Project Introduction

Why did the rabbit deny his partner's marrigage proposal. Because the ring wasn't 24 carets! 😂

Recently, my friends and I traveled to Austin, Texas to celebrate the upcoming marriage of our close friend. In all, 8 of us were making a trip. Immediately, we needed to find housing where each one of us could easily access the other. Hotel rooms were an option; however, for 8 of us it could have easily become expensive. Thankfully, we were able to book the perfect house to enjoy our stay with Air bnb.

Airbnb ("Air Bed and Breakfast) is an online (mostly) service that lets property owners rent housing to travelers. Whether celebrating a wedding, or traveling for a vacation, airbnb has helped many customers find affordable, quality housing. Listings can range from shared rooms to an entire house and can last from 1 day to the maximum days the owner decides. Ultimately, hosts determine the price of the rental; however, Airbnb is a free market so the tend to price fairly.

This project will analyze the characteristics that make up the listing prices. Also, we will use machine learning to develop a model to predict rental prices.

Data Description

Unforunately, Airbnb doe not release its own data regarding listings within the brand. However, we will grab data collected by Inside Airbnb . Inside Airbnb compiles public data regarding popular cities across the globe. The data we are using was scraped on March 20, 2022.

Each observation contains one rental listing in Austin, Texas.

Some features that are belived to be related tot he pricing are:

- `property_type` : Whether the rental is a Home, apartment, hotel room or private rooms.
  - Different properties have different sizes which, in return, will cost more. For example, one would epect a house to be priced higher than an apartment
- `host_location` : For Airbnbs, downtown rentals may be more expensive than housing on the outside of the city

    - A hosts proximity may help them understand the market better. Thus, they would price more in accordance to the local market. On ther other hand, hosts that live in expensive areas may charge more for their properties.
- `accommodates` : the number of people the housing can accomodate
    - A larger group will require a larger rental. Therefore, they will pay more
- `bathrooms` : the number of bathrooms in a house
    - More bathrooms are often correlated with a larger property. Thus, it is more expensive.
- `bedrooms` : the number of bedrooms in the rental
    - For the same reason as the number of bathrooms, more bedrooms would cause the rental to be more expensive.
- `beds` : the number of beds in the rental
    - Same reason as bedrooms
- `amenities` : Extra ammenities with the housing (Pool, internet, parking, etc.)
    - I expect more amenities to correlate to a higher price. Amenities are 'extra' things include with the rental. Thus, the more amenities the property has, the more expensive.
- `host_neighbourhood / zipcode` : The zipcode of where the listing is located.
    - I expect some areas to have more expensive listings than others. Many locations have subareas that have a higher cost of licing than others
    - The `host_neighbourhood` predictor reflects the zipcode. Therefore, it will be renamed `zipcode` .

Downtown Austin: 30.2729 N, 97.7444 W

In [1]:
```
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import datetime as dt
import missingno as msno
from sklearn.model_selection import train_test_split as tst
import geopandas as gpd
import branca.colormap as cm
from matplotlib.colors import ListedColormap, LinearSegmentedColormap
import matplotlib.colors
import folium
from folium.plugins import FloatImage
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
```

```
/opt/anaconda3/lib/python3.9/site-packages/geopandas/_compat.py:111: UserWarning: The Shapely GEOS version
(3.10.2-CAPI-1.16.0) is incompatible with the GEOS version PyGEOS was compiled with (3.10.1-CAPI-1.16.0). Con
versions between both will be slow.
  warnings.warn(
```

In [2]:

```
pd.set_option('display.max_columns', 75)
pd.set_option('display.max_rows', 75)
```

Data Cleaning

In [3]:

```
main_df = pd.read_csv("data/main.csv")
original_df = main_df.copy()
main_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11972 entries, 0 to 11971
Data columns (total 74 columns):
 #   Column                                        Non-Null Count  Dtype  
---  ------                                        --------------  -----  
 0   id                                            11972 non-null  int64  
 1   listing_url                                   11972 non-null  object 
 2   scrape_id                                     11972 non-null  int64  
 3   last_scraped                                  11972 non-null  object 
 4   name                                          11972 non-null  object 
 5   description                                   11808 non-null  object 
 6   neighborhood_overview                         7059 non-null   object 
 7   picture_url                                   11971 non-null  object 
 8   host_id                                       11972 non-null  int64  
 9   host_url                                      11972 non-null  object 
 10  host_name                                     11969 non-null  object 
 11  host_since                                    11969 non-null  object 
 12  host_location                                 11954 non-null  object 
 13  host_about                                    7293 non-null   object 
 14  host_response_time                            8523 non-null   object 
 15  host_response_rate                            8523 non-null   object 
 16  host_acceptance_rate                          9110 non-null   object 
 17  host_is_superhost                             11969 non-null  object 
 18  host_thumbnail_url                            11969 non-null  object 
 19  host_picture_url                              11969 non-null  object 
 20  host_neighbourhood                            10254 non-null  object 
 21  host_listings_count                           11969 non-null  float64
 22  host_total_listings_count                     11969 non-null  float64
 23  host_verifications                            11972 non-null  object 
 24  host_has_profile_pic                          11969 non-null  object 
 25  host_identity_verified                        11969 non-null  object 
 26  neighbourhood                                 7059 non-null   object 
 27  neighbourhood_cleansed                        11972 non-null  int64  
 28  neighbourhood_group_cleansed                  0 non-null      float64
 29  latitude                                      11972 non-null  float64
 30  longitude                                     11972 non-null  float64
 31  property_type                                 11972 non-null  object 
 32  room_type                                     11972 non-null  object 
 33  accommodates                                  11972 non-null  int64  
 34  bathrooms                                     0 non-null      float64
 35  bathrooms_text                                11956 non-null  object 
 36  bedrooms                                      11261 non-null  float64
 37  beds                                          11822 non-null  float64
 38  amenities                                     11972 non-null  object 
 39  price                                         11972 non-null  object 
```

```
 40  minimum_nights                               11972 non-null  int64
 41  maximum_nights                               11972 non-null  int64
 42  minimum_minimum_nights                       11971 non-null  float64
 43  maximum_minimum_nights                       11971 non-null  float64
 44  minimum_maximum_nights                       11971 non-null  float64
 45  maximum_maximum_nights                       11971 non-null  float64
 46  minimum_nights_avg_ntm                       11971 non-null  float64
 47  maximum_nights_avg_ntm                       11971 non-null  float64
 48  calendar_updated                             0 non-null      float64
 49  has_availability                             11972 non-null  object
 50  availability_30                              11972 non-null  int64
 51  availability_60                              11972 non-null  int64
 52  availability_90                              11972 non-null  int64
 53  availability_365                             11972 non-null  int64
 54  calendar_last_scraped                        11972 non-null  object
 55  number_of_reviews                            11972 non-null  int64
 56  number_of_reviews_ltm                        11972 non-null  int64
 57  number_of_reviews_l30d                       11972 non-null  int64
 58  first_review                                 9026 non-null   object
 59  last_review                                  9026 non-null   object
 60  review_scores_rating                         9026 non-null   float64
 61  review_scores_accuracy                       8954 non-null   float64
 62  review_scores_cleanliness                    8954 non-null   float64
 63  review_scores_checkin                        8953 non-null   float64
 64  review_scores_communication                  8954 non-null   float64
 65  review_scores_location                       8952 non-null   float64
 66  review_scores_value                          8952 non-null   float64
 67  license                                      0 non-null      float64
 68  instant_bookable                             11972 non-null  object
 69  calculated_host_listings_count               11972 non-null  int64
 70  calculated_host_listings_count_entire_homes  11972 non-null  int64
 71  calculated_host_listings_count_private_rooms 11972 non-null  int64
 72  calculated_host_listings_count_shared_rooms  11972 non-null  int64
 73  reviews_per_month                            9026 non-null   float64
dtypes: float64(24), int64(18), object(32)
memory usage: 6.8+ MB
```

In [4]:

```
main_df.head(5)
```

Out[4]:

	id	listing_url	scrape_id	last_scraped	name	description	neighborhood_overview	
0	5456	https://www.airbnb.com/rooms/5456	20220312074014	2022-03-13	Walk to 6th, Rainey St and Convention Ctr	Great central location for walking to Convent...	My neighborhood is ideally located if you want...	https://a
1	5769	https://www.airbnb.com/rooms/5769	20220312074014	2022-03-31	NW Austin Room	The space Looking for a comfortabl...	Quiet neighborhood with lots of trees and good...	https://a
2	6413	https://www.airbnb.com/rooms/6413	20220312074014	2022-03-31	Gem of a Studio near Downtown	Great studio apartment, perfect a single perso...	Travis Heights is one of the oldest neighborho...	https:/
3	6448	https://www.airbnb.com/rooms/6448	20220312074014	2022-03-12	Secluded Studio @ Zilker - King Bed, Bright & ...	Clean, private space with everything you need ...	The neighborhood is fun and funky (but quiet)!...	https:/
4	8502	https://www.airbnb.com/rooms/8502	20220312074014	2022-03-13	Woodland Studio Lodging	The space Fully furnished suite wi...	NaN	https:/

Initial Look

The dataset contains 11972 Airbnb listings and 73 predictors. 32 are strings while the rest are numeric features. Immediately, we can see a few columns contain text excerpts from the listings. Also, some columns have no non-null values. Therefore, we will drop the following columns:

- For providing information we can find in other predictors
  - `name`
  - `description`
  - `neighborhood_overview`

- Unnecessary Info
    - `picture_url`
    - `host_url`
    - `id`
    - `scrape_id`
    - `last_scraped`
    - `host_id`
    - `host_thumbnail_url`
    - `host_picture_url`
    - `calendar_last_scraped`
    - `host_about`
    - `license`
    - `listing_url`
    - `host_verifications`
- Do not want to include private information
    - `host_name`
- Contains no nonnull values
    - `neighbourhood_group_cleansed`
    - `bathrooms`
    - `calendar_updated`
    - `license`

Some other things to note:

```
- neighbourhood_cleansed lists the zipcode
- neighbourhood states if in Austin or not
```

In [5]:

```
#drop columns that have no potential of giving info
main_df.drop(['picture_url', 'host_url', 'id','scrape_id', 'last_scraped', 'host_id',
              'host_thumbnail_url', 'host_picture_url', 'calendar_last_scraped', 'host_about', 'license','list
              'name', 'description','neighborhood_overview', 'host_verifications', 'host_name',
              'neighbourhood_group_cleansed', 'bathrooms','calendar_updated', 'license'], axis=1, inplace=True
```

Cleaning predictors, Handling Missingness and Univariate Analysis

Several predictors need further handling to extract more information. Also, depending on their importance, we may need to add a level to categorical features to represent the missing values in the distribution.

About 14 predictors have more than 20% of their values missing. As we clean the individual features, we will determine a features importance. If needed for further analysis, we will then determine how to handle the missingness of the data. For categorical features, we will create a seperate level representing a missing value. For continuous data, we can fill the missing values with the median if less than 7.5% of the data is missing. If more than 7.5% of the data is missing, further analysis is needed.

`review_scores_value`, `review_scores_location`, `review_scores_checkin`, `review_scores_communication`, `review_scores_communication`, `review_scores_cleanliness`, `review_scores_accuracy` are scores used to rate indiividual aspects of the property/host while staying at the rental. The average of the scores are stored in the predictor `review_scores_rating`. Keeping all `review_scores_` predictors other than `review_scores_rating` would result in redundant information. We can remove them from the dataset.

Last, to prevent data leakage, we will split the data into a test set and a training set before we fill missing values

In [6]:
```
missing_count = main_df.isnull().sum()
perc_missing = (missing_count * 100)/len(main_df)
missing_df = pd.DataFrame({'Column': main_df.columns, 'Count': missing_count, '%': perc_missing})
missing_df.sort_values(by='%', ascending=False).reset_index(drop=True)
```

Out[6]:

	Column	Count	%
0	neighbourhood	4913	41.037421
1	host_response_time	3449	28.808887
2	host_response_rate	3449	28.808887
3	review_scores_value	3020	25.225526
4	review_scores_location	3020	25.225526
5	review_scores_checkin	3019	25.217173
6	review_scores_accuracy	3018	25.208821
7	review_scores_communication	3018	25.208821
8	review_scores_cleanliness	3018	25.208821
9	first_review	2946	24.607417
10	last_review	2946	24.607417
11	review_scores_rating	2946	24.607417
12	reviews_per_month	2946	24.607417
13	host_acceptance_rate	2862	23.905780
14	host_neighbourhood	1718	14.350150
15	bedrooms	711	5.938857
16	beds	150	1.252923
17	host_location	18	0.150351
18	bathrooms_text	16	0.133645
19	host_since	3	0.025058
20	host_identity_verified	3	0.025058
21	host_has_profile_pic	3	0.025058
22	host_total_listings_count	3	0.025058
23	host_listings_count	3	0.025058
24	host_is_superhost	3	0.025058
25	minimum_maximum_nights	1	0.008353

	Column	Count	%
26	maximum_maximum_nights	1	0.008353
27	maximum_nights_avg_ntm	1	0.008353
28	minimum_nights_avg_ntm	1	0.008353
29	maximum_minimum_nights	1	0.008353
30	minimum_minimum_nights	1	0.008353
31	longitude	0	0.000000
32	calculated_host_listings_count	0	0.000000
33	instant_bookable	0	0.000000
34	calculated_host_listings_count_entire_homes	0	0.000000
35	calculated_host_listings_count_private_rooms	0	0.000000
36	calculated_host_listings_count_shared_rooms	0	0.000000
37	neighbourhood_cleansed	0	0.000000
38	latitude	0	0.000000
39	accommodates	0	0.000000
40	property_type	0	0.000000
41	number_of_reviews_l30d	0	0.000000
42	amenities	0	0.000000
43	number_of_reviews	0	0.000000
44	availability_365	0	0.000000
45	availability_90	0	0.000000
46	availability_60	0	0.000000
47	availability_30	0	0.000000
48	has_availability	0	0.000000
49	room_type	0	0.000000
50	maximum_nights	0	0.000000
51	minimum_nights	0	0.000000

	Column	Count	%
52	price	0	0.000000
53	number_of_reviews_ltm	0	0.000000

In [7]:
```
main_df.drop(['review_scores_value', 'review_scores_location', 'review_scores_checkin', 'review_scores_commun
           'review_scores_communication', 'review_scores_cleanliness','review_scores_accuracy'], axis=1, inplac
```

In [8]:
```
#Split into training and test sets
train_df, test_df = tst(main_df, test_size=0.10, random_state=15413)
df_list = [train_df, test_df]

print('The amount of listings in the training set is %d' % len(train_df) )
print('The amount of listings in the test set is %d' % len(test_df) )
```

```
The amount of listings in the training set is 10774
The amount of listings in the test set is 1198
```

In [9]:
```
#Helper function to visualize categorical distribution
def get_normalized_count_plot(df, col, labels=None):
    """
        get normalized count for a dataframe column

        df: dataframe where column is located
        col: predictor to visualize
    """
    norm_df = df[col].value_counts(normalize=True).mul(100).reset_index().rename(
        columns={'index': col, col: '%'})

    sns.barplot(x=col,y='%', data=norm_df)
    if labels != None:
        label_ticks = np.arange(len(df[col].unique()))
        plt.xticks(ticks = [0,1], labels=labels)

    plt.title(f"Percentage Distribution of '{col}'")
    plt.xlabel(None)
    plt.show()
```

host_since

Currently, the predictor lists the date the host joined airbnb. To track their tenure, the date will be converted to the number of days from the hosts join date to the day the data was scraped (March 20, 2022). Only 3 listings are missing host tenure date; in fact, the three listings are missing all host data. Instead of imputing all host information, the three rows are dropped.

After conversion, we can see the days of the host's tenure is approximately normal. A majority of hosts have a tenure between 2000 and 3000 days.

In [10]:
```
def get_host_tenure(date_compiled, form, dates):
    "create column that counts the number of days a host has been with Airbnb"
    dates = pd.to_datetime(dates)
    date_compiled = pd.to_datetime(date_compiled, format=form)

    dates = (date_compiled - dates).dt.days
    return dates
```

In [11]:
```
train_df[train_df['host_since'].isnull()]
```

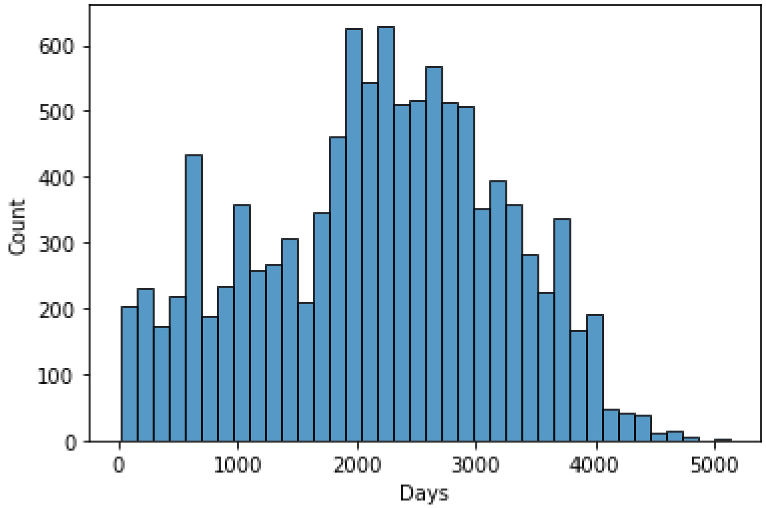
Out[11]:

	host_since	host_location	host_response_time	host_response_rate	host_acceptance_rate	host_is_superhost	host_neighbour
881	NaN	NaN	NaN	NaN	NaN	NaN	
807	NaN	NaN	NaN	NaN	NaN	NaN	
1491	NaN	NaN	NaN	NaN	NaN	NaN	

In [12]:
```
DATE_COMPILED = "2022-03-20"
for df_temp in df_list:
    df_temp.dropna(subset='host_since', inplace=True)
    df_temp['host_since'] = get_host_tenure(DATE_COMPILED,"%Y-%m-%d", df_temp['host_since'])
    df_temp.rename(columns={'host_since': 'host_tenure'},inplace=True)
```

In [13]:
```python
sns.histplot(x='host_tenure', data=train_df)
plt.xlabel('Days')
```

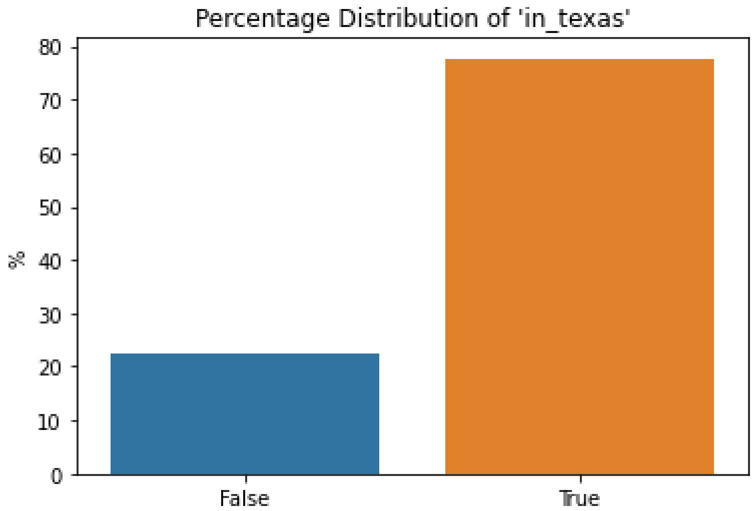
Out[13]: Text(0.5, 0, 'Days')



`host_location`

About 80% of the hosts' location are within Texas. On the other hand, only 18 listings are missing. Does a hosts proximity to the Airbnb affect how much they charge? I created a new predictor `in_texas` to measure the hosts proximity to their Airbnb. An 'unknown' category was created to track if the host's location was missing. We can see about 75% of hosts live in Texas, 20% live outside of the state and about 2% of listings do not have the hosts location listed. `host_location` was no longer needed so it was dropped.

In [14]:
```python
df_host_loc = train_df.copy()
df_host_loc.dropna(subset='host_location', inplace=True)
df_host_loc['in_texas'] = np.where(df_host_loc['host_location'].str.contains('Texas', case=False), 1, 0)
# df_host_loc.loc[df_host_loc['host_location'].str.contains('Texas', case=False), 'in_texas'] = 1

get_normalized_count_plot(df_host_loc, 'in_texas', labels=['False', 'True'])
```



In [15]:
```
for df1 in df_list:
    df1['in_texas'] = np.where(df1['host_location'].str.contains('Texas', case=False), 'Yes', 'No')
    df1.loc[df1['host_location'].isnull(), 'in_texas'] = 'unknown'
    df1.drop('host_location', axis=1, inplace=True)
```

In [16]:
```
train_df
```

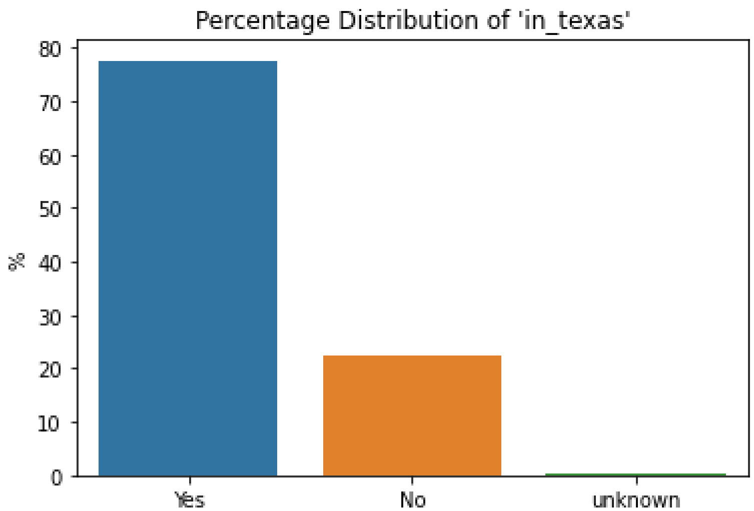
Out[16]:

	host_tenure	host_response_time	host_response_rate	host_acceptance_rate	host_is_superhost	host_neighbourhood	host_l
5022	3341	within a few hours	100%	83%	t	Rosewood	
1067	2684	within a few hours	100%	100%	f	Upper Boggy Creek	
604	2948	NaN	NaN	NaN	f	University of Texas	
1750	2701	NaN	NaN	NaN	f	Hyde Park	
8342	342	within a day	60%	87%	f	South Lamar	
...	...	...	...	...	...	...	
5592	2238	within an hour	100%	98%	t	NaN	
9913	923	within an hour	97%	99%	t	East Downtown	
889	2709	within an hour	100%	93%	t	Padre Island	

	host_tenure	host_response_time	host_response_rate	host_acceptance_rate	host_is_superhost	host_neighbourhood	host_l
9083	600	within an hour	100%	89%	f	Bouldin Creek	
8809	1440	within an hour	93%	99%	t	Steiner Ranch Neighborhood Association	

10771 rows × 48 columns

In [17]:
```
get_normalized_count_plot(train_df, 'in_texas')
```



`host_response_time`

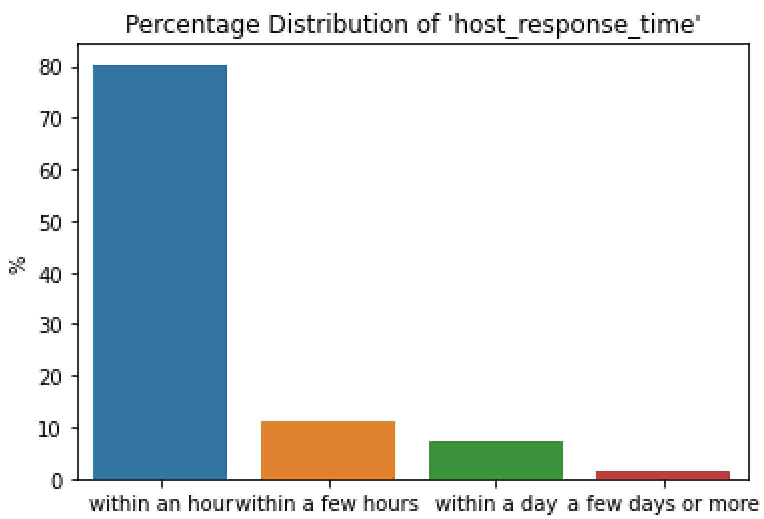
About 30% of listings are missing information for `host_response_time`. Understandibly, both `host_response_time` and `host_response_rate` are missing for the same listings in the dataset. If a host doesn't respond to an inquiry, then they wouldn't have a rate.

Of the listings with a host response time, about 80% of the hosts responded within an hour. When considering the high amount of missing data within `host_response_time`, its questionable if this predictor will provide much information. For now, we replaced the missing values with 'unkown'. However, further within the EDA we wil review the impact of the response time and the pricing of the Airbnb.

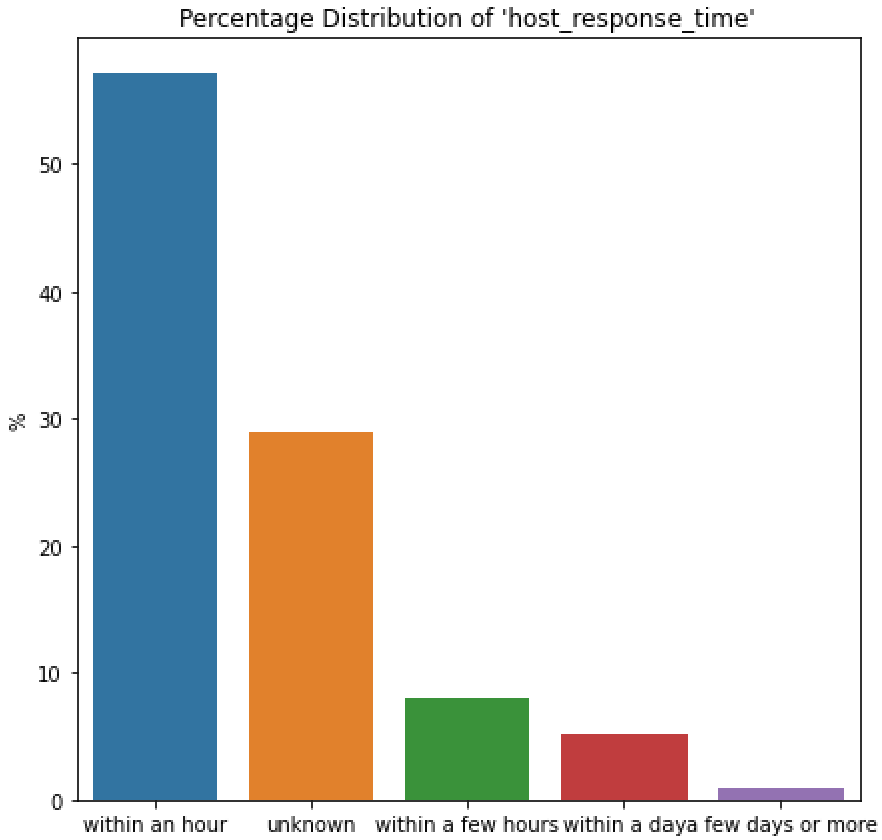
In [18]:
```
time_rate_miss =train_df[train_df.loc[:, ['host_response_time', 'host_response_rate']].isnull().sum(axis=1) ==
time_miss = train_df[train_df['host_response_time'].isnull()].shape[0]
rate_miss = train_df[train_df['host_response_rate'].isnull()].shape[0]

print("Percentage of 'host_response_time' where 'host_response_rate is also missing: %d'" %
      (time_rate_miss / time_miss))
print("Percentage of 'host_response_rate' where 'host_response_time is also missing: %d'" %
      (time_rate_miss / time_miss))
```

```
Percentage of 'host_response_time' where 'host_response_rate is also missing: 1'
Percentage of 'host_response_rate' where 'host_response_time is also missing: 1'
```

In [19]:
```
get_normalized_count_plot(train_df, 'host_response_time')
```



In [20]:
```
#After adding 'unknown' as part of the distribution
for df1 in df_list:
    df1['host_response_time'].fillna('unknown', inplace=True)

plt.figure(figsize=(7, 7))
get_normalized_count_plot(train_df, 'host_response_time')
```



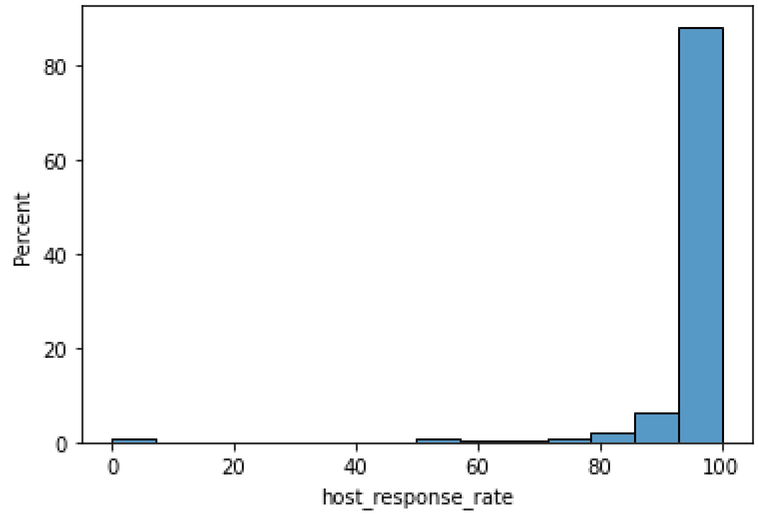
`host_response_rate`

About 80% of the received response rates are 100%. Very few listings have a rate below 80%. Because of the high percentage of missingness and listings with a 100% rating, the values are binned.

After binning, 86% of the listings either have a 100% host response rate or a horst response rate that is unknown.

In [21]:
```
for df1 in df_list:
    df1['host_response_rate'] = pd.to_numeric(df1['host_response_rate'].str.strip('%'))

sns.histplot(x='host_response_rate', data=train_df, stat='percent')
```

Out[21]:
```
<AxesSubplot:xlabel='host_response_rate', ylabel='Percent'>
```



In [22]:
```
for df1 in df_list:
    df1['host_response_rate'] =  pd.cut(df1['host_response_rate'], bins=[0, 90, 99, 100],
            labels=['Less than 90', '90-99', '100']).astype('str').str.replace('nan', 'unknown')

train_df['host_response_rate'].value_counts(normalize=True)
```

Out[22]:
```
100             0.563643
unknown         0.297001
90-99           0.073159
Less than 90    0.066196
Name: host_response_rate, dtype: float64
```

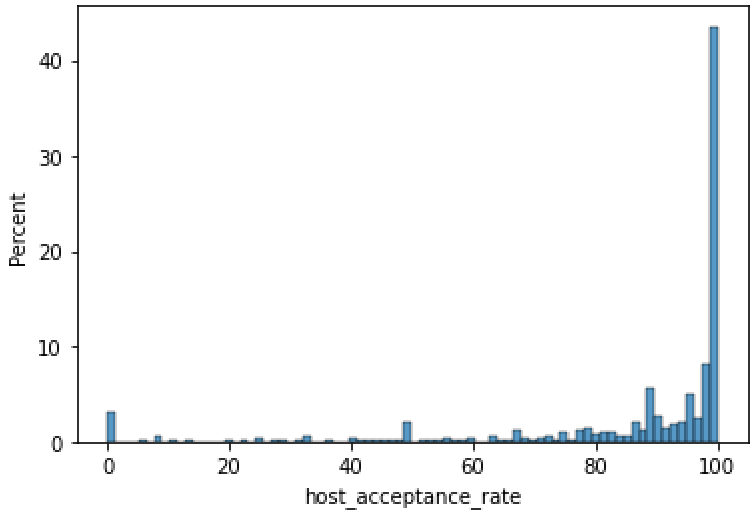
host_acceptance_rate

Like the response rate, the host's acceptance rate is heavily skewed to the left with a large percentage for the listings having a 100% host acceptance rate. The rates are placed into 5 bins.

