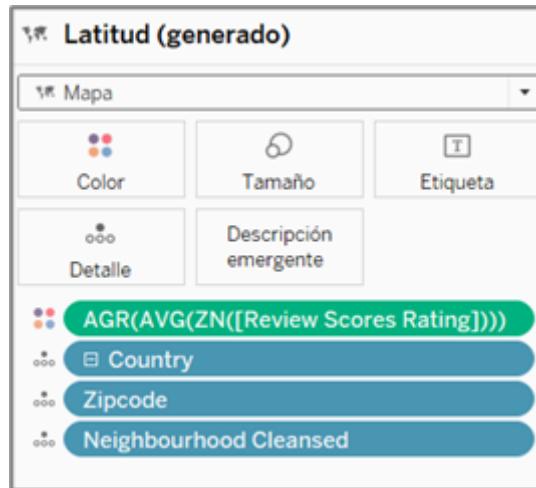
Este mapa nos permite representar la popularidad de los barrios de Madrid a través de la relación que existe entre la calificación promedio y la cantidad de calificaciones que han recibido por parte de los usuarios.



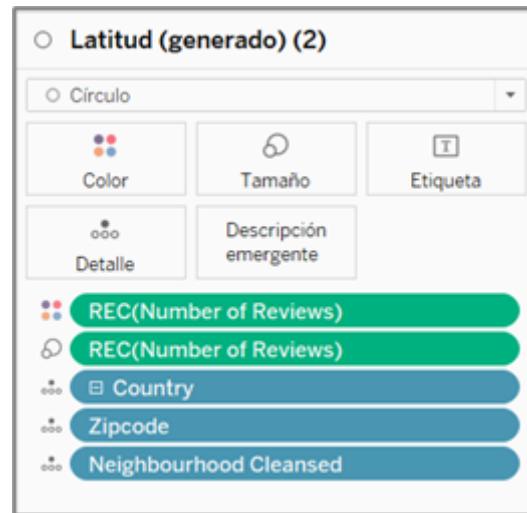
Como filtros de columnas utilizamos las variables geográficas de “Longitud (Generado)” y para las filas, “Latitud (Generado)” en eje doble para poder solapar mapas, uno sobre otro.



En el primer mapa, se consideró el promedio de la calificación de los distintos barrios en marca de color, el Zipcode de los barrios dentro de la jerarquía creada: geolocalización y la columna de barrios como parte del detalle.



Y para el segundo, el recuento o cantidad de calificaciones que han recibido los barrios en marca de tamaño y color, el Zipcode de los barrios dentro de la jerarquía creada: geolocalización y la columna de barrios como parte del detalle.

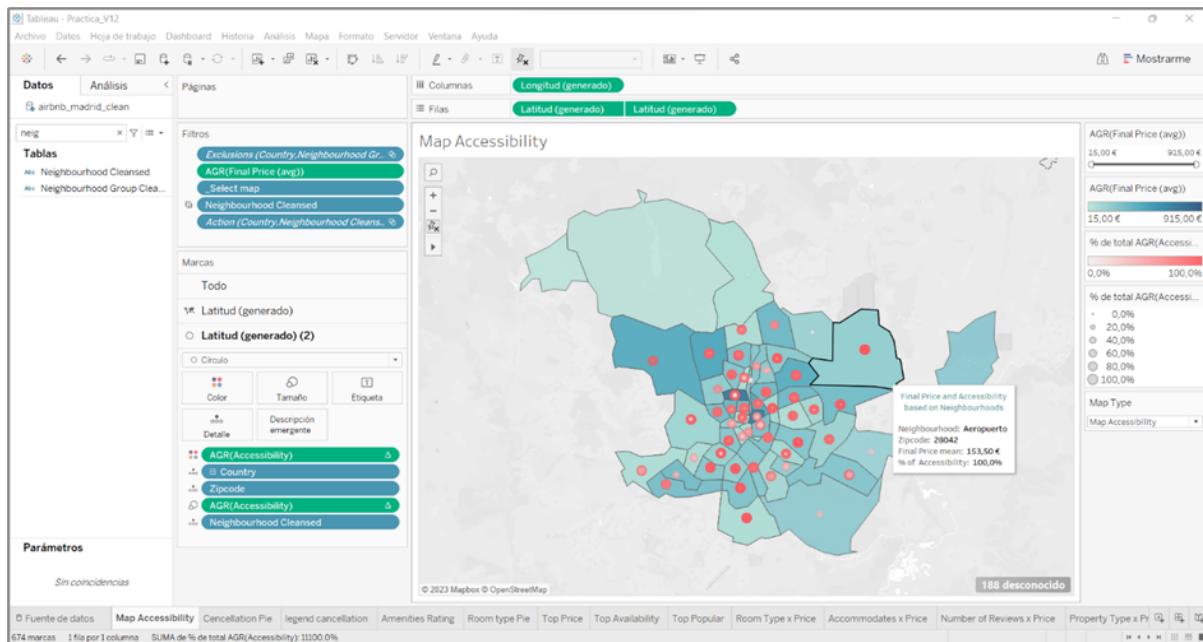


Adicionalmente, aprovechamos la oportunidad de añadir dentro de la marca de descripción emergente un resumen de la información mostrada en el mapa:

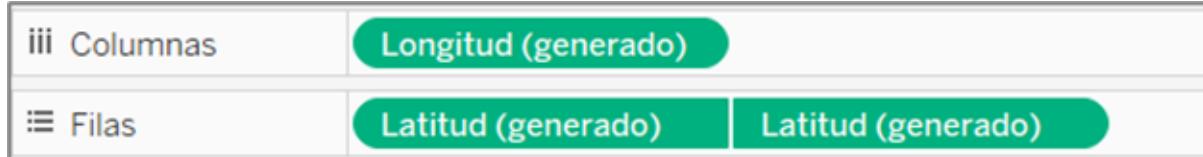
- Nombre del barrio
- Cantidad de calificaciones
- Calificación promedio

Mapa de accesibilidad: Map Accessibility

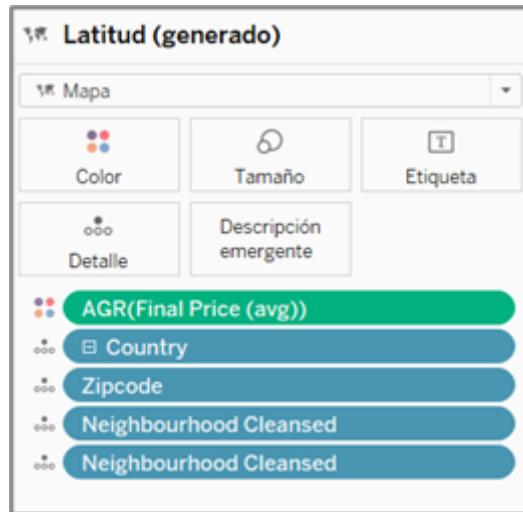
En este tercer mapa, buscamos mostrar el porcentaje promedio de accesibilidad de las distintas ofertas que hay en cada barrio relacionándolo con el precio final.



Como filtros de columnas utilizamos las variables geográficas de “Longitud (Generado)” y para las filas, “Latitud (Generado)” en eje doble para poder solapar mapas, uno sobre otro.



En el primer mapa, se consideró el promedio de variable calculada “Precio Final” (precio, recargo de limpieza y depósito de seguridad) dentro de marcas de color, el Zipcode de los barrios dentro de la jerarquía creada: geolocalización y la columna de barrios como parte del detalle.



Y para el segundo, el promedio en porcentaje de la variable calculada “accesibilidad” (que considera la suma la cantidad de ofertas con accesibilidad a silla de ruedas y elevadores dentro de cada barrio) como marca de tamaño y color, el Zipcode de los barrios dentro de la jerarquía creada: geolocalización y la columna de barrios como parte del detalle.



Adicionalmente, se activa la descripción emergente que resume la información mostrada en el mapa:

- Nombre del barrio
- Zip Code
- Promedio de precio final
- Promedio de porcentaje de accesibilidad

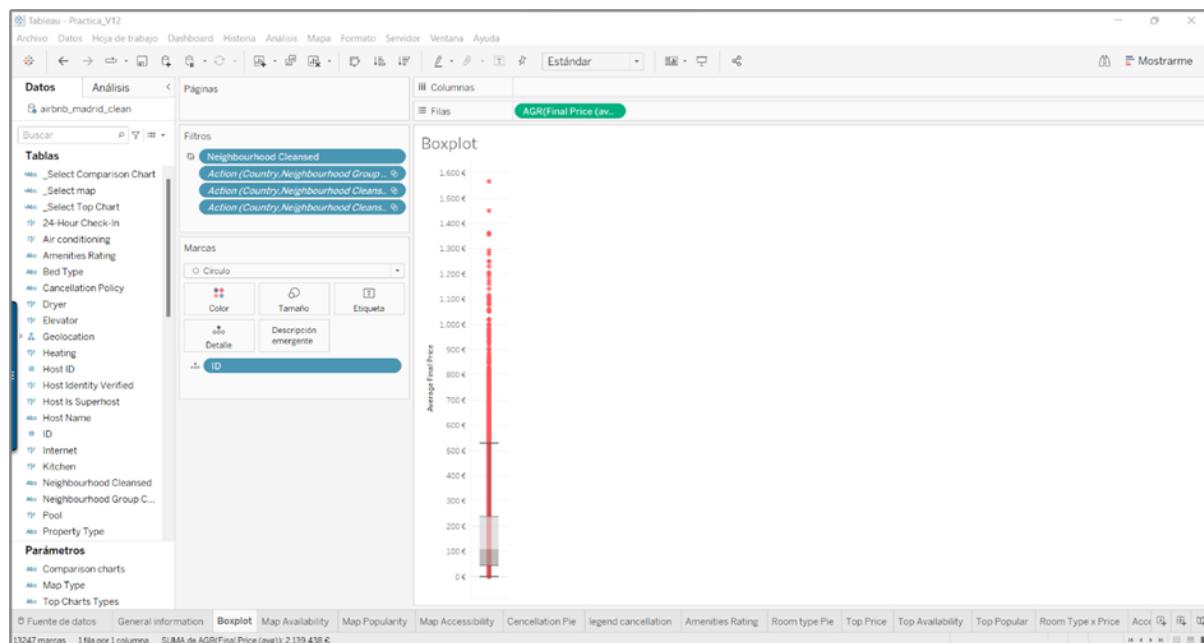
Conclusión:

Buscamos dar flexibilidad y dinamismo al usuario para que este pueda elegir un barrio de una forma más interactiva y visual a través de los distintos mapas y así, explorar los barrios de Madrid con mayor facilidad, permitiéndole además tener información para comparar sus características tanto con otros barrios como también de forma general.

