ML PROJECT REPORT

Task 1: Data Exploration

This code imports libraries for data manipulation and visualization. Then, it loads a dataset named 'Gurgaon_RealEstate.csv' into a DataFrame called 'df'.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
df = pd.read_csv('Gurgaon_RealEstate.csv')
```

This code below prints information about the DataFrame 'df', including the features (columns) and their corresponding data types.

```
# Identify features and data types
print(df.info())
```

the output is :-

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3803 entries, 0 to 3802
Data columns (total 23 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   property_type        3803 non-null   object 
 1   society              3802 non-null   object 
 2   sector               3803 non-null   object 
 3   price                3785 non-null   float64
 4   price_per_sqft       3785 non-null   float64
 5   area                 3785 non-null   float64
 6   areaWithType         3803 non-null   object 
 7   bedRoom              3803 non-null   int64  
 8   bathroom             3803 non-null   int64  
 9   balcony              3803 non-null   object 
 10  floorNum             3784 non-null   float64
 11  facing               2698 non-null   object 
 12  agePossession        3803 non-null   object 
 13  super_built_up_area  1915 non-null   float64
 14  built_up_area        1733 non-null   float64
 15  carpet_area          1944 non-null   float64
 16  study room           3803 non-null   int64  
 17  servant room         3803 non-null   int64  
 18  store room           3803 non-null   int64  
 19  pooja room           3803 non-null   int64  
```

```
 20   others                  3803 non-null   int64
 21   furnishing_type         3803 non-null   int64
 22   luxury_score            3803 non-null   int64
dtypes: float64(7), int64(9), object(7)
memory usage: 683.5+ KB

None
```

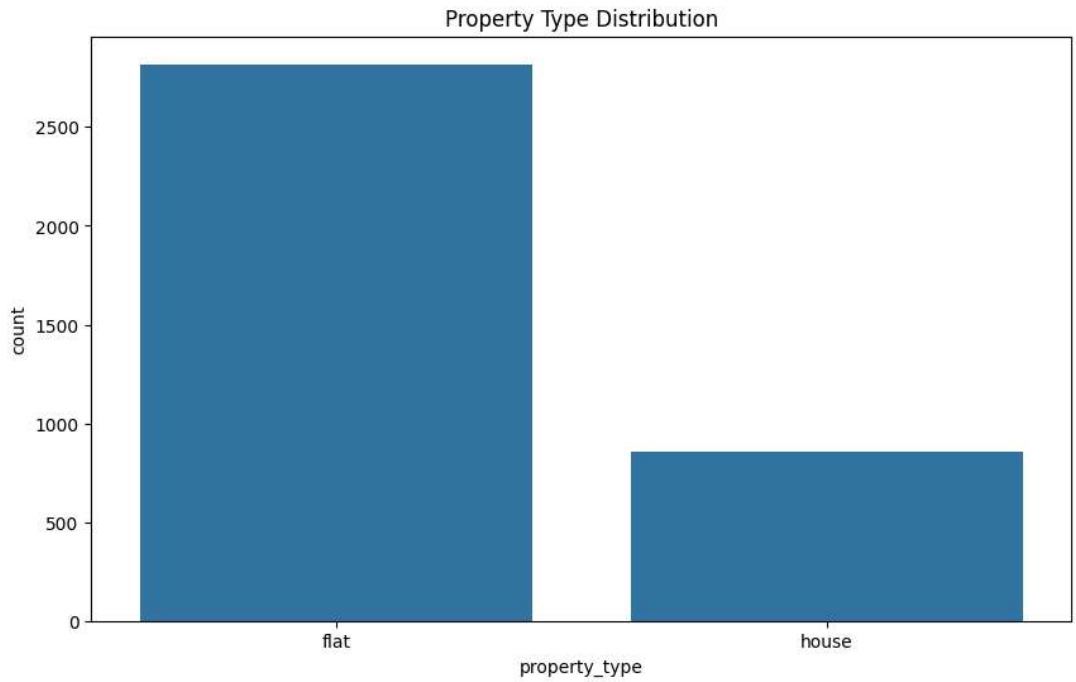
This code below removes any duplicate rows from the DataFrame 'df'. By setting `inplace=True`, the changes are applied directly to the DataFrame, meaning it modifies the original DataFrame rather than creating a copy with duplicate rows removed. This ensures that the DataFrame 'df' now contains only unique rows, which helps in reducing bias and maintaining the integrity of the data for further analysis.

```
# Remove duplicate rows
df.drop_duplicates(inplace=True)
```

This code below creates a count plot using seaborn to visualize the distribution of different property types in the dataset. The `x='property_type'` parameter specifies that the property type column should be used for the x-axis of the plot. The `data=df` parameter indicates that the data for the plot is taken from the DataFrame 'df'. The plot is displayed using matplotlib.pyplot with a specified figure size and a title.

```
# Explore property_type column
plt.figure(figsize=(10, 6))
sns.countplot(x='property_type', data=df)
plt.title('Property Type Distribution')
plt.show()
```

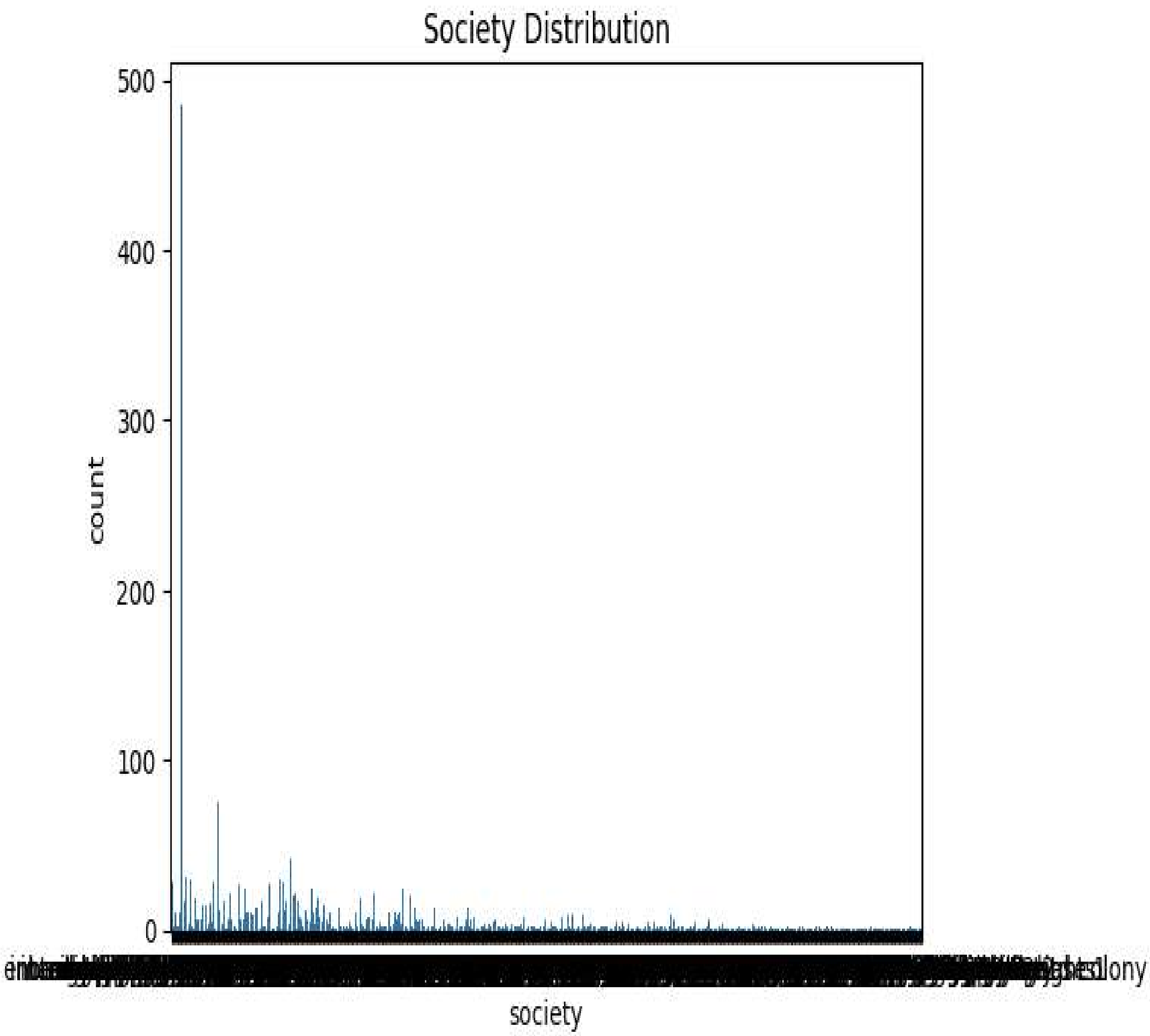
output :-



This code below first visualizes the distribution of the 'society' column using a count plot. It then calculates the frequency of each society and identifies the societies with fewer than 3 flats/houses. These societies are stored in the variable 'societies_to_remove'. Finally, it removes the rows corresponding to these societies from the DataFrame 'df' using boolean indexing with the `isin()` function.

```
# Remove societies with less than 3 flats/houses
sns.countplot(x='society', data=df)
plt.title('Society Distribution')
plt.show()
society_freq = df['society'].value_counts()
societies_to_remove = society_freq[society_freq < 3].index
df = df[~df['society'].isin(societies_to_remove)]
```

the output :-



The following code says :-

1. Price Column Summary Statistics:

   - This code prints summary statistics for the 'price' column, including count, mean, standard deviation, minimum, 25th percentile (Q1), median (50th percentile), 75th percentile (Q3), and maximum value.

2. Price Histogram:

   - A histogram is created using seaborn to visualize the distribution of prices. The histogram displays the frequency of different price ranges. The `kde=True` parameter adds a kernel density estimate line to the plot.

3. Price Box Plot:

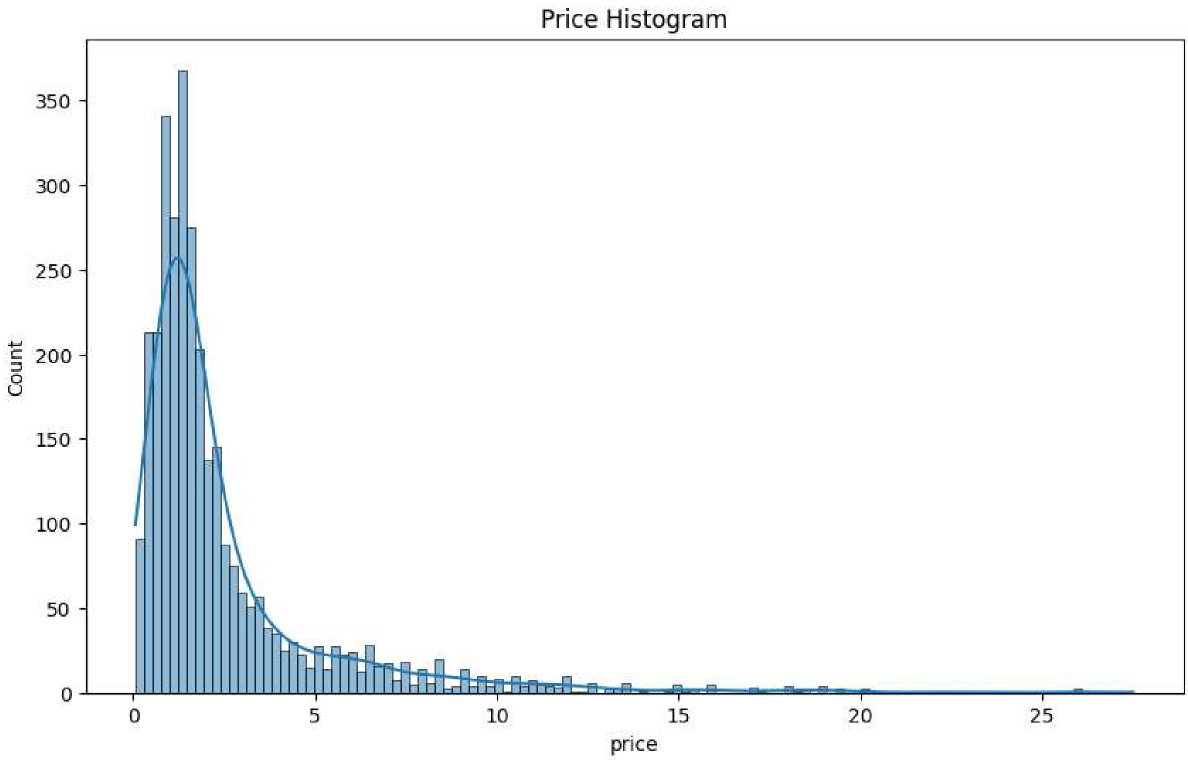
   - A box plot is generated using seaborn to visualize the distribution of prices. The box plot displays the median, quartiles, and potential outliers in the data.

