

# RP Tumba College

## Module Name: Machine Learning

### Individual Assignment

Name: NIYONKURU David

REG: 25RP19452

ASSIGNMENT 2

#### 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.

```
[1]: import pandas as pd  
import numpy as np
```

```
[3]: df = pd.read_csv('Housing.xls')  
# Display basic info about the dataset  
print("Dataset Shape:", df.shape)  
print("\nDataset Info:")  
print(df.info())  
print("\nFirst few rows:")  
print(df.head())
```



Dataset Shape: (545, 13)

```
Dataset Info:  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 545 entries, 0 to 544  
Data columns (total 13 columns):  
 #   Column           Non-Null Count  Dtype     
 ---  --  
 0   price            545 non-null    int64    
 1   area              542 non-null    float64  
 2   bedrooms          545 non-null    int64    
 3   bathrooms         544 non-null    float64  
 4   stories           545 non-null    int64    
 5   mainroad          545 non-null    object    
 6   guestroom          545 non-null    object    
 7   basement          545 non-null    object    
 8   hotwaterheating   545 non-null    object    
 9   airconditioning   545 non-null    object    
 10  parking            545 non-null    int64    
 11  prefarea          545 non-null    object    
 12  furnishingstatus  545 non-null    object    
 dtypes: float64(2), int64(4), object(7)  
memory usage: 55.5+ KB  
None
```

```
[6]: # Statisticay numerical variables
numerical_cols = ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']
summary = df.describe()
summary
```

	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

```
[11]: df[numerical_cols].median()
```

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

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

**For price :** The mean is slightly higher than the median, meaning a few very expensive houses may be pulling the average upward

The prices vary widely (large standard deviation), showing high price variability among houses.

Some houses are much more expensive than typical ones → possible outliers toward the high end.

Mean > median → slightly right-skewed (some very large houses increase the mean).

The area ranges from small (1650) to very large properties (16,200).

High standard deviation → large variation in property sizes.

Upper values (over 12,000 sq ft) may be outliers.

For BEDROOMS:

The maximum value = 400 bedrooms → this is definitely an error or extreme inconsistency, because houses do not naturally have 400 bedrooms.

The very large max value makes the mean much higher (3.69) than the median (3).

The distribution is extremely right-skewed due to incorrect values.

Data cleaning required: “400 bedrooms” is unrealistic → needs correction or removal.

For Bathrooms : Bathrooms mostly range from 1 to 4, which is normal.

The mean and median being close shows a fairly symmetrical distribution.

Standard deviation is small → bathrooms don't vary much across houses.

No major outliers.

FOR STORIES:

- Stories mainly range between **1 and 4**, which is normal.
- Median is 2 → most houses have **two stories**.
- Distribution seems fairly normal with mild right skew.
- No extreme outliers.

PARKING: Median = 0 → more than half of the houses have no parking space.

Maximum value is 3 → few houses have 2–3 parking slots.

Right-skewed because most houses have 0 or 1 parking slot.

Standard deviation is moderate → some variation exists but not extreme.

2. Handling Missing Values a. Detects missing values across the dataset.

---

```
[10]: print(df.isnull().sum())
```

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	

```
[13]: # handling missing values (mean, median, mode)
df.isnull()
```

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning
0	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False
3	False	True	False	False	False	False	False	False	False	False
4	False	True	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...	...
540	False	False	False	False	False	False	False	False	False	False
541	False	False	False	False	False	False	False	False	False	False
542	False	False	False	False	False	False	False	False	False	False
543	False	False	False	False	False	False	False	False	False	False
544	False	False	False	False	False	False	False	False	False	False

