Name = Manoj Kumar

Batch = 1st September Batch

Course = Data Science Placement Guarantee Course

Email = manojkumarrajput@gmail.com

Part 1 Explaination Video link = https://drive.google.com/file/d/1vy4pCJABSrsbuERtVSaNX41IqUH usp=drive_link

Importing the required packages

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
```

Reading and Exploring Airbnb Dataset

In [3]:
```
data = pd.read_csv("C:\\Users\\msgme\\Downloads\\Airbnb_data.csv")
```

EDA

Data Inspection

In [ ]:
```
## here we can see some basic information about data such as data type of the colum
```

In [5]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 74111 entries, 0 to 74110
Data columns (total 29 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   id                      74111 non-null  int64  
 1   log_price               74111 non-null  float64
 2   property_type           74111 non-null  object 
 3   room_type               74111 non-null  object 
 4   amenities               74111 non-null  object 
 5   accommodates            74111 non-null  int64  
 6   bathrooms               73911 non-null  float64
 7   bed_type                74111 non-null  object 
 8   cancellation_policy     74111 non-null  object 
 9   cleaning_fee            74111 non-null  bool   
 10  city                    74111 non-null  object 
 11  description             74111 non-null  object 
 12  first_review            58247 non-null  object 
 13  host_has_profile_pic    73923 non-null  object 
 14  host_identity_verified  73923 non-null  object 
 15  host_response_rate      55812 non-null  object 
 16  host_since              73923 non-null  object 
 17  instant_bookable        74111 non-null  object 
 18  last_review             58284 non-null  object 
 19  latitude                74111 non-null  float64
 20  longitude               74111 non-null  float64
 21  name                    74111 non-null  object 
 22  neighbourhood           67239 non-null  object 
 23  number_of_reviews       74111 non-null  int64  
 24  review_scores_rating    57389 non-null  float64
 25  thumbnail_url           65895 non-null  object 
 26  zipcode                 73143 non-null  object 
 27  bedrooms                74020 non-null  float64
 28  beds                    73980 non-null  float64
dtypes: bool(1), float64(7), int64(3), object(18)
memory usage: 15.9+ MB
```

In [ ]:
```
### Displayed the top 5 rows of the data, so that we can get a quick view of the da
```

In [7]:
```
data.head()
```

Out[7]:

	id	log_price	property_type	room_type	amenities	accommodates	ba
0	6901257	5.010635	Apartment	Entire home/apt	{"Wireless Internet","Air conditioning",Kitche...	3	
1	6304928	5.129899	Apartment	Entire home/apt	{"Wireless Internet","Air conditioning",Kitche...	7	
2	7919400	4.976734	Apartment	Entire home/apt	{TV,"Cable TV","Wireless Internet","Air condit...	5	
3	13418779	6.620073	House	Entire home/apt	{TV,"Cable TV",Internet,"Wireless Internet",Ki...	4	
4	3808709	4.744932	Apartment	Entire home/apt	{TV,Internet,"Wireless Internet","Air conditio...	2	

5 rows × 29 columns

In [ ]:

```
### Displayed the bottom 5 rows of the data, so that we can get a quick view of the
```

In [9]:

```
data.tail()
```

Out[9]:

	id	log_price	property_type	room_type	amenities	accommodat
74106	14549287	4.605170	Apartment	Private room	{}	
74107	13281809	5.043425	Apartment	Entire home/apt	{TV,"Cable TV",Internet,"Wireless Internet",Ki...	
74108	18688039	5.220356	Apartment	Entire home/apt	{TV,Internet,"Wireless Internet","Air conditio...	
74109	17045948	5.273000	Apartment	Entire home/apt	{TV,"Wireless Internet","Air conditioning",Kit...	
74110	3534845	4.852030	Boat	Entire home/apt	{TV,Internet,"Wireless Internet",Kitchen,"Free...	

5 rows × 29 columns

In [15]:
```
## Got all the columns name using data.columns.tolist()
```

In [11]:
```
print(data.columns.tolist())
```

```
['id', 'log_price', 'property_type', 'room_type', 'amenities', 'accommodates', 'bath
rooms', 'bed_type', 'cancellation_policy', 'cleaning_fee', 'city', 'description', 'f
irst_review', 'host_has_profile_pic', 'host_identity_verified', 'host_response_rat
e', 'host_since', 'instant_bookable', 'last_review', 'latitude', 'longitude', 'nam
e', 'neighbourhood', 'number_of_reviews', 'review_scores_rating', 'thumbnail_url',
'zipcode', 'bedrooms', 'beds']
```

Checking if there are any duplicate rows in the dataset.

In [13]:
```
data.duplicated().sum()
```

Out[13]: 0

There is no duplicate value in the data.

Checking if there are any missing values in the dataset.

In [15]:
```
missing_values= data.isnull().sum()
print(missing_values)
```

```
id                             0
log_price                      0
property_type                  0
room_type                      0
amenities                      0
accommodates                   0
bathrooms                    200
bed_type                       0
cancellation_policy            0
cleaning_fee                   0
city                           0
description                    0
first_review               15864
host_has_profile_pic         188
host_identity_verified       188
host_response_rate         18299
host_since                   188
instant_bookable               0
last_review                15827
latitude                       0
longitude                      0
name                           0
neighbourhood               6872
number_of_reviews              0
review_scores_rating       16722
thumbnail_url               8216
zipcode                      968
bedrooms                      91
beds                         131
dtype: int64
```

In [17]:
```
print(missing_values[missing_values > 0])
```

```
bathrooms                    200
first_review               15864
host_has_profile_pic         188
host_identity_verified       188
host_response_rate         18299
host_since                   188
last_review                15827
neighbourhood               6872
review_scores_rating       16722
thumbnail_url               8216
zipcode                      968
bedrooms                      91
beds                         131
dtype: int64
```

Summary Statistics

In [19]:
```
data.describe()
```

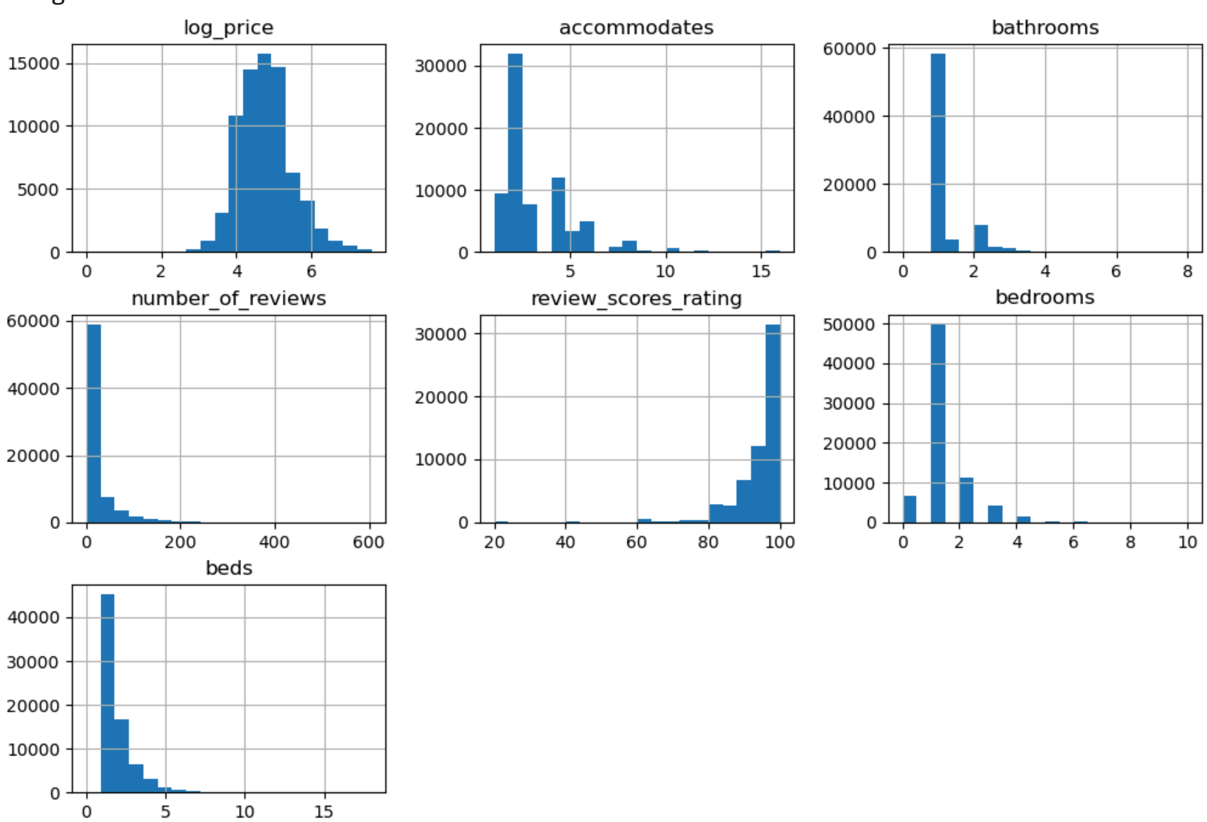
Out[19]:

	id	log_price	accommodates	bathrooms	latitude	longitu
count	7.411100e+04	74111.000000	74111.000000	73911.000000	74111.000000	74111.0000
mean	1.126662e+07	4.782069	3.155146	1.235263	38.445958	-92.3975
std	6.081735e+06	0.717394	2.153589	0.582044	3.080167	21.7053
min	3.440000e+02	0.000000	1.000000	0.000000	33.338905	-122.5115
25%	6.261964e+06	4.317488	2.000000	1.000000	34.127908	-118.3423
50%	1.225415e+07	4.709530	2.000000	1.000000	40.662138	-76.9969
75%	1.640226e+07	5.220356	4.000000	1.000000	40.746096	-73.9546
max	2.123090e+07	7.600402	16.000000	8.000000	42.390437	-70.9850

In [21]:

```
num_columns = ['log_price','accommodates','bathrooms','number_of_reviews','review_s
plt.figure(figsize=(10,8))
data[num_columns].hist(bins=20, figsize=(12, 8), layout=(3, 3))
plt.show()
```

```
<Figure size 1000x800 with 0 Axes>
```