After cleaning, we can see about 75% have an either have unknown acceptance rate or one higher than 90%.

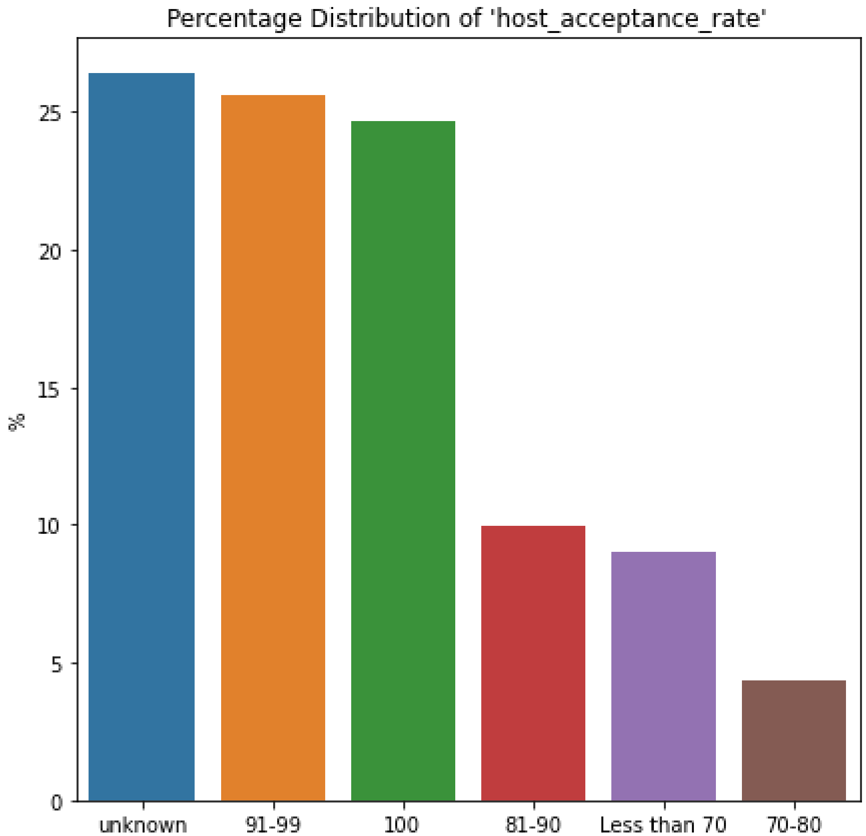
In [23]:
```
for df1 in df_list:
    df1['host_acceptance_rate'] = pd.to_numeric(df1['host_acceptance_rate'].str.strip('%'))

sns.histplot(x='host_acceptance_rate', data=train_df, stat='percent')
```

Out[23]:
```
<AxesSubplot:xlabel='host_acceptance_rate', ylabel='Percent'>
```



In [24]:
```
for df1 in df_list:
    df1['host_acceptance_rate'] = pd.cut(df1['host_acceptance_rate'], bins=[0, 70, 80, 90, 99, 100],
           labels=['Less than 70', '70-80', '81-90', '91-99', '100']) \
    .astype('str').str.replace('nan', 'unknown')
```

In [25]:
```
plt.figure(figsize=(7, 7))
get_normalized_count_plot(train_df, 'host_acceptance_rate')
```



`host_neighbourhood`

I'll drop `host_neighbourhood` from the dataset since we are capturing the hosts proximity with `in_texas` and `zipcode`. The predictor fails to add information to the model. All cities listed are in Texas. The listings with hosts that do not live in texas have a value of `NaN`.

In [26]:
```
for df1 in df_list:
    df1.drop('host_neighbourhood', axis=1, inplace=True)
```

`host_listings_count`

The value for `host_listing_count` and `host_total_listings_count` are the same for all listings. Therefore, I'll drop `host_total_listings_count`. Also, due to redundancy, the following predictors will be dropped as well:

- `calculated_host_listings_count`
- `calculated_host_listings_count_entire_homes`
- `calculated_host_listings_count_private_rooms`
- `calculated_host_listings_count_shared_rooms`

In [27]:
```
dif_listing_count = sum(~(train_df['host_total_listings_count'] == train_df['host_listings_count']))
print("The amount of listings where the host_total_listings_count is not the same as the host_listings_count:
     dif_listing_count)
```

```
The amount of listings where the host_total_listings_count is not the same as the host_listings_count:  0
```

In [28]:
```
for df1 in df_list:
    df1.drop(['host_total_listings_count', 'calculated_host_listings_count', 'calculated_host_listings_count_e
              'calculated_host_listings_count_private_rooms', 'calculated_host_listings_count_shared_rooms'],
             axis=1)
```

`neighbourhood`, `neighbourhood_cleansed`

`neighbourhood_cleansed` contains the zipcode of each listing. I will drop `neighbourhood`. Also, `latitude` and `longitude` contain information regarding the property's location. However, we will use these predictors for EDA. Once EDA has been completed, the predictors will be dropped. Last, `neighbourhood_cleansed` is changed to `zipcode`.

In [29]:
```
for df1 in df_list:
    df1.drop('neighbourhood', axis=1, inplace=True)
    df1['neighbourhood_cleansed'] = pd.Categorical(df1['neighbourhood_cleansed'])
    df1.rename(columns={'neighbourhood_cleansed': 'zipcode'}, inplace=True)
```

`Property Type` and `Room Type`

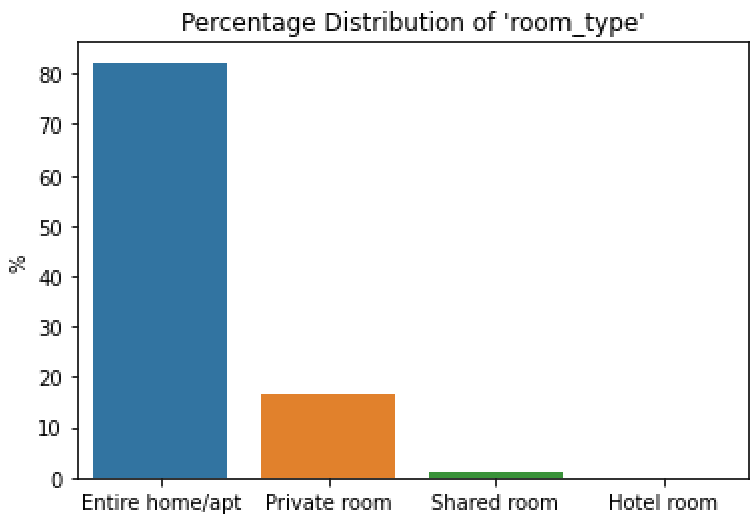
Due to the free form text, `property_type` has large cardinality. The property type can be extracted and placed within one of the following bins:

- House
- Rental Unit
- Apartment
- Loft
- Hotel

- Other

`room_type` gives more detail on how much private space the guest will have during the stay. About 80% of listings provide the tenant with the entire unit. Hotel rooms are only available in hotels. Therefore, almost 0% of the listings are hotel rooms. To remove this correlation, we will replace 'Hotel room' with 'Private Room'. `property_type` states what type of building the guest is staying in. Around 57% of listings provide a house, 25% a rental unit and 15% an aprartment.

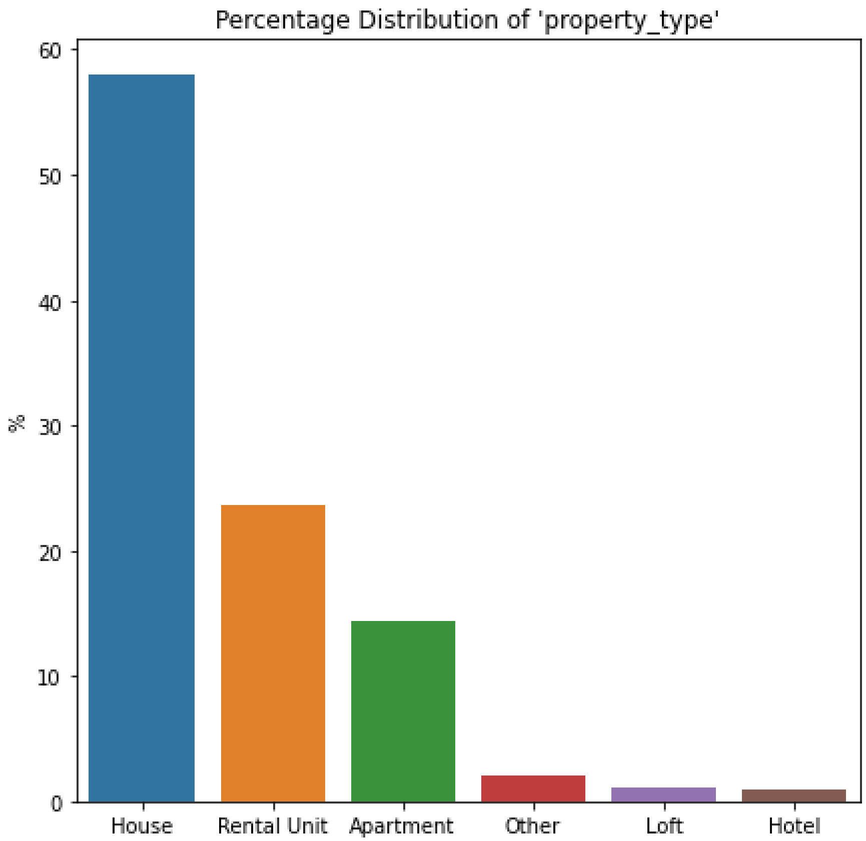
In [30]:
```
get_normalized_count_plot(train_df, 'room_type')
```



In [31]:
```
for df1 in df_list:
    df1.loc[df1['room_type'] == 'Hotel room', 'room_type'] = 'Private room'
    df1.loc[df1['property_type'].str.contains(
        'home|house|cottage|bungalow|villa|chalet|guest suite| casa ', case=False), 'property_type'] = 'House
    df1.loc[df1['property_type'].str.contains(
        'rental unit', case=False), 'property_type'] = 'Rental Unit'
    df1.loc[df1['property_type'].str.contains(
        'condominium | condo|apartment', case=False), 'property_type'] = 'Apartment'
    df1.loc[df1['property_type'].str.contains(
        'loft', case=False), 'property_type'] = 'Loft'
    df1.loc[df1['property_type'].str.contains(
        'hotel|resort|bed and breakfast', case=False), 'property_type'] = 'Hotel'
    df1.loc[~df1['property_type'].isin(
        ['House', 'Rental Unit', 'Apartment', 'Loft', 'Hotel']), 'property_type'] = 'Other'
```

In [32]:
```
plt.figure(figsize=(7, 7))
```

```
get_normalized_count_plot(train_df, 'property_type')
```



`bathrooms_text`

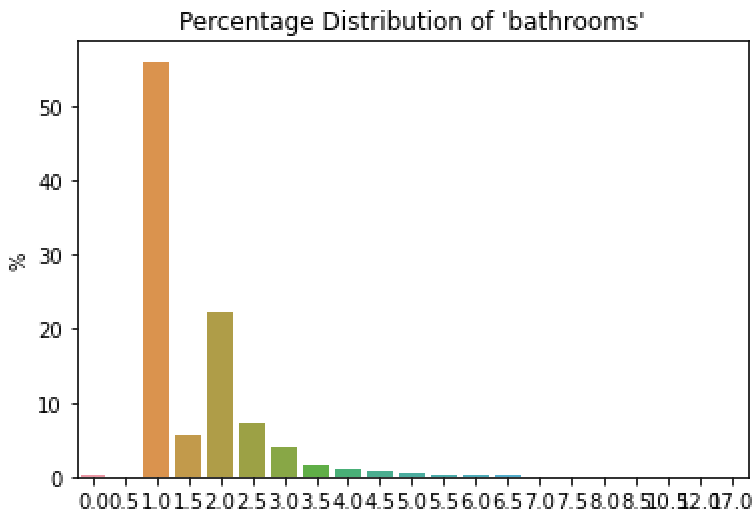
The original `bathrooms` contained no information. However, we can extract the number of bathrooms from `bathrooms_text` and create a new `bathrooms` column. `bathrooms_text` will be dropped.

As expected, approximately 95% of listings have between 1.0 and 4.0 bathrooms.

In [33]:
```
for df1 in df_list:
    df1['bathrooms'] = df1['bathrooms_text'].str.extract(r'(\d*\.?\d+)').astype(float)
    df1.loc[df1['bathrooms_text'].str.contains('half-bath', case=False, na=False), 'bathrooms'] = 0.5
    df1.drop('bathrooms_text', axis=1, inplace=True)

median_bath = train_df['bathrooms'].median()
```

```
for df1 in df_list:
    df1['bathrooms'].fillna(median_bath, inplace=True)
```

In [34]:
```
get_normalized_count_plot(train_df, 'bathrooms')
```



`amenities`

Amenities describe the 'extra' things the guest receives with the property. Some examples are a pool, washer, dryer, or gym. Currenntly, the amenities predictor is a string of a list of amenities. I will convert the string into a list. Afterwards, I'll extract each amenity from the list and create a new column stating whether the listing has it or not.

After extraction, we can drop `wifi`, `air conditioning`, and `pet` as nearly 100% of the listings are skewed to either True or False.

In [35]:
```
def clean_list_as_string(val):
    """
        function for lambda to change from list in a string to list of strings

        val: string of the list
    """

    val = val.strip("[]").replace(r"'", "").replace(", ", ",").replace('"', '').split(",")
    val = list(map(str.strip, val))
    return val
```

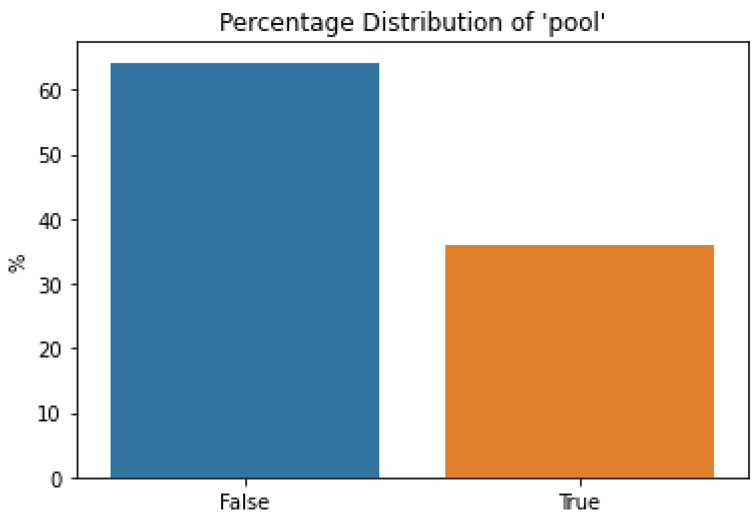
```python
def clean_amenities(df, col):
    """
        Clean each amenities string. Change from string of list to list of strings.

        col: the column name containing the amenities
    """

    df[col] = df[col].apply(clean_list_as_string)
    return df



def create_amenity_boolean_df(amenity_series, amenity_list):
    """
        Returns a dataframe stating whether each listing contained the specific amenity in their list

        amenity_series: series that contains the amenity list for each airbnb listing
        amenity_list: The list of amenities we want as predictors
    """

    amenity_dict = dict()
    for amenity in amenity_list:
        amenity = amenity.lower()
        amenity_dict[amenity] = amenity_series.apply(lambda x: any(amenity in string.lower() for string in x))
    return pd.DataFrame(amenity_dict)



def add_amenities(df, col, amenity_list):
    """
        Concatenate the boolean_df for the dataframe to the dataframe passed in the function

        df: DataFrame where the amenity series is located
        col: column containing the amenities
        amenity_list: The list of amenities we want as predictors
    """
    amenity_series = df[col]
    boolean_df = create_amenity_boolean_df(amenity_series, amenity_list)
    return pd.concat([df, boolean_df], axis=1)
```

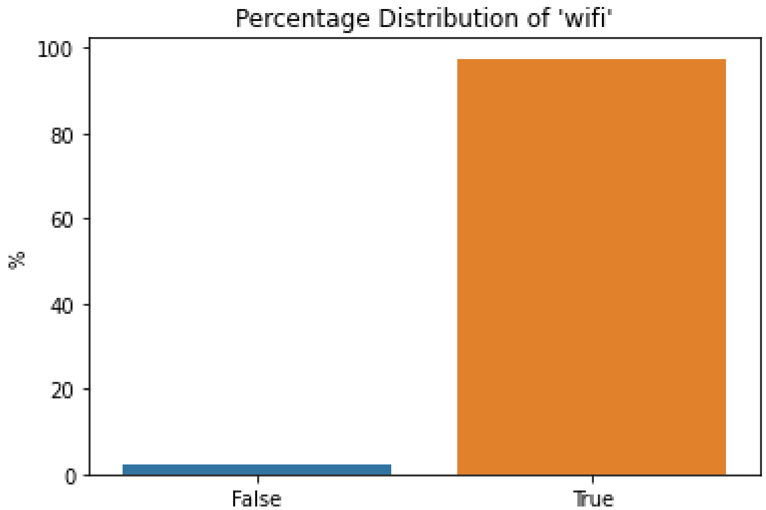
In [36]:

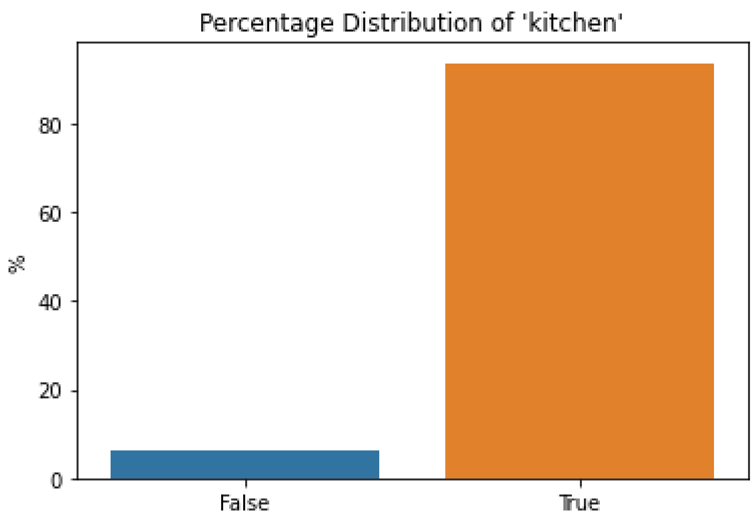
```python
amenity_try = ['pool', 'wifi', 'kitchen', 'free parking', 'free street parking','hot tub', 'washer'
               ,'air conditioning', 'workspace', 'pet', 'gym']
```

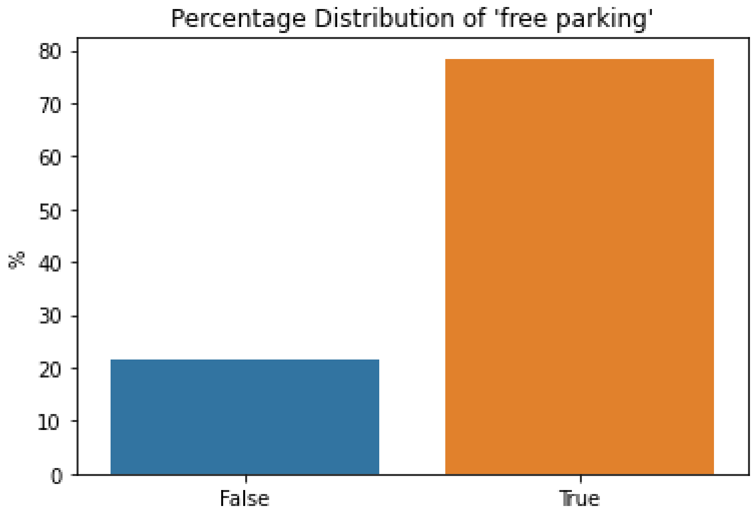
```
train_df = clean_amenities(train_df, 'amenities')
train_df = add_amenities(train_df, 'amenities', amenity_try)
train_df.drop('amenities', axis=1, inplace=True)

test_df = clean_amenities(test_df, 'amenities')
test_df = add_amenities(test_df, 'amenities', amenity_try)
test_df.drop('amenities', axis=1, inplace=True)
```

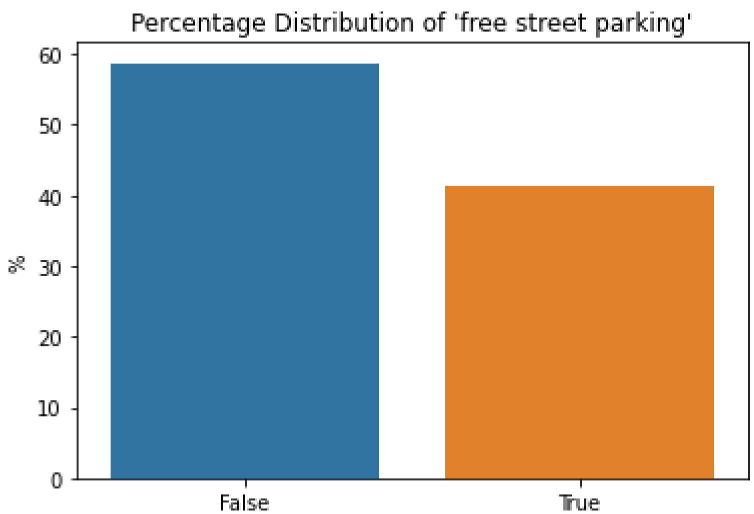
In [37]:
```
get_normalized_count_plot(train_df, 'pool')
get_normalized_count_plot(train_df, 'wifi')
get_normalized_count_plot(train_df, 'kitchen')
get_normalized_count_plot(train_df, 'free parking')
get_normalized_count_plot(train_df, 'free street parking')
get_normalized_count_plot(train_df, 'hot tub')
get_normalized_count_plot(train_df, 'washer')
get_normalized_count_plot(train_df, 'air conditioning')
get_normalized_count_plot(train_df, 'workspace')
get_normalized_count_plot(train_df, 'pet')
get_normalized_count_plot(train_df, 'gym')
```

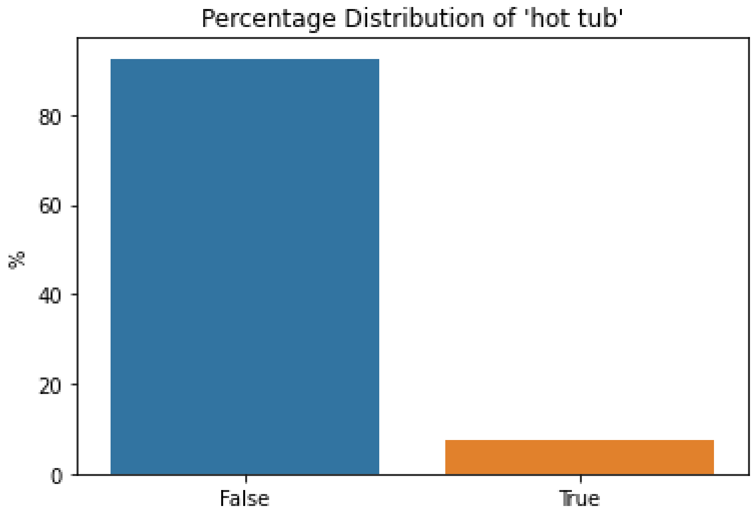



Percentage Distribution of 'wifi'
%
0
20
40
60
80
100
False
True

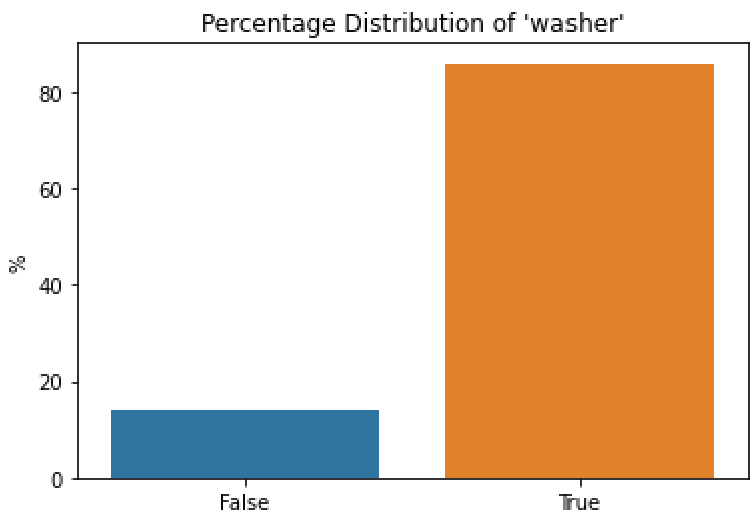

Percentage Distribution of 'kitchen'
%
0
20
40
60
80
False
True


Percentage Distribution of 'free parking'
%
0
10
20
30
40
50
60
70
80
False
True

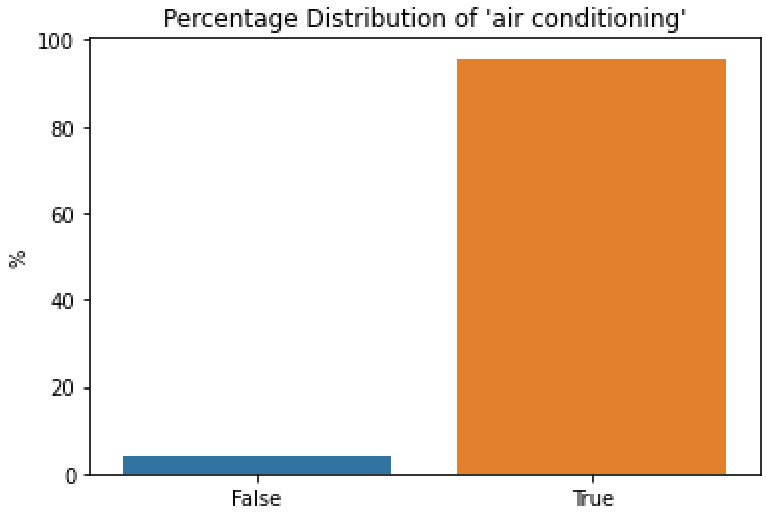

Percentage Distribution of 'free street parking'
%
0
10
20
30
40
50
60
False
True

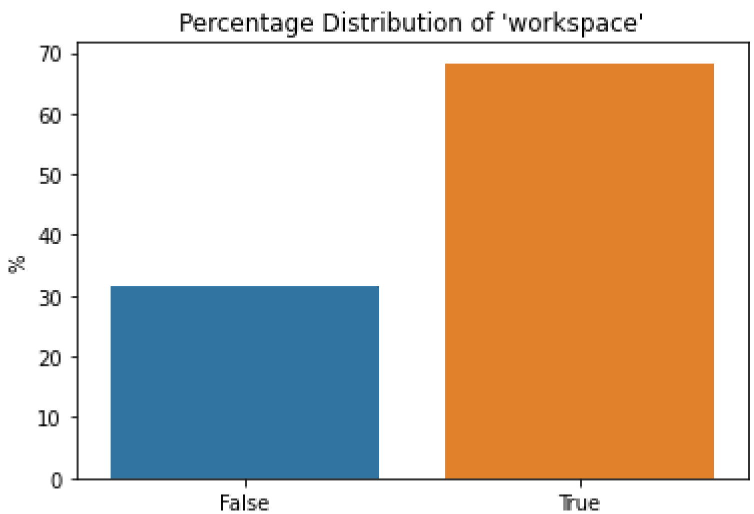


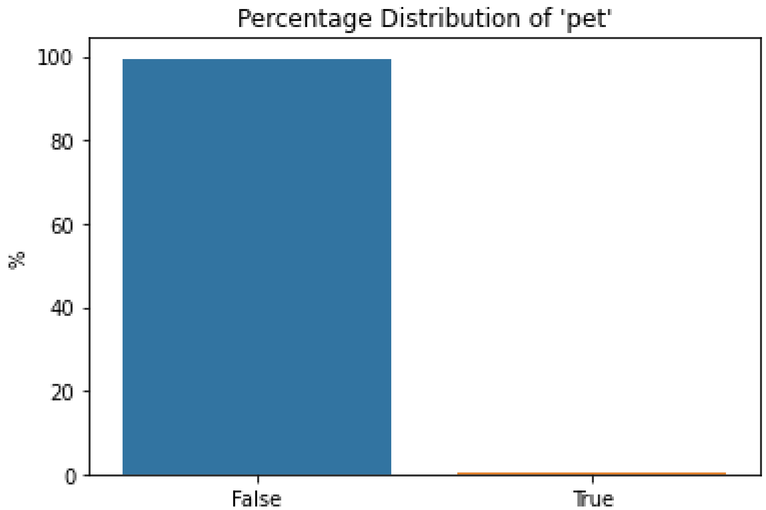
Percentage Distribution of 'hot tub'
%
0
20
40
60
80
False
True

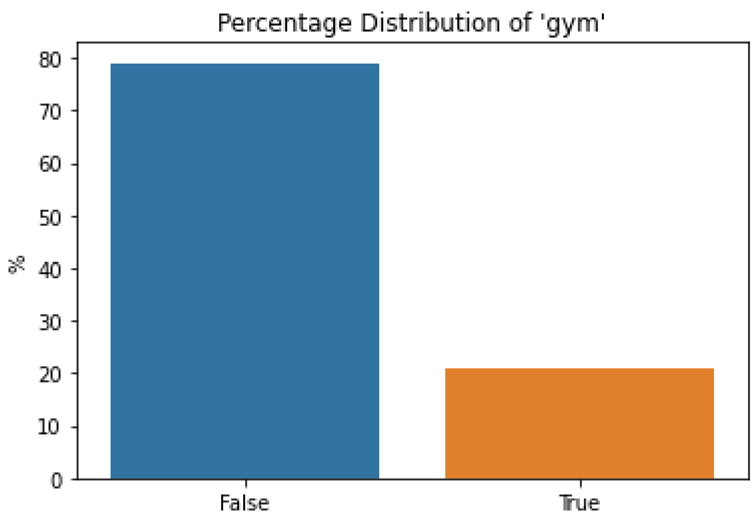


Percentage Distribution of 'washer'
%
0
20
40
60
80
False
True


Percentage Distribution of 'air conditioning'
100
80
60
40
20
0
%
False
True


Percentage Distribution of 'workspace'
70
60
50
40
30
20
10
0
%
False
True





In [38]:
```
train_df.head()
```

Out[38]:

	host_tenure	host_response_time	host_response_rate	host_acceptance_rate	host_is_superhost	host_listings_count	host_ha
5022	3341	within a few hours	100	81-90	t	1.0	
1067	2684	within a few hours	100	100	f	0.0	
604	2948	unknown	unknown	unknown	f	1.0	
1750	2701	unknown	unknown	unknown	f	1.0	
8342	342	within a day	Less than 90	81-90	f	5.0	

In [39]:

```
df_list = [train_df, test_df]
for df1 in df_list:
    df1.drop(['wifi', 'air conditioning', 'pet'], axis=1, inplace=True)
```

`price`

We need to convert `price` from a string to a float.

