Project 01 : **House Price Prediction**

Import all the required libraries

Install all the required libraries -> pip install numpy

Install libraries using your terminal

conda install pandas numpy seaborn matplotlib

- Pandas: For data manipulation and analysis.
- NumPy: For numerical operations.
- Seaborn and Matplotlib: For data visualization.

In [3]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

Load the dataset california housing

conda install scikit-learn

- fetch_california_housing: Loads the California Housing dataset from sklearn.
- Determines the type of the housing object, which is a Bunch (a dictionary-like object).
- Displays the dataset's details, including data and target values.
- Prints the description of the dataset, explaining features, target, and other information.
- Lists the names of the features (input variables).
- Displays the target variable, which is the median house value.
- Displays the feature data in a NumPy array.

In [4]:
```
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
```

In [5]:
```
type(housing)
```

Out[5]:
```
sklearn.utils._bunch.Bunch
```

In [6]:
```
housing
```

Out[6]:
```
{'data': array([[   8.3252    ,   41.        ,    6.98412698, ...,    2.
55555556,
           37.88      , -122.23      ],
        [   8.3014    ,   21.        ,    6.23813708, ...,    2.1098418
3,
           37.86      , -122.22      ],
        [   7.2574    ,   52.        ,    8.28813559, ...,    2.8022598
9,
           37.85      , -122.24      ],
        ...,
        [   1.7       ,   17.        ,    5.20554273, ...,    2.3256351
,
           39.43      , -121.22      ],
        [   1.8672    ,   18.        ,    5.32951289, ...,    2.1232091
7,
           39.43      , -121.32      ],
        [   2.3886    ,   16.        ,    5.25471698, ...,    2.6169811
3,
           39.37      , -121.24      ]]),
 'target': array([4.526, 3.585, 3.521, ..., 0.923, 0.847, 0.894]),
 'frame': None,
 'target_names': ['MedHouseVal'],
 'feature_names': ['MedInc',
  'HouseAge',
  'AveRooms',
  'AveBedrms',
  'Population',
  'AveOccup',
  'Latitude',
  'Longitude'],
 'DESCR': '.. _california_housing_dataset:\n\nCalifornia Housing dataset
\n--------------------------\n\n**Data Set Characteristics:**\n\n:Number
of Instances: 20640\n\n:Number of Attributes: 8 numeric, predictive attr
ibutes and the target\n\n:Attribute Information:\n    - MedInc        me
dian income in block group\n    - HouseAge      median house age in bloc
k group\n    - AveRooms      average number of rooms per household\n    
- AveBedrms     average number of bedrooms per household\n    - Populati
on    block group population\n    - AveOccup      average number of hous
ehold members\n    - Latitude      block group latitude\n    - Longitude
block group longitude\n\n:Missing Attribute Values: None\n\nThis dataset
was obtained from the StatLib repository.\nhttps://www.dcc.fc.up.pt/~lto
rgo/Regression/cal_housing.html\n\nThe target variable is the median hou
se value for California districts,\nexpressed in hundreds of thousands o
f dollars ($100,000).\n\nThis dataset was derived from the 1990 U.S. cen
sus, using one row per census\nblock group. A block group is the smalles
t geographical unit for which the U.S.\nCensus Bureau publishes sample d
ata (a block group typically has a population\nof 600 to 3,000 peopl
e).\n\nA household is a group of people residing within a home. Since th
e average\nnumber of rooms and bedrooms in this dataset are provided per
household, these\ncolumns may take surprisingly large values for block g
roups with few households\nand many empty houses, such as vacation resor
ts.\n\nIt can be downloaded/loaded using the\n:func:`sklearn.datasets.fe
tch_california_housing` function.\n\n.. rubric:: References\n\n- Pace,
R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,\n  Statistic
s and Probability Letters, 33 (1997) 291-297\n'}
```

In [7]:
```python
print(housing.DESCR)
```

```
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

:Number of Instances: 20640

:Number of Attributes: 8 numeric, predictive attributes and the target

:Attribute Information:
    - MedInc        median income in block group
    - HouseAge      median house age in block group
    - AveRooms      average number of rooms per household
    - AveBedrms     average number of bedrooms per household
    - Population    block group population
    - AveOccup      average number of household members
    - Latitude      block group latitude
    - Longitude     block group longitude

:Missing Attribute Values: None

This dataset was obtained from the StatLib repository.
https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html

The target variable is the median house value for California districts,
expressed in hundreds of thousands of dollars ($100,000).

This dataset was derived from the 1990 U.S. census, using one row per cens
us
block group. A block group is the smallest geographical unit for which the
U.S.
Census Bureau publishes sample data (a block group typically has a populat
ion
of 600 to 3,000 people).

A household is a group of people residing within a home. Since the average
number of rooms and bedrooms in this dataset are provided per household, t
hese
columns may take surprisingly large values for block groups with few house
holds
and many empty houses, such as vacation resorts.

It can be downloaded/loaded using the
:func:`sklearn.datasets.fetch_california_housing` function.

.. rubric:: References

- Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
  Statistics and Probability Letters, 33 (1997) 291-297
```

In [8]:
```
print(housing.feature_names)
```

```
['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup',
'Latitude', 'Longitude']
```

In [9]:
```
print(housing.target)
```

```
[4.526 3.585 3.521 ... 0.923 0.847 0.894]
```

In [10]:
```
print(housing.data)
```

```
[[   8.3252       41.            6.98412698 ...    2.55555556
    37.88       -122.23      ]
 [   8.3014       21.            6.23813708 ...    2.10984183
    37.86       -122.22      ]
 [   7.2574       52.            8.28813559 ...    2.80225989
    37.85       -122.24      ]
 ...
 [   1.7          17.            5.20554273 ...    2.3256351
    39.43       -121.22      ]
 [   1.8672       18.            5.32951289 ...    2.12320917
    39.43       -121.32      ]
 [   2.3886       16.            5.25471698 ...    2.61698113
    39.37       -121.24      ]]
```

Prepare the data

Steps taken:

- Convert the NumPy array into a Pandas DataFrame for easier data manipulation.
- Verify that the dataset is a Pandas DataFrame.
- Display the first five rows of the DataFrame to get an overview of the data.
- Display the last five rows of the DataFrame.
- Add target variable (Price) to the DataFrame.
- Display the first few rows, including the newly added Price column.
- Provide a concise summary of the DataFrame, including data types and non-null counts.
- Display a statistical summary (mean, standard deviation, min, max, etc.) of the dataset.
- Check if there are any missing values in the dataset.

In [11]:
```
dataset = pd.DataFrame(housing.data, columns=housing.feature_names)
```

In [12]:
```
type(dataset)
```

Out[12]:
```
pandas.core.frame.DataFrame
```

In [13]:
```
dataset.head()
```

Out[13]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Lo
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	

In [14]:
```
dataset.tail()
```

Out[14]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitud
20635	1.5603	25.0	5.045455	1.133333	845.0	2.560606	39.4
20636	2.5568	18.0	6.114035	1.315789	356.0	3.122807	39.4
20637	1.7000	17.0	5.205543	1.120092	1007.0	2.325635	39.4
20638	1.8672	18.0	5.329513	1.171920	741.0	2.123209	39.4
20639	2.3886	16.0	5.254717	1.162264	1387.0	2.616981	39.3

In [15]:
```
dataset['Price'] = housing.target
```

In [16]:
```
dataset.head()
```

Out[16]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Lo
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	

In [17]:
```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 9 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   MedInc      20640 non-null  float64
 1   HouseAge    20640 non-null  float64
 2   AveRooms    20640 non-null  float64
 3   AveBedrms   20640 non-null  float64
 4   Population  20640 non-null  float64
 5   AveOccup    20640 non-null  float64
 6   Latitude    20640 non-null  float64
 7   Longitude   20640 non-null  float64
 8   Price       20640 non-null  float64
dtypes: float64(9)
memory usage: 1.4 MB
```

Statistical description of the data

In [18]:
```
dataset.describe()
```

Out[18]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population
count	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	3.870671	28.639486	5.429000	1.096675	1425.476744
std	1.899822	12.585558	2.474173	0.473911	1132.462122
min	0.499900	1.000000	0.846154	0.333333	3.000000
25%	2.563400	18.000000	4.440716	1.006079	787.000000
50%	3.534800	29.000000	5.229129	1.048780	1166.000000
75%	4.743250	37.000000	6.052381	1.099526	1725.000000
max	15.000100	52.000000	141.909091	34.066667	35682.000000

In [19]:
```
## check the null values
dataset.isnull().sum()
```

Out[19]:
```
MedInc        0
HouseAge      0
AveRooms      0
AveBedrms     0
Population    0
AveOccup      0
Latitude      0
Longitude     0
Price         0
dtype: int64
```

What is EDA (Exploratory Data Analysis)?

