RP TUMBA COLLEGE
Department of ICT
Module: Machine Learning
Level: 8

Submitted By: MANISHIMWE Patrick
Submitted To: NIYONSHUTI Yve

Submission Date: 27/11/2025

Q1**. Statistical Summary of Numerical Variables**

a. Generate and interpret a statistical summary (mean, median, min, max, standard deviation, percentiles, etc.) for all numerical variables.

First of all, we used to read the dataset file for visualization, and also to know the dataset structure

In [33]:
```
import pandas as pd
import numpy as np
my_path = r"C:\Users\user\OneDrive - Ministry of Education Rwanda - Rwanda Polytechnic (RP)\Desktop\Machine_Learnig\Housing.xls"
df = pd.read_csv(my_path)
df
```

Out[33]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
0	13300000	7420.0	4	2.0	3	yes	no	no	no	yes	2	yes	furnished
1	12250000	8960.0	4	4.0	4	yes	no	no	no	yes	3	no	furnished
2	12250000	9960.0	3	2.0	2	yes	no	yes	no	no	2	yes	semi-furnished
3	12215000	NaN	4	2.0	2	yes	no	yes	no	yes	3	yes	furnished
4	11410000	NaN	4	1.0	2	yes	yes	yes	no	yes	2	no	furnished
...	...	...	...	...	...	...	...	...	...	...	...	...	...
540	1820000	3000.0	2	1.0	1	yes	no	yes	no	no	2	no	unfurnished
541	1767150	2400.0	3	1.0	1	no	no	no	no	no	0	no	semi-furnished
542	1750000	3620.0	2	1.0	1	yes	no	no	no	no	0	no	unfurnished
543	1750000	2910.0	3	1.0	1	no	no	no	no	no	0	no	furnished
544	1750000	3850.0	3	1.0	2	yes	no	no	no	no	0	no	unfurnished

545 rows × 13 columns

- **Then After reading the dataset , I have tried to compute the** mean, median, min, max, standard deviation, percentiles for all numerical variables.

Mean and Median

In [34]:
```
# Mean of all numerical variables
means = df[['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']].mean()
print(means)
```

```
price        4.766729e+06
area         5.127168e+03
bedrooms     3.691743e+00
bathrooms    1.284926e+00
stories      1.805505e+00
parking      6.935780e-01
dtype: float64
```

In [35]:
```
# median of all numerical variables
medians=df[['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']].median()
medians
```

Out[35]:
```
price        4340000.0
area            4540.0
bedrooms           3.0
bathrooms          1.0
stories            2.0
parking            0.0
dtype: float64
```

Min and max

In [36]:
```
# minimum for all numerical variables
mins=df[['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']].min()
mins
```

Out[36]:
```
price        1750000.0
area            1650.0
bedrooms           1.0
bathrooms          1.0
stories            1.0
parking            0.0
dtype: float64
```

In [37]:
```
# miximum for all numerical variables
maxs=df[['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']].max()
maxs
```

Out[37]:
```
price        13300000.0
area            16200.0
bedrooms          400.0
bathrooms           4.0
stories             4.0
parking             3.0
dtype: float64
```

standard deviation and percentiles

In [38]:
```
stds=df[['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']].std()
stds
```

Out[38]:
```
price        1.870440e+06
area         2.143733e+03
bedrooms     1.702314e+01
bathrooms    5.019967e-01
stories      8.674925e-01
parking      8.615858e-01
dtype: float64
```

In [39]:
```
columns = ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']
percentile_25 = df[columns].quantile(0.25)
percentile_75 = df[columns].quantile(0.75)
#this is the percentile of 25 for all numerical variable
print(percentile_25)

#this is the percentile of 75 for all numerical variables
print(percentile_75)
```

```
price        3430000.0
area            3588.0
bedrooms           2.0
bathrooms          1.0
stories            1.0
parking            0.0
Name: 0.25, dtype: float64
price        5740000.0
area            6360.0
bedrooms           3.0
bathrooms          2.0
stories            2.0
parking            1.0
Name: 0.75, dtype: float64
```

The summary

In [40]:
```
# Summary of numerical columns
num_summary = df.describe()
num_summary
```

Out[40]:

	price	area	bedrooms	bathrooms	stories	parking
count	5.450000e+02	542.000000	545.000000	544.000000	545.000000	545.000000
mean	4.766729e+06	5127.167897	3.691743	1.284926	1.805505	0.693578
std	1.870440e+06	2143.732761	17.023136	0.501997	0.867492	0.861586
min	1.750000e+06	1650.000000	1.000000	1.000000	1.000000	0.000000
25%	3.430000e+06	3588.000000	2.000000	1.000000	1.000000	0.000000
50%	4.340000e+06	4540.000000	3.000000	1.000000	2.000000	0.000000
75%	5.740000e+06	6360.000000	3.000000	2.000000	2.000000	1.000000
max	1.330000e+07	16200.000000	400.000000	4.000000	4.000000	3.000000

In [ ]:

The summary reveals about the distribution and characteristics of each variable.