Also, Airbnb doesn't allow a listing price less than 10 dollars. Therefore, we will remove any row with a price less than 10 dollars.

The distribtuion of price is right-skewed and resembles a lognormal distribution. Airbnb has a max price filter of 12000 dollars: some of the listings in the dataset have prices well over the benchmark. Further research was conducted to find the accurate price of listings over 10000 dollars. The following correction were made:

- A listing is priced at 20000 dollars. However, further researcher shows the pricing was 10000 dollars around March 20, 2022. The price has been adjusted to 10000 dollars

Most prices are listed between 75 and 300 dollars. Very few listings are more than $10000 a night. We need to keep in mind that the prices listed are advertised prices. Many hosts tend to list the maximum price or the "smart price" as their nightly prices. However, many of these hosts tend to let charge their clients much less.

In [40]:

```
train_df['price'] = train_df['price'].str.replace("\$|,", "").astype('float')
```

```
train_df = train_df[train_df['price'] >= 10]

test_df['price'] = test_df['price'].str.replace("\$|,", "").astype('float')
test_df = test_df[test_df['price'] >= 10]
```

```
/var/folders/dv/14k38qsd37xgbxgkxr0c7fgm0000gn/T/ipykernel_4261/800042654.py:1: FutureWarning: The default va
lue of regex will change from True to False in a future version.
  train_df['price'] = train_df['price'].str.replace("\$|,", "").astype('float')
/var/folders/dv/14k38qsd37xgbxgkxr0c7fgm0000gn/T/ipykernel_4261/800042654.py:4: FutureWarning: The default va
lue of regex will change from True to False in a future version.
  test_df['price'] = test_df['price'].str.replace("\$|,", "").astype('float')
```

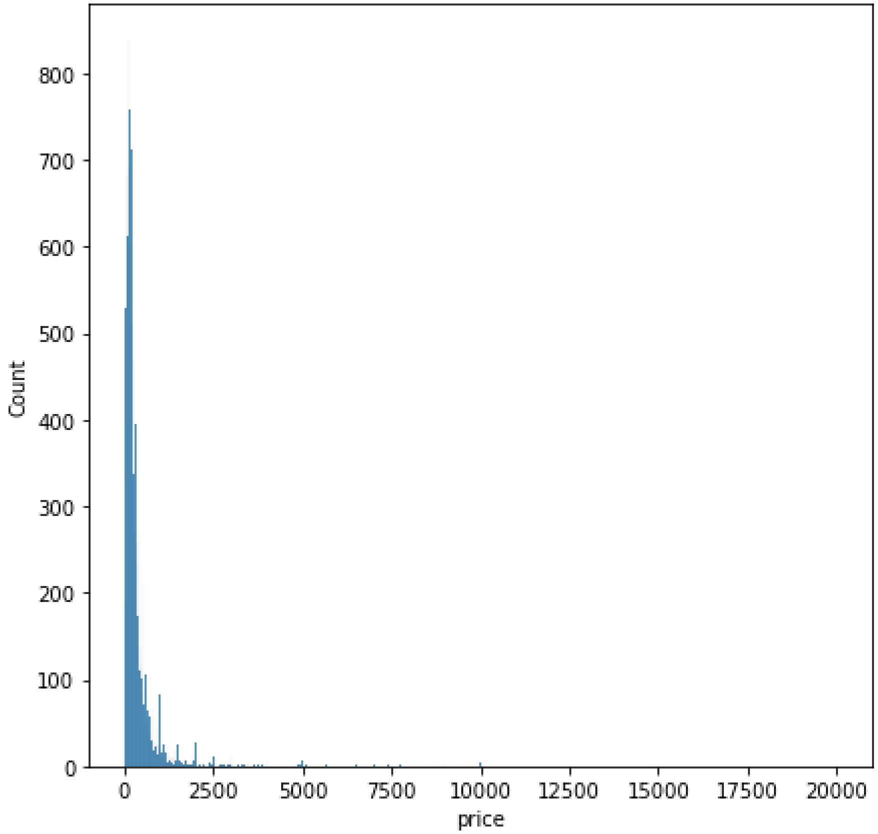
In [41]:
```
train_df[train_df['price'] >= 10000]
```

Out[41]:

	host_tenure	host_response_time	host_response_rate	host_acceptance_rate	host_is_superhost	host_listings_count	host_l
6507	787	a few days or more	unknown	unknown	f	0.0	
10559	725	within a day	90-99	81-90	t	0.0	
3713	4057	unknown	unknown	unknown	f	1.0	
10663	877	a few days or more	unknown	70-80	f	0.0	
7312	1308	within an hour	100	Less than 70	f	46.0	

In [42]:
```
plt.figure(figsize=(7, 7))
sns.histplot(x='price', data=train_df)
```

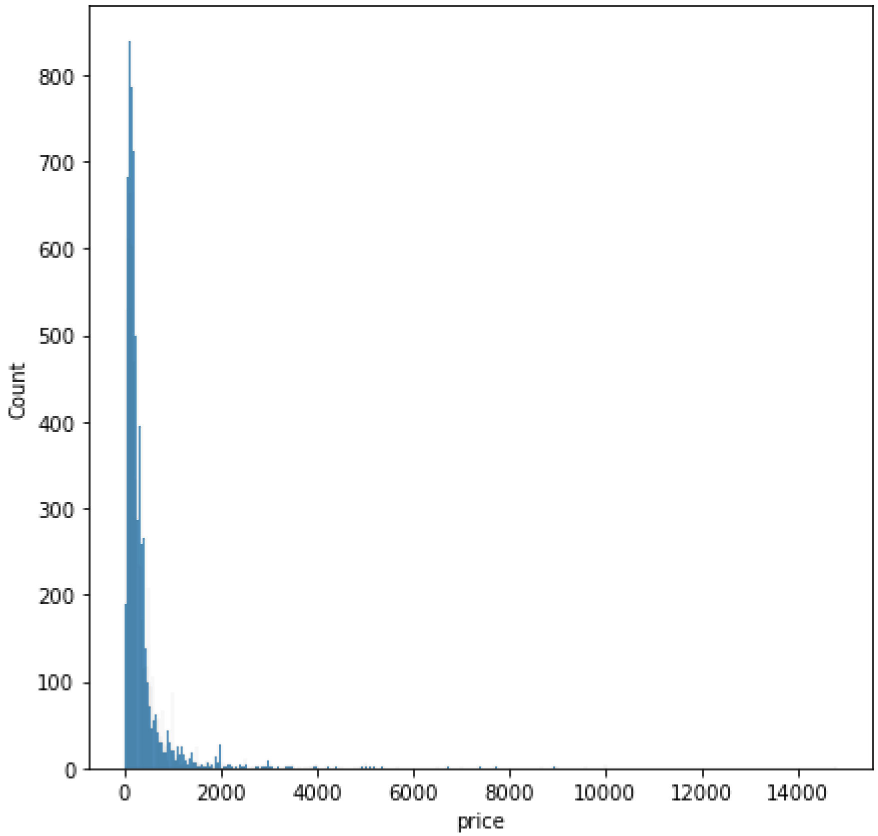
Out[42]:
```
<AxesSubplot:xlabel='price', ylabel='Count'>
```



In [43]:
```
train_df.loc[train_df['price'] == 20000, 'price'] = 10000
```

In [44]:
```
plt.figure(figsize=(7, 7))
sns.histplot(x='price', data=train_df)
```

Out[44]:
```
<AxesSubplot:xlabel='price', ylabel='Count'>
```



`minimum_nights` and `maximum_nights`

`minimum_nights` states the required minimum nights to stay at the listing. `minimum_minimum_nights` is a value *Inside Airbnb* calculated to predict the minimum required nights for a listing in the next 365 days. The same concept applies to `maximum_nights` and `maximum_maximum_nights`. Most of the calculated `minimum_*` features are the same as `minimum_nights`. The same applies to `maximum_nights`. Thus, we will drop:

- `minimum_minimum_nights`
- `maximum_minimum_nights`
- `minimum_maximum_nights`
- `maximum_maximum_nights`
- `minimum_nights_avg_ntm`
- `maximum_nights_avg_ntm`

The distribution of `minimum_nights` is heavily right-skewed. 75% of listings require 1, 2, or 3 minimum nights. On the other hand, `maximum_nights` has a median maximum night count of 375 days. Over 4000 listings have a maximum night requirement of 1000 days.

In [45]:

```
dif_listing_count = (sum((train_df['minimum_nights'] != train_df['minimum_minimum_nights'])) / train_df.shape
print('The % of listings where minimum_nights does not equal minimum_minimum_nights is', dif_listing_count)

dif_listing_count = (sum((train_df['maximum_nights'] != train_df['maximum_maximum_nights'])) / train_df.shape
print('The % of listings where maximum_nights does not equal maximum_aximum_nights is', dif_listing_count)
```

```
The % of listings where minimum_nights does not equal minimum_minimum_nights is 7.763744427934621
The % of listings where maximum_nights does not equal maximum_aximum_nights is 20.533060921248143
```

In [46]:

```
df_list = [train_df, test_df]
for df1 in df_list:
    df1.drop(['minimum_minimum_nights', 'maximum_minimum_nights', 'minimum_maximum_nights', 'maximum_maximum_n
              'minimum_nights_avg_ntm', 'maximum_nights_avg_ntm'], axis=1, inplace=True)
```

In [47]:

```
train_df['minimum_nights'].value_counts(normalize=True).head(5)
```

Out[47]:

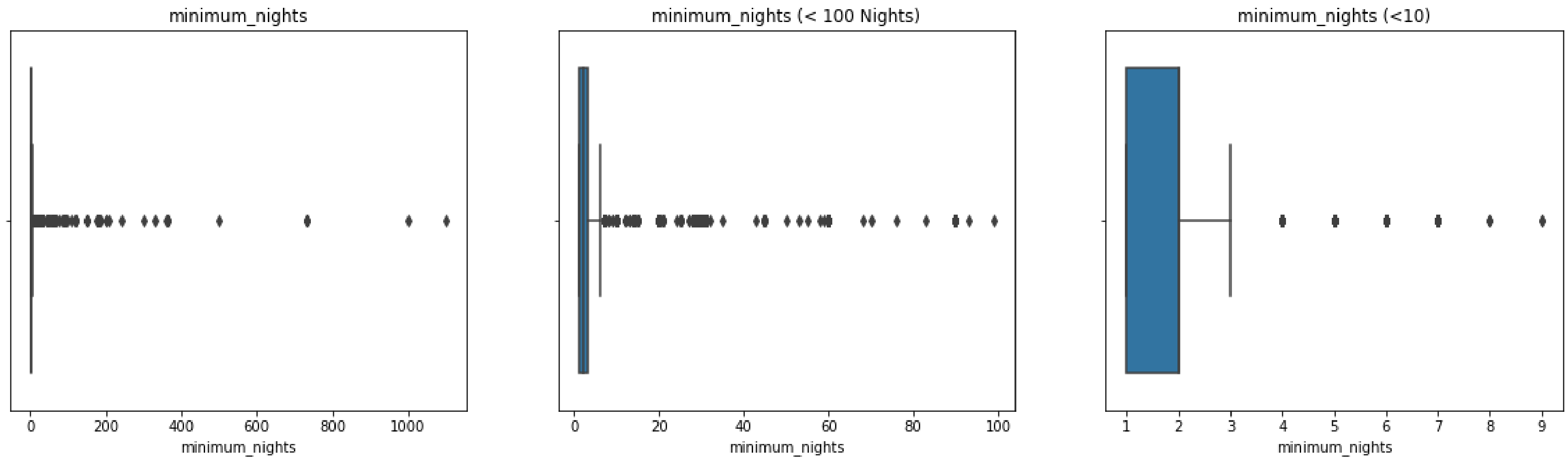
```
2     0.334881
1     0.309621
3     0.122493
30    0.106798
4     0.024981
Name: minimum_nights, dtype: float64
```

In [48]:

```
fig, ax = plt.subplots(1,3, figsize=(20,5))
sns.boxplot(x="minimum_nights", data=train_df, ax=ax[0])
ax[0].set_title("minimum_nights")

sns.boxplot(x="minimum_nights", data=train_df[train_df['minimum_nights'] < 100], ax=ax[1])
ax[1].set_title("minimum_nights (< 100 Nights)")

sns.boxplot(x="minimum_nights", data=train_df[train_df['minimum_nights'] < 10], ax=ax[2])
ax[2].set_title("minimum_nights (<10)")
```

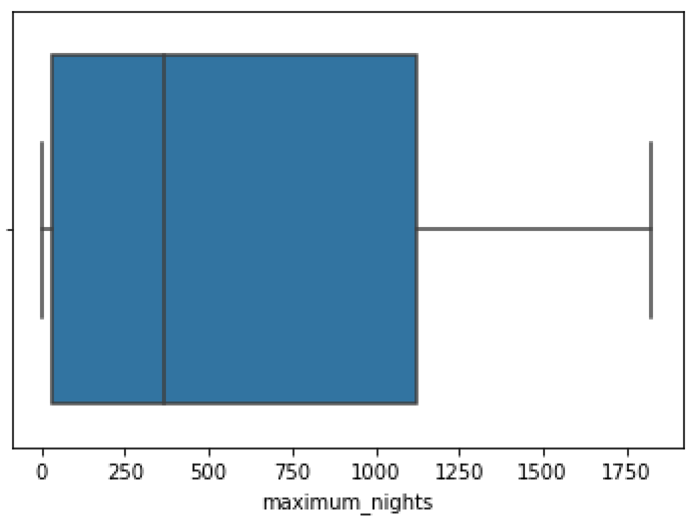
Out[48]:

```
Text(0.5, 1.0, 'minimum_nights (<10)')
```



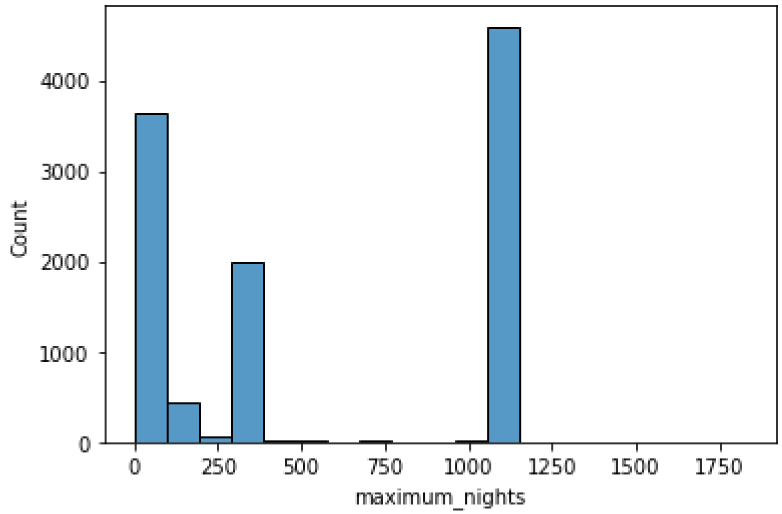
In [49]:
```
sns.boxplot(x="maximum_nights", data=train_df)
```

Out[49]:
```
<AxesSubplot:xlabel='maximum_nights'>
```



In [50]:
```
sns.histplot(x="maximum_nights", data=train_df)
```

Out[50]:
```
<AxesSubplot:xlabel='maximum_nights', ylabel='Count'>
```



has_availability

Similar to `minimum_nights`, the features related to `has_availability` are forecasts from *Inside Airbnb*. The predictors forecast at least 30% of the listings will be available for all 3 predictor availability predictors. However, currently 87% of listings are available. We will drop the following columns due to the uncertainty of how reliable the forecasts are:

- `availability_30`
- `availability_60`
- `availability_90`

In [51]:
```
train_df['has_availability'].value_counts(normalize=True).head(5)
```

Out[51]:
```
t    0.873328
f    0.126672
Name: has_availability, dtype: float64
```

In [52]:
```
train_df['availability_30'].value_counts(normalize=True).head(5)
```

Out[52]:
```
0     0.408432
30    0.054978
10    0.031296
1     0.027953
8     0.026653
Name: availability_30, dtype: float64
```

In [53]:
```
train_df['availability_60'].value_counts(normalize=True).head(5)
```

Out[53]:
```
0     0.346025
60    0.053027
10    0.016995
40    0.016159
31    0.015695
Name: availability_60, dtype: float64
```

In [54]:
```
train_df['availability_90'].value_counts(normalize=True).head(5)
```

Out[54]:
```
0     0.316029
90    0.051542
83    0.012630
40    0.012444
70    0.012444
Name: availability_90, dtype: float64
```

In [55]:
```
train_df['availability_365'].value_counts(normalize=True).head(5)
```

Out[55]:
```
0      0.281575
365    0.044112
364    0.006408
345    0.005944
70     0.005944
Name: availability_365, dtype: float64
```

In [56]:
```
for df1 in df_list:
    df1.drop(['availability_30', 'availability_60', 'availability_90', 'availability_365'], axis=1, inplace=Tr
```

number_of_reviews_ltm

Will remove `number_reviews_ltm` (number of reviews the last 12 months) and `reviews_per_month` due to their high correlation with `number_of_reviews`.

Within the the last 30 days the listings were scraped, more than 90% of the listings have between 0 and 5 days. About 90% of listings have less than 100 reviews.

In [57]:
```
print('Correlation between # of review and # of reviews the last 12 months: ',
      train_df['number_of_reviews'].corr(train_df['number_of_reviews_ltm']))
print('Correlation between # of review and # of reviews the last 30 days: ',
      train_df['number_of_reviews'].corr(train_df['number_of_reviews_l30d']))
```

```
print('Correlation between # of review and # of reviews the last 30 days: ',
      train_df['number_of_reviews'].corr(train_df['reviews_per_month']))
```

```
Correlation between # of review and # of reviews the last 12 months:  0.6883637747838991
Correlation between # of review and # of reviews the last 30 days:  0.47224232715841236
Correlation between # of review and # of reviews the last 30 days:  0.590968701160347
```

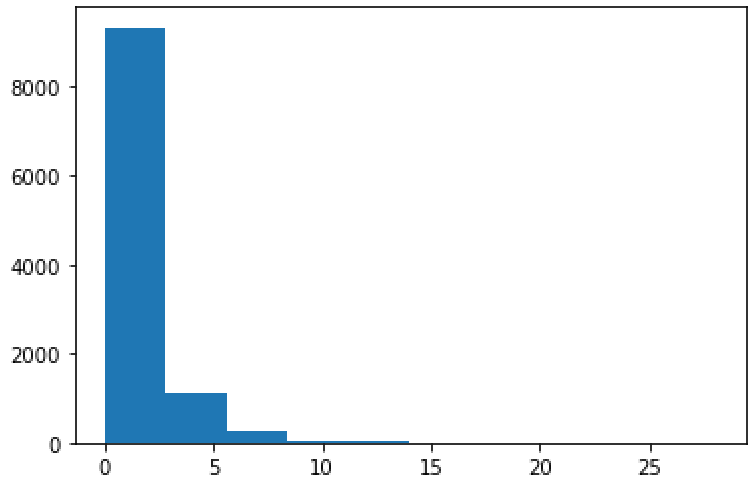
In [58]:

```
for df1 in df_list:
    df1.drop(['number_of_reviews_ltm', 'reviews_per_month'], axis=1, inplace=True)
```

In [59]:

```
train_df["number_of_reviews_l30d"].hist(grid=False)
```

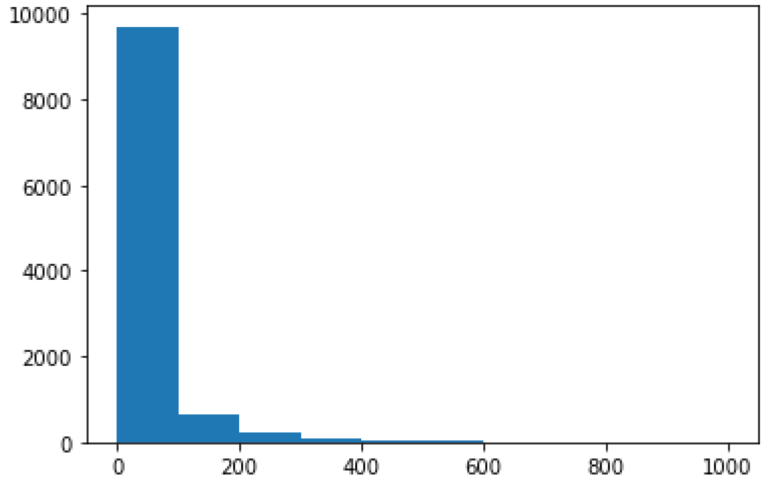
Out[59]: <AxesSubplot:>



In [60]:

```
train_df["number_of_reviews"].hist(grid=False)
```

Out[60]: <AxesSubplot:>



`first_review` and `last_review`

When reading reviews clients are evaluating the quality of the the lisitng. However, using the number of days from both `first_review` and `last_review` will measure the age more than the quality of the listing. Depending on the effect either predictor has on the price, the magnitude will be larger for older listings. As a result, we will drop both from the model

In [61]:
```
for df1 in df_list:
    df1.drop(['first_review', 'last_review'], axis=1, inplace=True)
```

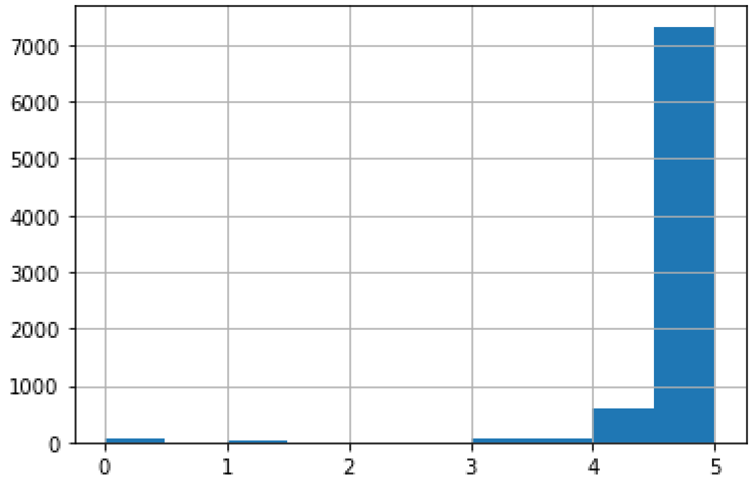
review_scores_rating

In [62]:
```
df_review_rating = train_df[train_df['review_scores_rating'].notnull()]
df_review_rating[df_review_rating['review_scores_rating'] > 4.5].shape[0] / df_review_rating.shape[0]
```

Out[62]:
```
0.8774582104228122
```

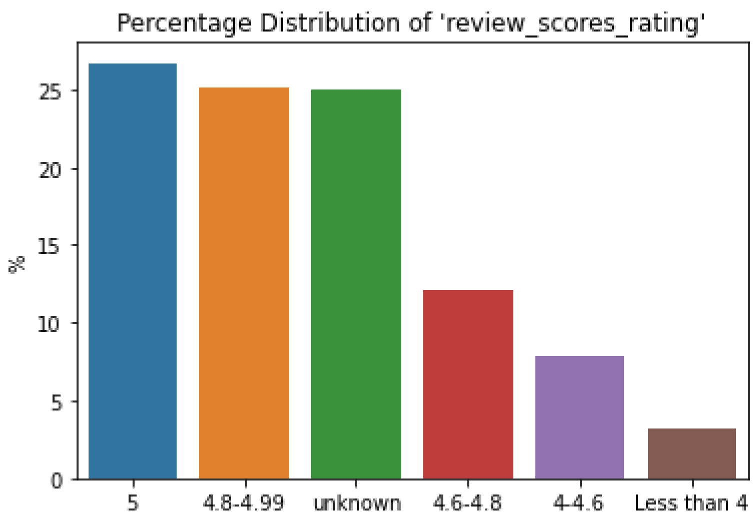
In [63]:
```
train_df['review_scores_rating'].hist()
```

Out[63]:
```
<AxesSubplot:>
```




```
In [64]: for df1 in df_list:
             df1['review_scores_rating'] = pd.cut(df1['review_scores_rating'], bins=[0, 4, 4.6, 4.8, 4.99, 5],
                           labels=['Less than 4', '4-4.6', '4.6-4.8', '4.8-4.99', '5']).astype('str')
             df1['review_scores_rating'] = df1['review_scores_rating'].str.replace('nan', 'unknown')
```

```
In [65]: get_normalized_count_plot(train_df, 'review_scores_rating')
```



`beds` and `bedrooms`

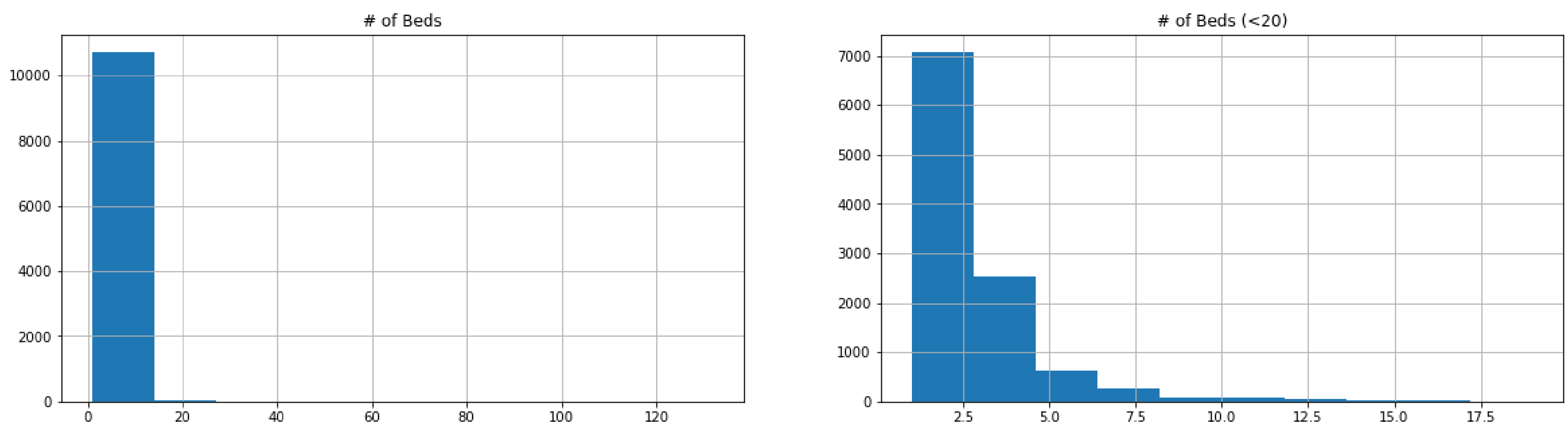
We are missing about 5% of `bedrooms` data and 1% of data for `beds`. We can fill missing values for both columns with their respective medians.

The distributions of beds and bedrooms are heavily skewed to the right, as expected. For both predictors, Most listings have less than 5.