Exploratory Data Analysis (EDA) is a critical process in data science and machine learning where analysts use various techniques to summarize and visualize the main characteristics of a dataset. The primary goal of EDA is to understand the underlying structure of the data, identify patterns, detect anomalies, test hypotheses, and check assumptions before applying more complex modeling techniques.

Key Objectives of EDA

1. **Understanding Data Distribution**:
   - Analyze the central tendency (mean, median, mode), variability (variance, standard deviation), and shape (skewness, kurtosis) of the data.
   - Visualize data distribution using histograms, boxplots, and density plots to see how the data is spread across different values.
2. **Identifying Patterns and Relationships**:
   - Use scatter plots, pair plots, and correlation matrices to examine the relationships between different variables.
   - Detect linear or non-linear associations that can inform feature selection and engineering.
3. **Detecting Outliers**:

- Identify unusual data points that deviate significantly from other observations using boxplots, z-scores, or scatter plots.
- Determine whether to remove or transform outliers depending on their impact on the analysis or model.

4. **Handling Missing Data**:
    - Assess the amount and pattern of missing data in the dataset.
    - Decide on strategies to handle missing values, such as imputation or removal, based on the nature and extent of the missing data.
5. **Feature Engineering and Selection**:
    - Identify potential new features that can be created from existing data (feature engineering).
    - Evaluate the relevance and importance of features to the target variable, which can help in selecting the most informative features for modeling.
6. **Checking Assumptions**:
    - Validate the assumptions required for statistical models (e.g., normality, homoscedasticity, linearity).
    - Adjust the data or choose alternative models if assumptions are violated.
7. **Preliminary Data Cleaning**:
    - Identify and address inconsistencies, errors, or data quality issues in the dataset.
    - Ensure the data is in a suitable format for analysis and modeling.

Common Techniques and Tools Used in EDA

- **Visualizations**:
    - **Histograms**: To understand the distribution of a single variable.
    - **Boxplots**: To detect outliers and understand the spread of the data.
    - **Scatter plots**: To examine relationships between two variables.
    - **Correlation matrices**: To assess the strength and direction of linear relationships between multiple variables.
    - **Heatmaps**: To visualize the correlation matrix or missing data patterns.
- **Descriptive Statistics**:
    - **Mean, Median, Mode**: To measure the central tendency of data.
    - **Variance and Standard Deviation**: To understand the spread of the data.
    - **Skewness and Kurtosis**: To assess the symmetry and peakedness of the data distribution.
- **Data Aggregation and Grouping**:
    - Summarize data using group-by operations to explore differences across categories or time periods.
- **Dimensionality Reduction**:

    - Techniques like PCA (Principal Component Analysis) can be used to visualize high-dimensional data in lower dimensions.

Importance of EDA

- **Informed Decision-Making**: EDA helps in making informed decisions about data preprocessing, feature selection, and model choice.
- **Data Quality Assessment**: It provides insights into the quality of the data and helps in identifying potential issues early on.
- **Hypothesis Testing**: EDA allows you to test hypotheses and explore the data before formal modeling, reducing the risk of model overfitting or bias.

Conclusion

EDA is an essential step in the data analysis process that helps analysts gain a deep understanding of their data. By leveraging various statistical tools and visualization techniques, EDA enables you to uncover insights, identify patterns, and make data-driven decisions that guide subsequent modeling and analysis efforts.

EDA -> Exploratory Data Analysis

- Compute the correlation matrix, which shows the relationships between the features and the target.
- Create a pairplot to visualize the relationships between features and the target.
- Generate a boxplot to detect outliers in the dataset and saves the plot as an image.
- Separate the independent variables (features) and the dependent variable (target).
- Display the feature data.
- Display the target data.
- Split the data into training (70%) and testing (30%) sets using a random seed for reproducibility.
- Display the training feature data.
- Display the testing feature data.
- Display the training target data.
- Display the testing target data.
- Apply standard normalization to the training data (mean = 0, variance = 1).
- Display the normalized training data.
- Create a boxplot for the normalized training data and saves it as an image.
- Apply the same normalization to the testing data.
- Create a boxplot for the normalized testing data and saves it as an image.
- Comment explaining the typical process of fitting the scaler on the training data and transforming both the training and testing data.

Correlation

Correlation and Its Importance in Machine Learning (with Respect to Feature Selection)

What is Correlation?

Correlation is a statistical measure that expresses the extent to which two variables are linearly related. In other words, it shows the degree of association between two variables. The value of correlation ranges between -1 and 1:

- **+1** indicates a perfect positive linear relationship.
- **-1** indicates a perfect negative linear relationship.
- **0** indicates no linear relationship.

Types of Correlation:

- **Positive Correlation**: As one variable increases, the other variable also increases (e.g., height and weight).
- **Negative Correlation**: As one variable increases, the other variable decreases (e.g., temperature and heating bill).
- **Zero Correlation**: No relationship between the variables.

Importance of Correlation in Feature Selection

Feature selection is a crucial step in building a machine learning model. Correlation plays an important role in this process for several reasons:

1. **Identifying Redundant Features**:

    - Highly correlated features provide similar information. Including them all can lead to redundancy, causing the model to overfit or become unnecessarily complex.
    - For example, if `feature A` and `feature B` have a correlation close to +1 or -1, it may be beneficial to keep only one of them to simplify the model.
2. **Improving Model Performance**:

    - Reducing multicollinearity by removing one of the correlated features can improve the performance and interpretability of the model.
    - Multicollinearity, which occurs when two or more features are highly correlated, can cause problems in linear models (e.g., regression), making it difficult to determine the effect of each feature.
3. **Dimensionality Reduction**:

    - By identifying and removing features that are highly correlated with others, the dimensionality of the data can be reduced, leading to faster training times and potentially better model generalization.
4. **Feature Engineering**:

    - Understanding the correlation between features and the target variable helps in feature engineering. Features that are highly correlated with the target are often more informative and can be prioritized or transformed to improve the model.

How to Use Correlation in Feature Selection:

- **Correlation Matrix**:
  - A correlation matrix is a table showing correlation coefficients between variables. It is often used to visualize the relationships between multiple features.
  - By analyzing this matrix, one can identify pairs of features that are highly correlated and consider removing one of them.
- **Thresholding**:
  - A common practice is to set a threshold (e.g., 0.8 or 0.9) to decide when to drop one of the correlated features. If the absolute value of the correlation between two features exceeds this threshold, one of them can be removed.
- **Feature Selection Techniques**:
  - Techniques like Variance Inflation Factor (VIF) can be used to quantify multicollinearity and help in selecting features. VIF is a measure of how much the variance of a regression coefficient is inflated due to multicollinearity.

Example:

Consider a dataset where `feature A` and `feature B` have a correlation coefficient of 0.95. Since these two features are highly correlated, they provide similar information. Including both in a regression model might lead to multicollinearity, making it hard to interpret the coefficients. Therefore, it might be advisable to drop one of these features to simplify the model.

Conclusion:

Correlation is a powerful tool in the feature selection process of machine learning. By understanding and leveraging correlation, one can reduce redundancy, avoid multicollinearity, and ultimately create more efficient and interpretable models.

In current example we have below noticible corelation:

- AveBedrooms and AveRooms : 0.847621
- Longitude and Latitude : -0.924664

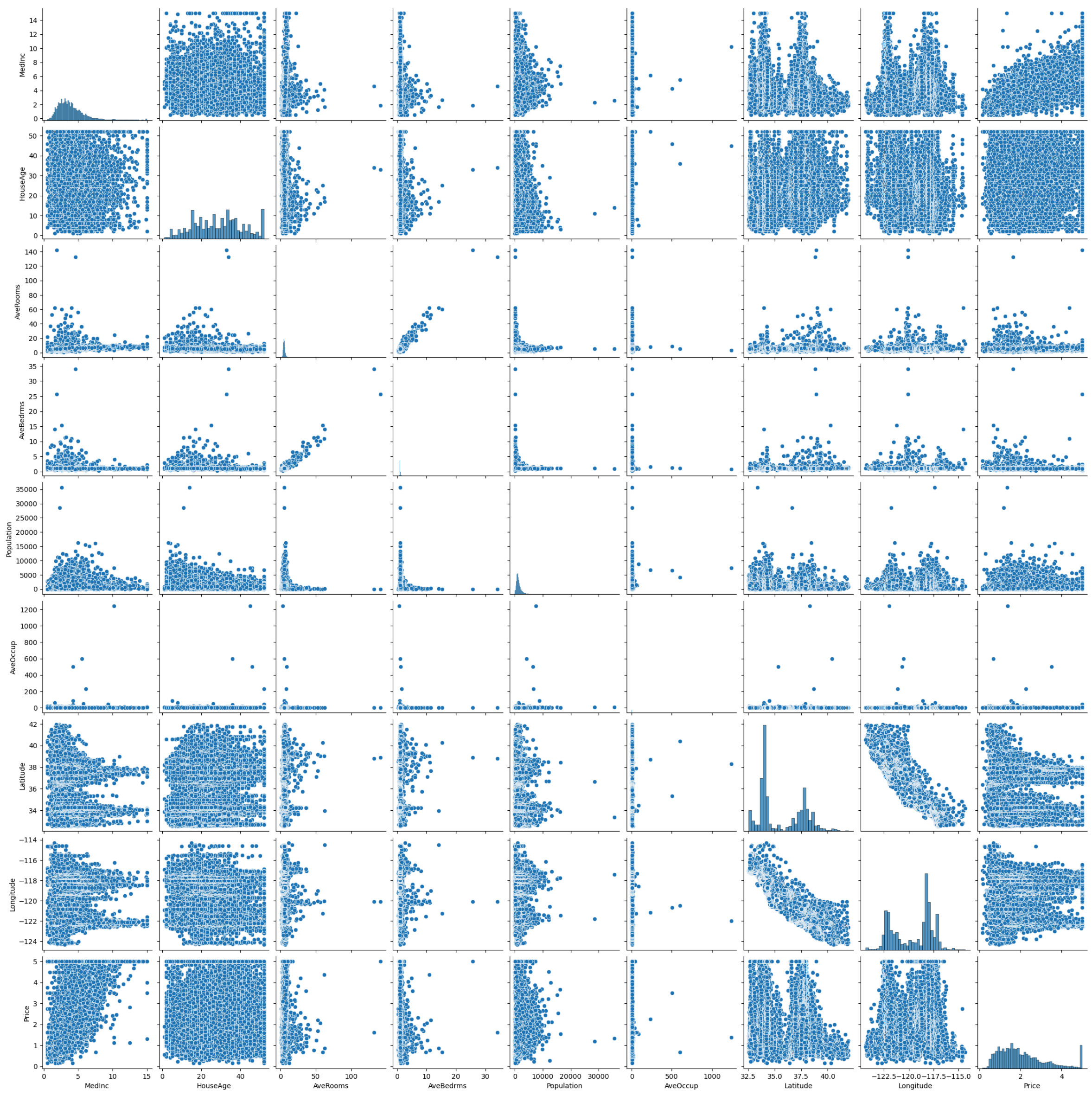
In [21]:
```
dataset.corr()
```

Out[21]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup
MedInc	1.000000	-0.119034	0.326895	-0.062040	0.004834	0.018766
HouseAge	-0.119034	1.000000	-0.153277	-0.077747	-0.296244	0.013191
AveRooms	0.326895	-0.153277	1.000000	0.847621	-0.072213	-0.004852
AveBedrms	-0.062040	-0.077747	0.847621	1.000000	-0.066197	-0.006181
Population	0.004834	-0.296244	-0.072213	-0.066197	1.000000	0.069863
AveOccup	0.018766	0.013191	-0.004852	-0.006181	0.069863	1.000000
Latitude	-0.079809	0.011173	0.106389	0.069721	-0.108785	0.002366
Longitude	-0.015176	-0.108197	-0.027540	0.013344	0.099773	0.002476
Price	0.688075	0.105623	0.151948	-0.046701	-0.024650	-0.023737

In [22]:
```python
# With below pairplot we can visualize the correlation
sns.pairplot(dataset)
```

Out[22]:
```
<seaborn.axisgrid.PairGrid at 0x16d0f2d40>
```



Normalization and Standardization in Machine Learning

What is Normalization?

Normalization is the process of scaling individual features to a specific range, typically [0, 1]. This technique is especially useful when you want to ensure that each feature contributes equally to the distance computations in algorithms like k-Nearest Neighbors (k-NN) or when the data has varying scales.

- **Formula**:

X_norm = (X − X_min)/(X_max − X_min)

where X_min and X_max are the minimum and maximum values of the feature, respectively.

What is Standardization?

Standardization (also known as Z-score normalization) transforms data to have a mean of 0 and a standard deviation of 1. This technique is particularly useful for algorithms that assume the data is normally distributed (e.g., linear regression, logistic regression, support vector machines).

- **Formula**:

X_std = (X − mue)/sigma

where mue is the mean of the feature and sigma is the standard deviation.

Why Use Normalization and Standardization?

1. **Improves Convergence Speed**: Many machine learning algorithms (like gradient descent) converge faster when features are on similar scales.

2. **Prevents Dominance**: Features with larger ranges can dominate those with smaller ranges if not normalized, leading to biased models.

3. **Better Performance in Distance-Based Algorithms**: In algorithms like k-NN or SVM, features with larger scales can dominate distance calculations, leading to inaccurate predictions.

4. **Ensures Equal Contribution**: Both techniques ensure that each feature contributes equally to the model, particularly in models sensitive to feature magnitude (e.g., neural networks).

Steps to Identify the Need for Normalization and Standardization

1. **Inspect Feature Ranges**:

- **Check the range of each feature** in the dataset. If the features have significantly different scales (e.g., one feature ranges from 0 to 1 while another ranges from 1 to 10,000), normalization or standardization might be needed.

2. **Analyze the Data Distribution**:

   - **Check the distribution of features**. If features are normally distributed, standardization might be preferred. If not, normalization might be more suitable.

3. **Use Correlation Matrix**:

   - **Examine the correlation matrix** to understand the relationships between features. Highly correlated features might benefit from standardization, especially for linear models.

Steps to Normalize and Standardize the Data

1. **Split the Data into Training and Testing Sets**:

   - Split your data into training and testing sets to ensure that the scaling is not influenced by the testing data, which could lead to data leakage.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

2. **Implement Normalization**:

   - Apply normalization to the training data and then transform the testing data using the same parameters.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)
```

3. **Implement Standardization**:

   - Apply standardization to the training data and then transform the testing data using the same parameters.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.transform(X_test)
```

Checking the Effects of Normalization and Standardization

1. **Re-Evaluate Feature Distributions**:

   - After normalization or standardization, plot the distributions of the transformed features. For normalized features, they should now lie within the [0, 1] range, and standardized features should have a mean of 0 and standard deviation of 1.

```
import seaborn as sns
import matplotlib.pyplot as plt
```

```
sns.histplot(X_train_norm[:,0], kde=True)
plt.title('Distribution after Normalization')
plt.show()

sns.histplot(X_train_std[:,0], kde=True)
plt.title('Distribution after Standardization')
plt.show()
```

2. **Re-Train the Model**:

   - Train your machine learning model on the normalized or standardized data and compare performance metrics (e.g., accuracy, MSE, R-squared).

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Train on Normalized Data
model_norm = LinearRegression()
model_norm.fit(X_train_norm, y_train)
y_pred_norm = model_norm.predict(X_test_norm)

# Train on Standardized Data
model_std = LinearRegression()
model_std.fit(X_train_std, y_train)
y_pred_std = model_std.predict(X_test_std)

# Compare Performance
print("MSE (Normalized):", mean_squared_error(y_test,
y_pred_norm))
print("R-squared (Normalized):", r2_score(y_test,
y_pred_norm))

print("MSE (Standardized):", mean_squared_error(y_test,
y_pred_std))
print("R-squared (Standardized):", r2_score(y_test,
y_pred_std))
```

3. **Compare Model Convergence**:

   - If using iterative algorithms (like gradient descent), compare the number of iterations required to converge with and without normalization or standardization. Normalized and standardized data often lead to faster convergence.

Conclusion

Normalization and standardization are critical preprocessing steps in machine learning, ensuring that features contribute equally to the model and preventing one feature from dominating others. By carefully analyzing your data, implementing these techniques, and evaluating their effects, you can significantly improve your model's performance and reliability.

BoxPlot : To detect the outliers in a given dataset

How Boxplot Helps in Identifying the Need for Normalization

A **boxplot** (or box-and-whisker plot) is a graphical representation of data that summarizes the distribution of a dataset through its quartiles. It provides a visual insight into the data's central tendency, dispersion, and outliers, making it a useful tool for identifying the need for normalization.

Components of a Boxplot:

- **Median (Q2)**: The line inside the box represents the median of the data.
- **First Quartile (Q1)**: The lower edge of the box, representing the 25th percentile.
- **Third Quartile (Q3)**: The upper edge of the box, representing the 75th percentile.
- **Interquartile Range (IQR)**: The range between Q1 and Q3, showing the middle 50% of the data.
- **Whiskers**: Lines extending from the box, typically representing 1.5 times the IQR from the quartiles. They show the range of the data, excluding outliers.
- **Outliers**: Data points outside the whiskers, often plotted as individual dots.

Identifying the Need for Normalization Using Boxplots

1. **Detecting Disparities in Scale**:

    - **Uneven Spread**: If different features have boxplots with significantly different spreads (the distance between Q1 and Q3), it indicates that the features are on different scales. This disparity suggests the need for normalization to bring the features onto a similar scale.
    - **Example**: A boxplot of `feature A` might show a very narrow box (small IQR) while `feature B` shows a much wider box. This difference in spread indicates that the features have different scales and may require normalization.
2. **Comparing Multiple Features**:

    - **Multiple Boxplots**: When comparing the boxplots of several features side by side, you can easily see if the features are on different scales. Features with wider boxes (larger IQRs) and longer whiskers (wider range) might dominate those with narrower boxes.
    - **Effect of Skewness**: If one feature's boxplot is heavily skewed compared to others, normalization can help adjust for this skewness and make the data more comparable across features.
3. **Presence of Outliers**:

    - **Impact of Outliers**: A boxplot that shows many outliers (points outside the whiskers) might indicate that the feature has a long tail or extreme values. Normalization can help reduce the impact of these outliers, making the data more robust for algorithms sensitive to scale.
    - **Normalization Strategy**: After identifying outliers through the boxplot, normalization can be applied to compress the range and minimize the

influence of these extreme values.

4. **Before-and-After Normalization Comparison**:

   - **Visual Validation**: After normalizing the data, you can create a new boxplot to compare it with the original. Ideally, the normalized data should have more uniform boxplots across features, indicating that they are now on a similar scale.
   - **Consistency in Distribution**: Post-normalization, if the boxplots of all features show similar spreads and the whiskers are more consistent, it confirms that normalization has effectively brought the data to a comparable scale.

Example: Boxplot in Identifying the Need for Normalization

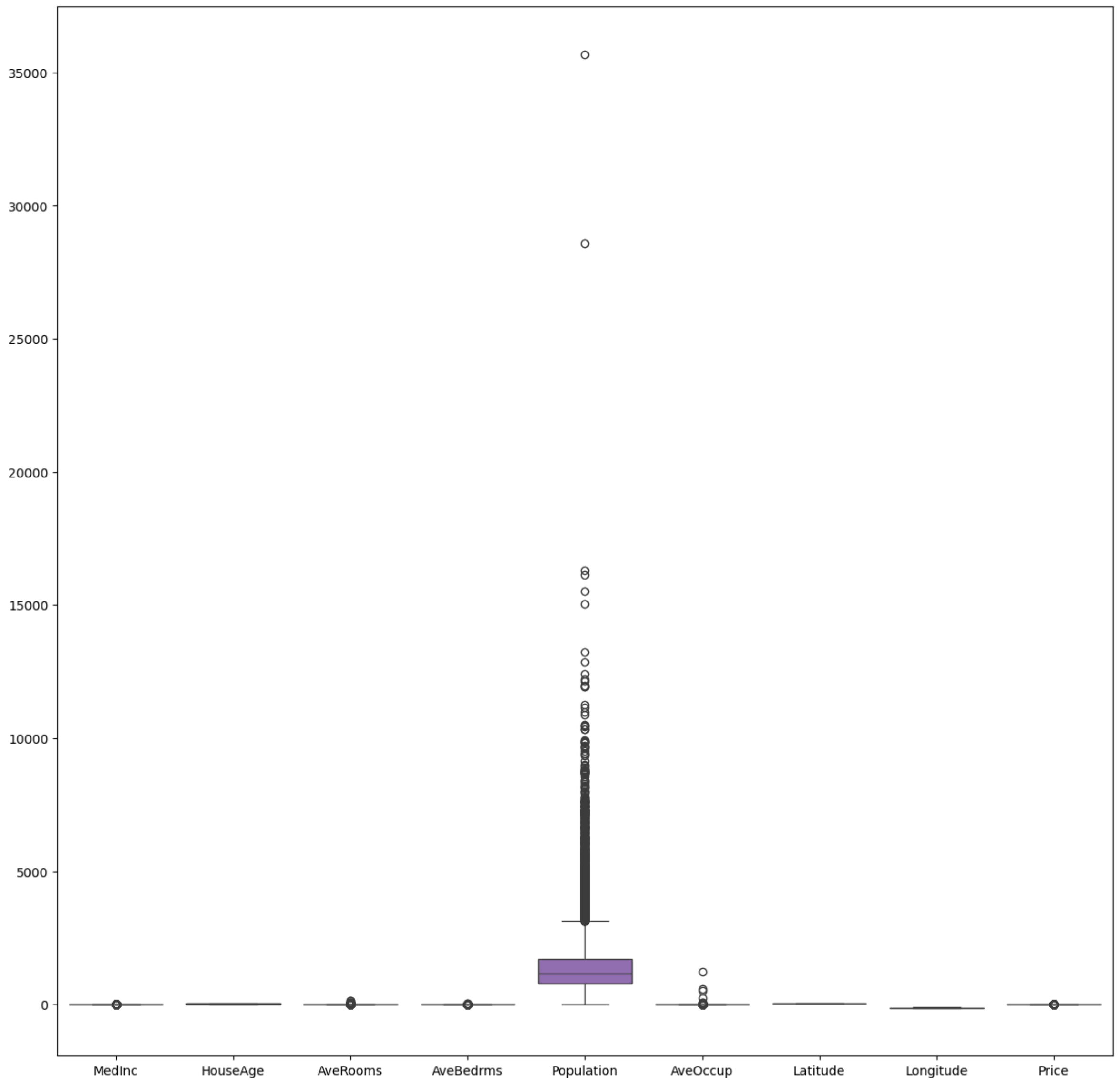
```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data with different scales
data = {'Feature A': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        'Feature B': [100, 200, 300, 400, 500, 600, 700, 800,
900, 1000],
        'Feature C': [10, 12, 14, 16, 18, 20, 22, 24, 26, 28]}

# Create a boxplot
sns.boxplot(data=data)
plt.title('Boxplot of Features Before Normalization')
plt.show()
```

- **Interpretation**: If the boxplot shows `Feature B` with a much larger range than `Feature A` or `Feature C`, it indicates that the features are on different scales, suggesting the need for normalization.

Conclusion

Boxplots provide a quick and effective way to visually inspect the distribution and scale of features in a dataset. By analyzing the spread, range, and presence of outliers in boxplots, you can identify whether normalization is needed to bring features to a similar scale, ensuring they contribute equally to your machine learning model.

In [23]:
```
fig, ax = plt.subplots(figsize=(15,15))
sns.boxplot(data = dataset, ax=ax)
plt.savefig("boxPlot.jpg")
```



In [54]:
```
## Split the data into independent and dependent features
X = dataset.iloc[:,:-1]
y = dataset.iloc[:,-1]
```

In [25]:
```
X
```

Out[25]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitud
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.8
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.8
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.8
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.8
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.8
...	...	...	...	...	...	...	.
20635	1.5603	25.0	5.045455	1.133333	845.0	2.560606	39.4
20636	2.5568	18.0	6.114035	1.315789	356.0	3.122807	39.4
20637	1.7000	17.0	5.205543	1.120092	1007.0	2.325635	39.4
20638	1.8672	18.0	5.329513	1.171920	741.0	2.123209	39.4
20639	2.3886	16.0	5.254717	1.162264	1387.0	2.616981	39.3

20640 rows × 8 columns

In [26]:
```
y
```

Out[26]:
```
0        4.526
1        3.585
2        3.521
3        3.413
4        3.422
         ...
20635    0.781
20636    0.771
20637    0.923
20638    0.847
20639    0.894
Name: Price, Length: 20640, dtype: float64
```

What is `random_state`?

The `random_state` parameter in scikit-learn is a crucial component that controls the randomness of various operations, such as splitting data into training and test sets, initializing random numbers, and shuffling data. It ensures that the results are reproducible and consistent across different runs of your code.

Why Use `random_state`?

1. **Reproducibility**:
   - When you set a specific `random_state` (e.g., `random_state=42`), it ensures that every time you run the code, you get the same result. This is important for debugging, model comparison, and sharing your work with others.
2. **Consistency in Model Evaluation**:

- In machine learning, consistent training and test splits allow you to evaluate different models on the same subsets of data. This makes it easier to compare the performance of models, as they are tested under identical conditions.

3. **Control Over Random Processes**:

   - Algorithms like train-test splitting, cross-validation, and parameter initialization often involve randomness. The `random_state` acts as a seed to control these random processes, making your experiments more controlled and interpretable.

Example: Splitting Data with `train_test_split`

The `train_test_split` function is used to split a dataset into training and testing subsets. The `random_state` parameter ensures that the split is the same every time you run the code.

```
from sklearn.model_selection import train_test_split

# Splitting the data into training and testing sets with
random_state set to 42
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

Explanation:

- **`X`** : Features (input data).
- **`y`** : Target variable (output data).
- **`test_size=0.3`** : 30% of the data is allocated to the test set, and the remaining 70% to the training set.
- **`random_state=42`** : Ensures the data split is reproducible every time the code is run.

Why Choose `random_state=42`?

- The value `42` is arbitrary and commonly used as a convention. It doesn't hold any special significance, but it's frequently used in examples and tutorials for simplicity. You can choose any integer value for `random_state`.

Summary

Using `random_state` is essential for ensuring that your machine learning experiments are reproducible and consistent. Whether you are splitting data, initializing models, or performing random operations, setting a `random_state` helps you maintain control over the randomness in your code, leading to more reliable and interpretable results.

In [27]:
```
## Split the data into train and test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
```

In [28]:
```
X_train
```

Out[28]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitud
7061	4.1312	35.0	5.882353	0.975490	1218.0	2.985294	33.9
14689	2.8631	20.0	4.401210	1.076613	999.0	2.014113	32.7
17323	4.2026	24.0	5.617544	0.989474	731.0	2.564912	34.5
10056	3.1094	14.0	5.869565	1.094203	302.0	2.188406	39.2
15750	3.3068	52.0	4.801205	1.066265	1526.0	2.298193	37.7
...	...	...	...	...	...	...	.
11284	6.3700	35.0	6.129032	0.926267	658.0	3.032258	33.7
11964	3.0500	33.0	6.868597	1.269488	1753.0	3.904232	34.0
5390	2.9344	36.0	3.986717	1.079696	1756.0	3.332068	34.0
860	5.7192	15.0	6.395349	1.067979	1777.0	3.178891	37.5
15795	2.5755	52.0	3.402576	1.058776	2619.0	2.108696	37.7

14448 rows × 8 columns

In [29]:
```
X_test
```

Out[29]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitud
20046	1.6812	25.0	4.192201	1.022284	1392.0	3.877437	36.0
3024	2.5313	30.0	5.039384	1.193493	1565.0	2.679795	35.1
15663	3.4801	52.0	3.977155	1.185877	1310.0	1.360332	37.8
20484	5.7376	17.0	6.163636	1.020202	1705.0	3.444444	34.2
9814	3.7250	34.0	5.492991	1.028037	1063.0	2.483645	36.6
...	...	...	...	...	...	...	.
17505	2.9545	47.0	4.195833	1.020833	581.0	2.420833	37.3
13512	1.4891	41.0	4.551852	1.118519	994.0	3.681481	34.1
10842	3.5120	16.0	3.762287	1.075614	5014.0	2.369565	33.6
16559	3.6500	10.0	5.502092	1.060371	5935.0	3.547519	37.8
5786	3.0520	17.0	3.355781	1.019695	4116.0	2.614994	34.1

6192 rows × 8 columns

In [30]:
```
y_train
```

Out[30]:
```
7061     1.93800
14689    1.69700
17323    2.59800
10056    1.36100
15750    5.00001
          ...
11284    2.29200
11964    0.97800
5390     2.22100
860      2.83500
15795    3.25000
Name: Price, Length: 14448, dtype: float64
```

In [31]:
```
y_test
```

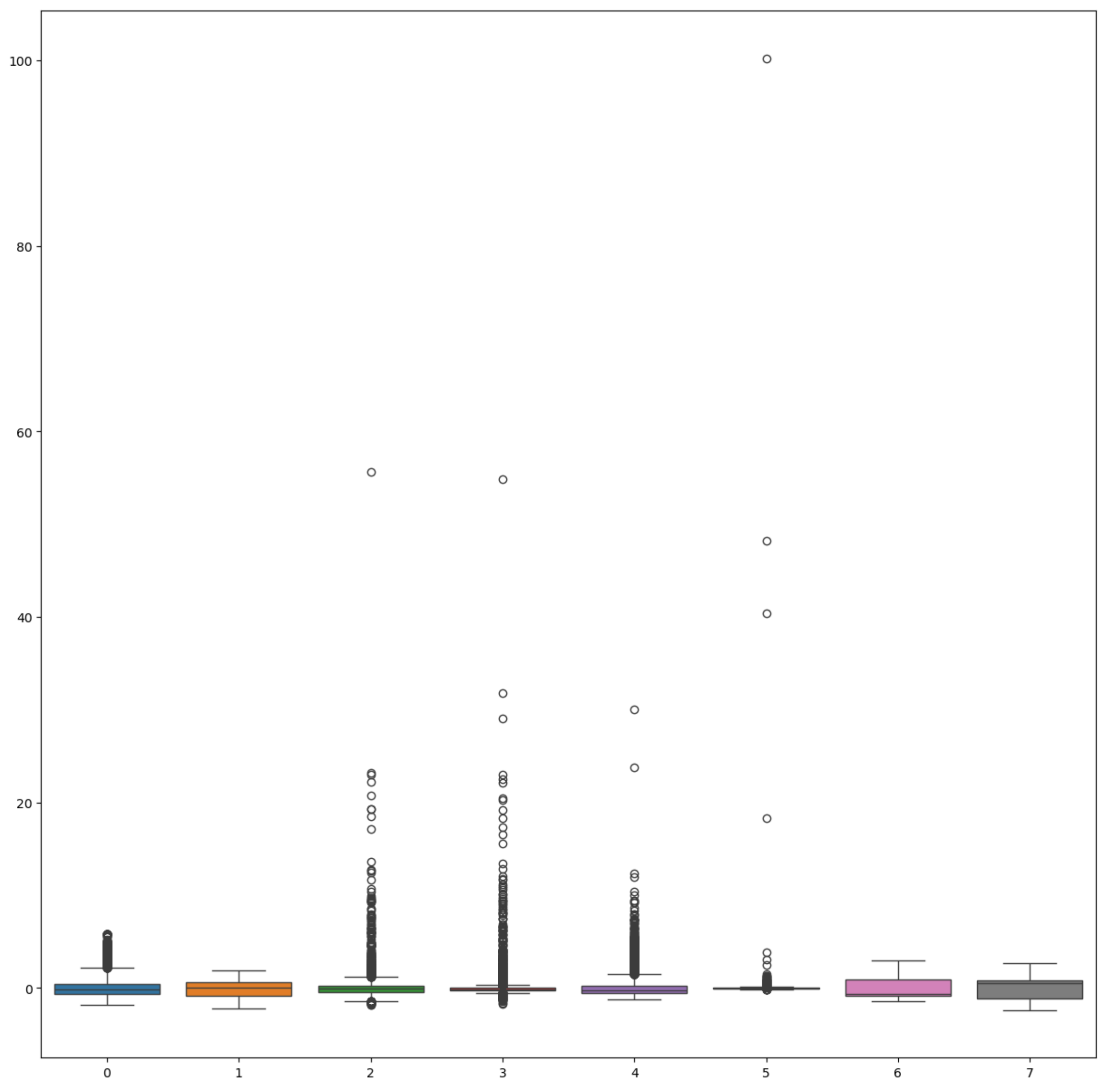
Out[31]:
```
20046    0.47700
3024     0.45800
15663    5.00001
20484    2.18600
9814     2.78000
          ...
17505    2.37500
13512    0.67300
10842    2.18400
16559    1.19400
5786     2.09800
Name: Price, Length: 6192, dtype: float64
```

In [32]:
```
## Normalization of the given data points
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_norm = scaler.fit_transform(X_train)
```

In [33]:
```
X_train_norm
```

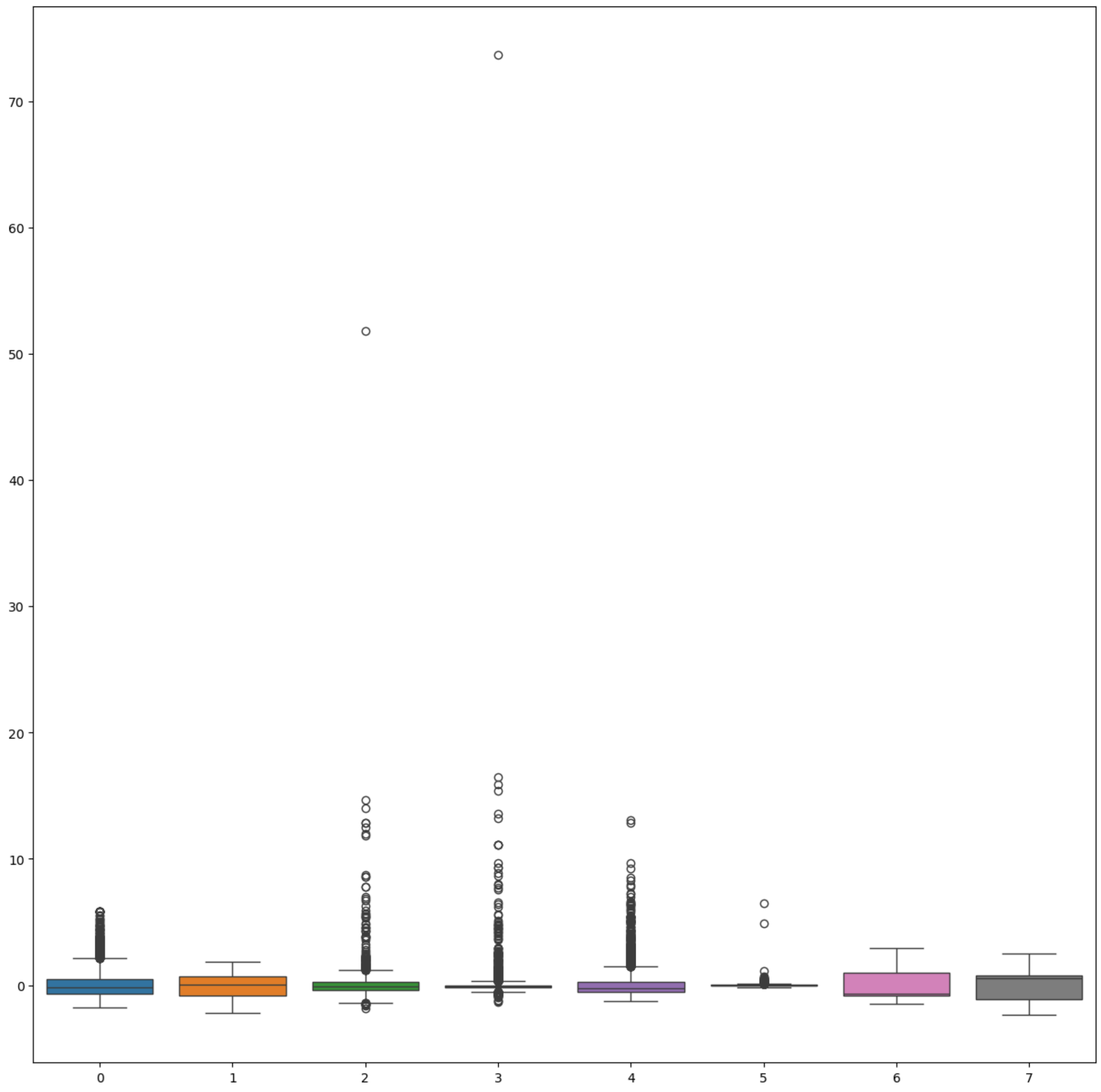
Out[33]:
```
array([[ 0.13350629,  0.50935748,  0.18106017, ..., -0.01082519,
        -0.80568191,  0.78093406],
       [-0.53221805, -0.67987313, -0.42262953, ..., -0.08931585,
        -1.33947268,  1.24526986],
       [ 0.1709897 , -0.36274497,  0.07312833, ..., -0.04480037,
        -0.49664515, -0.27755183],
       ...,
       [-0.49478713,  0.58863952, -0.59156984, ...,  0.01720102,
        -0.75885816,  0.60119118],
       [ 0.96717102, -1.07628333,  0.39014889, ...,  0.00482125,
         0.90338501, -1.18625198],
       [-0.68320166,  1.85715216, -0.82965604, ..., -0.0816717 ,
         0.99235014, -1.41592345]])
```

In [34]:
```
fig, ax = plt.subplots(figsize=(15,15))
sns.boxplot(data = X_train_norm, ax=ax)
plt.savefig("boxPlotTrainData.jpg")
```



In [35]:
```
X_test_norm = scaler.transform(X_test)
```

In [36]:
```
fig, ax = plt.subplots(figsize=(15,15))
sns.boxplot(data = X_test_norm, ax=ax)
plt.savefig("boxPlotTestData.jpg")
```



Why We Use `fit_transform` on the Training Set and `transform` on the Test Set

In machine learning, data preprocessing is a crucial step, especially when dealing with scaling, normalization, or any other transformation that modifies the input features. Understanding the distinction between `fit_transform` and `transform` is essential for ensuring that your model generalizes well to unseen data.

1. `fit_transform` on the Training Set

- **Purpose**: The `fit_transform` method is used to compute the parameters needed for the transformation (e.g., mean and standard deviation for standardization, or min and max for normalization) and then apply the transformation to the data.

- **Why on Training Data?**: The model should be trained and validated on data that reflects how the model will operate in the real world. The training set is the only data that the model is allowed to "see" and use to learn these parameters. This ensures that the transformation reflects the data distribution that the model will learn from.

- **Example**:

  ```
  from sklearn.preprocessing import StandardScaler

  scaler = StandardScaler()
  X_train_scaled = scaler.fit_transform(X_train)
  ```

  - **Here**: `scaler.fit_transform(X_train)` calculates the mean and standard deviation of `X_train` and then scales `X_train` using these statistics.

2. `transform` on the Test Set

- **Purpose**: The `transform` method applies the previously computed transformation parameters (from the training set) to the test data. It does not re-compute the parameters, ensuring that the test set remains completely unseen during training.

- **Why on Test Data?**: To evaluate the model's performance on unseen data, it's crucial that the test set is transformed using the same parameters that were learned from the training set. This mimics real-world scenarios where new data will not influence the model's previously learned parameters.

- **Example**:

  ```
  X_test_scaled = scaler.transform(X_test)
  ```

  - **Here**: `scaler.transform(X_test)` scales `X_test` using the mean and standard deviation calculated from `X_train`. This ensures that the test data is processed in the same way as the training data, without leaking information from the test set into the model.

3. Why This Matters: Preventing Data Leakage

- **Data Leakage**: If you use `fit_transform` on both the training and test sets, you would allow the test data to influence the scaling parameters, leading to overly optimistic performance estimates. This is known as data leakage, where information from outside the training dataset is used to create the model, leading to overfitting.

- **Generalization**: Using `fit_transform` on the training set and `transform` on the test set ensures that your model generalizes well to new, unseen data by not allowing any information from the test set to influence the training process.

Summary

- **`fit_transform(X_train)`**: Computes and applies the transformation based on the training data.
- **`transform(X_test)`**: Applies the same transformation, using the parameters computed from the training data, to the test data.

This methodology ensures that your model is trained and evaluated under conditions that mimic real-world deployment, leading to more accurate and reliable predictions.

Model Training

What is Model Training?

Model training in machine learning refers to the process of teaching an algorithm to recognize patterns or make decisions based on data. During this phase, the model learns from a dataset by adjusting its parameters so that it can make accurate predictions or decisions on new, unseen data.

Steps Involved in Model Training

1. **Data Collection**:
    - The first step is to gather the data that the model will learn from. This data is usually split into two parts: a **training set** and a **test set** (and sometimes a validation set).
    - The training set is used to train the model, while the test set is used to evaluate its performance.
2. **Data Preprocessing**:
    - Before feeding the data into the model, it's often necessary to preprocess it. This can involve:
        - **Cleaning**: Handling missing values, correcting errors.
        - **Normalization/Standardization**: Scaling the data to a standard range or distribution.
        - **Feature Engineering**: Creating new features or selecting important ones.
        - **Encoding**: Converting categorical variables into numerical ones.
3. **Choosing a Model/Algorithm**:
    - Based on the problem (classification, regression, clustering, etc.), an appropriate machine learning algorithm is selected.
    - Common algorithms include Linear Regression, Decision Trees, Neural Networks, and Support Vector Machines, among others.
4. **Model Training**:
    - **Feeding Data**: The training data is fed into the chosen algorithm.
    - **Learning Process**: The model attempts to learn the relationship between the input features (independent variables) and the target variable (dependent variable). This is done by minimizing a loss function, which measures the difference between the model's predictions and the actual values.
    - **Parameter Tuning**: During training, the model's parameters (e.g., weights in a neural network) are adjusted iteratively to reduce the loss function. This

process is typically guided by an optimization algorithm like Gradient Descent.

5. **Evaluation**:
   - After training, the model's performance is evaluated on a separate dataset (test set) that it hasn't seen before. This helps in understanding how well the model generalizes to new data.
   - Common metrics for evaluation include accuracy, precision, recall, F1-score, Mean Squared Error (MSE), etc.
6. **Model Tuning (Optional)**:
   - Sometimes, the model is further fine-tuned by adjusting hyperparameters (like learning rate, number of layers in a neural network) or by using techniques like cross-validation.
7. **Deployment**:
   - Once the model is trained and evaluated, it can be deployed to make predictions on new, unseen data in real-world applications.

Importance of Model Training

- **Learning Patterns**: Model training allows the algorithm to learn patterns from historical data, which it can then apply to make predictions or decisions on new data.
- **Optimization**: The goal of training is to optimize the model's parameters so that it makes accurate predictions with the least possible error.
- **Generalization**: Effective training ensures that the model generalizes well to unseen data, avoiding issues like overfitting (where the model learns noise in the training data) or underfitting (where the model fails to capture the underlying pattern).

Summary

Model training is a fundamental step in the machine learning pipeline where the model learns from data by adjusting its parameters to make accurate predictions. It involves feeding the training data into the algorithm, optimizing the model's parameters through a learning process, and evaluating its performance to ensure that it can generalize well to new data.

Steps Taken

- Trains a linear regression model on the normalized training data.
- Prints the coefficients (weights) of the linear regression model.
- Prints the intercept term of the linear regression model.

Model Training

In [37]:
```
from sklearn.linear_model import LinearRegression
regression = LinearRegression()
```

```
regression.fit(X_train_norm, y_train)
```

Out[37]:
```
▾ LinearRegression
LinearRegression()
```

How `regression.coef_` and Features Are Related

In linear regression, the relationship between the target variable (dependent variable) and the input features (independent variables) is modeled as a linear combination of the features. The coefficients (`regression.coef_`) represent the weights assigned to each feature in this linear combination.

1. Understanding `regression.coef_`

- **`regression.coef_`**: This attribute in a trained linear regression model stores the coefficients (or weights) for each feature in the model. These coefficients determine how much influence each feature has on the predicted outcome.
- **Linear Regression Equation**: The linear regression model can be represented by the following equation: y = beta_0 + beta_1.x_1 + beta_2.x_2 + ... + beta_n.x_n
  - Here, y is the predicted target variable.
  - x_1, x_2, ..., x_n are the input features.
  - beta_0 is the intercept (also called `regression.intercept_` in scikit-learn).
  - beta_1, beta_2, ..., beta_n are the coefficients corresponding to each feature, stored in `regression.coef_`.

2. How `regression.coef_` Relates to Features

- **Magnitude of Coefficients**:
  - The magnitude (absolute value) of each coefficient indicates the strength of the relationship between the corresponding feature and the target variable. A larger magnitude means the feature has a stronger impact on the prediction.
- **Sign of Coefficients**:
  - The sign (positive or negative) of each coefficient indicates the direction of the relationship:
    - **Positive Coefficient**: A positive coefficient means that as the feature increases, the predicted target value increases.
    - **Negative Coefficient**: A negative coefficient means that as the feature increases, the predicted target value decreases.
- **Interpretation**:
  - For example, if `regression.coef_` for a feature like `SquareFootage` is 200, this means that for every additional square foot, the predicted house

price increases by 200 units (assuming other features are held constant).

3. **Feature Importance and Selection**

- **Feature Importance**:
    - In linear regression, the absolute values of the coefficients can be used as a measure of feature importance. Features with larger coefficients contribute more to the prediction, which can help in understanding which features are most influential.
- **Feature Selection**:
    - By analyzing the coefficients, you can decide which features to keep or remove. Features with coefficients close to zero might be considered less important, although this decision should also consider other factors like multicollinearity.

4. **Example**

Let's say you have a linear regression model predicting house prices based on features like `SquareFootage`, `NumberOfBedrooms`, and `AgeOfHouse`. After training the model:

```python
from sklearn.linear_model import LinearRegression

# Example features and target
X = [[1500, 3, 10], [1700, 4, 15], [1200, 2, 5]]
y = [300000, 350000, 200000]

# Training the model
regression = LinearRegression()
regression.fit(X, y)

# Coefficients
print(regression.coef_)
```

If `regression.coef_` returns `[200, 50000, -3000]`, it means:

- For every additional square foot, the house price increases by 200 units.
- For every additional bedroom, the house price increases by 50,000 units.
- For every additional year of house age, the house price decreases by 3,000 units.

Summary

- **`regression.coef_`** contains the coefficients for each feature in a linear regression model.
- The coefficients indicate the strength and direction of the relationship between the features and the target variable.
- Understanding these coefficients helps in interpreting the model, assessing feature importance, and making decisions about feature selection.

In [38]:
```
print(regression.coef_)
```

```
[ 8.49221760e-01  1.22119309e-01 -2.99558449e-01  3.48409673e-01
 -8.84488134e-04 -4.16980388e-02 -8.93855649e-01 -8.68616688e-01]
```

What is `regression.intercept_`?

In linear regression, the **intercept** is a key component of the linear equation that describes the relationship between the input features (independent variables) and the target variable (dependent variable). The intercept is represented by the `regression.intercept_` attribute in a trained linear regression model using libraries like scikit-learn.

1. Understanding `regression.intercept_`

- **Linear Regression Equation**: The linear regression model is typically expressed as: y = beta_0 + beta_1.x_1 + beta_2.x_2 + ... + beta_n.x_n
  - Here, y is the predicted target variable.
  - x_1, x_2, ..., x_n are the input features.
  - beta_0 is the intercept (also called `regression.intercept_` in scikit-learn).
  - beta_1, beta_2, ..., beta_n are the coefficients corresponding to each feature, stored in `regression.coef_`.
- **`regression.intercept_`**:
  - It represents the **value of the target variable** when all the input features are set to zero.
  - In other words, it's the point where the regression line crosses the y-axis (when all (x_i = 0)).

2. Significance of the Intercept

- **Baseline Prediction**:
  - The intercept provides a baseline prediction for the target variable when none of the features are contributing to the model (i.e., all features are zero). It's the predicted outcome if there were no influence from any independent variables.
- **Model Interpretation**:
  - The intercept allows you to understand how much of the target variable's value can be explained by factors that are not included in the model. For example, in a model predicting house prices, the intercept might represent the base price of a house without considering size, location, or other factors.
- **Effect on Predictions**:
  - The intercept is added to the weighted sum of the features during prediction. For a simple linear model, the prediction (\hat{y}) is calculated

as:

[\hat{y} = \text{regression.intercept_} + \sum_{i=1}^{n} \text{regression.coef_}[i] \cdot x_i]

- **Context-Dependent Interpretation**:
  - The meaningfulness of the intercept can vary depending on the context. For some datasets, an intercept might not have a clear physical interpretation, especially if the concept of all features being zero is not realistic (e.g., in cases where a feature like square footage or age cannot logically be zero).

3. Example

Consider a model predicting house prices (`y`) based on features like square footage (`x1`) and number of bedrooms (`x2`):

```
from sklearn.linear_model import LinearRegression

# Example features and target
X = [[1500, 3], [1700, 4], [1200, 2]]
y = [300000, 350000, 200000]

# Training the model
regression = LinearRegression()
regression.fit(X, y)

# Intercept
print(regression.intercept_)
```

- Suppose `regression.intercept_` returns `50000`. This means that if both square footage and number of bedrooms were zero (a theoretical scenario), the model would predict a baseline house price of $50,000.

Summary

- **`regression.intercept_`** is the y-intercept in a linear regression model, representing the predicted value of the target variable when all input features are zero.
- It serves as a baseline prediction and helps in interpreting the model by showing the contribution to the target variable from factors not captured by the input features.
- While the intercept is crucial for model predictions, its interpretability depends on the context and the specific features of the dataset.

```
In [39]: print(regression.intercept_)
```

```
2.0692396089424165
```

Model Prediction

- Uses the trained model to predict house prices on the normalized testing data.

- Computes the residuals (difference between actual and predicted values).
- Displays the residuals.
- Plots the distribution of residuals to analyze the errors.
- Calculates and prints performance metrics: Mean Squared Error (MSE), Mean Absolute Error (MAE), R-squared (R2), and Root Mean Squared Error (RMSE).
- Stores the R-squared score.
- Displays the R-squared score.
- Calculates the adjusted R-squared score.
- Displays the number of features in the testing set.

In [40]:
```
reg_pred = regression.predict(X_test_norm)
reg_pred
```

Out[40]:
```
array([0.72604907, 1.76743383, 2.71092161, ..., 2.07465531, 1.57371395,
       1.82744133])
```

In [41]:
```
## Calculate the error or the residual
residuals = y_test - reg_pred
```

In [42]:
```
residuals
```

Out[42]:
```
20046   -0.249049
3024    -1.309434
15663    2.289088
20484   -0.649147
9814     0.173042
           ...
17505    0.155059
13512   -0.237516
10842    0.109345
16559   -0.379714
5786     0.270559
Name: Price, Length: 6192, dtype: float64
```

Why Residuals Should Follow a Normal Distribution

In regression analysis, residuals are the differences between the observed values and the values predicted by the model. The distribution of these residuals is a critical aspect of evaluating the assumptions and validity of the model.

1. What Are Residuals?

- **Residuals** are calculated as: [\text{Residual} = \text{Observed Value} - \text{Predicted Value}]
  - They represent the error or deviation of the predicted values from the actual values.

2. Importance of Normal Distribution of Residuals

- **Assumption of Linear Regression**:
  - One of the key assumptions of linear regression is that the residuals (errors) are normally distributed.

- This assumption is crucial for the validity of various statistical tests and confidence intervals associated with the model.

- **Why Should Residuals Be Normally Distributed?**:

  1. **Unbiased Estimates**:
     - If residuals follow a normal distribution, the estimates of the coefficients (slopes) are unbiased. This means the average of the predictions will be close to the actual value.
  2. **Valid Hypothesis Testing**:
     - Many statistical tests, like t-tests and F-tests, rely on the assumption that residuals are normally distributed. These tests help determine the significance of the model's coefficients and overall model fit. If the residuals are not normally distributed, the results of these tests may be invalid, leading to incorrect conclusions.
  3. **Confidence Intervals**:
     - The accuracy of confidence intervals for predictions and coefficients depends on the normality of residuals. If residuals deviate significantly from normality, the intervals may be misleading.
  4. **Predictive Accuracy**:
     - When residuals are normally distributed, the model is more likely to perform well on new, unseen data. This is because the errors are random and centered around zero, indicating that the model has captured the true underlying pattern without systematic bias.
  5. **Identifying Model Issues**:
     - If residuals are not normally distributed, it may indicate issues with the model, such as:
       - **Non-Linearity**: The relationship between the independent and dependent variables is not linear.
       - **Heteroscedasticity**: The variance of residuals changes with the level of the independent variable(s).
       - **Outliers**: The presence of outliers may skew the distribution of residuals.

3. How to Check for Normal Distribution of Residuals

- **Visual Inspection**:
  - **Histogram**: Plotting a histogram of the residuals can help you visually assess whether they follow a normal distribution (should look like a bell curve).
  - **Q-Q Plot**: A Quantile-Quantile plot compares the quantiles of the residuals to a normal distribution. If the points lie approximately along the diagonal line, the residuals are likely normally distributed.
- **Statistical Tests**:

  - **Shapiro-Wilk Test**: This test checks the normality of residuals. A p-value greater than a significance level (e.g., 0.05) suggests that the residuals are normally distributed.
  - **Kolmogorov-Smirnov Test**: Another test that compares the distribution of residuals to a normal distribution.

4. What if Residuals Are Not Normally Distributed?

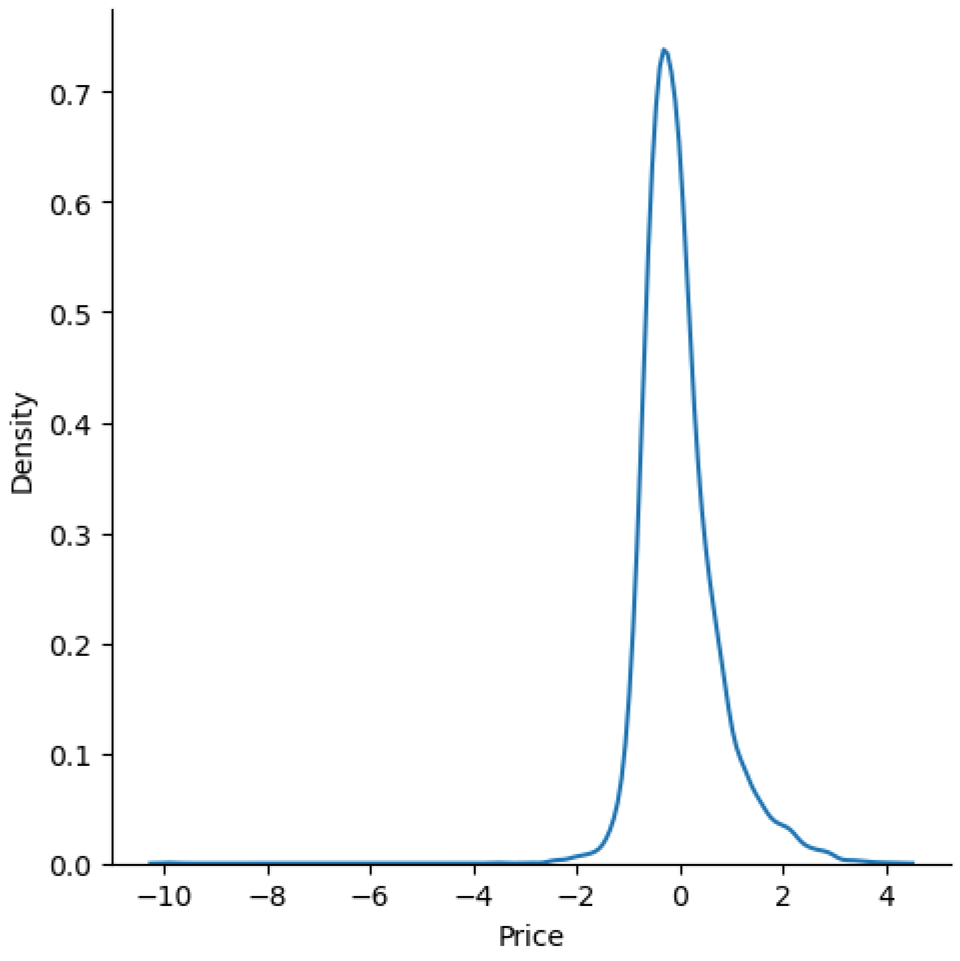
- **Transformations**: Apply transformations to the dependent variable (e.g., logarithmic, square root) to achieve normality.
- **Use Different Models**: If normality cannot be achieved, consider using a different modeling technique that does not assume normally distributed residuals, like non-parametric models.

Summary

Residuals should follow a normal distribution in regression models because this assumption underpins the validity of many statistical tests, hypothesis tests, and confidence intervals. Normal distribution of residuals ensures unbiased estimates, accurate hypothesis testing, and reliable confidence intervals, leading to better predictive performance and model validity. If residuals are not normally distributed, it can indicate issues with the model, and steps should be taken to address these issues.

In [43]:
```
## distribution plot of the residuals
sns.displot(residuals, kind="kde")
```

Out[43]:
```
<seaborn.axisgrid.FacetGrid at 0x3181860e0>
```



Model Performance

In [44]:
```
## lower error value - MSE and MAE
## higher value - r2 score and adjusted r2 score
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_s
print(mean_squared_error(y_test, reg_pred))
print(mean_absolute_error(y_test, reg_pred))
print(r2_score(y_test, reg_pred))
print(np.sqrt(mean_squared_error(y_test, reg_pred)))
```

```
0.5305677824766755
0.5272474538305956
0.5957702326061662
0.7284008391515454
```

In [45]:
```
score = r2_score(y_test, reg_pred)
```

In [46]:
```
score
```

Out[46]:
```
0.5957702326061662
```

Adjusted R-square

In [47]:
```
1 - (1-score)*(len(y_test)-1)/(len(y_test)-X_test_norm.shape[1]-1)
```

Out[47]:
```
0.5952472117200025
```

In [48]:
```
X_test_norm.shape[1]
```

```
Out[48]: 8
```

Save the model -> Pickle File

- Saves the trained model to a file using pickle.
- Loads the saved model from the file.
- Predicts the house price for the first instance in the dataset after normalizing it

```
In [50]: import pickle
         pickle.dump(regression, open('model.pkl', 'wb'))
```

Load the file and use it for future test data predictions

```
In [51]: model = pickle.load(open('model.pkl', 'rb'))
```

```
In [52]: model.predict(scaler.transform(housing.data[0].reshape(1,-1)))
```

```
/Users/vaibhavarde/Desktop/DATASCIENCE/ChaiCode/MLNotes/MLNotes/lib/python
3.10/site-packages/sklearn/base.py:493: UserWarning: X does not have valid
feature names, but StandardScaler was fitted with feature names
  warnings.warn(
```

```
Out[52]: array([4.14333441])
```

```
In [53]: model.predict(X_test_norm)
```

```
Out[53]: array([0.72604907, 1.76743383, 2.71092161, ..., 2.07465531, 1.57371395,
                1.82744133])
```