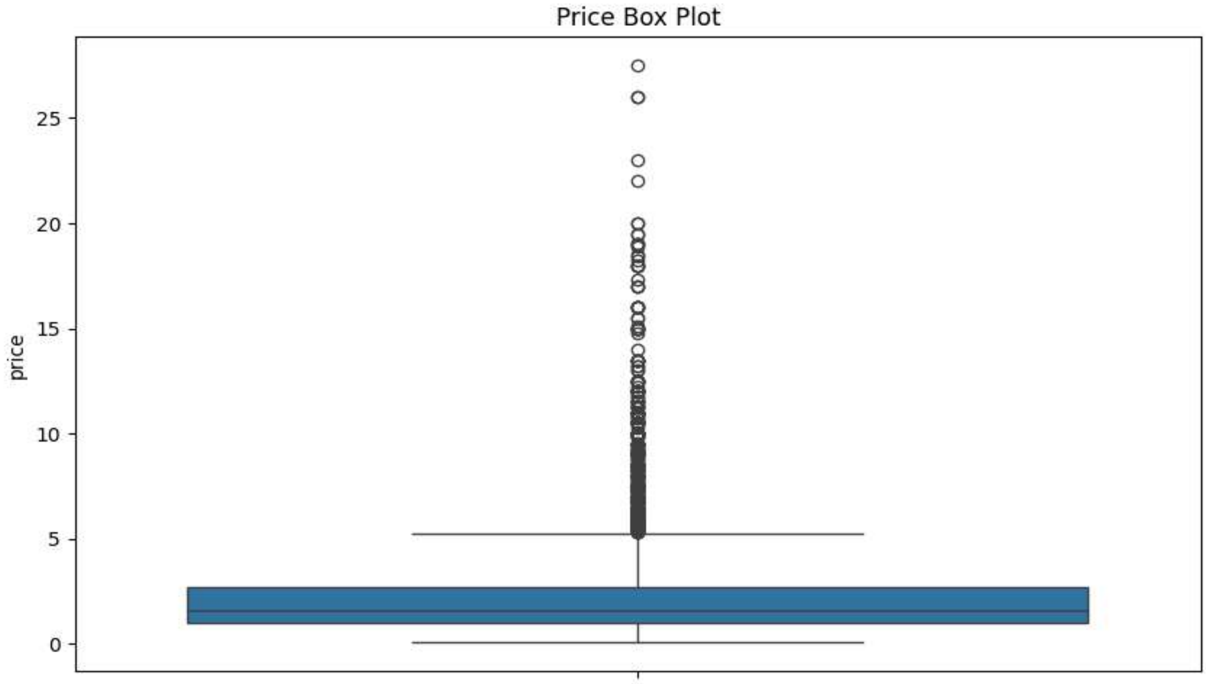
4. Price Skewness and Kurtosis:

   - The code calculates the skewness and kurtosis of the 'price' column. Skewness measures the asymmetry of the distribution, with a skewness of 0 indicating a symmetric distribution. Kurtosis measures the peakedness of the distribution, with a kurtosis of 3 indicating a normal distribution. Positive values indicate heavier tails, while negative values indicate lighter tails compared to a normal distribution.

```
# Explore price column
print("Price Column Summary Statistics:")
print(df['price'].describe())

plt.figure(figsize=(10, 6))
sns.histplot(df['price'], kde=True)
plt.title('Price Histogram')
plt.show()

plt.figure(figsize=(10, 6))
sns.boxplot(y='price', data=df)
plt.title('Price Box Plot')
plt.show()

print("Price Skewness:", df['price'].skew())
print("Price Kurtosis:", df['price'].kurt())
```

```
output :-
Price Column Summary Statistics:
count    3167.000000
mean        2.520249
std         2.903951
min         0.070000
25%         0.970000
50%         1.550000
75%         2.700000
max        27.500000
Name: price, dtype: float64
```






```
Price Skewness: 3.1517923067840323
Price Kurtosis: 13.258233016312998
```

1. Bathroom Summary Statistics:

- This code prints summary statistics for the 'bathroom' column, including count, mean, standard deviation, minimum, 25th percentile (Q1), median (50th percentile), 75th percentile (Q3), and maximum value. These statistics provide insights into the distribution and central tendency of the number of bathrooms in the dataset.

2. Bathroom Histogram:

- A histogram is created using seaborn to visualize the distribution of the number of bathrooms. The histogram displays the frequency of different bathroom counts. Each bar represents a range of bathroom counts, and the height of the bar corresponds to the frequency of properties falling within that range. The `kde=True` parameter adds a kernel density estimate line to the plot, which represents the smooth approximation of the distribution's probability density function.

The histogram allows us to:

- Understand the central tendency and spread of the bathroom counts.

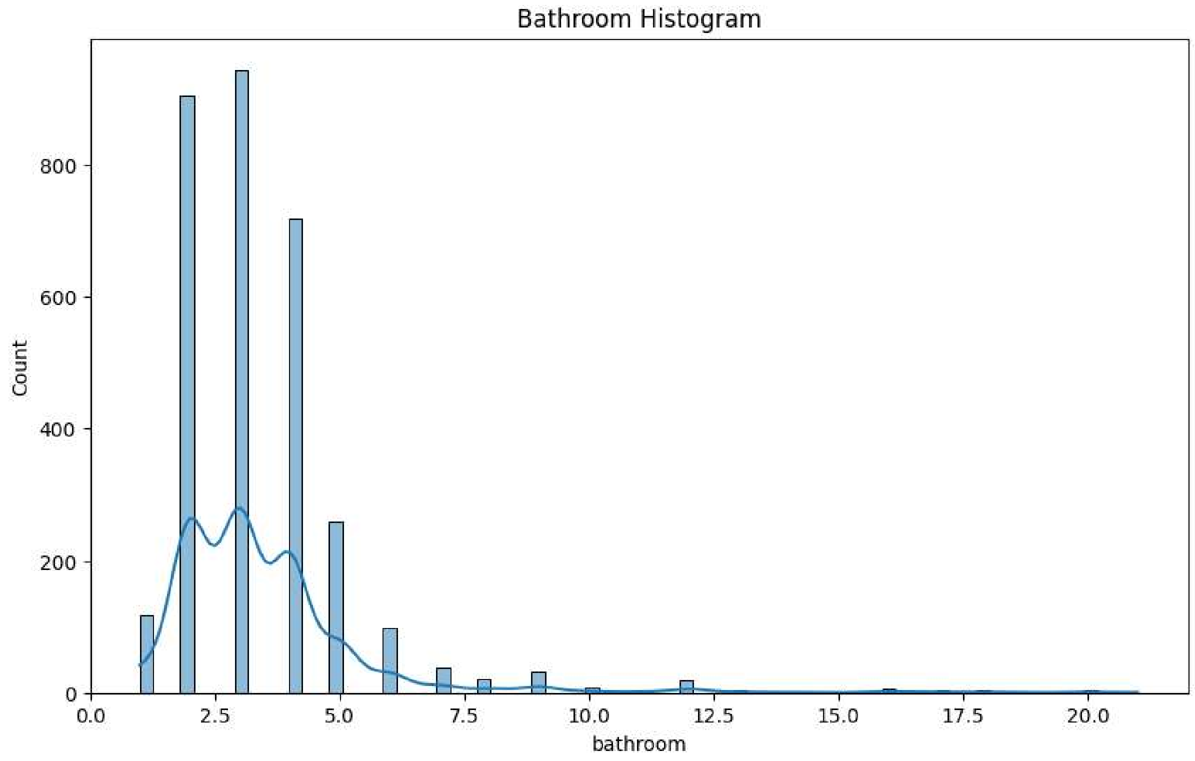
- Identify any outliers or unusual patterns in the distribution.

- Assess the overall shape of the distribution, including its skewness and kurtosis.

```
# Explore other columns (e.g., bathroom, bedroom, balcony)
print("Bathroom Summary Statistics:")
print(df['bathroom'].describe())

plt.figure(figsize=(10, 6))
sns.histplot(df['bathroom'], kde=True)
plt.title('Bathroom Histogram')
plt.show()
```

the output :-

```
Bathroom Summary Statistics:
count    3181.000000
mean        3.430053
std         1.927106
min         1.000000
25%         2.000000
50%         3.000000
75%         4.000000
max        21.000000
Name: bathroom, dtype: float64
```



1. Price vs Area Scatter Plot:

- This code creates a scatter plot using seaborn to visualize the relationship between the area and price of real estate properties. The x-axis represents the area of the properties, while the y-axis represents the corresponding prices. Each point in the plot represents an individual property, and its position reflects both its area and price. This visualization helps in understanding how the price of properties varies with their size.

2. Property Type vs Price Box Plot:

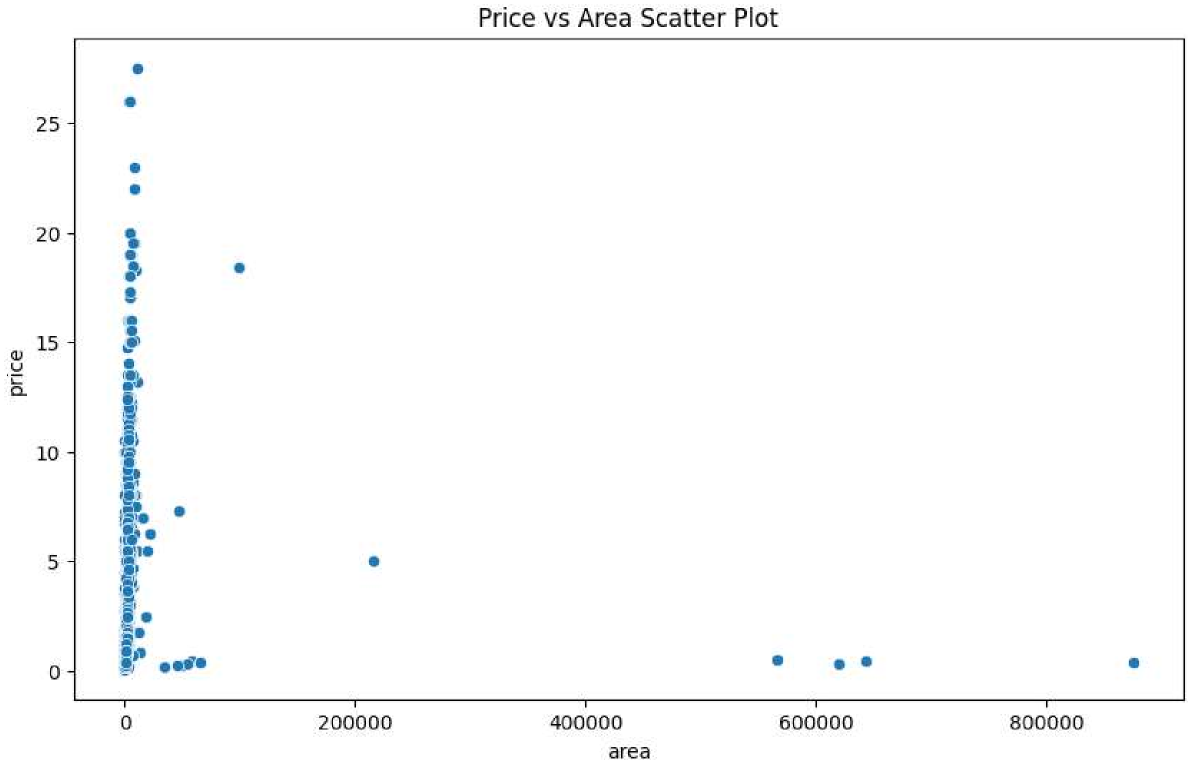
- Another seaborn box plot is generated to analyze the relationship between property types and prices. The x-axis represents different property types, and the y-axis represents the corresponding prices. The box plot displays the median, quartiles, and potential outliers in price distribution for each property type. This visualization allows for the comparison of price distributions across different property types.

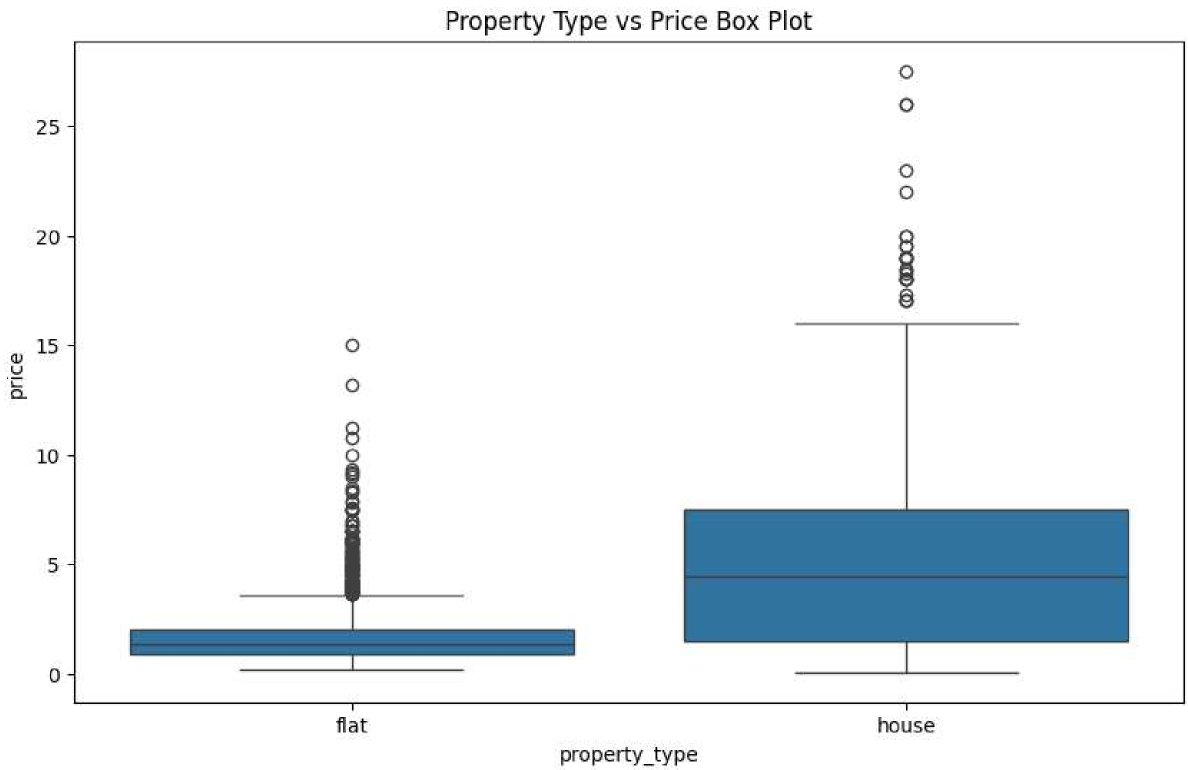
3. Missing Values Distribution:

- This code prints the distribution of missing values across all columns in the DataFrame 'df'. It calculates the sum of missing values for each column using the `isnull().sum()` method. This information helps in identifying columns with missing data, which is crucial for data cleaning and preprocessing. Identifying missing values is the first step towards handling them appropriately in the dataset.

```
# Multivariate analysis
plt.figure(figsize=(10, 6))
sns.scatterplot(x='area', y='price', data=df)
plt.title('Price vs Area Scatter Plot')
plt.show()

plt.figure(figsize=(10, 6))
sns.boxplot(x='property_type', y='price', data=df)
plt.title('Property Type vs Price Box Plot')
plt.show()

# Check for missing values
print("Missing Values Distribution:")
print(df.isnull().sum())
```

output :-






```
Missing Values Distribution:
property_type      0
society            1
sector             0
price             14
price_per_sqft    14
area              14
```

```
areaWithType            0
bedRoom                 0
bathroom                0
balcony                 0
floorNum               16
facing                844
agePossession           0
super_built_up_area  1465
built_up_area        1714
carpet_area          1588
study room              0
servant room            0
store room              0
pooja room              0
others                  0
furnishing_type         0
luxury_score            0
dtype: int64
```

TASK2:-

This process calculates and prints the number of missing entries in each column that contains missing values:

1. Calculate Missing Values:

   - The number of missing values in each column is calculated.

2. Filter Columns with Missing Values:

   - From the total missing values, only the columns with more than zero missing entries are filtered out.