3. Gráfico o diagrama de cajas y bigotes (boxplot)

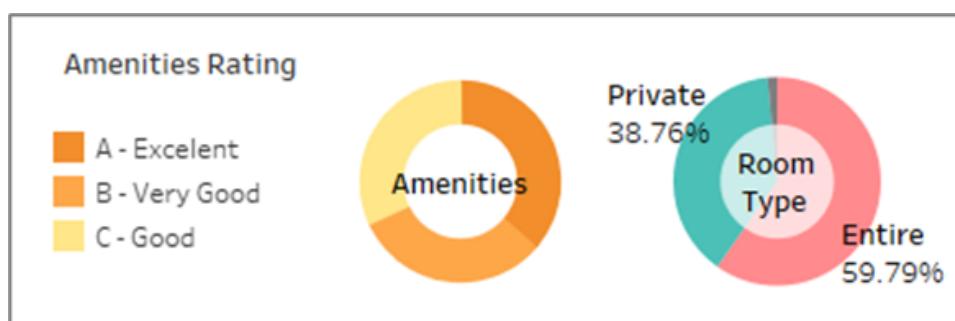
Este gráfico funciona de forma dinámica según un barrio elegido en el mapa o en el filtro “Neighbourhood”.

Podemos observar aspectos de distribución de cada piso por barrio en relación con el precio final. Su importancia es comprobar los outliers que influyen en el precio medio de los barrios, los cuartiles y mediana.



4. Gráficos de donas

Representamos con dos gráficos también conocidos como “gráfico anillo” o “gráfico donuts” que funcionan según un barrio elegido.



Podemos observar a través de estos gráficos características de forma general de cada barrio.

El primero muestra en proporciones la distribución según la clasificación de los amenities, o sea, la clasificación de la comodidad ofrecida:

- A (excelente)
- B (muy bueno)
- C (bueno)

El segundo, es la distribución del tipo de habitación.

5. Gráficos de magnitud: “Top Charts”

Optamos por los gráficos de magnitud, específicamente los diagramas de barras que permiten visualizar a través de la característica del tamaño, la representación de variables relevantes en cada barrio.

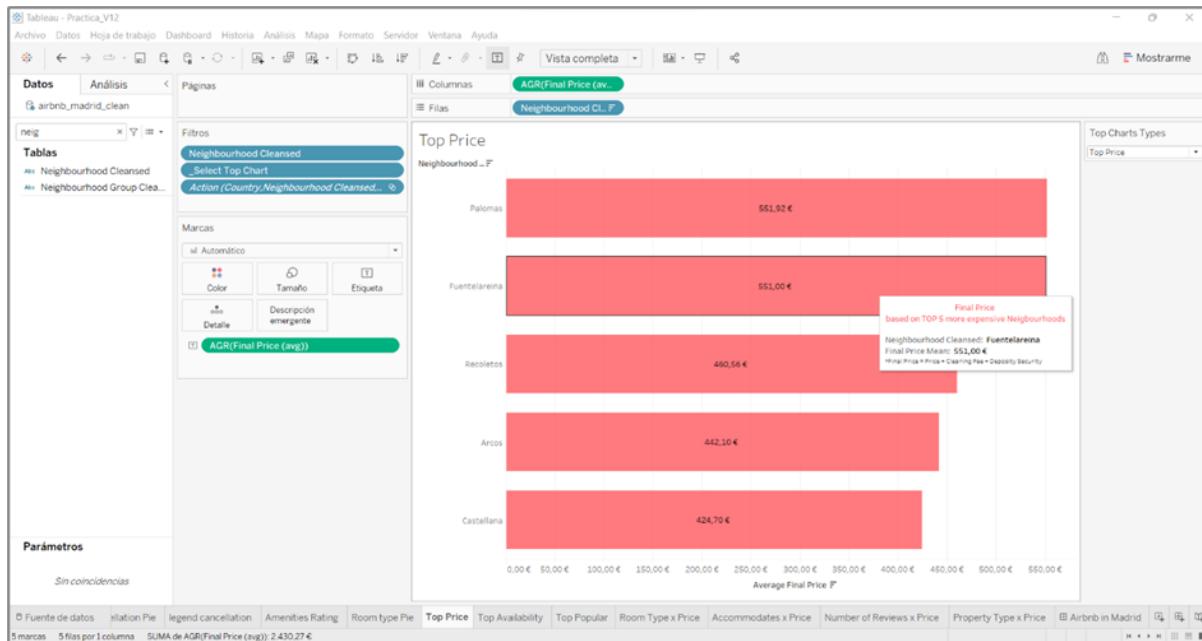
Además, nos ha dado la posibilidad de crear un “TOP 5” comparativo de los barrios de Madrid más relevantes según cada uno de los KPIs seleccionados.

Para esta sección del dashboard, elaboramos 3 distintos gráficos de barras a los cuales se puede acceder dentro del dashboard principal a través de un desplegable de lista compacta:



Diagrama de Precio: Top Chart - Top Price

Este diagrama de barras muestra el promedio de la variable calculada “Precio Final” (precio, recargo de limpieza y depósito de seguridad) en relación con los barrios en Madrid, agrupados en el TOP 5 de los barrios con **más costosos**.



Como filtros de columnas utilizamos el promedio de variable calculada “Precio Final” y para las filas, los distintos barrios.

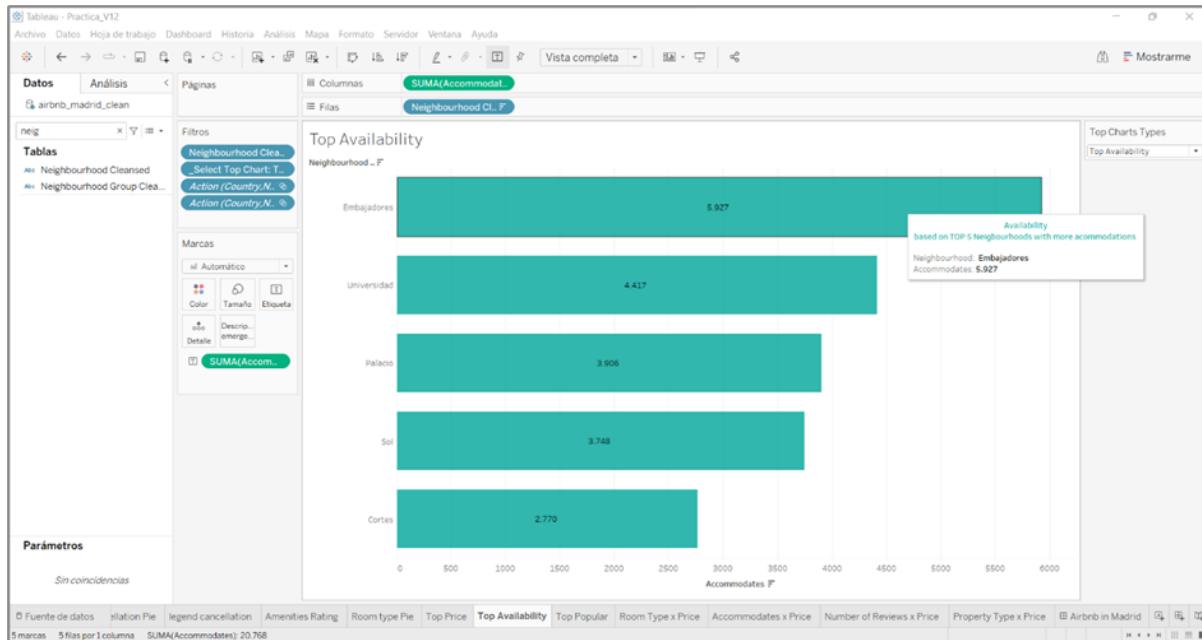
La variable calculada de “Precio Final” lleva marca de texto para facilitar la interpretación del usuario sobre el gráfico.

También se activa la descripción emergente con la siguiente información:

- Nombre del barrio
- Promedio de precio final

Diagrama de Disponibilidad: Top Chart - Top Availability

Este diagrama de barras muestra la cantidad de personas que pueden ser ubicadas en relación con los barrios en Madrid, agrupados en el TOP 5 de los barrios con mayor disponibilidad.



Como filtros de columnas utilizamos la suma de las “accommodates” y para las filas, los distintos barrios.

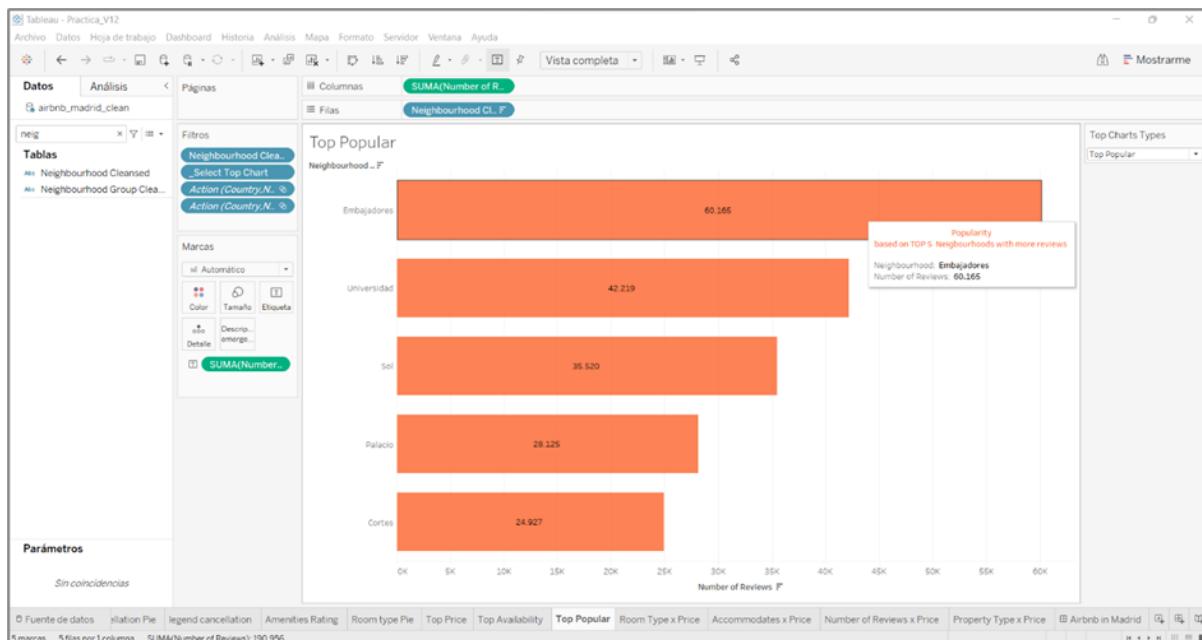
La variable de “accommodates” lleva marca de texto para facilitar la interpretación del usuario.

También se activa la descripción emergente con la siguiente información:

- Nombre del barrio
- Cantidad de “accommodates”

Diagrama de Popularidad: Top Chart - Top Popularity

Este diagrama de barras muestra la cantidad de calificaciones en relación con los barrios en Madrid, agrupados en el TOP 5 de los barrios con mayor **popularidad**.



Como filtros de columnas utilizamos la suma de las calificaciones o “Number of Reviews” y para las filas, los distintos barrios.

