RP Tumba College

Module Name: Machine Learning

Level 8

Learning Outcome 1

25RP18235

1(a)

```
In [7]: import pandas as pd
        df = pd.read_csv("housing.xls")
        #df.head()
```

```
In [6]: numeric_df = df.select_dtypes(include=["int64", "float64"])
        for col in numeric_df.columns:
            print(f"=== Statistics for {col} ===")
            print(f"Count: {numeric_df[col].count()}")
            print(f"Mean: {numeric_df[col].mean()}")
            print(f"Median: {numeric_df[col].median()}")
            print(f"Standard Deviation: {numeric_df[col].std()}")
            print(f"Minimum: {numeric_df[col].min()}")
            print(f"Maximum: {numeric_df[col].max()}")
            print(f"25th Percentile (Q1): {numeric_df[col].quantile(0.25)}")
            print(f"50th Percentile (Median): {numeric_df[col].quantile(0.50)}")
            print(f"75th Percentile (Q3): {numeric_df[col].quantile(0.75)}")
            print(f"10th Percentile: {numeric_df[col].quantile(0.10)}")
            print(f"90th Percentile: {numeric_df[col].quantile(0.90)}")
            print(f"Interquartile Range (IQR): {numeric_df[col].quantile(0.75) - numeric_df[col].quantile(0.25)}")
            print("\n")
```

```
            print("\n")

=== Statistics for price ===
Count: 545
Mean: 4766729.247706422
Median: 4340000.0
Standard Deviation: 1870439.615657394
Minimum: 1750000
Maximum: 13300000
25th Percentile (Q1): 3430000.0
50th Percentile (Median): 4340000.0
75th Percentile (Q3): 5740000.0
10th Percentile: 2835000.0
90th Percentile: 7350000.0
Interquartile Range (IQR): 2310000.0


=== Statistics for area ===
Count: 542
Mean: 5127.167896678967
Median: 4540.0
Standard Deviation: 2143.732761109976
Minimum: 1650.0
Maximum: 16200.0
25th Percentile (Q1): 3588.0
50th Percentile (Median): 4540.0
75th Percentile (Q3): 6360.0
10th Percentile: 3000.0
90th Percentile: 7977.000000000001
Interquartile Range (IQR): 2772.0


=== Statistics for bedrooms ===
Count: 545
Mean: 3.691743119266055
Median: 3.0
Standard Deviation: 17.023135902744258
Minimum: 1
Maximum: 400
25th Percentile (Q1): 2.0
50th Percentile (Median): 3.0
75th Percentile (Q3): 3.0
10th Percentile: 2.0
90th Percentile: 4.0
Interquartile Range (IQR): 1.0


=== Statistics for bathrooms ===
Count: 544
Mean: 1.2849264705882353
Median: 1.0
Standard Deviation: 0.5019967442441436
Minimum: 1.0
Maximum: 4.0
25th Percentile (Q1): 1.0
50th Percentile (Median): 1.0
75th Percentile (Q3): 2.0
10th Percentile: 1.0
90th Percentile: 2.0
Interquartile Range (IQR): 1.0


=== Statistics for stories ===
Count: 545
Mean: 1.8055045871559634
Median: 2.0
Standard Deviation: 0.8674924629255298
Minimum: 1
Maximum: 4
25th Percentile (Q1): 1.0
50th Percentile (Median): 2.0
75th Percentile (Q3): 2.0
10th Percentile: 1.0
90th Percentile: 3.0
Interquartile Range (IQR): 1.0


=== Statistics for parking ===
Count: 545
Mean: 0.6935779816513762
Median: 0.0
Standard Deviation: 0.8615857504605449
Minimum: 0
Maximum: 3
25th Percentile (Q1): 0.0
50th Percentile (Median): 0.0
75th Percentile (Q3): 1.0
10th Percentile: 0.0
90th Percentile: 2.0
Interquartile Range (IQR): 1.0
```

1(b) Interpretation of Numerical Variables

1. Price

- **Mean:** 4,766,729 RWF
- **Median:** 4,340,000 RWF
- **Std Dev:** 1,870,439 RWF
- **Min/Max:** 1,750,000 / 13,300,000 RWF
- **IQR:** 2,310,000 RWF

Interpretation:
The mean is slightly higher than the median, suggesting **a right-skewed distribution** (some expensive houses pulling the average up). The large standard deviation and wide range indicate **high variability in house prices**, with some extreme values likely being outliers.

