King Saud University

College of Computer and Information Sciences

Information Technology department

IT 326: Data Mining

Name	ID
Shouq Altamimi	**4442000920**
Haifa Alsaif	**443202006**
Majd Alruways	**444200722**
Norah Alfaheed	**444200779**

1. Problem

The housing dataset poses the challenge of predicting housing prices based on factors such as area, number of bedrooms, proximity to main roads, and more. The dataset is relatively small, but it presents complexity due to the strong multicollinearity among features, where some variables are highly correlated. This correlation can lead to issues like overfitting and redundancy in predictive models. The goal of this project is to overcome these challenges by applying appropriate data preprocessing and machine learning techniques to build a model that can predict housing prices effectively.

2. Data mining task

In this project, two main data mining tasks were applied: **classification** and **clustering**. For **classification**, we used a decision tree classifier to predict housing price categories (Low, Medium, High) by discretizing the continuous price variable. We utilized feature selection based on mutual information to reduce the dimensionality and identify the most important features. The model was trained and evaluated using different training and test splits, and we tested multiple decision tree criteria (Gini and entropy) to compare their performance. For **clustering**, we applied K-means and hierarchical clustering techniques to group houses based on their features like area, bedrooms, and parking. After scaling the features, we evaluated the optimal number of clusters using the elbow method and silhouette scores. The results of both tasks provide insights into the dataset's structure and help build predictive models for housing prices.

3. Data

Data source : https://www.kaggle.com/datasets/yasserh/housing-prices-dataset

Number of objects: 545 objects
Number of attributes: 13 attributes

- ## Data types

Attribute	Data type	Possible values
Price	Numeric (ratio)	1750000 - 13300000
Area	Numeric (ratio)	1650 – 16200
Bedrooms	Numeric (ratio)	1-6
Bathrooms	Numeric (ratio)	1-4
Stories	Numeric (ratio)	1-4
Mainroad	Binary (asymmetric)	Yes (1) , No (0)
Guestroom	Binary (asymmetric)	Yes (1) , No (0)
Basement	Binary (asymmetric)	Yes (1) , No (0)
Hotwaterheating	Binary (asymmetric)	Yes (1) , No (0)
Airconditioning	Binary (asymmetric)	Yes (1) , No (0)
Parking	Numeric (ratio)	0-3
Prefarea	Binary (asymmetric)	Yes (1) , No (0)
Furnishingstatus	Ordinal	Furnished(1) Semifurnished (2) unfurnished(3)

```
# Column names and types
Housing.dtypes
```

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

- ## Missing values :

```python
#check for missing values in the entire dataset
missing_values = Housing.isnull().sum()

#display the result
print("Missing values in each column:")
print(missing_values)

#total number of missing value
print("\nTotal number of missing values in the dataset:",missing_values.sum())

#Setting up the figure size
plt.figure(figsize=(10, 6))

#Creating the heatmap for missing values, using blue for non-missing values and red for missing values.
sns.heatmap(Housing.isnull(), cmap='coolwarm', cbar=False)

#Adding a title to the heatmap
plt.title("Missing Values Heatmap")

#Displaying the plot
plt.show()

#Since our dataset doesn't have any missing values, we don't need to handle missing data or drop any rows or columns.
```

```
Missing values in each column:
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
Class_Label         0
dtype: int64

Total number of missing values in the dataset: 0
```

- ## Statistical Measures (five number summary):

[51]:
```python
#desciptive stats
Housing.describe()
```

[51]:

	price	area	bedrooms	bathrooms	stories	parking
count	5.450000e+02	545.000000	545.000000	545.000000	545.000000	545.000000
mean	4.766729e+06	5150.541284	2.965138	1.286239	1.805505	0.693578
std	1.870440e+06	2170.141023	0.738064	0.502470	0.867492	0.861586
min	1.750000e+06	1650.000000	1.000000	1.000000	1.000000	0.000000
25%	3.430000e+06	3600.000000	2.000000	1.000000	1.000000	0.000000
50%	4.340000e+06	4600.000000	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	6.000000	4.000000	4.000000	3.000000

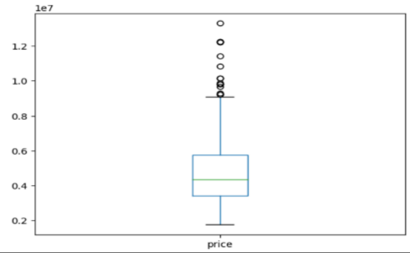
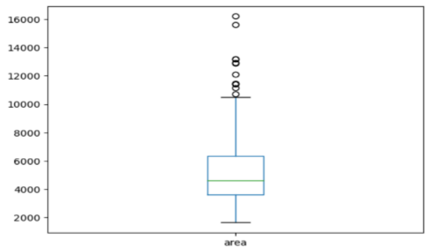
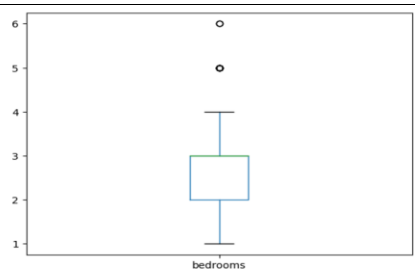
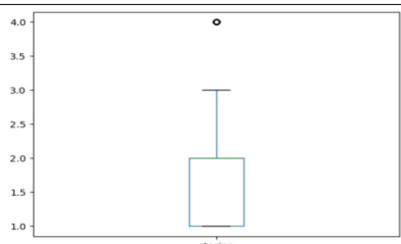
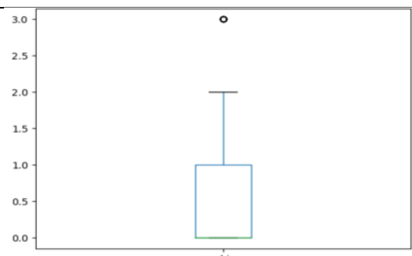
- ## Outliers:

Attributes	Number of outliers
Price	**15**
Area	**15**
Bedrooms	**12**
Bathrooms	**1**
Stories	**41**
Parking	**12**

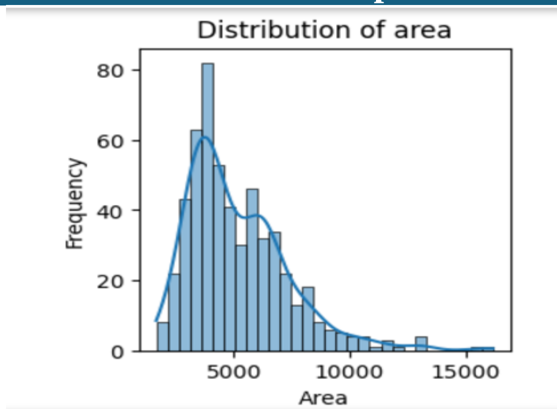
```python
# Create an empty list to store the output indices from multiple columns and drop the outliers by looping
index_list = []
outlier_counts = {}  # Dictionary to store the count of outliers for each feature

for feature in ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']:
    outliers = find_outliers(Housing, feature)  # Call your function to find outliers
    outlier_counts[feature] = len(outliers)  # Store the number of outliers for each feature
    index_list += outliers.tolist()  # Convert Index object to list and accumulate

# Drop all the outliers from the DataFrame
HousingC = Housing.drop(index_list, errors='ignore')

# Print the number of outliers for each column
for feature, count in outlier_counts.items():
    print(f"Number of outliers in '{feature}': {count}")
```