La variable de calificaciones o “Number of Reviews” lleva marca de texto para facilitar la interpretación del usuario.

También se activa la descripción emergente con la siguiente información:

- Nombre del barrio
- Cantidad de calificaciones o “Number of Reviews”

6. “Comparison Charts” o diagramas comparativos

Buscando comparar dos o más variables y tratando de descubrir tendencias y relaciones, decidimos realizar gráficos compartidos a través de diagramas de barras y líneas de tendencia. Nuestra principal intención era encontrar relación entre nuestros distintos KPIs, siendo el foco más relevante el “Precio Final” (precio, recargo de limpieza y depósito de seguridad)

Hemos elaborado 4 gráficos distintos, los cuales se pueden seleccionar dentro del dashboard desde un desplegable de lista compacta:

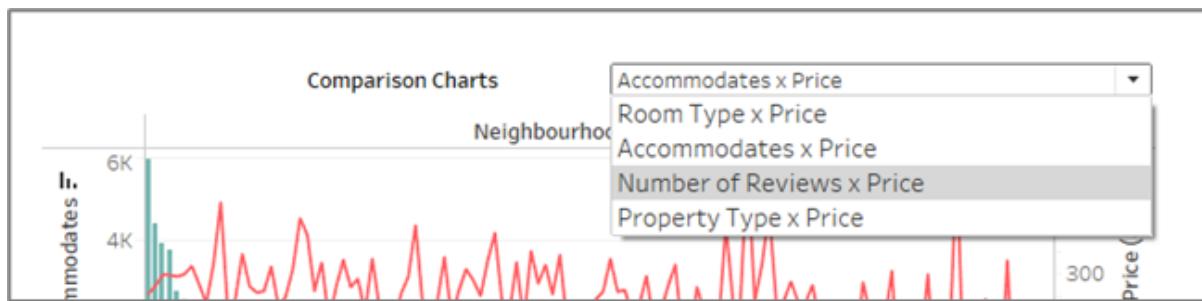
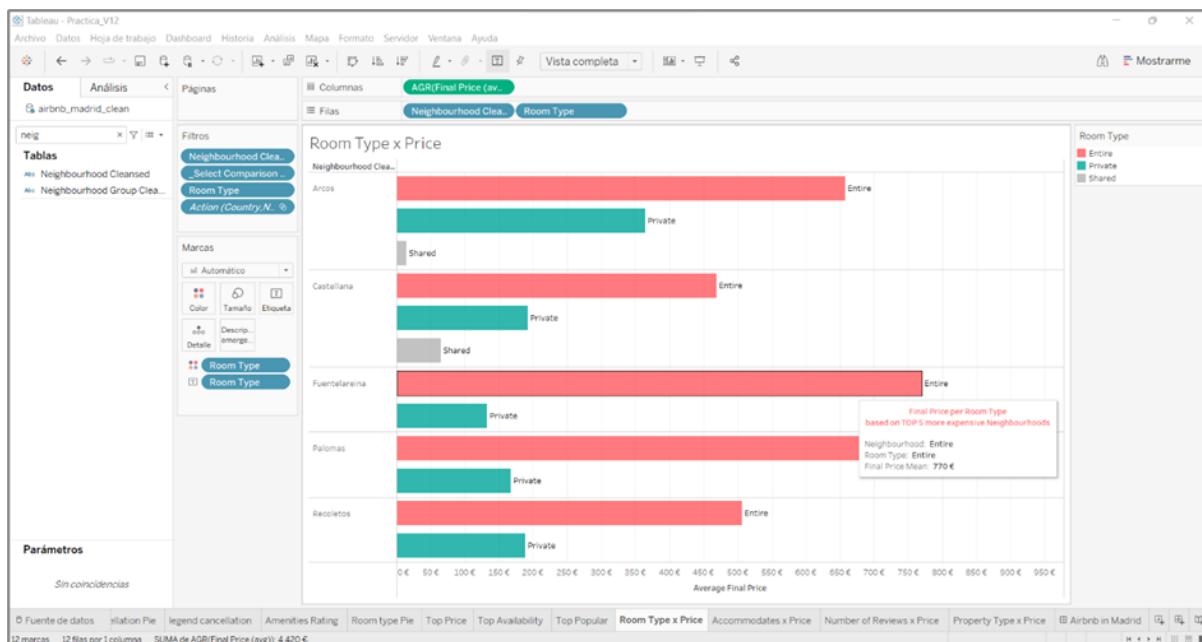


Diagrama comparativo: Room Type x Final Price

Este gráfico de barras permite la comparación del precio según el tipo de habitación para el “TOP 5” de barrios más costosos.



Como filtros de columnas utilizamos el promedio de la variable calculada “Precio Final” (precio, recargo de limpieza y depósito de seguridad) y para las filas, el barrio y el tipo de habitación.

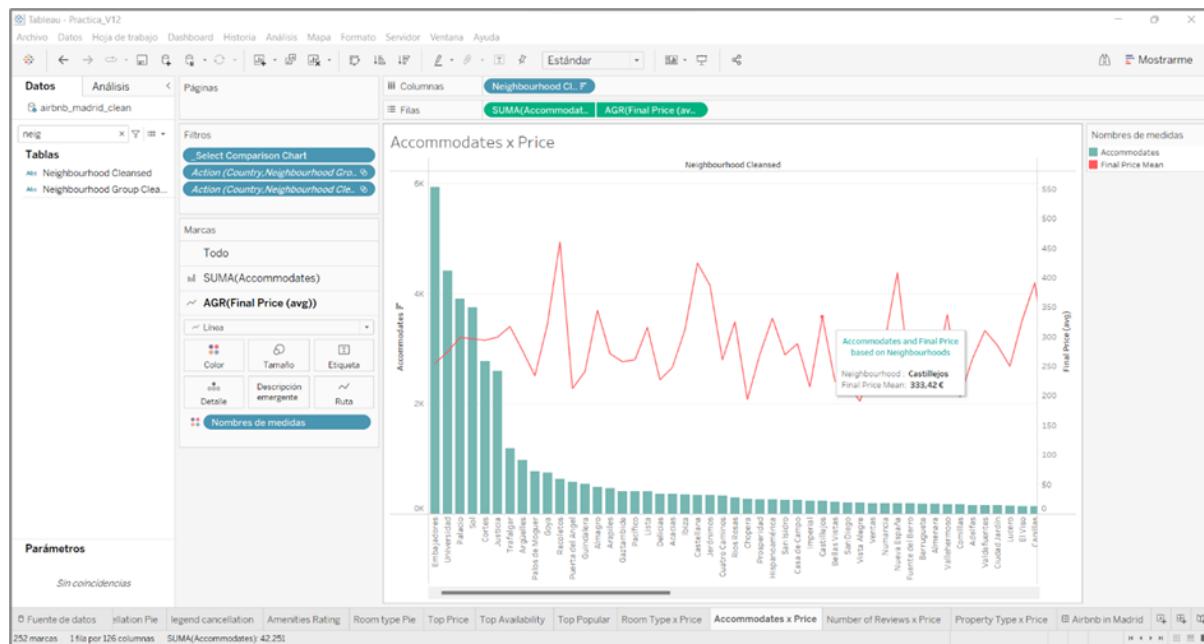
La variable de tipo de habitación lleva marca de color y texto para facilitar la interpretación del usuario.

También se activa la descripción emergente con la siguiente información:

- Nombre del barrio
- Tipo de habitación
- Promedio de precio final

Diagrama comparativo: Accommodates x Final Price

Este gráfico de barras permite ver la tendencia de la línea de precio y la comparación entre los barrios según la cantidad de “accommodates”.



Como filtros de columnas utilizamos los barrios y para las filas, la suma de “accommodates” y la variable calculada “Precio Final” (precio, recargo de limpieza y depósito de seguridad) en eje doble para poder solapar gráficos, uno sobre otro.

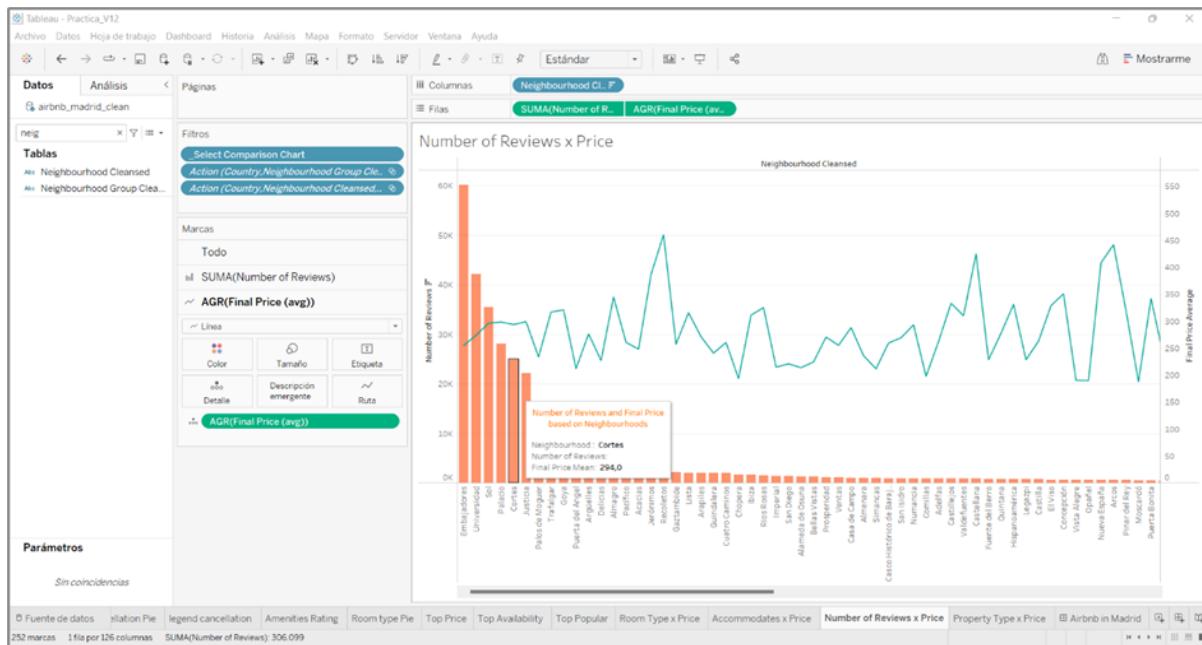
Las variables de precio final y “accommodates” en este gráfico llevan marca de color, adicionalmente precio final se representa con marca de línea sobre el gráfico para evaluar relación y tendencia.

También se activa la descripción emergente con la siguiente información:

- Nombre del barrio
- Promedio de precio final
- Cantidad de accommodates

Diagrama comparativo: Number of Reviews x Final Price

Este gráfico de barras permite ver la tendencia de la línea de precio y la comparación entre los barrios según la cantidad de total de calificaciones o “Number of Reviews”.



Como filtros de columnas utilizamos los barrios y para las filas, la suma de la cantidad de calificaciones o “Number of Reviews” y la variable calculada “Precio Final” (precio, recargo de limpieza y depósito de seguridad) en eje doble para poder solapar gráficos, uno sobre otro.