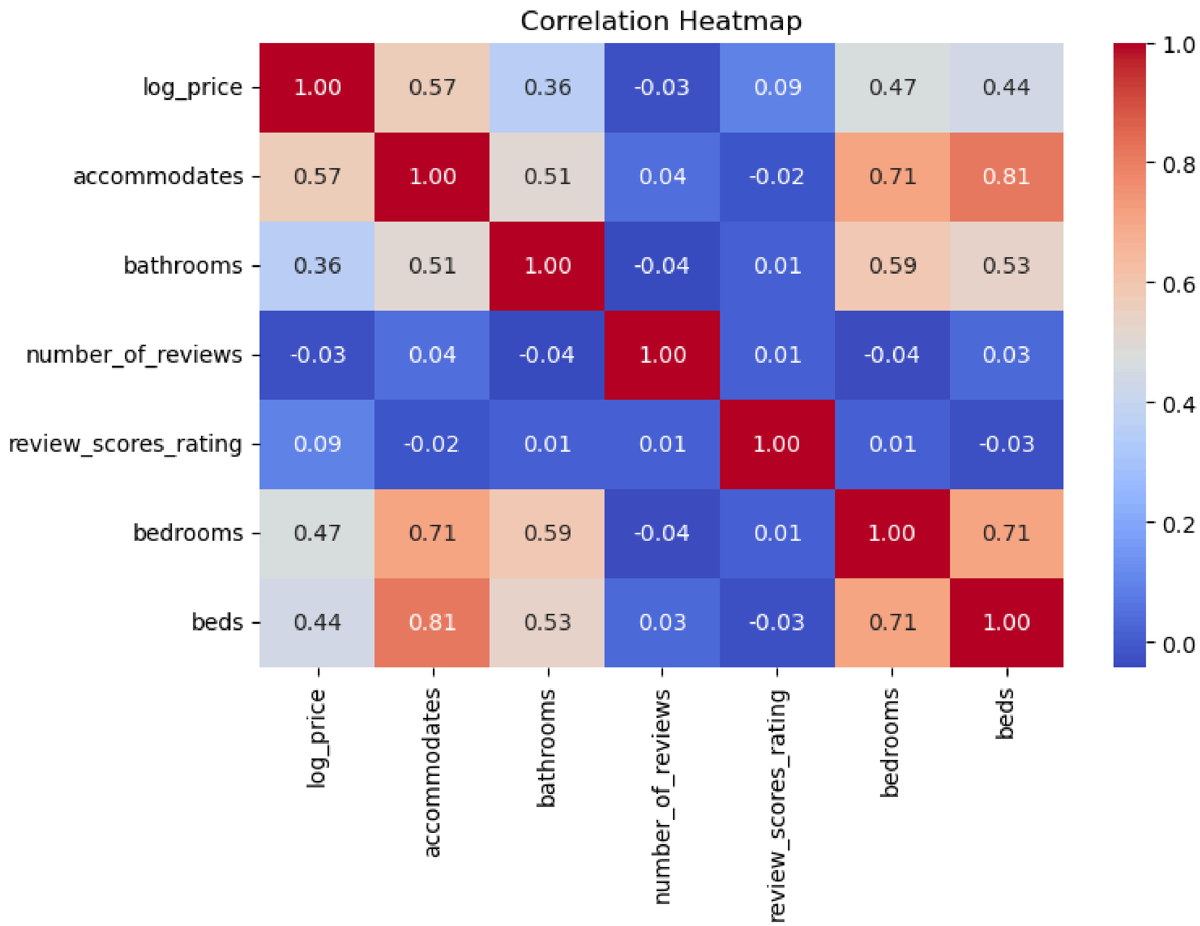
Here, we are analyzing the distribution of all numerical columns. As seen above, some column values are more

concentrated within a specific range, while others are more widely spread.

Additionally, certain graphs show left or right skewness, such as 'number_of_reviews,' 'review_scores_rating,'

and 'beds'. This suggests the presence of outliers in these columns.

In [23]:
```
plt.figure(figsize=(8, 5))
sns.heatmap(data[num_columns].corr(), annot=True, cmap='coolwarm', fmt='.2f')
plt.title("Correlation Heatmap")
plt.show()
```



We found that the correlation between Log Price and Accommodates is 0.57, which indicates a moderate

positive relationship. This means that as the number of guests a listing can accommodate increases,

the price also tends to go up. Similarly, the more people a place can host, the more bedrooms it usually requires.

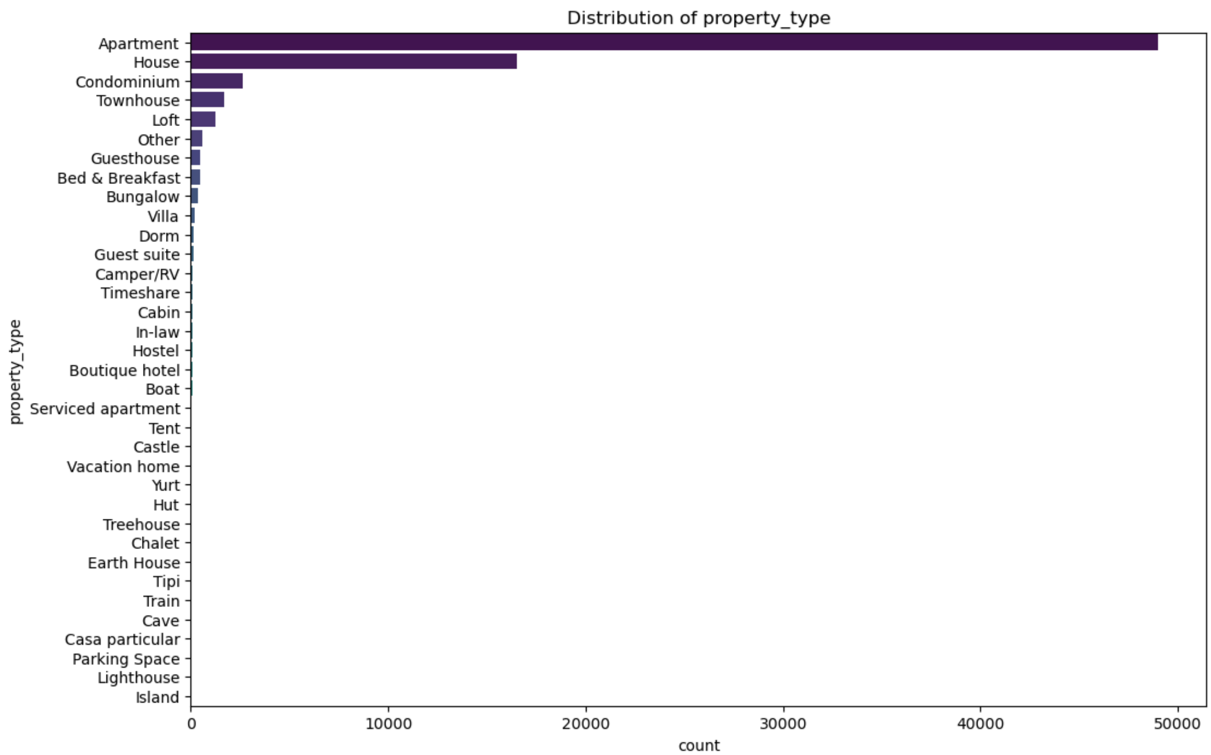
A correlation of 0.71 suggests a strong connection, showing that listings with higher accommodation capacity often

have more bedrooms. Likewise, a correlation of 0.81 indicates a very strong relationship, meaning the number of beds

is closely linked to the number of guests a place can accommodate.

Additionally, some factors have a positive correlation, while others have a negative one, as seen in the correlation heatmap.

In [25]:
```
property_ty = ['property_type']
for colmn in property_ty:
    plt.figure(figsize=(12, 8))
    sns.countplot(y=data[colmn], order=data[colmn].value_counts().index, palette='v
    plt.title(f"Distribution of {colmn}")
    plt.show()
```



Airbnb listings are mostly concentrated in urban areas, where apartments are widely available. This is

because apartments are easier to manage for property owners and investors, and they are also a preferred

choice for tourists. This likely explains why the number of apartment listings is higher than other property types.

The second most common type is houses, as they are ideal for families and group travelers. Condominiums fall

somewhere in between—these are essentially luxury apartments that offer additional amenities like gyms, pools, and security.

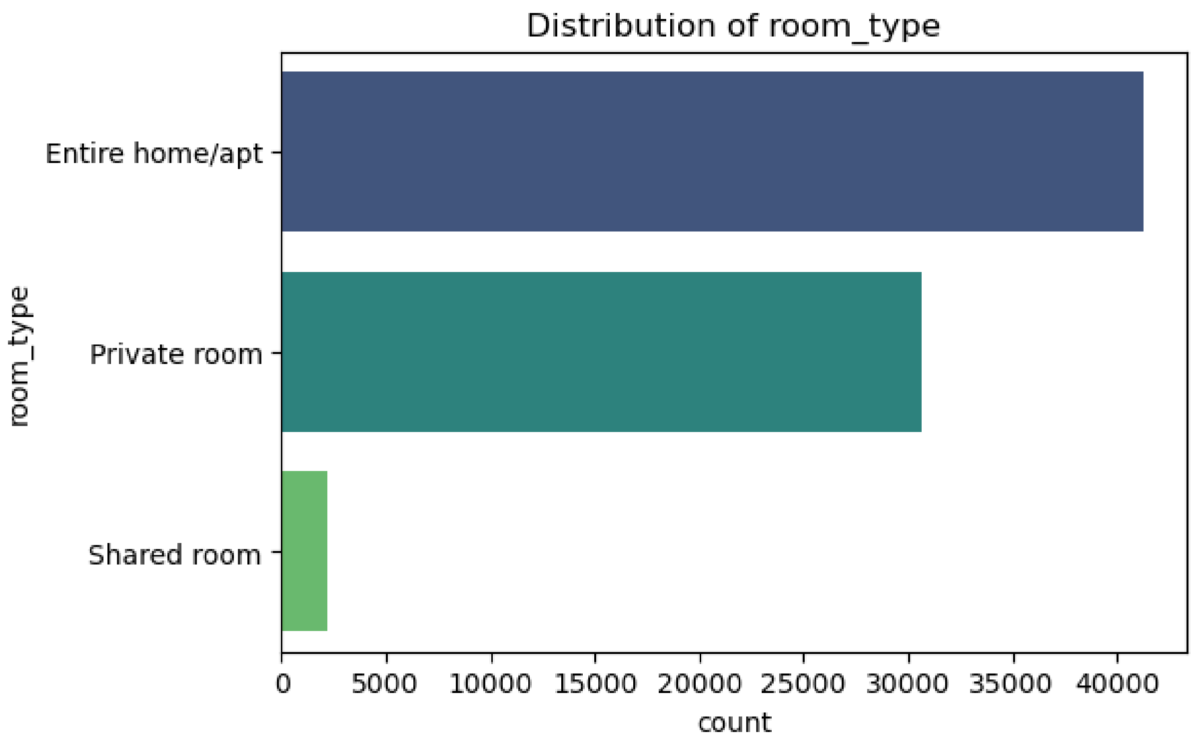
On the other hand, townhouses have fewer listings compared to apartments and houses. They are typically located in

residential and suburban areas, making them less common for urban tourists.

As seen above, the most frequent property type is 'Apartment,' followed by 'House' and then 'Condominium.' This indicates

that these types of properties are rented out more often than others.