---

2. Area

- **Mean:** 5,127 sqft
- **Median:** 4,540 sqft
- **Std Dev:** 2,144 sqft
- **Min/Max:** 1,650 / 16,200 sqft
- **IQR:** 2,772 sqft

Interpretation:
Area also shows **right-skewness**, as the mean > median. High max values indicate **some very large houses** in the dataset. Standard deviation is moderate compared to the mean, showing decent variability.

---

3. Bedrooms

- **Mean:** 3.69
- **Median:** 3
- **Std Dev:** 17.02 (!)
- **Min/Max:** 1 / 400
- **IQR:** 1

Interpretation:
The extremely high std deviation and max value (400) are clear **outliers** or data entry errors.

Most houses have 2–4 bedrooms (as seen from Q1, Q3), so the **median and IQR** better represent typical houses.

---

4. Bathrooms

- **Mean:** 1.28
- **Median:** 1
- **Std Dev:** 0.50
- **Min/Max:** 1 / 4
- **IQR:** 1

Interpretation:
Bathrooms are fairly consistent, mostly **1–2 bathrooms per house**. Distribution is slightly right-skewed (mean > median), but there are no extreme outliers.

---

5. Stories

- **Mean:** 1.80
- **Median:** 2
- **Std Dev:** 0.87
- **Min/Max:** 1 / 4
- **IQR:** 1

Interpretation:
Most houses are **1–2 stories**. Distribution is close to symmetric; small number of houses have 3–4 stories.

---

6. Parking

- **Mean:** 0.69
- **Median:** 0
- **Std Dev:** 0.86
- **Min/Max:** 0 / 3
- **IQR:** 1

Interpretation:
Median = 0 shows **most houses have no parking space**. Distribution is skewed, and a few

houses have 1–3 parking spaces. Low mean and high skewness suggest a **sparse parking variable**.

Overall observations:

- bedrooms has a **critical outlier (400 bedrooms)** that must be handled in preprocessing.
- Price and area have **high variability and right skewness**, which may require **normalization or transformation** for modeling.
- Bathrooms, stories, and parking are relatively small-range discrete variables, so **median/mode imputation** or encoding is sufficient later.

1. heyyyyyyyyyyyyyyyyyy
2. Handling Missing Values

    a. Detects missing values across the dataset.

```
In [12]: missing_values = df.isnull().sum()
         print("Missing values per column:\n", missing_values)
         #percentage of missing values
         missing_percentage = (df.isnull().sum() / len(df)) * 100
         print("\nPercentage of missing values per column:\n", missing_percentage)

         Missing values per column:
          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

         Percentage of missing values per column:
          price             0.000000
         area               0.550459
         bedrooms           0.000000
         bathrooms          0.183486
         stories            0.000000
         mainroad           0.000000
         guestroom          0.000000
         basement           0.000000
         hotwaterheating    0.000000
         airconditioning    0.000000
         parking            0.000000
         prefarea           0.000000
         furnishingstatus   0.000000
         dtype: float64
```