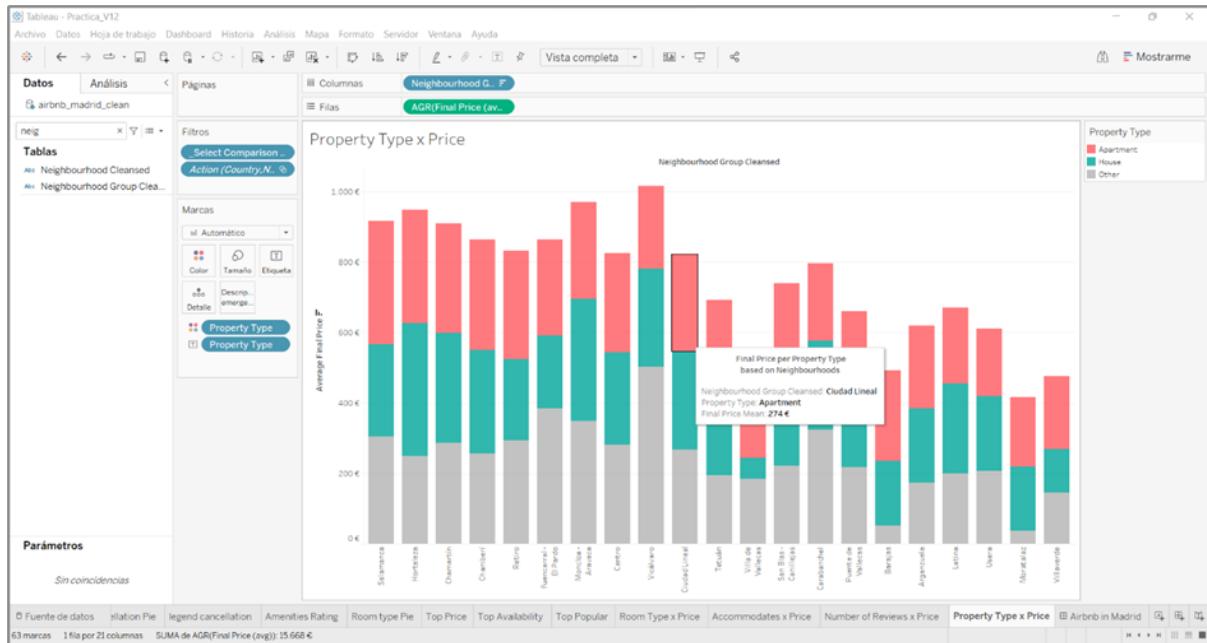
Las variables de precio final y la cantidad de calificaciones o “Number of Reviews” en este gráfico llevan marca de color, adicionalmente precio final se representa con marca de línea sobre el gráfico para evaluar relación y tendencia.

También se activa la descripción emergente con la siguiente información:

- Nombre del barrio
- Promedio de precio final
- Cantidad de calificaciones o “Number of Reviews”

Diagrama comparativo: Property Type x Final Price

Este gráfico de barras permite la comparación de precio según el tipo de propiedad entre los distintos barrios.



Como filtros de columnas utilizamos los barrios y para las filas, la variable calculada “Precio Final” (precio, recargo de limpieza y depósito de seguridad).

La variable de tipo de propiedad lleva marca de color para segmentar las barras según el tipo de propiedad y texto para facilitar la interpretación del usuario.

También se activa la descripción emergente con la siguiente información:

- Nombre del barrio
- Promedio de precio final
- Tipo de propiedad

Conclusiones:

En los dos primeros gráficos, no pudimos identificar una correlación de la disponibilidad “Accommodates” o la cantidad de calificaciones o “Number of Reviews” con el Precio Final.

Mientras que en el diagrama comparativo de “Property Type x Price” observamos que hay un patrón de precios en relación al tipo de propiedad, siendo el “Apartment” y en segundo lugar, el tipo casa o “House” para cada uno de los barrios en Madrid.

Lista de campos calculados utilizados:

Parámetro Map Type y campo calculado _Select map

Utilizado para posibilitar el cambio de mapas en el mismo sitio: Mapa Availability, Mapa Popularity y Mapa Accessibility

Edit Parameter [Map Type]

Name:

Properties:

Data type: String	Display format: Map Availability
Current value: Map Availability	Value when workbook opens: Current value

Allowable values:

All List Range

Value: Map Availability	Display As: Map Availability
Map Popularity	Map Popularity
Map Accessibility	Map Accessibility
Click to add	

Fixed When workbook opens

Add values from:

Remove Selected

Cancel **OK**

Select map

[[Map Type]]

The calculation is valid. 4 Dependencies

Apply **OK**

Parámetro Top Charts Type y campo calculado _Select Top Chart

Utilizado para posibilitar el cambio de gráficos en el mismo sitio: Top Price, Top Availability y Top Popular

Edit Parameter [Top Charts Types]

Name: Top Charts Types

Properties:

Data type: String	Display format: Top Price
Current value: Top Price	Value when workbook opens: Current value

Allowable values:

All List Range

Value: Top Price	Display As: Top Price
Top Availability	Top Availability
Top Popular	Top Popular
Click to add	

Fixed When workbook opens

Add values from:

Remove Selected

Cancel **OK**

Select Top Chart

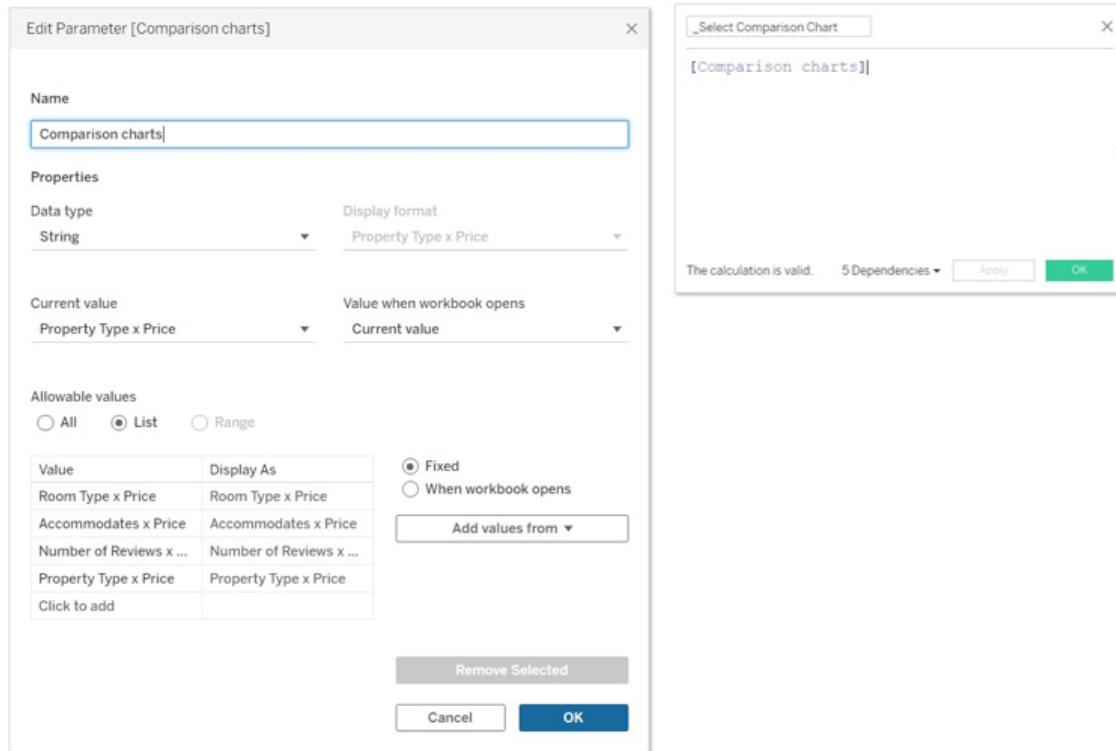
[[Top Charts Types]]

The calculation is valid. 4 Dependencies

Apply **OK**

Parámetro Comparison charts y campo calculado _Select Comparison Chart

Utilizado para posibilitar el cambio de gráficos en el mismo sitio.



The image shows two overlapping dialog boxes. The left dialog is titled 'Edit Parameter [Comparison charts]' and contains fields for 'Name' (set to 'Comparison charts'), 'Properties' (Data type: String, Display format: Property Type x Price), 'Current value' (Property Type x Price), 'Value when workbook opens' (Current value), and 'Allowable values' (List: Room Type x Price, Accommodates x Price, Number of Reviews x ..., Property Type x Price). The right dialog is titled 'Select Comparison Chart' and shows the parameter name '[Comparison charts]'. It includes a message 'The calculation is valid. 5 Dependencies' and buttons for 'Apply' and 'OK'.

Precio Final o “Final Price (avg)”

La variable calculada de precio final corresponde al promedio de la suma de precio, recargo de limpieza y depósito de seguridad.



The dialog box for 'Final Price (avg)' displays the formula: `ZN(AVG([Price])) + ZN(AVG([Cleaning Fee])) + ZN(AVG([Security Deposit]))`. At the bottom, it shows a validation message 'El cálculo es válido.', dependency information '10 dependencias', and buttons for 'Aplicar' and 'Aceptar'.

Accesibilidad o “Accessibility”

La variable calculada accesibilidad corresponde a la suma de las propiedades en donde la columna “Wheelchair accessible” y “Elevator” se hayan marcado con un TRUE.



The screenshot shows a dialog box titled "Accessibility". The formula entered is:

```
SUM(IF [Wheelchair accessible] = True  
AND [Elevator] = True  
THEN 1 ELSE 0 END)
```

Below the formula, it says "El cálculo es válido." (The calculation is valid.) and "2 dependencias" (2 dependencies). There are "Aplicar" (Apply) and "Aceptar" (Accept) buttons at the bottom.

Número total de propiedades o “Total Listing”

Esta variable calculada de número total de propiedades corresponde al conteo de las filas en la columna ID.



