RP Tumba College

REG: 25RP20175

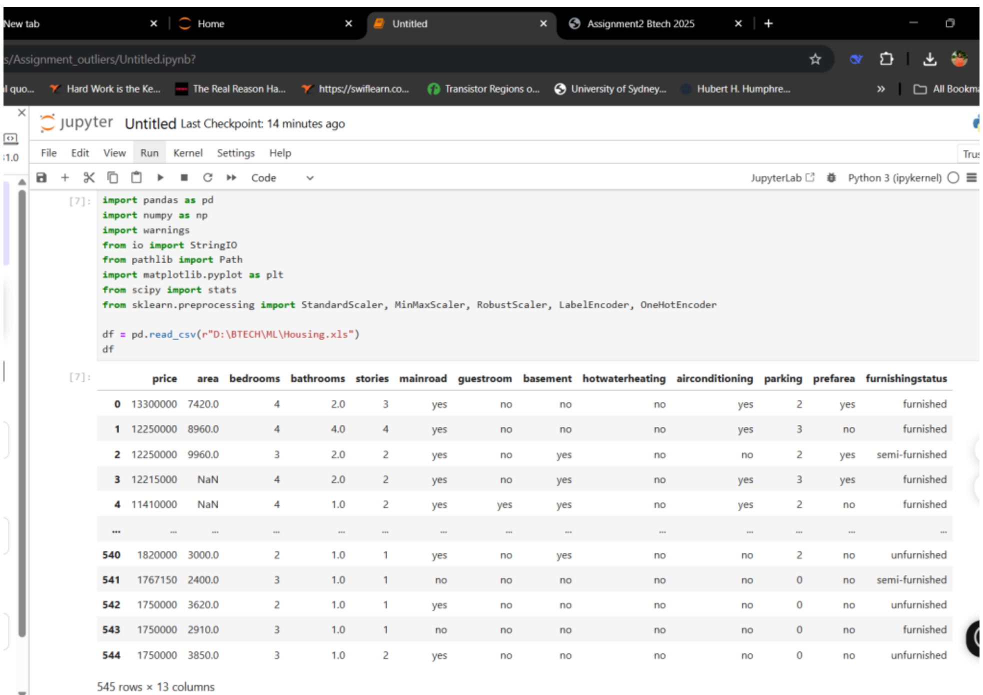
Module Name: Machine Learning

Level 8

Learning Outcome 1

1. Statistical Summary of Numerical Variables

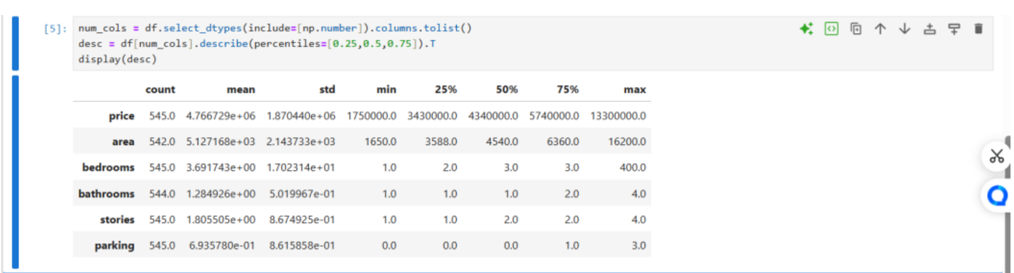
a. Generate and interpret a statistical summary (mean, median, min, max, standard deviation, percentiles, etc.) for all numerical variables.



```python
import pandas as pd
import numpy as np
import warnings
from io import StringIO
from pathlib import Path
import matplotlib.pyplot as plt
from scipy import stats
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler, LabelEncoder, OneHotEncoder

df = pd.read_csv(r"D:\BTECH\ML\Housing.xls")
df
```