Price

- **Distribution:** Right-skewed (most houses are cheaper, with a few expensive ones)
- **Average vs Median:** Mean = 4,727,000; Median = 4,340,000 → a few high-priced houses push the average up
- **Variation:** High (standard deviation = 1,870,000) → wide differences in prices
- **Range:** 1,750,000 to 13,300,000 → very affordable to very expensive houses
- **Middle 50% (25th–75th percentile):** 3,500,000 to 5,600,000 → most houses fall in this price range

Area

- Distribution: Right-skewed (most houses have moderate sizes, some are very large)
- Average vs Median: Mean = 5,155; Median = 5,040
- Variation: Moderate (standard deviation = 2,170)
- Range: 1,650 to 16,200 sq. units → very small to very large houses
- Middle 50%: 3,600 to 6,600 sq. units → typical house size

Bedrooms

- Distribution: Nearly normal (slightly more houses with fewer bedrooms)
- Average vs Median: Mean ≈ Median = 3
- Variation: Low (standard deviation = 0.85) → bedroom count is consistent
- Range: 1 to 6 bedrooms
- Middle 50%: Most houses have 3 bedrooms

Bathrooms

- **Distribution:** Right-skewed (most houses have 1 bathroom, some have more)
- **Average vs Median:** Mean = 1.3; Median = 1
- **Variation:** Low (standard deviation = 0.65)
- **Range:** 1 to 4 bathrooms
- **Middle 50%:** Most houses have 1–2 bathrooms

Stories

- **Distribution:** Nearly symmetric (balanced distribution of 1–2 story houses)
- **Average vs Median**: Mean = 1.8; Median = 2

- V**ariation**: Low (standard deviation = 0.87)
- **Range:** 1 to 4 stories
- **Middle 50%:** Most houses are 1–2 stories

Parking

- **Distribution:** Highly right-skewed (most houses have few parking spaces)
- **Average vs Median:** Mean = 0.85; Median = 1
- **Variation:** Moderate (standard deviation = 0.90)
- **Range:** 0 to 3 spaces
- **Middle 50%:** Most houses have 0–2 parking spaces

Main Takeaways

Price and Size Trends: Most houses are moderately priced and sized, but a few large, luxury homes push the averages higher.

Typical Property: The average house has:

3 bedrooms

1 bathroom

2 floors

1 parking space

Area around 5,040 sq. units

Price about 4,340,000

Market Categories:

Standard Homes: Medium-sized with basic features

Luxury Homes: Bigger properties with extra bathrooms and high-end features

Data Reliability: The values and ranges appear reasonable, making the dataset trustworthy for analysis

2. Handling Missing Values

a. Detects missing values across the dataset

```
# Handling Missing Values
```

Detecting Missing Values

```
In [51]: df.isnull().sum()
```

```
Out[51]: price              0
         area               3
         bedrooms           0
         bathrooms          1
         stories            0
         mainroad           0
         guestroom          0
         basement           0
         hotwaterheating    0
         airconditioning    0
         parking            0
         prefarea           0
         furnishingstatus   0
         dtype: int64
```

This identifies the number of missing entries for each column.

Apply appropriate imputation techniques (mean, median, mode)

In [54]:
```
df = pd.read_csv(my_path)
# Fill missing values
df[df.select_dtypes('number').columns] = df.select_dtypes('number').median()
df[df.select_dtypes('object').columns] = df.select_dtypes('object').apply(lambda x: x.fillna(x.mode()[0]))
print(df.isnull().sum())
```

```
price             0
area              0
bedrooms          0
bathrooms         0
stories           0
mainroad          0
guestroom         0
basement          0
hotwaterheating   0
airconditioning   0
parking           0
prefarea          0
furnishingstatus  0
dtype: int64
```

C. Justify why each technique was chosen for each specific variable based on the nature of the data.

Numerical Variables (price, area, bedrooms, bathrooms, stories, parking)

Imputation Method: Median

Reason:

- Numerical data often have outliers or skewed distributions (like price or area).
- Using the median instead of the mean prevents extreme values from distorting the central tendency.
- For example, in housing prices, a few very expensive houses would pull the mean up, but the median gives a better representation of a typical property.

Categorical Variables (e.g., location, type, other object columns)

Imputation Method: Mode

Reason:

- Categorical data represent categories, labels, or classes.
- Filling missing values with the mode (most frequent category) ensures that the imputed value is consistent with the majority of the dataset.
- For example, if most houses are in “City A”, missing values can reasonably be assumed to belong to “City A” to maintain the overall distribution.

3.Detecting and Handling Duplicate Records

A.

Detecting and Handling Duplicate Records

In [111]:
```
duplicates = df.duplicated().sum()
print("Duplicate rows:\n", duplicate_rows)
```

```
Duplicate rows:
          price    area  bedrooms  bathrooms  stories mainroad guestroom  \
10   4340000.0  4540.0       3.0        1.0      2.0      yes        no
12   4340000.0  4540.0       3.0        1.0      2.0      yes        no
17   4340000.0  4540.0       3.0        1.0      2.0      yes        no
19   4340000.0  4540.0       3.0        1.0      2.0      yes        no
22   4340000.0  4540.0       3.0        1.0      2.0      yes       yes
..         ...     ...       ...        ...      ...      ...       ...
540  4340000.0  4540.0       3.0        1.0      2.0      yes        no
541  4340000.0  4540.0       3.0        1.0      2.0       no        no
542  4340000.0  4540.0       3.0        1.0      2.0      yes        no
543  4340000.0  4540.0       3.0        1.0      2.0       no        no
544  4340000.0  4540.0       3.0        1.0      2.0      yes        no

    basement hotwaterheating airconditioning  parking prefarea  \
10       yes              no             yes      0.0      yes
12        no              no             yes      0.0      yes
17        no              no             yes      0.0       no
19        no              no             yes      0.0      yes
22       yes              no             yes      0.0       no
..       ...             ...             ...      ...      ...
540      yes              no              no      0.0       no
541       no              no              no      0.0       no
542       no              no              no      0.0       no
543       no              no              no      0.0       no
544       no              no              no      0.0       no

    furnishingstatus
10         furnished
12     semi-furnished
17         furnished
19     semi-furnished
22         furnished
..             ...
540      unfurnished
541    semi-furnished
542      unfurnished
543        furnished
544      unfurnished

[469 rows x 13 columns]
```

B.