3. Print the Results:

   - A message is printed to indicate the allocation of empty values according to their features, followed by the actual counts of missing values for each relevant column.

```python
empty_entries = df.isnull().sum()
empty_entries_distribution = empty_entries[empty_entries > 0]
print("allocating the empty values accoring to their features")
print(empty_entries_distribution)
# to calculate the empty entries in each column and printing it
```

Output:-

```
allocating the empty values accoring to their features
society                 1
price                  14
price_per_sqft         14
```

```
area                    14
floorNum                16
facing                 844
super_built_up_area   1465
built_up_area         1714
carpet_area           1588
dtype: int64
```

This code calculates and visualizes the percentage of missing (empty) entries for each feature in the DataFrame. Here's a detailed explanation:

1. Calculate the Percentage of Empty Entries:

- The percentage of missing values for each feature is calculated by dividing the number of missing entries in each column by the total number of rows in the DataFrame, then multiplying by 100.

2. Create a Summary DataFrame:

- A new DataFrame, `empty_summary`, is created to store the number of empty entries and their corresponding percentages for each feature.

3. Filter the Summary DataFrame:

- The summary DataFrame is filtered to include only the features that have missing entries.

4. Print the Summary:

- The summary DataFrame is printed to show the number of empty entries and their percentages for each feature with missing data.

5. Visualize the Percentage of Empty Entries:

- A bar plot is created using seaborn to visualize the percentage of missing entries for each feature with missing data. The x-axis represents the feature names, and the y-axis represents the percentage of missing values. The features are rotated for better readability.

This visualization helps identify which features have significant amounts of missing data and can guide decisions on how to handle them.

1. Calculate the percentage of missing entries for each feature.

2. Create a DataFrame to summarize the number of empty entries and their percentage for each feature.

3. Filter the summary DataFrame to show only features with missing entries.

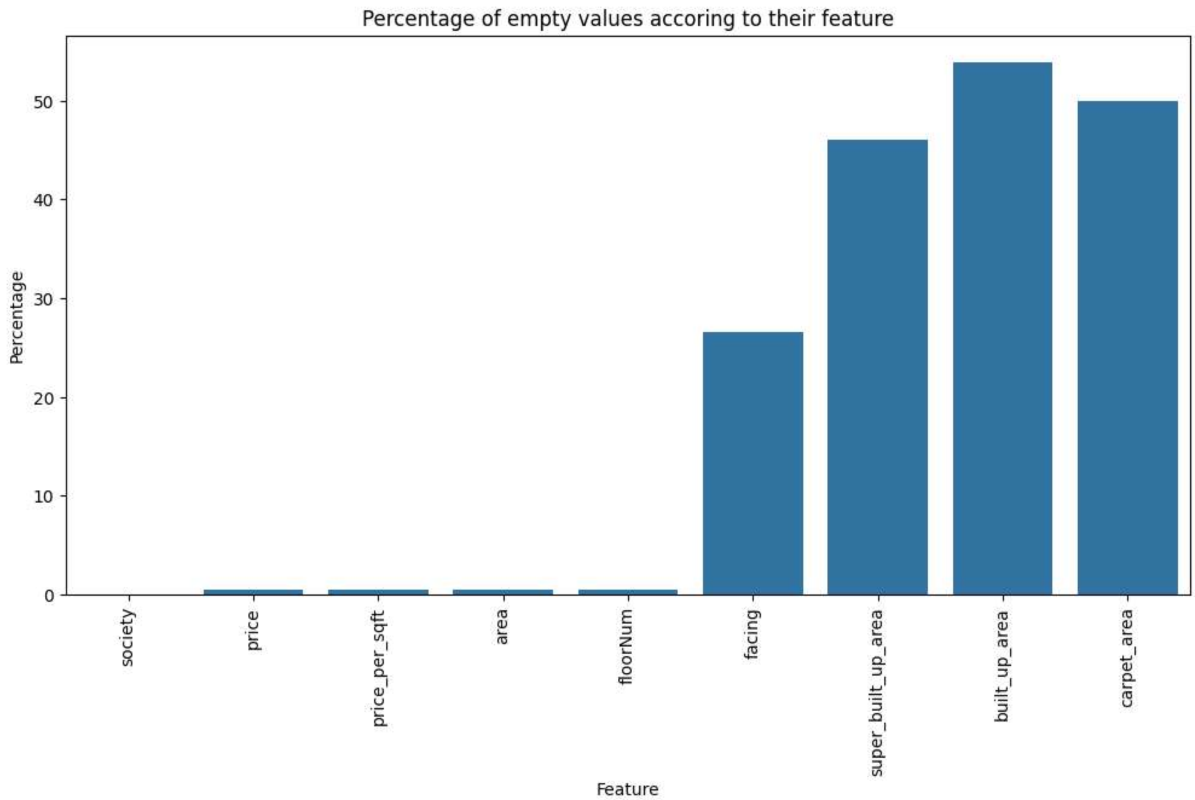
4. Print the summary DataFrame to display the number of empty entries and their percentages.

5. Create a bar plot to visualize the percentage of missing entries for each feature with missing data. The x-axis shows the feature names, and the y-axis shows the percentage of missing values, with the feature names rotated for better readability.

```
# Calculate the percentage of empty entries for each feature
empty_percentage = (empty_entries / len(df)) * 100
# Create a DataFrame to summarize the empty entries and their
distribution
empty_summary = pd.DataFrame({'Empty Entries': empty_entries,
'Percentage': empty_percentage})
# Filter the summary to show only features with empty entries
empty_summary = empty_summary[empty_summary['Empty Entries'] > 0]
print(empty_summary)
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=(12, 6))
sns.barplot(x=empty_summary.index, y=empty_summary['Percentage'])
plt.xticks(rotation=90)
plt.title('Percentage of empty values accoring to their feature')
plt.ylabel('Percentage')
plt.xlabel('Feature')
plt.show()
```

output:-

```
                     Empty Entries  Percentage
society                          1    0.031437
price                           14    0.440113
price_per_sqft                  14    0.440113
area                            14    0.440113
floorNum                        16    0.502986
facing                         844   26.532537
super_built_up_area           1465   46.054700
built_up_area                 1714   53.882427
carpet_area                   1588   49.921408
```



Here's a detailed explanation of what the below code does:

1. Import Necessary Libraries:

   - Import pandas for data manipulation and the KNNImputer from scikit-learn for imputation.

2. Select Numerical Data:

   - Filter the DataFrame `df` to select only the numerical columns, creating `numeric_data`.

3. Initialize KNN Imputer:

   - Create an instance of `KNNImputer` with `n_neighbors=5`, which means the imputer will use the 5 nearest neighbors to fill missing values.

4. Impute Missing Values:

   - For each numerical column, create a temporary DataFrame `temp_df` containing just that column.

   - Apply the KNN imputer to `temp_df` to generate `imputed_values`.

- Replace the original column in `df` with these imputed values.

5. Check for Remaining Missing Values:

- Calculate the number of remaining missing entries in each column of `df`.
- Filter to show only columns that still have missing entries after the imputation.
- Print the distribution of these remaining missing entries, if any.

.

```
# Using KNN to fill missing values
import pandas as pd
from sklearn.impute import KNNImputer
numeric_data = df.select_dtypes(include='number')
knn = KNNImputer(n_neighbors=5)
for col in numeric_data.columns:
    temp_df = pd.DataFrame(df[col])
    imputed_values = knn.fit_transform(temp_df)
    df[col] = imputed_values
#we used knn only to fill missing values on numerical columns
#just to print the empty values / entires , which arent filled using
KNN
empty_entries = df.isnull().sum()
empty_entries_distribution = empty_entries[empty_entries > 0]
print("allocating the empty values accoring to their features")
print(empty_entries_distribution)
```

output:-

```
allocating the empty values accoring to their features
society       1
facing      844
dtype: int64
```

Here's a detailed explanation of what the below code does:

1. Import Necessary Libraries:

   - Import pandas for data manipulation.

2. Convert Categorical Data to Numerical Data:

   - Convert the 'facing' column in the DataFrame `df` from a categorical data type to numerical codes.

    - The `.astype('category')` method changes the 'facing' column to a categorical type.

    - The `.cat.codes` attribute then converts these categories to numerical codes.

  - The transformed numerical codes are assigned back to the 'facing' column in `df`.

3. Store Encoded Data:

  - The encoded 'facing' column is stored in a variable called `facing_encoded`.

4. Display Encoded Data:

  - The variable `facing_encoded` now contains the numerical representation of the 'facing' column.

```
#here we are changeing categotical data into numerical data
import pandas as pd
df['facing'] = df['facing'].astype('category').cat.codes
facing_encoded = df['facing']
facing_encoded
```

output:-

```
0    -1
1    -1
2    -1
3    -1
4     3
    ..
3795  -1
3796   0
3798   3
3799   6
3801   0
Name: facing, Length: 3181, dtype: int8
```

Here's a detailed explanation of what the below code does:

1. Label Encode 'facing' Column:

  - The `LabelEncoder` from scikit-learn is used to convert the categorical 'facing' column into numerical labels. This encoding assigns a unique integer to each category.

  - The `fit_transform` method is applied to the 'facing' column, which converts the categorical data to numerical codes.

2. Divide the Data into Missing and Non-missing Parts:

- The DataFrame `df` is split into two parts: `missing_values` containing rows where the 'facing' value is missing, and `non_missing_values` containing rows where the 'facing' value is present.

3. Train the Model Using Non-missing Values:

- The numeric columns from `non_missing_values` are selected, excluding the 'facing' column.

- `X_train` is created by dropping the 'facing' column from the numeric columns.

- `y_train` is set as the 'facing' column from `non_missing_values`.

- A `RandomForestClassifier` model is instantiated with 100 estimators and a random state of 42 for reproducibility.

- The model is trained using `X_train` and `y_train`.

4. Predict Missing Values:

- If there are any missing 'facing' values, the corresponding features (`X_missing`) are selected from `missing_values`.

- A `SimpleImputer` with the strategy 'most_frequent' is used to fill any remaining missing values in `X_missing` with zeros (as a temporary measure).

- The trained model is used to predict the 'facing' values for these rows, and the predicted values are assigned back to the 'facing' column in `missing_values`.

5. Combine the DataFrames:

- Finally, the `missing_values` and `non_missing_values` DataFrames are concatenated to form the complete DataFrame `n_filled` with no missing 'facing' values.

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
# Assuming df is your DataFrame

# 1. Label Encode 'facing' column
label_encoder = LabelEncoder()
df['facing'] = label_encoder.fit_transform(df['facing'].astype(str))
```

```
# 2 now lets divide the data into 2 paerts , missing and non-missing
missing_values = df[df['facing'].isnull()]
non_missing_values = df[df['facing'].notnull()]

# 3 now we need to train the model by suing non-missing values
numeric_columns = non_missing_values.select_dtypes(include='number')
X_train = numeric_columns.drop(columns=['facing'])
y_train = non_missing_values['facing']
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# 4 once the model is trained predict the missing values
if not missing_values.empty:
    X_missing = missing_values[numeric_columns.columns]
    if not X_missing.empty:
        imputer = SimpleImputer(strategy='most_frequent')
        imputed_values = imputer.fit_transform(X_missing.fillna(0))  #
Fill any remaining missing values with zeros
        missing_values['facing'] = model.predict(imputed_values)

# 5) at the end fill the missing values
n_filled = pd.concat([missing_values, non_missing_values])
```

Here's a detailed explanation of what the below code does:

1. Import Necessary Libraries:

- Import pandas for data manipulation.

2. Calculate the Mode of the 'society' Column:

- The `mode()` function is used to find the most frequently occurring value (mode) in the 'society' column. The `[0]` index is used to extract the mode value from the result, as `mode()` returns a Series.

3. Replace Missing Values with the Mode:

- The `fillna()` method is applied to the 'society' column, replacing all missing values with the calculated mode (`mode_society`).

4. Print the Updated DataFrame:

This process ensures that the missing values in the 'society' column are filled with the most common value, which is a simple and effective imputation method for categorical data.

```
# here we are finding the mode of the 'society' column and then replace the mode with missing values
import pandas as pd
mode_society = df['society'].mode()[0]
df['society'] = df['society'].fillna(mode_society)
print(df)
```

output:-

```
  property_type                         society       sector  price  \
0          flat          signature global park 4    sector 36   0.82
1          flat                 smart world gems    sector 89   0.95
2          flat                    pyramid elite    sector 86   0.46
3          flat             breez global hill view  sohna road   0.32
4          flat        bestech park view sanskruti   sector 92   1.60
...         ...                              ...         ...    ...
3795       flat                 eldeco accolade  sohna road   0.87
3796       flat                       paras dews  sector 106   0.92
3798       flat                   pivotal devaan   sector 84   0.37
3799      house  international city by sobha phase 1  sector 109   6.00
3801      house                      independent   sector 43  15.50

     price_per_sqft     area  \
0          7585.0   1081.0
1          8600.0   1105.0
2            79.0  58228.0
3          5470.0    585.0
4          8020.0   1995.0
...           ...      ...
3795       5965.0   1459.0
3796       6642.0   1385.0
3798       6346.0    583.0
3799       9634.0   6228.0
3801      28233.0   5490.0

                                    areaWithType  bedRoom  bathroom  \
0    Super Built up area 1081(100.43 sq.m.)Carpet a...      3.0       2.0
1                  Carpet area: 1103 (102.47 sq.m.)      2.0       2.0
2                 Carpet area: 58141 (5401.48 sq.m.)      2.0       2.0
3    Built Up area: 1000 (92.9 sq.m.)Carpet area: 5...      2.0       2.0
4    Super Built up area 1995(185.34 sq.m.)Built Up...      3.0       4.0
...                                              ...      ...       ...
3795  Super Built up area 1457(135.36 sq.m.)Carpet a...      2.0       2.0
3796  Super Built up area 1385(128.67 sq.m.)Built Up...      2.0       2.0
3798  Super Built up area 583(54.16 sq.m.)Carpet are...      2.0       2.0
3799                  Plot area 692(578.6 sq.m.)      5.0       5.0
3801                  Plot area 610(510.04 sq.m.)      5.0       6.0

  balcony  ...  super_built_up_area  built_up_area  carpet_area  \
0       2  ...          1081.000000    2443.657825   650.000000
1       2  ...          1922.293945    2443.657825  1103.000000
2       1  ...          1922.293945    2443.657825 58141.000000
3       1  ...          1922.293945    1000.000000   585.000000
4      3+  ...          1995.000000    1615.000000  1476.000000
```

```
...    ... ...          ...         ...         ...
3795    3+ ...     1457.000000  2443.657825   849.000000
3796    3+ ...     1385.000000   940.000000   845.000000
3798     1 ...      583.000000  2443.657825   483.000000
3799    3+ ...     1922.293945  6228.000000  2689.945937
3801     3 ...     1922.293945  5490.000000  2689.945937

      study room  servant room  store room  pooja room  others  \
0            0.0           0.0         0.0         0.0     0.0
1            1.0           1.0         0.0         0.0     0.0
2            0.0           0.0         0.0         0.0     0.0
3            0.0           0.0         0.0         0.0     0.0
4            0.0           1.0         0.0         0.0     1.0
...          ...           ...         ...         ...     ...
3795         1.0           0.0         0.0         0.0     0.0
3796         0.0           0.0         0.0         0.0     0.0
3798         0.0           0.0         0.0         0.0     0.0
3799         1.0           1.0         1.0         1.0     0.0
3801         1.0           1.0         1.0         1.0     0.0

      furnishing_type  luxury_score
0                 0.0           8.0
1                 0.0          38.0
2                 0.0          15.0
3                 0.0          49.0
4                 1.0         174.0
...               ...           ...
3795              0.0          72.0
3796              0.0         174.0
3798              0.0          73.0
3799              0.0         160.0
3801              0.0          76.0

[3181 rows x 23 columns]
```

Here's a detailed explanation of the code below:-

1. Calculate Missing Values:

- The code uses the `isnull()` method on the DataFrame `df` to identify missing values. This method returns a DataFrame of the same shape as `df` with `True` for missing values and `False` for non-missing values.

- The `sum()` method is then applied to this DataFrame to count the number of `True` values in each column, resulting in the total number of missing entries per column.

2. Filter Columns with Missing Values:

- From the total missing values calculated in the previous step, the code filters out the columns that have zero missing entries. This is done by selecting only those columns where the count of missing entries is greater than zero.

- The result is stored in the variable `empty_entries_distribution`.

3. Print the Distribution of Missing Values:

- A message is printed to indicate the allocation of empty values according to their features.

- The code then prints the `empty_entries_distribution`, which shows the number of remaining missing entries for each column that still contains missing values.