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

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

- Use **median** for bedrooms and bathrooms → robust to outliers.
- Use **mean** for price and area → typical central value for skewed distributions.
- Use **mode** for stories and parking → discrete categorical-like variables.

```
In [15]: # Imputation for numerical variables
         # Mean imputation for price and area
         df['price'].fillna(df['price'].mean(), inplace=True)
         df['area'].fillna(df['area'].mean(), inplace=True)

         # Median imputation for bedrooms and bathrooms (robust to outliers)
         df['bedrooms'].fillna(df['bedrooms'].median(), inplace=True)
         df['bathrooms'].fillna(df['bathrooms'].median(), inplace=True)

         # Mode imputation for stories and parking (categorical/discrete)
         df['stories'].fillna(df['stories'].mode()[0], inplace=True)
         df['parking'].fillna(df['parking'].mode()[0], inplace=True)

         # Verify no missing values remain
         print("\nMissing values after imputation:\n", df.isnull().sum())
```

```
Missing values after imputation:
 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	Imputation Method	Justification
price	Mean	Central value is representative; distribution is right-skewed but no extreme outlier in missing entries.
area	Mean	Similar reason as price; mean reflects the typical house size.
bedrooms	Median	High outlier exists (400 bedrooms); median prevents skewing imputation.
bathrooms	Median	Limited range, median ensures typical bathroom count without skew.
stories	Mode	Discrete variable; mode represents the most common house structure.
parking	Mode	Sparse distribution; mode reflects majority of houses with 0 parking.

3. Detecting and Handling Duplicate Records

a. Check for duplicate observations in the dataset.

```
In [17]: # Check for duplicate rows
duplicate_rows = df.duplicated()
print("Number of duplicate rows:", duplicate_rows.sum())
#Display the duplicate rows
duplicates = df[duplicate_rows]
duplicates

Number of duplicate rows: 0

Out[17]:
```

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

b. Decide whether to remove or retain duplicates.

```
In [18]: # Remove duplicate rows
df_cleaned = df.drop_duplicates()
# Verify duplicates are removed
print("Number of rows after removing duplicates:", df_cleaned.shape[0])

Number of rows after removing duplicates: 545
```

- drop_duplicates() removes rows that are **exact copies** of previous rows.
- Keeping duplicates can **bias models**, especially regression or machine learning algorithms.

c. Explain and justify your decision.

Action	Reasoning
Remove duplicates	Exact duplicate entries provide no new information and may bias statistical analysis or machine learning models.
Retain duplicates	Only consider retaining if duplicates are **valid repeated observations** (e.g., multiple identical sales at different times), which is **not the case here**.

In this house dataset, duplicates likely occur due to **data entry errors**, so removing them improves dataset quality.

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).

Figure 1: Check data types

```
In [19]: df.dtypes

Out[19]: 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
```

Figure 2: Look for unrealistic val

```
In [21]: # Describe to spot strange ranges
         df.describe()
         # Check manually suspicious columns
         print("Bedrooms - unique values:", df['bedrooms'].unique())
         print("Stories - unique values:", df['stories'].unique())
         print("Parking - unique values:", df['parking'].unique())

         Bedrooms - unique values: [  4   3   5 400   2   6   1]
         Stories - unique values: [3 4 2 1]
         Parking - unique values: [2 3 0 1]
```

Figure 3: Find categorical inconsistencies

```
In [22]: # Check unique categorical values
         for col in df.select_dtypes(include=['object']).columns:
             print(col, df[col].unique())

         mainroad ['yes' 'no']
         guestroom ['no' 'yes']
         basement ['no' 'yes']
         hotwaterheating ['no' 'yes']
         airconditioning ['yes' 'no']
         prefarea ['yes' 'no']
         furnishingstatus ['furnished' 'semi-furnished' 'unfurnished']
```

Figure 4: Check mixed units (example: area)

```
In [23]: df['area'].sample(10)

Out[23]: 278    3400.0
         53     5150.0
         275    4032.0
         47     6600.0
         301    3520.0
         451    6750.0
         92     4800.0
         528    3970.0
         32     4880.0
         379    3520.0
         Name: area, dtype: float64
```

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

1. Fix incorrect bedroom outlier (400 bedrooms)

This is **impossible**, clearly a data entry error.
Typical houses have 2–4 bedrooms → replace using the **median**.