In [66]:
```
bed_median = train_df['beds'].median()
bedroom_median = train_df['bedrooms'].median()

for df1 in df_list:
    df1['beds'].fillna(bed_median, inplace=True)
    df1['bedrooms'].fillna(bedroom_median, inplace=True)
```

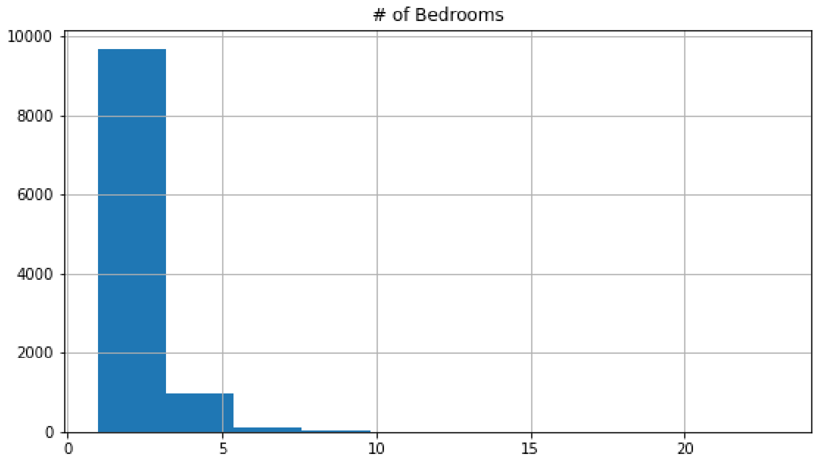
In [67]:
```
fig, ax = plt.subplots(1,2, figsize=(20,5))
train_df['beds'].hist(ax=ax[0])
ax[0].set_title('# of Beds')

train_df[train_df['beds'] < 20]['beds'].hist(ax=ax[1])
ax[1].set_title('# of Beds (<20)')
```

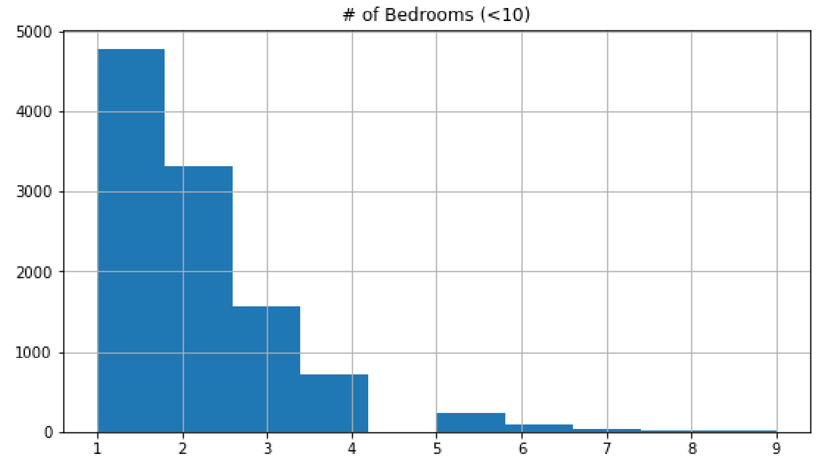
Out[67]:
```
Text(0.5, 1.0, '# of Beds (<20)')
```



In [68]:
```
fig, ax = plt.subplots(1,2, figsize=(20,5))
train_df['bedrooms'].hist(ax=ax[0])
ax[0].set_title('# of Bedrooms')

train_df[train_df['bedrooms'] < 10]['bedrooms'].hist(ax=ax[1])
ax[1].set_title('# of Bedrooms (<10)')
```

Out[68]: Text(0.5, 1.0, '# of Bedrooms (<10)')





In [69]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10768 entries, 5022 to 8809
Data columns (total 34 columns):
 #   Column                  Non-Null Count  Dtype   
---  ------                  --------------  -----   
 0   host_tenure             10768 non-null  int64   
 1   host_response_time      10768 non-null  object  
 2   host_response_rate      10768 non-null  object  
 3   host_acceptance_rate    10768 non-null  object  
 4   host_is_superhost       10768 non-null  object  
 5   host_listings_count     10768 non-null  float64 
 6   host_has_profile_pic    10768 non-null  object  
 7   host_identity_verified  10768 non-null  object  
 8   zipcode                 10768 non-null  category
 9   latitude                10768 non-null  float64 
 10  longitude               10768 non-null  float64 
 11  property_type           10768 non-null  object  
 12  room_type               10768 non-null  object  
 13  accommodates            10768 non-null  int64   
 14  bedrooms                10768 non-null  float64 
 15  beds                    10768 non-null  float64 
 16  price                   10768 non-null  float64 
 17  minimum_nights          10768 non-null  int64   
 18  maximum_nights          10768 non-null  int64   
 19  has_availability        10768 non-null  object  
 20  number_of_reviews       10768 non-null  int64   
 21  number_of_reviews_l30d  10768 non-null  int64   
 22  review_scores_rating    10768 non-null  object  
 23  instant_bookable        10768 non-null  object  
 24  in_texas                10768 non-null  object  
 25  bathrooms               10768 non-null  float64 
 26  pool                    10768 non-null  bool    
 27  kitchen                 10768 non-null  bool    
 28  free parking            10768 non-null  bool    
 29  free street parking     10768 non-null  bool    
 30  hot tub                 10768 non-null  bool    
 31  washer                  10768 non-null  bool    
 32  workspace               10768 non-null  bool    
 33  gym                     10768 non-null  bool    
dtypes: bool(8), category(1), float64(7), int64(6), object(12)
memory usage: 2.5+ MB
```

In [70]:

```
train_df.head(5)
```

Out[70]:

	host_tenure	host_response_time	host_response_rate	host_acceptance_rate	host_is_superhost	host_listings_count	host_ha
5022	3341	within a few hours	100	81-90	t	1.0	
1067	2684	within a few hours	100	100	f	0.0	
604	2948	unknown	unknown	unknown	f	1.0	
1750	2701	unknown	unknown	unknown	f	1.0	
8342	342	within a day	Less than 90	81-90	f	5.0	

Post-Cleaning Thoughts

Overall, we dropped about 50 features while creating 10. Unfortunately, I believe the number of `bathrooms`, `restrooms`, `beds` and the number of `accommodates` will be highly correlated. This can cause issues when modeling data. First, multicollinearity can increase a predictor's variance within. Also, if they are collinear, then we are overfitting data in every model. Thus, we may need to drop some of them. However, they are important when discussing the prices of a house. Therefore, we will wait until we review multicollinearity before dropping them.

In [71]:

```
train_df.to_csv('data/train_df')
test_df.to_csv('data/test_df')
```

Exploratory Data Analysis

Earlier we took a quick glance at the distribtuions of different predictors. Now, we will take a further look into a few predictors and their relationship to our dependent variable: `price`.

First, `price` is heavily skewed to the right. About 817 listings are priced at more than 1000 dollars per night. A majority of listings lie between 0 and 400 dollars. It's important to note that Airbnb doesn't allow a property to list under 10 dollars a night.

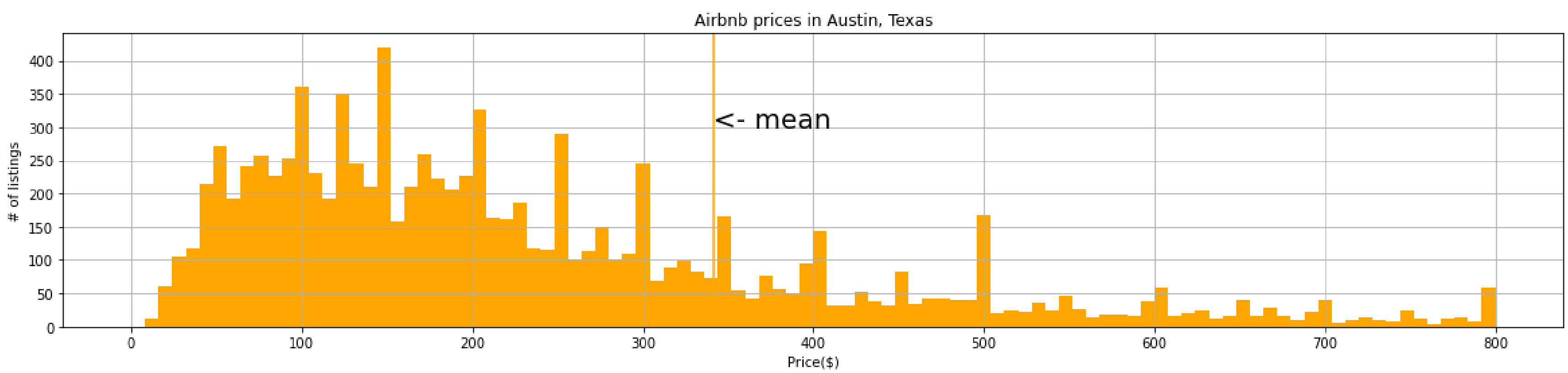
Two listings have a price for over 10000 dollars. The prices do not seem to be an error as one house has 23 beds with numerous amenities and the other has 15 beds a variety of amenities as well.

In [72]:
```
print('The # of listings with a price above $1000: ', train_df[train_df['price'] > 800].shape[0])
```

```
The # of listings with a price above $1000:  817
```

In [73]:
```
price_mean = train_df['price'].mean()

plt.figure(figsize=(20,4))
train_df.price.hist(bins=100, range=(0,800), color='orange')
plt.axvline(price_mean, color='orange')
plt.annotate('<- mean', xy= (price_mean,300), fontsize=20, color='black')
plt.title('Airbnb prices in Austin, Texas')
plt.xlabel('Price($)')
plt.ylabel('# of listings')
```

Out[73]:
```
Text(0, 0.5, '# of listings')
```



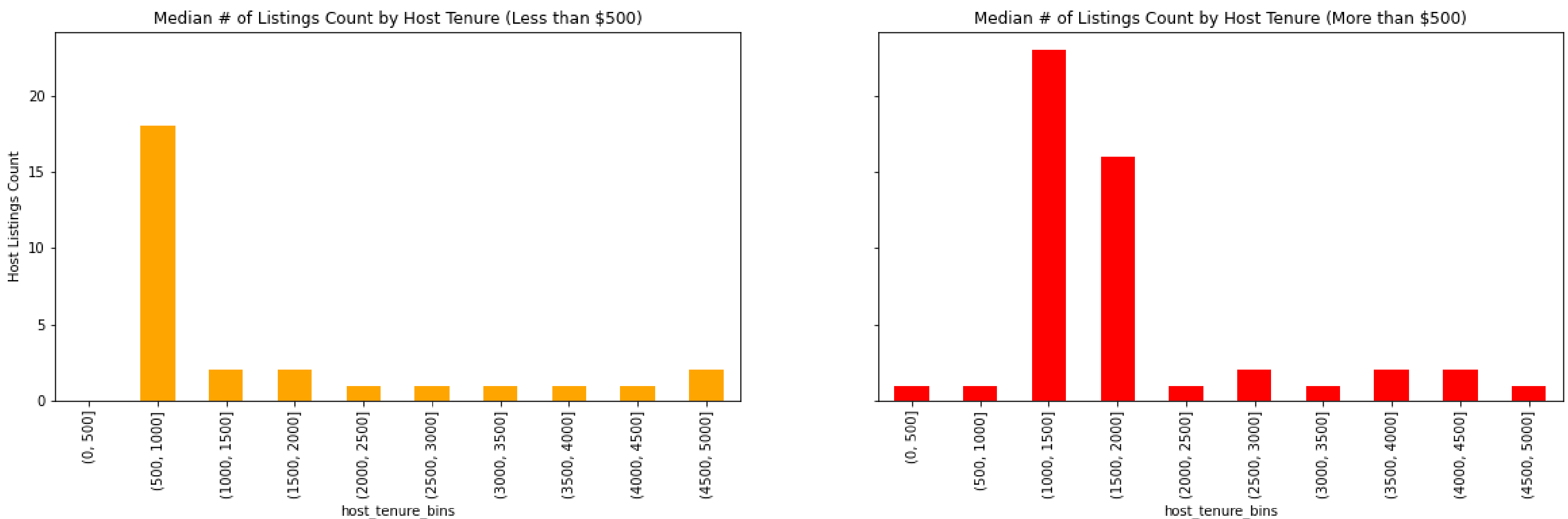
host_tenure

For listings priced for less than 500 dollars, hosts with a tenure between 500 and 1000 days have much more listings than hosts (by about 15 listings). On the otherhand, for listings prices more than 500 dollars, hosts with a tenure between 1000 and 2000 days tend to have more listings than hosts with a tenure outside of the range.

In [74]:
```
df2 = train_df.copy()
df2['host_tenure_bins'] = pd.cut(df2['host_tenure'],
                                 bins=[0,500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000])
```

```
fig, ax = plt.subplots(1,2, figsize=(20,5), sharey=True)

df2[df2['price'] < 500].groupby('host_tenure_bins')['host_listings_count'].median().plot.bar(color='orange', a
ax[0].set_title('Median # of Listings Count by Host Tenure (Less than $500)')
ax[0].set_ylabel('Host Listings Count')

df2[df2['price'] > 500].groupby('host_tenure_bins')['host_listings_count'].median().plot.bar(color='red', ax=a
ax[1].set_title('Median # of Listings Count by Host Tenure (More than $500)')
ax[1].set_ylabel('Host Listings Count')
```

Out[74]: Text(0, 0.5, 'Host Listings Count')

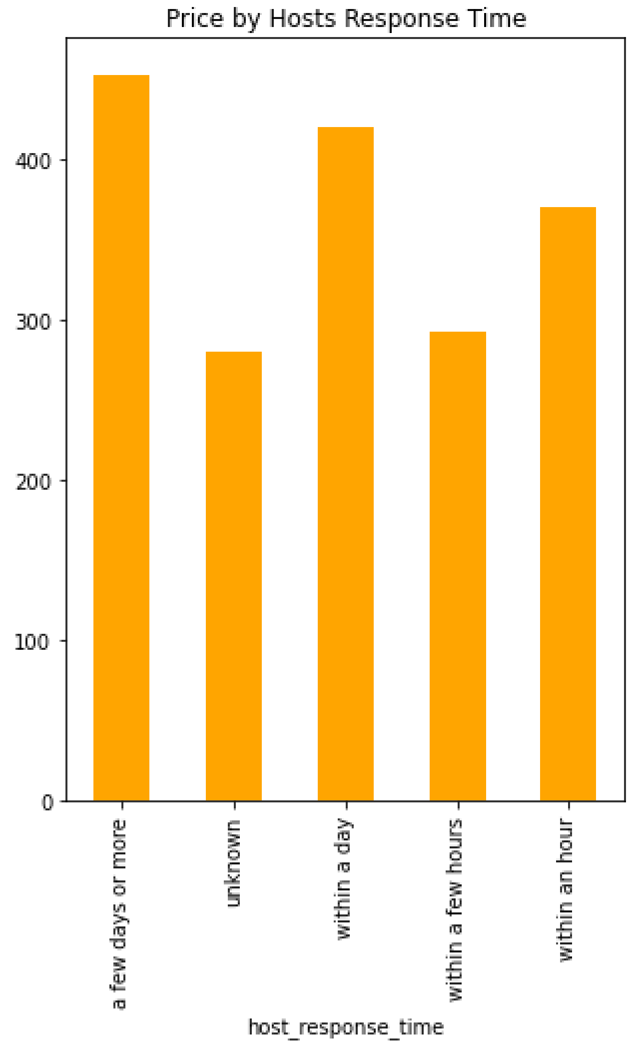


`host_response_time`

Hosts of the more expensive properties tend to take longer to response to inquiries. Surprisingly, the hosts that have an unknown response time have cheaper listing prices.

In [75]:
```
plt.figure(figsize=(5,7))
df2.groupby('host_response_time')['price'].mean().plot.bar(color='orange')
plt.title('Price by Hosts Response Time')
```

Out[75]: Text(0.5, 1.0, 'Price by Hosts Response Time')



`host_is_superhost`

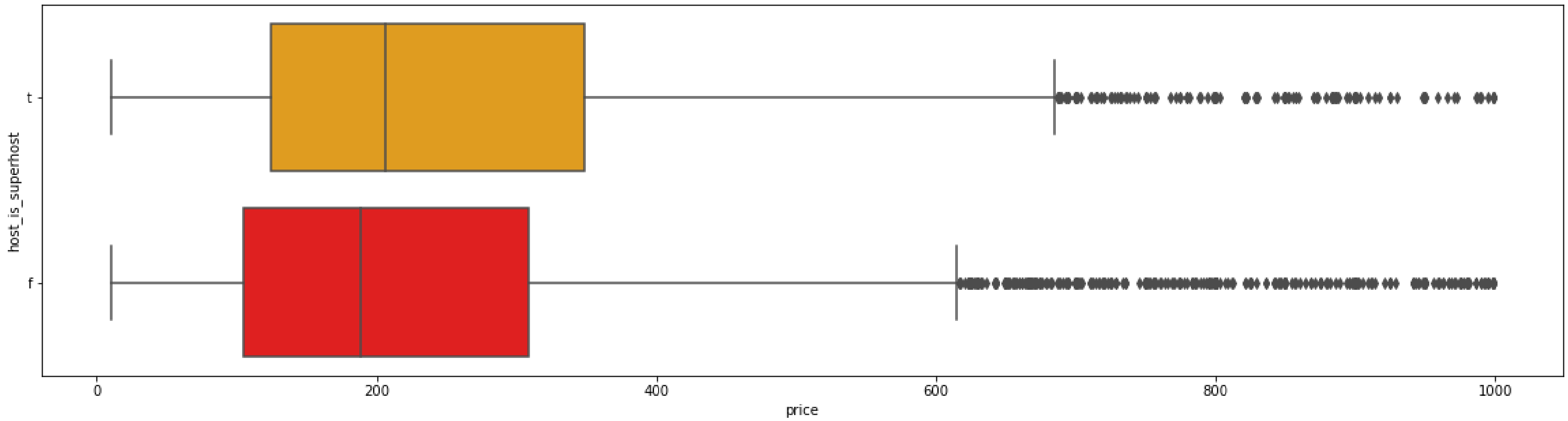
Listings without a superhost have sligtly lower prices than those that do. A superhost is someone who goes above and beyond for their tenants. One would expect to pay more for high quality service. There are quite a few outlier in both distributions. This is expected as larger and better quality properties will price much higher than average ones.

In [76]:
```
plt.figure(figsize=(20,5))
df2 = train_df[train_df['price'] < 1000]
sns.boxplot(df2['price'], df2['host_is_superhost'], orient="h", palette= {"t": "orange", "f": "red"})
```

```
/opt/anaconda3/lib/python3.9/site-packages/seaborn/_decorators.py:36: FutureWarning: Pass the following varia
bles as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, and passing
other arguments without an explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```

Out[76]:

```
<AxesSubplot:xlabel='price', ylabel='host_is_superhost'>
```

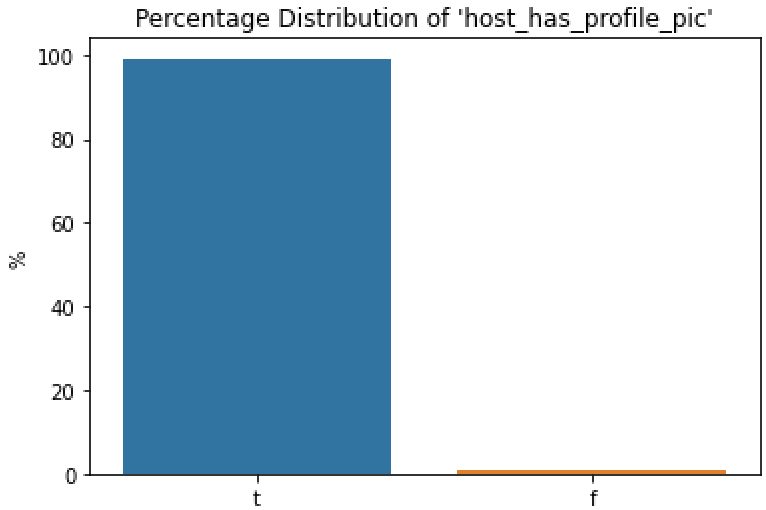


`host_has_profile_pic`

Almost 100% of the listings have hosts with a profile pic. I will drop this predictor as it won't provide much information to our model.

In [77]:

```python
get_normalized_count_plot(train_df, 'host_has_profile_pic')
```



In [78]:
```
for df1 in df_list:
    df1.drop('host_has_profile_pic', axis=1, inplace=True)
```

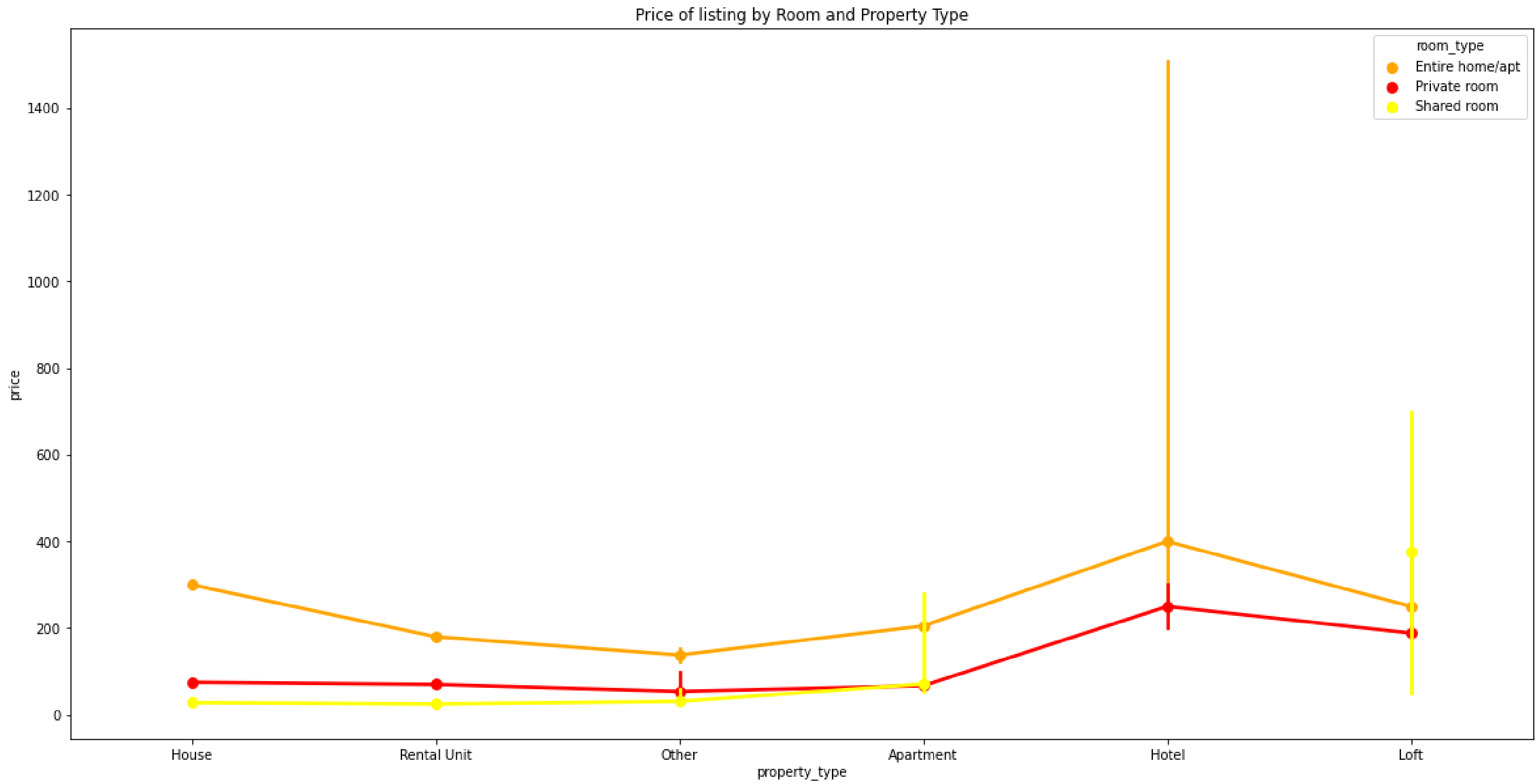
`property_type` and `room_type`

Hosts of hotel rooms tend to have more variable prices than other property types. This is understandable due to the varying types of hotel rooms available. Cheaper hotel may contain a bed and a television. On the other end, expensive hotel rooms tend to have more luxurious features such as hot tubs, private pools, multiple bedrooms. Also, it is understandable that hotel rooms are more expensive. Airbnb was created to create a cheaper alternative to hotel rooms. Thus, hotel rooms advertised on the platform will be more expensive.

Overall, it is more expensive to rent an entire home or apartment. Shared rooms are more expensive for hotel rooms and lofts than private rooms, otherwise they are prices relatively the same.

In [79]:
```
plt.figure(figsize=(20,10))
sns.pointplot(x='property_type', y='price', hue='room_type', data=train_df, estimator=np.median,
             palette= {"Entire home/apt": "orange", "Private room": "red", "Shared room": "yellow"})
plt.title('Price of listing by Room and Property Type')
```

Out[79]:
```
Text(0.5, 1.0, 'Price of listing by Room and Property Type')
```



`accomodates`, `bedrooms`, `beds`

For all three predictors, price increases as the quantity increases.

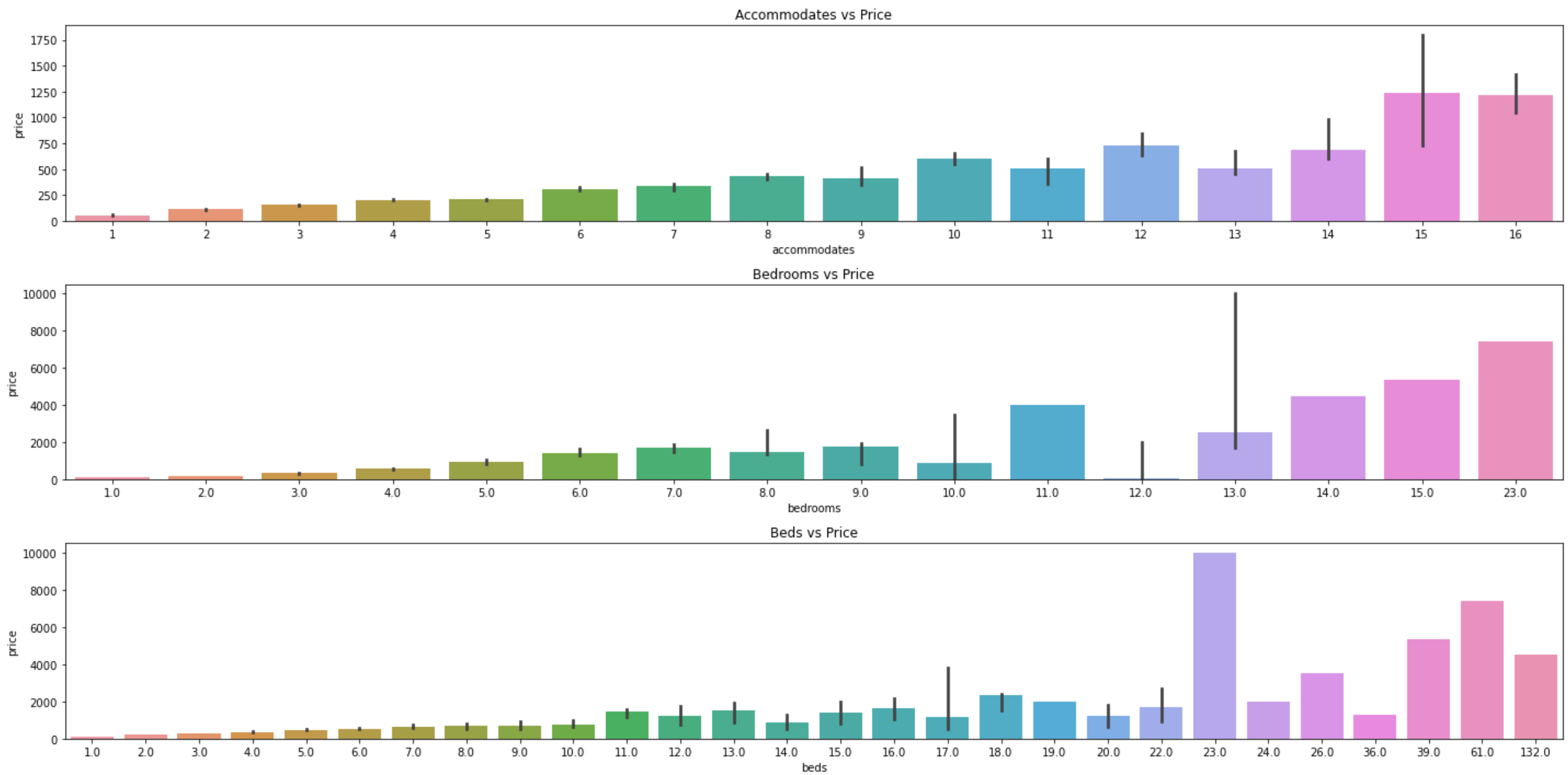
`beds` has a few outliers. However, upon further research, we found the data to be accurate. The outlier beds represent the following:

- 132 beds: The host is lending an entire lodge that hosts a large size of people
- 61 beds: An entire hotel is available for renting
- 39 beds: An entire townhouse with a pool and roofdeck
- 26 bedrs: An entire townhose with a pools and roof deck

The confidence interval for listings with 13 `bedrooms` is quite large. The listing which is causing a the large standard error is listed at a price of 10000 dollars while the other two cost 2532 and 1704 dollars. The 10000 dollar listing a is a property connecting two houses in one of the most coveted areas in Austin. On the other hand, the 2532 dollar home is on a campground and the 1704 is a house in a typical Austin residential neighborood. The large variance with a small sample size makes the large confidence interval accurate.

In [80]:

```
fig, ax = plt.subplots(3,1, figsize=(20,10))
sns.barplot(x='accommodates', y='price', data=train_df, ax=ax[0], estimator=np.median)
ax[0].set_title('Accommodates vs Price')

sns.barplot(x='bedrooms', y='price', data=train_df, ax=ax[1], estimator=np.median)
ax[1].set_title('Bedrooms vs Price')

sns.barplot(x='beds', y='price', data=train_df, ax=ax[2], estimator=np.median)
ax[2].set_title('Beds vs Price')
fig.tight_layout()
```



`Latitude`, `Longitude` and `zipcode`

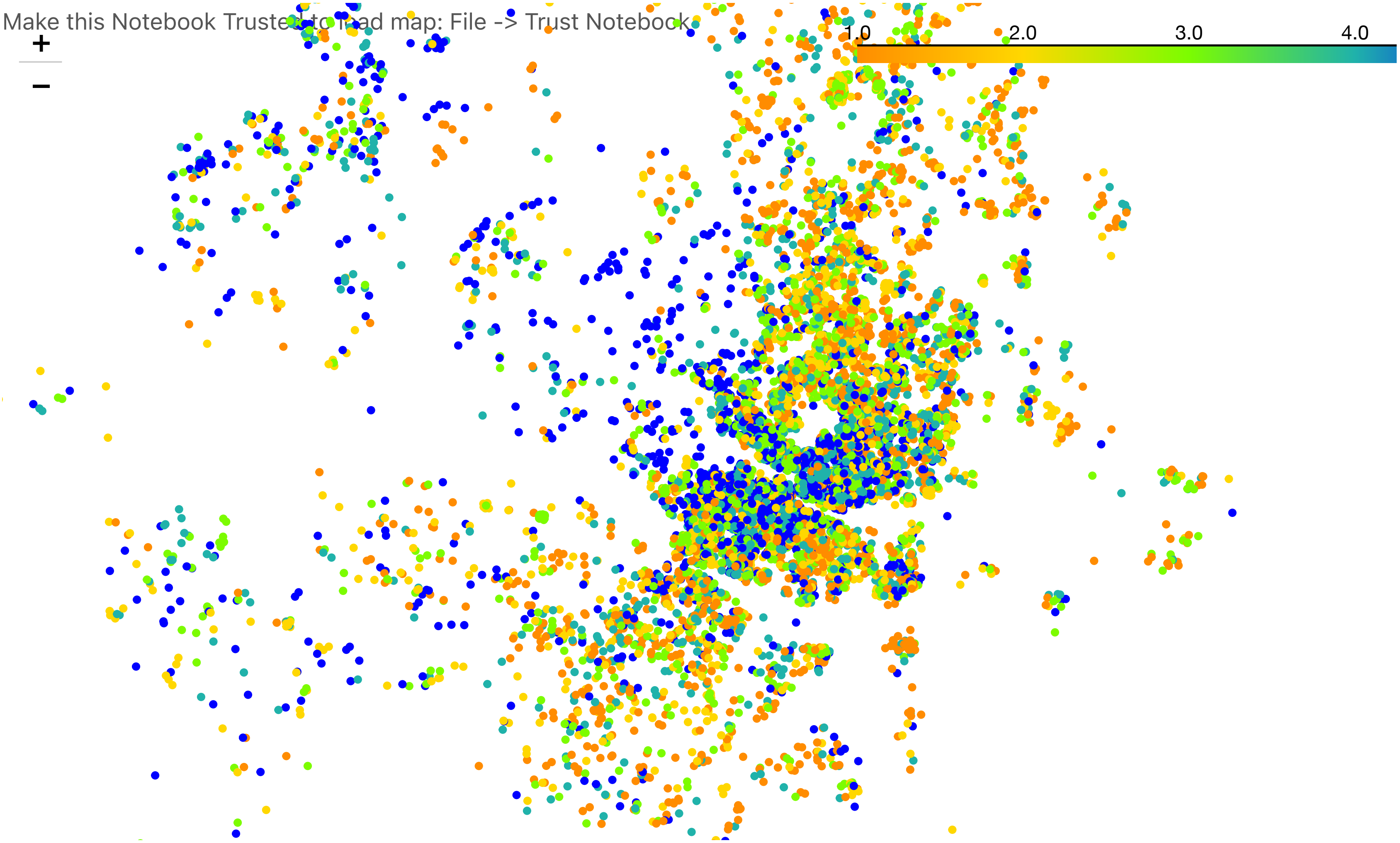
The three most expensive airbnbs are located in the northwest area of Austin Most predominantly, they are located near Lae Travis and downtown. of expensive Airbnb's are located either downtown or on the west side of Austin. All other price ranges seem to be evenly dispersed throughout Austin. The least expensive areas are located in southern Austin.