```
# here we are printing the empty enetires which are left
empty_entries = df.isnull().sum()
empty_entries_distribution = empty_entries[empty_entries > 0]
print("allocating the empty values accoring to their features")
print(empty_entries_distribution)
```

0s

Output:-

```
allocating the empty values accoring to their features
society                   1
price                    18
price_per_sqft           18
area                     18
floorNum                 19
facing                 1105
super_built_up_area    1888
built_up_area          2070
carpet_area            1859
dtype: int64
```

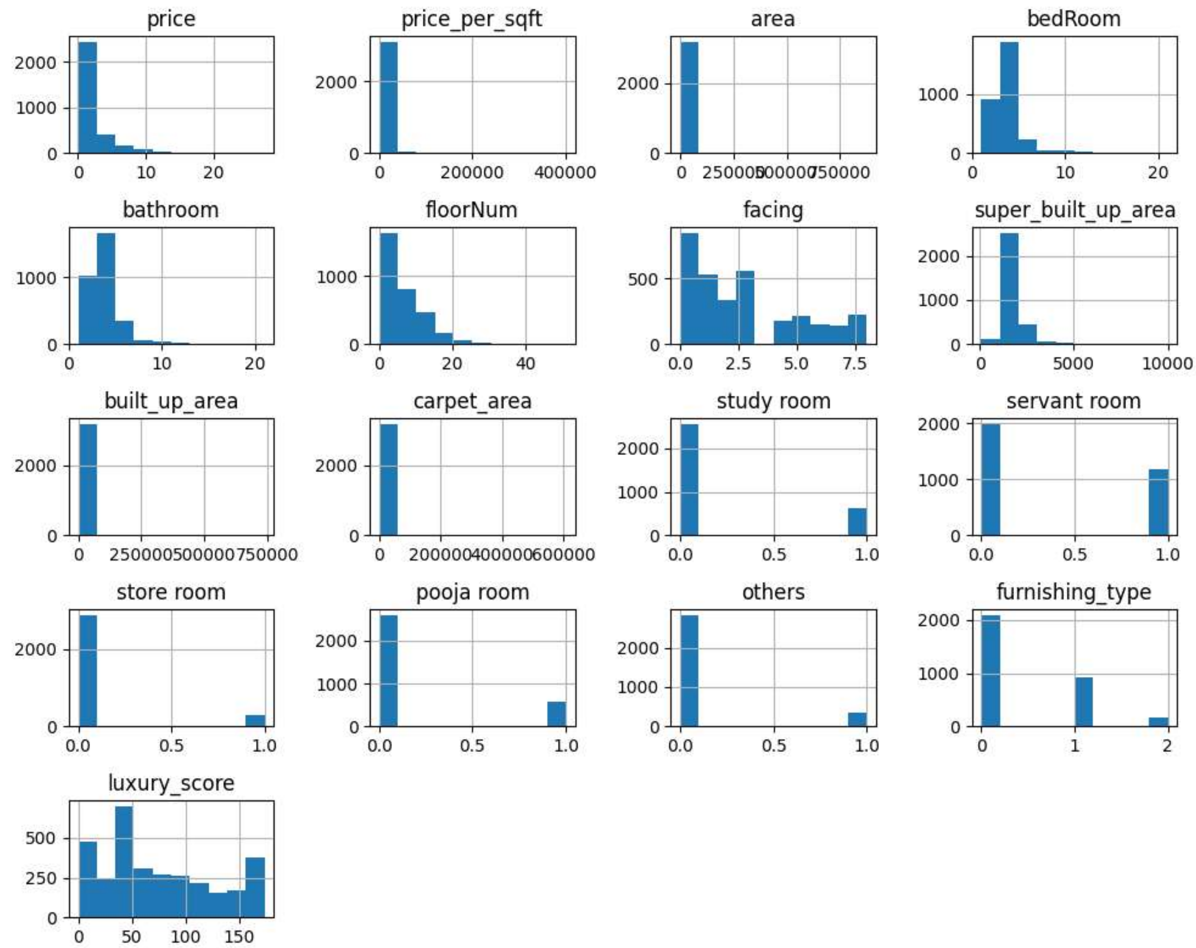
TASK-3:-

```
# Task 3
# here we are suing z-score method to calculate the outliers
import numpy as np
num_columns = df.select_dtypes(include=np.number).columns
outliers_z_score = []
for col in num_columns:
    z_scores = np.abs((df[col] - df[col].mean()) / df[col].std())
    outliers_z_score.extend(df[z_scores > 3].index)

outliers_iqr = []
for col in num_columns:
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
    IQR = Q3 - Q1
    outliers_iqr.extend(df[(df[col] < Q1 - 1.5 * IQR) | (df[col] > Q3 +
1.5 * IQR)].index)

outliers_combined = list(set(outliers_z_score) | set(outliers_iqr))
print("Indices of outliers detected using Z-score method:",
outliers_z_score)
print("Indices of outliers detected using IQR method:", outliers_iqr)
print("Indices of outliers detected using both methods:",
outliers_combined)
```

```
Here's a breakdown of what the below code does:

1.Import Matplotlib:
   - The code imports the `matplotlib.pyplot` module as `plt`, which
provides a MATLAB-like plotting interface.

2. Print Message:
   - It prints a message indicating that histogram plots of numerical
values are being generated.

3. Generate Histogram Plots:
   - The `hist()` method is called on the DataFrame `df` to create
histogram plots for all numerical columns.
   - The `figsize=(10, 8)` argument sets the size of the figure to 10
inches in width and 8 inches in height.
   - Each histogram represents the distribution of values in a
numerical column.

4. Adjust Layout:
   - The `tight_layout()` function adjusts the subplot parameters to
fit the figure area, ensuring that the plots are properly spaced and do
not overlap.

5. Show Plots:
   - Finally, `plt.show()` is called to display the histogram plots.

This code provides a visual representation of the distribution of
numerical values in the DataFrame, making it easier to understand the
data's characteristics and identify any patterns or outliers.
```

```python
import matplotlib.pyplot as plt
print("histogram plots of numerical values ")
df.hist(figsize=(10, 8))
plt.tight_layout()
plt.show()
```

output:-



This code utilizes seaborn and matplotlib to visualize numerical features in the DataFrame `df` through box plots. Here's what each part does:

1. Import Seaborn and Matplotlib:
   - The code imports seaborn as sns and matplotlib.pyplot as plt to facilitate data visualization.

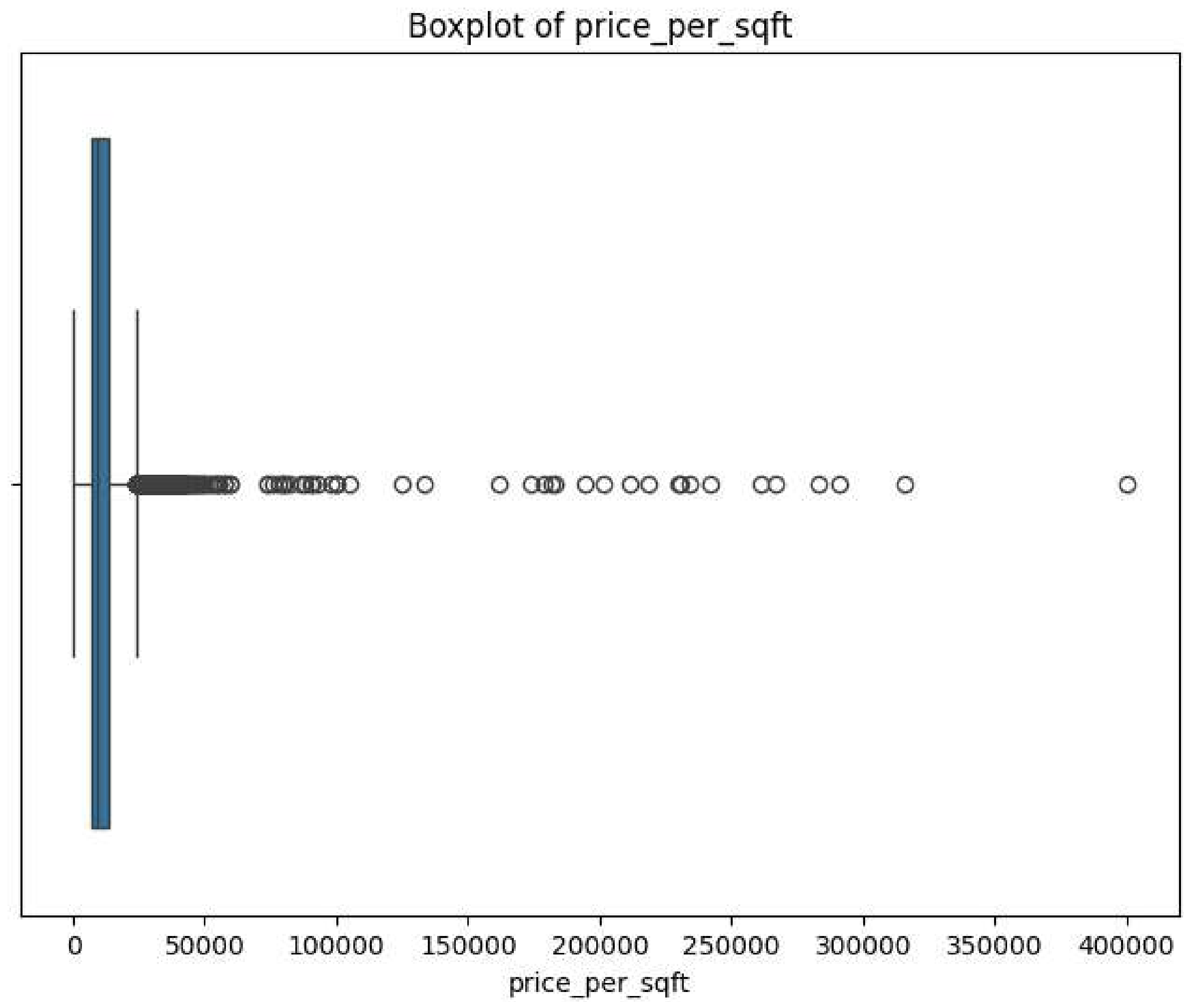
2. Identify Numerical Columns:
   - It selects numerical columns from the DataFrame `df` using `select_dtypes(include=['number']).columns` and stores them in the variable `numeric_columns`.

3. Generate Box Plots:
   - For each numerical column in `numeric_columns`, the code creates a new figure with a size of 8 inches in width and 6 inches in height (`plt.figure(figsize=(8, 6))`).
   - A box plot for the current numerical column is generated using `sns.boxplot(data=df[col], orient='h')`. The `orient='h'` parameter specifies that the box plot should be horizontal.
   - A title is added to the plot indicating the name of the column being visualized (`plt.title(f'Boxplot of {col}')`).
   - X-axis label is set as the column name (`plt.xlabel(col)`).
   - Finally, `plt.show()` is called to display the box plot.

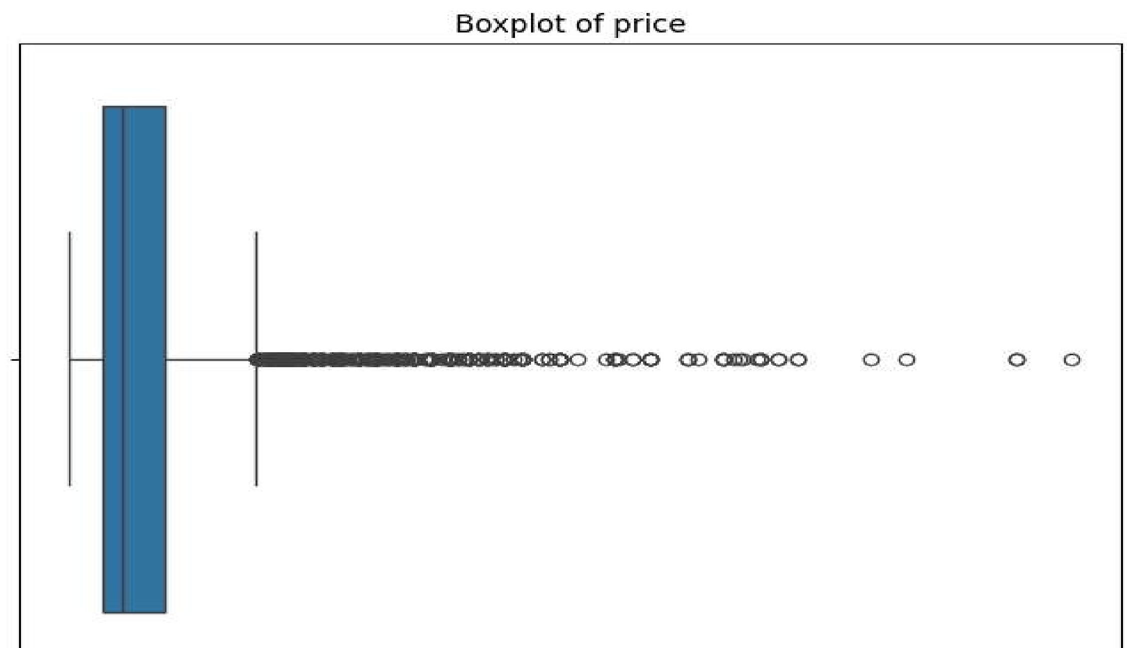
4. Repeat for Each Column:
   - The above steps are repeated for each numerical column in the DataFrame.