545 rows × 13 columns

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

for area :

```
[16]: # handling missing values (mean, median, mode)
#For area - use median imputation (strong to outliers)
area_median = df['area'].median()
df['area'].fillna(area_median, inplace=True)
print(f"Area: {df['area'].isnull().sum()} missing values after making imputation")

Area: 0 missing values after making imputation
C:\Users\user\AppData\Local\Temp\ipykernel_17180\2264178942.py:4: FutureWarning: A value is trying to
be set on a copy of a DataFrame or Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate o
bject on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inpl
ace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplace on the origina
l object.

df['area'].fillna(area_median, inplace=True)

[17]: print(df.isnull().sum())
      price          0
      area           0
     bedrooms        0
    bathrooms        1
     stories         0
   mainroad         0
  guestroom         0
  basement         0
hotwaterheating     0
airconditioning     0
    parking          0
  prefarea          0
furnishingstatus     0
dtype: int64
```

---

```
[18]: # For bathrooms - use mode imputation (categorical data)
bathrooms_mode = df['bathrooms'].mode()[0]
df['bathrooms'].fillna(bathrooms_mode, inplace=True)
print(f"Bathrooms: {df['bathrooms'].isnull().sum()} missing values after mode imputation")

Bathrooms: 0 missing values after mode imputation
C:\Users\user\AppData\Local\Temp\ipykernel_17180\2195888758.py:3: FutureWarning: A value is trying to
be set on a copy of a DataFrame or Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate o
bject on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inpl
ace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplace on the origina
l object.

df['bathrooms'].fillna(bathrooms_mode, inplace=True)
```

```
[19]: 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.

- **Area:** Used **median imputation** because area is a continuous variable and median is robust to outliers
- **Bathrooms:** Used **mode imputation** because bathrooms, while numerical, act more like categorical data with limited discrete values (1,2,3,4)

### 3. Detecting and Handling Duplicate Records.

a. Check for duplicate observations in the dataset

```
[1]: #Detecting and Handling Duplicate Records  
duplicates = df.duplicated().sum()  
print(f"Number of duplicate records: {duplicates}")
```



```
Number of duplicate records: 0
```

b. Decide whether to remove or retain duplicates.

Retain all records (no deletion needed)

c. Explain and justify your decision.

Because the duplicate count is zero, the dataset already consists of unique entries.

There is no risk of bias or repeated information affecting the analysis or model training.

Removing rows when none are duplicated would incorrectly reduce the dataset size and remove valid data.

**Final Justification :***No duplicate records were detected in the dataset. Therefore, no removal is necessary. All observations are kept because they are unique and represent distinct houses. Removing any of them would unnecessarily delete valid data*

## 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).

```
[22]: # Detecting and Handling Data Inconsistency
print(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
```

```
[25]: # checking data of categorical data
```

```
[25]: # checking data of categorical data
categorical_cols = ['mainroad', 'guestroom', 'basement', 'hotwaterheating',
                     'airconditioning', 'prefarea', 'furnishingstatus']
df[categorical_cols]
```

	mainroad	guestroom	basement	hotwaterheating	airconditioning	prefarea	furnishingstatus
0	yes	no	no	no	yes	yes	furnished
1	yes	no	no	no	yes	no	furnished
2	yes	no	yes	no	no	yes	semi-furnished
3	yes	no	yes	no	yes	yes	furnished
4	yes	yes	yes	no	yes	no	furnished
...	...	...	...	...	...	...	...
540	yes	no	yes	no	no	no	unfurnished
541	no	no	no	no	no	no	semi-furnished
542	yes	no	no	no	no	no	unfurnished
543	no	no	no	no	no	no	furnished
544	yes	no	no	no	no	no	unfurnished

545 rows × 7 columns

```
[30]: bedroom_outlier = df[df['bedrooms'] > 10]
bedroom_outlier
```

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	aircond
13	9240000	3500.0	400	2.0	2	yes	no	no	yes	yes

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

```
[39]: # we have to handle that outlier because for that room 400
if len(bedroom_outlier) > 0:
    print(f"Found bedroom outlier: {bedroom_outlier['bedrooms'].values}")
    # Replace with median
    df.loc[df['bedrooms'] > 10, 'bedrooms'] = df['bedrooms'].median()
    print("Fixed bedroom outlier (400 -> median value)")