In [81]:

```
df= train_df.copy()
# df['price_quantiles'] = pd.qcut(x=df['price'], q=4, labels = [''])
AUSTIN_COORD = [30.2672, -97.7431]
quantiles = np.quantile(df['price'], q=[0.0, 0.2, 0.4, 0.6, 0.8, 1])
print(quantiles)
#['Under $100', '$100 - $165', '$165 - $250', '$250 - $424', 'Above $424']
df['price_cut'] = pd.cut(x=df['price'], bins=quantiles, labels=np.arange(1,6))
df['price_cut'] = pd.to_numeric(df['price_cut'])
df['price_cut']. fillna(1, inplace=True)
print(df['price_cut'].unique())

cmap = cm.LinearColormap(colors=["darkorange", "gold", "lawngreen", "lightseagreen", 'blue'],
                        vmin=1, vmax=5)
austin_map = folium.Map(location = AUSTIN_COORD, width=950, height=550,
                       zoom_start=10.5, tiles= 'cartodb positron')

for lat, long, price in zip(df['latitude'], df['longitude'], df['price_cut']):
    folium.Circle(
        location = [lat, long],
        radius=5,
        fill = True,
        color = cmap(price)
    ). add_to(austin_map)

austin_map.add_child(cmap)
```

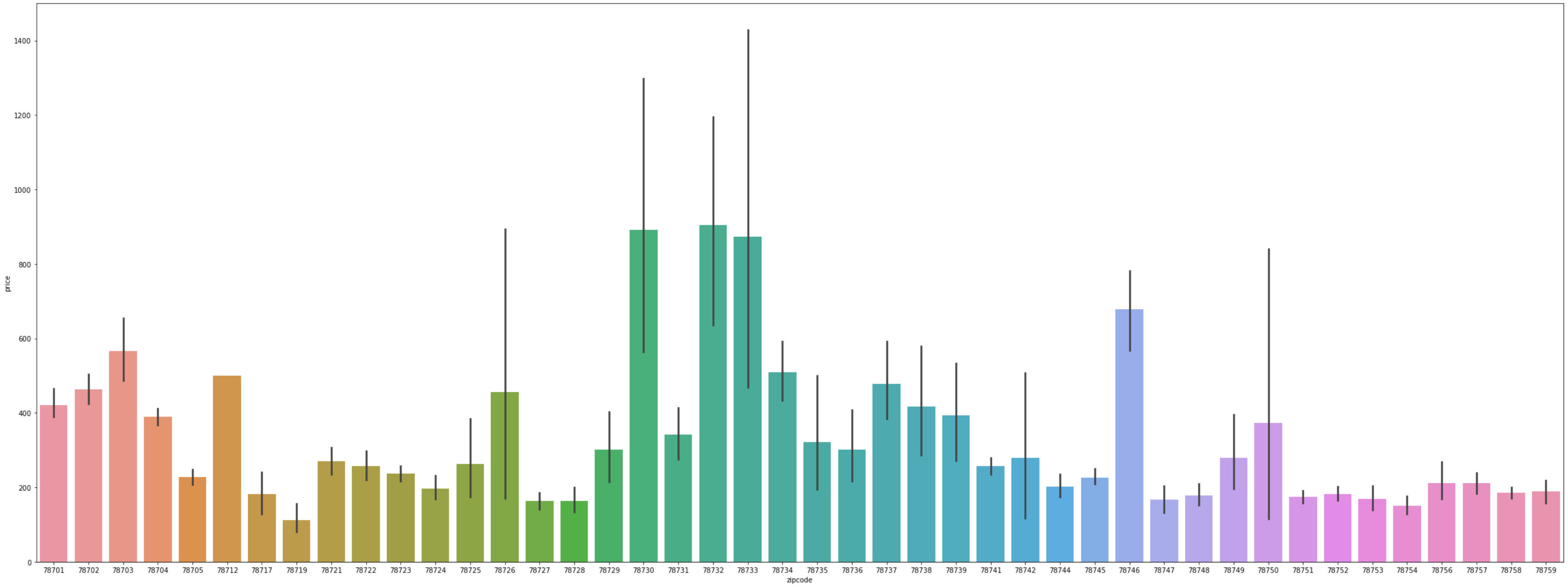
```
[1.0000e+01 1.0000e+02 1.6600e+02 2.5000e+02 4.2660e+02 1.4795e+04]
[1. 5. 2. 3. 4.]
```

Out [81]: Make this Notebook Trusted to load map: File -> Trust Notebook



In [82]:
```
plt.figure(figsize=(40,15))
sns.barplot(x='zipcode', y='price', data=train_df)
```

Out [82]: <AxesSubplot:xlabel='zipcode', ylabel='price'>



In [83]:
```python
for df1 in df_list:
    df1.drop(['latitude', 'longitude'], inplace=True, axis=1)
```

In [84]:
```python
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10768 entries, 5022 to 8809
Data columns (total 31 columns):
 #   Column                  Non-Null Count  Dtype   
---  ------                  --------------  -----   
 0   host_tenure             10768 non-null  int64   
 1   host_response_time      10768 non-null  object  
 2   host_response_rate      10768 non-null  object  
 3   host_acceptance_rate    10768 non-null  object  
 4   host_is_superhost       10768 non-null  object  
 5   host_listings_count     10768 non-null  float64 
 6   host_identity_verified  10768 non-null  object  
 7   zipcode                 10768 non-null  category
 8   property_type           10768 non-null  object  
 9   room_type               10768 non-null  object  
 10  accommodates            10768 non-null  int64   
 11  bedrooms                10768 non-null  float64 
 12  beds                    10768 non-null  float64 
 13  price                   10768 non-null  float64 
 14  minimum_nights          10768 non-null  int64   
 15  maximum_nights          10768 non-null  int64   
 16  has_availability        10768 non-null  object  
 17  number_of_reviews       10768 non-null  int64   
 18  number_of_reviews_l30d  10768 non-null  int64   
 19  review_scores_rating    10768 non-null  object  
 20  instant_bookable        10768 non-null  object  
 21  in_texas                10768 non-null  object  
 22  bathrooms               10768 non-null  float64 
 23  pool                    10768 non-null  bool    
 24  kitchen                 10768 non-null  bool    
 25  free parking            10768 non-null  bool    
 26  free street parking     10768 non-null  bool    
 27  hot tub                 10768 non-null  bool    
 28  washer                  10768 non-null  bool    
 29  workspace               10768 non-null  bool    
 30  gym                     10768 non-null  bool    
dtypes: bool(8), category(1), float64(5), int64(6), object(11)
memory usage: 2.2+ MB
```

In [85]:

```python
# train_df.to_csv('data/train_df')
# test_df.to_csv('data/test_df')
```

Model Preparation

To begin, we will need to adjust our datatypes. It is good to note that although currency is typically represented as a continuous data point, on Airbnb, price is always listed as an integer. Therefore, it will be represented as an integer in our models.

In [86]:
```
for df1 in df_list:
    df1['price'] = df1['price'].astype('int64')
    df1['pool'] = df1['pool'].astype('category')
    df1['kitchen'] = df1['kitchen'].astype('category')
    df1['free parking'] = df1['free parking'].astype('category')
    df1['free street parking'] = df1['free street parking'].astype('category')
    df1['hot tub'] = df1['hot tub'].astype('category')
    df1['washer'] = df1['washer'].astype('category')
    df1['workspace'] = df1['workspace'].astype('category')
    df1['gym'] = df1['gym'].astype('category')
```

Testing Multicollinearity

Testing our data for multicollinnearity is important. We cannot determine the individual effectiveness of predictors when our predictors are collinear.

In [87]:
```
y_train, y_test = train_df['price'], test_df['price']
x_train = train_df.drop('price', axis=1)
x_test = test_df.drop('price', axis=1)
```

In [88]:
```
x_train, x_valid, y_train, y_valid = tst(x_train, y_train, test_size=0.20, random_state=15444)
```

`bedrooms`, `beds`, `accomodates` and `bathrooms` are highly correlated to each other. This aligns with out beginning hypothessis that all 4 predictors would are related. The more people needed for housing, the more bedrooms, beds and bathrooms are desired. Multicollinearity is determinental when attempting to understand the effects of individual predictors on a model in regression analysis. However, it doesn't affect prediction. Regardless, we do not want to use predictors that provide redundant information. Therefore, I will drop `bedrooms`, `beds`, and `bathrooms`. When searching for airbnbs, it's most popular for tenants to search for properites that can lodge all members of the group. Therefore, it is best to measure the price with how many people the property can accommodate.

It is also good to note `number_of_reviews_l30d` and `number_of_reviews` are slightly correlated; however, the correlation is not bad so we will keep it in the model.

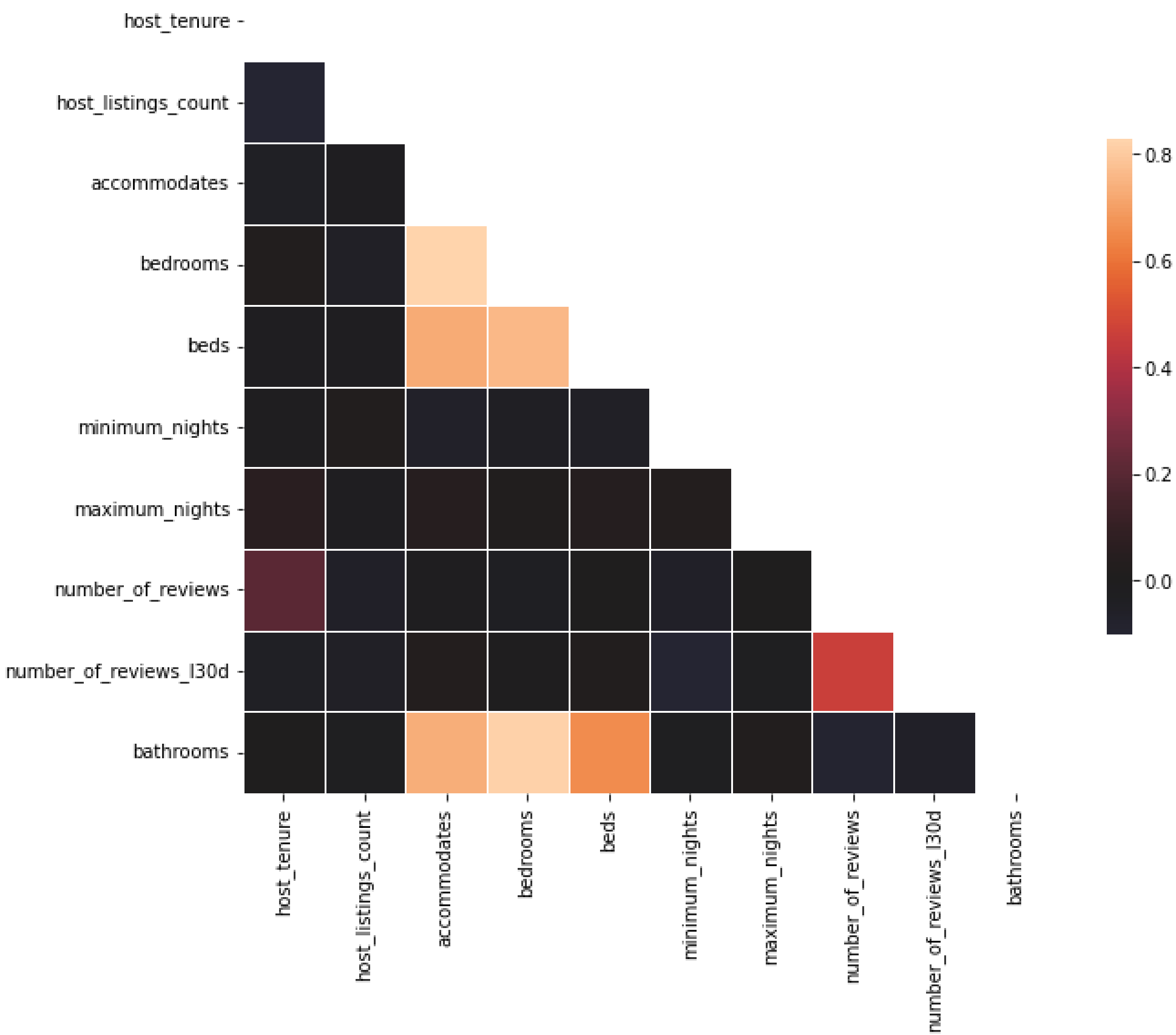
In [89]:
```
plt.figure(figsize=(10,10))
```

```
corr = x_train.corr()

mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask) ] = True

sns.heatmap(corr, mask=mask, center=0, square=True, linewidths=0.5, cbar_kws={"shrink": 0.5})
```

```
/var/folders/dv/14k38qsd37xgbxgkxr0c7fgm0000gn/T/ipykernel_4261/3239408036.py:4: DeprecationWarning: `np.bool
` is a deprecated alias for the builtin `bool`. To silence this warning, use `bool` by itself. Doing this wil
l not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#
deprecations
  mask = np.zeros_like(corr, dtype=np.bool)
```

Out[89]:

```
<AxesSubplot:>
```



In [90]:
```
x_train.drop(['bedrooms', 'beds', 'bathrooms'], axis=1, inplace=True)
x_valid.drop(['bedrooms', 'beds', 'bathrooms'], axis=1, inplace=True)
x_test.drop(['bedrooms', 'beds', 'bathrooms'], axis=1, inplace=True)
```

To prepare our model, we need to create dummy variables for our datasets. We will create two

In [91]:
```
df_concat = pd.concat([x_train, x_valid, x_test], axis=0)
```

```
df_concat['zipcode'] = df_concat['zipcode'].astype('category')
df_concat = pd.get_dummies(df_concat, drop_first=True)
df_concat.columns = df_concat.columns.str.replace(' ','_')


valid_end_indice = len(x_train) + len(x_valid)
x_train = df_concat.iloc[:len(x_train)]
x_valid = df_concat[len(x_train):valid_end_indice]
x_test = df_concat.iloc[valid_end_indice:]
```

In [92]:
```
x_test
```

Out[92]:

	host_tenure	host_listings_count	accommodates	minimum_nights	maximum_nights	number_of_reviews	number_of_reviews
8055	725	0.0	1	90	100	0	
2570	2976	1.0	1	1	1125	3	
8396	822	9.0	8	1	112	4	
618	2945	1.0	2	1	1125	1	
7983	3683	5.0	6	1	1125	74	
...	...	...	...	...	...	...	
5816	932	0.0	2	1	30	6	
9559	3336	27.0	4	2	1125	14	
9798	1014	0.0	6	30	365	0	
11276	2484	3.0	7	1	365	5	
1774	3100	2.0	8	2	1125	0	

1197 rows × 88 columns

In [93]:
```
scaler = StandardScaler()
scaler.fit(x_train)
x_train_std = pd.DataFrame(scaler.transform(x_train), columns= x_train.columns)
x_valid_std = pd.DataFrame(scaler.transform(x_valid), columns= x_train.columns)
x_test_std = pd.DataFrame(scaler.transform(x_test), columns= x_train.columns)
```

Modeling

I will evaluate the following models for use:

- Log of Y Linear Regression
- Ridge Regression
- XGBoost

Linear regression is a great model due to the the interpretability. However, a majority of cases do not meet the normality assumption. Our response variable is nonnegative. Would someone list their property on Airbnb to give you money? If so, please send me the listing link! Because of this, a normal linear regression model is not appropriate. A downfall is, multicollinearity may cause issues interpreting the affects of different predictors.

Ridge regression is uses regulation to punish collinear predictors. The model could help us find a smaller subset of predictors while handling our multicollinearity problem. However, the model can become problematic as it still uses least squares to measure MSE. Thus, it is very susceptible to outliers and can cause prediction issues. As we saw earlier, our data has contains a few outliers.

Last, XGBoost can make accurate predictions when our model is not linear. Collinear predictors do not affect model prediction and, because distance measures are not used, it is not susceptible to ouliers. KNN and SVM regression were both considered, however, both models would have been heavily influenced by the price outliers.

In [94]:
```
#create lists to track R-squared and MSE
fnl_r_squareds = []
fnl_mse = []
fnl_model_name = []
```

Model 1: Log Normal Linear Regression

As stated before, Linear Regression is the best model when attempting to interpret how different predictors affect a response variable. However, Linear Regression requires the relationship between X and Y to be linear. If not, some issues can arist. In our case, we know the relationship between X and Y will not be linear. Thus, we building a regression model on the log of the price.

When first looking at our model, we see we have numerous predictors that are insignificant. First, we must address any collinearity issues within the model.

In [95]:

```
def get_vifs(df):
    """
        Get a VIF for each predictor
        df: a dataframe containing all of the predictors.
    """
    vif = {}
    for i in range(len(df.columns)):
        vif[df.columns[i]] = variance_inflation_factor(df.values,i)
    return pd.DataFrame(vif.items(), columns=['Predictor', 'Variance Inflation Factor']).sort_values(
        by='Variance Inflation Factor', ascending=False)
```

In [96]:

```
import statsmodels.formula.api as smf
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn import linear_model
from sklearn.metrics import mean_squared_error
```

In [97]:

```
y_train_log = np.log(y_train)
y_valid_log = np.log(y_valid)
y_test_log = np.log(y_test)
model1 = sm.OLS(y_train_log, sm.add_constant(x_train)).fit()
print(model1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.598
Model:                            OLS   Adj. R-squared:                  0.594
Method:                 Least Squares   F-statistic:                     144.3
Date:                Thu, 16 Jun 2022   Prob (F-statistic):               0.00
Time:                        19:30:02   Log-Likelihood:                -7633.5
No. Observations:                8614   AIC:                         1.544e+04
Df Residuals:                    8525   BIC:                         1.607e+04
Df Model:                          88
Covariance Type:            nonrobust
===========================================================================================================
                                           coef    std err          t      P>|t|      [0.025      0.975]
-----------------------------------------------------------------------------------------------------------
const                                    4.9057      0.150     32.632      0.000       4.611       5.200
host_tenure                           4.176e-05   7.25e-06      5.764      0.000    2.76e-05     5.6e-05
host_listings_count                   2.617e-05   1.32e-05      1.986      0.047    3.35e-07     5.2e-05
accommodates                             0.1394      0.003     53.714      0.000       0.134       0.144
minimum_nights                          -0.0018      0.000     -8.098      0.000      -0.002      -0.001
maximum_nights                       -2.503e-05   1.32e-05     -1.893      0.058   -5.09e-05    8.84e-07
number_of_reviews                       -0.0004      0.000     -3.542      0.000      -0.001      -0.000
number_of_reviews_l30d                   0.0046      0.004      1.045      0.296      -0.004       0.013
host_response_time_unknown               0.0858      0.078      1.104      0.270      -0.067       0.238
host_response_time_within_a_day          0.0541      0.134      0.403      0.687      -0.209       0.317
host_response_time_within_a_few_hours   -0.0673      0.135     -0.498      0.618      -0.332       0.197
host_response_time_within_an_hour        0.0604      0.134      0.450      0.653      -0.203       0.323
host_response_rate_90-99                -0.0694      0.026     -2.646      0.008      -0.121      -0.018
host_response_rate_Less_than_90         -0.0977      0.030     -3.237      0.001      -0.157      -0.039
host_response_rate_unknown              -0.1290      0.154     -0.839      0.402      -0.431       0.173
host_acceptance_rate_70-80              -0.0428      0.035     -1.212      0.226      -0.112       0.026
host_acceptance_rate_81-90               0.0606      0.026      2.325      0.020       0.010       0.112
host_acceptance_rate_91-99              -0.0075      0.019     -0.397      0.691      -0.044       0.029
host_acceptance_rate_Less_than_70        0.0575      0.029      1.979      0.048       0.001       0.114
host_acceptance_rate_unknown            -0.1311      0.032     -4.054      0.000      -0.195      -0.068
host_is_superhost_t                     -0.0005      0.017     -0.031      0.975      -0.035       0.034
host_identity_verified_t                -0.0520      0.018     -2.868      0.004      -0.087      -0.016
zipcode_78702                           -0.2165      0.036     -6.024      0.000      -0.287      -0.146
zipcode_78703                           -0.0458      0.044     -1.050      0.294      -0.131       0.040
zipcode_78704                           -0.1918      0.034     -5.571      0.000      -0.259      -0.124
zipcode_78705                           -0.3940      0.045     -8.841      0.000      -0.481      -0.307
zipcode_78712                            1.4823      0.592      2.506      0.012       0.323       2.642
zipcode_78717                           -0.9585      0.097     -9.857      0.000      -1.149      -0.768
zipcode_78719                           -0.8211      0.227     -3.616      0.000      -1.266      -0.376
zipcode_78721                           -0.6003      0.053    -11.302      0.000      -0.704      -0.496
zipcode_78722                           -0.4871      0.060     -8.087      0.000      -0.605      -0.369
```

zipcode_78723	-0.5593	0.048	-11.734	0.000	-0.653	-0.466
zipcode_78724	-0.7581	0.074	-10.193	0.000	-0.904	-0.612
zipcode_78725	-0.7836	0.105	-7.448	0.000	-0.990	-0.577
zipcode_78726	-0.8186	0.190	-4.315	0.000	-1.190	-0.447
zipcode_78727	-0.8490	0.068	-12.433	0.000	-0.983	-0.715
zipcode_78728	-0.7375	0.072	-10.293	0.000	-0.878	-0.597
zipcode_78729	-0.6752	0.069	-9.781	0.000	-0.810	-0.540
zipcode_78730	-0.4334	0.118	-3.671	0.000	-0.665	-0.202
zipcode_78731	-0.4169	0.064	-6.509	0.000	-0.542	-0.291
zipcode_78732	-0.4480	0.092	-4.862	0.000	-0.629	-0.267
zipcode_78733	-0.2990	0.091	-3.286	0.001	-0.477	-0.121
zipcode_78734	-0.4647	0.053	-8.820	0.000	-0.568	-0.361
zipcode_78735	-0.5528	0.083	-6.642	0.000	-0.716	-0.390
zipcode_78736	-0.6065	0.090	-6.753	0.000	-0.783	-0.430
zipcode_78737	-0.6890	0.064	-10.812	0.000	-0.814	-0.564
zipcode_78738	-0.4709	0.091	-5.186	0.000	-0.649	-0.293
zipcode_78739	-0.4175	0.123	-3.400	0.001	-0.658	-0.177
zipcode_78741	-0.5022	0.040	-12.578	0.000	-0.580	-0.424
zipcode_78742	-0.8179	0.267	-3.068	0.002	-1.341	-0.295
zipcode_78744	-0.7642	0.056	-13.639	0.000	-0.874	-0.654
zipcode_78745	-0.6562	0.043	-15.167	0.000	-0.741	-0.571
zipcode_78746	-0.1829	0.051	-3.557	0.000	-0.284	-0.082
zipcode_78747	-0.8272	0.094	-8.814	0.000	-1.011	-0.643
zipcode_78748	-0.7748	0.062	-12.558	0.000	-0.896	-0.654
zipcode_78749	-0.6975	0.071	-9.758	0.000	-0.838	-0.557
zipcode_78750	-0.7732	0.105	-7.363	0.000	-0.979	-0.567
zipcode_78751	-0.6336	0.045	-14.090	0.000	-0.722	-0.545
zipcode_78752	-0.5112	0.059	-8.689	0.000	-0.627	-0.396
zipcode_78753	-0.7528	0.069	-10.849	0.000	-0.889	-0.617
zipcode_78754	-0.8265	0.066	-12.591	0.000	-0.955	-0.698
zipcode_78756	-0.5877	0.063	-9.301	0.000	-0.712	-0.464
zipcode_78757	-0.6803	0.057	-12.004	0.000	-0.791	-0.569
zipcode_78758	-0.6024	0.044	-13.576	0.000	-0.689	-0.515
zipcode_78759	-0.7433	0.068	-10.856	0.000	-0.878	-0.609
property_type_Hotel	0.7443	0.072	10.363	0.000	0.604	0.885
property_type_House	0.3581	0.024	14.617	0.000	0.310	0.406
property_type_Loft	0.2886	0.063	4.550	0.000	0.164	0.413
property_type_Other	-0.0887	0.051	-1.733	0.083	-0.189	0.012
property_type_Rental_Unit	0.0653	0.024	2.721	0.007	0.018	0.112
room_type_Private_room	-0.6435	0.021	-30.821	0.000	-0.684	-0.603
room_type_Shared_room	-1.1364	0.068	-16.795	0.000	-1.269	-1.004
has_availability_t	-0.0430	0.026	-1.660	0.097	-0.094	0.008
review_scores_rating_4.6-4.8	0.0204	0.030	0.690	0.490	-0.038	0.078
review_scores_rating_4.8-4.99	0.0420	0.029	1.446	0.148	-0.015	0.099
review_scores_rating_5	0.1347	0.027	4.965	0.000	0.081	0.188

```
review_scores_rating_Less_than_4         0.0829      0.043      1.926      0.054      -0.001       0.167
review_scores_rating_unknown             0.3154      0.028     11.264      0.000       0.261       0.370
instant_bookable_t                      -0.0307      0.016     -1.907      0.056      -0.062       0.001
in_texas_Yes                             0.0166      0.017      0.955      0.340      -0.017       0.051
in_texas_unknown                         0.0882      0.198      0.445      0.656      -0.300       0.477
pool_True                                0.0847      0.018      4.593      0.000       0.049       0.121
kitchen_True                            -0.0447      0.031     -1.460      0.144      -0.105       0.015
free_parking_True                       -0.0165      0.018     -0.939      0.348      -0.051       0.018
free_street_parking_True                -0.0219      0.016     -1.361      0.174      -0.054       0.010
hot_tub_True                             0.1970      0.026      7.521      0.000       0.146       0.248
washer_True                              0.0414      0.022      1.895      0.058      -0.001       0.084
workspace_True                          -0.0281      0.015     -1.932      0.053      -0.057       0.000
gym_True                                 0.0284      0.022      1.279      0.201      -0.015       0.072
==============================================================================
Omnibus:                      900.843   Durbin-Watson:                   2.008
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             6396.853
Skew:                           0.231   Prob(JB):                         0.00
Kurtosis:                       7.196   Cond. No.                     2.29e+05
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.29e+05. This might indicate that there are
strong multicollinearity or other numerical problems.
```

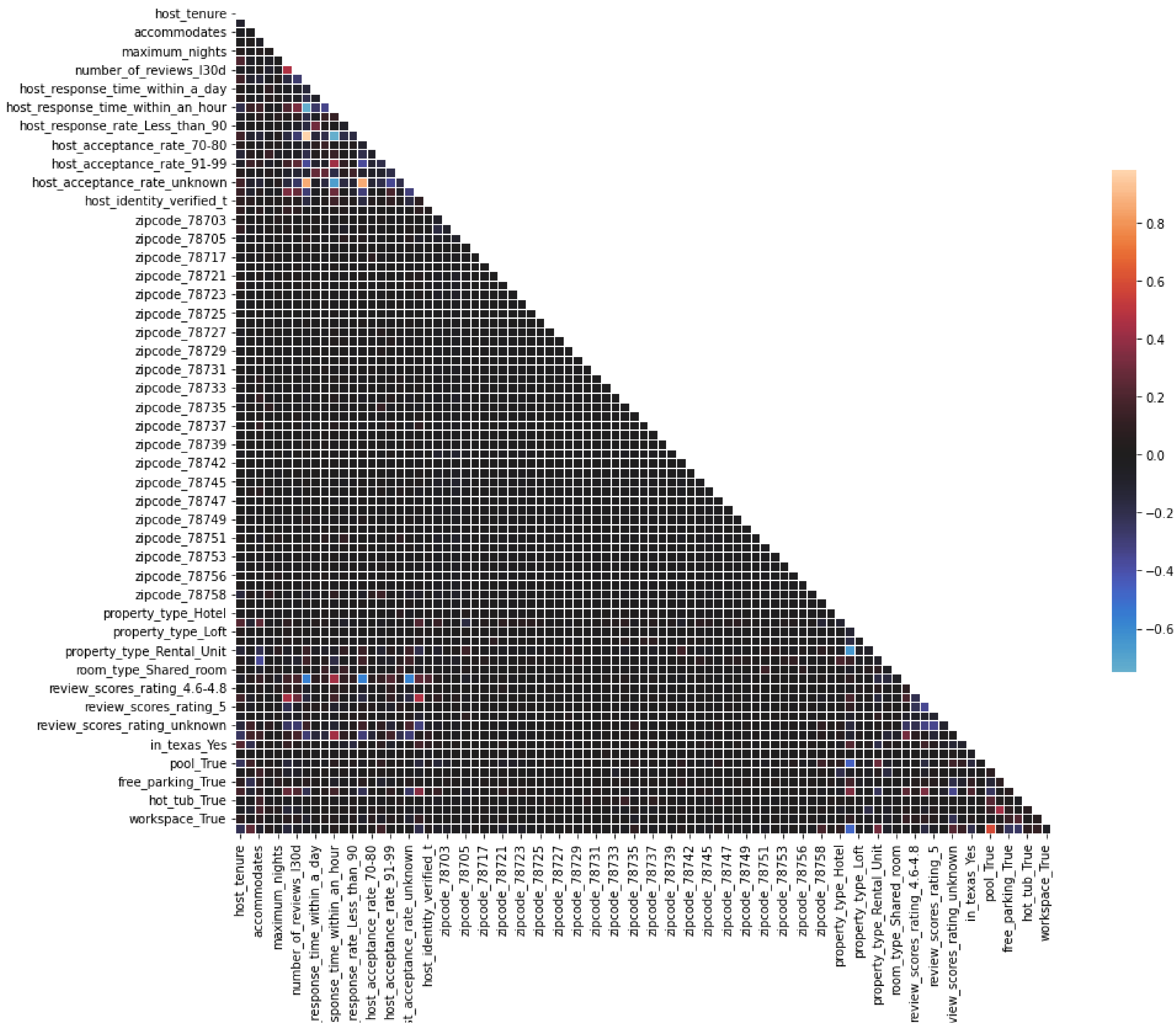
In [98]:

```python
plt.figure(figsize=(15,15))
corr = x_train_std.corr()

mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask) ] = True

sns.heatmap(corr, mask=mask, center=0, square=True, linewidths=0.5, cbar_kws={"shrink": 0.5})
```

```
/var/folders/dv/14k38qsd37xgbxgkxr0c7fgm0000gn/T/ipykernel_4261/2113875286.py:4: DeprecationWarning: `np.bool
` is a deprecated alias for the builtin `bool`. To silence this warning, use `bool` by itself. Doing this wil
l not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#
deprecations
  mask = np.zeros_like(corr, dtype=np.bool)
```

Out[98]:

```
<AxesSubplot:>
```



host_tenure
accommodates
maximum_nights
number_of_reviews_l30d
host_response_time_within_a_day
host_response_time_within_an_hour
host_response_rate_Less_than_90
host_acceptance_rate_70-80
host_acceptance_rate_91-99
host_acceptance_rate_unknown
host_identity_verified_t
zipcode_78703
zipcode_78705
zipcode_78717
zipcode_78721
zipcode_78723
zipcode_78725
zipcode_78727
zipcode_78729
zipcode_78731
zipcode_78733
zipcode_78735
zipcode_78737
zipcode_78739
zipcode_78742
zipcode_78745
zipcode_78747
zipcode_78749
zipcode_78751
zipcode_78753
zipcode_78756
zipcode_78758
property_type_Hotel
property_type_Loft
property_type_Rental_Unit
room_type_Shared_room
review_scores_rating_4.6-4.8
review_scores_rating_5
review_scores_rating_unknown
in_texas_Yes
pool_True
free_parking_True
hot_tub_True
workspace_True
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

None of the zipcode factors have collinearity with other variables. We can get a better look at the other predictors by creating another heatmap without the `zipcode` predictors.