```python
import seaborn as sns
import matplotlib.pyplot as plt
# Visualize numerical features
numeric_columns = df.select_dtypes(include=['number']).columns
print ("these are the box plots")
for col in numeric_columns:
    plt.figure(figsize=(8, 6))
    sns.boxplot(data=df[col], orient='h')
    plt.title(f'Boxplot of {col}')
    plt.xlabel(col)
    plt.show()
```

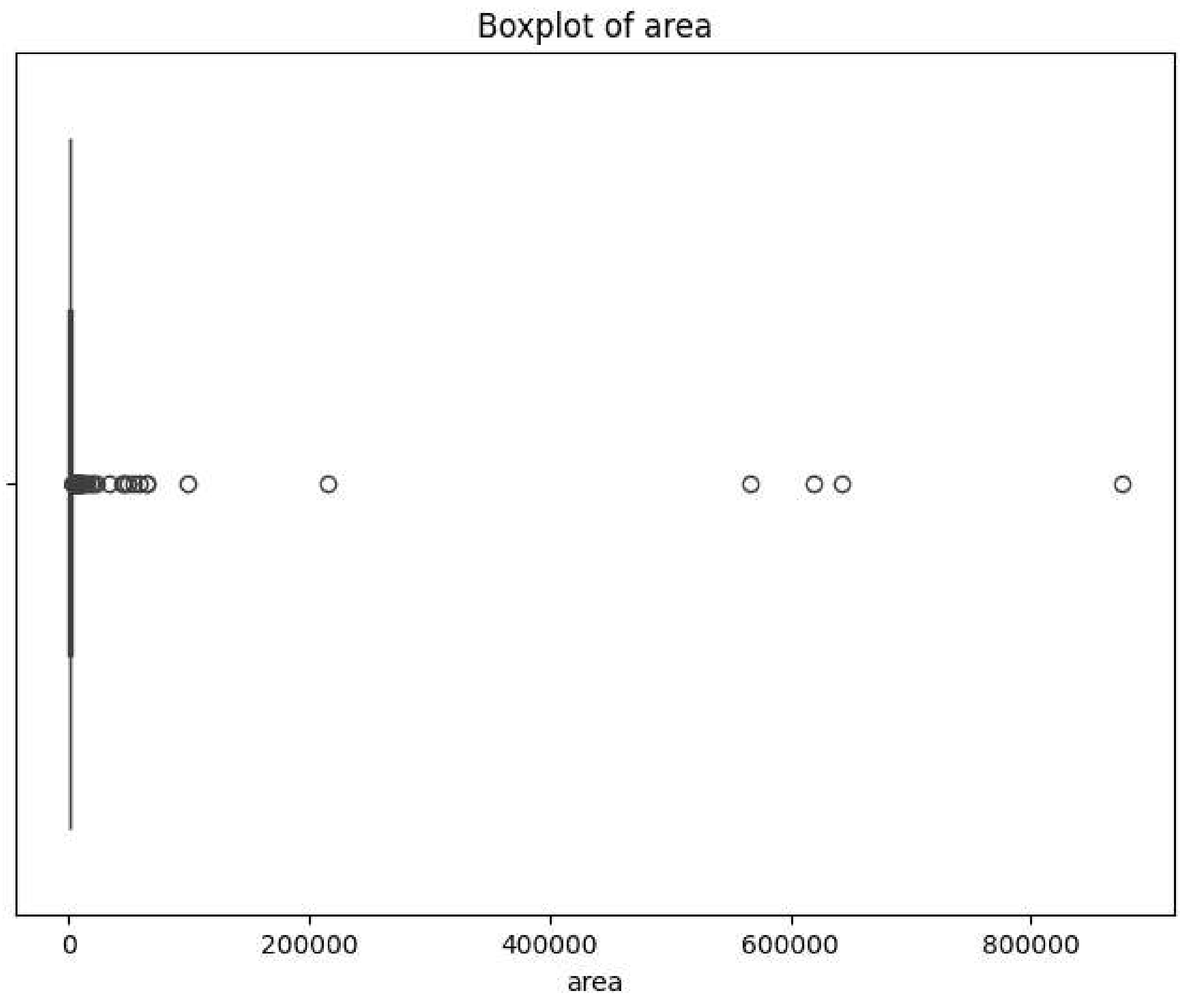
output:-

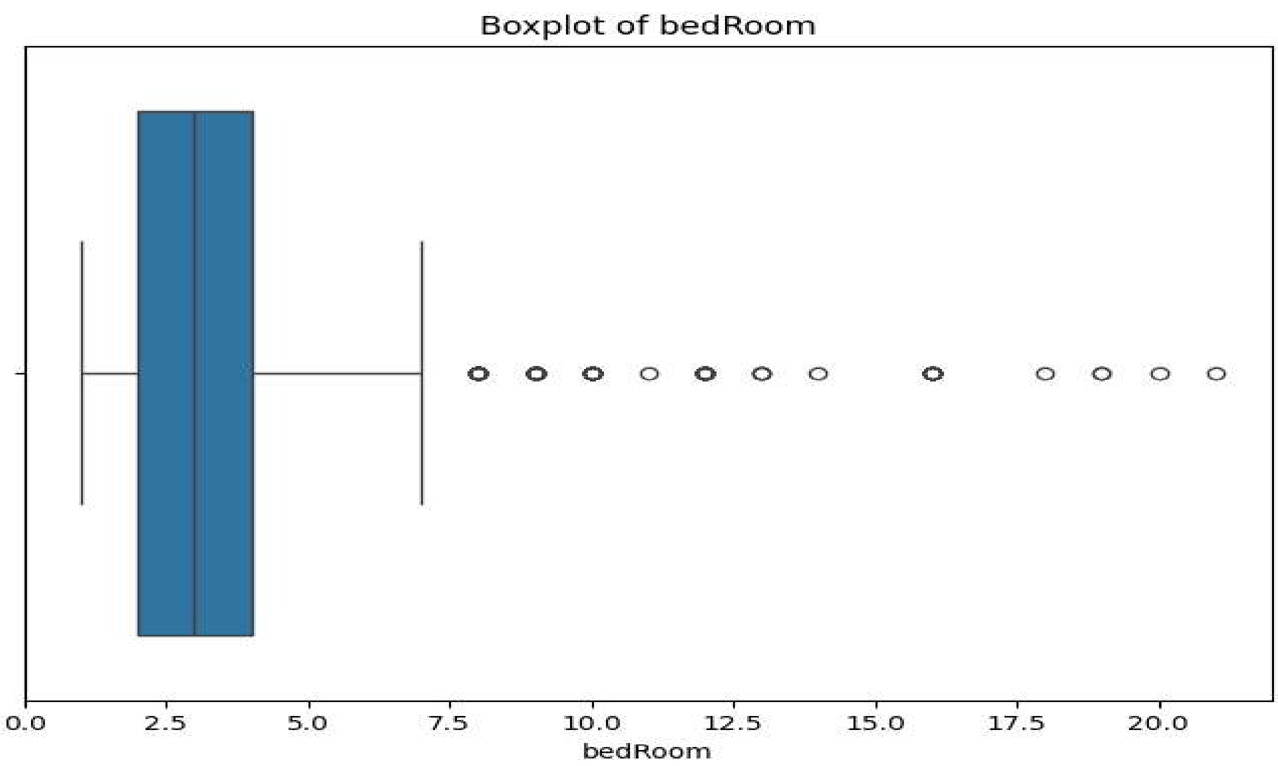


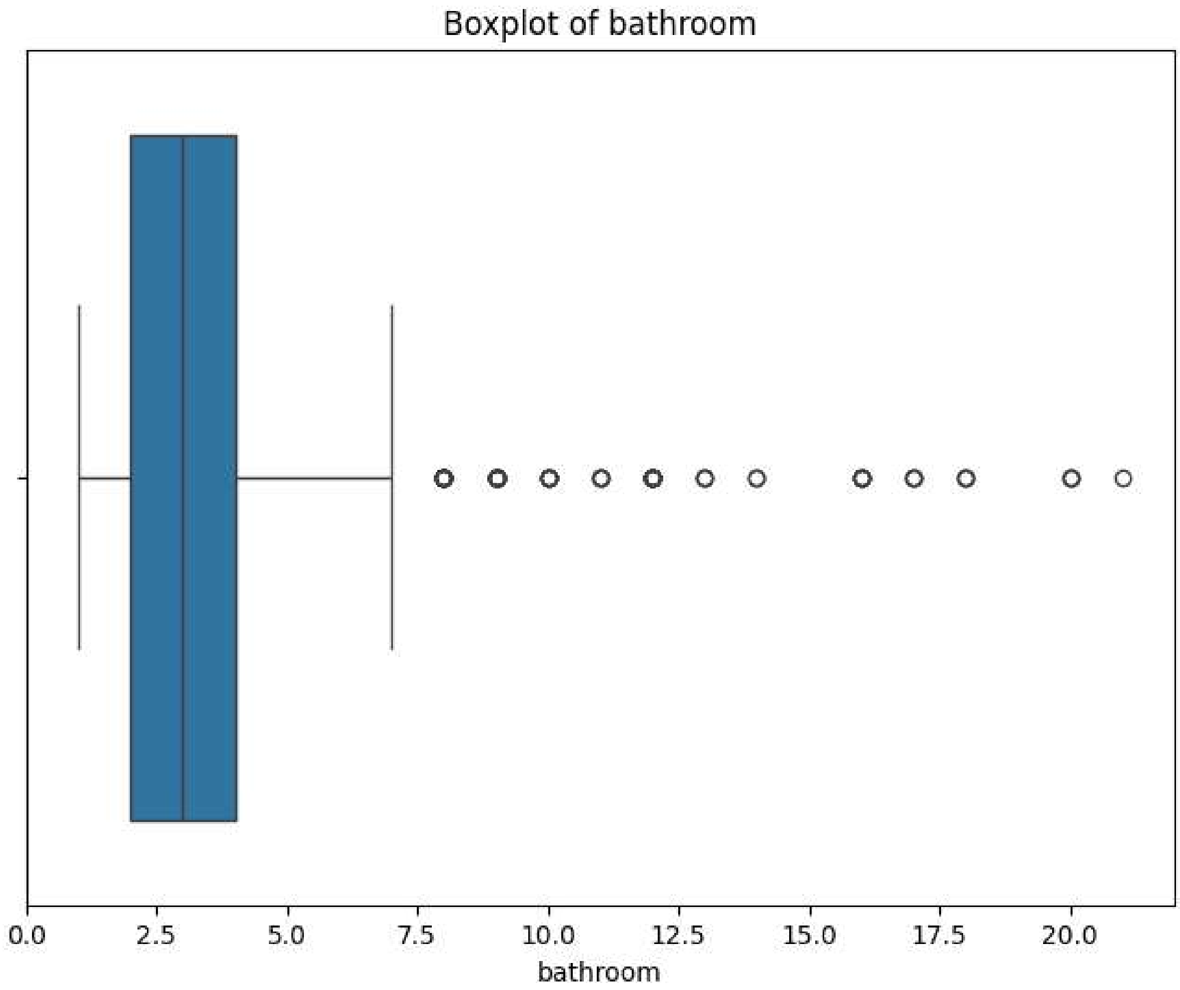
Boxplot of price_per_sqft
0
50000
100000
150000
200000
250000
300000
350000
400000
price_per_sqft

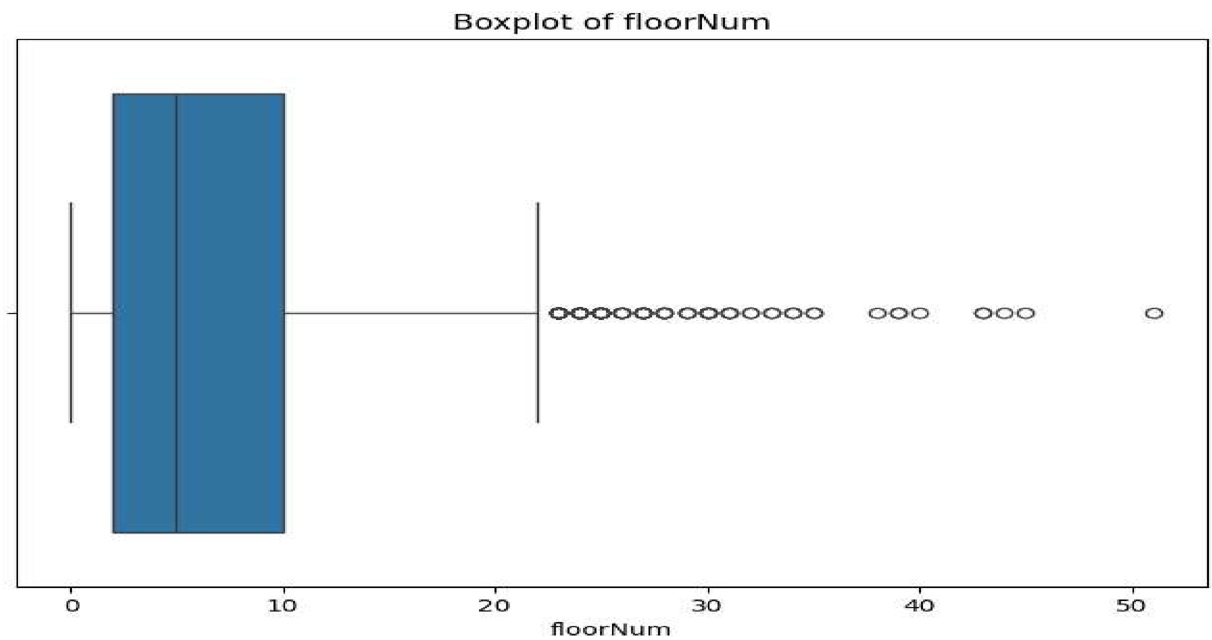


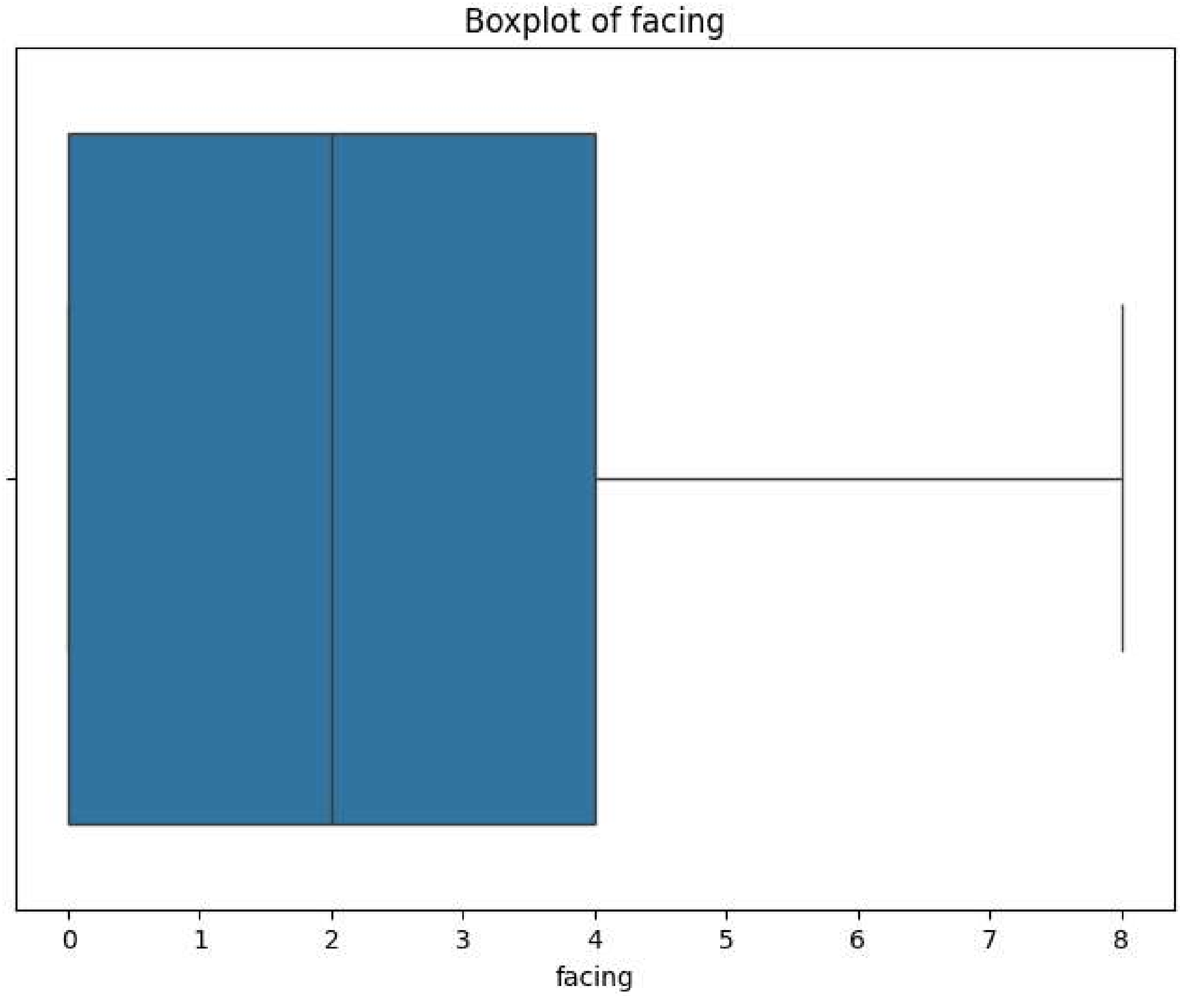
Boxplot of price
0
5
10
15
20
25
price

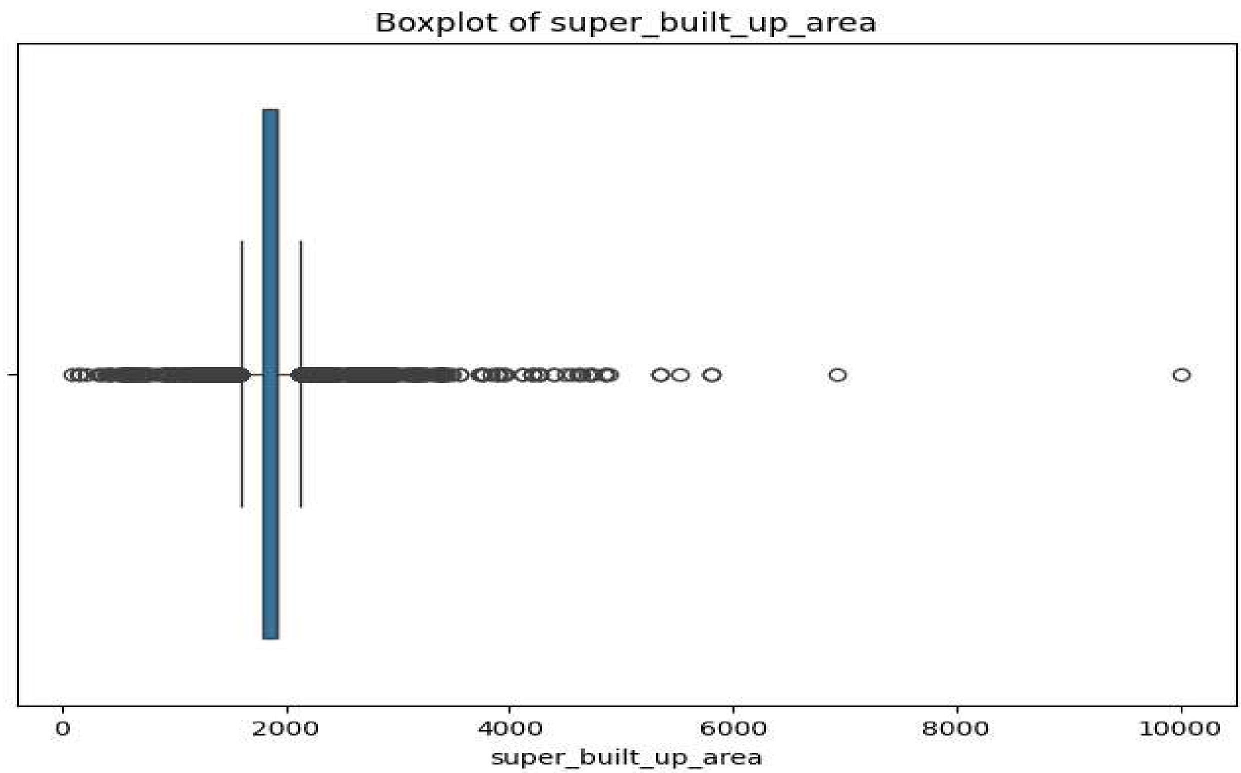

Boxplot of area
0
200000
400000
600000
800000
area

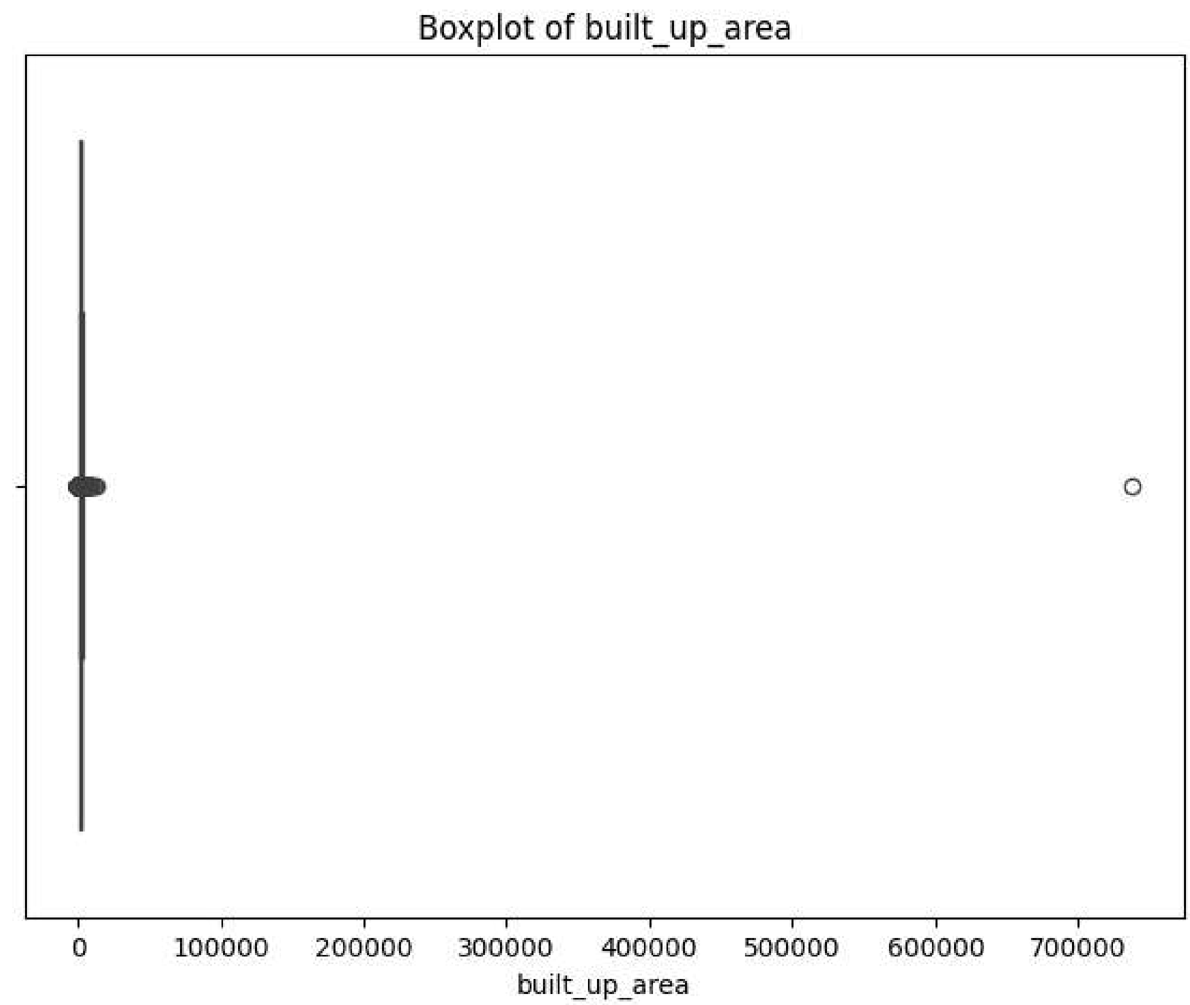

Boxplot of bedRoom
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
bedRoom

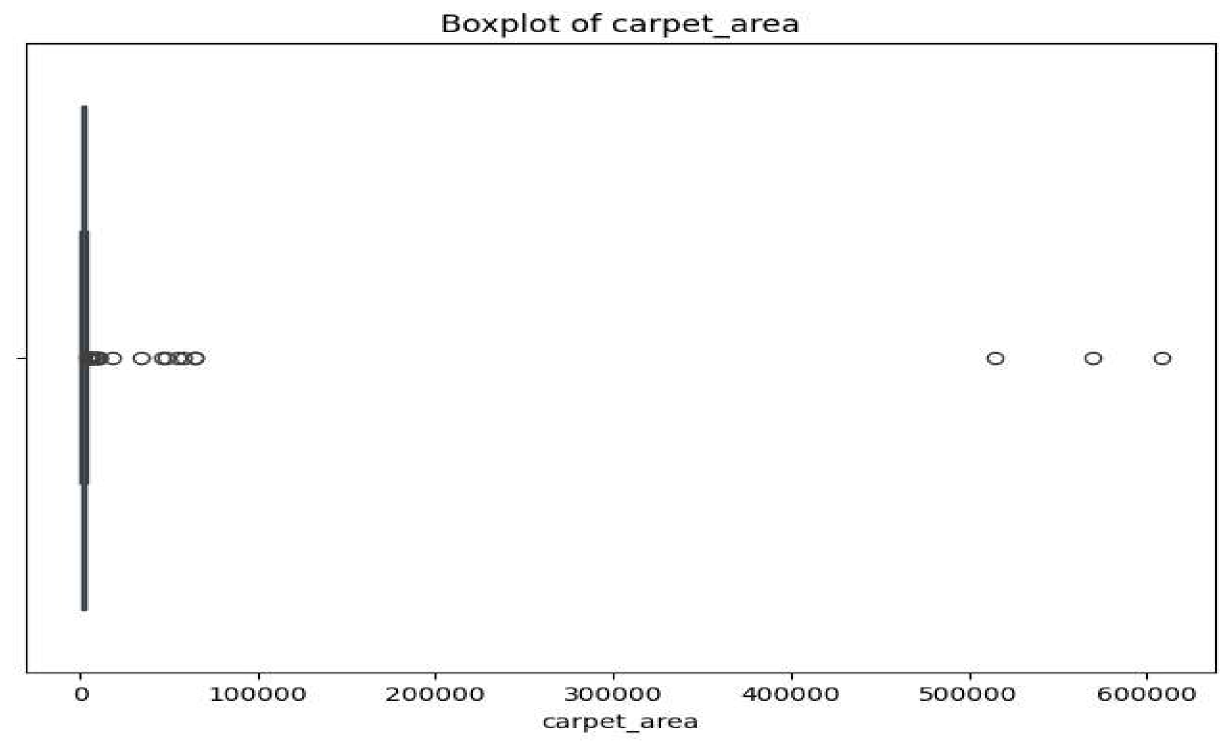

Boxplot of bathroom
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
bathroom

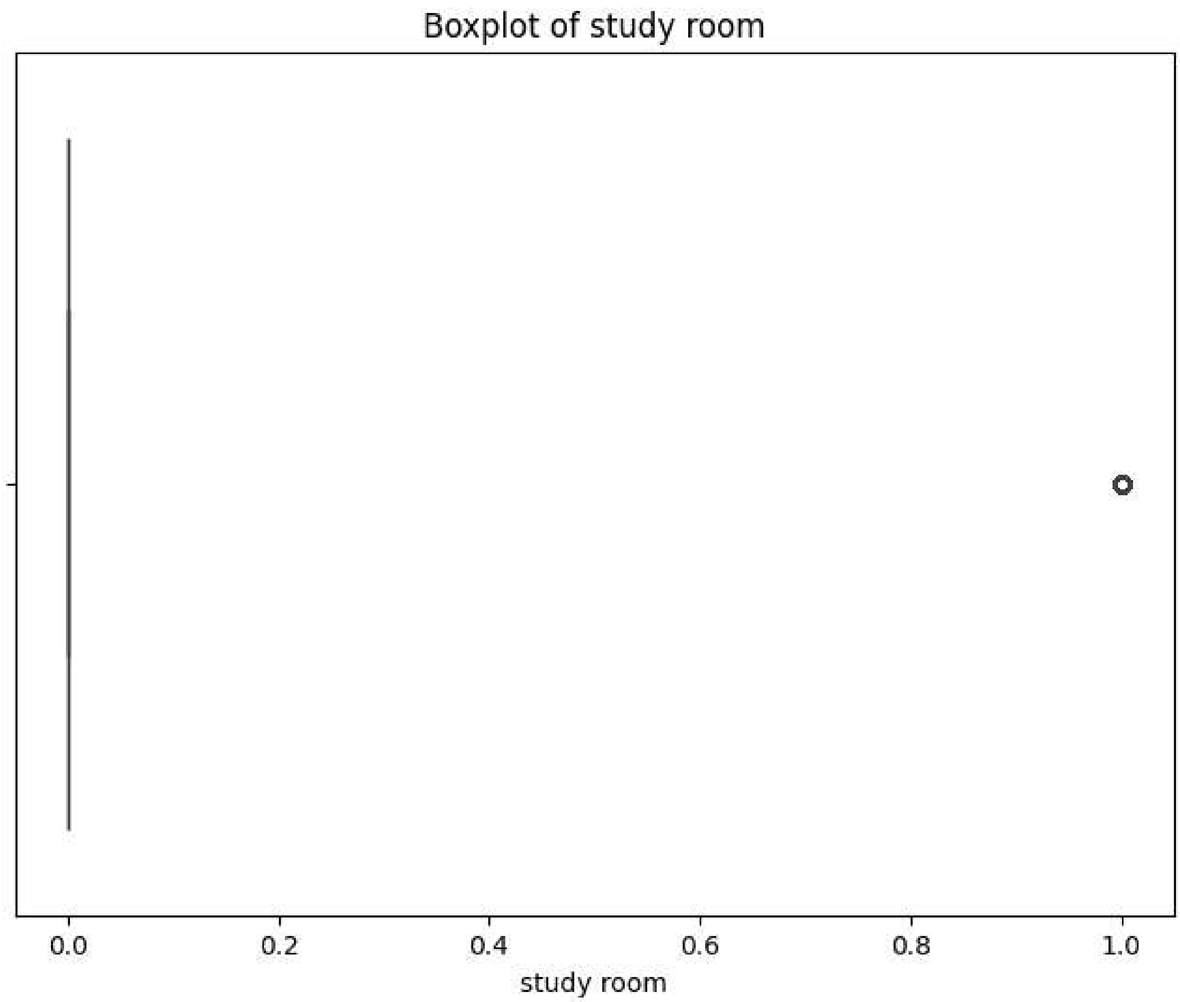

Boxplot of floorNum
0
10
20
30
40
50
floorNum


Boxplot of facing
0
1
2
3
4
5
6
7
8
facing

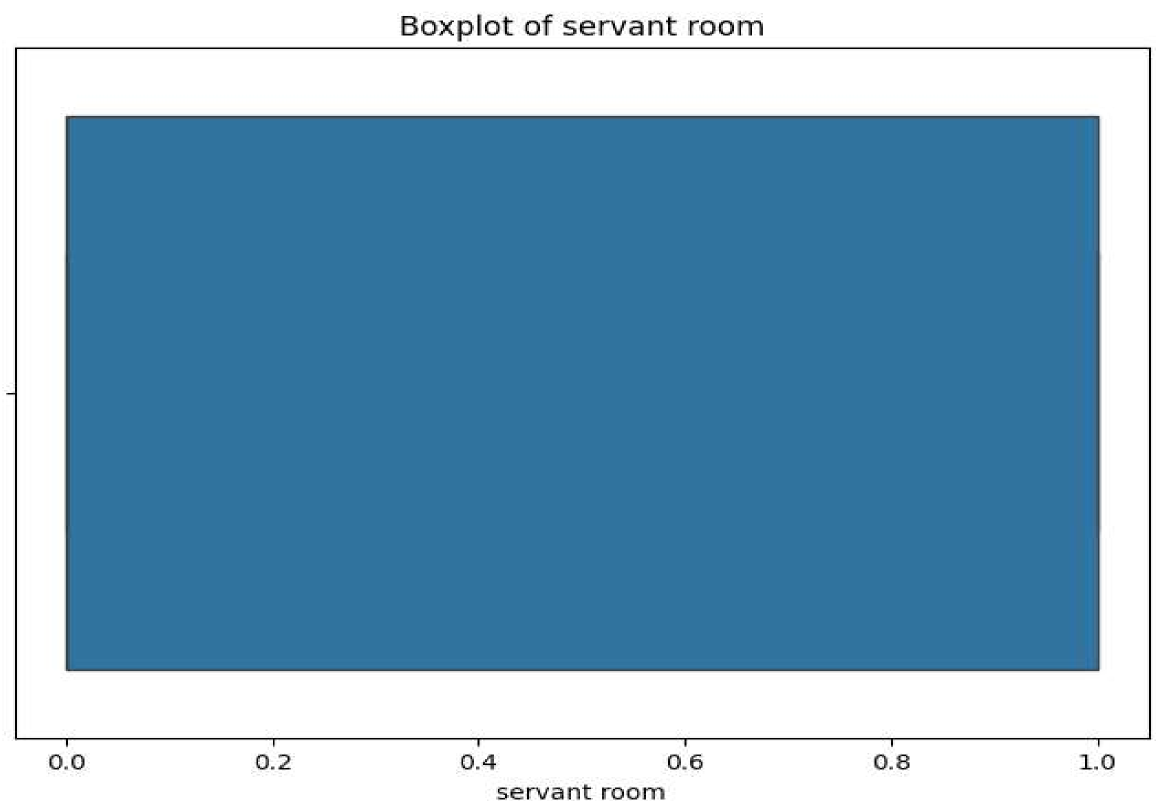

Boxplot of super_built_up_area
0
2000
4000
6000
8000
10000
super_built_up_area


Boxplot of built_up_area
0
100000
200000
300000
400000
500000
600000
700000
built_up_area

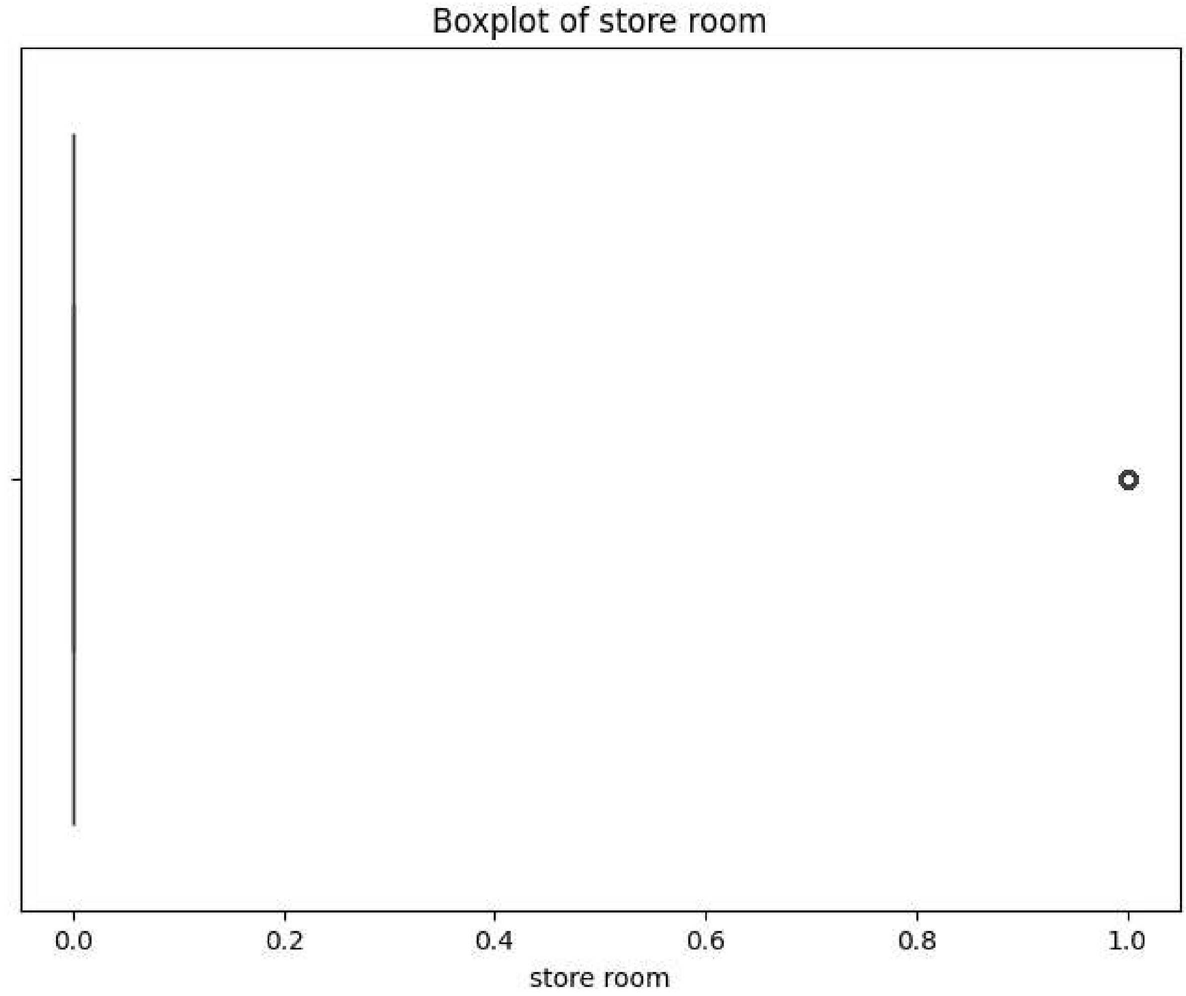

Boxplot of carpet_area
0
100000