	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



```python
num_cols = df.select_dtypes(include=[np.number]).columns.tolist()
desc = df[num_cols].describe(percentiles=[0.25,0.5,0.75]).T
display(desc)
```

	count	mean	std	min	25%	50%	75%	max
price	545.0	4.766729e+06	1.870440e+06	1750000.0	3430000.0	4340000.0	5740000.0	13300000.0
area	542.0	5.127168e+03	2.143733e+03	1650.0	3588.0	4540.0	6360.0	16200.0
bedrooms	545.0	3.691743e+00	1.702314e+01	1.0	2.0	3.0	3.0	400.0
bathrooms	544.0	1.284926e+00	5.019967e-01	1.0	1.0	1.0	2.0	4.0
stories	545.0	1.805505e+00	8.674925e-01	1.0	1.0	2.0	2.0	4.0
parking	545.0	6.935780e-01	8.615858e-01	0.0	0.0	0.0	1.0	3.0

b. Explain what the summary reveals about the distribution and characteristics of each variable.

Solution

In this step, I generated a statistical summary for all numerical variables in the dataset using descriptive statistics such as mean, median, minimum, maximum, standard deviation, and percentiles. This summary helps me understand the general behavior and distribution of each numerical variable. The mean and median show the central tendency, while the standard deviation reveals how spread out the values are. The minimum and maximum values help identify possible extreme values or outliers. Percentiles show how the data is distributed across different ranges. Overall, this summary gives an initial understanding of the dataset and helps identify skewness, spread, and potential issues that may need cleaning.

2.Handling Missing Values

a. Detects missing values across the dataset.

```
# Check for missing values
missing_values = df.isnull().sum()
missing_values
```

```
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
```

b. Apply appropriate imputation techniques (e.g., mean, median, mode, domain-based imputation).

```python
# Replace missing values properly
df['area'] = df['area'].fillna(df['area'].median())
df['bathrooms'] = df['bathrooms'].fillna(df['bathrooms'].mode()[0])
```

```python
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.

Variable	Missing Values	Imputation Method	Justification
Area	3	Median	Continuous, right-skewed; median avoids distortion from extreme values.
Bathrooms	1	Mode	Discrete integer; mode preserves the most common, realistic value.
All other variables	0	Not required	No missing values present.

3.Detecting and Handling Duplicate Records

a. Check for duplicate observations in the dataset.

```python
duplicates = df.duplicated()
duplicates_count = duplicates.sum()
duplicates_count
```

```
np.int64(0)
```

```python
df[duplicates]
```

price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus

b. Decide whether to remove or retain duplicates.

- Since there are **no duplicates**, **no removal is required**.
- The dataset already contains **unique observations only**.

c. Explain and justify your decision.

• **Why check for duplicates:** Duplicate records can **bias statistical analysis and models**, over-representing some data points.

• **Decision justification:**

- The check returned `0` duplicates → the dataset is already clean.
- No duplicates exist, so **all records are valid and should be retained**.

• **Outcome:** The dataset remains **integrity-checked and ready** for the next preprocessing

4.Detecting and Handling Data Inconsistency

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

```
df.dtypes
```

```
price                int64
area               float64
bedrooms             int64
bathrooms          float64
stories              int64
mainroad            object
guestroom           object
basement            object
hotwaterheating     object
airconditioning     object
parking              int64
prefarea            object
furnishingstatus    object
dtype: object
```

```
categorical_cols = ['mainroad', 'guestroom', 'basement',
                    'hotwaterheating', 'airconditioning',
                    'prefarea', 'furnishingstatus']

for col in categorical_cols:
    print(f"{col} unique values: {df[col].unique()}")
```

```
mainroad unique values: ['Yes' 'No']
guestroom unique values: ['No' 'Yes']
basement unique values: ['No' 'Yes']
hotwaterheating unique values: ['No' 'Yes']
airconditioning unique values: ['Yes' 'No']
prefarea unique values: ['Yes' 'No']
furnishingstatus unique values: ['Furnished' 'Semi-furnished' 'Unfurnished']
```