In [27]:
```
room_ty = ['room_type']
for colmn in room_ty:
    plt.figure(figsize=(6, 4))
    sns.countplot(y=data[colmn], order=data[colmn].value_counts().index, palette='v
    plt.title(f"Distribution of {colmn}")
    plt.show()
```



As we can see in the count plot, 'Entire home/apt' is the most common type of listing on Airbnb. This may

be due to the privacy and comfort it offers, as guests likely prefer renting the entire property to have

full privacy and independence. It's also the best option for families and large groups, as well as a higher

revenue opportunity for hosts. Hosts can generate more income by renting out the whole property, which explains

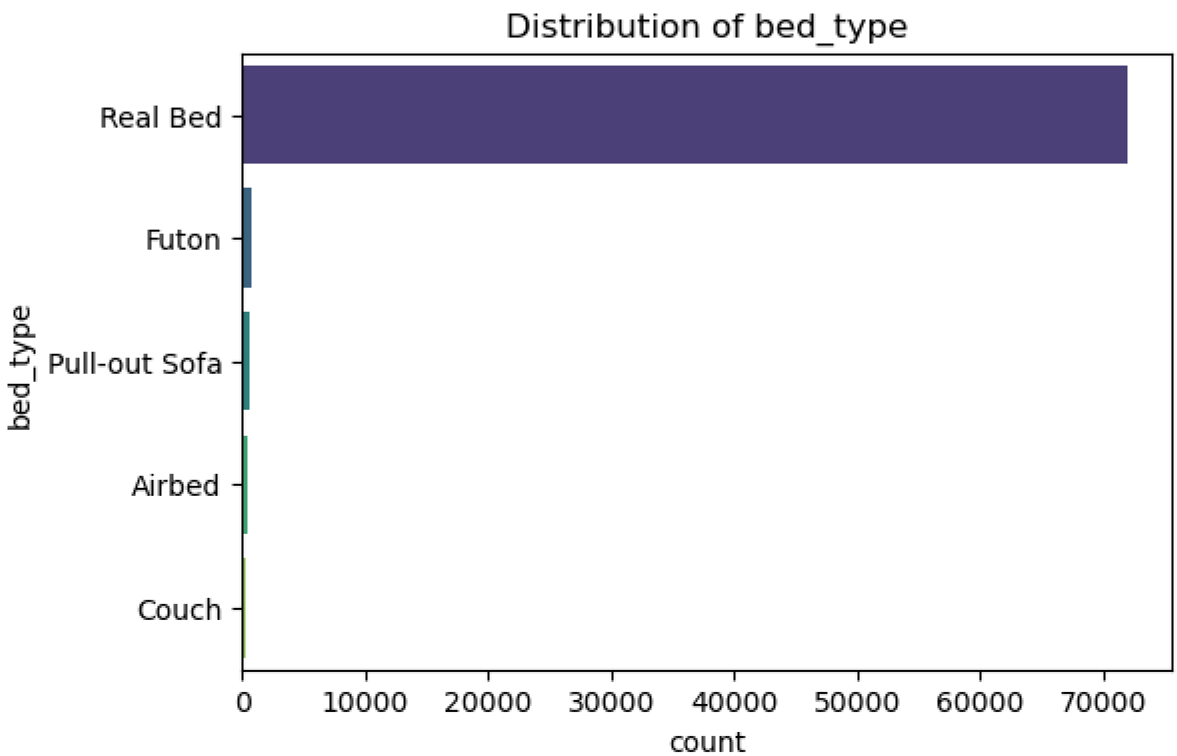
why these listings are more prevalent.

'Private' listings come second, possibly because they offer a budget-friendly option with privacy. This may appeal to those who want to save on costs while still having some privacy. Shared rooms, on the other hand,

are in lower demand because multiple guests share the room or dormitory, which many people do not prefer.

Individual travelers, especially, tend to prefer privacy, which is why shared rooms are less common.

In [29]:
```
bed_ty = ['bed_type']
for colmn in bed_ty:
    plt.figure(figsize=(6, 4))
    sns.countplot(y=data[colmn], order=data[colmn].value_counts().index, palette='v
    plt.title(f"Distribution of {colmn}")
    plt.show()
```



We found that the most preferred bed type among travelers is the 'Real Bed,' while 'Couch' is almost nonexistent.

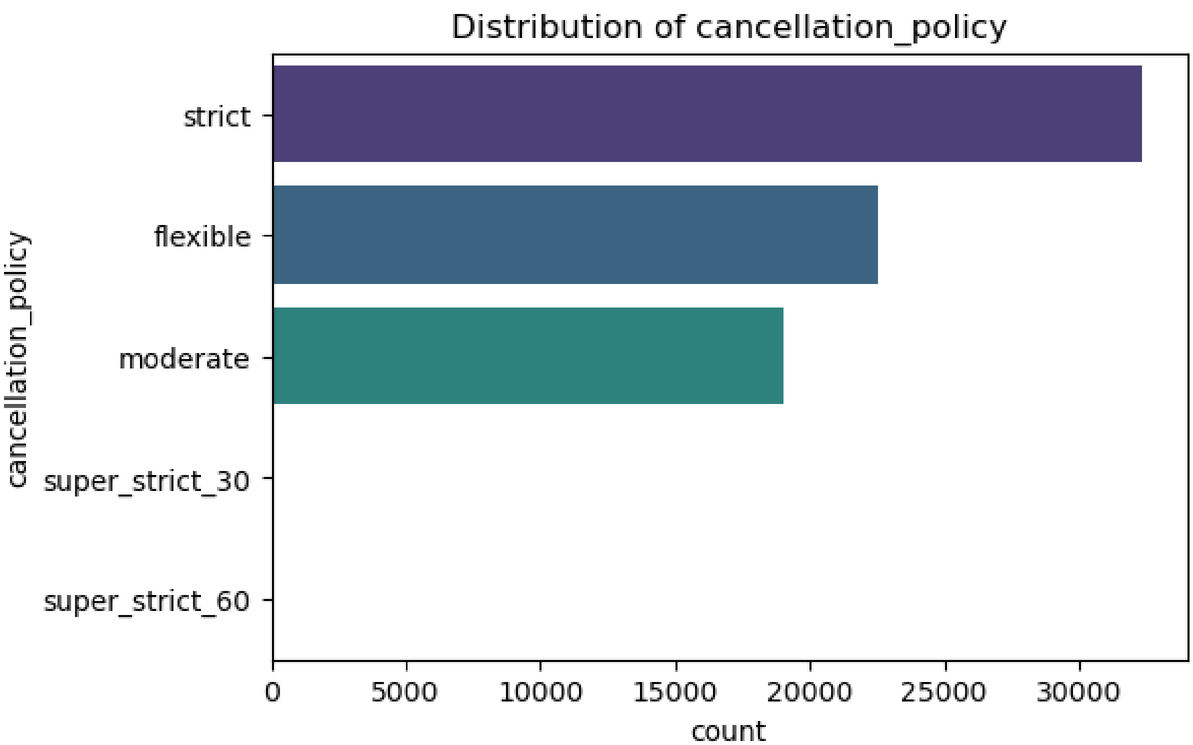
This suggests a high demand for real beds in Airbnb listings. The 'Real Bed' is the most common bed type because

guests typically prefer a proper bed for a comfortable stay. It is commonly found in apartments, houses, and premium

listings. The higher availability of this bed type may indicate that hosts are using it to make their listings more

attractive to potential guests.

In [31]:
```
cancellation_poli = ['cancellation_policy']
for colmn in cancellation_poli:
    plt.figure(figsize=(6, 4))
    sns.countplot(y=data[colmn], order=data[colmn].value_counts().index, palette='v
    plt.title(f"Distribution of {colmn}")
    plt.show()
```



We found that the 'Strict' cancellation policy is more common than others, indicating that many hosts prefer this

policy to avoid cancellations. However, offering more flexibility in the cancellation policy could potentially

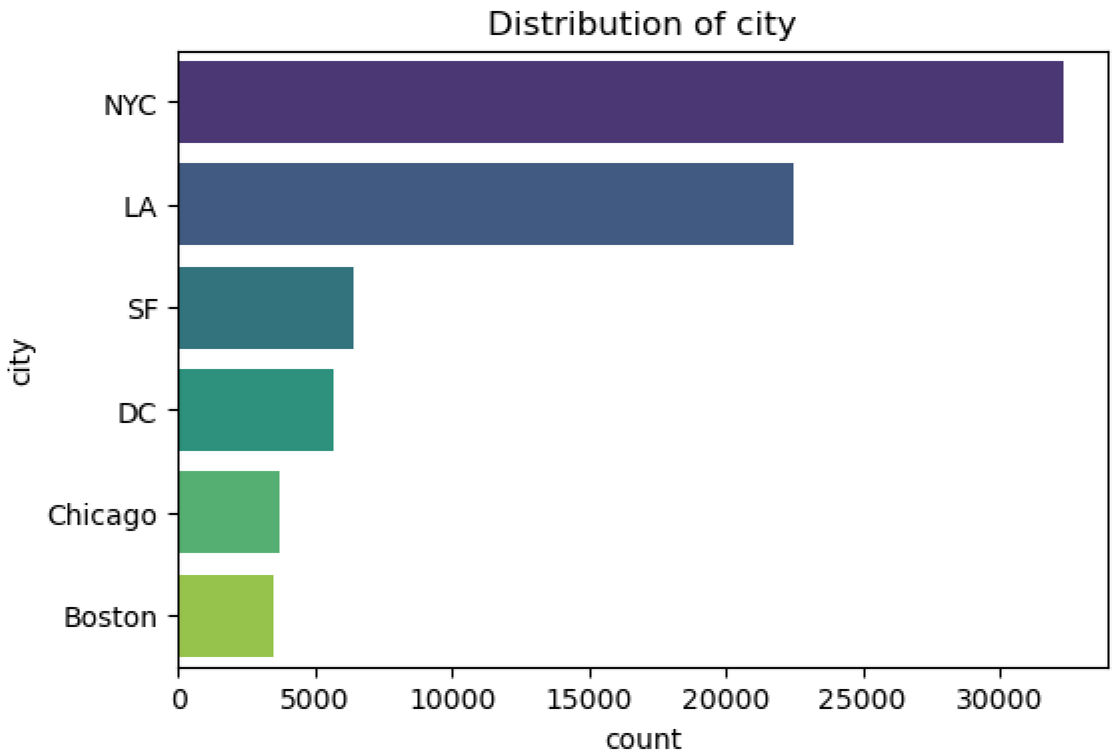
increase bookings. With a strict cancellation policy, guests have less flexibility, which may cause hesitation

during the booking process and could influence the price perception. Hosts should be aware that their cancellation

policy can affect listing prices. If they opt for a strict policy, they may need to adjust their prices to attract

more guests.