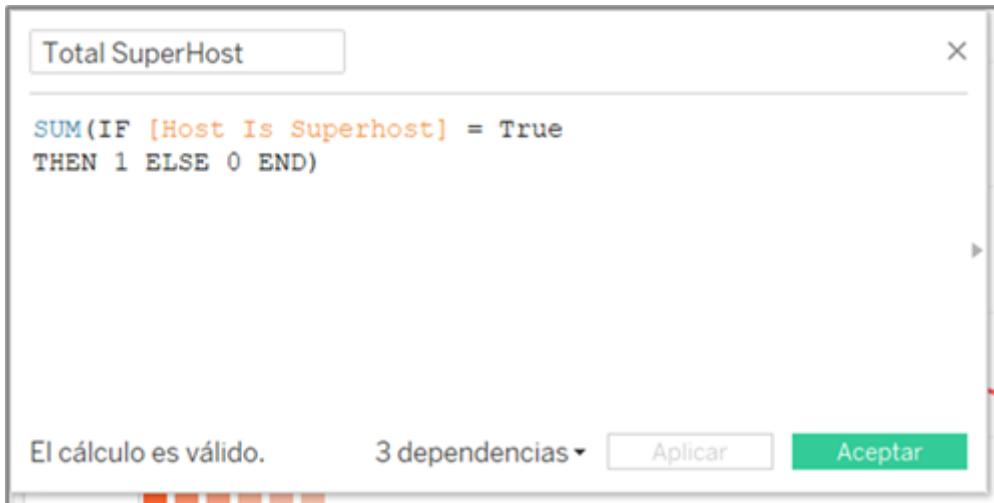
The screenshot shows a dialog box titled "Total listing". The formula entered is:

```
COUNTD([ID])
```

Below the formula, it says "El cálculo es válido." (The calculation is valid.) and "3 dependencias" (3 dependencies). There are "Aplicar" (Apply) and "Aceptar" (Accept) buttons at the bottom.

Cantidad total de Superhost o “Total SuperHost”

La variable calculada de Superhost corresponde a la suma de las propiedades en donde la columna “Host Is Superhost” se haya marcado con un TRUE.



5. Pre-procesamiento y Modelado

La tarea asignada es hacer un algoritmo de regresión lineal que prediga el precio de un inmueble en función de las características que elijáis.

Comenzamos eliminando las columnas de nuestro data frame que no íbamos a usar en nuestro modelo.

```
df_madrid = pd.read_csv("airbnb_madrid_clean.csv")

##first we remove the columns that won't be used in the linear model
df_madrid = remove_columns(["Host ID", "Host Name", "Street", "Neighbourhood Cleansed", "City", "State", "Bed Type",
"Zipcode", "Country", "Latitude", "Longitude", "Amenities Rating", "Host Is Superhost", "Host Identity Verified"], df_madrid)

important_amenities = ['Kitchen', 'Internet', 'Air conditioning', 'Heating', 'Washer',
'Dryer', 'Elevator', 'Wheelchair accessible', 'TV', 'Pool', '24-hour check-in']
df_madrid = remove_columns(important_amenities, df_madrid)
```

El primer problema con el que nos encontramos fue que no sabíamos cómo podíamos usar correctamente nuestras columnas categóricas. Consultando cual podría ser nuestra mejor opción, nos pusimos a leer acerca de *One Hot Encoding* y decidimos usarlo para las mismas. Una vez hecho esto hicimos prematuramente nuestro primer modelo.

```

##then we make dummy variables for the categorical columns
cat_columns = ["Neighbourhood Group Cleansed", "Property Type", "Room Type", "Cancellation Policy"]

for col in cat_columns:
    one_hot = pd.get_dummies(df_madrid[col])
    df_madrid = df_madrid.drop([col, axis=1])
    df_madrid = df_madrid.join(one_hot)

##train, test split
target = df_madrid["Price"]
predictors = df_madrid.drop(columns = ["Price"])
X_train, X_test, y_train, y_test = train_test_split(predictors, target, test_size=0.3, random_state=40)

##this should work after removing Nans
lr = LinearRegression()
lr.fit(X_train, y_train)

##making predictions
y_pred = lr.predict(X_test)

preds_df = pd.DataFrame(dict(observed=y_test, predicted=y_pred))
preds_df.head()

##evaluating the model
print('Score: {}'.format(lr.score(X_test, y_test)))
print('MSE: {}'.format(mean_squared_error(y_test, y_pred)))

```

```

name: Price, dtype: float64
Traceback (most recent call last):
  File "d:\anil\estudios\bootcampglovo\data\entrega_final_data_squad\memoria.py", line 30, in <module>
    lr.fit(X_train, y_train)
  File "C:\Python310\lib\site-packages\sklearn\linear_model\_base.py", line 648, in fit
    X, y = self._validate_data(
  File "C:\Python310\lib\site-packages\sklearn\base.py", line 565, in _validate_data
    X, y = check_X_y(X, y, **check_params)
  File "C:\Python310\lib\site-packages\sklearn\utils\validation.py", line 1186, in check_X_y
    X = check_array(
  File "C:\Python310\lib\site-packages\sklearn\utils\validation.py", line 921, in check_array
    _assert_all_finite(
  File "C:\Python310\lib\site-packages\sklearn\utils\validation.py", line 161, in _assert_all_finite
    raise ValueError(msg_err)
ValueError: Input X contains NaN.
LinearRegression does not accept missing values encoded as NaN natively. For supervised learning, you might want to consider sklearn.ensemble.HistGradientBoostingClassifier and Regressor which accept missing values encoded as NaNs natively. Alternatively, it is possible to preprocess the data, for instance by using an imputer transformer in a pipeline or drop samples with missing values. See https://scikit-learn.org/stable/modules/impute.html You can find a list of all estimators that handle NaN values at the following page: https://scikit-learn.org/stable/modules/impute.html#estimators-that-handle-nan-values

```

Como era de esperarse el archivo no se ejecutó por completo y nos devolvió un error que nos ayudó a saber por dónde seguir.

Como primera medida observamos que los valores nulos en las columnas *Beds*, *Bathroom*, *Bedrooms* y *Price* eran un porcentaje ínfimo respecto al tamaño del data frame y procedimos a eliminarlos. Para las columnas *Security Deposit* y *Cleaning Fee* no pudimos hacer lo mismo ya que los valores nulos eran demasiados. Si bien podríamos haber pensado en los nulos como valor 0, creímos que esto podría no reflejar realmente la realidad. La solución pensada fue entonces crear un diccionario con la mediana de cada columna según el número de habitaciones y usarla en lugar de los valores nulos.

```
##remove row with nan values in beds, bathrooms and bedrooms
df_madrid = df_madrid.dropna(subset=["Beds", "Bathrooms", "Bedrooms", "Price"])

##we need to deal with nan values in security deposit and cleaning fee
##we will use the median considering the number of rooms
dict_sd = {}
dict_cf = {}
number_of_rooms = df_madrid["Bedrooms"].unique()
for number in number_of_rooms:
    df_filtered = df_madrid[df_madrid["Bedrooms"] == number]
    median_sd = df_filtered["Security Deposit"].median()
    dict_sd[number] = median_sd
    median_cf = df_filtered["Cleaning Fee"].median()
    dict_cf[number] = median_cf

for index in range(len(df_madrid)):
    if np.isnan(df_madrid["Security Deposit"].iat[index]):
        number_rooms = df_madrid["Bedrooms"].iat[index]
        median_sd = dict_sd[number_rooms]
        df_madrid["Security Deposit"].iat[index] = median_sd
    if np.isnan(df_madrid["Cleaning Fee"].iat[index]):
        number_rooms = df_madrid["Bedrooms"].iat[index]
        median_cf = dict_cf[number_rooms]
        df_madrid["Cleaning Fee"].iat[index] = median_cf
```

Una vez resuelta esta primera etapa decidimos volver a correr nuestro modelo y observar los resultados, si bien no fueron nada favorables por primera vez nuestro archivo lograba ejecutarse hasta el final.

```
Score: 0.5253206917561672
MSE: 1552.1943194817914
Mean Percentual Error: 0.3611665777896075
```

Continuamos haciendo una limpieza de los outliers con los gráficos creados en el fichero *graphs.py* para ver como estos podían influir en nuestros resultados.



```
##cleaning outliers
df_madrid = df_madrid[df_madrid['Bedrooms'] <= 5]
df_madrid = df_madrid[(df_madrid['Bathrooms'] >= 1) & (df_madrid['Bathrooms'] <= 3)]
df_madrid = df_madrid[df_madrid['Accommodates'] <= 8]

df_madrid = df_madrid[(df_madrid['Bedrooms'] > 0) | ((df_madrid["Bedrooms"] == 0) & (df_madrid["Price"] <= 110 ))]
df_madrid = df_madrid[(df_madrid['Bedrooms'] != 1) | ((df_madrid["Bedrooms"] == 1) & (df_madrid["Price"] <= 125 ))]
df_madrid = df_madrid[(df_madrid['Bedrooms'] != 2) | ((df_madrid["Bedrooms"] == 2) & (df_madrid["Price"] <= 200 ))]
df_madrid = df_madrid[(df_madrid['Bedrooms'] != 3) | ((df_madrid["Bedrooms"] == 3) & (df_madrid["Price"] <= 280 ))]
df_madrid = df_madrid[(df_madrid['Bedrooms'] != 4) | ((df_madrid["Bedrooms"] == 4) & (df_madrid["Price"] <= 390 ))]
df_madrid = df_madrid[(df_madrid['Bedrooms'] != 5) | ((df_madrid["Bedrooms"] == 5) & (df_madrid["Price"] <= 500 ))]

df_madrid = df_madrid[(df_madrid['Bathrooms'] != 1) | ((df_madrid["Bathrooms"] == 1) & (df_madrid["Price"] <= 140 ))]
df_madrid = df_madrid[(df_madrid['Bathrooms'] != 1.5) | ((df_madrid["Bathrooms"] == 1.5) & (df_madrid["Price"] <= 180 ))]
df_madrid = df_madrid[(df_madrid['Bathrooms'] != 2) | ((df_madrid["Bathrooms"] == 2) & (df_madrid["Price"] <= 270 ))]
df_madrid = df_madrid[(df_madrid['Bathrooms'] != 2.5) | ((df_madrid["Bathrooms"] == 2.5) & (df_madrid["Price"] <= 330 ))]
df_madrid = df_madrid[(df_madrid['Bathrooms'] != 3) | ((df_madrid["Bathrooms"] == 3) & (df_madrid["Price"] <= 450 ))]

df_madrid = df_madrid[(df_madrid['Accommodates'] != 1) | ((df_madrid["Accommodates"] == 1) & (df_madrid["Price"] <= 60 ))]
df_madrid = df_madrid[(df_madrid['Accommodates'] != 2) | ((df_madrid["Accommodates"] == 2) & (df_madrid["Price"] <= 100 ))]
df_madrid = df_madrid[(df_madrid['Accommodates'] != 3) | ((df_madrid["Accommodates"] == 3) & (df_madrid["Price"] <= 120 ))]
df_madrid = df_madrid[(df_madrid['Accommodates'] != 4) | ((df_madrid["Accommodates"] == 4) & (df_madrid["Price"] <= 140 ))]
df_madrid = df_madrid[(df_madrid['Accommodates'] != 5) | ((df_madrid["Accommodates"] == 5) & (df_madrid["Price"] <= 180 ))]
df_madrid = df_madrid[(df_madrid['Accommodates'] != 6) | ((df_madrid["Accommodates"] == 6) & (df_madrid["Price"] <= 210 ))]
df_madrid = df_madrid[(df_madrid['Accommodates'] != 7) | ((df_madrid["Accommodates"] == 7) & (df_madrid["Price"] <= 270 ))]
df_madrid = df_madrid[(df_madrid['Accommodates'] != 8) | ((df_madrid["Accommodates"] == 8) & (df_madrid["Price"] <= 300 ))]

df_madrid = df_madrid[(df_madrid['Amenities Rating'] != 'C') | ((df_madrid["Amenities Rating"] == 'C') & (df_madrid["Price"] <= 150 ))]
df_madrid = df_madrid[(df_madrid['Amenities Rating'] != 'B') | ((df_madrid["Amenities Rating"] == 'B') & (df_madrid["Price"] <= 200 ))]
df_madrid = df_madrid[(df_madrid['Amenities Rating'] != 'A') | ((df_madrid["Amenities Rating"] == 'A') & (df_madrid["Price"] <= 300 ))]
```

El error porcentual medio si bien había disminuido notablemente aún teníamos mucho que mejorar.

```
Score: 0.696379479719496
MSE: 340.27383039041104
Mean Percentual Error: 0.27815177013729814
```

Mientras más investigamos y leímos sobre diferentes temas notamos que una recomendación era dividir nuestros datos en train y test desde el comienzo. Hasta el momento, no hacíamos esta división hasta un paso antes de entrenar nuestro modelo así que, lo invertimos.