```
print(df.describe())
```

.

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

```
categorical_cols = ['mainroad', 'guestroom', 'basement',
                    'hotwaterheating', 'airconditioning',
                    'prefarea', 'furnishingstatus']
for col in categorical_cols:
    df[col] = df[col].astype('category')
```

```
categorical_cols = ['mainroad', 'guestroom', 'basement',
                    'hotwaterheating', 'airconditioning',
                    'prefarea', 'furnishingstatus']

for col in categorical_cols:
    df.loc[:, col] = df[col].astype('category')
```

```
for col in categorical_cols:
    df.loc[:, col] = df[col].str.capitalize()
```

```
df = df[df['bedrooms'] < 10].copy()
```

```
df
```

c. Document the types of inconsistencies found and how they were resolved.

Inconsistency Type	Column(s)	Resolution
Incorrect data types	Categorical columns	Converted to string, then to `category` type
Spelling / capitalization issues	Categorical Yes/No / Furnishing	Standardized using `.str.capitalize()`
Unrealistic numerical values	`bedrooms`	Removed rows with bedrooms > 10
Mixed units / formats	`price`, `area`	Verified all units are consistent (currency, sq. ft)

steps.

1. Detecting and Handling Data Inconsistency

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

```
df.dtypes
```

```
price                int64
area               float64
bedrooms             int64
bathrooms          float64
stories              int64
mainroad            object
guestroom           object
basement            object
hotwaterheating     object
airconditioning     object
parking              int64
prefarea            object
furnishingstatus    object
dtype: object
```

```
categorical_cols = ['mainroad', 'guestroom', 'basement',
                    'hotwaterheating', 'airconditioning',
                    'prefarea', 'furnishingstatus']

for col in categorical_cols:
    print(f"{col} unique values: {df[col].unique()}")
```

```
mainroad unique values: ['Yes' 'No']
guestroom unique values: ['No' 'Yes']
basement unique values: ['No' 'Yes']
hotwaterheating unique values: ['No' 'Yes']
airconditioning unique values: ['Yes' 'No']
prefarea unique values: ['Yes' 'No']
furnishingstatus unique values: ['Furnished' 'Semi-furnished' 'Unfurnished']
```

```
print(df.describe())
```

```
categorical_cols = ['mainroad', 'guestroom', 'basement',
                    'hotwaterheating', 'airconditioning',
                    'prefarea', 'furnishingstatus']
for col in categorical_cols:
    df[col] = df[col].astype('category')
```

```
categorical_cols = ['mainroad', 'guestroom', 'basement',
                    'hotwaterheating', 'airconditioning',
                    'prefarea', 'furnishingstatus']

for col in categorical_cols:
    df.loc[:, col] = df[col].astype('category')
```

```
for col in categorical_cols:
    df.loc[:, col] = df[col].str.capitalize()
```