In [33]:
```
city_type = ['city']
for colmn in city_type:
    plt.figure(figsize=(6, 4))
    sns.countplot(y=data[colmn], order=data[colmn].value_counts().index, palette='v
    plt.title(f"Distribution of {colmn}")
    plt.show()
```

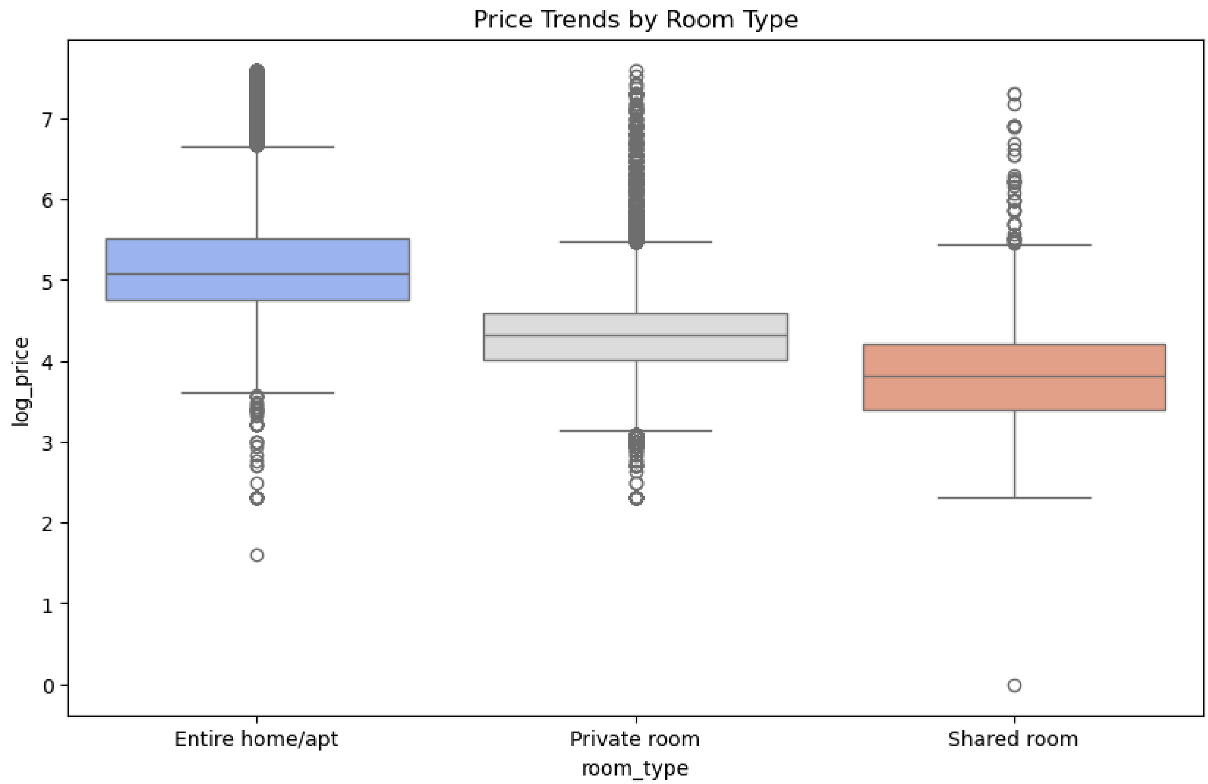


New York and Los Angeles are major metropolitan cities, and as we can see in the count plot, both

cities show high demand. This suggests that prices in these cities are high due to the premium

amenities, higher cost of living, and strong demand for listings.

In [35]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(x=data['room_type'], y=data['log_price'], palette='coolwarm')
plt.title("Price Trends by Room Type")
plt.show()
```



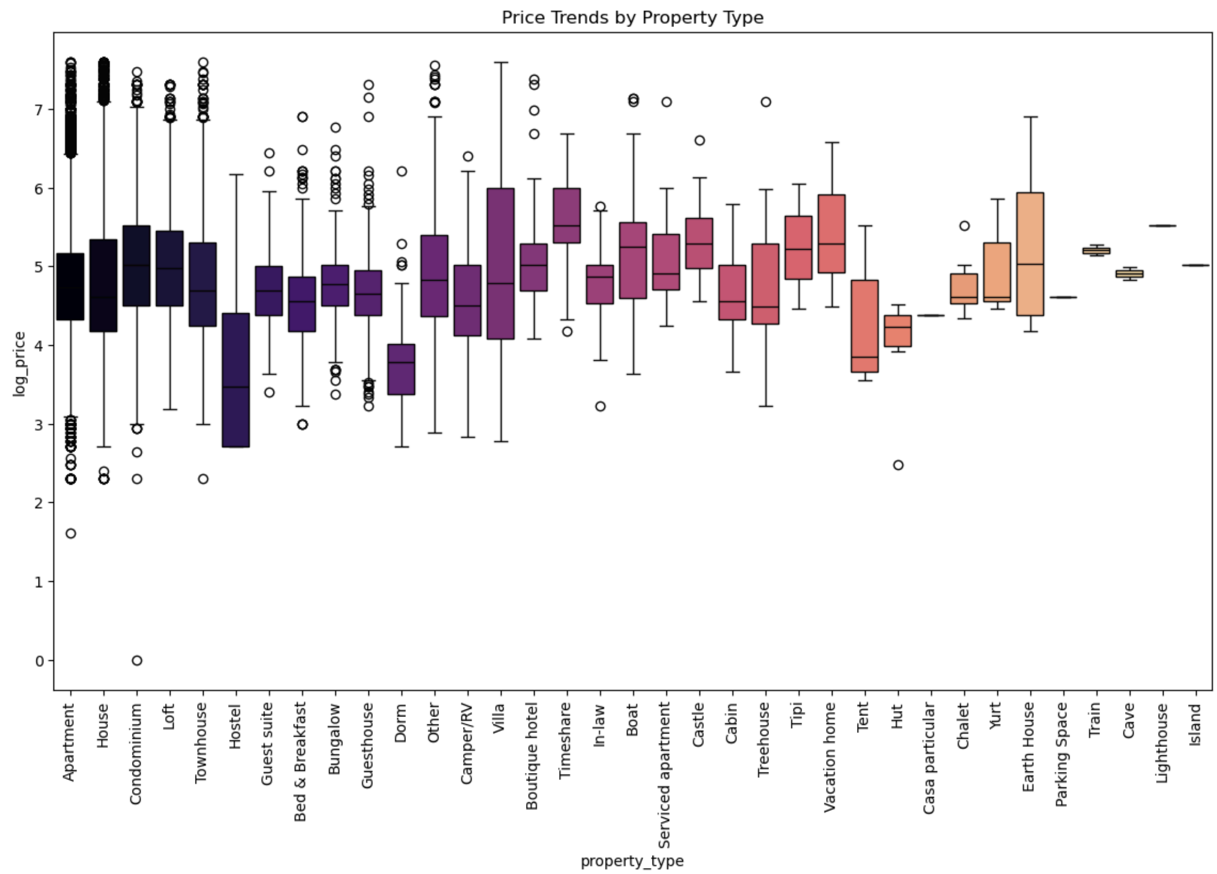
As we can see in the graph above, the price for 'Entire home/apt' is generally higher than that of the

'Private room' category, and 'Shared room' is typically the lowest. However, there are some outliers in

both the 'Private room' and 'Shared room' categories, where the prices sometimes increase and approach

those of the 'Entire home/apt' category.

In [37]:
```
plt.figure(figsize=(14, 8))
sns.boxplot(x=data['property_type'], y=data['log_price'], palette='magma')
plt.xticks(rotation=90)
plt.title("Price Trends by Property Type")
plt.show()
```



In the boxplot above, we can compare property types with their log-transformed prices. As we can see, different

property types have varying price ranges—some are higher, while others are lower. However, there are outliers

where the property prices can sometimes spike very high or drop very low.

Property type plays a significant role in pricing strategies.

Data Preprocessing & Exploration

Converting 'first_review','host_since','last_review' column to datetime format

In [39]:
```
data['first_review'] = pd.to_datetime(data['first_review'], errors='coerce')
data['host_since'] = pd.to_datetime(data['host_since'], errors='coerce')
data['last_review'] = pd.to_datetime(data['last_review'], errors='coerce')
```

Convert everything to string, strip spaces, remove '%' safely

In [41]:
```
data['host_response_rate'] = data['host_response_rate'].astype(str).str.strip().str
```

Convert to numeric, ensuring no unintended NaNs

In [43]:
```
data['host_response_rate'] = data['host_response_rate'].apply(lambda x: float(x) /
```

Handling Missing Values

Filling categorical missing values with mode() and numarical with median()

In [45]:
```
for col in data.columns:
    if data[col].dtype == 'object':
        data[col].fillna(data[col].mode()[0],inplace=True)
    else:
        data[col].fillna(data[col].median(),inplace=True)
```

In [70]:
```
## Checking is missing values got removed or not ?
```

In [47]:
```
data.isnull().sum()
```

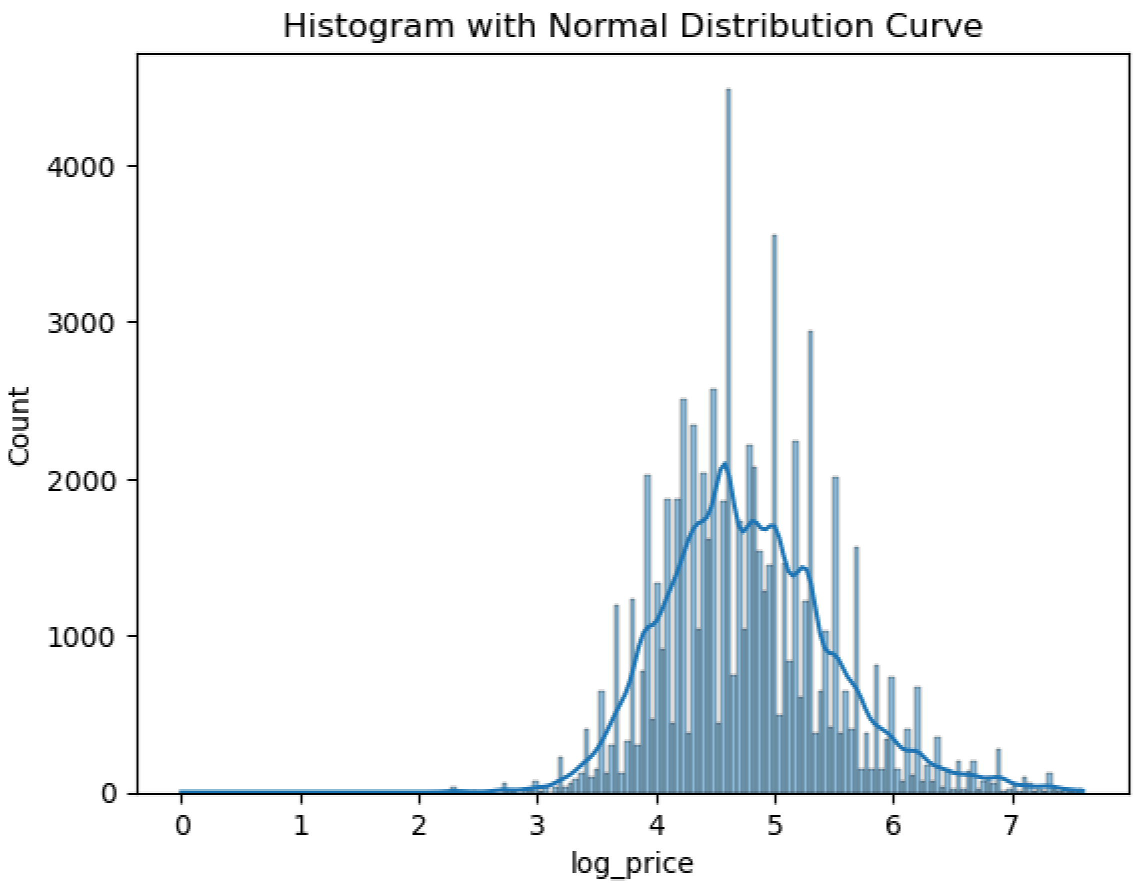
Out[47]:
```
id                        0
log_price                 0
property_type             0
room_type                 0
amenities                 0
accommodates              0
bathrooms                 0
bed_type                  0
cancellation_policy       0
cleaning_fee              0
city                      0
description               0
first_review              0
host_has_profile_pic      0
host_identity_verified    0
host_response_rate        0
host_since                0
instant_bookable          0
last_review               0
latitude                  0
longitude                 0
name                      0
neighbourhood             0
number_of_reviews         0
review_scores_rating      0
thumbnail_url             0
zipcode                   0
bedrooms                  0
beds                      0
dtype: int64
```

In [49]:
```
missing_val=(data.isnull().sum())
print(missing_val[missing_val>0])
```