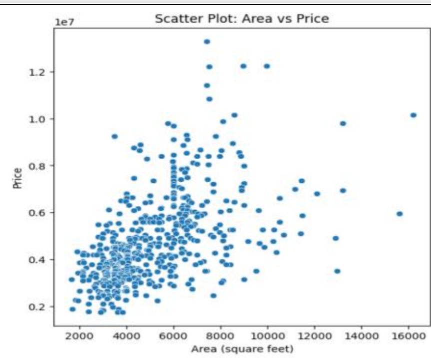
```
Number of outliers in 'price': 15
Number of outliers in 'area': 12
Number of outliers in 'bedrooms': 12
Number of outliers in 'bathrooms': 1
Number of outliers in 'stories': 41
Number of outliers in 'parking': 12
```

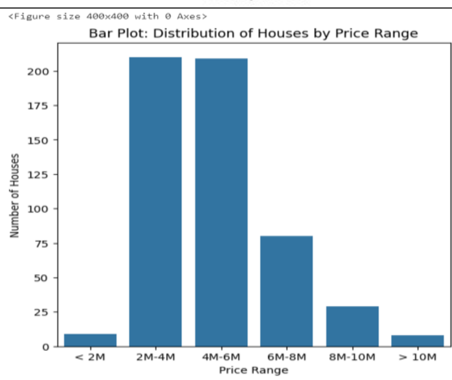
- **Box plots :**

Attribute Graph	Description
	The price box plot illustrate multiple outliers, indicating properties with significantly higher prices than the norm.
	The area box plot illustrate several outliers in the area distribution. These represent properties with significantly larger areas than the majority
	The bedroom box plot illustrate that there are noticeable outliers, representing houses with a much higher number of bedrooms than the majority.
	The bathrooms box plot illustrate that there is one prominent outlier above 3, representing houses with an unusually high number of bathrooms.
	The stories box plot illustrate some outliers above 3, representing houses with an unusually high number of stories compared to the majority.
	The parking box plot illustrate some outliers with a value of 3, which is significantly higher than the rest of the data.

- **Plotting Methods:**

<table>
<tr><th>Attribute Graph</th><th>Description</th></tr>
<tr><td>


</td><td>The histogram represents the frequency of house sizes in the dataset. Most values fall within the normal range of 3000 to 5000 square feet. However, it also highlights the presence of several outliers, with very large and very small houses in the dataset.</td></tr>
<tr><td>


</td><td>The scatter plot reveals a strong positive correlation between house area and price, indicating that larger houses generally cost more. However, a few outliers show smaller houses with higher prices, likely influenced by factors such as location, luxury features, or unique attributes.</td></tr>
<tr><td>


</td><td>The bar graph represents the distribution of houses by price range. Each bar indicates a specific price range (e.g., less than 2M, 2M-4M, etc.) and the number of houses within it. The tallest bar highlights the most common price range, such as 4M-6M, suggesting that the majority of houses fall within this category.</td></tr>
</table>

- ## Box plots code :

```
[2]: #define a function called "plot_boxplot" to show the boxplots for the data

def plot_boxplot(df , ft):
    df.boxplot(column=[ft])
    plt.grid(False)
    plt.show()
```

```
[7]: #show the boxplot for each numeric column

plot_boxplot(Housing , "price")
plot_boxplot(Housing , "area")
plot_boxplot(Housing , "bedrooms")
plot_boxplot(Housing , "bathrooms")
plot_boxplot(Housing , "stories")
plot_boxplot(Housing , "parking")
```

- ## Plotting Methods code (histogram , scatter plot , bar graph respectivel):

```
# Create a Histogram
# List of numerical columns
columns = ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']

for col in columns:
    plt.figure(figsize=(3, 3))
    sns.histplot(Housing[col], bins=30, kde=True)
    #adds a title to the histogram
    plt.title(f'Distribution of {col}')
    plt.xlabel(col.capitalize())
    plt.ylabel('Frequency')
    # display the histogram
    plt.show()
```

```
# Create a scatter plot of area vs price
plt.figure(figsize=(4, 4))
sns.scatterplot(x=Housing['area'], y=Housing['price'])
plt.title('Scatter Plot: Area vs Price')
plt.xlabel('Area (square feet)')
plt.ylabel('Price')
plt.show()
```

```
#Bar plots

# Define price ranges
price_bins = [0, 2000000, 4000000, 6000000, 8000000, 10000000, 14000000]

# Assign names to each bin
price_labels = ['< 2M', '2M-4M', '4M-6M', '6M-8M', '8M-10M', '> 10M']


#Bar Plot: Distribution of houses in different price ranges (initializes a new figure for the plot)
plt.figure(figsize=(6, 6))

#e creates the bar plot using Seaborn's countplot() function
sns.countplot(x=pd.cut(Housing['price'], bins=price_bins, labels=price_labels))

#Creating the bar plot
plt.title("Bar Plot: Distribution of Houses by Price Range")

#Labeling the axes clearly to eliminate any ambiguity in understanding the data represented in the plot.
plt.xlabel("Price Range")
plt.ylabel("Number of Houses")

#display the plot
plt.show()
```

4. Data preprocessing:

• Checking for missing values:

In [24]:

```
#check for missing values in the entire dataset
missing_values = Housing.isnull().sum()

#display the result
print("Missing values in each column:")
print(missing_values)

#total number of missing value
print("\nTotal number of missing values in the dataset:",missing_values.sum())

#Setting up the figure size
plt.figure(figsize=(10, 6))

#Creating the heatmap for missing values, using blue for non-missing values and red for missing values.
sns.heatmap(Housing.isnull(), cmap='coolwarm', cbar=False)

#Adding a title to the heatmap
plt.title("Missing Values Heatmap")

#Displaying the plot
plt.show()

#Since our dataset doesn’t have any missing values, we don’t need to handle missing data or drop any rows or columns.
```

```
Missing values in each column:
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
Class_Label         0
dtype: int64

Total number of missing values in the dataset: 0
```

Description:

Null and missing values can negatively impact the accuracy and efficiency of the dataset and the insights extracted from it during data analysis or modeling. Therefore, we performed a thorough check to identify if there are any missing or null values in the dataset.Upon investigation, we found that the dataset does not contain any missing or null values. As a result, no further actions, such as row deletion or data imputation, were required, ensuring the dataset's completeness and reliability.

• Detecting and removing the outliers:

Detecting the outliers:

In [5]:

```
#define a function called "find_outliers" which returns a list for the outliers indexs

def find_outliers(df, ft):
    Q1 = df[ft].quantile(0.25)
    Q3 = df[ft].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    ls = df.index[(df[ft] < lower_bound) | (df[ft] > upper_bound)]
    return ls
```

In [31]:

```
#calling the list with the outliers index
index_list
```

Out[31]:

```
[0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10,
 11,
 12,
 13,
 14,
 7,
 10,
 56,
 64,
 66,
 69,
 125,
 129,
 186,
 191,
 211,
 403,
 7,
 28,
 34,
 89,
 112,
 143,
 152,
 271,
 340,
 356,
 395,
 536,
 1,
 1,
 6,
 9,
 17,
 26,
 30,
 31,
 35,
 37,
 38,
 39,
 41,
 42,
 43,
 44,
 46,
 47,
 50,
 51,
 52,
 53,
 57,
 58,
 59,
 71,
 72,
 73,
 83,
 92,
 94,
 102,
 105,
 124,
 131,
 135,
 140,
 145,
 160,
 220,
 226,
 247,
 1,
 3,
 47,
 93,
 225,
 247,
 299,
 304,
 323,
 331,
 401,
 472]
```

Removing the outliers:

In [6]:

```
# Create an empty list to store the output indices from multiple columns and drop the outliers by looping
index_list = []
for feature in ['price', 'area', 'bedrooms', 'bathrooms', 'stories', 'parking']:
    outliers = find_outliers(Housing, feature)
    index_list += outliers.tolist()  # Convert Index object to list and accumulate

# Drop all the outliers from the DataFrame
HousingC = Housing.drop(index_list, errors='ignore')
```

Description:

In this step, we identified and removed outliers from the dataset to improve its quality and ensure the accuracy of the analysis. We utilized the Interquartile Range (IQR) method to detect values outside the acceptable range (Q1 - 1.5×IQR and Q3 + 1.5×IQR). This process was applied to key columns such as price, area, number of bedrooms, and parking spaces. After detecting the outliers, they were removed from the dataset to minimize their negative impact on predictive models and enhance performance during analysis.