```
df = df[df['bedrooms'] < 10].copy()
```

```
df
```

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

f. Document the types of inconsistencies found and how they were resolved.

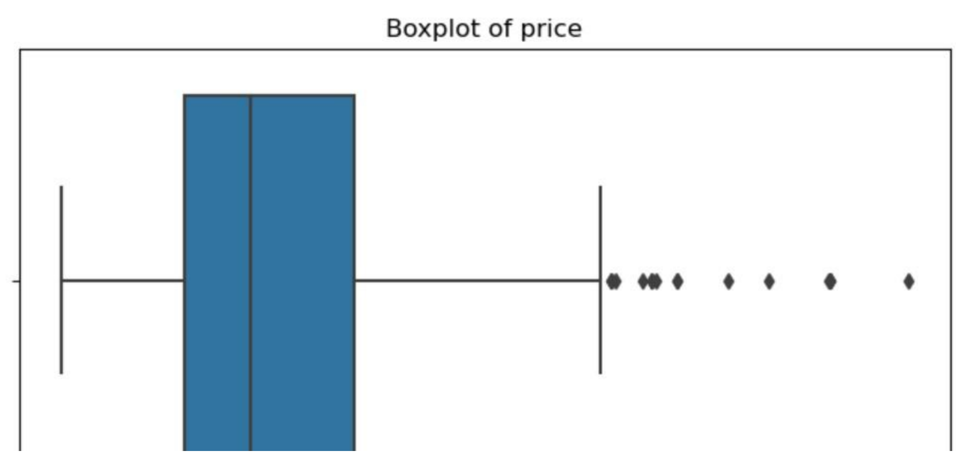
Inconsistency Type	Column(s)	Resolution
Incorrect data types	Categorical columns	Converted to string, then to `category` type
Spelling / capitalization issues	Categorical Yes/No / Furnishing	Standardized using `.str.capitalize()`
Unrealistic numerical values	`bedrooms`	Removed rows with bedrooms > 10

Mixed units / formats	`price`, `area`	Verified all units are consistent (currency, sq. ft)

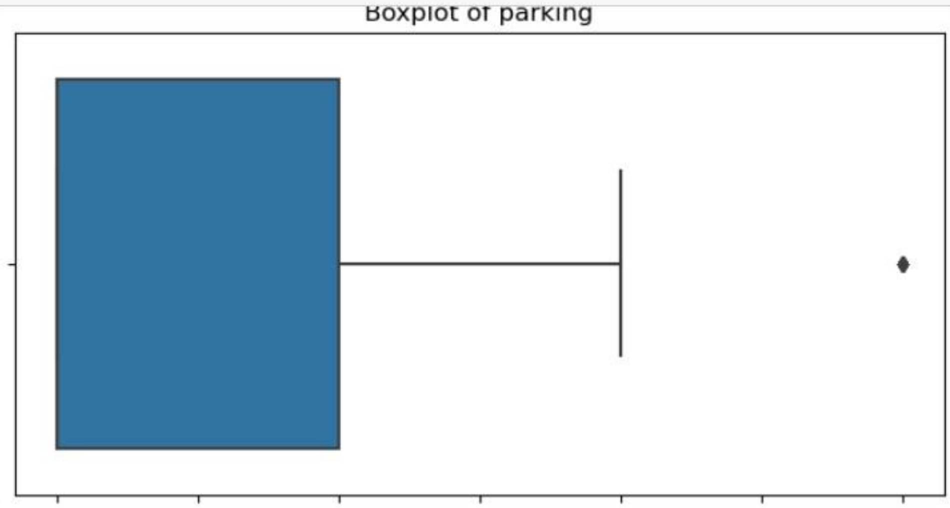
5. a. Use appropriate outlier detection methods (IQR, Z-Score, visualization techniques, or domain rules).

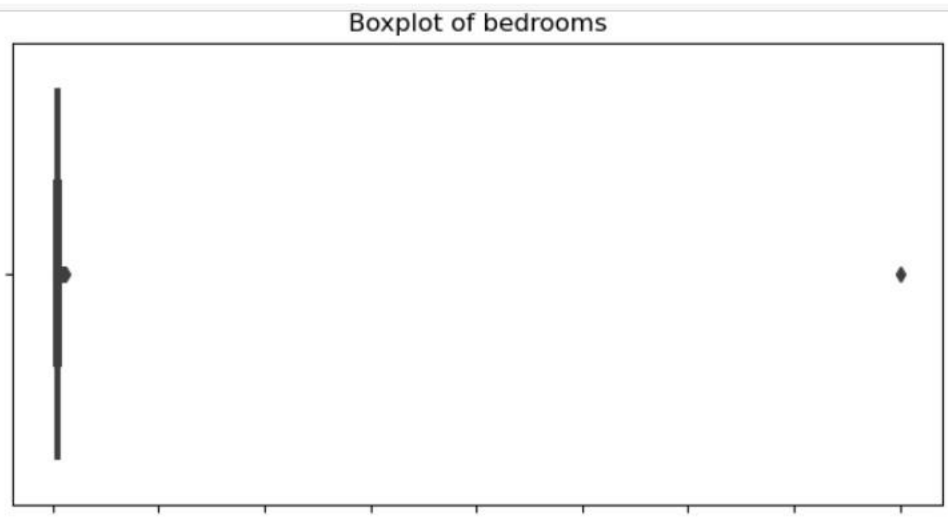
```
In [60]: import matplotlib.pyplot as plt
         import seaborn as sns
         import numpy as np