```
In [38]: # Replace unrealistic bedroom values (>= 20 considered unrealistic)
median_bedrooms = df['bedrooms'].median()
df.loc[df['bedrooms'] >= 20, 'bedrooms'] = median_bedrooms
print(df['bedrooms'].max())

6
```

2. Standardize categorical values

Convert all Yes/No values to lowercase, clean spaces:

```
In [40]: categorical_cols = df.select_dtypes(include='object').columns

for col in categorical_cols:
    df[col] = df[col].str.lower().str.strip()
```

Fix known categorical patterns:

```
In [41]: #Fix known categorical patterns
df['furnishingstatus'] = df['furnishingstatus'].replace({
    'semi-furnished': 'semi furnished',
    'semi furnished ': 'semi furnished',
    'furnished ': 'furnished'
})
```

3. Ensure numerical columns are numeric

Sometimes CSV files load numbers as strings:

```
})
```

```
In [42]: # Ensure numerical columns are numeric
         #Sometimes CSV files load numbers as strings:
         num_cols = ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']

         for col in num_cols:
             df[col] = pd.to_numeric(df[col], errors='coerce')
```

4. Fix parking spaces

Parking cannot be negative or extremely high:

```
In [43]: df.loc[df['parking'] < 0, 'parking'] = 0
         df.loc[df['parking'] > 4, 'parking'] = df['parking'].mode()[0]
```

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

Identified Inconsistencies

1. **Unrealistic Values**
   - *Bedrooms* contained an outlier value of **400**, which is not realistic for residential homes.
   - Some entries in *parking* and *bathrooms* exceeded normal limits.
2. **Categorical Inconsistencies**
   - Values like **"Yes", "yes", " YES "** appeared in different cases and formats.
   - *furnishingstatus* had mixed labels such as:
     - "semi-furnished"
     - "Semi Furnished"
     - "semi furnished "
3. **Format and Case Issues**
   - Extra spaces, uppercase/lowercase mismatches, and inconsistent naming.
4. **Potential Mixed Units**
   - Extreme values in *area* suggested possible incorrect units, but after verification, only outliers existed.
5. **Incorrect Data Types**
   - Some numeric columns loaded as strings (due to CSV formatting).

How They Were Resolved

1. **Outlier Correction**
   - Bedrooms ≥ 20 were replaced with the median to maintain realistic ranges.

2. **Categorical Standardization**
    - All categorical values converted to lowercase and trimmed.
    - Furnishing status labels unified into:
        - “furnished”
        - “unfurnished”
        - “semi furnished”
3. **Format Cleaning**
    - Removed leading/trailing spaces with str.strip().
4. **Data Type Correction**
    - Numerical columns converted to proper numeric types using pd.to_numeric().
5. **Value Constraints Applied**
    - Parking <0 or >4 corrected using mode.

---

5. Detecting and Handling Outliers

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

Figure 5: Outlier Detection Using IQR & Z-Score

```
In [45]: import numpy as np
# Select ONLY numerical columns
num_cols = df.select_dtypes(include=['int64', 'float64']).columns
df_num = df[num_cols]
# ---- IQR METHOD ----
outlier_summary = {}
for col in num_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)]
    outlier_summary[col] = len(outliers)

# Show IQR-based outliers
print("=== IQR Outliers Count ===")
for col, count in outlier_summary.items():
    print(f"{col}: {count}")
# ---- Z-SCORE METHOD ----
from scipy import stats
zscore_summary = {}
for col in num_cols:
    z_scores = np.abs(stats.zscore(df[col].dropna()))
    z_outliers = len(df[col].dropna()[z_scores > 3])
    zscore_summary[col] = z_outliers
print("=== Z-score Outliers Count (|z| > 3) ===")
for col, count in zscore_summary.items():
    print(f"{col}: {count}")
```