Mejoramos la lectura de la limpieza de outliers creando listas de tuplas con el número de habitaciones y su correspondiente precio según la columna que queríamos limpiar.

```
##We obtained these values in the lists from the file graphs.py, analyzing the data with a boxplot
bedrooms_price = [(0,110), (1,125), (2,200), (3,280), (4,390), (5,500)]
bathrooms_price = [(1,140), (1.5,180), (2,270), (2.5,330), (3,450)]
accommodates_price = [(1,60), (2,100), (3,120), (4,140), (5,180), (6,210), (7,270), (8,300)]
amenities_rating_price = [('C',150), ('B',200), ('A',300)]

for i in range(len(bedrooms_price)):
    train = train[(train['Bedrooms'] != bedrooms_price[i][0]) | ((train["Bedrooms"] == bedrooms_price[i][0]) & (train["Price"] <= bedrooms_price[i][1]))]

for i in range(len(bathrooms_price)):
    train = train[(train['Bathrooms'] != bathrooms_price[i][0]) | ((train["Bathrooms"] == bathrooms_price[i][0]) & (train["Price"] <= bathrooms_price[i][1]))]

for i in range(len(accommodates_price)):
    train = train[(train['Accommodates'] != accommodates_price[i][0]) | ((train["Accommodates"] == accommodates_price[i][0]) & (train["Price"] <= accommodates_price[i][1]))]

for i in range(len(amenities_rating_price)):
    train = train[(train['Amenities Rating'] != amenities_rating_price[i][0]) | ((train["Amenities Rating"] == amenities_rating_price[i][0]) & (train["Price"] <= amenities_rating_price[i][1]))]
```

En consecuencia, del primer cambio y al notar cuatro bucles *for* seguidos, decidimos crear un diccionario con el nombre de la columna como clave y su correspondiente lista como valor y agrupar todo en un solo bucle.

```
##We obtained these values in the lists from the file graphs.py, analyzing the data with a boxplot
bedrooms_price = [(0,110), (1,125), (2,200), (3,280), (4,390), (5,500)]
bathrooms_price = [(1,140), (1.5,180), (2,270), (2.5,330), (3,450)]
accommodates_price = [(1,60), (2,100), (3,120), (4,140), (5,180), (6,210), (7,270), (8,300)]
amenities_rating_price = [('C',150), ('B',200), ('A',300)]

dict_columns = {"Bedrooms": bedrooms_price, "Bathrooms": bathrooms_price, "Accommodates": accommodates_price, "Amenities Rating": amenities_rating_price}

for column_name in dict_columns.keys():
    list_prices = dict_columns[column_name]
    for i in range(len(list_prices)):
        train = train[(train[column_name] != list_prices[i][0]) | ((train[column_name] == list_prices[i][0]) & (train["Price"] <= list_prices[i][1]))]
```

A su vez, incluimos un la limpieza de outliers la columna de *Guest Included*.

```
##cleaning outliers
train = train[train['Bedrooms'] <= 5]
train = train[(train['Bathrooms'] >= 1) & (train['Bathrooms'] <= 3)]
train = train[train['Accommodates'] <= 8]
train = train[train['Guests Included'] <= 6]

##We obtained these values in the lists from the file graphs.py, analyzing the data with a boxplot
bedrooms_price = [(0,110), (1,125), (2,200), (3,280), (4,390), (5,500)]
bathrooms_price = [(1,140), (1.5,180), (2,270), (2.5,330), (3,450)]
accommodates_price = [(1,60), (2,100), (3,120), (4,140), (5,180), (6,210), (7,270), (8,300)]
amenities_rating_price = [('C',150), ('B',200), ('A',300)]
guests_included_price = [(1, 125), (2, 135), (3, 150), (4,220), (5,230), (6,300)]

dict_columns = {"Bedrooms": bedrooms_price, "Bathrooms": bathrooms_price, "Accommodates": accommodates_price,
"Amenities Rating": amenities_rating_price, "Guests Included": guests_included_price}

for column_name in dict_columns.keys():
    list_prices = dict_columns[column_name]
    for i in range(len(list_prices)):
        train = train[(train[column_name] != list_prices[i][0]) | ((train[column_name] == list_prices[i][0]) & (train["Price"] <= list_prices[i][1]))]
```

Con estos cambios logramos mejorar un poco más nuestro error porcentual medio.

```
MSE: 913.7819356929089
Mean Percentual Error: 0.2690826397559599
```

A partir de este momento entramos en un periodo de llanura en cuanto a mejoras, estábamos un poco agobiadas de probar y ver correlaciones que no afectaban en lo más mínimo nuestros resultados. Decidimos aplicar también un modelo de *Random Forest* para ver el error porcentual que nos devolvía y tener de referencia cuanto más podría seguir mejorando nuestro modelo o, si ya estábamos entrando en un loop de obsesión por ver resultados. Unos de los últimos cambios que creímos favorables fue dejar de usar la columna *Neighbourhood Cleansed* para usar la columna *Zipcode*.

```
df_madrid = remove_columns(["Host ID", "Host Name", "Street", "Neighbourhood Cleansed", "City", "State", "Bed Type",
"Country", "Latitude", "Longitude", "ID", "Number of Reviews", "Host Identity Verified", "Neighbourhood Group Cleansed"], df_madrid)
df_madrid = remove_columns(important_amenities, df_madrid)
df_madrid.drop(df_madrid.columns[0], axis=1, inplace=True)
```

```
MSE: 925.4264764167567
Mean Percentual Error: 0.262460208420863
Mean Percentual Error (RF): 0.22897556415610595
```

Seguimos haciendo algunas mejoras en cuanto a la lectura del código en general, escribimos una función que nos crea un diccionario de medianas para la columna elegida

según la columna de referencia. En nuestro caso, la referencia fue la columna *Bedrooms*. La otra función cambia el valor de un dato si este es nan según el diccionario creado con anterioridad. En paralelo comenzamos a leer sobre la creación de pipelines para machine learning.

```

##we need to deal with nan values in security deposit and cleaning fee
##we will use the median considering the number of rooms
number_of_rooms = train['Bedrooms'].unique()

dict_sd = create_median_dict(train, 'Bedrooms', 'Security Deposit', number_of_rooms)
dict_cf = create_median_dict(train, 'Bedrooms', 'Cleaning Fee', number_of_rooms)

for index in range(len(train)):
    change_nan_to_median(train, 'Bedrooms', 'Security Deposit', index, dict_sd)
    change_nan_to_median(train, 'Bedrooms', 'Cleaning Fee', index, dict_cf)

##cleaning outliers
train = train[train['Bedrooms'] <= 5]
train = train[(train['Bathrooms'] >= 1) & (train['Bathrooms'] <= 3)]
train = train[train['Accommodates'] <= 8]
train = train[train['Guests Included'] <= 6]

##We obtained these values in the lists from the file graphs.py, analyzing the data with a boxplot
bedrooms_price = [(0,110), (1,125), (2,200), (3,280), (4,390), (5,580)]
bathrooms_price = [(1,140), (1.5,180), (2,270), (2.5,330), (3,450)]
accommodates_price = [(1,60), (2,100), (3,120), (4,140), (5,180), (6,210), (7,270), (8,300)]
guests_included_price = [(1, 125), (2, 135), (3, 150), (4,220), (5,230), (6,300)]

dict_columns = {'Bedrooms' : bedrooms_price, 'Bathrooms': bathrooms_price, 'Accommodates': accommodates_price,
'Guests Included': guests_included_price}

for column_name in dict_columns.keys():
    list_prices = dict_columns[column_name]
    for i in range(len(list_prices)):
        train = train[(train[column_name] != list_prices[i][0]) | ((train[column_name] == list_prices[i][0]) & (train['Price'] <= list_prices[i][1]))]

def change_nan_to_median(df, column_reference, column_change, index, median_dict):
    if np.isnan(df[column_change].iat[index]):
        key = df[column_reference].iat[index]
        median = median_dict[key]
        df[column_change].iat[index] = median

def create_median_dict(df, column_reference, column_median, list_keys):
    mydict = {}
    for key in list_keys:
        df_filtered = df[df[column_reference] == key]
        median = df_filtered[column_median].median()
        mydict[key] = median
    return mydict

```

Separamos en columnas numéricas y categóricas, para cada grupo debíamos crear una pipeline con su correspondiente transformer.