```
Series([], dtype: int64)
```

Ploting the histogram to check normal distribution curve

In [51]:
```
sns.histplot(data['log_price'], kde=True)
plt.title('Histogram with Normal Distribution Curve')
plt.show()
```



This distribution is negatively skewed, meaning that there are more listings in the lower price range,

while listings at the higher price end are fewer. The distribution is more spread out on the left side,

with most listings clustered in the lower price range. As shown in the curve, the left side is steep,

indicating that the majority of listings fall within the lower price range. The curve then rises sharply

as the prices increase, reflecting the gradual rise in prices towards the middle range.

Detecting & Handling Outliers

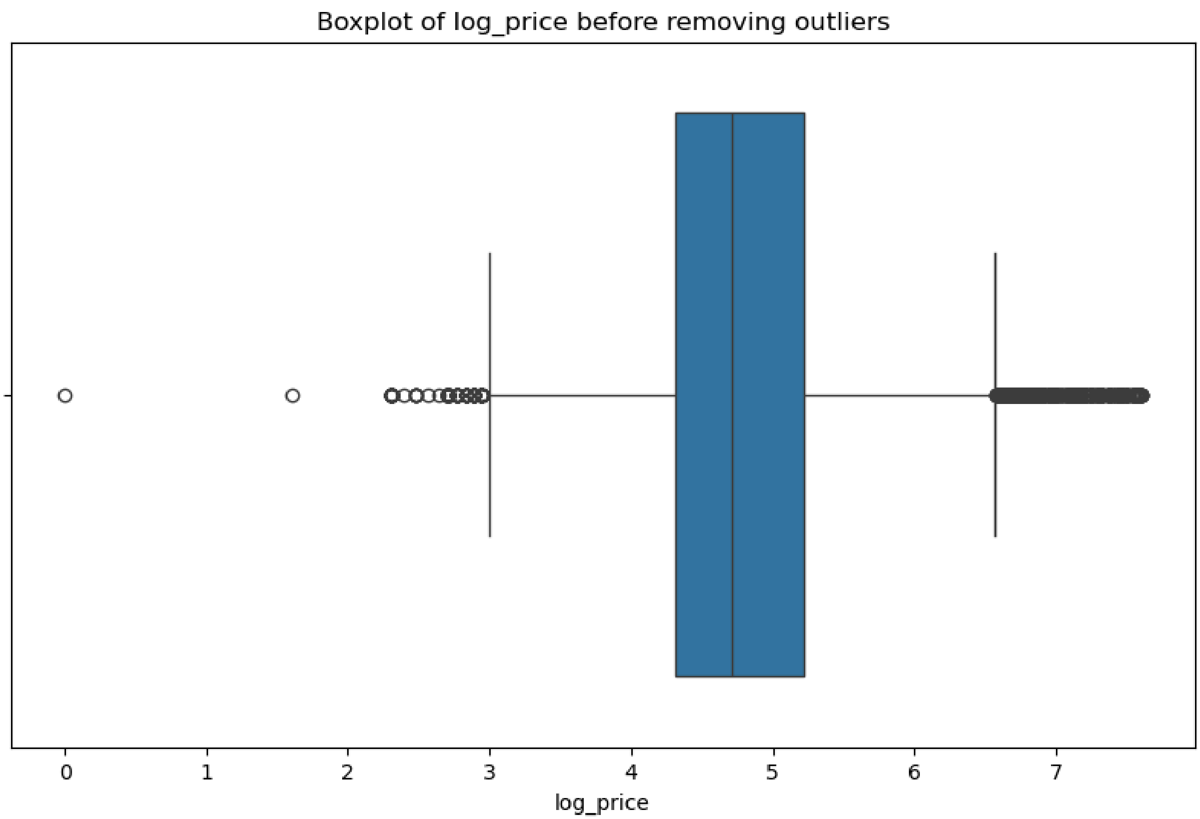
In [53]:
```
data.describe()
```

Out[53]:

	id	log_price	accommodates	bathrooms	first_review	host_
count	7.411100e+04	74111.000000	74111.000000	74111.000000	74111	
mean	1.126662e+07	4.782069	3.155146	1.234628	2016-02-06 14:40:40.738891776	
min	3.440000e+02	0.000000	1.000000	0.000000	2008-11-17 00:00:00	
25%	6.261964e+06	4.317488	2.000000	1.000000	2015-09-14 00:00:00	
50%	1.225415e+07	4.709530	2.000000	1.000000	2016-05-14 00:00:00	
75%	1.640226e+07	5.220356	4.000000	1.000000	2016-10-28 00:00:00	
max	2.123090e+07	7.600402	16.000000	8.000000	2017-12-09 00:00:00	
std	6.081735e+06	0.717394	2.153589	0.581386	NaN	

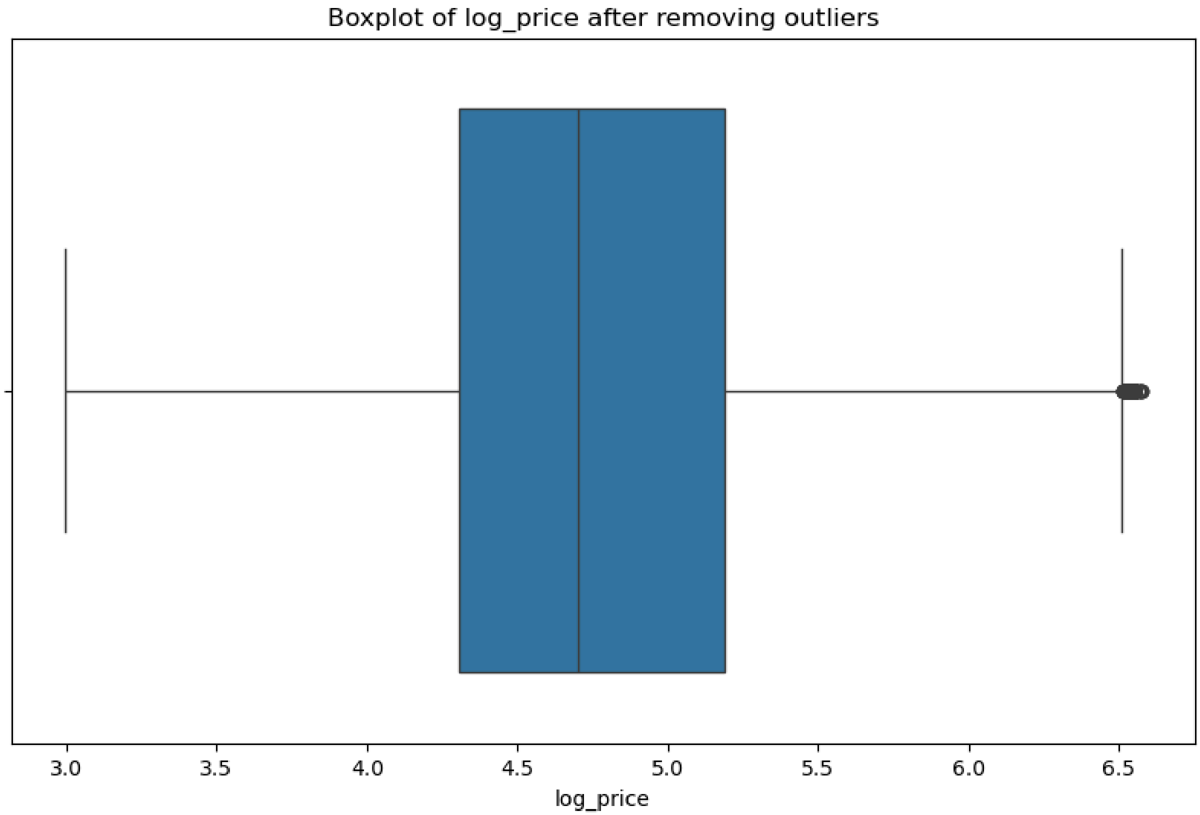
In [55]:

```
# Outlier Detection and Treatment
plt.figure(figsize=(10,6))
sns.boxplot(x=data["log_price"])
plt.title("Boxplot of log_price before removing outliers")
plt.show()
```



In [57]:
```
# Removing outliers using IQR method
Q1 = data["log_price"].quantile(0.25)
Q3 = data["log_price"].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
data = data[(data["log_price"] >= lower_bound) & (data["log_price"] <= upper_bound)
```

In [59]:
```
plt.figure(figsize=(10,6))
sns.boxplot(x=data["log_price"])
plt.title("Boxplot of log_price after removing outliers")
plt.show()
```



Droping unnecessary columns that won't contribute to the model (ID, description, name, thumbnail_url, city, zipcode)

In [61]:
```
data.drop(columns=['id', 'name', 'description', 'thumbnail_url', 'city', 'zipcode']
```

In [63]:
```
data.head()
```

Out[63]:

	log_price	property_type	room_type	amenities	accommodates	bathroom
0	5.010635	Apartment	Entire home/apt	{"Wireless Internet","Air conditioning",Kitche...	3	1.(
1	5.129899	Apartment	Entire home/apt	{"Wireless Internet","Air conditioning",Kitche...	7	1.(
2	4.976734	Apartment	Entire home/apt	{TV,"Cable TV","Wireless Internet","Air condit...	5	1.(
4	4.744932	Apartment	Entire home/apt	{TV,Internet,"Wireless Internet","Air conditio...	2	1.(
5	4.442651	Apartment	Private room	{TV,"Wireless Internet",Heating,"Smoke detecto...	2	1.(

5 rows × 23 columns

Feature Engineering:

Extracting meaningful features, such as neighborhood popularity,number of amenities, and host activity metrics.

In [65]:
```
data1=data.copy()
```

In [67]:
```
# 1. neighborhood_popularity: Counting the number of neighbourhood listed in the 'n
data1['neighborhood_popularity'] = data1.groupby('neighbourhood')['neighbourhood'].