```
=== IQR Outliers Count ===
price: 15
area: 11
bedrooms: 12
bathrooms: 1
stories: 41
parking: 12
=== Z-score Outliers Count (|z| > 3) ===
price: 6
area: 6
bedrooms: 2
bathrooms: 11
stories: 0
parking: 0
```

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

1. price

- IQR outliers: **15**
- Z-score outliers: **6**
- Typical right-skewed housing prices.

Decision: WINSORIZE (not remove)
Reason: Outliers represent legitimate expensive houses; removing them hurts model generality.

---

2. area

- 11 IQR outliers + 6 Z-score outliers
  But **area range (1650 – 16200)** is natural for plots/houses.

  Decision: KEEP
  Reason: Large areas are real domain variations, not errors.

---

3. bedrooms

- 12 IQR outliers, 2 Z-score outliers
- Since your max is **400**, we know some entries are *impossible*.

  Decision: REMOVE unrealistic values only (e.g., bedrooms > 20)
  Reason: Real houses do not have 50–400 bedrooms → clear data errors.

---

4. bathrooms

- IQR outliers: **1**
- Z-score outliers: **11**
- Bathroom range: **1 to 4** → all realistic.

  Decision: KEEP
  Reason: No values are unrealistic.
  The Z-score method is unfair here because bathrooms are low-range discrete integers → Z-score misfires.

---

5. stories

- IQR outliers: **41**
- Z-score outliers: **0**
- Range: **1 to 4** → all valid.

  Decision: KEEP
  Reason: IQR incorrectly flags many values because data is tightly clustered.
  But all values are realistic.

---

6. parking

- IQR outliers: **12**
- Z-score: **0**
- Range **0–3** → all valid.

Decision: KEEP
Reason: IQR falsely flags common values like 0 or 1 due to skewness.

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

Figure 6: Outlier Handling

```
In [46]: from scipy.stats.mstats import winsorize

#1. Winsorize price (keep domain shape but reduce influence of extreme values)
df['price'] = winsorize(df['price'], limits=[0.01, 0.01])

#2. Remove unrealistic bedroom values (e.g., > 20)
df = df[df['bedrooms'] <= 20]

#3. No changes for area, bathrooms, stories, parking
# All their outliers are domain-valid
print("Outlier handling completed successfully.")

Outlier handling completed successfully.
```

Outlier Detection and Handling

Method Used

I used both the **IQR Method** and the **Z-Score Method** to detect outliers. Results were compared and validated against domain knowledge of real estate.

---

price — Winsorized

- IQR detected **15** outliers and Z-score detected **6**, mostly extremely expensive houses.
- These values are legitimate high-end properties, not errors.
- Instead of deleting them, I applied **1% winsorization** to reduce their influence while preserving information.

---

area — Kept

- IQR flagged **11** outliers, Z-score flagged **6**, but all values are realistic (up to 16,200 sq ft).
- Real estate properties naturally vary a lot in size.
- Therefore, the outliers were **retained**.

---

bedrooms — Cleaned

- IQR flagged **12**, Z-score flagged **2** outliers.
- The dataset contained **unrealistic values (e.g., 400 bedrooms)** which are impossible.
- I removed rows where bedrooms > 20 since they are clearly data entry errors.

---

bathrooms — Kept

- IQR: **1** outlier
- Z-score: **11** outliers
- All bathroom values ranged from **1 to 4**, which is normal.
- Z-score falsely flags low-range discrete integer variables.
- I kept all values.

---

stories — Kept

- IQR flagged **41**, Z-score flagged **0**.
- All values (1–4) are realistic building heights.
- Outliers were kept.

---

parking — Kept

- IQR flagged **12**, Z-score flagged **0**.
- All values (0–3) are realistic for residential homes.
- No handling required.

---

6. Normalization and Scaling

    a. Identify which variables require scaling or normalization.

Variable	Scaling Needed?	Reason
price	YES	Very large values (1.7M – 13M). Avoid dominating ML models.
area	YES	Large numerical range (1,650 – 16,200).
bedrooms	NO	Small integer values (1–5 after cleaning). Scaling doesn't help.
bathrooms	NO	Small range (1–4).

stories	NO	Small range (1–4).
parking	NO	Small range (0–3).

Rule: Only scale variables whose magnitude will distort model training. Small discrete integer values do NOT need scaling.

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

✓ Use Standardization (Z-score scaling) for:

- **price** → right-skewed but continuous, and Z-score works well with winsorized distributions
- **area** → wide range but relatively normally distributed after removing inconsistencies

✓ Do NOT scale:

bedrooms, bathrooms, stories, parking
(scaling small discrete categorical-like numbers adds no real value)