Como experimentación, creamos diferentes transformers que luego no fueron aplicados en nuestro modelo lineal, pero nos sirvieron para entender el funcionamiento de los mismos y sus clases.

```
class DropAmenities(TransformerMixin):
    def __init__(self):
        super().__init__()

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

    def transform(self, X, y=None):
        # drop incomplete rows
        Xp = pd.DataFrame(X)
        Xdrop = Xp.drop('Amenities Rating', axis=1, inplace=True).to_numpy()
        return Xdrop
```

```
class HostNumerical(TransformerMixin):
    def __init__(self):
        super().__init__()

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

    def transform(self, X, y=None):
        Xp = pd.DataFrame(X)
        for index in range(len(Xp)):
            if Xp['Host Is Superhost'].iat[index]:
                Xp['Host Is Superhost'].iat[index] = 1
            else:
                Xp['Host Is Superhost'].iat[index] = 0
        Xmod = Xp.to_numpy()
        return Xmod
```

```
class ImputeMedian(TransformerMixin):
    def __init__(self):
        super().__init__()

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

    def transform(self, X, y=None):
        for index in range(len(X)):
            change_nan_to_median(X, 'Bedrooms', 'Security Deposit', index, dict_sd)
            change_nan_to_median(X, 'Bedrooms', 'Cleaning Fee', index, dict_cf)
```

```
class RemoveColumns(TransformerMixin):
    def __init__(self):
        super().__init__()

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

    def transform(self, X, y=None):
        #remove the selected columns
        columns_list = ['Host ID', 'Host Name', 'Street', 'Neighbourhood Cleansed', 'City', 'State', 'Bed Type', 'Amenities Rating',
        'Country', 'Latitude', 'Longitude', 'ID', 'Number of Reviews', 'Host Identity Verified', 'Neighbourhood Group Cleansed']
        Xp = pd.DataFrame(X)
        Xdrop = remove_columns(columns_list, Xp).to_numpy()
        return Xdrop
```

Luego de estas pruebas dimos finalmente con los dos transformers que aplicamos en las pipelines. A su vez, dejamos de usar la función `get_dummies` para crear el `One Hot Encoding` y decidimos usar la clase `OneHotEncoder` que nos proporciona Sklearn.

```

numeric_columns = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
print('numeric columns')
print(numeric_columns)
print(type(numeric_columns))
cat_columns = X_train.select_dtypes(include=['object', 'category']).columns.to_list()
print('cat columns')
print(cat_columns)
median_columns = ['Bedrooms', 'Cleaning Fee', 'Security Deposit']

numeric_transformer = Pipeline(
    steps=[
        ('imputer', ImputeMedian()),
        ('scaler', StandardScaler())
    ]
)

categorical_transformer = Pipeline(
    steps= [
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ]
)

preprocessor = ColumnTransformer(
    transformers= [
        ('numeric', numeric_transformer, numeric_columns),
        ('cat', categorical_transformer, cat_columns)
    ],
    remainder='passthrough'
)

pipe = Pipeline([('preprocessing', preprocessor),
                 ('modelo', LinearRegression())])
print('llegue hasta aca')
pipe.fit(X=X_train, y=y_train)

print('hice fit')

```

```

class DropNans(TransformerMixin):
    def __init__(self):
        super().__init__()

    def fit(self, X, y):
        self.X = X
        self.y = y
        return self

    def transform(self, X, y):
        # drop incomplete rows
        columns = ['Beds', 'Bathrooms', 'Bedrooms', 'Price', 'Review Scores Rating', 'Host Response Rate']
        Xp = pd.DataFrame(X)
        Yp = pd.DataFrame(y)
        data = Xp.join(Yp)
        datadrop = data.dropna(subset=columns).to_numpy()
        Ydrop = datadrop['Price']
        Xdrop = datadrop.drop('Price', axis = 1)
        return Xdrop, Ydrop

```

Una transformación importante de nuestros datos era el imputar las medianas en las columnas *Cleaning Fee* y *Security Deposit* que anteriormente teníamos dentro de la función *change_nan_to_median*, ahora la convertimos en un transformer.

```

class ImputeMedian(BaseEstimator, TransformerMixin):
    def __init__(self):
        super().__init__()

    def fit(self, X, y=None):
        self.X = X
        number_of_rooms = X['Bedrooms'].unique()
        self.dict_sd = create_median_dict(X, 'Bedrooms', 'Security Deposit', number_of_rooms)
        self.dict_cf = create_median_dict(X, 'Bedrooms', 'Cleaning Fee', number_of_rooms)
        return self

    def transform(self, X, y=None):
        for index in range(len(X)):
            change_nan_to_median(X, 'Bedrooms', 'Security Deposit', index, self.dict_sd)
            change_nan_to_median(X, 'Bedrooms', 'Cleaning Fee', index, self.dict_cf)

```

Cuando ejecutamos nuestro archivo nos encontramos con el primer error, habíamos malinterpretado cómo funcionan los datos dentro de una pipeline. Nosotras pensábamos que estos se mantenían con su forma original de Dataframe y que podíamos seguir accediendo por el nombre de la columna, pero nos equivocamos.

Luego de estar un largo rato entendiendo el error, ya que era una situación y herramientas completamente nuevas para nosotras, entendimos que los datos se transformaban en matrices y perdían su propiedad de nombre de columna el cual usábamos en nuestro transformer de *ImputeMedian* al invocar a la función de *create_median_dict*. como mostramos anteriormente para crear el diccionario esta función usa los nombres de las columnas tanto de referencia como para la que se quiere crear el diccionario de medianas.

```

File "C:\Python310\lib\site-packages\joblib\parallel.py", line 200, in <listcomp>
    return [func(*args, **kwargs)
File "C:\Python310\lib\site-packages\sklearn\utils\parallel.py", line 123, in __call__
    return self.function(*args, **kwargs)
File "C:\Python310\lib\site-packages\sklearn\pipeline.py", line 893, in _fit_transform_one
    res = transformer.fit_transform(X, y, **fit_params)
File "C:\Python310\lib\site-packages\sklearn\pipeline.py", line 437, in fit_transform
    Xt = self._fit(X, y, **fit_params_steps)
File "C:\Python310\lib\site-packages\sklearn\pipeline.py", line 359, in _fit
    X, fitted_transformer = fit_transform_one_cached(
File "C:\Python310\lib\site-packages\joblib\memory.py", line 349, in __call__
    return self.func(*args, **kwargs)
File "C:\Python310\lib\site-packages\sklearn\pipeline.py", line 893, in _fit_transform_one
    res = transformer.fit_transform(X, y, **fit_params)
File "C:\Python310\lib\site-packages\sklearn\utils\_set_output.py", line 142, in wrapped
    data_to_wrap = f(self, X, *args, **kwargs)
File "C:\Python310\lib\site-packages\sklearn\base.py", line 862, in fit_transform
    return self.fit(X, y, **fit_params).transform(X)
File "d:\ani\estudios\bootcampGlovo\data\entrega_final_data_squad\transformers.py", line 34, in fit
    if not isnan(sd):
TypeError: must be real number, not str

```

```

class ImputeMedian(BaseEstimator, TransformerMixin):
    def __init__(self):
        super().__init__()

    def fit(self, X, y=None):
        self.X = X
        dict_list_sd = {}
        dict_list_cf = {}
        for row in X:
            bedrooms = row[3]
            sd = row[5]
            cf = row[6]
            if bedrooms not in dict_list_sd:
                dict_list_sd[bedrooms] = []
                dict_list_cf[bedrooms] = []
            if not isnan(sd):
                dict_list_sd[bedrooms].append(sd)
            if not isnan(cf):
                dict_list_cf[bedrooms].append(cf)
        self.dict_sd = {}
        self.dict_cf = {}
        for bedroom in dict_list_sd.keys():
            self.dict_sd[bedroom] = median(dict_list_sd[bedroom])
            self.dict_cf[bedroom] = median(dict_list_cf[bedroom])
        print(self.dict_cf)
        return self

    def transform(self, X, y=None):
        for index in range(len(X)):
            change_nan_to_median(X, 'Bedrooms', 'Security Deposit', index, self.dict_sd)
            change_nan_to_median(X, 'Bedrooms', 'Cleaning Fee', index, self.dict_cf)

```

Para este momento decidimos aplicar en nuestra pipeline el transformer *DataFrameSelector* que obtuvimos del libro *Hands-On Machine Learning with Scikit-Learn & TensorFlow* de Aurélien Géron, el cual tomamos de referencia en el uso de pipelines.

```

numeric_transformer = Pipeline(
    steps=[
        ('selector', DataFrameSelector(numeric_columns)),
        ('imputer', ImputeMedian()),
        ('scaler', StandardScaler())
    ]
)

categorical_transformer = Pipeline(
    steps=[
        ('selector', DataFrameSelector(cat_columns)),
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ]
)

```

```

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values

```

Para esta instancia ya habíamos decidido cambiar la escala de nuestras columnas numéricas con el objetivo de normalizarlas y hacerlas más resistentes a outliers.

Usamos el transformer *LabelEncoder* para cambiar los datos categóricos a números antes de pasar por *OneHotEncoder* y usamos *handle_unknown='ignore'* para evitar errores si hay una categoría en test que no está presente en train.

```
##try robust scaler later
numeric_transformer = Pipeline(
    steps=[
        ('selector', DataFrameSelector(numeric_columns)),
        ('imputer', ImputeMedian()),
        ('scaler', StandardScaler())
    ]
)

categorical_transformer = Pipeline(
    steps= [
        ('selector', DataFrameSelector(cat_columns)),
        ('encode', LabelEncoder()),
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ]
)

preprocessor = FeatureUnion(
    transformer_list = [
        ('numeric', numeric_transformer),
        ('cat', categorical_transformer)
    ]
)
```

Finalmente retrocedimos en los cambios y nuestro archivo pasaba la etapa de fit nuestro transformer. Si bien, aún daba error en el proceso lo tomamos como un avance.

```
##try robust scaler later
numeric_transformer = Pipeline(
    steps=[
        ('selector', DataFrameSelector(numeric_columns)),
        ('imputer', ImputeMedian()),
        ('scaler', StandardScaler())
    ]
)

categorical_transformer = Pipeline(
    steps= [
        ('selector', DataFrameSelector(cat_columns)),
        ('onehot', OneHotEncoder(sparse=False))
    ]
)

preprocessor = FeatureUnion(
    transformer_list = [
        ('numeric', numeric_transformer),
        ('cat', categorical_transformer)
    ]
)
```

```

Traceback (most recent call last):
  File "d:\ani\estudios\bootcampGlovo\data\entrega_final_data_squad\memoria.py", line 239, in <module>
    lr.fit(X_train, y_train)
  File "C:\Python310\lib\site-packages\sklearn\linear_model\base.py", line 648, in fit
    X, y = self._validate_data(
  File "C:\Python310\lib\site-packages\sklearn\base.py", line 565, in _validate_data
    X, y = check_X_y(X, y, **check_params)
  File "C:\Python310\lib\site-packages\sklearn\utils\validation.py", line 1106, in check_X_y
    X = check_array(
  File "C:\Python310\lib\site-packages\sklearn\utils\validation.py", line 879, in check_array
    array = _asarray_with_order(array, order=order, dtype=dtype, xp=xp)
  File "C:\Python310\lib\site-packages\sklearn\utils\_array_api.py", line 185, in _asarray_with_order
    array = numpy.asarray(array, order=order, dtype=dtype)
  File "C:\Python310\lib\site-packages\pandas\core\generic.py", line 2070, in __array__
    return np.asarray(self._values, dtype=dtype)
ValueError: could not convert string to float: 'Madrid 28004'

```

Para resolver el error notamos que nuestro data frame de train podría no contener datos que luego si aparecían en test, así que utilizamos el parámetro `handle_unknown`. Usamos el parámetro `sparse` para evitar usar `LabelEncoder`.

```

categorical_transformer = Pipeline(
    steps= [
        ('selector', DataFrameSelector(cat_columns)),
        ('onehot', OneHotEncoder(sparse=False, handle_unknown='ignore'))
    ]
)

```

```

lr = LinearRegression()
lr.fit(X_train_prepared, y_train)

# ##making predictions
y_pred = lr.predict(X_test_prepared)

_preds_df = pd.DataFrame(dict(observed=y_test, predicted=y_pred))
_preds_df.head()

# ##evaluating the model
print('MSE: {}'.format(mean_squared_error(y_test, y_pred)))
print('Mean Percentual Error: {}'.format(mean_absolute_percentage_error(y_test, y_pred)))

# ##random forest for comparison
rf = RandomForestRegressor()
rf.fit(X_train_prepared, y_train)

y_pred_rd = rf.predict(X_test_prepared)

print('Mean Percentual Error (RF): {}'.format(mean_absolute_percentage_error(y_test, y_pred_rd)))

```

```

MSE: 9.509640857325075e+19
Mean Percentual Error: 2957364.882992695
Mean Percentual Error (RF): 0.23407084381819718

```

El error medio porcentual se había disparado en comparación a nuestro primer modelo, pero por primera vez desde que comenzamos con las pipelines funcionó.