In addition to creating a correlation heatmap I calculated a variance inflaction factor for each predictor (VIF). VIF's tell us how correlated a predictor is to all other predictors. Viewing the heatmap, we can see `host_response_rate_unknown` is heavily correlated with: `host_response_time_unknown` , `host_response_time_within_an_hour` , `host_acceptance_rate_unknown` , `host_is_superhost_t` , `has_availability_t` , and `instant_bookable_t` . This is confirmed by having a VIF factor of 64. Therfore, we dropped it from the dataset. Typically, a VIF above 10 is highly concerning and anyhting above 5 needs consideration to be dropped.

`pool_True` and `gym_true` were also higly correlated. Because `pool_true` is more correlated with `price` , I decided to keep it. Other predictors dropped due to collineaerity issues are:

- `host_response_time_within_a_few_hours`
- `host_response_time_within_an_hour`
- `host_response_time_unknown`
- `host_response_time_within_a_day`
- `has_availability_t`
- `kitchen_True`
- `property_type_House`
- `gym_True`
- `host_is_superhost_t`

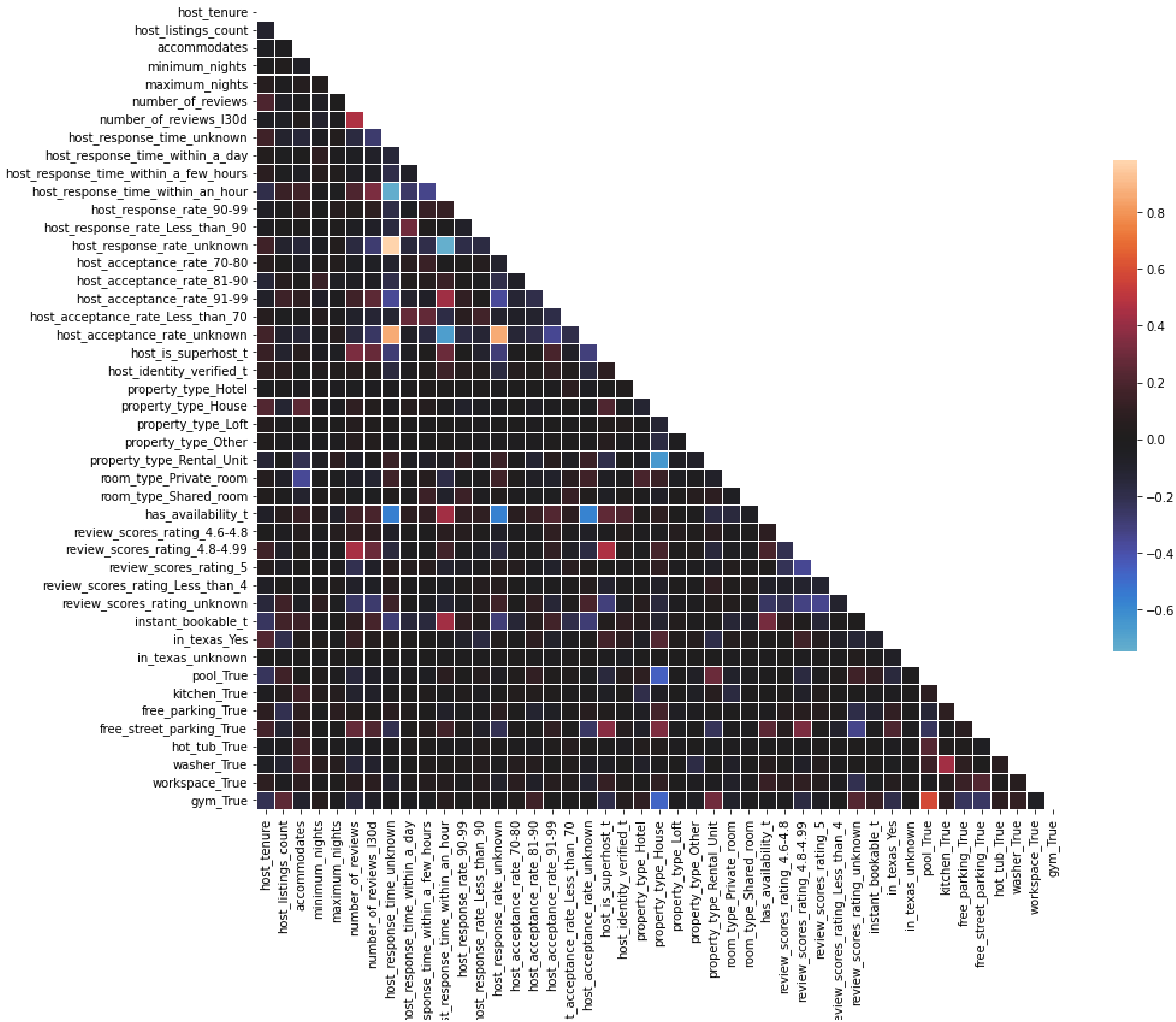
In [99]:

```
zipcodes = x_train.loc[:, x_train.columns.str.contains('zipcode')].columns
matrix_cols = x_train_std.drop(zipcodes, axis=1)

plt.figure(figsize=(15,15))
corr = matrix_cols.corr()

mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask) ] = True

sns.heatmap(corr, mask=mask, center=0, square=True, linewidths=0.5, cbar_kws={"shrink": 0.5})
```

```
/var/folders/dv/14k38qsd37xgbxgkxr0c7fgm0000gn/T/ipykernel_4261/1060428307.py:7: DeprecationWarning: `np.bool
` is a deprecated alias for the builtin `bool`. To silence this warning, use `bool` by itself. Doing this wil
l not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#
deprecations
  mask = np.zeros_like(corr, dtype=np.bool)
```

Out[99]:

```
<AxesSubplot:>
```


host_tenure
host_listings_count
accommodates
minimum_nights
maximum_nights
number_of_reviews
number_of_reviews_l30d
host_response_time_unknown
host_response_time_within_a_day
host_response_time_within_a_few_hours
host_response_time_within_an_hour
host_response_rate_90-99
host_response_rate_Less_than_90
host_response_rate_unknown
host_acceptance_rate_70-80
host_acceptance_rate_81-90
host_acceptance_rate_91-99
host_acceptance_rate_Less_than_70
host_acceptance_rate_unknown
host_is_superhost_t
host_identity_verified_t
property_type_Hotel
property_type_House
property_type_Loft
property_type_Other
property_type_Rental_Unit
room_type_Private_room
room_type_Shared_room
has_availability_t
review_scores_rating_4.6-4.8
review_scores_rating_4.8-4.99
review_scores_rating_5
review_scores_rating_Less_than_4
review_scores_rating_unknown
instant_bookable_t
in_texas_Yes
in_texas_unknown
pool_True
kitchen_True
free_parking_True
free_street_parking_True
hot_tub_True
washer_True
workspace_True
gym_True
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

In [100…
```
get_vifs(x_train).head(15)
```

Out[100]:

	Predictor	Variance Inflation Factor
13	host_response_rate_unknown	64.190319
10	host_response_time_within_an_hour	43.640061
7	host_response_time_unknown	43.356595
81	kitchen_True	21.188898
71	has_availability_t	14.065250
85	washer_True	10.110810
65	property_type_House	8.529728
0	host_tenure	7.440383
9	host_response_time_within_a_few_hours	7.013602
18	host_acceptance_rate_unknown	6.783710
20	host_identity_verified_t	6.779704
82	free_parking_True	6.003626
78	in_texas_Yes	5.756088
2	accommodates	5.245320
73	review_scores_rating_4.8-4.99	5.096264

In [101…
```
x_train['pool_True'].corr(np.log(y_train))
```

Out[101]:
```
0.07078968988925738
```

In [102…
```
x_train['gym_True'].corr(np.log(y_train))
```

Out[102]:
```
0.01952890606196009
```

In [103…
```
x_train.drop(
    ['host_response_time_within_a_few_hours', 'host_response_time_within_an_hour', 'host_response_time_unknown
    'host_response_time_within_a_day', 'has_availability_t', 'kitchen_True', 'property_type_House',
```

```
    'host_response_rate_unknown', 'gym_True', 'host_is_superhost_t'], axis=1, inplace=True)

x_valid.drop(
    ['host_response_time_within_a_few_hours', 'host_response_time_within_an_hour', 'host_response_time_unknown
    'host_response_time_within_a_day', 'has_availability_t', 'kitchen_True', 'property_type_House',
     'host_response_rate_unknown', 'gym_True', 'host_is_superhost_t'], axis=1, inplace=True)

x_test.drop(
    ['host_response_time_within_a_few_hours', 'host_response_time_within_an_hour', 'host_response_time_unknown
    'host_response_time_within_a_day', 'has_availability_t', 'kitchen_True', 'property_type_House',
     'host_response_rate_unknown', 'gym_True', 'host_is_superhost_t'], axis=1, inplace=True)
```

```
/var/folders/dv/14k38qsd37xgbxgkxr0c7fgm0000gn/T/ipykernel_4261/3972050056.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#r
eturning-a-view-versus-a-copy
  x_train.drop(
/var/folders/dv/14k38qsd37xgbxgkxr0c7fgm0000gn/T/ipykernel_4261/3972050056.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#r
eturning-a-view-versus-a-copy
  x_valid.drop(
/var/folders/dv/14k38qsd37xgbxgkxr0c7fgm0000gn/T/ipykernel_4261/3972050056.py:11: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#r
eturning-a-view-versus-a-copy
  x_test.drop(
```

Sequential Forward Selection

After removing collinear predictors, there are still insignificant features. In fact, there are A LOT of features. To find the best subset, I will use forward selection to find the appropriate subset. In short, with sequential forward selection we will find the best regression model for each k subset of predictors. For example, we will find the best simple regression model (k=1) where `price` is the response variable and the predictor that develops the highest coeffecient of determination (R^2). Lets call this predictor *bestie*. Next, we will find a predictor that, along with *bestie*, that develops the next highest highest R^2 when `price` is regressed against them. Since there are two predictors in this subset, this is the 2nd subset. We iterate until we reach the maxium length of predictors. Once finished, we plot the R-squared at each subset and determine which one gives use the most information gain.

By not including all predictors, we are able to find the most significant subset of them. Also, this method prevents us from overfitting.

In [104…
```
model1 = sm.OLS(y_train_log, sm.add_constant(x_train)).fit()
print(model1.summary())
```

OLS Regression Results

Dep. Variable:	price	R-squared:	0.586
Model:	OLS	Adj. R-squared:	0.583
Method:	Least Squares	F-statistic:	155.1
Date:	Thu, 16 Jun 2022	Prob (F-statistic):	0.00
Time:	19:30:08	Log-Likelihood:	-7760.5
No. Observations:	8614	AIC:	1.568e+04
Df Residuals:	8535	BIC:	1.624e+04
Df Model:	78		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	5.0001	0.052	96.088	0.000	4.898	5.102
host_tenure	5.072e-05	7.29e-06	6.954	0.000	3.64e-05	6.5e-05
host_listings_count	2.97e-05	1.32e-05	2.248	0.025	3.8e-06	5.56e-05
accommodates	0.1480	0.003	57.934	0.000	0.143	0.153
minimum_nights	-0.0020	0.000	-8.886	0.000	-0.002	-0.002
maximum_nights	-2.742e-05	1.33e-05	-2.056	0.040	-5.36e-05	-1.28e-06
number_of_reviews	-0.0004	0.000	-3.511	0.000	-0.001	-0.000
number_of_reviews_l30d	0.0085	0.004	1.914	0.056	-0.000	0.017
host_response_rate_90-99	-0.0811	0.026	-3.071	0.002	-0.133	-0.029
host_response_rate_Less_than_90	-0.0940	0.028	-3.303	0.001	-0.150	-0.038
host_acceptance_rate_70-80	-0.0588	0.035	-1.665	0.096	-0.128	0.010
host_acceptance_rate_81-90	0.0227	0.026	0.877	0.381	-0.028	0.074
host_acceptance_rate_91-99	0.0019	0.019	0.099	0.921	-0.035	0.039
host_acceptance_rate_Less_than_70	0.0180	0.028	0.651	0.515	-0.036	0.072
host_acceptance_rate_unknown	-0.1857	0.022	-8.612	0.000	-0.228	-0.143
host_identity_verified_t	-0.0627	0.018	-3.445	0.001	-0.098	-0.027
zipcode_78702	-0.1094	0.035	-3.105	0.002	-0.179	-0.040
zipcode_78703	0.0232	0.043	0.538	0.591	-0.061	0.108
zipcode_78704	-0.1036	0.034	-3.071	0.002	-0.170	-0.037
zipcode_78705	-0.3737	0.044	-8.462	0.000	-0.460	-0.287
zipcode_78712	1.6068	0.600	2.679	0.007	0.431	2.783
zipcode_78717	-0.7915	0.098	-8.093	0.000	-0.983	-0.600
zipcode_78719	-0.6830	0.230	-2.972	0.003	-1.134	-0.233
zipcode_78721	-0.4848	0.053	-9.193	0.000	-0.588	-0.381
zipcode_78722	-0.3601	0.060	-5.983	0.000	-0.478	-0.242
zipcode_78723	-0.4362	0.047	-9.278	0.000	-0.528	-0.344
zipcode_78724	-0.6104	0.074	-8.213	0.000	-0.756	-0.465
zipcode_78725	-0.6307	0.106	-5.958	0.000	-0.838	-0.423
zipcode_78726	-0.6958	0.192	-3.623	0.000	-1.072	-0.319
zipcode_78727	-0.7392	0.069	-10.771	0.000	-0.874	-0.605
zipcode_78728	-0.6260	0.072	-8.688	0.000	-0.767	-0.485

zipcode_78729	-0.5664	0.069	-8.184	0.000	-0.702	-0.431
zipcode_78730	-0.3426	0.119	-2.868	0.004	-0.577	-0.108
zipcode_78731	-0.3267	0.064	-5.096	0.000	-0.452	-0.201
zipcode_78732	-0.3027	0.093	-3.263	0.001	-0.484	-0.121
zipcode_78733	-0.1158	0.091	-1.275	0.202	-0.294	0.062
zipcode_78734	-0.3329	0.052	-6.383	0.000	-0.435	-0.231
zipcode_78735	-0.4936	0.084	-5.866	0.000	-0.659	-0.329
zipcode_78736	-0.4777	0.090	-5.289	0.000	-0.655	-0.301
zipcode_78737	-0.5385	0.063	-8.508	0.000	-0.663	-0.414
zipcode_78738	-0.3649	0.091	-3.993	0.000	-0.544	-0.186
zipcode_78739	-0.2700	0.124	-2.184	0.029	-0.512	-0.028
zipcode_78741	-0.4153	0.040	-10.496	0.000	-0.493	-0.338
zipcode_78742	-0.6879	0.270	-2.546	0.011	-1.217	-0.158
zipcode_78744	-0.6274	0.056	-11.274	0.000	-0.736	-0.518
zipcode_78745	-0.5516	0.043	-12.894	0.000	-0.636	-0.468
zipcode_78746	-0.0662	0.051	-1.292	0.196	-0.167	0.034
zipcode_78747	-0.6993	0.094	-7.416	0.000	-0.884	-0.514
zipcode_78748	-0.6270	0.061	-10.200	0.000	-0.748	-0.507
zipcode_78749	-0.5604	0.071	-7.838	0.000	-0.701	-0.420
zipcode_78750	-0.6625	0.106	-6.263	0.000	-0.870	-0.455
zipcode_78751	-0.5539	0.045	-12.438	0.000	-0.641	-0.467
zipcode_78752	-0.4102	0.059	-6.964	0.000	-0.526	-0.295
zipcode_78753	-0.6313	0.070	-9.083	0.000	-0.768	-0.495
zipcode_78754	-0.6912	0.066	-10.519	0.000	-0.820	-0.562
zipcode_78756	-0.5026	0.063	-7.928	0.000	-0.627	-0.378
zipcode_78757	-0.5892	0.057	-10.402	0.000	-0.700	-0.478
zipcode_78758	-0.5121	0.044	-11.527	0.000	-0.599	-0.425
zipcode_78759	-0.6441	0.069	-9.356	0.000	-0.779	-0.509
property_type_Hotel	0.5690	0.071	8.064	0.000	0.431	0.707
property_type_Loft	0.0665	0.062	1.069	0.285	-0.055	0.188
property_type_Other	-0.3963	0.047	-8.401	0.000	-0.489	-0.304
property_type_Rental_Unit	-0.1470	0.019	-7.808	0.000	-0.184	-0.110
room_type_Private_room	-0.5977	0.021	-28.803	0.000	-0.638	-0.557
room_type_Shared_room	-1.1517	0.068	-16.905	0.000	-1.285	-1.018
review_scores_rating_4.6-4.8	0.0266	0.030	0.888	0.375	-0.032	0.085
review_scores_rating_4.8-4.99	0.0537	0.028	1.899	0.058	-0.002	0.109
review_scores_rating_5	0.1502	0.027	5.527	0.000	0.097	0.204
review_scores_rating_Less_than_4	0.0793	0.044	1.821	0.069	-0.006	0.165
review_scores_rating_unknown	0.3292	0.028	11.712	0.000	0.274	0.384
instant_bookable_t	-0.0368	0.016	-2.361	0.018	-0.067	-0.006
in_texas_Yes	0.0192	0.018	1.092	0.275	-0.015	0.054
in_texas_unknown	0.0575	0.201	0.286	0.775	-0.336	0.451
pool_True	0.0218	0.017	1.307	0.191	-0.011	0.054
free_parking_True	-0.0032	0.018	-0.182	0.855	-0.038	0.031
free_street_parking_True	0.0073	0.016	0.455	0.649	-0.024	0.039

```
hot_tub_True                                 0.2251      0.026      8.505      0.000       0.173       0.277
washer_True                                  0.0058      0.020      0.288      0.774      -0.034       0.046
workspace_True                              -0.0214      0.015     -1.454      0.146      -0.050       0.007
==============================================================================
Omnibus:                      833.632   Durbin-Watson:                   2.008
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             5412.644
Skew:                           0.214   Prob(JB):                         0.00
Kurtosis:                       6.860   Cond. No.                     2.29e+05
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.29e+05. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In [105…

```python
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs
from sklearn.linear_model import LinearRegression
```

We can see R-squared begin to slow down at a subset of 18 features. As predicted, `accommodates` contributes to the priving of an Airbnb. As stated before, more area is required to house more people. Therefore, larger groups will pay more. Understandibly, the second and third important features are `room_type_Private_Room` and `room_type_Shared_room`. Typically, rooms are much smaller than enitre buildings; therefore, hosts will charge more.

In [106…

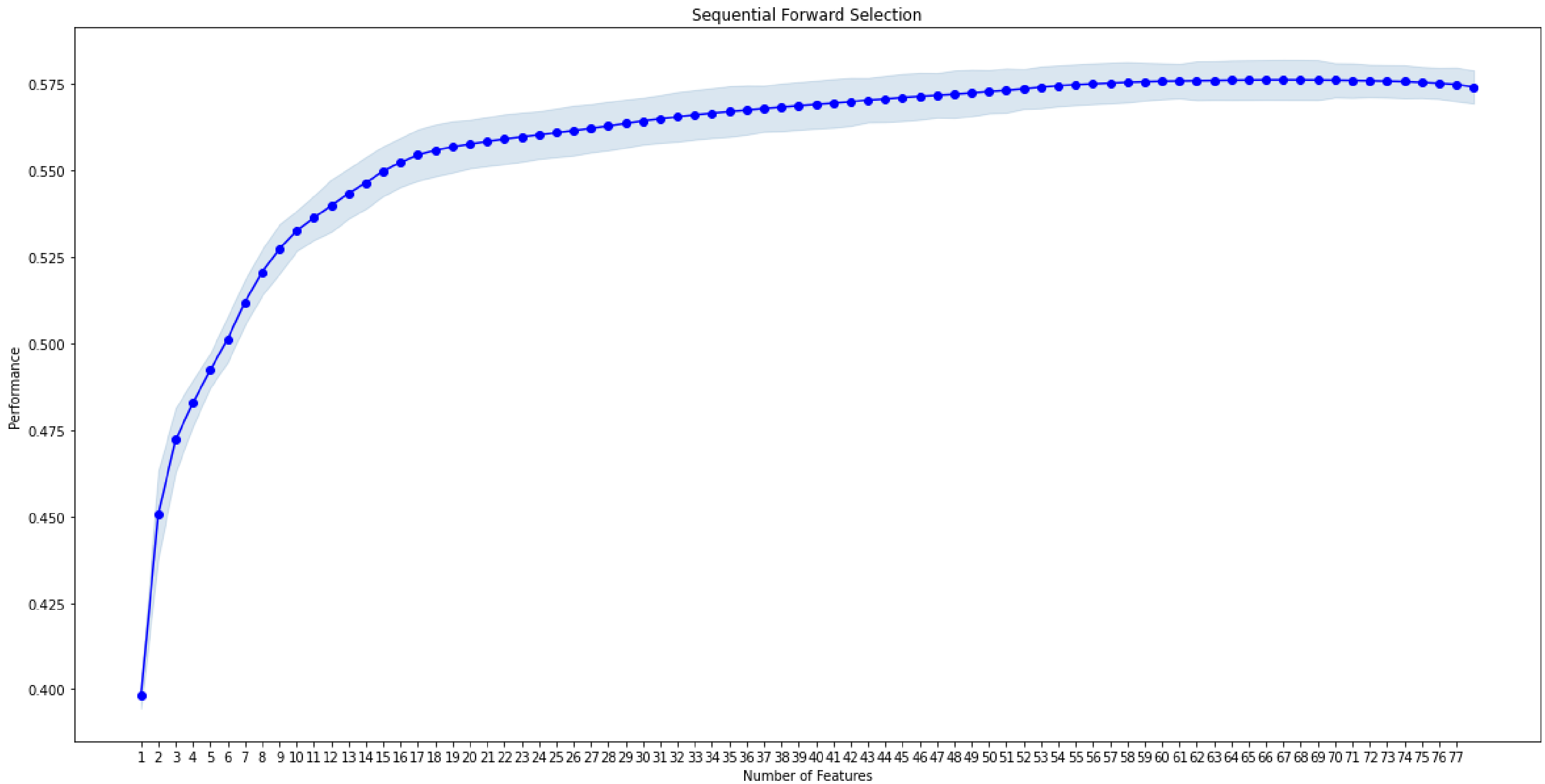
```python
# See what the best number of estimators are
lr = LinearRegression()
sfs = SFS(
    lr,
    k_features=x_train.shape[1],
    forward=True,
    cv=3
)
sfs.fit(x_train, y_train_log)


plot_sfs(sfs.get_metric_dict(), kind='std_err', figsize=(20,10))
plt.title('Sequential Forward Selection')
plt.xticks(np.arange(1, x_train.shape[1], ))
```

```
Out[106]: ([<matplotlib.axis.XTick at 0x7f7bcf50a070>,
  <matplotlib.axis.XTick at 0x7f7bcf50aeb0>,
  <matplotlib.axis.XTick at 0x7f7bcf4fa550>,
  <matplotlib.axis.XTick at 0x7f7bcf545100>,
  <matplotlib.axis.XTick at 0x7f7bcf545850>,
  <matplotlib.axis.XTick at 0x7f7bcf54c070>,
  <matplotlib.axis.XTick at 0x7f7bcf545940>,
  <matplotlib.axis.XTick at 0x7f7bcf54c0d0>,
  <matplotlib.axis.XTick at 0x7f7bcf54cd60>,
  <matplotlib.axis.XTick at 0x7f7bcf5524f0>,
  <matplotlib.axis.XTick at 0x7f7bcf552c40>,
  <matplotlib.axis.XTick at 0x7f7bcf5563d0>,
  <matplotlib.axis.XTick at 0x7f7bcf552d00>,
  <matplotlib.axis.XTick at 0x7f7bcf54ccd0>,
  <matplotlib.axis.XTick at 0x7f7bcf556ca0>,
  <matplotlib.axis.XTick at 0x7f7bcf560100>,
  <matplotlib.axis.XTick at 0x7f7bcf560850>,
  <matplotlib.axis.XTick at 0x7f7bcf564070>,
  <matplotlib.axis.XTick at 0x7f7bcf564730>,
  <matplotlib.axis.XTick at 0x7f7bcf560940>,
  <matplotlib.axis.XTick at 0x7f7bcf556760>,
  <matplotlib.axis.XTick at 0x7f7bcf5643a0>,
  <matplotlib.axis.XTick at 0x7f7bcfa223d0>,
  <matplotlib.axis.XTick at 0x7f7bcfa22b20>,
  <matplotlib.axis.XTick at 0x7f7bcfa272b0>,
  <matplotlib.axis.XTick at 0x7f7bcfa27a00>,
  <matplotlib.axis.XTick at 0x7f7bcfa22880>,
  <matplotlib.axis.XTick at 0x7f7bcf5454c0>,
  <matplotlib.axis.XTick at 0x7f7bcfa273a0>,
  <matplotlib.axis.XTick at 0x7f7bcfa2e580>,
  <matplotlib.axis.XTick at 0x7f7bcfa2ecd0>,
  <matplotlib.axis.XTick at 0x7f7bcfa34460>,
  <matplotlib.axis.XTick at 0x7f7bcfa34bb0>,
  <matplotlib.axis.XTick at 0x7f7bcfa2e670>,
  <matplotlib.axis.XTick at 0x7f7bcfa27a30>,
  <matplotlib.axis.XTick at 0x7f7bcf54c4f0>,
  <matplotlib.axis.XTick at 0x7f7bcded4220>,
  <matplotlib.axis.XTick at 0x7f7bcfa34c70>,
  <matplotlib.axis.XTick at 0x7f7bcfa3e190>,
  <matplotlib.axis.XTick at 0x7f7bcfa34490>,
  <matplotlib.axis.XTick at 0x7f7bcf4fadf0>,
  <matplotlib.axis.XTick at 0x7f7bcfa3ea30>,
  <matplotlib.axis.XTick at 0x7f7bcfa3ed60>,
  <matplotlib.axis.XTick at 0x7f7bcfa444f0>,
  <matplotlib.axis.XTick at 0x7f7bcfa44c40>,
```

```
 <matplotlib.axis.XTick at 0x7f7bcfa4a3d0>,
 <matplotlib.axis.XTick at 0x7f7bcfa448b0>,
 <matplotlib.axis.XTick at 0x7f7bcfa2e4f0>,
 <matplotlib.axis.XTick at 0x7f7bcfa4a340>,
 <matplotlib.axis.XTick at 0x7f7bcfa52070>,
 <matplotlib.axis.XTick at 0x7f7bcfa527c0>,
 <matplotlib.axis.XTick at 0x7f7bcfa58070>,
 <matplotlib.axis.XTick at 0x7f7bcfa586a0>,
 <matplotlib.axis.XTick at 0x7f7bcfa52a90>,
 <matplotlib.axis.XTick at 0x7f7bcfa2ec40>,
 <matplotlib.axis.XTick at 0x7f7bcfa58310>,
 <matplotlib.axis.XTick at 0x7f7bcfa62220>,
 <matplotlib.axis.XTick at 0x7f7bcfa62970>,
 <matplotlib.axis.XTick at 0x7f7bcfa66100>,
 <matplotlib.axis.XTick at 0x7f7bcfa66850>,
 <matplotlib.axis.XTick at 0x7f7bcfa62610>,
 <matplotlib.axis.XTick at 0x7f7bcfa520d0>,
 <matplotlib.axis.XTick at 0x7f7bcfa66c40>,
 <matplotlib.axis.XTick at 0x7f7bcfa703d0>,
 <matplotlib.axis.XTick at 0x7f7bcfa70b20>,
 <matplotlib.axis.XTick at 0x7f7bcfa762b0>,
 <matplotlib.axis.XTick at 0x7f7bcfa70dc0>,
 <matplotlib.axis.XTick at 0x7f7bcfa66c10>,
 <matplotlib.axis.XTick at 0x7f7bcfa769a0>,
 <matplotlib.axis.XTick at 0x7f7bcfa76e80>,
 <matplotlib.axis.XTick at 0x7f7bcfa7c610>,
 <matplotlib.axis.XTick at 0x7f7bcfa7cd60>,
 <matplotlib.axis.XTick at 0x7f7bcfa834f0>,
 <matplotlib.axis.XTick at 0x7f7bcfa7c8b0>,
 <matplotlib.axis.XTick at 0x7f7bcfa58220>,
 <matplotlib.axis.XTick at 0x7f7bcfa83250>,
 <matplotlib.axis.XTick at 0x7f7bcfa8b190>],
[Text(1, 0, '1'),
 Text(2, 0, '2'),
 Text(3, 0, '3'),
 Text(4, 0, '4'),
 Text(5, 0, '5'),
 Text(6, 0, '6'),
 Text(7, 0, '7'),
 Text(8, 0, '8'),
 Text(9, 0, '9'),
 Text(10, 0, '10'),
 Text(11, 0, '11'),
 Text(12, 0, '12'),
 Text(13, 0, '13'),
```

```
Text(14, 0, '14'),
Text(15, 0, '15'),
Text(16, 0, '16'),
Text(17, 0, '17'),
Text(18, 0, '18'),
Text(19, 0, '19'),
Text(20, 0, '20'),
Text(21, 0, '21'),
Text(22, 0, '22'),
Text(23, 0, '23'),
Text(24, 0, '24'),
Text(25, 0, '25'),
Text(26, 0, '26'),
Text(27, 0, '27'),
Text(28, 0, '28'),
Text(29, 0, '29'),
Text(30, 0, '30'),
Text(31, 0, '31'),
Text(32, 0, '32'),
Text(33, 0, '33'),
Text(34, 0, '34'),
Text(35, 0, '35'),
Text(36, 0, '36'),
Text(37, 0, '37'),
Text(38, 0, '38'),
Text(39, 0, '39'),
Text(40, 0, '40'),
Text(41, 0, '41'),
Text(42, 0, '42'),
Text(43, 0, '43'),
Text(44, 0, '44'),
Text(45, 0, '45'),
Text(46, 0, '46'),
Text(47, 0, '47'),
Text(48, 0, '48'),
Text(49, 0, '49'),
Text(50, 0, '50'),
Text(51, 0, '51'),
Text(52, 0, '52'),
Text(53, 0, '53'),
Text(54, 0, '54'),
Text(55, 0, '55'),
Text(56, 0, '56'),
Text(57, 0, '57'),
Text(58, 0, '58'),
```