```
In [47]: from sklearn.preprocessing import StandardScaler

         # Create scaler
         scaler = StandardScaler()

         # Columns to scale
         columns_to_scale = ["price", "area"]

         # Fit and transform
         df_scaled = df.copy()
         df_scaled[columns_to_scale] = scaler.fit_transform(df[columns_to_scale])

         # Show sample of scaled data
         df_scaled.head()
```

Out[47]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
0	3.353076	1.070539	4	2.0	3	yes	no	no	no	yes	2	yes	furnished
1	3.353076	1.789576	4	4.0	4	yes	no	no	no	yes	3	no	furnished
2	3.353076	2.256483	3	2.0	2	yes	no	yes	no	no	2	yes	semi furnished
3	3.353076	NaN	4	2.0	2	yes	no	yes	no	yes	3	yes	furnished
4	3.353076	NaN	4	1.0	2	yes	yes	yes	no	yes	2	no	furnished

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

Normalization and Scaling

1. Variables Selected for Scaling

After analyzing the ranges and statistical distributions, only **price** and **area** required scaling.
These features have very large numeric ranges and would dominate distance-based or gradient-based models.

- **price:** ranges from ~1.7M to 13M
- **area:** ranges from ~1,650 to 16,200

Variables such as **bedrooms, bathrooms, stories, parking** were not scaled because they have small integer ranges (0–5), and scaling them adds no modeling benefit.

---

2. Scaling Technique Used: Standardization (Z-score Scaling)

✓ Applied to:

- **price**
- **area**

✓ Formula:

$$z = \frac{x - \mu}{\sigma}$$

✓ Reason:

- Works well on continuous numerical variables
- Handles slightly skewed distributions
- Keeps the effect of each variable balanced
- More robust for ML models like Linear Regression, SVM, Logistic Regression, and Neural Networks

---

3. Why Min-Max Scaling Was NOT Used

Min-max scaling shrinks data to [0, 1], but:

- It is **very sensitive to outliers**
- It would distort price and area because they already had extreme values before winsorization
- It weakens interpretation in models like regression

---

4. Why Robust Scaling Was NOT Used

Robust scaling is useful only when:

- A variable is extremely skewed
- Contains heavy outliers

But since **price was already winsorized** and **area is acceptable**, standardization was more appropriate and preserves interpretability.

Variable	Decision	Reason
price	Standardization	Wide range; needed for ML stability
area	Standardization	Similar reason as price
bedrooms	Not scaled	Small integer values
bathrooms	Not scaled	Small category-like values
stories	Not scaled	Small range; scaling has no effect
parking	Not scaled	Discrete small range

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

7.1a – Variables Encoded:

- mainroad (yes/no)
- guestroom (yes/no)
- basement (yes/no)
- hotwaterheating (yes/no)
- airconditioning (yes/no)
- prefarea (yes/no)

These are **binary categorical variables**.

7.1b – Why Label Encoding is Appropriate

Label Encoding is suitable because:

- The variables have **only two categories (yes/no)**
- There is **no ordinal relationship** but binary fields are safely represented as 0/1.
- Label Encoding maintains a clean, compact form ideal for tree-based models.

This avoids unnecessary dimensionality that would result from One-Hot Encoding.

7.1c – Transformation results

In [50]:
```
#before label encoding
df.select_dtypes(include=['object']).head()
```

Out[50]:

	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

In [51]:
```
#after label encoding
from sklearn.preprocessing import LabelEncoder
binary_cols = ["mainroad", "guestroom", "basement", "hotwaterheating", "airconditioning", "prefarea"]
label_encoders = {}
df_label = df.copy()
for col in binary_cols:
    le = LabelEncoder()
    df_label[col] = le.fit_transform(df_label[col])
    label_encoders[col] = le
df_label[binary_cols].head()
```

Out[51]:

	mainroad	guestroom	basement	hotwaterheating	airconditioning	prefarea
0	1	0	0	0	1	1
1	1	0	0	0	1	0
2	1	0	1	0	0	1
3	1	0	1	0	1	1
4	1	1	1	0	1	0

- One-Hot Encoding

7.2a – Variable Encoded

Variable:

- furnishingstatus

Unique categories:

furnished

semi-furnished

unfurnished

7.2b – Why One-Hot Encoding is Appropriate

- furnishingstatus is **nominal** (no natural order).
- One-Hot Encoding avoids creating false ordinal relationships.
- Models interpret each furnishing category independently using dummy variables.

7.2c – Transformation Results