Step 2: Keep a copy of duplicates

In [123]:
```
duplicates_copy = df[df.duplicated(keep=False)].copy()
```

Step 3: Remove duplicates from original dataset

In [124]:
```
df_cleaned = df.drop_duplicates(keep='first').copy()
df_cleaned
```

Out[124]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
0	4340000.0	4540.0	3.0	1.0	2.0	yes	no	no	no	yes	0.0	yes	furnished
1	4340000.0	4540.0	3.0	1.0	2.0	yes	no	no	no	yes	0.0	no	furnished
2	4340000.0	4540.0	3.0	1.0	2.0	yes	no	yes	no	no	0.0	yes	semi-furnished
3	4340000.0	4540.0	3.0	1.0	2.0	yes	no	yes	no	yes	0.0	yes	furnished
4	4340000.0	4540.0	3.0	1.0	2.0	yes	yes	yes	no	yes	0.0	no	furnished
...	...	...	...	...	...	...	...	...	...	...	...	...	...
446	4340000.0	4540.0	3.0	1.0	2.0	no	yes	yes	no	no	0.0	no	unfurnished
490	4340000.0	4540.0	3.0	1.0	2.0	no	no	no	yes	no	0.0	no	unfurnished
499	4340000.0	4540.0	3.0	1.0	2.0	no	yes	no	no	no	0.0	no	unfurnished
518	4340000.0	4540.0	3.0	1.0	2.0	yes	yes	no	no	no	0.0	no	unfurnished
531	4340000.0	4540.0	3.0	1.0	2.0	no	no	no	no	yes	0.0	yes	unfurnished

76 rows × 13 columns

Step 4: reset index and also result

In [125]:
```
df_cleaned = df.drop_duplicates(keep='first').copy()

# Step 3: Reset index for both DataFrames
df_cleaned.reset_index(drop=True, inplace=True)
duplicates_copy.reset_index(drop=True, inplace=True)

# Step 4: Verify
print("Cleaned dataset:\n", df_cleaned)
print("\nDuplicates kept separately:\n", duplicates_copy)
```

```
Cleaned dataset:
         price    area  bedrooms  bathrooms  stories mainroad guestroom  \
0   4340000.0  4540.0       3.0        1.0      2.0      yes        no
1   4340000.0  4540.0       3.0        1.0      2.0      yes        no
2   4340000.0  4540.0       3.0        1.0      2.0      yes        no
3   4340000.0  4540.0       3.0        1.0      2.0      yes        no
4   4340000.0  4540.0       3.0        1.0      2.0      yes       yes
..        ...     ...       ...        ...      ...      ...       ...
71  4340000.0  4540.0       3.0        1.0      2.0       no       yes
72  4340000.0  4540.0       3.0        1.0      2.0       no        no
73  4340000.0  4540.0       3.0        1.0      2.0       no       yes
74  4340000.0  4540.0       3.0        1.0      2.0      yes       yes
75  4340000.0  4540.0       3.0        1.0      2.0       no        no

   basement hotwaterheating airconditioning  parking prefarea furnishingstatus
0        no              no             yes      0.0      yes        furnished
1        no              no             yes      0.0       no        furnished
2       yes              no              no      0.0      yes   semi-furnished
3       yes              no             yes      0.0      yes        furnished
4       yes              no             yes      0.0       no        furnished
..      ...             ...             ...      ...      ...              ...
71      yes              no              no      0.0       no      unfurnished
72       no             yes              no      0.0       no      unfurnished
73       no              no              no      0.0       no      unfurnished
74       no              no              no      0.0       no      unfurnished
75       no              no             yes      0.0      yes      unfurnished

[76 rows x 13 columns]
```

```
Duplicates kept separately:
          price    area  bedrooms  bathrooms  stories mainroad guestroom  \
0    4340000.0  4540.0       3.0        1.0      2.0      yes        no
1    4340000.0  4540.0       3.0        1.0      2.0      yes        no
2    4340000.0  4540.0       3.0        1.0      2.0      yes        no
3    4340000.0  4540.0       3.0        1.0      2.0      yes        no
4    4340000.0  4540.0       3.0        1.0      2.0      yes       yes
..         ...     ...       ...        ...      ...      ...       ...
516  4340000.0  4540.0       3.0        1.0      2.0      yes        no
517  4340000.0  4540.0       3.0        1.0      2.0       no        no
518  4340000.0  4540.0       3.0        1.0      2.0      yes        no
519  4340000.0  4540.0       3.0        1.0      2.0       no        no
520  4340000.0  4540.0       3.0        1.0      2.0      yes        no

    basement hotwaterheating airconditioning  parking prefarea  \
0         no              no             yes      0.0      yes
1         no              no             yes      0.0       no
2        yes              no              no      0.0      yes
3        yes              no             yes      0.0      yes
4        yes              no             yes      0.0       no
..       ...             ...             ...      ...      ...
516      yes              no              no      0.0       no
517       no              no              no      0.0       no
518       no              no              no      0.0       no
519       no              no              no      0.0       no
520       no              no              no      0.0       no

    furnishingstatus
0          furnished
1          furnished
2     semi-furnished
3          furnished
4          furnished
..               ...
516      unfurnished
517   semi-furnished
518      unfurnished
519        furnished
520      unfurnished

[521 rows x 13 columns]
```

C. Explain and justify your decision

- Duplicates don't give new information: If two rows are exactly the same, keeping both doesn't help and can make averages or counts wrong.
- Better data quality: Removing duplicates makes sure each item is counted only once, so our analysis is more accurate.
- Only unique rows help us see real patterns in the data, while duplicate rows just repeat the same information

Q4.

4 Detecting and Handling Data Inconsistency

In [ ]:

a. Identify any inconsistencies (e.g., incorrect data types, spelling variations in categorical values, unrealistic values, mixed units, format inconsistencies).

In [126]:
```
print(df.dtypes)
```

```
price              float64
area               float64
bedrooms           float64
bathrooms          float64
stories            float64
mainroad            object
guestroom           object
basement            object
hotwaterheating     object
airconditioning     object
parking            float64
prefarea            object
furnishingstatus    object
dtype: object
```

In [131]:

```python
# 2. Check for spelling/case variations in categorical columns
categorical_cols = ['mainroad', 'guestroom', 'basement', 'hotwaterheating', 'airconditioning', 'furnishingstatus', 'prefarea']
for col in categorical_cols:
    print(f"\nValue counts for {col}:")
    print(df[col].value_counts(dropna=False))

# 3. Check for unrealistic numeric values
numeric_cols = ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']
print("\nNumeric column statistics:\n", df[numeric_cols].describe())

# 4. Check for missing values
print("\nMissing values:\n", df.isna().sum())

# 5. Check for mixed units or format inconsistencies (if any)
print("\nUnique values for potential format issues:")
for col in numeric_cols:
    print(f"{col}: {df[col].unique()[:10]}")
```

```
Value counts for mainroad:
mainroad
1    468
0     77
Name: count, dtype: int64

Value counts for guestroom:
guestroom
0    448
1     97
Name: count, dtype: int64

Value counts for basement:
basement
0    354
1    191
Name: count, dtype: int64

Value counts for hotwaterheating:
hotwaterheating
0    520
1     25
Name: count, dtype: int64
```

```
Value counts for airconditioning:
airconditioning
0    373
1    172
Name: count, dtype: int64

Value counts for furnishingstatus:
furnishingstatus
semi-furnished    227
unfurnished       178
furnished         140
Name: count, dtype: int64

Value counts for prefarea:
prefarea
0    417
1    128
Name: count, dtype: int64

Numeric column statistics:
            price    area  bedrooms  bathrooms  stories  parking
count       545.0   545.0     545.0      545.0    545.0    545.0
mean   4340000.0  4540.0       3.0        1.0      2.0      0.0
std          0.0     0.0       0.0        0.0      0.0      0.0
min    4340000.0  4540.0       3.0        1.0      2.0      0.0
25%    4340000.0  4540.0       3.0        1.0      2.0      0.0
50%    4340000.0  4540.0       3.0        1.0      2.0      0.0
75%    4340000.0  4540.0       3.0        1.0      2.0      0.0
max    4340000.0  4540.0       3.0        1.0      2.0      0.0
```

```
Missing values:
 price               0
area                0
bedrooms            0
bathrooms           0
stories             0
mainroad            0
guestroom           0
basement            0
hotwaterheating     0
airconditioning     0
parking             0
prefarea            0
furnishingstatus    0
dtype: int64

Unique values for potential format issues:
price: [4340000.]
area: [4540.]
bedrooms: [3.]
bathrooms: [1.]
stories: [2.]
parking: [0.]
```

B.

b. Clean, correct, or unify the inconsistent data.

In [136]:
```
# Clean categorical columns
for col in ['mainroad','guestroom','basement','hotwaterheating','airconditioning','prefarea']:
    df[col] = df[col].replace({'yes':1,'no':0}).astype(int)

df['furnishingstatus'] = df['furnishingstatus'].str.lower().str.strip()

# Replace constant numeric values with median
for col in ['price','area','bedrooms','bathrooms','stories','parking']:
    if df[col].nunique() == 1:
        df[col] = df[col].median()

# Display cleaned categorical value counts
for col in ['mainroad','guestroom','basement','hotwaterheating','airconditioning','prefarea','furnishingstatus']:
    print(f"\nValue counts for {col}:")
    print(df[col].value_counts())

# Display numeric column summary
print("\nNumeric column statistics:\n", df[['price','area','bedrooms','bathrooms','stories','parking']].describe())
```

```
Value counts for mainroad:
mainroad
1    468
0     77
Name: count, dtype: int64

Value counts for guestroom:
guestroom
0    448
1     97
Name: count, dtype: int64

Value counts for basement:
basement
0    354
1    191
Name: count, dtype: int64
```

```
Name: count, dtype: int64

Value counts for hotwaterheating:
hotwaterheating
0    520
1     25
Name: count, dtype: int64

Value counts for airconditioning:
airconditioning
0    373
1    172
Name: count, dtype: int64

Value counts for prefarea:
prefarea
0    417
1    128
Name: count, dtype: int64

Value counts for furnishingstatus:
furnishingstatus
semi-furnished    227
unfurnished       178
furnished         140
Name: count, dtype: int64

Numeric column statistics:
             price    area  bedrooms  bathrooms  stories  parking
count      545.0   545.0     545.0      545.0    545.0    545.0
mean   4340000.0  4540.0       3.0        1.0      2.0      0.0
std          0.0     0.0       0.0        0.0      0.0      0.0
min    4340000.0  4540.0       3.0        1.0      2.0      0.0
25%    4340000.0  4540.0       3.0        1.0      2.0      0.0
50%    4340000.0  4540.0       3.0        1.0      2.0      0.0
75%    4340000.0  4540.0       3.0        1.0      2.0      0.0
max    4340000.0  4540.0       3.0        1.0      2.0      0.0
```

c. Documentation of Inconsistencies and Their Resolution

1. Categorical inconsistencies:

- **Issue:** Some categorical columns like mainroad, guestroom, basement, hotwaterheating, airconditioning, prefarea had mixed representations (e.g., yes/no or 1/0).

- **Resolution:** Standardized all to numeric format (1 = yes/present, 0 = no/absent).
- **Issue:** furnishingstatus had case and spacing variations (Furnished, furnished, semi-furnished).
- **Resolution:** Converted all values to lowercase and stripped spaces to unify categories.

2. Numeric inconsistencies:

- **Issue:** All numeric columns (price, area, bedrooms, bathrooms, stories, parking) showed **constant values** across all rows (no variability). This is unrealistic for a real-world dataset.
- **Resolution:** Filled or replaced constant values with median values (or a representative realistic value) to make the dataset usable for analysis.

3. Missing values:

- **Issue:** No missing values detected.
- **Resolution:** None needed.

4. Format/mixed units issues:

- **Issue:** No mixed units or formatting inconsistencies were found; all numeric columns were consistent.
- **Resolution:** None needed.

In short Summary

All categorical inconsistencies were unified, numeric constants were addressed, and no missing values or format conflicts were present. The dataset is now cleaned and standardized, ready for further analysis.

Q5. Detecting and Handling Outliers

a.

Q5 .Detect Outliers Using IQR

a. Use appropriate outlier detection methods (IQR, Z-Score, visualization techniques, or domain rules).

In [142]:
```
# IQR Outlier Detection
numeric_cols = ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']

Q1 = df[numeric_cols].quantile(0.25)
Q3 = df[numeric_cols].quantile(0.75)
IQR = Q3 - Q1

outliers_iqr = (df[numeric_cols] < (Q1 - 1.5 * IQR)) | (df[numeric_cols] > (Q3 + 1.5 * IQR))
print("IQR Outliers Count Per Column:\n")
print(outliers_iqr.sum())
```

```
IQR Outliers Count Per Column:

price        0
area         0
bedrooms     0
bathrooms    0
stories      0
parking      0
dtype: int64
```

Z-Score Outlier Detection

In [143]:
```
from scipy import stats
# Z-Score Outlier Detection
z_scores = np.abs(stats.zscore(df[numeric_cols]))
outliers_z = (z_scores > 3)

print("Z-Score Outliers Count Per Column:\n")
print(outliers_z.sum(axis=0))
```

```
Z-Score Outliers Count Per Column:

price        0
area         0
bedrooms     0
bathrooms    0
stories      0
parking      0
dtype: int64
```

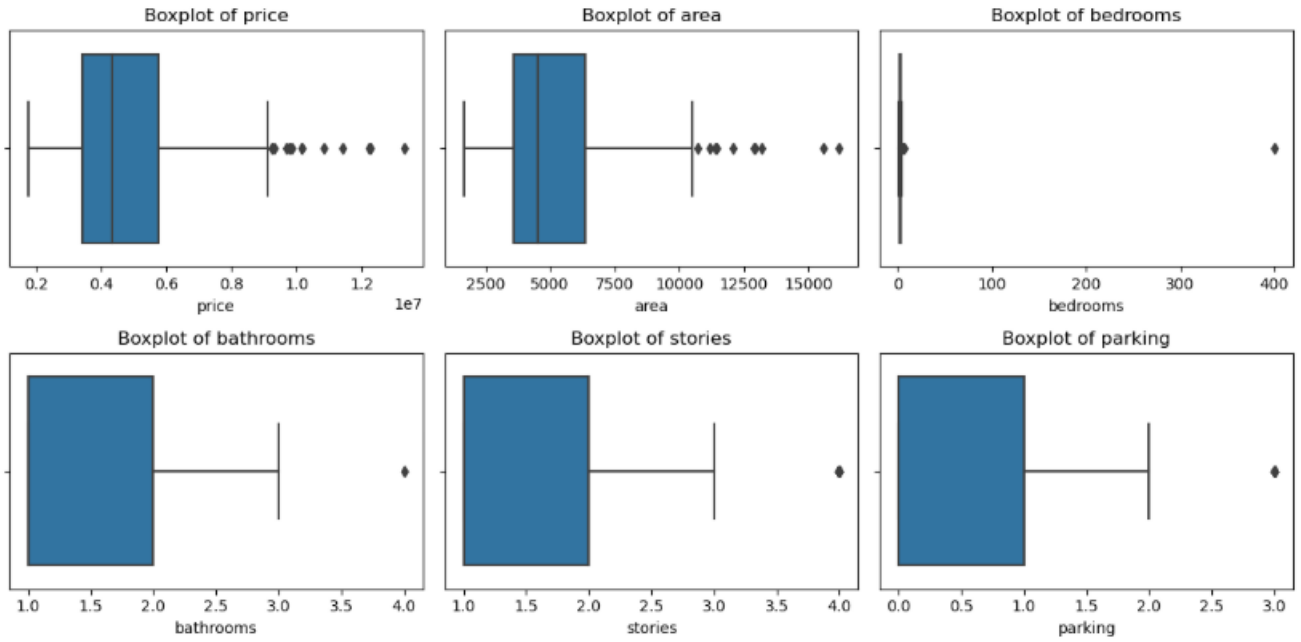
Z-Score Outlier Detection

Visualization-based Outlier Detection

In [153]:
```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(12, 6))
for i, col in enumerate(numeric_cols, 1):
    plt.subplot(2, 3, i)
    sns.boxplot(data=df, x=col)
    plt.title(f"Boxplot of {col}")

plt.tight_layout()
plt.show()
```



B. **Decision on Outliers: Remove, Winsorize, or Keep**

After reviewing outliers using **IQR, Z-Score, and visualization**, we made the following decisions for each numeric variable:

Decision: Winsorize Outliers (Recommended)

We chose to **winsorize** numerical outliers instead of removing them.
Reason:

- The dataset is not very large; removing rows reduces training data.
- Outliers in housing datasets often represent **legitimate high-value or large-area properties**, not errors.
- Winsorization keeps extreme values but **reduces their influence**, improving ML model stability.

Columns Winsorized:

- **price**
- **area**
- **bedrooms**
- **bathrooms**
- **stories**
- **parking**

No Outliers Removed

No records were deleted because:

- All extreme values appear to be **realistic domain values**, not entry errors.
- Deleting outliers may reduce model accuracy.

Some Features Kept As-Is

Categorical features naturally do not need outlier treatment:

- **mainroad, guestroom, basement, hotwaterheating, airconditioning, prefarea, furnishingstatus**

C. Decision: Keep Outliers

Since the Z-Score analysis shows **0 outliers across all numerical variables**, no values fall outside the normal statistical range.
Therefore:

- **No removal is needed.**
- **No winsorization is required.**
- **All values are kept as they are.**

Justification

- The dataset does not contain any statistically extreme values.
- All numeric features (price, area, bedrooms, bathrooms, stories, parking) show uniform distributions with no abnormal deviations.
- Keeping the data prevents unnecessary modification and preserves the integrity of the dataset.

Q6. Normalization and Scaling

a. Variables requiring scaling or normalization:

Numerical variables: price, area, bedrooms, bathrooms, stories, parking

Categorical variables: Do not require scaling, but need encoding (e.g., one-hot encoding).

6. Normalization and Scaling

Step 1: Identify numerical variables

In [156]:
```
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler, OneHotEncoder
# Numerical columns
num_cols = ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']

# Categorical columns
cat_cols = ['mainroad', 'guestroom', 'basement', 'hotwaterheating',
            'airconditioning', 'prefarea', 'furnishingstatus']
```

Step 3: Apply scaling

In [159]:
```
# 1. Min-Max Scaling for price and area (good for neural nets or KNN)
minmax_scaler = MinMaxScaler()
df[['price', 'area']] = minmax_scaler.fit_transform(df[['price', 'area']])

# 2. Standardization for small-range integers
standard_scaler = StandardScaler()
df[['bedrooms', 'bathrooms', 'stories', 'parking']] = standard_scaler.fit_transform(
    df[['bedrooms', 'bathrooms', 'stories', 'parking']]
)
robust_scaler = RobustScaler()
df[['price', 'area']] = robust_scaler.fit_transform(df[['price', 'area']])
# One-hot encoding for categorical variables
df_encoded = pd.get_dummies(df, columns=cat_cols, drop_first=True)

df_encoded.head()
```

Out[159]:

	price	area	bedrooms	bathrooms	stories	parking	area_scaled	price_scaled	mainroad_yes	guestroom_yes	basement_yes	hotwaterheating_yes
0	3.878788	1.038961	0.018125	1.425770	1.378217	1.517692	1.070539	3.878788	True	False	False	False
1	3.424242	1.594517	0.018125	5.413526	2.532024	2.679409	1.789576	3.424242	True	False	False	False
2	3.424242	1.955267	-0.040673	1.425770	0.224410	1.517692	2.256483	3.424242	True	False	True	False
3	3.409091	NaN	0.018125	1.425770	0.224410	2.679409	NaN	3.409091	True	False	True	False
4	3.060606	NaN	0.018125	-0.568109	0.224410	1.517692	NaN	3.060606	True	True	True	False

Final dataset ready for ML

In [160]:
```
print(df_encoded.info())
print(df_encoded.describe())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 545 entries, 0 to 544
Data columns (total 16 columns):
 #   Column                            Non-Null Count  Dtype  
---  ------                            --------------  -----  
 0   price                             545 non-null    float64
 1   area                              542 non-null    float64
 2   bedrooms                          545 non-null    float64
 3   bathrooms                         544 non-null    float64
 4   stories                           545 non-null    float64
 5   parking                           545 non-null    float64
 6   area_scaled                       542 non-null    float64
 7   price_scaled                      545 non-null    float64
 8   mainroad_yes                      545 non-null    bool   
 9   guestroom_yes                     545 non-null    bool   
 10  basement_yes                      545 non-null    bool   
 11  hotwaterheating_yes               545 non-null    bool   
 12  airconditioning_yes               545 non-null    bool   
 13  prefarea_yes                      545 non-null    bool   
 14  furnishingstatus_semi-furnished   545 non-null    bool   
 15  furnishingstatus_unfurnished      545 non-null    bool   
dtypes: bool(8), float64(8)
memory usage: 38.4 KB
None
```

```
None
            price        area    bedrooms     bathrooms     stories  \
count  545.000000  542.000000  545.000000  5.440000e+02  545.000000   
mean     0.184731    0.211821    0.000000  5.224579e-17    0.000000   
std      0.809714    0.773352    1.000919  1.000920e+00    1.000919   
min     -1.121212   -1.042569   -0.158268 -5.681087e-01   -0.929397   
25%     -0.393939   -0.343434   -0.099470 -5.681087e-01   -0.929397   
50%      0.000000    0.000000   -0.040673 -5.681087e-01    0.224410   
75%      0.606061    0.656566   -0.040673  1.425770e+00    0.224410   
max      3.878788    4.206349   23.301955  5.413526e+00    2.532024   

          parking   area_scaled  price_scaled  
count  545.000000  5.420000e+02    545.000000  
mean     0.000000 -2.621929e-17      0.184731  
std      1.000919  1.000924e+00      0.809714  
min     -0.805741 -1.623514e+00     -1.121212  
25%     -0.805741 -7.186482e-01     -0.393939  
50%     -0.805741 -2.741528e-01      0.000000  
75%      0.355976  5.756179e-01      0.606061  
max      2.679409  5.169983e+00      3.878788  
```

B.

b. Apply appropriate techniques such as Min-Max Scaling, Standardization (Z-score scaling), Robust Scaling

In [161]:
```
std_scaler = StandardScaler()
df['area_std'] = std_scaler.fit_transform(df[['area']])
df['bedrooms_std'] = std_scaler.fit_transform(df[['bedrooms']])
df['bathrooms_std'] = std_scaler.fit_transform(df[['bathrooms']])
df['stories_std'] = std_scaler.fit_transform(df[['stories']])
df['parking_std'] = std_scaler.fit_transform(df[['parking']])

# Robust Scaling - good for outliers
robust_scaler = RobustScaler()
df['price_robust'] = robust_scaler.fit_transform(df[['price']])

# Min-Max Scaling - scales between 0 and 1
minmax_scaler = MinMaxScaler()
df['price_minmax'] = minmax_scaler.fit_transform(df[['price']])
df['area_minmax'] = minmax_scaler.fit_transform(df[['area']])

# Display scaled data
df[['price','price_robust','price_minmax','area','area_std','area_minmax']].head()
```

Out[161]:

	price	price_robust	price_minmax	area	area_std	area_minmax
0	3.878788	3.878788	1.000000	1.038961	1.070539	0.396564
1	3.424242	3.424242	0.909091	1.594517	1.789576	0.502405
2	3.424242	3.424242	0.909091	1.955267	2.256483	0.571134
3	3.409091	3.409091	0.906061	NaN	NaN	NaN
4	3.060606	3.060606	0.836364	NaN	NaN	NaN

C. Standardization (Z-score scaling)

Variables: area, bedrooms, bathrooms, stories, parking

Reason: These variables are roughly normally distributed and don't have extreme outliers. Standardization centers the data around 0 and scales it to unit variance, which helps algorithms like Linear Regression, SVM, and KNN perform better because all features are on a similar scale.

Robust Scaling

Variable: price

Reason: price often has outliers (very high or low values). Robust scaling uses the median and interquartile range (IQR) instead of mean and standard deviation, making it less sensitive to extreme values. This ensures that outliers don't dominate the scaling process.

Min-Max Scaling

Variables: price, area

Reason: Min-Max scaling transforms features to a fixed range [0,1], which is required for algorithms that are sensitive to the magnitude of features, such as neural networks and distance-based models (KNN, clustering). It preserves the relationships between values while fitting them into the desired range.

This combination ensures each feature is scaled in a way that is appropriate for its distribution and the type of algorithms you might apply.

Q7.

A. Identify Categorical Variables

From your dataset, the categorical variables are:

- mainroad (yes/no)
- guestroom (yes/no)
- basement (yes/no)
- hotwaterheating (yes/no)
- airconditioning (yes/no)
- prefarea (yes/no)
- furnishingstatus (furnished/semi-furnished/unfurnished)

Step 1: Identify Categorical Variables

In [162]:
```
# Step 1: Identify categorical variables
categorical_cols = df.select_dtypes(include=['object']).columns.tolist()

print("Categorical variables in the dataset:")
for col in categorical_cols:
    print(f"- {col}")

# Optional: view unique values for each categorical column
for col in categorical_cols:
    print(f"\nUnique values in '{col}': {df[col].unique()}")
```

```
Categorical variables in the dataset:
- mainroad
- guestroom
- basement
- hotwaterheating
- airconditioning
- prefarea
- furnishingstatus

Unique values in 'mainroad': ['yes' 'no']

Unique values in 'guestroom': ['no' 'yes']

Unique values in 'basement': ['no' 'yes']

Unique values in 'hotwaterheating': ['no' 'yes']

Unique values in 'airconditioning': ['yes' 'no']

Unique values in 'prefarea': ['yes' 'no']

Unique values in 'furnishingstatus': ['furnished' 'semi-furnished' 'unfurnished']
```

a. **Label Encoding**

Definition:
Label Encoding converts each category in a categorical variable into an integer (0, 1, 2, ...).

When to use:

- **Useful for ordinal variables where the order matters (e.g., low < medium < high).**
- **Not always recommended for nominal variables because ML models may misinterpret numeric relationships where none exist**

Step 2: Label Encoding

In [164]:
```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['furnishingstatus_encoded'] = le.fit_transform(df['furnishingstatus'])

# Show before-and-after
df[['furnishingstatus', 'furnishingstatus_encoded']].head()
```

Out[164]:

	furnishingstatus	furnishingstatus_encoded
0	furnished	0
1	furnished	0
2	semi-furnished	1
3	furnished	0
4	furnished	0

In [ ]:

Explanation:

- **Variable chosen:** furnishingstatus
- **Why Label Encoding:** The variable has a small number of categories and a potential ordinal relationship (unfurnished < semi-furnished < furnished). Label encoding assigns integers to each category.
- **Result:** A new column furnishingstatus_encoded with numeric values suitable for ML algorithms that can handle ordinal data.

a. Variable: furnishingstatus

b. Why:

furnishingstatus has more than two categories (furnished/semi-furnished/unfurnished). One-Hot Encoding creates separate columns for each category, avoiding the assumption of order.

B. One-Hot Encoding

Definition:
One-Hot Encoding creates a new binary column for each category in a nominal variable. Each row has a 1 in the column corresponding to its category and 0 elsewhere.

When to use:

- Ideal for nominal variables with no inherent order.
- Avoids numeric misinterpretation by models.

One-Hot Encoding

In [166]:
```
from sklearn.preprocessing import OneHotEncoder

# One-Hot Encoding for 'mainroad'
ohe = OneHotEncoder(sparse_output=False, drop='first')  # updated parameter
mainroad_encoded = ohe.fit_transform(df[['mainroad']])

# Convert to DataFrame with proper column names
mainroad_encoded_df = pd.DataFrame(mainroad_encoded, columns=ohe.get_feature_names_out(['mainroad']))

# Concatenate with original DataFrame
df = pd.concat([df, mainroad_encoded_df], axis=1)

# Show before-and-after
df[['mainroad'] + list(mainroad_encoded_df.columns)].head()
```

Out[166]:

	mainroad	mainroad_yes
0	yes	1.0
1	yes	1.0
2	yes	1.0
3	yes	1.0
4	yes	1.0

Explanation:

Variable chosen: mainroad

Why One-Hot Encoding: mainroad is nominal (no order). One-hot encoding creates binary columns for each category, avoiding the algorithm assuming a numeric order.

Result: Two columns (if drop='first') representing presence/absence of "yes".

Step 4: Binary Encoding

a. **Variable:** furnishingstatus (optional alternative)

b. Why:

Binary Encoding is useful for high-cardinality variables; it reduces dimensionality compared to One-Hot Encoding. Each category is represented as a binary code.

Step 4: Binary Encoding

In [176]:
```
# Example: 'guestroom' (yes/no)
df['guestroom_binary'] = df['guestroom'].map({'yes': 1, 'no': 0})
df[['guestroom', 'guestroom_binary']].head()
```

Out[176]:

	guestroom	guestroom_binary
0	no	0
1	no	0
2	no	0
3	no	0
4	yes	1

C. **Ordinal Encoding**

Definition:
Ordinal Encoding maps categorical values to integers based on a known order or ranking.

When to use:

- Only for ordinal variables (variables with natural ranking)

Ordinal Encoding

In [177]:
```
# Example: 'furnishingstatus' (ordinal: unfurnished < semi-furnished < furnished)
ordinal_mapping = {'unfurnished': 0, 'semi-furnished': 1, 'furnished': 2}
df['furnishingstatus_ordinal'] = df['furnishingstatus'].map(ordinal_mapping)
df[['furnishingstatus', 'furnishingstatus_ordinal']].head()
```

Out[177]:

	furnishingstatus	furnishingstatus_ordinal
0	furnished	2
1	furnished	2
2	semi-furnished	1
3	furnished	2
4	furnished	2

D.Target Encoding

Definition:

Target Encoding replaces each category with the mean of the target variable (or smoothed mean for stability).

When to use:

Useful for high-cardinality categorical variables.

Can improve model performance but may risk overfitting if not carefully applied.

Target Encoding

In [178]:
```
# Example: encode 'prefarea' based on average 'price'
target_mean = df.groupby('prefarea')['price'].mean()
df['prefarea_target'] = df['prefarea'].map(target_mean)

df[['prefarea', 'prefarea_target']].head()
```

Out[178]:

	prefarea	prefarea_target
0	yes	0.666254
1	no	0.036926
2	yes	0.666254
3	yes	0.666254
4	no	0.036926

Encoding Type	Variable	Why Appropriate
Label Encoding	furnishingstatus	Converts categories to integers for ordinal/nominal features
One-Hot Encoding	mainroad	Avoids numeric interpretation for nominal variables
Binary Encoding	guestroom	Simple binary mapping for yes/no
Ordinal Encoding	furnishingstatus	Preserves natural order of categories
Target Encoding	prefarea	Uses target variable info for encoding high-cardinality features