```
Text(59, 0, '59'),
Text(60, 0, '60'),
Text(61, 0, '61'),
Text(62, 0, '62'),
Text(63, 0, '63'),
Text(64, 0, '64'),
Text(65, 0, '65'),
Text(66, 0, '66'),
Text(67, 0, '67'),
Text(68, 0, '68'),
Text(69, 0, '69'),
Text(70, 0, '70'),
Text(71, 0, '71'),
Text(72, 0, '72'),
Text(73, 0, '73'),
Text(74, 0, '74'),
Text(75, 0, '75'),
Text(76, 0, '76'),
Text(77, 0, '77')])
```



In [107…
```
print('The feature that contributes the most to the price of an Airbnb is: ', list(sfs.subsets_[1]['feature_na
print('The two features that contribute the most to the price of an Airbnb are: ', list(sfs.subsets_[2]['featu
print('The three features that contribute the most to the price of an Airbnb are: ', list(sfs.subsets_[3]['fea
```

```
The feature that contributes the most to the price of an Airbnb is:  accommodates
The two features that contribute the most to the price of an Airbnb are:  ['accommodates', 'room_type_Private
_room']
The three features that contribute the most to the price of an Airbnb are:  ['accommodates', 'room_type_Priva
te_room', 'room_type_Shared_room']
```

Our final model's R-squared is 0.56 with a MSE of 0.378318. For, the MSE is a benchmark we can use to compare to other models. It's essentially the the squared difference between the predicted value and the real value of the data point. Also, our goal of this model was to make it interpretable. We can use the coeffecients to state how a predictor changes the price of an Airbnb. We can do this by transforming the predictor. Let X be predictor of the interested predictor. Then a 1 change increase in X increases the `price` by:

$$100 * (e^{Xcoefficient} - 1)$$

For example, increasing `accommodates` by 1 will increase an Airbnb's price by 15.8%. This can be shown by:

$$100 * (e^{0.1467} - 1) \approx 15.8\%$$

Finally, when testing our model on the validation data we get a MSE of 0.393 and an R^2 of 0.527.

In [108…

```
linear_features = list(sfs.subsets_[18]['feature_names'])
temp = x_train[linear_features]
model1 = sm.OLS(y_train_log, sm.add_constant(temp)).fit()
print(model1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.560
Model:                            OLS   Adj. R-squared:                  0.559
Method:                 Least Squares   F-statistic:                     607.6
Date:                Thu, 16 Jun 2022   Prob (F-statistic):               0.00
Time:                        19:30:48   Log-Likelihood:                -8026.7
No. Observations:                8614   AIC:                         1.609e+04
Df Residuals:                    8595   BIC:                         1.623e+04
Df Model:                          18
Covariance Type:            nonrobust
================================================================================================
                                  coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------------------------
const                           4.5704      0.028    166.174      0.000       4.516       4.624
host_tenure                  5.202e-05    6.8e-06      7.653      0.000    3.87e-05    6.53e-05
host_listings_count          6.612e-05   1.26e-05      5.233      0.000    4.14e-05    9.09e-05
accommodates                    0.1467      0.002     58.826      0.000       0.142       0.152
minimum_nights                 -0.0023      0.000     -9.855      0.000      -0.003      -0.002
host_acceptance_rate_unknown   -0.1469      0.016     -8.975      0.000      -0.179      -0.115
zipcode_78702                   0.3286      0.021     15.772      0.000       0.288       0.369
zipcode_78703                   0.4627      0.033     14.200      0.000       0.399       0.527
zipcode_78704                   0.3423      0.019     17.901      0.000       0.305       0.380
zipcode_78746                   0.3785      0.044      8.517      0.000       0.291       0.466
property_type_Hotel             0.8375      0.070     12.009      0.000       0.701       0.974
property_type_Other            -0.4156      0.047     -8.834      0.000      -0.508      -0.323
property_type_Rental_Unit      -0.1476      0.017     -8.738      0.000      -0.181      -0.114
room_type_Private_room         -0.6654      0.020    -32.595      0.000      -0.705      -0.625
room_type_Shared_room          -1.1997      0.068    -17.744      0.000      -1.332      -1.067
review_scores_rating_5          0.1363      0.016      8.484      0.000       0.105       0.168
review_scores_rating_unknown    0.3164      0.018     17.851      0.000       0.282       0.351
free_parking_True              -0.0957      0.017     -5.660      0.000      -0.129      -0.063
hot_tub_True                    0.2913      0.026     11.301      0.000       0.241       0.342
==============================================================================
Omnibus:                      796.617   Durbin-Watson:                   2.004
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             4468.789
Skew:                           0.258   Prob(JB):                         0.00
Kurtosis:                       6.491   Cond. No.                     2.53e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.53e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In [109…
```
temp_valid = sm.add_constant(x_valid[linear_features])
lin_reg_pred = model1.predict(temp_valid)
r_squared_lnr = r2_score(y_valid_log, lin_reg_pred)
mse_lnr = mean_squared_error(y_valid_log, lin_reg_pred)
fnl_r_squareds.append(r_squared_lnr)
fnl_mse.append(mse_lnr)
fnl_model_name.append('Linear Regression')
print('The MSE of the validation data is: ', mse_lnr)
print('The R^2 of the validation data is: ', r_squared_lnr)
```

```
The MSE of the validation data is:  0.3932776361013266
The R^2 of the validation data is:  0.527486096276466
```

Ridge Regression

Next, ridge regression is a model which decreases the coefficents of less important features. Essentially, the model conducts its own feature selection by eliminating features that are not contribution to the prediction. Thus, it is a great regression model to run on our dataset. We can control how much we want to decrease the value of the less important features by adjusting the parameter α. α has a range of 0 to 1; the larger α is, the more it punishes unimportant predictors. We'll fit a ridge regression model on various values of α and compare their R^2 and MSE values. It is important to note that I used standardized x-values for this model. As stated before, Ridge Regression uses a strategy that punishes large coefficients; therofre, scaling or predictors allows each predictor to be fairly evaluated.

R^2 is approximately 0.58 until α > 1000; beyond 1000, R^2 begins to decrease dramatically. On the other hand, MSE increases dramatically when α > 1000. Thus, 1000 is the best value for α. On the bottom plot we can see how Ridge Regresion works. In summary, shrinks the unimportant predictors. The faster a predictor converges to 0, the less important it is. We can see `accommodates` and `room_type_Private_room` have large weights. Also, compared to other predictors, they fail to converge close to 0. Thus, the model suggests these predictors are the most important in predicting the price in Airbnb.

In [110…
```
def get_ridge_coefs(cols, ridge_model_ins, sortby ='coef', ascend=False):
    """
        return sorted ridge coefficients and their associated features

        cols: names of the features
        ridge_model_ins : instance of a ridge regression model
        ascending: If True, will return dataframe sorted with the smallest value first
        sortby: How to sort the dataframe
                - 'coef': sorts dataframe by features coefficient value
                - 'magnitude': sorts by the absolute value of the coefficient
```

```
    """
    #get indices of sorted coefficients, smallest to largest
    coefs_temp = ridge_model_ins.coef_
    coefs_inds_sorted = np.argsort(coefs_temp)
    #sort features and coefficents from smallest to largest
    coefs_sorted = coefs_temp[coefs_inds_sorted]
    features_sorted = cols[coefs_inds_sorted]

    coefficients = pd.DataFrame(zip(features_sorted, coefs_sorted), columns=['feature', 'coef'])
    coefficients['magnitude'] = abs(coefficients['coef'])

    return coefficients.sort_values(by=sortby, ascending=ascend)
```

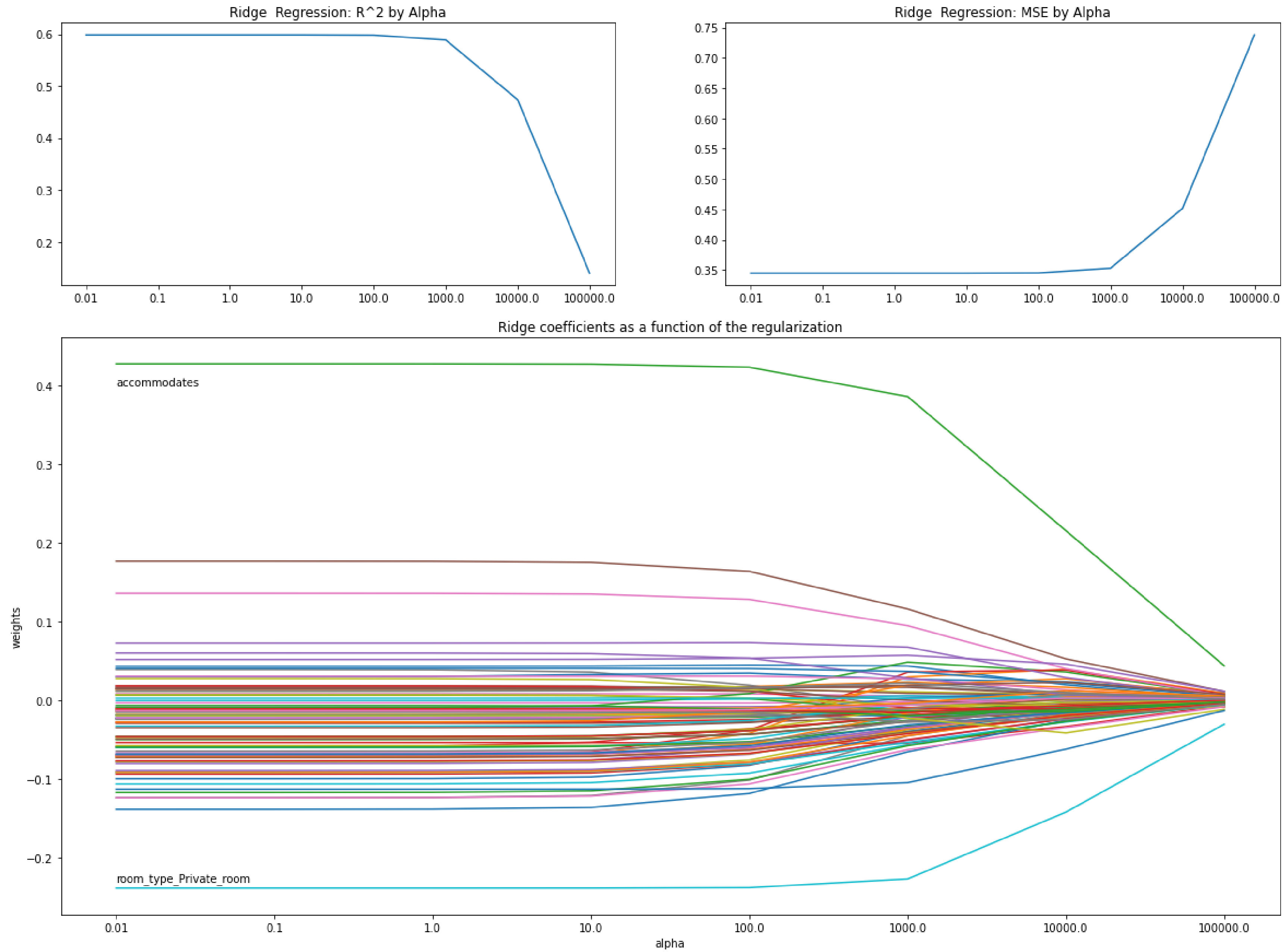
In [111…

```
alpha_list = []
r_squared = []
mse = []
coefs = []
max_coef = None
min_coef = None
for i in range (-2,6):
    alpha = 10**i
    rm = linear_model.Ridge(alpha=alpha)
    ridge_model = rm.fit(x_train_std, y_train_log)
    preds_ridge = ridge_model.predict(x_train_std)

    #get max, and min
    if i == 5:
        feature_df = get_ridge_coefs(x_train_std.columns, ridge_model )
        #get min and max features
        min_coef1 = feature_df['feature'].iloc[-1]
        max_coef1 = feature_df['feature'].iloc[0]


    alpha_list.append(alpha)
    r_squared.append(ridge_model.score(x_train_std, y_train_log))
    mse.append(mean_squared_error(y_train_log, preds_ridge))
    coefs.append(ridge_model.coef_)

#Dataframe of results for alpha
results = pd.DataFrame(zip(alpha_list, r_squared, mse), columns=['Alpha', 'R^2', 'MSE'])
results['Alpha'] = results['Alpha'].astype('str')


fig = plt.figure(figsize=(20,15))
```

```python
grid = fig.add_gridspec(3, 2)
ax0 = fig.add_subplot(grid[0, 0])
ax1 = fig.add_subplot(grid[0, 1])
ax2 = fig.add_subplot(grid[1:, :])

ax0.plot(results["Alpha"], results['R^2'])
ax0.set_title("Ridge  Regression: R^2 by Alpha")

ax1.plot(results["Alpha"], results['MSE'])
ax1.set_title("Ridge  Regression: MSE by Alpha")

ax2 = plt.gca()
ax2.plot(results["Alpha"], coefs)
ax2.set_xlabel("alpha")
ax2.set_ylabel("weights")
print(min_coef1)
ax2.annotate(text=max_coef1, xy=(0,0.4))
ax2.annotate(text=min_coef1, xy=(0, -0.23))
ax2.set_title("Ridge coefficients as a function of the regularization")
```

```
room_type_Private_room
```

Out[111]:
```
Text(0.5, 1.0, 'Ridge coefficients as a function of the regularization')
```



For our final ridge regression model α = 1000. This results in an MSE of 0.3638 and a R^2 of 0.5628. This is an improvement from our Linear Regression model.

In [112…
```
rm = linear_model.Ridge(alpha=1000)
```

```
ridge_model = rm.fit(x_train_std, y_train_log)
preds_ridge = ridge_model.predict(x_valid_std)

final_ridge_r_squared = ridge_model.score(x_valid_std, y_valid_log)
final_mse = mean_squared_error(y_valid_log, preds_ridge)

fnl_r_squareds.append(final_ridge_r_squared)
fnl_mse.append(final_mse)
fnl_model_name.append('Ridge Regression')

print('The MSE of our final ridge model is ', final_mse)
print('The R^2 of our final model is ', final_ridge_r_squared)
ridge_coefs = get_ridge_coefs(x_train_std.columns, ridge_model, sortby='magnitude')
ridge_coefs.head(6)
```

```
The MSE of our final ridge model is  0.36385915042551825
The R^2 of our final model is  0.562831720416481
```

Out[112]:

	feature	coef	magnitude
87	accommodates	0.385814	0.385814
0	room_type_Private_room	-0.227003	0.227003
86	property_type_House	0.116002	0.116002
1	room_type_Shared_room	-0.104623	0.104623
85	review_scores_rating_unknown	0.094790	0.094790
84	property_type_Hotel	0.067476	0.067476

Extreme Gradient Boosted Decision Trees (XGBoost)

Decision Trees are known as one of the most interpretable models. By starting at the root node, one can travel down stems asking themself whether or not they meet a certain criteria. Once you reach a leaf, you know what your output should be. XGBoost follows a similar pattern except it creates an ensemble of trees while using 'boosting'to create the best model. 'Boosting' refers to a sequential technique in which an ensemble model takes a bunch of 'weak learners' and attempts to improve its predecessor. The model I chose, XGBoost using Regression, attempts to accomplish the task by reducing the difference between the prediction and the true value. For our task, this is the residuals of error.

Overall, XGBoost is great for our trask as it is easy to interpret, has string prediction power and can be resistant to overfitting. However, our data has a few outliers and XGBoost is sensitive to that. To begin, we need to instantiate our model.

In [113…
```
import xgboost as xgb
```

Before adjusting any hyperparameters, we can see that our model predicts fairly well. However, XGBoost regressors can easily overfit the training data. We can review this once we test our final moel against the validation. Now, we need to find appropriate values for hyperparameter we will use in our model. Hyper models are variables we can adjust to fine tune our model. Below we will create plots to evaluate the the performance of the model with different values regarding:

- **reg_lambda**: this can be thought of as a regularization term on the size of the coefficients/weights. This parameter is similar to alpha in the Ridge Regression model we completed beforehand
- **eta**: this is the learning rate in the model Think of this as preventing us from overfitting the data. Overfitting is an issue where our model predicts well on our training data, but fails to generalize to unseen data. The learning rate helps us shrink the new weights received on an iteration. Having one too large results in more iterations until we reach the best value. This process takes longer, and we may never get to the optimized value. On the other hand, a smaller learning rate can result in growing weights too fast. This can cause us to miss the boptimized value completly.
- **gamma**: gamma is another regularization parameter. Basically, it's the minimum information gain (or what many call 'loss reduction) required to split a node. It's essentially a pruning technique. The larger gamma is, the more difficult it will be for a tree to split nodes.
- **max_depth**: The number of levels a tree can have. The larger the number, the more likely it is to over fit
- **n_estimators**: number of trees in our ensemble

Similar to the other models, our goal is to mimimize squared error.

In [114…
```
xgb_regressor = xgb.XGBRegressor()
xgb_regressor.fit(x_train, y_train_log)
xgb_preds = xgb_regressor.predict(x_train)

print("MSE: ", mean_squared_error(y_train_log, xgb_preds))
print("R^2: ", r2_score(y_train_log, xgb_preds))
```

```
MSE:  0.13028960249860458
R^2:  0.8481216787556805
```

As with Ridge Regression, we find the value of the hyper parmeter when R^2 is either at its highest value or close to it withe a slope close to 0. On the other hand, viewing the MSE plot, we want the value where the plot is at it's lowest and beginning to flat. Therefore, the following plots give us the following values as the best values for our hyperparameters:

- **reg_lambda**: 125

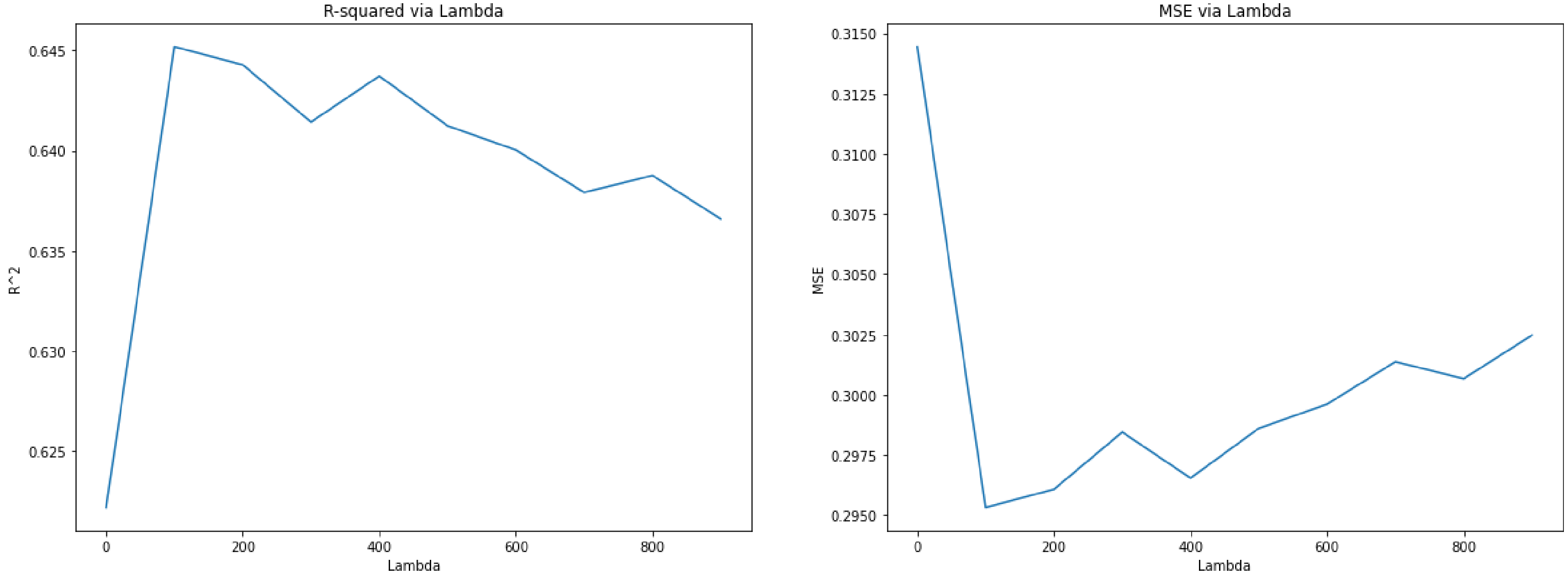
- **eta (learning_rate)**: 0.2
- **gamma**: 1
- **max_depth**: 8
- **n_estimators**: 50

In [115…

```
lambda_ = []
r_squared = []
mse = []
for i in range (0,1000, 100):
    xgb_regressor = xgb.XGBRegressor(objective='reg:squarederror', reg_lambda = i)
    xgb_regressor.fit(x_train, y_train_log)
    xgb_preds = xgb_regressor.predict(x_valid)

    lambda_.append(i)
    mse.append(mean_squared_error(y_valid_log, xgb_preds))
    r_squared.append(r2_score(y_valid_log, xgb_preds))

fig,ax = plt.subplots(1,2, figsize=(20,7))
ax[0].plot(lambda_, r_squared)
ax[0].set_title("R-squared via Lambda")
ax[0].set_xlabel("Lambda")
ax[0].set_ylabel('R^2')
ax[1].plot(lambda_, mse)
ax[1].set_xlabel("Lambda")
ax[1].set_ylabel('MSE')
ax[1].set_title("MSE via Lambda")
plt.plot()
```

Out[115]:

```
[]
```

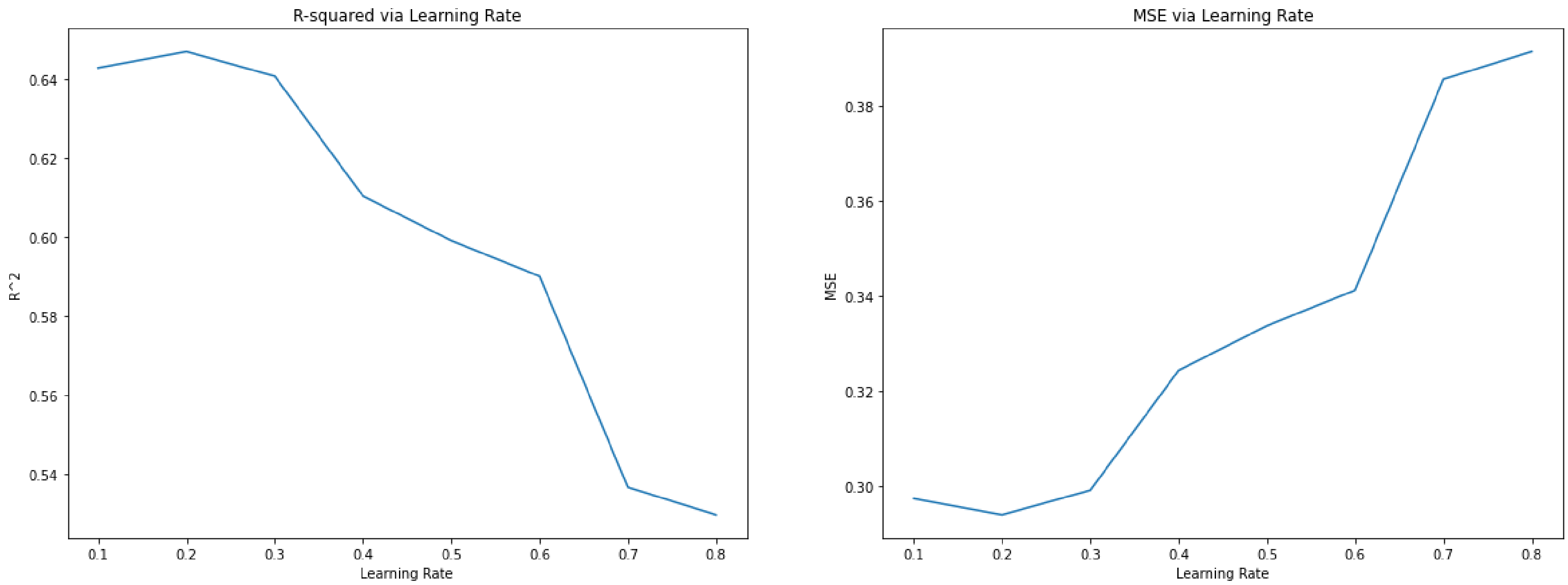


In [116…

```
eta=np.arange(0.1, 0.9, 0.1)
print(eta)
r_squared = []
mse = []
for i in eta:
    xgb_regressor = xgb.XGBRegressor(objective='reg:squarederror', eta=i)
    xgb_regressor.fit(x_train, y_train_log)
    xgb_preds = xgb_regressor.predict(x_valid)

    mse.append(mean_squared_error(y_valid_log, xgb_preds))
    r_squared.append(r2_score(y_valid_log, xgb_preds))

fig,ax = plt.subplots(1,2, figsize=(20,7))
ax[0].plot(eta, r_squared)
ax[0].set_title("R-squared via Learning Rate")
ax[0].set_xlabel("Learning Rate")
ax[0].set_ylabel('R^2')
ax[1].plot(eta, mse)
ax[1].set_xlabel("Learning Rate")
ax[1].set_ylabel('MSE')
ax[1].set_title("MSE via Learning Rate")
plt.plot()
```

```
[0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8]
```

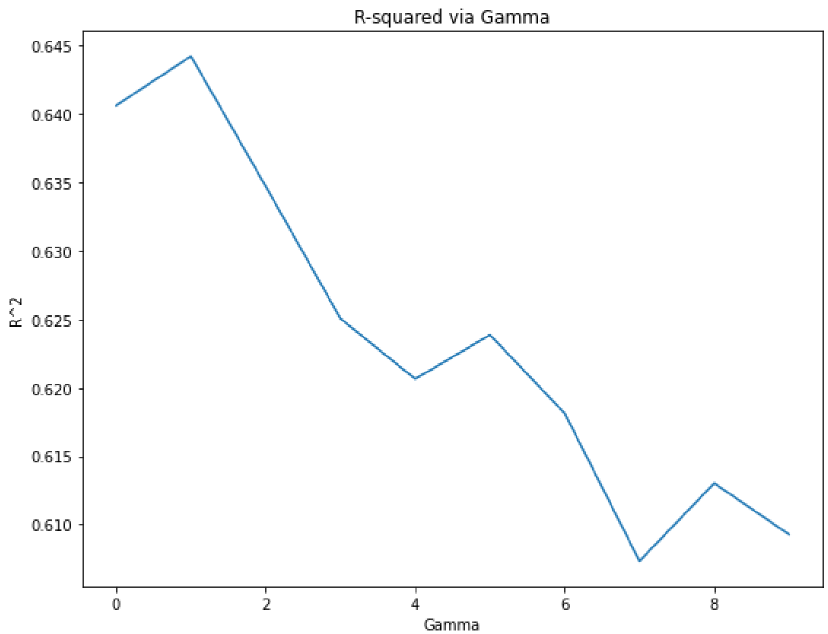
Out[116]: []

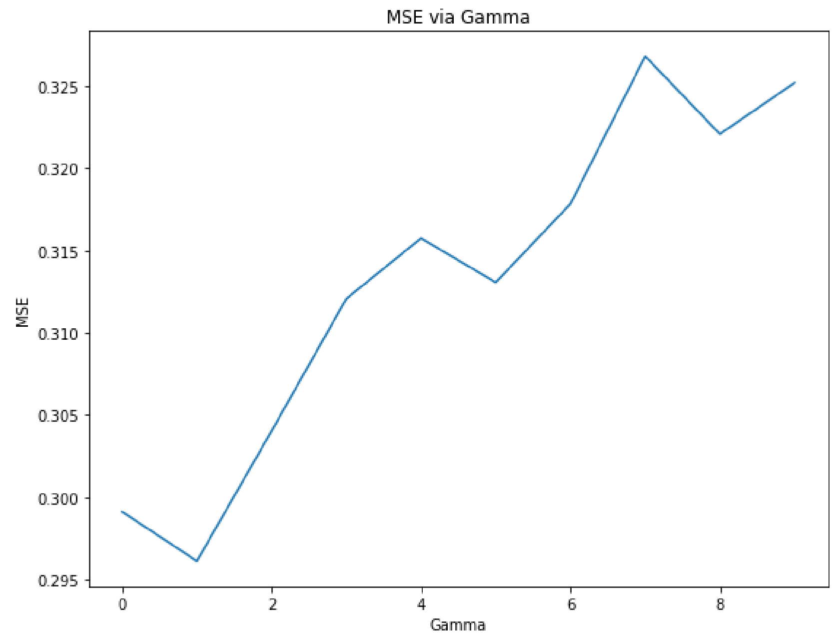


In [117…

```
gamma=np.arange(0,10, 1)
r_squared = []
mse = []
for i in gamma:
    xgb_regressor = xgb.XGBRegressor(objective='reg:squarederror', gamma=i)
    xgb_regressor.fit(x_train, y_train_log)
    xgb_preds = xgb_regressor.predict(x_valid)

    mse.append(mean_squared_error(y_valid_log, xgb_preds))
    r_squared.append(r2_score(y_valid_log, xgb_preds))

fig,ax = plt.subplots(1,2, figsize=(20,7))
ax[0].plot(gamma, r_squared)
ax[0].set_title("R-squared via Gamma")
ax[0].set_xlabel("Gamma")
ax[0].set_ylabel('R^2')
ax[1].plot(gamma, mse)
ax[1].set_xlabel("Gamma")
ax[1].set_ylabel('MSE')
ax[1].set_title("MSE via Gamma")
plt.plot()
```

Out[117]: []