```
In [55]: #before encoding
         df['furnishingstatus'].head()
```

```
Out[55]: 0         furnished
         1         furnished
         2    semi furnished
         3         furnished
         4         furnished
         Name: furnishingstatus, dtype: object
```

```
In [56]: #after encoding
         df_onehot = pd.get_dummies(df, columns=["furnishingstatus"], drop_first=False)
         df_onehot.head()
```

Out[56]:

ainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus_furnished	furnishingstatus_semi furnished	furnishingstatus_unfurnished
yes	no	no	no	yes	2	yes	True	False	False
yes	no	no	no	yes	3	no	True	False	False
yes	no	yes	no	no	2	yes	False	True	False
yes	no	yes	no	yes	3	yes	True	False	False
yes	yes	yes	no	yes	2	no	True	False	False

- Binary Encoding

7.3 BINARY ENCODING

7.3a – Variable Encoded

We apply binary encoding to:

- furnishingstatus

  Why not apply to yes/no variables? → Because binary encoding works best for **high-cardinality categorical features**, but here we use it to satisfy assignment requirements.

7.3b – Why Binary Encoding is Appropriate

- More compact than OHE (fewer columns).
- Useful for categories >2 but not extremely high-cardinality.
- Demonstrates understanding of alternative encoding strategies.

7.3c – Transformation Results

Example category-to-integer mapping:
furnished → 0

semi furnished → 1

unfurnished → 2

- Ordinal Encoding

7.4 ORDINAL ENCODING

7.4a – Variable Encoded

We apply ordinal encoding to:

- furnishingstatus *(based on quality level)*

We define a **logical order**:

1. **unfurnished** – lowest
2. **semi furnished** – medium
3. **furnished** – highest

7.4b – Why Ordinal Encoding is Appropriate

- Furnishing level has a **natural hierarchy of quality and value**, which influences house price.
- Ordinal encoding allows models to learn increasing value with improved furnishing.

This is more meaningful than random numeric labels.

7.4c – Transformation Results

- 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.

Submission Documentation and Justification

For every preprocessing step:

a. Show your code steps in the jupyter notebook file uploaded.

b. Show the before-and-after outputs of the data for each activity.

c. Provide a clear explanation and justification of why each technique was used using markdown.

d. Save and upload your cleaned dataset and ensure it is clean, consistent, encoded, and ready for modeling.