200000
300000
400000
500000
600000
carpet_area

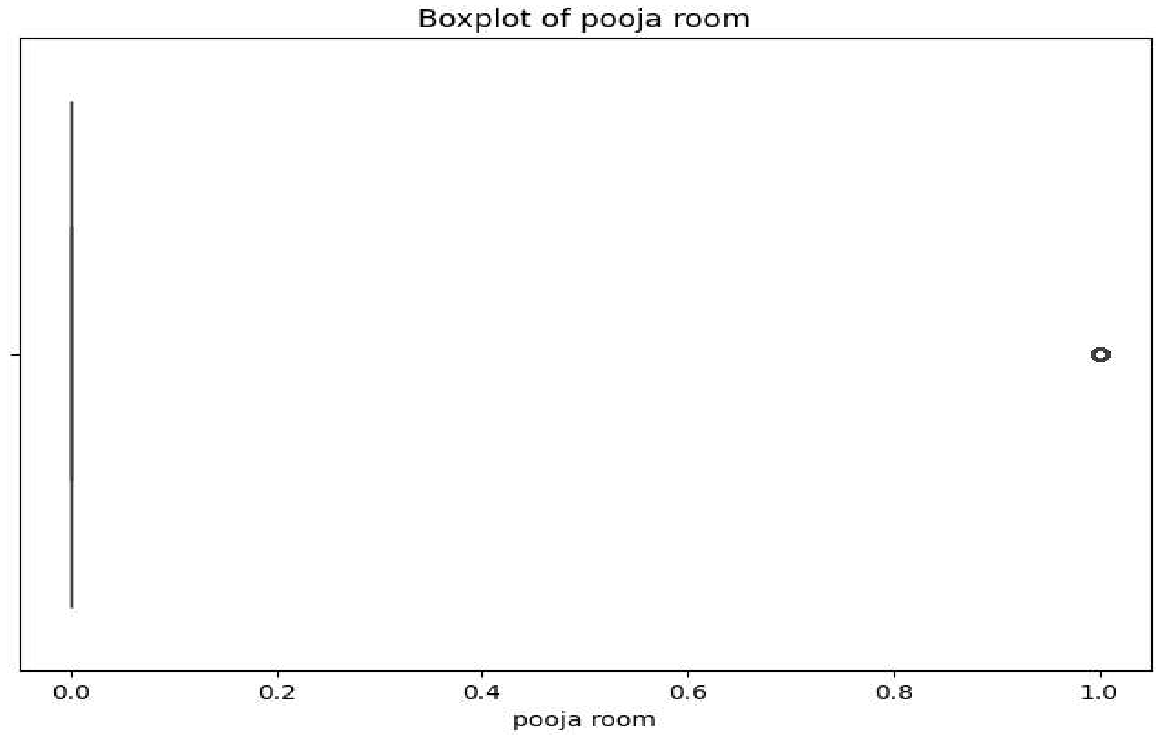


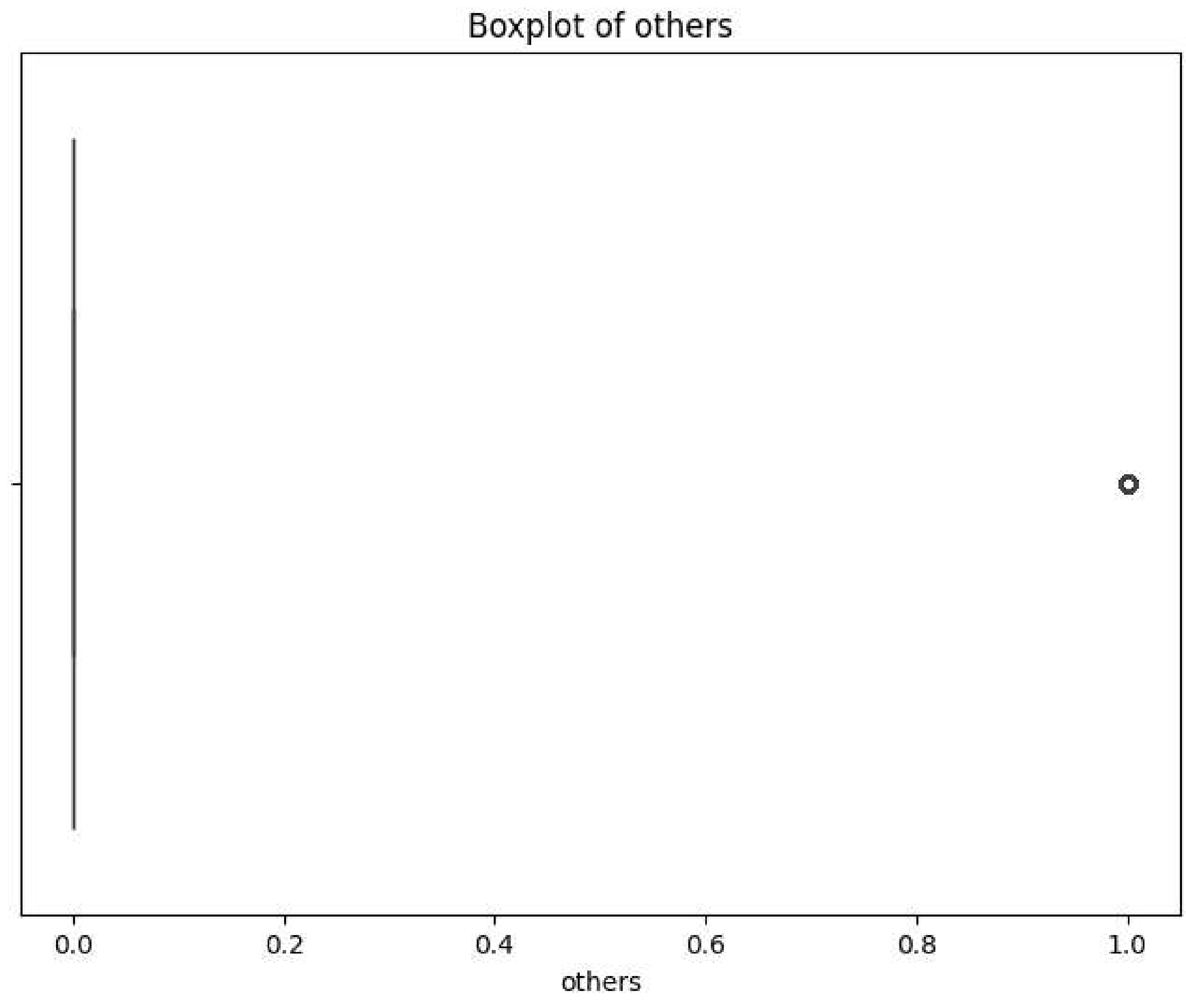
Boxplot of study room
0.0
0.2
0.4
0.6
0.8
1.0
study room

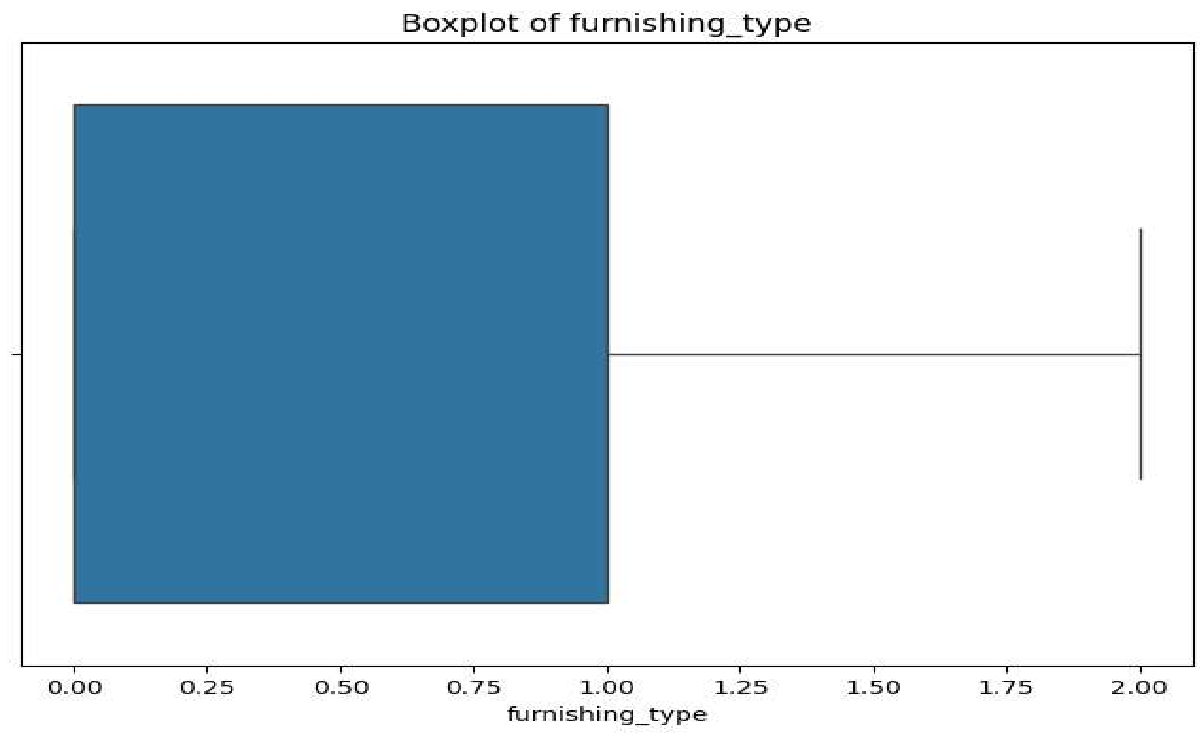


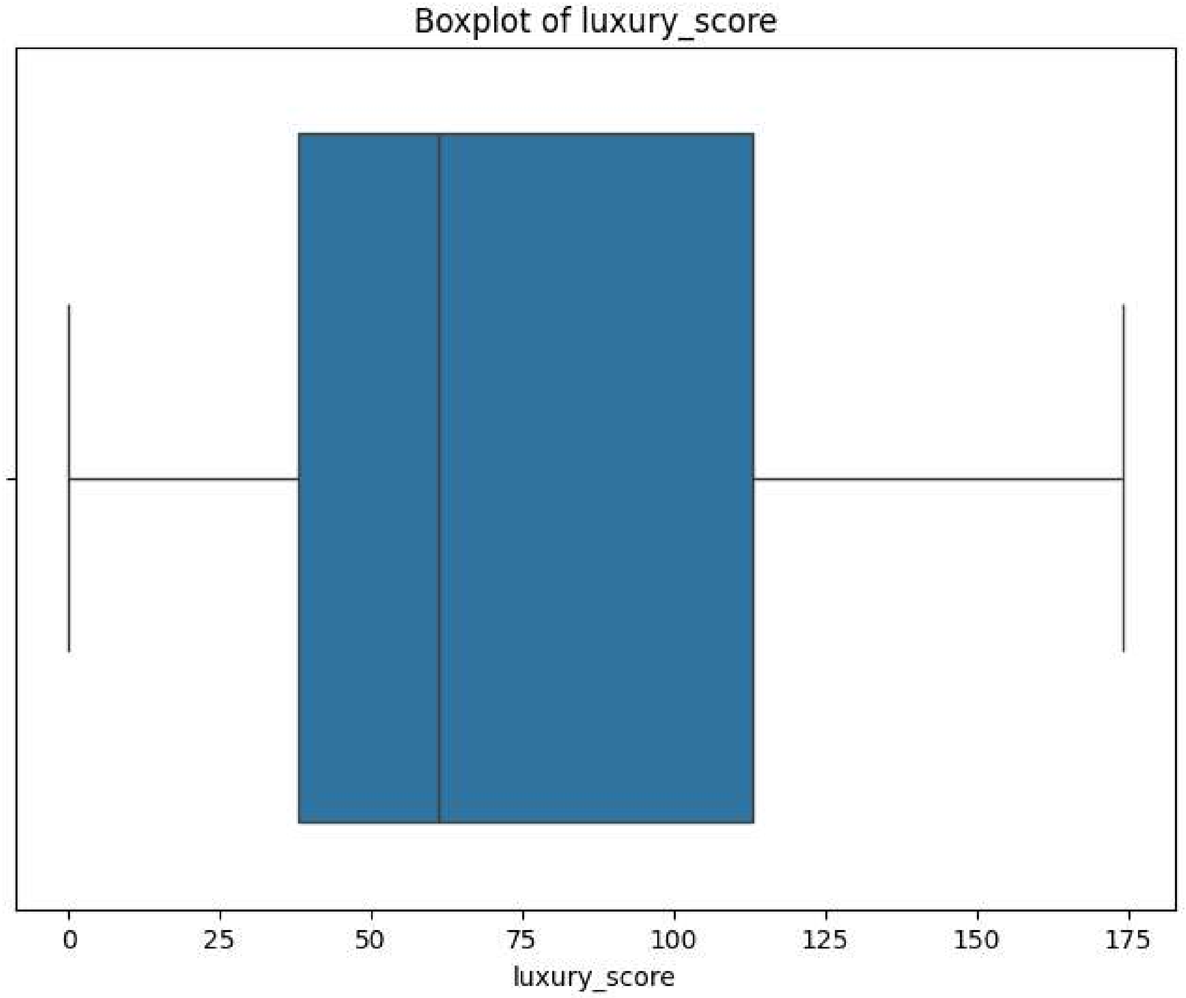
Boxplot of servant room
0.0
0.2
0.4
0.6
0.8
1.0
servant room


Boxplot of store room
0.0
0.2
0.4
0.6
0.8
1.0
store room


Boxplot of pooja room
0.0
0.2
0.4
0.6
0.8
1.0
pooja room


Boxplot of others
0.0
0.2
0.4
0.6
0.8
1.0
others


Boxplot of furnishing_type
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
furnishing_type



Here's a description what the below code does:

1. Import Libraries:
   - The code imports pandas as pd, matplotlib.pyplot as plt, and seaborn as sns for data manipulation and visualization.

2. Select Numerical Columns for Trimming:
   - It selects only numerical columns from the DataFrame `df` and stores them in the variable `numeric_data`.

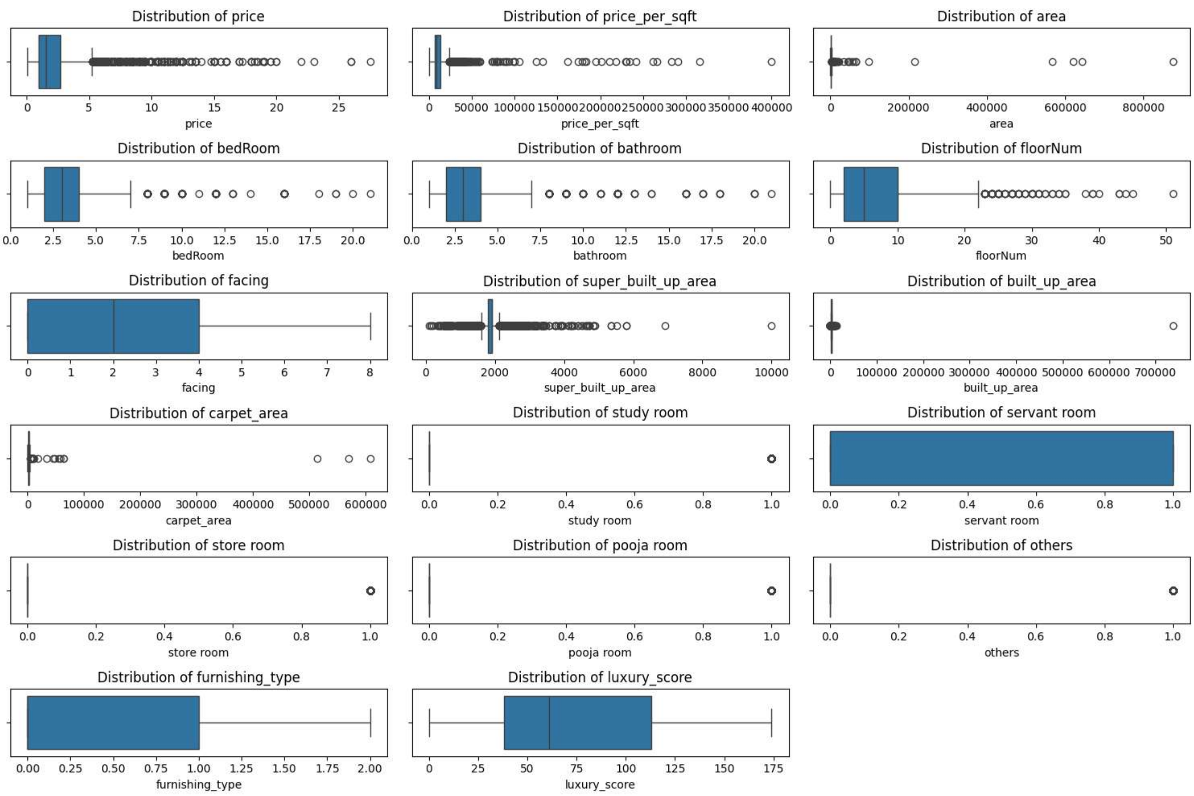
3. Calculate Trimming Boundaries:
   - The first quartile (Q1) and the third quartile (Q3) of each numerical column in `numeric_data` are calculated using the `quantile()` function with the respective quantile values (0.15 and 0.85).
   - The interquartile range (IQR) is computed as the difference between Q3 and Q1.
   - Lower and upper bounds for trimming are defined as 1.5 times the IQR below Q1 and above Q3, respectively.

4. Trim the DataFrame:

- A copy of the original DataFrame `df` is made and stored in `df_trimmed`.