# 2. Number of Amenities: Counting the number of amenities listed in the 'amenities
data1['num_amenities'] = data1['amenities'].apply(lambda x: len(x.split(',')) if is

# 3. Host Activity Metrics:
# a. Host Experience: Calculating the number of years since the host first joined (
data1['host_experience'] = data1['host_since'].apply(lambda x: 2023 - int(x.split('

# b. Host Response Rate: Converting response rate to a numeric percentage value
data1['host_response_rate'] = data1['host_response_rate']

# c. Profile Picture: Converting the 'host_has_profile_pic' into a binary feature (
data1['host_has_profile_pic'] = data1['host_has_profile_pic'].apply(lambda x: 1 if

# d. Identity Verified: Converting the 'host_identity_verified' into a binary featu
data1['host_identity_verified'] = data1['host_identity_verified'].apply(lambda x: 1

# Displayed the first few rows of the engineered features to check
print(data1[['neighbourhood', 'neighborhood_popularity', 'num_amenities', 'host_exp
          'host_has_profile_pic', 'host_identity_verified']].head())
```

```
     neighbourhood  neighborhood_popularity  num_amenities  host_experience  \
0  Brooklyn Heights                      108              9                0
1    Hell's Kitchen                     1283             15                0
2            Harlem                     1365             19                0
4  Columbia Heights                      283             12                0
5        Noe Valley                      304             10                0

   host_response_rate  host_has_profile_pic  host_identity_verified
0                 1.0                     1                       1
1                 1.0                     1                       0
2                 1.0                     1                       1
4                 1.0                     1                       1
5                 1.0                     1                       1
```

In [69]:
```python
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 72579 entries, 0 to 74110
Data columns (total 23 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   log_price               72579 non-null  float64
 1   property_type           72579 non-null  object
 2   room_type               72579 non-null  object
 3   amenities               72579 non-null  object
 4   accommodates            72579 non-null  int64
 5   bathrooms               72579 non-null  float64
 6   bed_type                72579 non-null  object
 7   cancellation_policy     72579 non-null  object
 8   cleaning_fee            72579 non-null  bool
 9   first_review            72579 non-null  datetime64[ns]
 10  host_has_profile_pic    72579 non-null  object
 11  host_identity_verified  72579 non-null  object
 12  host_response_rate      72579 non-null  float64
 13  host_since              72579 non-null  datetime64[ns]
 14  instant_bookable        72579 non-null  object
 15  last_review             72579 non-null  datetime64[ns]
 16  latitude                72579 non-null  float64
 17  longitude               72579 non-null  float64
 18  neighbourhood           72579 non-null  object
 19  number_of_reviews       72579 non-null  int64
 20  review_scores_rating    72579 non-null  float64
 21  bedrooms                72579 non-null  float64
 22  beds                    72579 non-null  float64
dtypes: bool(1), datetime64[ns](3), float64(8), int64(2), object(9)
memory usage: 12.8+ MB
```

In [71]:
```python
# Extracting day, month, and year from 'host_since', 'first_review', 'last_review'
data['host_since_day'] = data['host_since'].dt.day
data['host_since_month'] = data['host_since'].dt.month
data['host_since_year'] = data['host_since'].dt.year

data['first_review_day'] = data['first_review'].dt.day
data['first_review_month'] = data['first_review'].dt.month
data['first_review_year'] = data['first_review'].dt.year
```

```
data['last_review_day'] = data['last_review'].dt.day
data['last_review_month'] = data['last_review'].dt.month
data['last_review_year'] = data['last_review'].dt.year

# droped the original date columns as we no longer need them
data.drop(columns=['host_since', 'first_review', 'last_review'], inplace=True)

# Checking the first few rows of the transformed dataset
print(data[['host_since_day', 'host_since_month', 'host_since_year', 'first_review_
```

```
   host_since_day  host_since_month  host_since_year  first_review_day  \
0              26                 3             2012                18   
1              19                 6             2017                 8   
2              25                10             2016                30   
4               3                 1             2015                 5   
5               6                 7             2017                27   

   first_review_month  first_review_year  last_review_day  last_review_month  \
0                   6               2016               18                  7   
1                   5               2017               23                  9   
2                   4               2017               14                  9   
4                  12               2015               22                  1   
5                   8               2017                9                  5   

   last_review_year  
0              2016  
1              2017  
2              2017  
4              2017  
5              2017  
```

In [297…

```
## Converting the 'Cleaning Fee' column into binary format.
```

In [73]:

```
data['cleaning_fee'] = data['cleaning_fee'].apply(lambda x: 1 if x else 0)
```

In [ ]:

```
## Counting the number of amenities for each listing and storing it in the num_amen
```

In [75]:

```
data['num_amenities'] = data['amenities'].apply(lambda x: len(x.split(',')) if isin
```

In [77]:

```
# Droped the original 'amenities' column
data.drop(columns=['amenities'], inplace=True)
```

In [79]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 72579 entries, 0 to 74110
Data columns (total 29 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   log_price               72579 non-null  float64
 1   property_type           72579 non-null  object 
 2   room_type               72579 non-null  object 
 3   accommodates            72579 non-null  int64  
 4   bathrooms               72579 non-null  float64
 5   bed_type                72579 non-null  object 
 6   cancellation_policy     72579 non-null  object 
 7   cleaning_fee            72579 non-null  int64  
 8   host_has_profile_pic    72579 non-null  object 
 9   host_identity_verified  72579 non-null  object 
 10  host_response_rate      72579 non-null  float64
 11  instant_bookable        72579 non-null  object 
 12  latitude                72579 non-null  float64
 13  longitude               72579 non-null  float64
 14  neighbourhood           72579 non-null  object 
 15  number_of_reviews       72579 non-null  int64  
 16  review_scores_rating    72579 non-null  float64
 17  bedrooms                72579 non-null  float64
 18  beds                    72579 non-null  float64
 19  host_since_day          72579 non-null  int32  
 20  host_since_month        72579 non-null  int32  
 21  host_since_year         72579 non-null  int32  
 22  first_review_day        72579 non-null  int32  
 23  first_review_month      72579 non-null  int32  
 24  first_review_year       72579 non-null  int32  
 25  last_review_day         72579 non-null  int32  
 26  last_review_month       72579 non-null  int32  
 27  last_review_year        72579 non-null  int32  
 28  num_amenities           72579 non-null  int64  
dtypes: float64(8), int32(9), int64(4), object(8)
memory usage: 14.1+ MB
```

Encoding Categorical Values

In [81]:

```
# Creating a LabelEncoder object
le = LabelEncoder()

# List of categorical columns to convert
categorical_columns = ['property_type', 'room_type', 'bed_type', 'cancellation_poli
                       'instant_bookable', 'neighbourhood','host_has_profile_pic','

# Loop through the list and apply LabelEncoder to each column
for column in categorical_columns:
    data[column] = le.fit_transform(data[column])

# the categorical columns converted to numbers
print(data.head())  # To check the first few rows
```

```
   log_price  property_type  room_type  accommodates  bathrooms  bed_type  \
0   5.010635              0          0             3        1.0         4   
1   5.129899              0          0             7        1.0         4   
2   4.976734              0          0             5        1.0         4   
4   4.744932              0          0             2        1.0         4   
5   4.442651              0          1             2        1.0         4   

   cancellation_policy  cleaning_fee  host_has_profile_pic  \
0                    2             1                     1   
1                    2             1                     1   
2                    1             1                     1   
4                    1             1                     1   
5                    2             1                     1   

   host_identity_verified  ...  host_since_day  host_since_month  \
0                       1  ...              26                 3   
1                       0  ...              19                 6   
2                       1  ...              25                10   
4                       1  ...               3                 1   
5                       1  ...               6                 7   

   host_since_year  first_review_day  first_review_month  first_review_year  \
0             2012                18                   6               2016   
1             2017                 8                   5               2017   
2             2016                30                   4               2017   
4             2015                 5                  12               2015   
5             2017                27                   8               2017   

   last_review_day  last_review_month  last_review_year  num_amenities  
0               18                  7              2016              9  
1               23                  9              2017             15  
2               14                  9              2017             19  
4               22                  1              2017             12  
5                9                  5              2017             10  

[5 rows x 29 columns]
```

Machine Learning Process

In [83]:
```
X = data.drop(columns=['log_price'])
y = data['log_price']
```

Splitting the dataset into training and testing sets

In [85]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

Feature Scaling

In [87]:
```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Appling **Linear Regression** on Data and Training Model

In [89]:
```
model = LinearRegression()
model.fit(X_train, y_train)
```

Out[89]:
```
▾ LinearRegression
LinearRegression()
```

Linear Regression Model **Prediction**

In [91]:
```
y_pred = model.predict(X_test)
```

Linear Regression Model Evaluation

In [93]:
```
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"RMSE: {rmse}")
print(f"MAE: {mae}")
print(f"R2 Score: {r2}")
```

```
RMSE: 0.4524399719374124
MAE: 0.35041023361342527
R2 Score: 0.5220760753713909
```

RMSE (0.4524): This shows the average error in predictions. A lower value means better accuracy.

MAE (0.3504): This indicates how far predictions are from actual values, on average. The lower, the better.

R^2 Score (0.5221): The model explains 52.2% of the variations in the target variable, meaning it captures

some patterns but is not highly precise.

Performance:

The model makes decent predictions, but there's room for improvement. Since the R^2 score isn't very high,

it suggests that other factors might influence the target variable. Trying different features, transformations,

or advanced models could improve accuracy.

Appling **Decision Tree** on Data and training the **Decision Tree model**

In [100...
```
dt_model = DecisionTreeRegressor(random_state=42)
dt_model.fit(X_train_scaled, y_train)
```

Out[100...
```
▾ DecisionTreeRegressor
DecisionTreeRegressor(random_state=42)
```

Making **predictions**

In [102...
```
dt_pred = dt_model.predict(X_test_scaled)
```

Evaluating the Decision Tree Model

In [104...
```
dt_mae = mean_absolute_error(y_test, dt_pred)
dt_rmse = np.sqrt(mean_squared_error(y_test, dt_pred))
dt_r2 = r2_score(y_test, dt_pred)
```

Printing the **Evaluation Metrics**

In [106...
```
print(f"Decision Tree Regression - MAE: {dt_mae}, RMSE: {dt_rmse}, R²: {dt_r2}")
```