- ## Data transformation :
- ## Data Encoding:

1	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
2	13300000	7420	4	2	3	yes	no	no	no	yes	2	yes	furnished
3	12250000	8960	4	4	4	yes	no	no	no	yes	3	no	furnished
4	12250000	9960	3	2	2	yes	no	yes	no	no	2	yes	semi-furnished
5	12215000	7500	4	2	2	yes	no	yes	no	yes	3	yes	furnished
6	11410000	7420	4	1	2	yes	yes	yes	no	yes	2	no	furnished

Description:

Data Encoding is the process of converting categorical data into numerical values to make it compatible with machine learning models. Before encoding, the dataset included categorical columns like mainroad, guestroom, basement, hotwaterheating, airconditioning, prefarea, and furnishingstatus, which contained values such as "yes", "no", and other text labels. Using Label Encoding, these columns were transformed into numerical values (e.g., "yes" → 1, "no" → 0), ensuring the data is ready for analysis and modeling.

Data before Encoding:

```
Raw Samples:
      price  area  bedrooms  bathrooms  stories mainroad guestroom basement  \
0  13300000  7420         4          2        3      yes        no       no
1  12250000  8960         4          4        4      yes        no       no
2  12250000  9960         3          2        2      yes        no      yes
3  12215000  7500         4          2        2      yes        no      yes
4  11410000  7420         4          1        2      yes       yes      yes

  hotwaterheating airconditioning  parking prefarea furnishingstatus  \
0              no             yes        2      yes        furnished
1              no             yes        3       no        furnished
2              no              no        2      yes   semi-furnished
3              no             yes        3      yes        furnished
4              no             yes        2       no        furnished
```

Data after Encoding:

In [7]:

```python
#Encoding the dataset (AFTER cleaning)

from sklearn.preprocessing import LabelEncoder
from scipy import stats

le=LabelEncoder()
HousingC['mainroad'] =le.fit_transform(HousingC['mainroad'])
HousingC['guestroom'] =le.fit_transform(HousingC['guestroom'])
HousingC['basement'] =le.fit_transform(HousingC['basement'])
HousingC['hotwaterheating'] =le.fit_transform(HousingC['hotwaterheating'])
HousingC['airconditioning'] =le.fit_transform(HousingC['airconditioning'])
HousingC['prefarea'] =le.fit_transform(HousingC['prefarea'])
HousingC['furnishingstatus'] =le.fit_transform(HousingC['furnishingstatus'])

print(HousingC)
```

```
       price  area  bedrooms  bathrooms  stories  mainroad  guestroom  \
15   9100000  6000         4          1        2         1          0
16   9100000  6600         4          2        2         1          1
18   8890000  4600         3          2        2         1          1
19   8855000  6420         3          2        2         1          0
20   8750000  4320         3          1        2         1          0
..       ...   ...       ...        ...      ...       ...        ...
540  1820000  3000         2          1        1         1          0
541  1767150  2400         3          1        1         0          0
542  1750000  3620         2          1        1         1          0
543  1750000  2910         3          1        1         0          0
544  1750000  3850         3          1        2         1          0

     basement  hotwaterheating  airconditioning  parking  prefarea  \
15          1                0                0        2         0
16          1                0                1        1         1
18          0                0                1        2         0
19          0                0                1        1         1
20          1                1                0        2         0
..        ...              ...              ...      ...       ...
540         1                0                0        2         0
541         0                0                0        0         0
542         0                0                0        0         0
543         0                0                0        0         0
544         0                0                0        0         0

     furnishingstatus
15                  1
16                  2
18                  0
19                  1
20                  1
..                ...
540                 2
541                 1
542                 2
543                 0
544                 2

[463 rows x 13 columns]
```

- ## Normalization

1	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
2	13300000	7420	4	2	3	yes	no	no	no	yes	2	yes	furnished
3	12250000	8960	4	4	4	yes	no	no	no	yes	3	no	furnished
4	12250000	9960	3	2	2	yes	no	yes	no	no	2	yes	semi-furnished
5	12215000	7500	4	2	2	yes	no	yes	no	yes	3	yes	furnished
6	11410000	7420	4	1	2	yes	yes	yes	no	yes	2	no	furnished

Description:

Normalization is the process of scaling numerical values to a common range (typically between 0 and 1) to ensure consistency across numerical features like area. Min-Max Scaling was applied to reduce value disparities and ensure all features contribute equally to analysis and modeling. This step made the dataset more uniform and ready for predictive models.

Data before Normalization:

```
Raw Samples:
      price  area  bedrooms  bathrooms  stories mainroad guestroom basement  \
0  13300000  7420         4          2        3      yes        no       no
1  12250000  8960         4          4        4      yes        no       no
2  12250000  9960         3          2        2      yes        no      yes
3  12215000  7500         4          2        2      yes        no      yes
4  11410000  7420         4          1        2      yes       yes      yes

  hotwaterheating airconditioning  parking prefarea furnishingstatus  \
0              no             yes        2      yes        furnished
1              no             yes        3       no        furnished
2              no              no        2      yes   semi-furnished
3              no             yes        3      yes        furnished
4              no             yes        2       no        furnished
```

Data after Normalization:

```python
# normlize area column

from sklearn.preprocessing import MinMaxScaler
import pandas as pd

#Extract columns to normlize
columns_to_normalize = ['area']
data_to_normalize = HousingC[columns_to_normalize ]

#Min-Max scaling for selected columns
minmax_scaler = MinMaxScaler()
normalized_data_minmax = minmax_scaler.fit_transform(data_to_normalize)

#Replace the normalized values in the original DaataFrame
HousingC[columns_to_normalize] = normalized_data_minmax

print("Min-Max scaled data(only second column):")
print(HousingC)
```

```
Min-Max scaled data(only second column):
       price      area  bedrooms  bathrooms  stories  mainroad  guestroom  \
15   9100000  0.491525         4          1        2         1          0
16   9100000  0.559322         4          2        2         1          1
18   8890000  0.333333         3          2        2         1          1
19   8855000  0.538983         3          2        2         1          0
20   8750000  0.301695         3          1        2         1          0
..       ...       ...       ...        ...      ...       ...        ...
540  1820000  0.152542         2          1        1         1          0
541  1767150  0.084746         3          1        1         0          0
542  1750000  0.222599         2          1        1         1          0
543  1750000  0.142373         3          1        1         0          0
544  1750000  0.248588         3          1        2         1          0

     basement  hotwaterheating  airconditioning  parking  prefarea  \
15          1                0                0        2         0
16          1                0                1        1         1
18          0                0                1        2         0
19          0                0                1        1         1
20          1                1                0        2         0
..        ...              ...              ...      ...       ...
540         1                0                0        2         0
541         0                0                0        0         0
542         0                0                0        0         0
543         0                0                0        0         0
544         0                0                0        0         0

     furnishingstatus Class_Label
15                  1   Expensive
16                  2   Expensive
18                  0   Expensive
19                  1   Expensive
20                  1   Expensive
..                ...         ...
540                 2   Expensive
541                 1   Expensive
542                 2   Expensive
543                 0   Expensive
544                 2   Expensive
```

- ## Discretization:

1	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
2	13300000	7420	4	2	3	yes	no	no	no	yes	2	yes	furnished
3	12250000	8960	4	4	4	yes	no	no	no	yes	3	no	furnished
4	12250000	9960	3	2	2	yes	no	yes	no	no	2	yes	semi-furnished
5	12215000	7500	4	2	2	yes	no	yes	no	yes	3	yes	furnished

Description:

Discretization is the process of converting continuous numerical data into discrete categories to simplify analysis and improve interpretability. In this case, the price column was discretized into two categories: "Low" and "High," based on predefined bins. This step allows for grouping data into meaningful intervals, making it easier to analyze patterns and trends in the dataset. The resulting column, discretized_price, provides a categorical representation of the continuous price values.

Data before and after Discretization:

```python
import pandas as pd

# Discretization for a specific column('price')
column_to_discretize = 'price'
bins = [0, 5000000, float('inf')]
labels = ['Low','High']

#Perform discretization using the cut function
HousingC['discretized_' + column_to_discretize] = pd.cut(HousingC['price'], bins=bins, labels=labels)


print(HousingC[['price', 'discretized_price']])
```

```
       price discretized_price
15   9100000              High
16   9100000              High
18   8890000              High
19   8855000              High
20   8750000              High
..       ...               ...
540  1820000               Low
541  1767150               Low
542  1750000               Low
543  1750000               Low
544  1750000               Low
```

Row Data:

```
In [19]: Raw_samples = Housing.head()
         # Display the raw samples
         print("Raw Samples:\n", Raw_samples)
```

```
Raw Samples:
       price  area  bedrooms  bathrooms  stories mainroad guestroom basement  \
0  13300000  7420         4          2        3      yes        no       no
1  12250000  8960         4          4        4      yes        no       no
2  12250000  9960         3          2        2      yes        no      yes
3  12215000  7500         4          2        2      yes        no      yes
4  11410000  7420         4          1        2      yes       yes      yes

  hotwaterheating airconditioning  parking prefarea furnishingstatus  \
0              no             yes        2      yes        furnished
1              no             yes        3       no        furnished
2              no              no        2      yes   semi-furnished
3              no             yes        3      yes        furnished
4              no             yes        2       no        furnished

  Class_Label
0   Expensive
1   Expensive
2   Expensive
3   Expensive
4   Expensive
```

The first few rows of the dataset give us an idea of the range and structure of the data. we can see that houses vary widely in their attributes, such as the number of bedrooms, bathrooms, and prices. This helps confirm that the dataset is rich with information for further analysis

Data after preprocessing :

```
[33]: Data_after_preprocessing = HousingC.head()
      print("Datat after preprocessing\n", Data_after_preprocessing)
```

```
Datat after preprocessing
       price      area  bedrooms  bathrooms  stories  mainroad  guestroom  \
15  9100000  0.491525         4          1        2         1          0
16  9100000  0.559322         4          2        2         1          1
18  8890000  0.333333         3          2        2         1          1
19  8855000  0.538983         3          2        2         1          0
20  8750000  0.301695         3          1        2         1          0

    basement  hotwaterheating  airconditioning  parking  prefarea  \
15         1                0                0        2         0
16         1                0                1        1         1
18         0                0                1        2         0
19         0                0                1        1         1
20         1                1                0        2         0

    furnishingstatus Class_Label discretized_price
15                 1   Expensive              High
16                 2   Expensive              High
18                 0   Expensive              High
19                 1   Expensive              High
20                 1   Expensive              High
```

5-Data Mining Techniques for the Dataset

1. Classification: Decision Tree

Why Use Decision Tree?

- Decision Trees are straightforward and easy to interpret, making them a great choice for classifying data.
- They help us understand which attributes are most important for predicting the target variable.
- We'll use two methods to measure attribute importance:
    - **Information Gain (Entropy)**: Tells us how much uncertainty is reduced when we split the data.
    - **Gini Index**: Measures impurity in the data, encouraging binary splits.

How It's Done:

- **Python Tools**: We'll use the Python Package: **sklearn.tree,** and the Method: **DecisionTreeClassifier**
- **Steps**:
    - Split the dataset into training and testing sets with three partition sizes:
        - 90% training, 10% testing

- 80% training, 20% testing
- 70% training, 30% testing
- Train two Decision Tree models:
  - One using Gini Index by using DecisionTreeClassifier(criterion="gini")

One using Information Gain by using DecisionTreeClassifier(criterion="entropy")

- Evaluate the models by:
  - Measuring accuracy.
  - Analyzing confusion matrices to see how well the model classifies the data.
- We use **plot_tree** to visualize the Decision Trees to understand how they make predictions and identify key patterns in the data.

2. Clustering: K-Means

Why Use K-Means?

- K-Means is a widely used and effective clustering method for grouping similar data points.
- It is computationally efficient and reveals natural patterns in data without predefined labels.

How It's Done:

- **Python Tools**: We'll use the Python Package: **sklearn.cluster,** and the Method: **KMeans**
- **Steps**:
  - **Preprocessing**: Scale numerical features using **StandardScaler** to normalize them, ensuring all features contribute equally.
- **Cluster Formation**:
  - Use the K-Means algorithm with three different values of k=2,4,7.
  - For each k, the clusters are formed, and centroids are computed.

2. **Evaluate** the quality of the clusters using:
   - **Silhouette Coefficient**: Compute silhouette scores using **silhouette_score** to measure the quality of clustering.
   - **Elbow Method**: Plot the total within-cluster sum of squares (WCSS) for various k*k* values to find the optimal k*k*.
3. Use **scatter plots** to visualize the clusters and evaluate their quality to make sense of the groupings.

6-Evaluation and Comparison

Size of training and testing set :

- 90% training, 10% testing
- 80% training, 20% testing
- 70% training, 30% testing

First: Classification:

The codes:

Discission tree:

```
[28] from sklearn.tree import plot_tree

     # Visualize the decision tree for each configuration
     for partition_name, (train_size, test_size) in partitions.items():
         X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=train_size, test_size=test_size, random_state=42

         for criterion in ["gini", "entropy"]:
             # Initialize the Decision Tree classifier with a maximum depth to reduce complexity
             clf = DecisionTreeClassifier(criterion=criterion, max_depth=5, random_state=42)
             clf.fit(X_train, y_train)

             # Visualize the tree with adjusted sizes
             plt.figure(figsize=(8, 6))  # Compact size
             plot_tree(
                 clf,
                 filled=True,
                 feature_names=X.columns,
                 class_names=["Low", "High"],  # Update class names as needed
                 fontsize=8  # Reduced font size for compact display
             )
             plt.title(f"Decision Tree ({criterion.capitalize()} - {partition_name} Split)", fontsize=10)
             plt.show()
```