- For each numerical column, rows where the values fall outside the defined lower and upper bounds are filtered out using boolean indexing.

5. Visualize Distribution Before Trimming:
- A figure is created with a size of 15 inches in width and 10 inches in height (`plt.figure(figsize=(15, 10))`).
- For each numerical column, a subplot is created with a box plot showing the distribution of values.
- The subplot titles and x-axis labels are set to indicate the column name.
- `plt.tight_layout()` ensures that the subplots are properly arranged without overlapping.
- Finally, `plt.show()` displays the box plots.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
#lets make sure only numrical column are used for trimming
#and defining the trimming boundaries
numeric_data = df.select_dtypes(include='number')
Q1 = numeric_data.quantile(0.15)
Q3 = numeric_data.quantile(0.85)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
df_trimmed = df.copy()
for col in numeric_data.columns:
    df_trimmed = df_trimmed[(df_trimmed[col] >= lower_bound[col]) &
(df_trimmed[col] <= upper_bound[col])]
#Visualizing the distribution of each numerical column
#prior to trimming allows for comprehensive analysis.
plt.figure(figsize=(15, 10))
for i, col in enumerate(numeric_data.columns, 1):
    plt.subplot(len(numeric_data.columns) // 3 + 1, 3, i)
    sns.boxplot(x=df[col])
    plt.title(f'Distribution of {col}')
    plt.xlabel(col)
plt.tight_layout()
plt.show()
```