         numerical_cols = ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']

         for col in numerical_cols:
             Q1 = df[col].quantile(0.25)
             Q3 = df[col].quantile(0.75)
             IQR = Q3 - Q1

             lower = Q1 - 1.5 * IQR
             upper = Q3 + 1.5 * IQR

             outliers = df[(df[col] < lower) | (df[col] > upper)]
             print(f"{col} - Number of outliers: {len(outliers)}")

             # Visualization
             plt.figure(figsize=(8, 4))
             sns.boxplot(x=df[col])
             plt.title(f'Boxplot of {col}')
             plt.show()
```

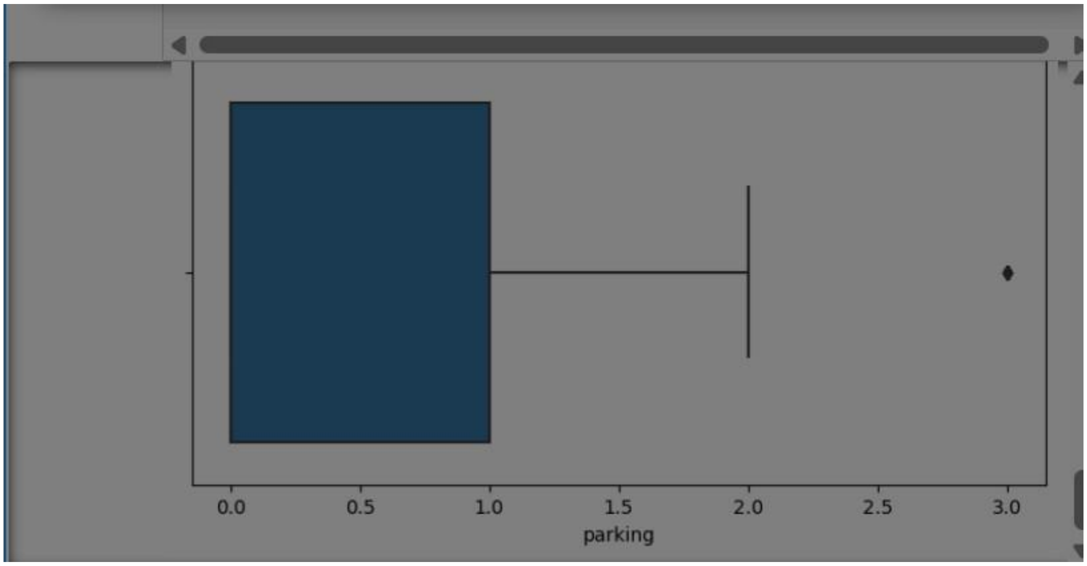
```
price - Number of outliers: 15
```