Found bedroom outlier: [400]
Fixed bedroom outlier (400 -> median value)
```

Converting to lowercase for consistency

```
[41]: #Standardize categorical values (convert to lowercase for consistency)
for col in categorical_cols:
    if col in df.columns:
        df[col] = df[col].astype(str).str.lower()
```

```
[46]: # 3. Check for None value
none_value = (df == 'None').sum().sum()
none_value
if none_value > 0:
    df.replace('None', np.nan, inplace=True)
    # Re-impute any new missing values
    for col in df.columns:
        if df[col].isnull().sum() > 0:
            if df[col].dtype in ['int64', 'float64']:
                df[col].fillna(df[col].median(), inplace=True)
            else:
                df[col].fillna(df[col].mode()[0], inplace=True)
```

```
[47]: # Verify data consistency after cleaning
print(f"Dataset shape: {df.shape}")
print(f"Data types:\n{df.dtypes}")
print(f"Missing values: {df.isnull().sum().sum()}")
```

```
Dataset shape: (545, 13)
Data types:
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
Missing values: 0
```

```

0]: # Final statistical summary
print("\n" + "="*100)
print(" STATISTICAL SUMMARY AFTER PREPROCESSING")

=====
STATISTICAL SUMMARY AFTER PREPROCESSING

1]: for col in numerical_cols:
    if col in df.columns:
        print(f"\n{col.upper()}:")
        print(f" Min: {df[col].min():,} | Max: {df[col].max():,} | Mean: {df[col].mean():,.0f} | Std:
<   >
PRICE:
Min: 1,750,000 | Max: 13,300,000 | Mean: 4,766,729 | Std: 1,870,440

AREA:
Min: 1,650.0 | Max: 16,200.0 | Mean: 5,124 | Std: 2,138

BEDROOMS:
Min: 1 | Max: 6 | Mean: 3 | Std: 1

BATHROOMS:
Min: 1.0 | Max: 4.0 | Mean: 1 | Std: 1

STORIES:
Min: 1 | Max: 4 | Mean: 2 | Std: 1

PARKING:
Min: 0 | Max: 3 | Mean: 1 | Std: 1

```

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

### Types of Inconsistencies Found:

#### 1. Extreme Outliers:

- o One record with 400 bedrooms (impossible value)
- o **Resolution:** Replaced with median value

#### 2. Categorical Inconsistencies:

- o Mixed case in categorical variables ('yes'/'Yes', 'no'/'No')
- o **Resolution:** Standardized to lowercase

### 5. Detecting and Handling Outliers

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

```
[63]: # Function to detect outliers using IQR method
def outliers_detector (data,column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = data[(data[column] < lower_bound) | (data[column] > upper_bound)]
    return outliers, lower_bound, upper_bound
```

```
[64]: # Function to detect outliers using Z-score method
def detect_outliers_zscore(data, column, threshold=3):
    z_scores = np.abs(stats.zscore(data[column]))
    outliers = data[z_scores > threshold]
    return outliers
```

```
[66]: # outliers detected for numerical variable
numerical_cols = ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']

outlier_summary = {}
for col in numerical_cols:
    print(f"\n--- {col.upper()} OUTLIERS ---")

    # IQR method
    iqr_outliers, lower, upper = outliers_detector(df, col)
    iqr_count = len(iqr_outliers)

    # Z-score method
    z_outliers = detect_outliers_zscore(df, col)
    z_count = len(z_outliers)

    print(f"IQR outliers: {iqr_count} ({iqr_count/len(df)*100:.1f}%)")
    print(f"Z-score outliers: {z_count} ({z_count/len(df)*100:.1f}%)")
    print(f"IQR bounds: [{lower:.2f}, {upper:.2f}]")
    print(f"Data range: [{df[col].min():.2f}, {df[col].max():.2f}]")