Here's a description of what the code below does:

1. Import Libraries:
   - The code imports matplotlib.pyplot as plt and seaborn as sns for visualization.

2. Visualize Distribution After Trimming:
   - A figure is created with a size of 15 inches in width and 10 inches in height (`plt.figure(figsize=(15, 10))`).
   - For each numerical column in `df_trimmed`, a subplot is created with a box plot showing the distribution of values.
   - The subplot titles are set to indicate the column name along with the text "(After Trimming)" to differentiate from the plots before trimming.
   - The x-axis labels are set to the column names.
   - `plt.tight_layout()` ensures that the subplots are properly arranged without overlapping.
   - Finally, `plt.show()` displays the box plots.

```
import matplotlib.pyplot as plt
import seaborn as sns
# Visualizing the distribution of each numerical column after
#  trimming allows for comprehensive analysis.
plt.figure(figsize=(15, 10))
for i, col in
enumerate(df_trimmed.select_dtypes(include='number').columns, 1):
    plt.subplot(len(df_trimmed.select_dtypes(include='number').columns)
// 3 + 1, 3, i)
```

```
    sns.boxplot(x=df_trimmed[col])
    plt.title(f'Distribution of {col} (After Trimming)')
    plt.xlabel(col)
plt.tight_layout()
plt.show()
```