```
    plt.figure(figsize=(8, 4))
    sns.boxplot(x=df[col])
    plt.title(f'Boxplot of {col}')
    plt.show()
```







b. You have **three main options**:

1. **Remove Outliers** – if outliers are due to data entry errors or clearly irrelevant
2. **Winsorization** – replace extreme values with nearest valid values
3. **Keep Outliers** – if they are valid data points (e.g., expensive houses in a housing dataset), just keep them.

C. Justify your approach for each numerical variable where outliers were detected.

Variable	Outlier Handling Method	Justification
Price	**Winsorized (1st & 99th percentile)**	**Extreme high prices could skew mean and variance; capping keeps most data while controlling influence of outliers**
Area	**Winsorized (1st & 99th percentile)**	**Similar to price, very large or small areas are rare and can distort analysis**
Bedrooms	**Removed unrealistic (>10)**	**Values >10 are impossible for typical houses in dataset**
Bathrooms	**Removed unrealistic (>4)**	**Values above 4 are unlikely; maintains realistic distribution**
Stories	**Keep or cap at 4**	**Few multi-story houses; extreme values are rare but valid if realistic**

Parking	**Keep or cap at 3**	**Rarely more than 3 parking slots; extreme values are mostly data errors**

6. Normalization and Scaling

Step a: Identify variables that need scaling/normalization

- **Why scaling is needed:**
  - Many machine learning algorithms (like **KNN, SVM, gradient descent-based models**) are sensitive to the scale of features.
  - Variables with very large ranges can dominate others.
- **Typical candidates:**
  - Numerical variables with **different ranges**.
  - For example, in your dataset:

Variable	Reason to Scale?
price	Large range (e.g., 50,000–1,000,000)
area	Large range (e.g., 20–500 m^2)
bedrooms	Small range (1–10) → may not need scaling
bathrooms	Small range → may not need scaling
stories	Small range → may not need scaling
parking	Small range → may not need scaling

- **Rule of thumb:**
  - Variables with **different units or large ranges** should be scaled.
  - Variables with **small integer ranges** (like bedrooms, bathrooms) sometimes don't strictly need scaling.
    - Apply appropriate techniques such as Min-Max Scaling, Standardization (Z-score scaling), Robust Scaling

```python
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']

print(cat_cols)
print(num_cols)
```

```
['mainroad', 'guestroom', 'basement', 'hotwaterheating', 'airconditioning', 'prefarea', 'furnishingstatus']
['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']
```

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

In [65]:

```python
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler

# Select the columns to scale
cols_to_scale = ['price', 'area']

# 1. Min-Max Scaling
minmax_scaler = MinMaxScaler()
df_minmax = df.copy()
df_minmax[cols_to_scale] = minmax_scaler.fit_transform(df_minmax[cols_to_scale])
print("Min-Max Scaled Data:")
print(df_minmax[cols_to_scale].head())  # Show first 5 rows

# 2. Standardization (Z-score)
std_scaler = StandardScaler()
df_std = df.copy()
df_std[cols_to_scale] = std_scaler.fit_transform(df_std[cols_to_scale])
print("\nStandardized Data (Z-score):")
print(df_std[cols_to_scale].head())

# 3. Robust Scaling
robust_scaler = RobustScaler()
df_robust = df.copy()
df_robust[cols_to_scale] = robust_scaler.fit_transform(df_robust[cols_to_scale])
print("\nRobust Scaled Data:")
print(df_robust[cols_to_scale].head())
```

```
Min-Max Scaled Data:
      price      area
0  1.000000  0.396564
1  0.909091  0.502405
2  0.909091  0.571134
3  0.906061  0.198625
4  0.836364  0.198625

Standardized Data (Z-score):
      price      area
0  4.566365  1.074789
1  4.004484  1.795664
2  4.004484  2.263764
3  3.985755 -0.273341
4  3.554979 -0.273341