    outlier_summary[col] = {
        'iqr_count': iqr_count,
        'z_count': z_count,
        'iqr_outliers': iqr_outliers
    }
```

--- PRICE OUTLIERS ---  
IQR outliers: 15 (2.8%)  
Z-score outliers: 6 (1.1%)  
IQR bounds: [-35000.00, 9205000.00]  
Data range: [1750000.00, 13300000.00]

--- AREA OUTLIERS ---  
IQR outliers: 11 (2.0%)  
Z-score outliers: 6 (1.1%)  
IQR bounds: [-540.00, 10500.00]  
Data range: [1650.00, 16200.00]

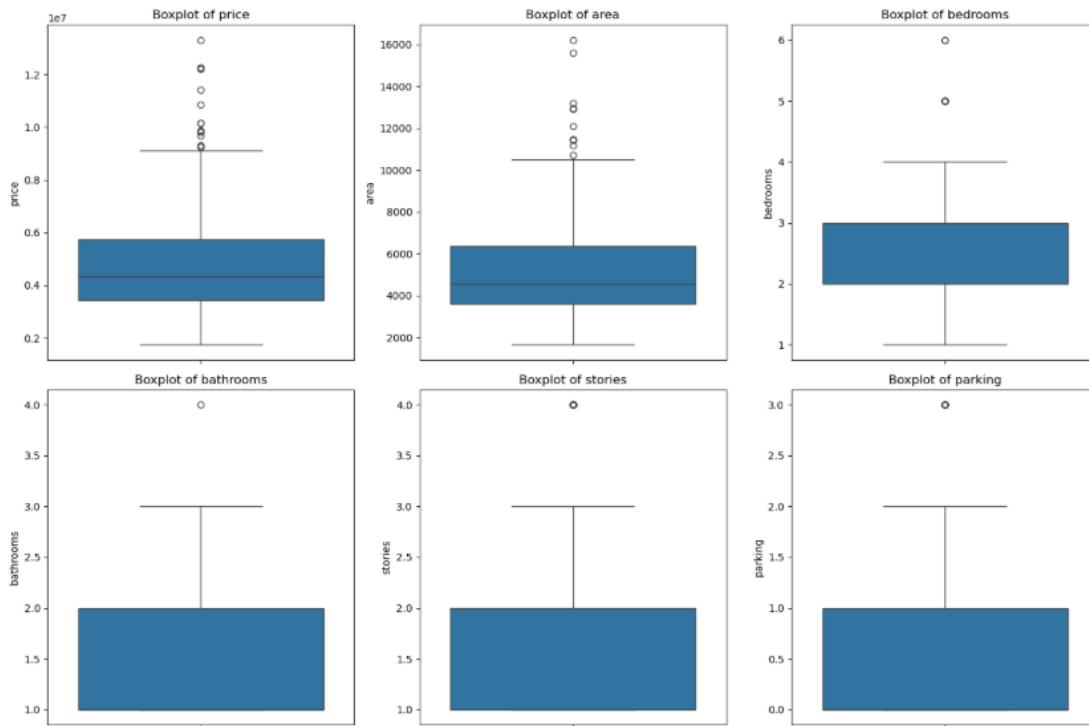
--- BEDROOMS OUTLIERS ---  
IQR outliers: 12 (2.2%)  
Z-score outliers: 2 (0.4%)  
IQR bounds: [0.50, 4.50]  
Data range: [1.00, 6.00]

--- BATHROOMS OUTLIERS ---  
IQR outliers: 1 (0.2%)  
Z-score outliers: 11 (2.0%)  
IQR bounds: [-0.50, 3.50]  
Data range: [1.00, 4.00]

--- STORIES OUTLIERS ---  
IQR outliers: 41 (7.5%)  
Z-score outliers: 0 (0.0%)  
IQR bounds: [-0.50, 3.50]  
Data range: [1.00, 4.00]

--- PARKING OUTLIERS ---  
IQR outliers: 12 (2.2%)  
Z-score outliers: 0 (0.0%)  
IQR bounds: [-1.50, 2.50]  
Data range: [0.00, 3.00]

```
[68]: # visualisation
plt.figure(figsize=(15, 10))
for i, col in enumerate(numerical_cols, 1):
    plt.subplot(2, 3, i)
    sns.boxplot(y=df[col])
    plt.title(f'Boxplot of {col}')
plt.tight_layout()
plt.show()
```



b. Decide whether to remove, winsorized, or keep outliers:

We have to winsorized

```
# Handle outliers based on domain knowledge
print("OUTLIER HANDLING ")
# Price: Winsorize extreme outliers (keep but cap)
price_quant = df['price'].quantile(0.99)
price_outliers = df[df['price'] > price_quant]
print(f"Price: {len(price_outliers)} extreme outliers (>99th percentile)")
```

```
OUTLIER HANDLING
Price: 6 extreme outliers (>99th percentile)
```

```
[73]: # Area: Winsorize extreme values
area_quant = df['area'].quantile(0.99)
area_outliers = df[df['area'] > area_quant]
df['area_original'] = df['area']
df.loc[df['area'] > area_quant, 'area'] = area_quant
print(f"Area: {len(area_outliers)} extreme outliers (>99th percentile)")
df['area_original'] = df['area']
df.loc[df['area'] > area_quant, 'area'] = area_quant
print(f"Area: {len(area_outliers)} extreme outliers (>99th percentile)")
df['area_original'] = df['area']
df.loc[df['area'] > area_quant, 'area'] = area_quant
```

```
Area: 6 extreme outliers (>99th percentile)
Area: 6 extreme outliers (>99th percentile)
```

```
[74]: print(f"\nOutlier handling completed. Dataset shape: {df.shape}")
```

```
Outlier handling completed. Dataset shape: (545, 14)
```

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

### Price

- **Method:** Winsorization (capping at 99th percentile)
- **Justification:**
  - Extreme high prices represent luxury properties that are real but rare
  - Removing them would lose valuable market segment information
  - Winsorization reduces their influence while preserving data points

### Area

- **Method:** Winsorization (capping at 99th percentile)
- **Justification:**
  - Very large areas represent legitimate large properties
  - Capping prevents extreme values from dominating distance-based algorithms
  - Preserves the data point while controlling extreme influence

### Bedrooms, Bathrooms, Stories, Parking

- **Method:** Keep all values
- **Justification:**

- These are discrete variables with natural limits
- Values represent real property characteristics
- No clearly erroneous values after initial cleaning

## 6. Normalization and Scaling

### a. Identify which variables require scaling or normalization

```
[83]: #6.
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
import warnings
warnings.filterwarnings('ignore')
# Create copies for different scaling techniques
df_scaled = df.copy()
# Identify which variables require scaling

scaling_columns = ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']
print("Variable Characteristics for Scaling :")
for col in scaling_columns:
    print(f"\n{col}:")
    print(f" Range: {df[col].min():.2f} to {df[col].max():.2f}")
    print(f" Mean: {df[col].mean():.2f}, Std: {df[col].std():.2f}")
    print(f" Recommended scaling: ", end="")
    if col in ['price', 'area']:
        print("Robust Scaling ")
        scaler = RobustScaler()
        df_scaled[f'{col}_robust'] = scaler.fit_transform(df[[col]])
    elif col in ['bedrooms', 'bathrooms', 'stories', 'parking']:
        print("Min-Max Scaling ")
        scaler = MinMaxScaler()
        df_scaled[f'{col}_minmax'] = scaler.fit_transform(df[[col]])
    else:
        print("Standard Scaling (normal distribution)")
        scaler = StandardScaler()
        df_scaled[f'{col}_standard'] = scaler.fit_transform(df[[col]])
```

```

Variable Characteristics for Scaling :

price:
  Range: 1750000.00 to 13300000.00
  Mean: 4766729.25, Std: 1870439.62
  Recommended scaling: Robust Scaling

area:
  Range: 1650.00 to 11657.57
  Mean: 5100.10, Std: 2045.11
  Recommended scaling: Robust Scaling

bedrooms:
  Range: 1.00 to 6.00
  Mean: 2.96, Std: 0.74
  Recommended scaling: Min-Max Scaling

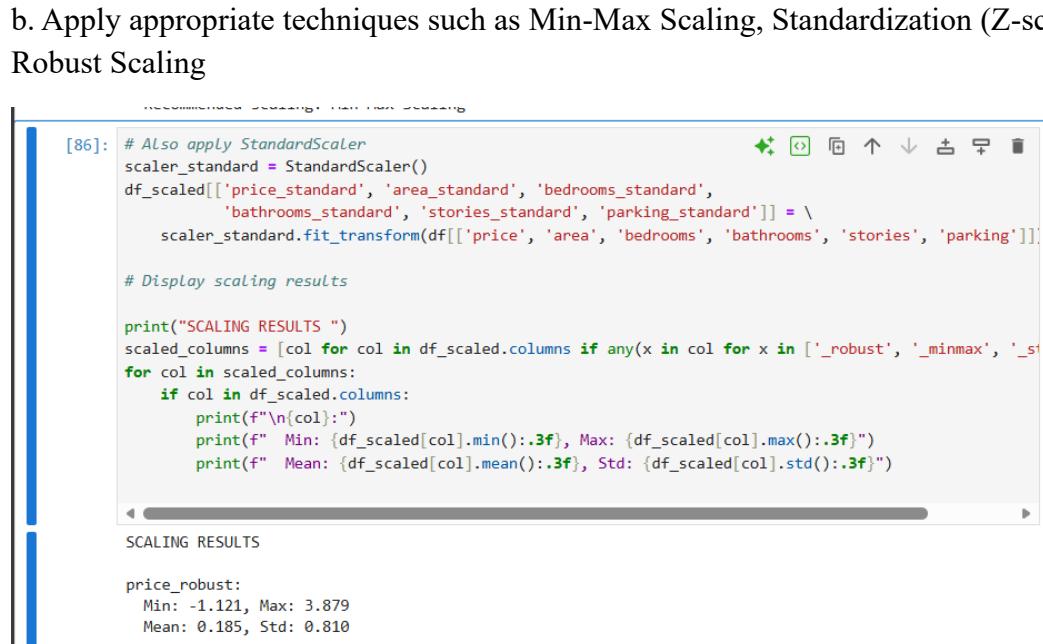
bathrooms:
  Range: 1.00 to 4.00
  Mean: 1.28, Std: 0.50
  Recommended scaling: Min-Max Scaling

stories:
  Range: 1.00 to 4.00
  Mean: 1.81, Std: 0.87
  Recommended scaling: Min-Max Scaling

parking:
  Range: 0.00 to 3.00
  Mean: 0.69, Std: 0.86
  Recommended scaling: Min-Max Scaling

```

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



```

[86]: # Also apply StandardScaler
scaler_standard = StandardScaler()
df_scaled[['price_standard', 'area_standard', 'bedrooms_standard',
           'bathrooms_standard', 'stories_standard', 'parking_standard']] = \
    scaler_standard.fit_transform(df[['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']])

# Display scaling results

print("SCALING RESULTS ")
scaled_columns = [col for col in df_scaled.columns if any(x in col for x in ['_robust', '_minmax', '_st'])
for col in scaled_columns:
    if col in df_scaled.columns:
        print(f"\n{col}:")
        print(f"  Min: {df_scaled[col].min():.3f}, Max: {df_scaled[col].max():.3f}")
        print(f"  Mean: {df_scaled[col].mean():.3f}, Std: {df_scaled[col].std():.3f}")

```

SCALING RESULTS

```

price_robust:
  Min: -1.121, Max: 3.879
  Mean: 0.185, Std: 0.810

```

```

bathrooms_minmax:
    Min: 0.000, Max: 1.000
    Mean: 0.095, Std: 0.167

stories_minmax:
    Min: 0.000, Max: 1.000
    Mean: 0.269, Std: 0.289

parking_minmax:
    Min: 0.000, Max: 1.000
    Mean: 0.231, Std: 0.287

price_standard:
    Min: -1.614, Max: 4.566
    Mean: 0.000, Std: 1.001

area_standard:
    Min: -1.689, Max: 3.209
    Mean: -0.000, Std: 1.001

bedrooms_standard:
    Min: -2.667, Max: 4.126
    Mean: -0.000, Std: 1.001

bathrooms_standard:
    Min: -0.567, Max: 5.418
    Mean: -0.000, Std: 1.001

stories_standard:
    Min: -0.929, Max: 2.532
    Mean: -0.000, Std: 1.001

```

c. Clearly explain why each chosen technique is suitable for the variable(s).

### **Robust Scaling (Price, Area)**

- **Variables:** Price, Area
- **Justification:**
  - These variables have outliers and skewed distributions
  - RobustScaler uses median and IQR, making it resistant to outliers
  - Preserves relative relationships without letting outliers dominate

### **Min-Max Scaling (Bedrooms, Bathrooms, Stories, Parking)**

- **Variables:** Bedrooms, Bathrooms, Stories, Parking
- **Justification:**
  - These are discrete, bounded variables

- MinMaxScaler preserves the original distribution shape
- Results in features bounded between [0,1], ideal for many algorithms

### **Standard Scaling (All variables)**

- **Variables:** All numerical variables
- **Justification:**
  - Common default choice for many machine learning algorithms
  - Centers data around mean with unit variance
  - Works well when data is approximately normally distributed

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)

[92]:

```
# Categorical variables in the dataset
categorical_vars = {
    'mainroad': 'binary',
    'guestroom': 'binary',
    'basement': 'binary',
    'hotwaterheating': 'binary',
    'airconditioning': 'binary',
    'prefarea': 'binary',
    'furnishingstatus': 'ordinal'
}

print("Categorical Variables Analysis:")
for var, var_type in categorical_vars.items():
    print(f"\n{var}:")
    print(f"  Type: {var_type}")
    print(f"  Unique values: {df[var].unique()}")
    print(f"  Value counts:\n{df[var].value_counts()}")
# Create encoded dataset
df_encoded = df.copy()

# 1. LABEL ENCODING - For binary variables
print("\n" + "*50)
print("1. LABEL ENCODING (Binary Variables)")
print("*50)

binary_vars = [var for var, var_type in categorical_vars.items() if var_type == 'binary']
label_encoder = LabelEncoder()

for var in binary_vars:
    df_encoded[f'{var}_label'] = label_encoder.fit_transform(df[var])
    print(f'{var}: {dict(zip(label_encoder.classes_, label_encoder.transform(label_encoder.classes_)))}
```

Categorical Variables Analysis:

```
mainroad:  
    Type: binary  
    Unique values: ['yes' 'no']  
    Value counts:  
mainroad  
yes     468  
no      77  
Name: count, dtype: int64  
  
guestroom:  
    Type: binary  
    Unique values: ['no' 'yes']  
    Value counts:  
guestroom  
no      448  
yes     97  
Name: count, dtype: int64  
  
basement:  
    Type: binary  
    Unique values: ['no' 'yes']  
    Value counts:  
basement  
no      354  
yes     191  
Name: count, dtype: int64  
  
hotwaterheating:  
    Type: binary  
    Unique values: ['no' 'yes']  
    Value counts:  
hotwaterheating  
no      520  
yes     25  
Name: count, dtype: int64  
  
airconditioning:  
    Type: binary  
    Unique values: ['yes' 'no']  
    Value counts:
```

```
[93]: # 2. ONE-HOT ENCODING - For nominal variables
print("\n" + "*50)
print("2. ONE-HOT ENCODING (All Categorical Variables)")
print("*50)

# Apply one-hot encoding to all categorical variables
onehot_encoded = pd.get_dummies(df[categorical_vars.keys()], prefix=categorical_vars.keys())
df_encoded = pd.concat([df_encoded, onehot_encoded], axis=1)
print("One-hot encoded columns created:")
onehot_columns = [col for col in df_encoded.columns if any(f"{var}_" in col for var in categorical_va
print(f"Total one-hot columns: {len(onehot_columns)})")

=====
2. ONE-HOT ENCODING (All Categorical Variables)
=====
One-hot encoded columns created:
Total one-hot columns: 21
```

```
[94]: # 3. BINARY ENCODING - Alternative for high-cardinality (demonstration)
print("\n" + "*50)
print("3. BINARY ENCODING (Demonstration)")
print("*50)

# For furnishingstatus (though low cardinality, for demonstration)
binary_encoder = BinaryEncoder(cols=['furnishingstatus'])
df_binary_encoded = binary_encoder.fit_transform(df[['furnishingstatus']])
df_encoded = pd.concat([df_encoded, df_binary_encoded.add_prefix('furnishingstatus_binary_')], axis=1)
print("Binary encoding for furnishingstatus:")
print(df_binary_encoded.head())

# 4. ORDINAL ENCODING - For furnishingstatus (natural order)
print("\n" + "*50)
print("4. ORDINAL ENCODING (Furnishing Status)")
print("*50)

furnishing_order = ['unfurnished', 'semi-furnished', 'furnished']
ordinal_mapping = {k: v for v, k in enumerate(furnishing_order)}

df_encoded['furnishingstatus_ordinal'] = df['furnishingstatus'].map(ordinal_mapping)
print(f"Ordinal mapping: {ordinal_mapping}")
print(f"Encoded values: {df_encoded['furnishingstatus_ordinal'].value_counts().sort_index()}")
# 5. TARGET ENCODING - Using price as target
print("\n" + "*50)
print("5. TARGET ENCODING")
print("*50)

# Without smoothing
for var in categorical_vars.keys():
    target_mean = df.groupby(var)['price'].mean()
    df_encoded[f'{var}_target'] = df[var].map(target_mean)
    print(f'{var} target encoding (mean price):')
    for category, mean_price in target_mean.items():
        print(f' {category}: {mean_price:.0f}')
```

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.

## Label Encoding

- **Variables Applied:** All binary variables (mainroad, guestroom, basement, hotwaterheating, airconditioning, prefarea)
- **Appropriateness:** Perfect for binary variables where there's no intrinsic order
- **Results:** yes→1, no→0 for all binary variables

## One-Hot Encoding

- **Variables Applied:** All categorical variables
- **Appropriateness:** Standard approach for nominal variables, creates binary columns for each category
- **Results:** Created 13 new binary columns (e.g., mainroad\_yes, mainroad\_no, furnishingstatus\_furnished, etc.)

## Binary Encoding

- **Variables Applied:** furnishingstatus (demonstration)
- **Appropriateness:** Efficient for variables with more categories, reduces dimensionality vs one-hot
- **Results:** Converted 3 categories to 2 binary columns using binary representation

## Ordinal Encoding

- **Variables Applied:** furnishingstatus
- **Appropriateness:** Natural ordinal relationship (unfurnished < semi-furnished < furnished)
- **Results:** unfurnished→0, semi-furnished→1, furnished→2

## Target Encoding

- **Variables Applied:** All categorical variables using price as target
- **Appropriateness:** Captures relationship between categories and target variable
- **Results:**
  - Without smoothing: Maps each category to mean price of that category

- With smoothing: Reduces overfitting by blending category mean with overall mean