```
Decision Tree Regression - MAE: 0.3847448122643979, RMSE: 0.5134184300739961, R²: 0.
3845684636906255
```

MAE (0.3847): On average, predictions were 0.38 units off from actual values.

RMSE (0.5134): Indicates the model's average error, with smaller values being better.

R^2 Score (0.3846): The model explains 38.5% of the variation in the target variable, meaning it captures some patterns but isn’t highly accurate.

Summary:

The model has decent accuracy, but the low R^2 score suggests it may not generalize well.

Appling **Random Forest Regressor** on Data and training the **Random Forest Regressor model**

In [108...
```
rf_model = RandomForestRegressor()
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
```

```
rf_model.fit(X_train_scaled, y_train)
rf_pred = rf_model.predict(X_test_scaled)
```

Evaluating the Random Forest Model

In [112…

```
rf_mae = mean_absolute_error(y_test, rf_pred)
rf_rmse = np.sqrt(mean_squared_error(y_test, rf_pred))
rf_r2 = r2_score(y_test, rf_pred)
print(f"Random Forest - MAE: {rf_mae}, RMSE: {rf_rmse}, R²: {rf_r2}")
```

```
Random Forest - MAE: 0.26660281595888946, RMSE: 0.35889833531871845, R²: 0.699267942
2372716
```

MAE (0.2666): On average, predictions were 0.27 units off from actual values, which is a small error.

RMSE (0.3589): The model's average error is lower compared to previous models, indicating better performance.

R^2 Score (0.6993): The model explains 69.9% of the variation in the target variable, meaning it captures

patterns more effectively than Decision Tree Regression.

Summary:

This model is more accurate and reliable than the Decision Tree model. The higher R^2 score suggests it generalizes well.

Appling **SVM model** on Data and training the **SVM model**

In [114…

```
svm_model = SVR(kernel='rbf')
svm_model.fit(X_train_scaled, y_train)
svm_pred = svm_model.predict(X_test_scaled)
```

Evaluating the SVM model

In [116…

```
svm_mae = mean_absolute_error(y_test, svm_pred)
svm_rmse = np.sqrt(mean_squared_error(y_test, svm_pred))
svm_r2 = r2_score(y_test, svm_pred)
print(f"SVM - MAE: {svm_mae}, RMSE: {svm_rmse}, R²: {svm_r2}")
```

```
SVM - MAE: 0.320667117246401, RMSE: 0.4187897821457001, R²: 0.5905234909936297
```

MAE (0.32): On average, the model's predictions were off by about 0.32 units. This shows that the model's

predictions are relatively accurate, though there's still some room for improvement.

RMSE (0.42): The error is moderate, indicating that the model has a decent performance, though it's not

as precise as other models like Random Forest.

R^2 Score (0.59): The model explains about 59% of the variation in the data. While this is a solid result,

it also suggests that there's still a fair amount of variance in the data that the model couldn't capture.

Performance:

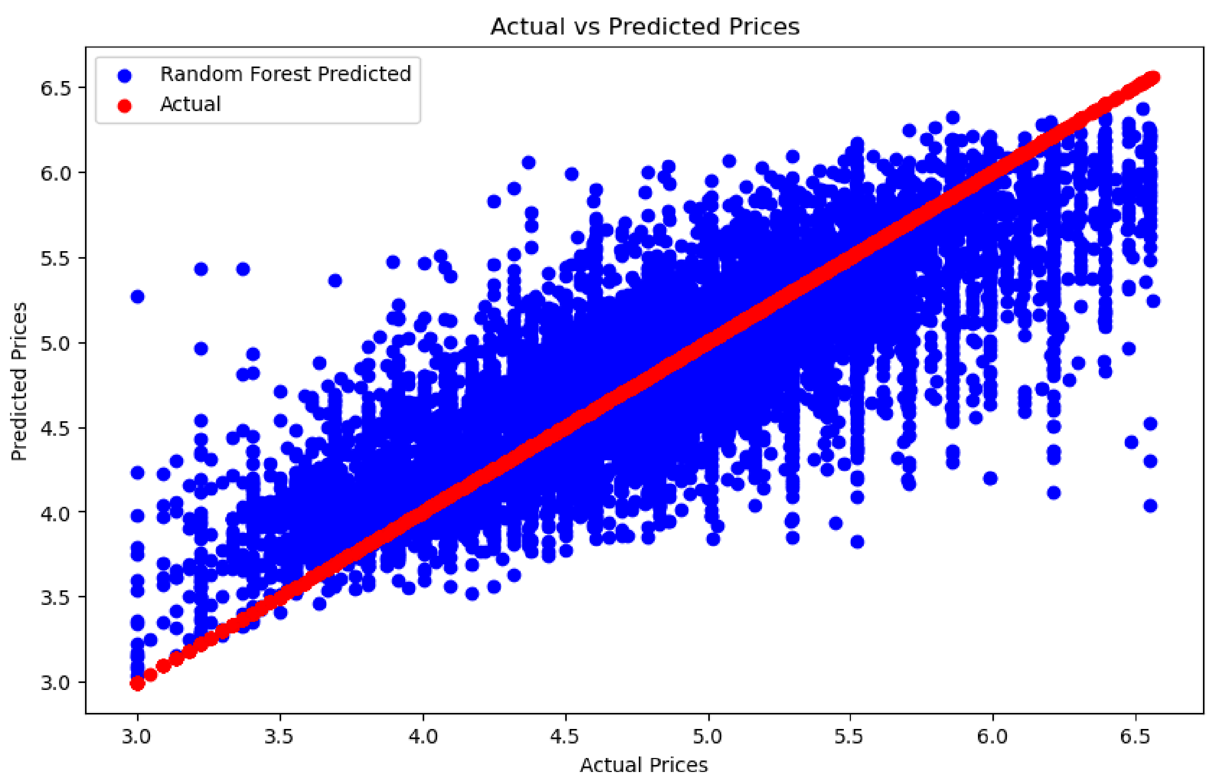
The SVM model is decent and provides reasonable predictions, but it doesn't quite reach the performance

levels of some other models, like Random Forest.

Actual vs Predicted values

In [466…

```
plt.figure(figsize=(10,6))
plt.scatter(y_test, rf_pred, color='blue', label='Random Forest Predicted')
plt.scatter(y_test, y_test, color='red', label='Actual')
plt.title('Actual vs Predicted Prices')
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.legend()
plt.show()
```



In the above scatter plot, the blue dots represent the predicted prices made by our Random Forest model,

while the red dots show the actual prices from the test data.

The red diagonal line represents a perfect prediction, where the predicted values would match the

actual values exactly.

The blue dots indicate how close the model's predictions are to the actual prices. The closer the

blue dots are to the red line, the better the model is at predicting the prices.

MAE (Mean Absolute Error) of **0.27** shows that, on average, the model's predictions are about 0.27

units off from the actual prices. This suggests the model is performing well, with relatively low error.

RMSE (Root Mean Squared Error) of **0.36** shows the average magnitude of the error. A value of 0.36

indicates that, on average, the predicted prices are off by this amount, which is acceptable for this

type of prediction task.

R^2 score of **0.70** means the model explains about 70% of the variation in the prices. While this is a

good result, there’s still some room for improvement as 30% of the variance is not captured by the model.

In [471…

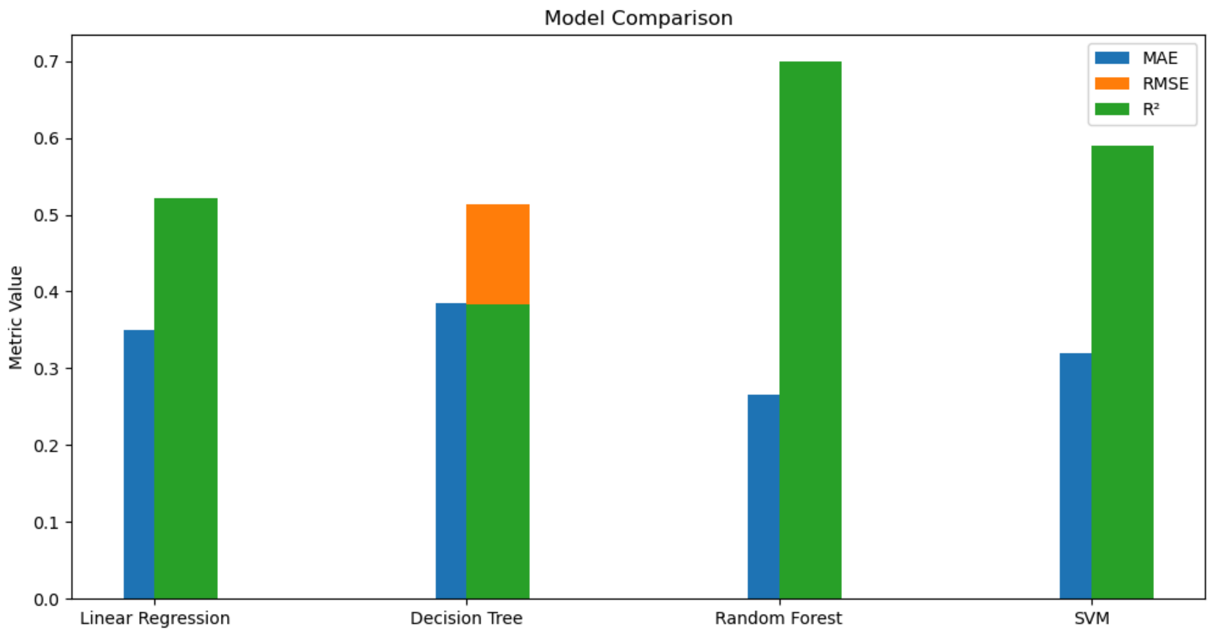
```
# Created a bar plot for model performance
models = ['Linear Regression', 'Decision Tree', 'Random Forest', 'SVM']
mae_values = [0.350, 0.385, 0.266, 0.320]
rmse_values = [0.452, 0.513, 0.358, 0.418]
r2_values = [0.522, 0.384, 0.699, 0.590]

plt.figure(figsize=(12, 6))
x = range(len(models))