Para mejorar aplicamos el modelo de regresión lineal `Ridge` y en la pipeline para la estandarización de los valores numéricos cambiamos por `RobustScaler`.



```
lr = Ridge()
lr.fit(X_train_prepared, y_train)
```

```
numeric_transformer = Pipeline(
    steps=[
        ('selector', DataFrameSelector(numeric_columns)),
        ('imputer', ImputeMedian()),
        ('scaler', RobustScaler())
    ]
)
```

Estas pequeñas modificaciones dieron resultados notablemente más favorables y cercanos a nuestros objetivos.

```
MSE: 793.6136463378847
Mean Percentual Error: 0.30016433205196513
Mean Percentual Error (RF): 0.2332766931331251
```

Decidimos cambiar *Ridge* por *RidgeCV* para poder modificar el *alpha* y mejorar nuestro modelo. También cambiamos de *RobustScaler* a *StandardScaler* porque pasamos a sacar los outliers del train antes de empezar la *Pipeline*.

Sacar los outliers de train al principio, nos hizo imposible usar el transformer *ImputeMedian* que habíamos hecho nosotros, porque generaría un error para cualquier valor en el test que no estaba presente en train. Así que decidimos usar la mediana de todo el train en lugar de los valores nulos, con el transformer *SimpleImputer* que nos proporciona *Sklearn*.

```
lr = RidgeCV(alphas=(0.1, 1.0, 6.6, 10.0))
lr.fit(X_train_prepared, y_train)
```

```
numeric_transformer = Pipeline(
    steps=[
        ('selector', DataFrameSelector(numeric_columns)),
        ('imputer', SimpleImputer(strategy="median"))
    ]
)
```

```
Name: Price, dtype: float64
MSE: 916.0905911196381
Mean Percentual Error: 0.2570981174026297
Mean Percentual Error (RF): 0.23337404024929456
```

Si bien a esta altura ya creímos que no podíamos mejorarlo más, logramos una última disminución en nuestro error porcentual medio al dejar de usar la columna *Zipcode* y cambiarla por el uso de *Neighbourhood Cleansed*.

También paramos de normalizar los valores de las columnas numéricas porque nuestro modelo *Ridge* no lo necesita.

Para pasar en limpio dejamos a continuación enumerados el paso a paso de nuestro modelo final:

1 - Removimos de nuestros data frame todas las columnas que no tenían valor para nuestro modelo.

```
df_madrid = pd.read_csv('airbnb_madrid_clean.csv')

##first we remove the columns that won't be used in the linear model
df_madrid = remove_columns(['Host ID', 'Host Name', 'Street', 'Zipcode', 'City', 'State', 'Bed Type', 'Amenities Rating', 'Amenities Score', 'Country', 'Latitude', 'Longitude', 'ID', 'Number of Reviews', 'Host Identity Verified', 'Neighbourhood Group Cleansed'], df_madrid)

df_madrid.drop(df_madrid.columns[0], axis=1, inplace= True)
```

2 - Dividimos desde el comienzo nuestro data frame en train y test. Eliminamos los valores nulos en cada uno.

```
##now we divide in train and test
train, test = train_test_split(df_madrid, test_size=0.2, random_state=40)

train = train.dropna(subset=['Beds', 'Bathrooms', 'Bedrooms', 'Price', 'Review Scores Rating', 'Host Response Rate'])
test= test.dropna(subset=['Beds', 'Bathrooms', 'Bedrooms', 'Price', 'Review Scores Rating', 'Host Response Rate'])
```

3 - Limpiamos los outliers de nuestra muestra train en base a todo el previo análisis que habíamos realizado con los gráficos.

```
##cleaning outliers
train = train[train['Bedrooms'] <= 5]
train = train[(train['Bathrooms'] >= 1) & (train['Bathrooms'] <= 3)]
train = train[train['Accommodates'] <= 8]
train = train[train['Guests Included'] <= 6]

##We obtained these values in the lists from the file graphs.py, analyzing the data with a boxplot
bedrooms_price = [(0,110), (1,125), (2,200), (3,280), (4,390), (5,500)]
bathrooms_price = [(1,140), (1.5,180), (2,270), (2.5,330), (3,450)]
accommodates_price = [(1,60), (2,100), (3,120), (4,140), (5,180), (6,210), (7,270), (8,300)]
guests_included_price = [(1, 125), (2, 135), (3, 150), (4,220), (5,230), (6,300)]

dict_columns = {'Bedrooms' : bedrooms_price, 'Bathrooms': bathrooms_price, 'Accommodates': accommodates_price,
'Guests Included': guests_included_price}

for column_name in dict_columns.keys():
    list_prices = dict_columns[column_name]
    for i in range(len(list_prices)):
        train = train[(train[column_name] != list_prices[i][0]) | ((train[column_name] == list_prices[i][0]) & (train['Price'] <= list_prices[i][1]))]
```

4 - Partimos nuestras muestras en “X” e “Y”, donde “X” son las variables que van a utilizarse para predecir nuestro “Y”. Observamos que nuestros “Y” fueren razonablemente semejantes en los valores de sus cuartiles.

```
##splitting in Y (value that we want to predict) and X (variables that will be used in order to predict Y)
y_train = train['Price']
X_train = train.drop('Price', axis=1)
y_test = test['Price']
X_test = test.drop('Price', axis = 1)

print(f'"Train describe"\n-----\n{y_train.describe()}\n')

print(f'"Test describe"\n-----\n{y_test.describe()}'")
```

```
"Train describe"
-----
count    7127.000000
mean      57.344605
std       31.242116
min       9.000000
25%      33.000000
50%      51.000000
75%      75.000000
max     260.000000
Name: Price, dtype: float64
"Test describe"
-----
count    1933.000000
mean      64.896017
std       48.162166
min      10.000000
25%      35.000000
50%      55.000000
75%      80.000000
max     550.000000
Name: Price, dtype: float64
```

5 - Creamos las correspondientes pipelines de transformación para ambos tipos de columnas y las aplicamos con éxito.

```
##now we divide the column in numerical and categorical
numerical_columns = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
cat_columns = X_train.select_dtypes(include=['object', 'category']).columns.to_list()

##first we create a pipeline for the numerical columns
numerical_transformer = Pipeline(
    steps=[
        ('selector', DataFrameSelector(numerical_columns)),
        ('imputer', SimpleImputer(strategy="median")),
        ('scaler', StandardScaler())
    ]
)

##then for the categorical
categorical_transformer = Pipeline(
    steps= [
        ('selector', DataFrameSelector(cat_columns)),
        ('onehot', OneHotEncoder(sparse=False, handle_unknown='ignore'))
    ]
)

##then we combine the two using FeatureUnion
preprocessor = FeatureUnion(
    transformer_list = [
        ('numeric', numerical_transformer),
        ('cat', categorical_transformer)
    ]
)

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

6 - Entrenamos nuestro modelo y realizamos las predicciones. Observamos algunas filas del eje “Y” de nuestro test y las predicciones que se realizaron en una tabla conjunta. Por último evaluamos nuestro modelo según su MSE y MAPE.

```
lr = RidgeCV(alphas=(0.1, 1.0, 6.6, 10.0))
lr.fit(X_train_prepared, y_train)

##making predictions
y_pred = lr.predict(X_test_prepared)

_preds_df = pd.DataFrame(dict(observed=y_test, predicted=y_pred))
print(_preds_df.head(20))

##evaluating the model
print('MSE: {}'.format(mean_squared_error(y_test, y_pred)))
print('Mean Percentual Error: {} .format(mean_absolute_percentage_error(y_test, y_pred)))')

##random forest for comparison
rf = RandomForestRegressor()
rf.fit(X_train_prepared, y_train)

y_pred_rf = rf.predict(X_test_prepared)

print('Mean Percentual Error (RF): {} .format(mean_absolute_percentage_error(y_test, y_pred_rf)))'
```

	observed	predicted
8801	65.0	63.419245
324	26.0	43.255928
11942	20.0	35.098969
995	38.0	50.188664
3913	22.0	29.459284
3143	40.0	34.994601
3862	536.0	143.671476
3659	60.0	73.119216
8895	120.0	107.625336
3157	95.0	72.661219
9101	99.0	89.329369
6991	140.0	93.274803
5801	60.0	72.666989
5287	35.0	38.542556
9029	72.0	80.390527
2897	23.0	38.326948
2159	25.0	33.222313
546	22.0	33.151779
7291	130.0	96.521606
3447	75.0	74.321377

MSE: 915.954754915413
Mean Percentual Error: 0.25716373559324307
Mean Percentual Error (RF): 0.23043541744362975

CONCLUSIONES

Al iniciar este proyecto final del Bootcamp teníamos un objetivo claro, realizar un modelo que predijera el precio de un inmueble en función de características que debíamos seleccionar. Considerando que se nos daba una base de datos, que en parte ya la habíamos trabajado en clase, asumimos que íbamos a obtener una respuesta clara al objetivo planteado.

Como hemos ido explicando a lo largo de la memoria nos fuimos encontrando con desafíos que paso a paso hemos ido solucionando con una búsqueda activa y creativa de alternativas sin perder el foco al problema a resolver.

La principal conclusión de nuestro trabajo es que nuestro modelo de predicción realizado, debido a limitaciones en la base de datos y tal vez las peculiaridades del mercado de alquiler vacacional, no nos aportó estimaciones de precio que nos dejarán satisfechas en su exactitud.

Como limitaciones al modelo, sentimos que para llegar a un mejor análisis de las correlaciones y estimación final de precio, nos hubiera sido necesario contar por ejemplo con una serie de tiempo más larga o con los metros cuadrados totales de la propiedad.

Sin embargo, luego de afinar y mejorar el modelo de predicción y con el foco puesto en entregar un producto digital que sirviera a un potencial usuario de Airbnb, elaboramos una interfaz en django donde, en base a las características del piso a alquilar y nuestro modelo predictivo final, resulta en un rango de precio estimado por noche con un error porcentual medio del 25%.

Los pasos siguientes en nuestro proyecto es avanzar en la mejora y expansión de las bases de datos que usaremos de insumo para poder bajar nuestro error porcentual medio y tener un producto digital que sea de utilidad tanto para Airbnb así como para futuros propietarios que quieran alquilar sus viviendas y así poder proyectar un flujo de ingresos ciertos sobre su inversión.

A modo de cierre compartimos a continuación el enlace al [github](#) donde hemos hecho commit de todos los entregables generados así como el archivo django para poder visualizar el simulador web realizado.