Confusion matrix:

```
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Plot confusion matrices
for partition_name, (train_size, test_size) in partitions.items():
    # Ensure consistent splits
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=train_size, test_size=test_size, random_state=42)

    for criterion in ["gini", "entropy"]:
        clf = DecisionTreeClassifier(criterion=criterion, max_depth=7, random_state=42)  # Ensure consistent parameters
        clf.fit(X_train, y_train)

        y_pred = clf.predict(X_test)
        conf_matrix = confusion_matrix(y_test, y_pred)

        # Visualize confusion matrix
        plt.figure(figsize=(8, 6))
        sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=["Low", "High"], yticklabels=["Low", "High"])
        plt.title(f"Confusion Matrix ({criterion.capitalize()} - {partition_name} Split)")
        plt.xlabel("Predicted")
        plt.ylabel("Actual")
        plt.show()
```

Precision, Sensitivity, and Specificity:

```
from sklearn.metrics import confusion_matrix, precision_score, recall_score

# Function to calculate specificity
def specificity(cm):
    tn, fp, fn, tp = cm.ravel()  # Unpack confusion matrix
    return tn / (tn + fp)

# Evaluate the metrics for each configuration
for partition_name, (train_size, test_size) in partitions.items():
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=train_size, test_size=test_size, random_state=42)

    for criterion in ["gini", "entropy"]:
        clf = DecisionTreeClassifier(criterion=criterion, max_depth=7, random_state=42)  # Ensure consistent parameters
        clf.fit(X_train, y_train)

        y_pred = clf.predict(X_test)
        cm = confusion_matrix(y_test, y_pred)

        # Calculate metrics
        precision = precision_score(y_test, y_pred)
        sensitivity = recall_score(y_test, y_pred)  # Recall is equivalent to sensitivity
        spec = specificity(cm)

        # Display the results
        print(f"Partition: {partition_name}, Criterion: {criterion}")
        print(f"Precision: {precision:.2f}")
        print(f"Sensitivity (Recall): {sensitivity:.2f}")
        print(f"Specificity: {spec:.2f}\n")
```

Accuracy:

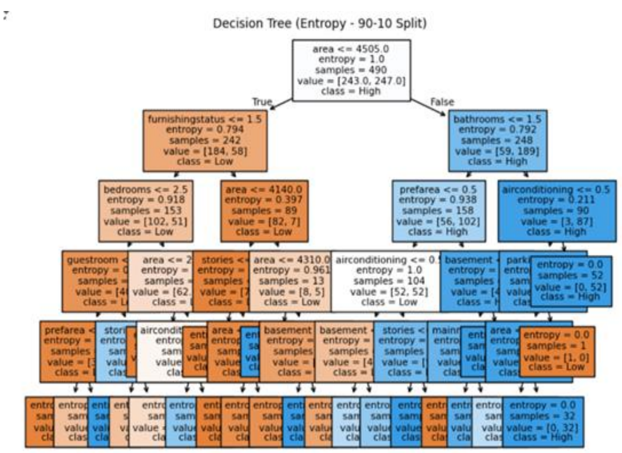
```python
# Convert results to a DataFrame
results_df = pd.DataFrame(results)
print("Classification Results:")
print(results_df)
```

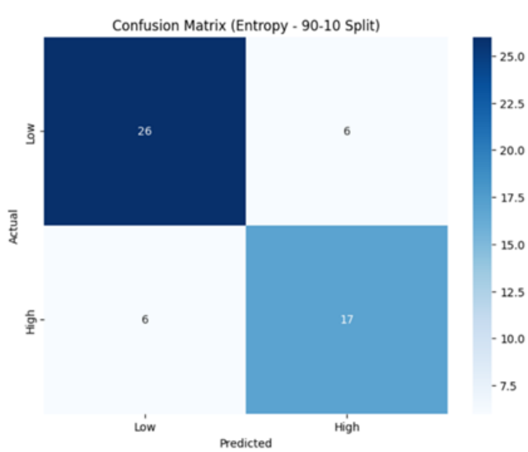
Show hidden output

```python
import matplotlib.pyplot as plt

# Plot accuracy comparison
plt.figure(figsize=(10, 6))
for criterion in ["gini", "entropy"]:
    subset = results_df[results_df['Criterion'] == criterion]
    plt.plot(subset['Partition'], subset['Accuracy'], marker='o', label=f"Criterion: {criterion.capitalize()}")

plt.title("Decision Tree Accuracy Comparison")
plt.xlabel("Partition")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.show()
```

1- 90% training, 10% testing

a) Information Gain (IG)





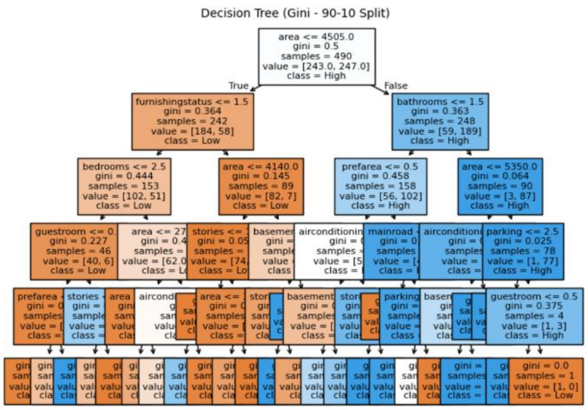
Accuracy:

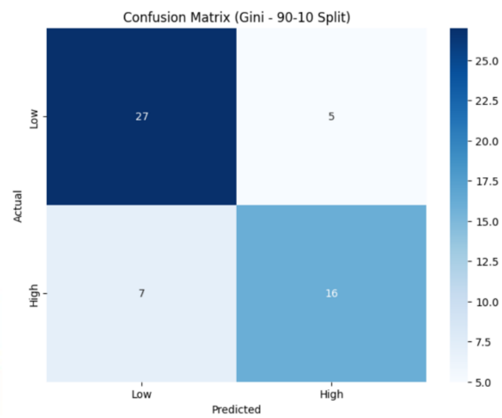
Precision, Sensitivity, and Specificity:

```
90-10   entropy   0.763636
```

```
Partition: 90-10, Criterion: entropy
Precision: 0.74
Sensitivity (Recall): 0.74
Specificity: 0.81
```

b) Gini Index





Accuracy:

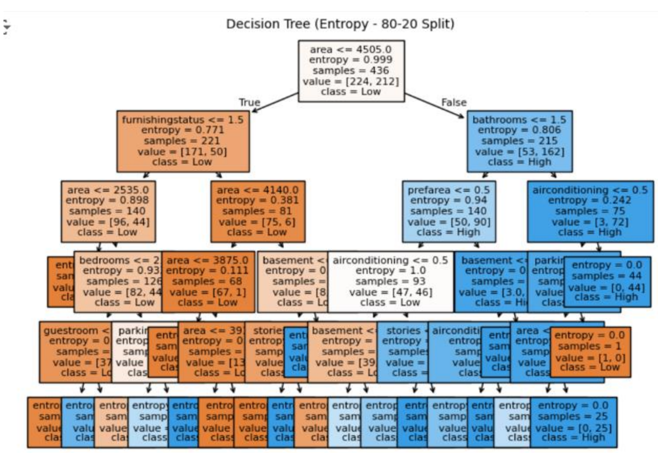
```
90-10     gini  0.800000
```

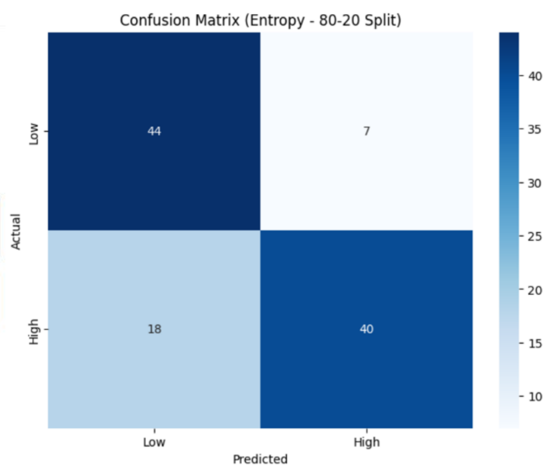
Precision, Sensitivity, and Specificity:

```
Partition: 90-10, Criterion: gini
Precision: 0.76
Sensitivity (Recall): 0.70
Specificity: 0.84
```

2- 80% training, 20% testing

a) Information Gain (IG)





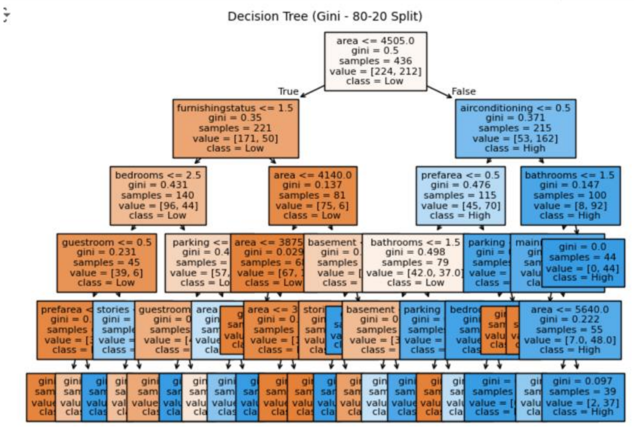
Accuracy:

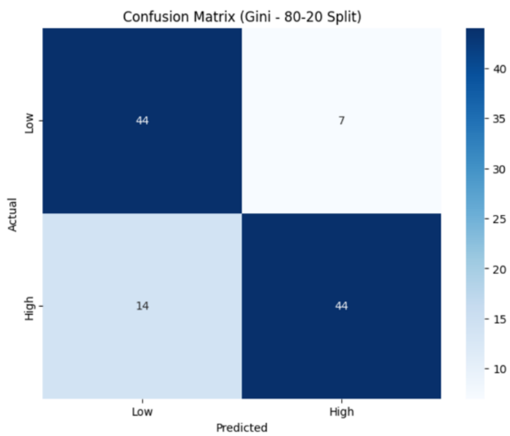
Precision, Sensitivity, and Specificity:

```
80-20    entropy   0.779817
```

```
Partition: 80-20, Criterion: entropy
Precision: 0.85
Sensitivity (Recall): 0.69
Specificity: 0.86
```

b) Gini Index





Accuracy:

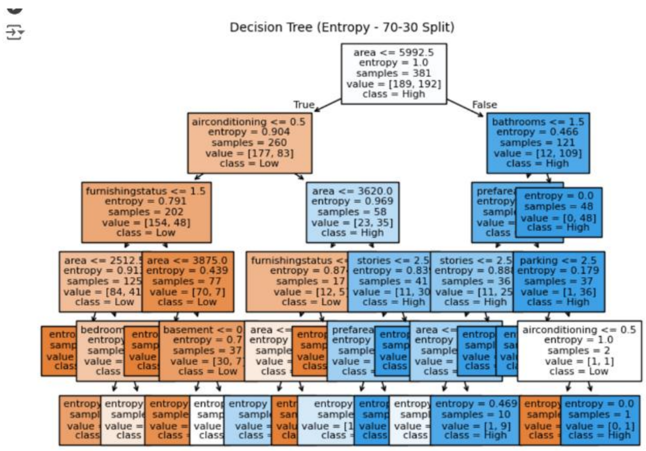
```
80-20       gini   0.798165
```

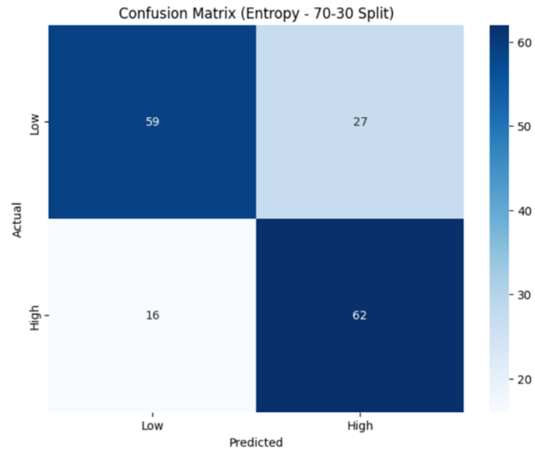
Precision, Sensitivity, and Specificity:

```
Partition: 80-20, Criterion: gini
Precision: 0.86
Sensitivity (Recall): 0.76
Specificity: 0.86
```

3- 70% training, 30% testing

a) Information Gain (IG)





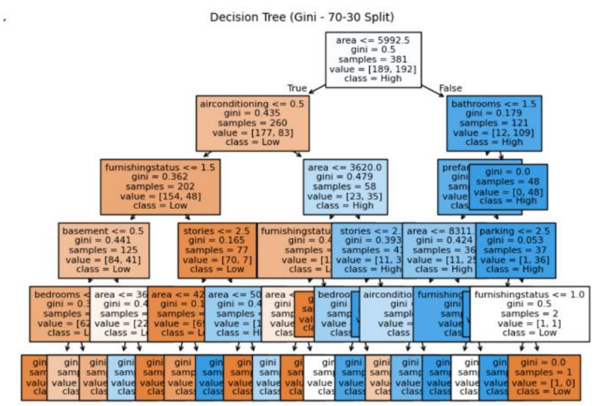
Accuracy:

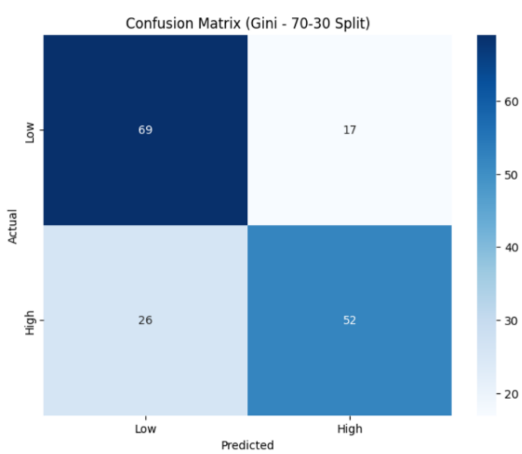
70-30   entropy   0.719512

Precision, Sensitivity, and Specificity:

```
Partition: 70-30, Criterion: entropy
Precision: 0.70
Sensitivity (Recall): 0.79
Specificity: 0.69
```

b) Gini Index





Accuracy:

Precision, Sensitivity, and Specificity:

```
70-30        gini   0.743902
```

```
Partition: 70-30, Criterion: gini
Precision: 0.75
Sensitivity (Recall): 0.67
Specificity: 0.80
```

Accuracy Comparision:

	90 %t raining set 10% testing set:		80 %t raining set 20% testing set:		70 %t raining set 30% testing set:	
	IG	**Gini Index**	**IG**	**Gini Index**	**IG**	**Gini Index**
Accuracy	**0. 764**	**0.800**	**0.780**	**0.798**	**0.720**	**0.744**

confusion matrix Comparision

Partition	Criterion	Precision	Sensitivity (Recall)	Specificity	Best Algorithm for Partition
90-10 Split	Gini	0.76	0.70	0.84	**Gini** (higher specificity and precision).
	Entropy	0.74	0.74	0.81	
80-20 Split	Gini	0.86	0.76	0.86	**Gini** (better balance across metrics).
	Entropy	0.85	0.69	0.86	
70-30 Split	Gini	0.75	0.67	0.80	

	Entropy	0.70	0.79	0.69	**Gini** (higher specificity and precision).

Best Partition Overall: 90-10 Split

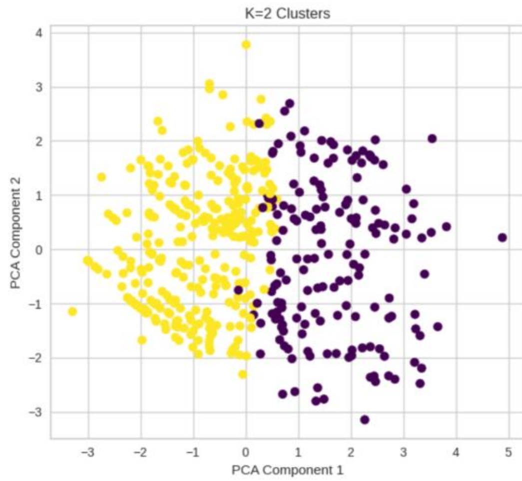
Reason: It balances **precision, sensitivity, and specificity** the best among the three splits, with Gini performing particularly well.

- ## Clusteing :

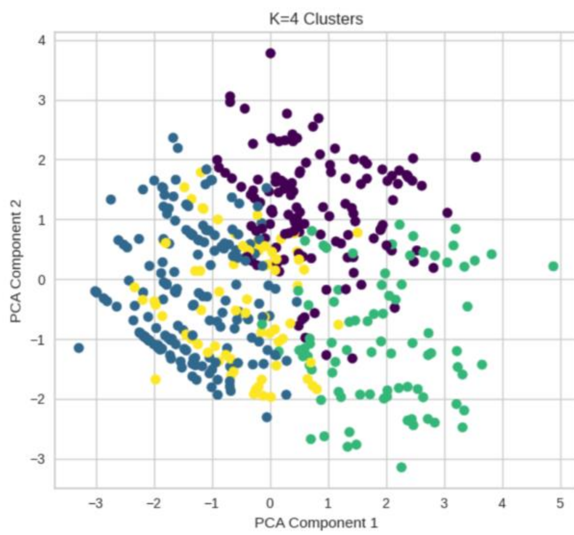
In [201…

```
import pandas as pd
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.metrics import silhouette_score
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.preprocessing import LabelEncoder, StandardScaler
import matplotlib.pyplot as plt

# Load the dataset
data = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/HousingProcessed.csv')  # Adjust path if necessary

# Convert categorical features to numeric using Label Encoding or One-Hot Encoding
for column in data.select_dtypes(include=['object']).columns:
    le = LabelEncoder()
    data[column] = le.fit_transform(data[column])

# Extract features for clustering
X = data.values  # or select specific columns if needed, e.g., `data[['Feature1', 'Feature2']].values`

# Standardize the features for better clustering performance
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Function to evaluate clustering
def evaluate_clustering(X, labels):
    silhouette_avg = silhouette_score(X, labels)
    print(f"Silhouette Score: {silhouette_avg:.3f}")
    return silhouette_avg

# Step 1: Hierarchical Clustering
print("Performing Hierarchical Clustering...")
hc = AgglomerativeClustering(n_clusters=4, linkage='ward')  # Example with 4 clusters
hc_labels = hc.fit_predict(X)

# Plot the dendrogram
linked = linkage(X, method='ward')
plt.figure(figsize=(10, 7))
dendrogram(linked, truncate_mode='level', p=5)
plt.title("Dendrogram for Hierarchical Clustering")
plt.xlabel("Data Points")
plt.ylabel("Euclidean Distance")
plt.show()

# Evaluate Hierarchical Clustering
print("Evaluating Hierarchical Clustering:")
evaluate_clustering(X, hc_labels)

# Step 2: K-means Clustering
k_values = [2, 4, 7]
kmeans_silhouettes = []

for k in k_values:
    print(f"\nPerforming K-means Clustering with K={k}...")
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans_labels = kmeans.fit_predict(X)

    # Evaluate K-means
    print(f"Evaluating K-means with K={k}:")
    silhouette_avg = evaluate_clustering(X, kmeans_labels)
    kmeans_silhouettes.append(silhouette_avg)
```

- **Clusteing with 3 different number of clusters:**
  K=2:

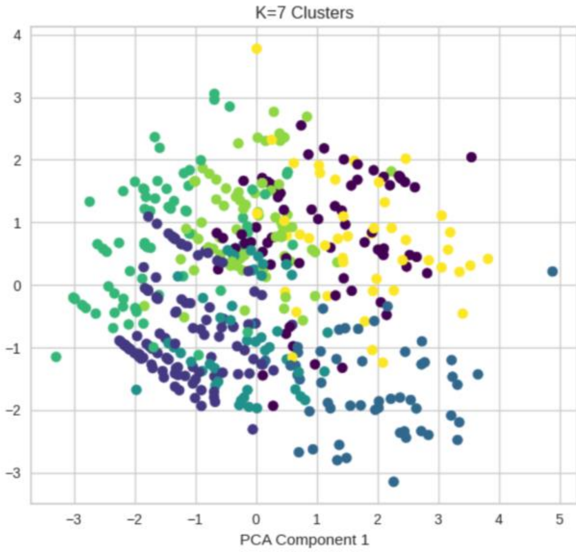


- **Clusteing with 3 different number of clusters: K=5:**

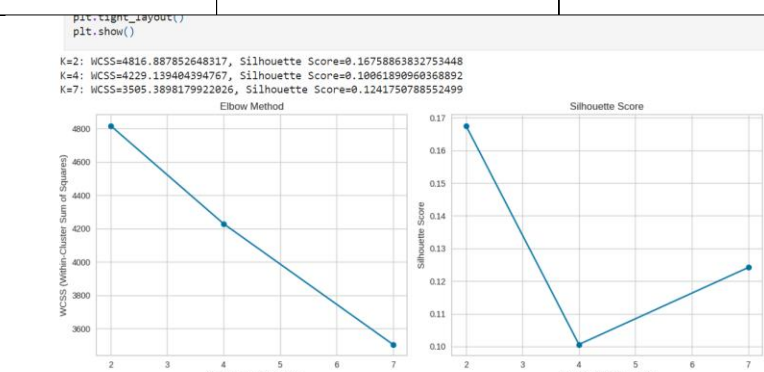


- **Clusteing with 3 different number of clusters:**

  K=7:



- **Clusteing with 3 different number of clusters:**
  K=2, K=5, K=8:

No.of Cluster	**K=2**	**K=5**	**K=8**
Average Silhouette width	**0.1676**	**0.1006**	**0.1242**
total within-cluster sum of square	**4816.89**	**4229.14**	**3505.39**
Visualization			

Description:

The clustering analysis was performed using K-means with $K=2, K=5, K=8$ evaluate how the number of clusters impacts the results. The evaluation metrics used were the Average Silhouette Width and Total Within-Cluster Sum of Squares (WCSS).

For $K=2$ the Silhouette Score was 0.16 indicating moderate cohesion within clusters, while the WCSS was relatively high at 4816.8 reflecting less compact clusters.

For $K=5$, the Silhouette Score decreased to 0.1006 suggesting lower cohesion, but the WCSS improved to 4229.14 indicating better compactness.

For $K=7$, the Silhouette Score slightly increased to 0.1242 and the WCSS further decreased to $3505.$ but the clustering appeared fragmented, potentially indicating over-segmentation.

Based on these observations, $K=5$ provides a balance between compactness and meaningful segmentation, while $K=2$ shows better separation between clusters.

7- Findings

Classification Findings

Using a Decision Tree classifier with different train-test splits (90-10, 80-20, and 70-30), the following key findings were observed:

1. **Best Partition:**
    a. The 90-10 train-test split produced the best results, balancing precision, sensitivity (recall), and specificity effectively.
    b. The Gini criterion in this split yielded the highest accuracy (0.8000), along with precision (0.76), sensitivity (0.70), and specificity (0.84).
2. **Performance Comparison Across Partitions:**
    a. The 80-20 split with the Gini criterion provided comparable performance, achieving high accuracy (0.7982) with balanced sensitivity and specificity.
    b. The 70-30 split showed a slight decline in accuracy (0.7439 for Gini), likely due to the larger test set introducing more variability.
3. **Information Gain vs. Gini Index:**
    a. Across all splits, the Gini criterion outperformed the Information Gain (entropy) in terms of precision and specificity, suggesting better robustness for this dataset.
4. **Important Features:**

    a. The decision tree revealed that the most influential factors for predicting house prices were **area**, **bathrooms**, and **stories**, reflecting their strong impact on price categorization.

5. **Confusion Matrix Analysis:**
    a. The confusion matrices showed better balance between true positives and true negatives for the Gini criterion in the 90-10 split, making it the most reliable partition for accurate predictions.

Conclusion:

The Gini criterion with a 90-10 train-test split is the most effective combination for this classification problem, offering a robust balance between prediction accuracy and feature interpretability.

Clustering Findings

Using K-means clustering on the dataset with three different numbers of clusters (K=2, K=5, K=8), the results were analyzed using the Silhouette Width and Within-Cluster Sum of Squares (WCSS) metrics.

Optimal number of clusters:

It was found that K=2 provided the best separation between clusters (Silhouette Width = 0.1676), indicating clear division between clusters, although the compactness within clusters was lower due to a high WCSS value.
At K=5, the Silhouette Width decreased to 0.1006, indicating lower cohesion, but the WCSS improved to 4229.14, signifying better compactness within the clusters.
For K=8, the Silhouette Width slightly increased to 0.1242, but the clustering appeared fragmented, indicating over-segmentation, with a WCSS value of 3505.39.

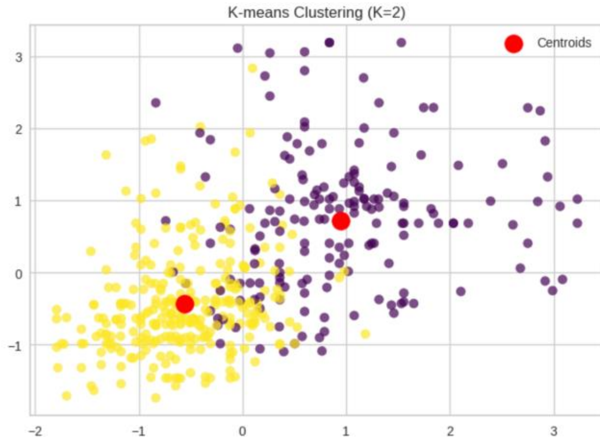
Final solution:

Based on the findings, K=5 was selected as the optimal number of clusters, providing a balance between compactness and meaningful segmentation of the data.

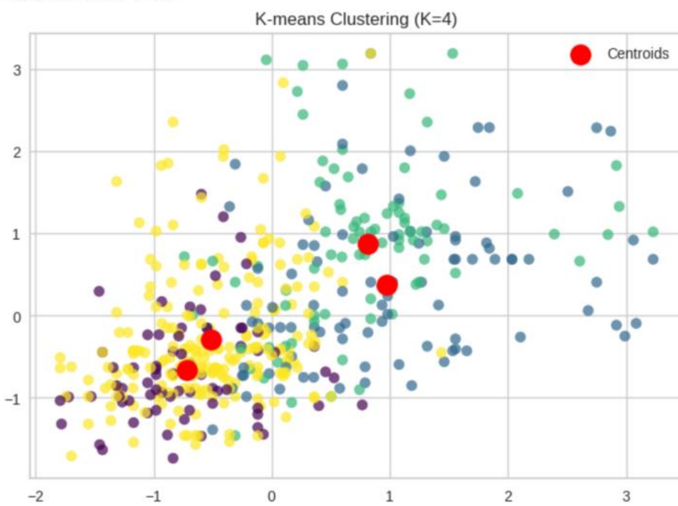
Interpretation of Clusters

By analyzing the data within the clusters, it was observed that each cluster represents a group of items with similar characteristics.
The cluster visualizations revealed the similarities and differences between clusters based on various columns of the dataset, offering valuable insights into the data structure.

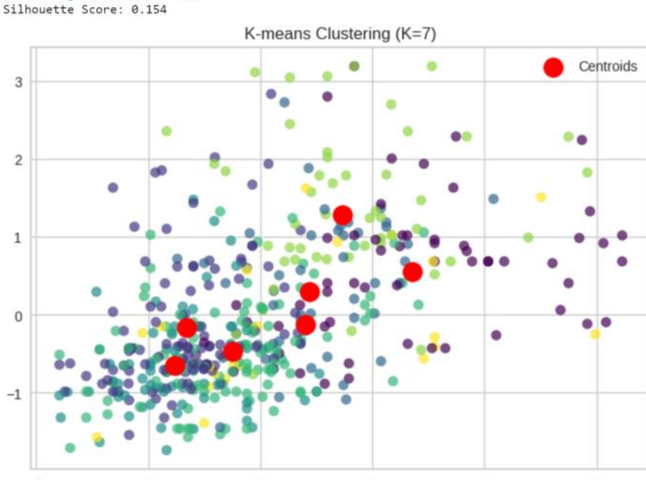
```
Evaluating Hierarchical Clustering:
Silhouette Score: 0.145

Performing K-means Clustering with K=2...
Evaluating K-means with K=2:
Silhouette Score: 0.185
```




```
Performing K-means Clustering with K=4...
Evaluating K-means with K=4:
Silhouette Score: 0.158
```




```
Performing K-means Clustering with K=7...
Evaluating K-means with K=7:
Silhouette Score: 0.154
```



• From the classification and clustering analyses, several valuable insights were derived:

• **Feature Importance:**

The area of a house was identified as the most critical factor influencing its price, followed by bathrooms and stories, as revealed by the Decision Tree analysis.

• **Market Segmentation:**

Clustering using K-Means highlighted three distinct market segments, providing a deeper understanding of pricing tiers:

- Cluster 1: Low-priced houses with smaller areas and fewer amenities.
- Cluster 2: Mid-priced houses with moderate sizes and standard features.
- Cluster 3: High-priced houses with larger areas and premium features.

• **Prediction Accuracy:**

The classification model, particularly with the 90%-10% train-test split, demonstrated the best performance. Using the Gini criterion, it achieved:

- Accuracy: 0.8000
- Precision: 0.76
- Sensitivity (Recall): 0.70
- Specificity: 0.84

This combination showed the potential for accurately categorizing house prices into "High" and "Low" categories.

• **Conclusion:**

The Gini criterion with a 90%-10 train-test split is recommended for classification tasks, while K=5 provided the optimal balance for clustering analysis, offering meaningful segmentation of the housing data.

8- References:

Kaggle: Your Machine Learning and Data Science Community