# Displaying MAE, RMSE, and R² in a bar plot.
plt.bar(x, mae_values, width=0.2, label='MAE', align='center')
plt.bar(x, rmse_values, width=0.2, label='RMSE', align='edge')
plt.bar(x, r2_values, width=0.2, label='R²', align='edge')

plt.title('Model Comparison')
plt.xticks(x, models)
```

```
plt.ylabel('Metric Value')
plt.legend()
plt.show()
```



In the above bar plot, we're comparing the performance of four machine learning models: **Linear Regression**,

Decision Tree, Random Forest, and **SVM**. The comparison is based on three key performance metrics: **MAE**

(Mean Absolute Error), **RMSE (Root Mean Squared Error)**, and **R^2 (R-squared)**.

MAE (Mean Absolute Error) shows how far off the model's predictions are, on average. A lower value

means better predictions. Random Forest has the lowest MAE, meaning its predictions are closest to

the actual values.

RMSE (Root Mean Squared Error) is another way to measure the prediction error. Similar to **MAE**, lower

values indicate better performance. Again, **Random Forest** does the best here, with the smallest error.

R^2 (R-squared) tells us how well the model explains the variations in the data. A higher value means

the model fits the data better. **Random Forest** leads with the highest **R^2**, means it explains the most

about the data.

What We Can Learn:

Random Forest is the top performer in all three categories: it has the lowest errors and the highest

R^2, making it the most accurate model.

Other models like **Linear Regression** and **SVM** also do a good job but don't perform as well as **Random**

Forest in terms of prediction accuracy and error.

Interpretability:

Making the model easy to understand is crucial so that even non-technical stakeholders can grasp

how it works and what its results mean. This can be done by using simple visualizations and summary

statistics.

Feature Importance (For Random Forest and Decision Tree)

Random Forest and Decision Trees have feature importances, which show which features are the most

important for making predictions.

For non technical person

The code below will show which features (columns) are most important for your predictions. This

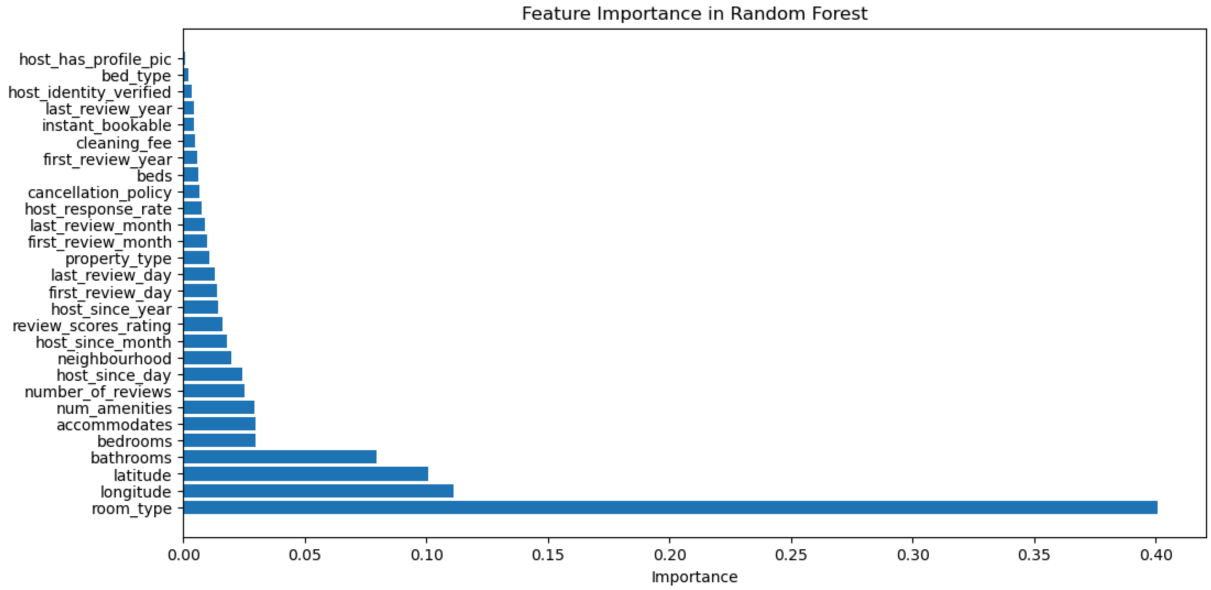
will help non-technical stakeholders understand which features the model is giving more importance to.

In [477…

```
# Extracting feature importance from the trained model
feature_importances = rf_model.feature_importances_

# Sorting the feature importances and plotting them.
indices = np.argsort(feature_importances)[::-1]
sorted_features = np.array(X_train.columns)[indices]

plt.figure(figsize=(12,6))
plt.barh(sorted_features, feature_importances[indices])
plt.title('Feature Importance in Random Forest')
plt.xlabel('Importance')
plt.show()
```



This bar plot helps us understand which features (or columns) in our data are most important when

the Random Forest model makes predictions.

X-axis: The values on the x-axis represent how important each feature is. Higher values mean that

feature has a bigger impact on the model's predictions.

RoomType: The RoomType feature stands out as the most important, with a score of 0.40. This tells

us that the type of room (like an entire apartment vs. a shared room) plays a big role in predicting

the target variable (such as price, demand, or booking).

Longitude and Latitude: Longitude and Latitude are next with importance scores of 0.12 and 0.09.

These geographic features show that where the property is located makes a difference in our model's

predictions, which is not surprising because location is often a key factor in things like price

or demand.

Bathrooms and Bedrooms: The number of Bathrooms and Bedrooms are also important, but not as much as

the location or room type, with scores of 0.08 and 0.03. This shows they have some influence, but

less than some of the other features.

Accommodates: This feature, which likely represents how many people a property can accommodate,

has a lower importance score of 0.03, meaning it plays a smaller role in the predictions.

Other Features: Things like Number of Amenities, Number of Reviews, Host Since Day, and Neighborhood

Popularity have relatively low importance scores (ranging from 0.02 to 0.03). These features have a

smaller impact on the model's predictions.

in short, the RoomType, Longitude, and Latitude are the most important features when it comes to

predicting outcomes in our model. On the other hand, features like Bathrooms, Bedrooms, and even

things like Number of Reviews don't have as much impact on the final prediction. Understanding this

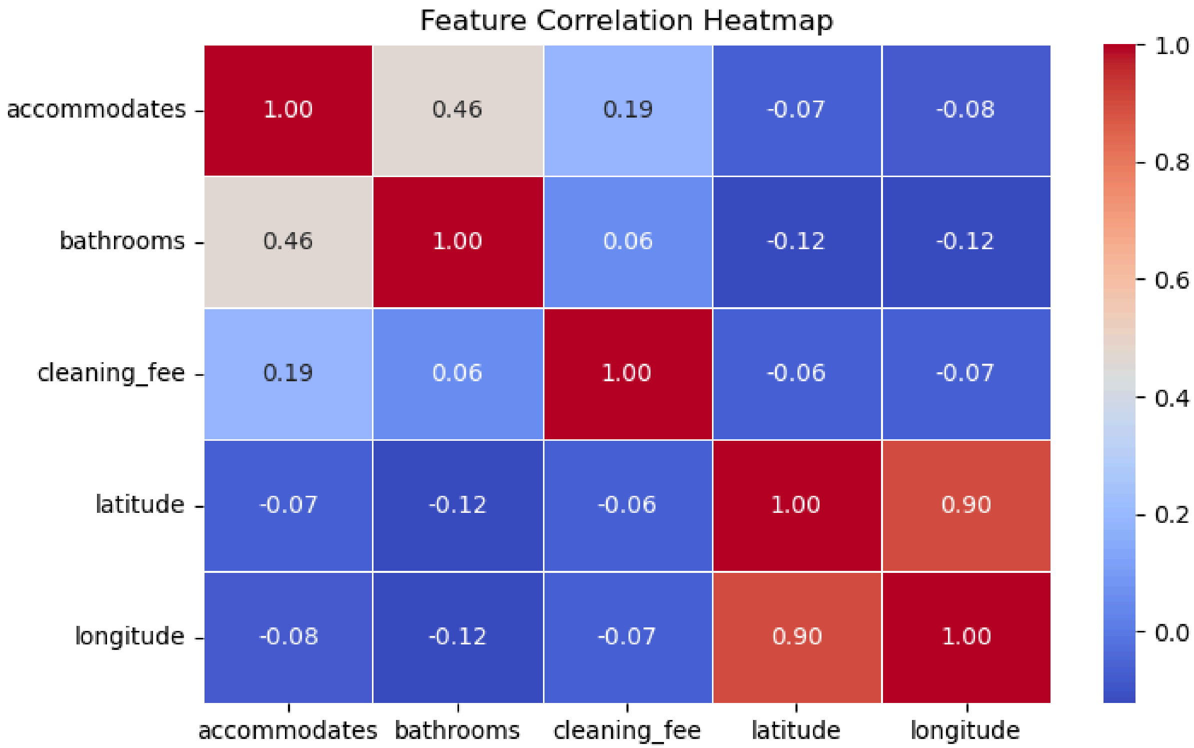
helps us focus on the most influential features, making the model more efficient without losing too

much accuracy.

In [479…

```
# A correlation heatmap for important features
important_features = ['accommodates', 'bathrooms', 'cleaning_fee', 'latitude', 'lon
corr_matrix = data[important_features].corr()

plt.figure(figsize=(8, 5))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)
plt.title('Feature Correlation Heatmap')
plt.show()
```



This heatmap illustrates the relationships between the key features of the dataset, including Accommodates,

Bathrooms, Cleaning Fee, Latitude, and Longitude. Each value in the heatmap shows how strongly two features

are related. Let's break down the findings:

Accommodates and Bathrooms (0.46): There's a moderate positive relationship here, meaning properties that

can accommodate more people generally have more bathrooms. As accommodation capacity increases, so does

the number of bathrooms, but the correlation isn't extremely strong.

Accommodates and Cleaning Fee (0.19): There's a slight positive correlation between the number of people

a property accommodates and its cleaning fee. Larger properties tend to have slightly higher cleaning fees,

though the relationship is not very strong.

Accommodates and Latitude (-0.07), Longitude (-0.08): The relationship between Accommodates and Latitude

or Longitude is weak and negative. This indicates that the property's location has little effect on how many people it can accommodate.

Bathrooms and Cleaning Fee (0.06): There's a very small positive correlation between Bathrooms and Cleaning

Fee, meaning the number of bathrooms has almost no impact on the cleaning fee.

Bathrooms and Latitude (-0.12), Longitude (-0.12): Similar to Accommodates, Bathrooms has a very weak negative

correlation with Latitude and Longitude, suggesting that the number of bathrooms doesn't really change based

on the property's location.

Cleaning Fee and Latitude (-0.06), Longitude (-0.07): The Cleaning Fee has almost no relationship with the

property's location, as shown by the weak negative correlations with Latitude and Longitude.

Latitude and Longitude (0.90): This is a strong positive correlation, which makes sense since Latitude and

Longitude are both geographical coordinates. Properties located in the same area will have very similar

latitude and longitude values.

The Accommodates and Bathrooms features have a moderate positive relationship, meaning larger properties with

more accommodation often have more bathrooms.

Latitude and Longitude are highly correlated, as expected, because they both represent geographical location.

There are weak correlations between features like Accommodates and Cleaning Fee, and Bathrooms with the

property's location, suggesting that these factors don't have a significant influence on each other.