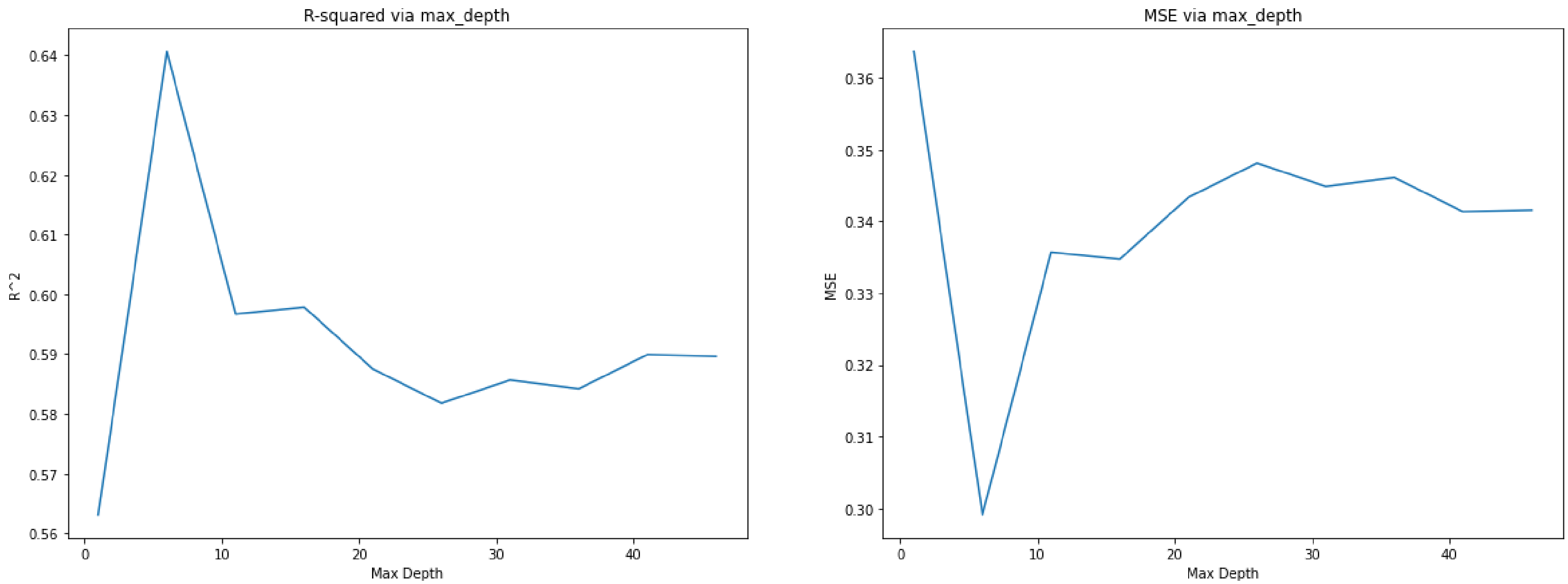




In [118…

```
depth=np.arange(1,50, 5)
r_squared = []
mse = []
for i in depth:
    xgb_regressor = xgb.XGBRegressor(objective='reg:squarederror', max_depth=i)
    xgb_regressor.fit(x_train, y_train_log)
    xgb_preds = xgb_regressor.predict(x_valid)

    mse.append(mean_squared_error(y_valid_log, xgb_preds))
    r_squared.append(r2_score(y_valid_log, xgb_preds))

fig,ax = plt.subplots(1,2, figsize=(20,7))
ax[0].plot(depth, r_squared)
ax[0].set_title("R-squared via max_depth")
ax[0].set_xlabel("Max Depth")
ax[0].set_ylabel('R^2')
ax[1].plot(depth, mse)
ax[1].set_xlabel("Max Depth")
ax[1].set_ylabel('MSE')
ax[1].set_title("MSE via max_depth")
plt.plot()
```

Out[118]: []



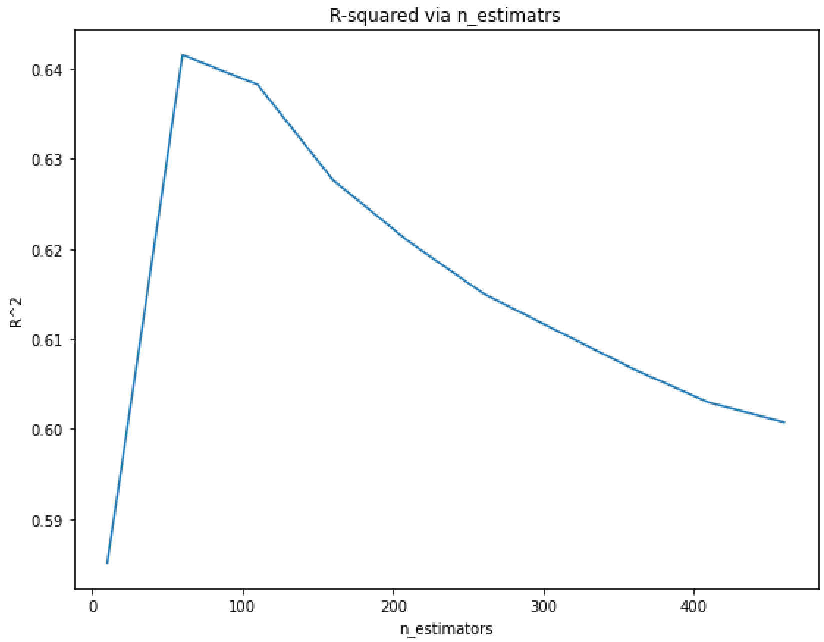
In [119…

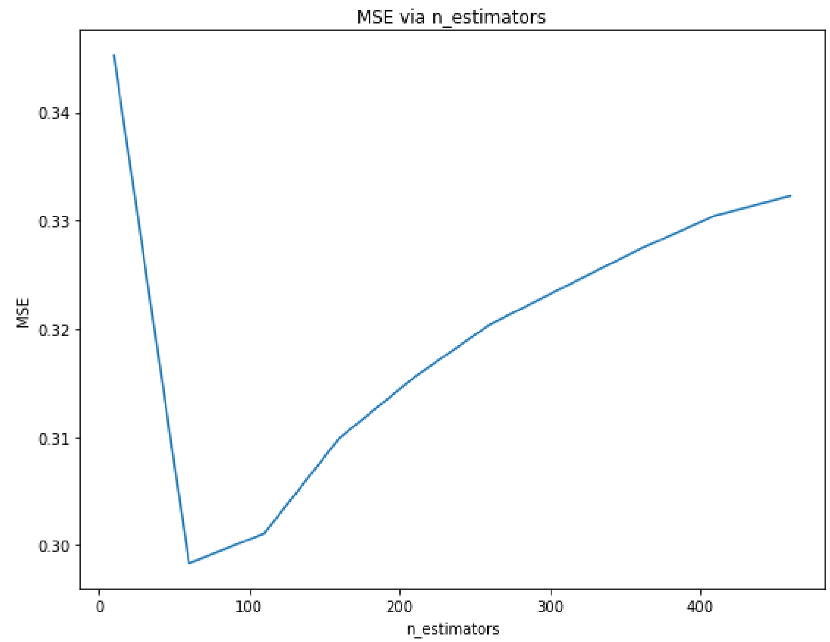
```
estimator_counts=np.arange(10,500, 50)
print(estimator_counts)
r_squared = []
mse = []
for i in estimator_counts:
    xgb_regressor = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=i)
    xgb_regressor.fit(x_train, y_train_log)
    xgb_preds = xgb_regressor.predict(x_valid)

    mse.append(mean_squared_error(y_valid_log, xgb_preds))
    r_squared.append(r2_score(y_valid_log, xgb_preds))

fig,ax = plt.subplots(1,2, figsize=(20,7))
ax[0].plot(estimator_counts, r_squared)
ax[0].set_title("R-squared via n_estimatrs")
ax[0].set_xlabel("n_estimators")
ax[0].set_ylabel('R^2')
ax[1].plot(estimator_counts, mse)
ax[1].set_xlabel("n_estimators")
ax[1].set_ylabel('MSE')
ax[1].set_title("MSE via n_estimators")
plt.plot()
```

```
[ 10  60 110 160 210 260 310 360 410 460]
```

Out[119]:

```
[]
```





In [120…

```
xgb_regressor = xgb.XGBRegressor(objective='reg:squarederror', reg_lambda=125, eta=0.2, gamma=1,
                                max_depth=8, n_estimators=50)
xgb_regressor.fit(x_train, y_train_log)

xgb_preds = xgb_regressor.predict(x_train)
xgb_preds_valid = xgb_regressor.predict(x_valid)

fnl_mse_xgb = mean_squared_error(y_train_log, xgb_preds)
fnl_r_squared_xgb = r2_score(y_train_log, xgb_preds)

fnl_mse_xgb_valid = mean_squared_error(y_valid_log, xgb_preds_valid)
fnl_r_squared_xgb_valid = r2_score(y_valid_log, xgb_preds_valid)


fnl_r_squareds.append(fnl_r_squared_xgb_valid)
fnl_mse.append(fnl_mse_xgb_valid)
fnl_model_name.append('XGB Regressor')

print('The MSE of our final on the Training Data XGB model is ', fnl_mse_xgb)
print('The R^2 of our final on the Training Data XGB model is ', fnl_r_squared_xgb)
print("--------------------------------------------")
```

```
print('The MSE of our final on the Validation Data XGB model is ', fnl_mse_xgb_valid)
print('The R^2 of our final on the Validation Data XGB model is ', fnl_r_squared_xgb_valid)
```

```
The MSE of our final on the Training Data XGB model is  0.2543476430970742
The R^2 of our final on the Training Data XGB model is  0.7035074763817266
--------------------------------------------
The MSE of our final on the Validation Data XGB model is  0.30197739157688774
The R^2 of our final on the Validation Data XGB model is  0.6371812098324292
```

Model Decision and Final Tuning

In [121…
```
from sklearn.model_selection import GridSearchCV
```

Overall, we can see the XGBoost model performed better than both regression models. XGBoost can catch irregularities within the data. Ridge Regression and linear regression perform better when the dependent variable can be expressed as a combination of the predictors. However, as we saw earlier, our data didn't necessarily fit a completly linear model. Because it is nonparameteric, XGB catches the irregularity of the data relation better than either linear model.

In [122…
```
model_metrics_df = pd.DataFrame(zip(fnl_model_name, fnl_r_squareds, fnl_mse), columns = ['Model', 'R^2', 'MSE
model_metrics_df
```

Out[122]:

	Model	R^2	MSE
0	Linear Regression	0.527486	0.393278
1	Ridge Regression	0.562832	0.363859
2	XGB Regressor	0.637181	0.301977

Fine tuning Parameters

We attempted to find the best combination of hyperparameters earlier. However, we only tested our metrics for each hyperparameter individually. We can use `GridSeachCV` to test differen combinations of our hyperparameters for the best result. Our results from the search are:

- **reg_lambda**: 5
- **eta (learning_rate)**: 0.1
- **gamma**: 1

- **max_depth**: 10
- **n_estimators**: 100

In [123…

```
parameters = {

    'reg_lambda': [5, 50, 100, 200],
    'learning_rate': [0.1,0.2,0.3],
    'gamma': [1, 2, 3],
    'max_depth': [5, 10, 15],
    'n_estimators' : [20, 50, 100]
}
```

In [124…

```
#Commented out for timing issues

# clf = GridSearchCV(xgb, parameters, n_jobs=5, cv=3, scoring = 'r2', verbose=2)
# clf.fit(x_train, y_train_log)
```

In [125…

```
#Commented out for timing issues: the params for thes test were shown above

# print(f'The best possible score given the different hyperparameters is: {clf.best_score_}')
# print(f'The best parameters are: {clf.best_params_}')
```

Fitting with final model parameters

Finally, we can fit our data with the best hyperparameters. Our final model test results in a MSE of

In [126…

```
final_model = xgb.XGBRegressor(objective='reg:squarederror', reg_lambda=5, eta=0.1, gamma=1,
                               max_depth=10, n_estimators=100)
final_model.fit(x_train, y_train_log)

final_model_preds = final_model.predict(x_valid)

fnl_mse = mean_squared_error(y_valid_log, final_model_preds)
fnl_r_squared = r2_score(y_valid_log, final_model_preds)

print('The MSE of our final on the Validation Data XGB model is ', fnl_mse)
print('The R^2 of our final on the Validation Data XGB model is ', fnl_r_squared)
```

```
The MSE of our final on the Validation Data XGB model is  0.2936062178998558
The R^2 of our final on the Validation Data XGB model is  0.6472389796870643
```

Feature Importance

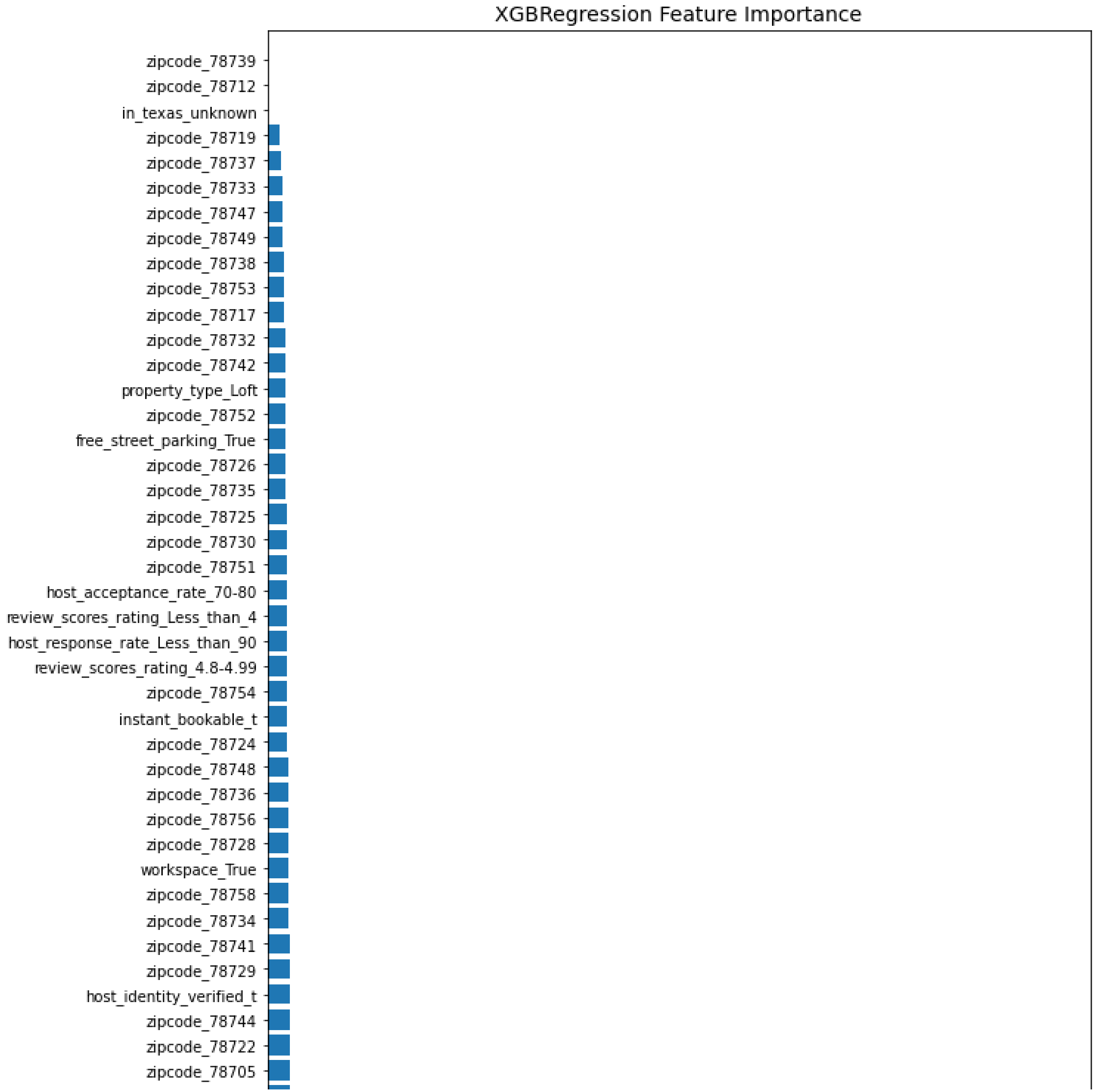
When comparing our training set MSE and R-squared vs our validation set, our training set MSE is slightly higher. This may be due to slight overfitting. We can improve the model by reducing the number of features.

The importance is a score given to a feature for the amount the attribute's pslit point improves the model performance. The score for each attribute is then averaged across all models.

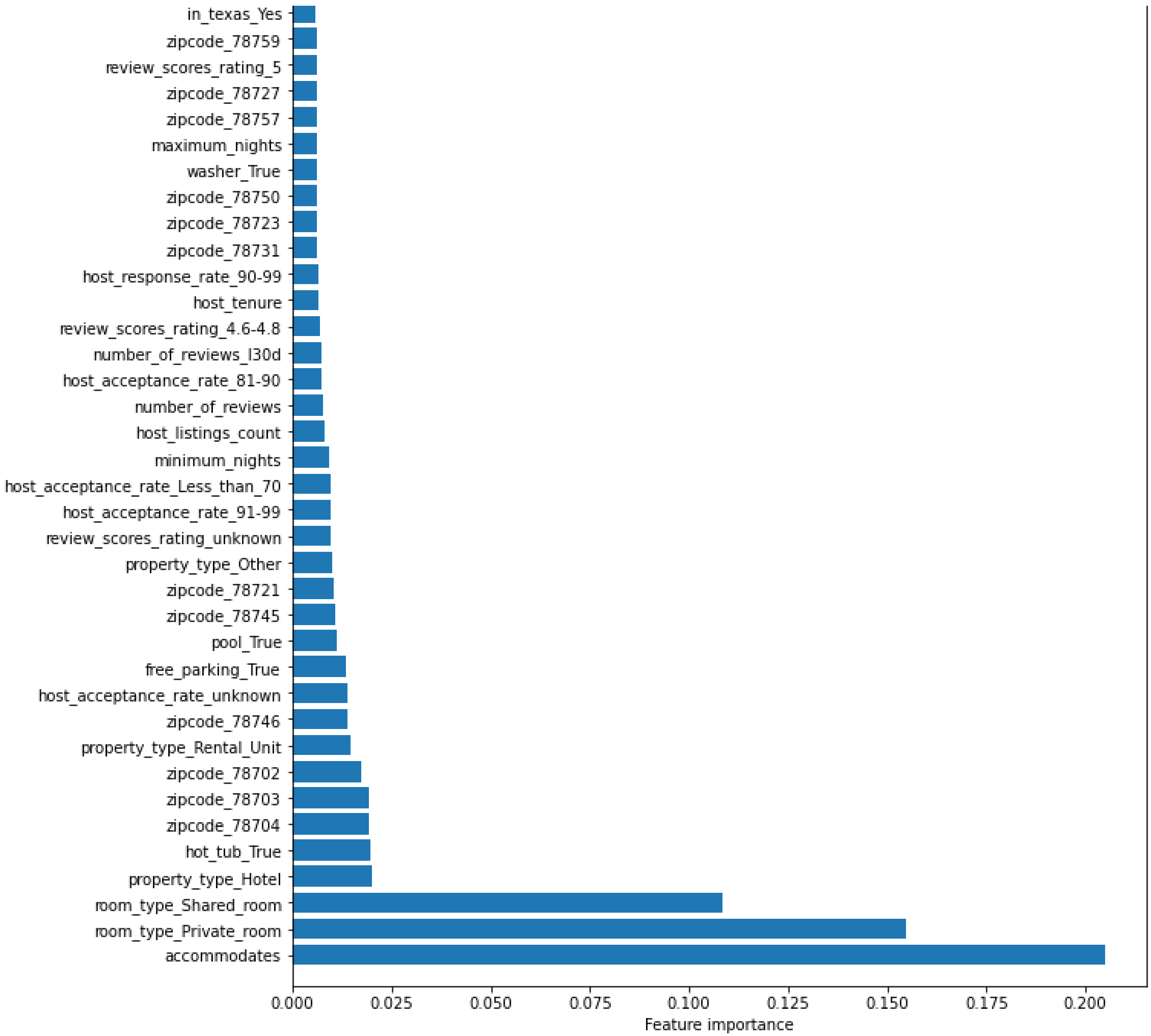
Our final model tends to agree with the decisions our ridge/linear regressions made. `Accommodates` , `room_type_Private_room` , and `room_typ_Shared_room` are, understandibly, the most important features in the model. As stated before, more people require more room on a property. If you have less room, you'll pay less for property. Many zipcodes, and being unaware of the hosts home location are not important in the pricing of the Airbnb.

In [127…

```
feature_weights = pd.DataFrame(zip(x_train.columns, final_model.feature_importances_), columns=['Feature', 'Im
feature_weights = feature_weights.sort_values('Importance', ascending=False)

plt.figure(figsize=(10,25))
plt.barh(feature_weights['Feature'], feature_weights['Importance'], align='center')
plt.title("XGBRegression Feature Importance", fontsize=14)
plt.xlabel("Feature importance")
plt.margins(y=0.01)
plt.show()
```



XGBRegression Feature Importance
zipcode_78739
zipcode_78712
in_texas_unknown
zipcode_78719
zipcode_78737
zipcode_78733
zipcode_78747
zipcode_78749
zipcode_78738
zipcode_78753
zipcode_78717
zipcode_78732
zipcode_78742
property_type_Loft
zipcode_78752
free_street_parking_True
zipcode_78726
zipcode_78735
zipcode_78725
zipcode_78730
zipcode_78751
host_acceptance_rate_70-80
review_scores_rating_Less_than_4
host_response_rate_Less_than_90
review_scores_rating_4.8-4.99
zipcode_78754
instant_bookable_t
zipcode_78724
zipcode_78748
zipcode_78736
zipcode_78756
zipcode_78728
workspace_True
zipcode_78758
zipcode_78734
zipcode_78741
zipcode_78729
host_identity_verified_t
zipcode_78744
zipcode_78722
zipcode_78705



Feature Reduction

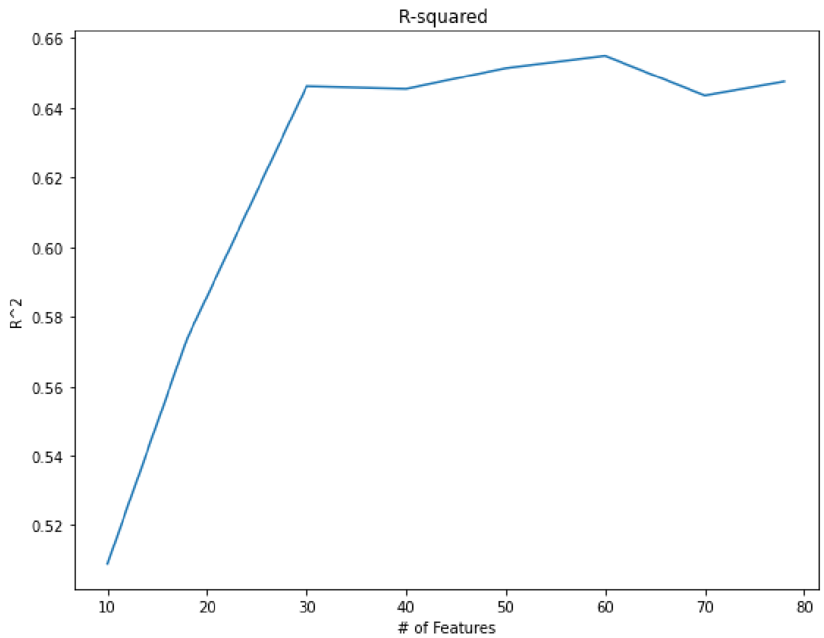
As we stated before, your model overfits the data. Lets test different subset sizes of our dataset to reduce the overfitting. From the plots below, our R^2 is begins to stay aroound 65 at 30 predictors. Again, the MSE states the same.

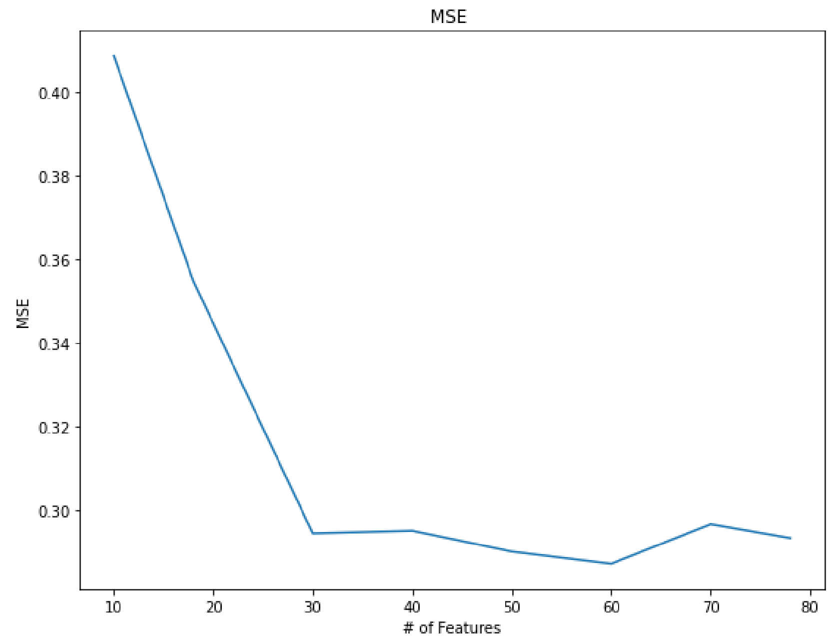
In [128…

```
feature_counts = [10, 18, 30, 40, 50, 60, 70, 78]

mse = []
r_squared = []
for i in feature_counts:

    cols = feature_weights['Feature'][0:i]
    subset_train = x_train[cols]
    subset_valid = x_valid[cols]

    xgb_regressor =  xgb.XGBRegressor(objective='reg:squarederror', reg_lambda=5, eta=0.1, gamma=1,
                                  max_depth=10, n_estimators=100)
    xgb_regressor.fit(subset_train, y_train_log)
    xgb_preds = xgb_regressor.predict(subset_valid)

    mse.append(mean_squared_error(y_valid_log, xgb_preds))
    r_squared.append(r2_score(y_valid_log, xgb_preds))

fig,ax = plt.subplots(1,2, figsize=(20,7))
ax[0].plot(feature_counts, r_squared)
ax[0].set_title("R-squared ")
ax[0].set_xlabel("# of Features")
ax[0].set_ylabel('R^2')
ax[1].plot(feature_counts, mse)
ax[1].set_xlabel("# of Features")
ax[1].set_ylabel('MSE')
ax[1].set_title("MSE ")
plt.plot()
```

Out[128]:

```
[]
```





Final Model after Feature Reduction

Reducing the number of features in our model can help prevent overfitting. With less predictors, we can generalize to unseen data better. Even with removing over half of our features, our Adjusted R-squared stays around 65%.

In [129…

```
cols = feature_weights['Feature'][0:30]
subset_train = x_train[cols]
subset_valid = x_valid[cols]

xgb_final =  xgb.XGBRegressor(objective='reg:squarederror', reg_lambda=5, eta=0.1, gamma=1,
                              max_depth=10, n_estimators=100)
xgb_final.fit(subset_train, y_train_log)
xgb_preds = xgb_final.predict(subset_valid)


print('The MSE of our final on the Validation Data XGB model is ', mean_squared_error(y_valid_log, xgb_preds)
print('The R^2 of our final on the Validation Data XGB model is ', r2_score(y_valid_log, xgb_preds))
```

```
The MSE of our final on the Validation Data XGB model is  0.2944711392678445
The R^2 of our final on the Validation Data XGB model is  0.6461997968439877
```

Testing our final model

Now that we have our final model. We can test it on our test data. Keeping a separate dataset from being used helps use test our model on data we haven't seen before.

In [130…

```
x_test = x_test[cols]
test_preds = xgb_final.predict(x_test)

print('The MSE of our final on the Validation Data XGB model is ', mean_squared_error(y_test_log, test_preds))
print('The R^2 of our final on the Validation Data XGB model is ', r2_score(y_test_log, test_preds))
```

```
The MSE of our final on the Validation Data XGB model is  0.28863948263390005
The R^2 of our final on the Validation Data XGB model is  0.6557908804179018
```

In [138…

```
pd.DataFrame(zip(np.exp(y_test_log), np.exp(test_preds)),
             columns=['Real Value', 'Prediction']).iloc[[0,2, 3, 4, 1191, 1193], :].reset_index(drop=True)
```

Out[138]:

	Real Value	Prediction
0	200.0	148.751328
1	936.0	386.938873
2	120.0	111.919853
3	426.0	326.494324
4	83.0	81.270981
5	191.0	248.900955

Conclusion

Model Summary

Our final model was able to predict 65% of the variation in price with an MSE of 0.29. The 5 most important features were:

- `accommodates` : the number of people staying at the rental
- `room_type_Private_room` : if the rental was a private room orn not
- `room_type_Shared_room` : if the rental was a Shared room or not

- `property_type_Hotel_room` : whether the rental was a hotel room or not
- `hot_tub_True` : Whether the property has a hot tub or not.

It's understandable that the number of people staying at the rental is the most important characteristic to pricing. Typically, this is the first search parameter renters look for. Also, it makes sense the type of room would affect the pricing. When renting a room instead of an entire property the tenants is using less space;therofre, hosts will charge their tenant less.

Next, it's not surprising that hotel room appeared in one of the most important features. Airbnb began as a cheaper alternative to hotels. Thus, charging for a hotel room on Airbnb would be more expensive than going through the hotel itself. Last, hot_tubs are a luxury amenity. So, for a host to charge more for having one is not surprising.

We can see that the host home location is not related much to the pricing of the house. Thus, hosts do a great job of charging thier tenants a good market price instead of the price that fits their home location's market. The location of the property itself seems to have a small amount of influence on pricing. `Zipcode` 78704, 78703, 78702 and 78746 are all located within the western area of Austin. All 4 of these zipcodes are heavily conisdered when pricing properties

What Didn't Work?

The goal of the model was to make it predictable and interpretable. In the end, we sacrificed some interpretability to fit a predictable model. From the beginning, it was obvious the relationship between or predictors and dependent variable were not linear. Thus, we have to log transform our dependent. Although this method does improce the model, it is not preferred. Also, much of our data was collinear. Many variables represented similar aspects of the listings. For example, we had about 10 predictors that represented the number of review for a listing. We also had about 15 variables representing the hosts resonse rate and time. It would have helped to have more features regarding the physical aspect of the house itself.

What about the Unexplained Variance?

The unexplained variance could be due to several of the following reasons:

1. Our dataset didn't capture all of the important features regarding Airbnb pricing. A lot of our features relied on the attributes of the host and reviews online. However, we didn't have much information about the buildings themselves. Looking back at our most important features of our model, the top 10 features dealt with physical characteristics of the property or their location. Thus, we know the physical charcteristics are much more important than the host or reviews left on line.

2. Inconsistencies in the user data. Within our dataset there were a few small 1 bedroom, 1 bath non-luxury apartments with advertised prices of 10000 dollars a night. Sometimes hosts list their properties at a very high prices to avoid anyone from renting their property. Also, people sometimes make errors when pricing their rentals. An extra 0 may be added or a similar mistake may be made. However, we couldn't drop the observations because we couldn't confirm if they were errors or not. If we had access to he actual paid price for these listings, we would have much more accurate data.
3. Typically, when someone rents an Airbnb they are wanting to enjoy the town. For future models, incorportating vicinity to attractions would be great to model. Also, the vicinity to neighbors may help as well

Future Reccommendations

When creating the model, we eliminated many features regarding reviews and descriptions of the lisitngs. For future studiess, text analysis can be used to discover relationships between the text of reviews and the pricing of a property. Also, analysis regarding the text of the house descriptions can be used. What key words are used to attract tenants? Is it related to the housing price? Last, our model was built regarding the housing in Austin, Texas. Future stdies could compare the costs of different cities. Are different features more important in different areas?

In [ ]:
```

```