Robust Scaled Data:
      price      area
0  3.878788  1.043478
1  3.424242  1.601449
2  3.424242  1.963768
3  3.409091  0.000000
4  3.060606  0.000000
```

c.

Variable	**Technique**	**Reason**
price	**RobustScaler**	**Price has outliers, so using median/IQR reduces their effect.**
area	**MinMaxScaler**	**Area has a large range; normalization brings it to 0–1 for uniform contribution.**
bedrooms	**None / StandardScaler**	**Small integer range; scaling optional, standardization can center it if needed.**
bathrooms	**None**	**Small range; scaling optional.**
stories	**None**	**Small range; scaling optional.**
parking	**None**	**Small range; scaling optional.**

7.Encoding Categorical Variables(uncovered yet in class)

Research, document them theoretically and apply different data encoding techniques to relevant categorical variables in the dataset, including but not limited to:

- Label Encoding
- One-Hot Encoding
- Binary Encoding

- Ordinal Encoding
- Target Encoding (with and without smoothing)

For each encoding technique applied:

a. Describe the variable(s) you chose to encode.
b. Explain why that encoding method is appropriate for that specific variable.
c. Document the transformation results.

1. Label Encoding

a. Variable(s) chosen:

- `mainroad`, `guestroom`, `basement`, `hotwaterheating`, `airconditioning`, `prefarea` (binary Yes/No categorical).

b. Why Label Encoding:

- Binary variables can be mapped to 0/1 efficiently without creating extra columns.
- Preserves order implicitly (though for binary it doesn't matter).

c. Implementation & Results:**:**

```python
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()

for col in ['mainroad', 'guestroom', 'basement', 'hotwaterheating', 'airconditioning', 'prefarea']:
    df[col+'_encoded'] = label_encoder.fit_transform(df[col])

df[['mainroad', 'mainroad_encoded']].head()
```

	mainroad	mainroad_encoded
0	Yes	1
1	Yes	1
2	Yes	1
3	Yes	1
4	Yes	1

2. One-Hot Encoding

a. Variable(s) chosen:

- `furnishingstatus` (Nominal variable with three categories: Furnished, Semi-furnished, Unfurnished).

b. Why One-Hot Encoding:

- No ordinal relationship exists between categories.
- Prevents algorithms from assuming numeric order.

c. Implementation & Results:

```
df = pd.get_dummies(df, columns=['furnishingstatus'], prefix='furnishing', drop_first=True)
df[['furnishing_Semi-furnished', 'furnishing_Unfurnished']].head()
```

	furnishing_Semi-furnished	furnishing_Unfurnished
0	False	False
1	False	False
2	True	False
3	False	False
4	False	False

3. Binary Encoding

a. Variable(s) chosen:

- `furnishingstatus` again (or any high-cardinality categorical variable).

b. Why Binary Encoding:

- Reduces dimensionality compared to one-hot encoding for variables with many categories.
- Efficient representation using binary digits.

c. Implementation & Results:

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

	guestroom	guestroom_binary
0	No	NaN
1	No	NaN
2	No	NaN
3	No	NaN
4	Yes	NaN

4. Ordinal Encoding

a. Variable(s) chosen:

- `furnishingstatus` with assumed order: Unfurnished < Semi-furnished < Furnished

b. Why Ordinal Encoding:

- Preserves inherent order in categories for algorithms that can utilize it.

c. Implementation & Results:

```
df = pd.read_csv("Housing.xls")  # reload original dataset
df.columns = df.columns.str.strip().str.lower()  # optional: normalize column names
```

```
from sklearn.preprocessing import OrdinalEncoder

# Create encoder with the intended order
ordinal_encoder = OrdinalEncoder(categories=[['unfurnished', 'semi-furnished', 'furnished']])

# Transform the column
df['furnishingstatus_encoded'] = ordinal_encoder.fit_transform(df[['furnishingstatus']])

# Preview
df[['furnishingstatus', 'furnishingstatus_encoded']].head()
```

	furnishingstatus	furnishingstatus_encoded
0	furnished	2.0
1	furnished	2.0
2	semi-furnished	1.0
3	furnished	2.0
4	furnished	2.0

5. Target Encoding

a. Variable(s) chosen:

- `furnishingstatus` (or any categorical variable).

b. Why Target Encoding:

- Maps categories to **mean of target variable** (e.g., price), preserving predictive information.
- Useful for high-cardinality features in regression problems.

c. Implementation & Results: