**Exploratory Data Analysis (EDA)** is a crucial step in understanding the characteristics of the dataset, identifying patterns, and gaining insights that can inform feature engineering and model selection. At its core, EDA is about asking questions of your data and seeking answers through visualizations and statistical analyses. It serves as the compass, guiding data scientists in making informed decisions about subsequent steps in the data science pipeline.

**Correlation Coefficient**

**Correlation coefficients** are essential tools in **Exploratory Data Analysis (EDA)** that quantify the strength and direction of relationships between variables in a dataset. They provide valuable insights into how changes in one variable correspond to changes in another, helping data scientists understand patterns and potential dependencies.

The **Pearson correlation coefficient**, often denoted as **r**, measures the linear relationship between two continuous variables. It ranges from **-1 to 1**, where:

- **r=1** means a perfect positive linear relationship.
- **r=−1** means a perfect negative linear relationship.
- **r=0** means no linear relationship.

Let's continue with the house price prediction project and perform an EDA using the **California Housing dataset**.

**Correlation Matrix**

A **correlation matrix** unveils the interdependence between numerical features, allowing us to identify potential patterns and relationships. The **dataset.corr()** method in pandas is used to compute the pairwise correlation of columns, excluding NA/null values. This method returns a correlation matrix, where each entry is the correlation coefficient between the columns.

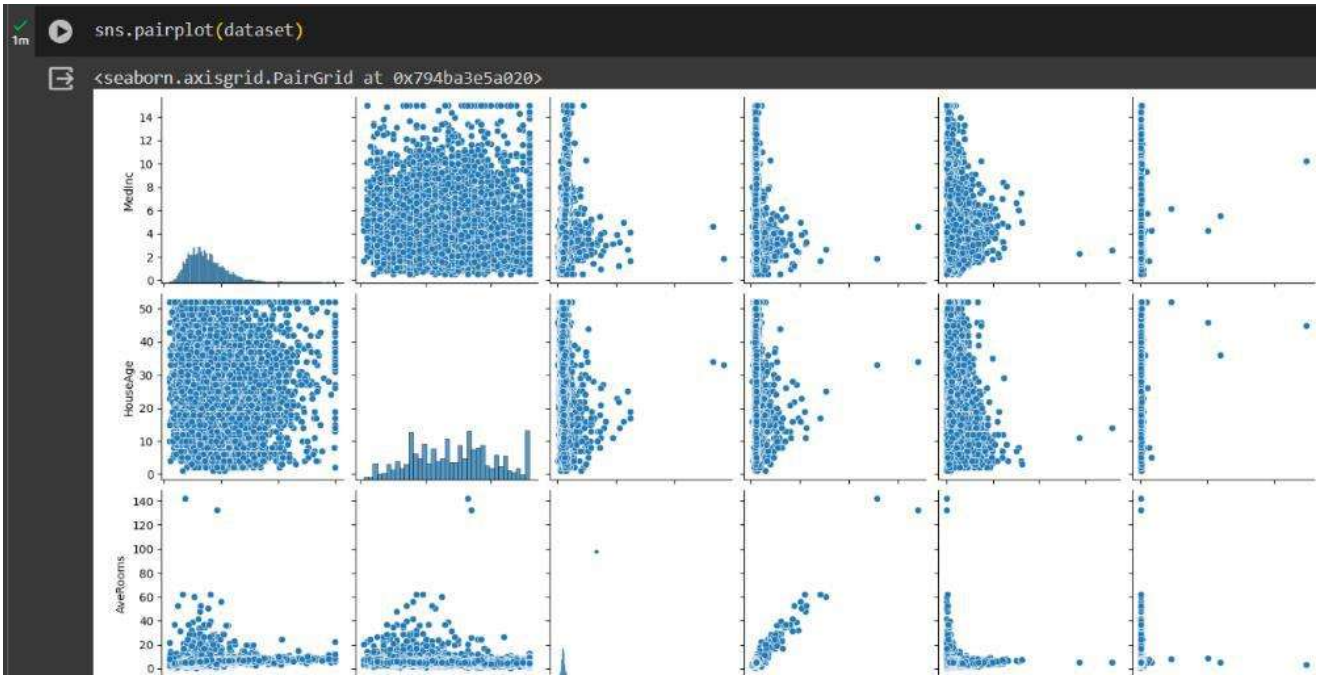
```
[17] dataset.corr()
```

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccup | Latitude | Longitude |
|---|---|---|---|---|---|---|---|---|
| **MedInc** | 1.000000 | -0.119034 | 0.326895 | -0.062040 | 0.004834 | 0.018766 | -0.079809 | -0.015176 |
| **HouseAge** | -0.119034 | 1.000000 | -0.153277 | -0.077747 | -0.296244 | 0.013191 | 0.011173 | -0.108197 |
| **AveRooms** | 0.326895 | -0.153277 | 1.000000 | 0.847621 | -0.072213 | -0.004852 | 0.106389 | -0.027540 |
| **AveBedrms** | -0.062040 | -0.077747 | 0.847621 | 1.000000 | -0.066197 | -0.006181 | 0.069721 | 0.013344 |
| **Population** | 0.004834 | -0.296244 | -0.072213 | -0.066197 | 1.000000 | 0.069863 | -0.108785 | 0.099773 |
| **AveOccup** | 0.018766 | 0.013191 | -0.004852 | -0.006181 | 0.069863 | 1.000000 | 0.002366 | 0.002476 |
| **Latitude** | -0.079809 | 0.011173 | 0.106389 | 0.069721 | -0.108785 | 0.002366 | 1.000000 | -0.924664 |
| **Longitude** | -0.015176 | -0.108197 | -0.027540 | 0.013344 | 0.099773 | 0.002476 | -0.924664 | 1.000000 |

Correlation Matrix

**Visualizing Correlation**

The **sns.pairplot()** function in the **Seaborn library** is a powerful tool for creating a grid of scatterplots for a given dataset. It displays relationships between pairs of variables, allowing for a quick visual inspection of the data. Here's how you can use it with the California Housing dataset:



Pairplot

The resulting pairplot allows you to visualize relationships between different features, spot potential patterns, and identify potential outliers.

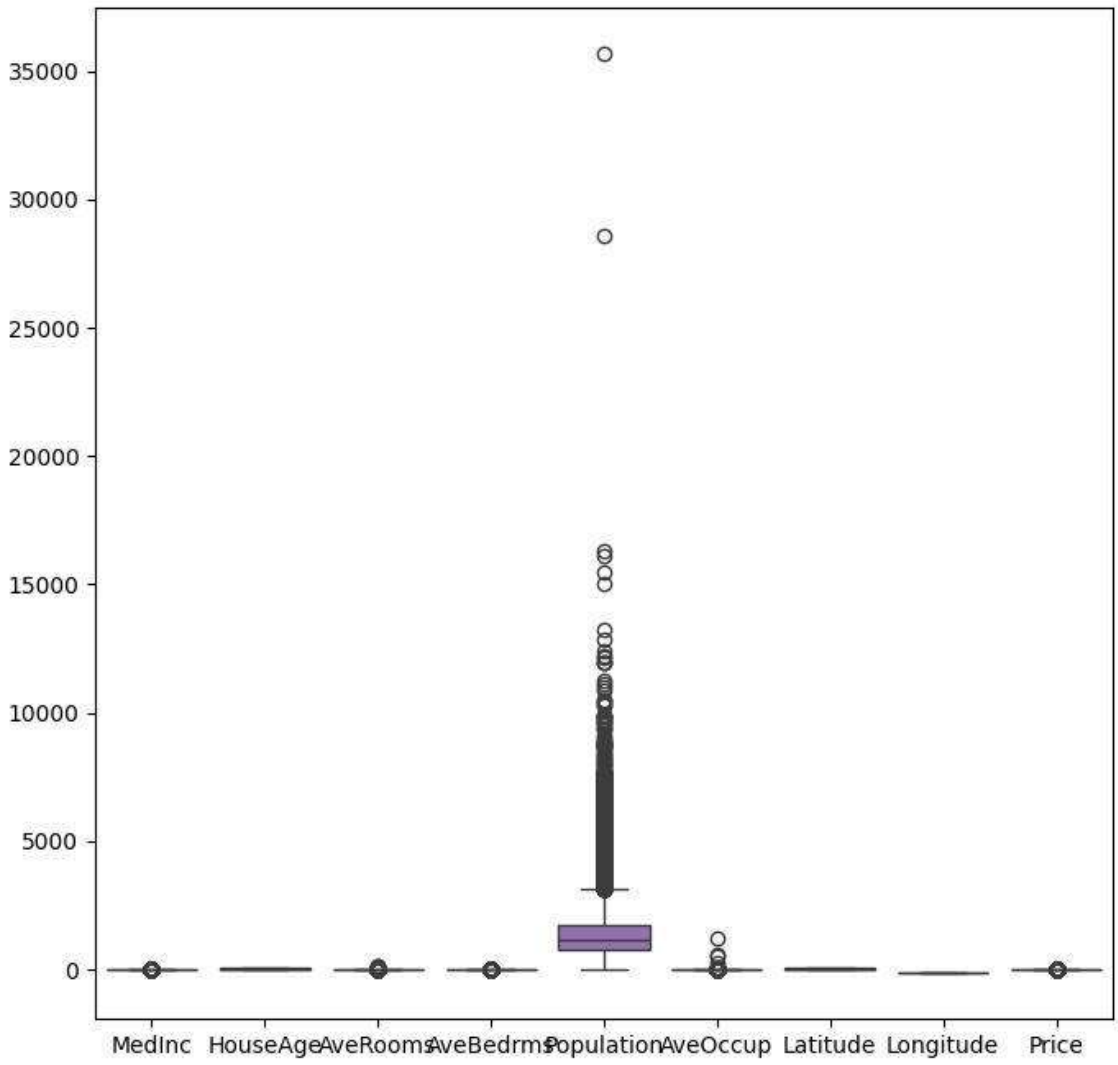
**Boxplot for Outliar Detection**

Visualizing outliers provides insights into data points that might significantly impact the model.

```
fig, ax = plt.subplots(figsize=(8,8))

sns.boxplot(data =dataset, ax=ax)

plt.savefig("boxplot.jpg")
```




Box Plot

- The box represents the interquartile range (IQR), spanning the 25th to 75th percentiles of the data.
- The line inside the box is the median.
- The "whiskers" extend from the box to the minimum and maximum values within a certain range, often determined by the interquartile range. Any data points beyond the whiskers are considered potential outliers.
- Outliers are individual points beyond the whiskers.

**Avoiding Bias**

Normalization is a crucial step in data preprocessing to avoid bias and ensure fair comparisons between features, especially when working with machine learning models. **Normalization**, also known as **feature scaling**, involves transforming the values of different features to a common scale.

**Min-Max Scaling**

Min-Max Scaling, also known as Min-Max normalization or Min-Max scaling, is a method of normalization that scales the values of a feature to a specific range, usually between 0 and 1. This method is useful when you want to ensure that all features contribute equally to the model, regardless of their original scales.

The formula for Min-Max Scaling is as follows:

msubmsubmsub$X$normalized=$X$max−$X$min$X$−$X$min

where:

- X is the original value of the feature.
- $X_{min}$ is the minimum value of the feature.
- $X_{max}$ is the maximum value of the feature.
- $X_{normalized}$ is the normalized value.

In Python, you can use the MinMaxScaler from the scikit-learn library to perform Min-Max Scaling:

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

normalized_data = scaler.fit_transform(data)

**Splitting the Data**

Splitting the data into training and test sets is a crucial step in machine learning to evaluate the model's performance on unseen data. The scikit-learn library provides a convenient function for this purpose called train_test_split. Below is an example of how you can split your data into training and test sets:

```
[16] features= dataset.iloc[: , :-1]
     target = dataset.iloc[:,-1]

     from sklearn.model_selection import train_test_split

     #Assuming 'features' is your feature matrix and 'target' is your target variable
     X_train, X_test, y_train, y_test=train_test_split(features, target, test_size=0.3,random_state=42)
```

Splitting the Dataset

- **features:** This represents your feature matrix, which includes all the independent variables.
- **target:** This is your target variable or dependent variable.

- **test_size:** This parameter specifies the proportion of the dataset to include in the test split. In this example, it's set to 0.2, meaning 20% of the data will be used for testing, and the remaining 80% for training.
- **random_state:** Setting a random seed ensures reproducibility. The same seed will result in the same split each time you run the code.

**Normalization of Given Data Points**

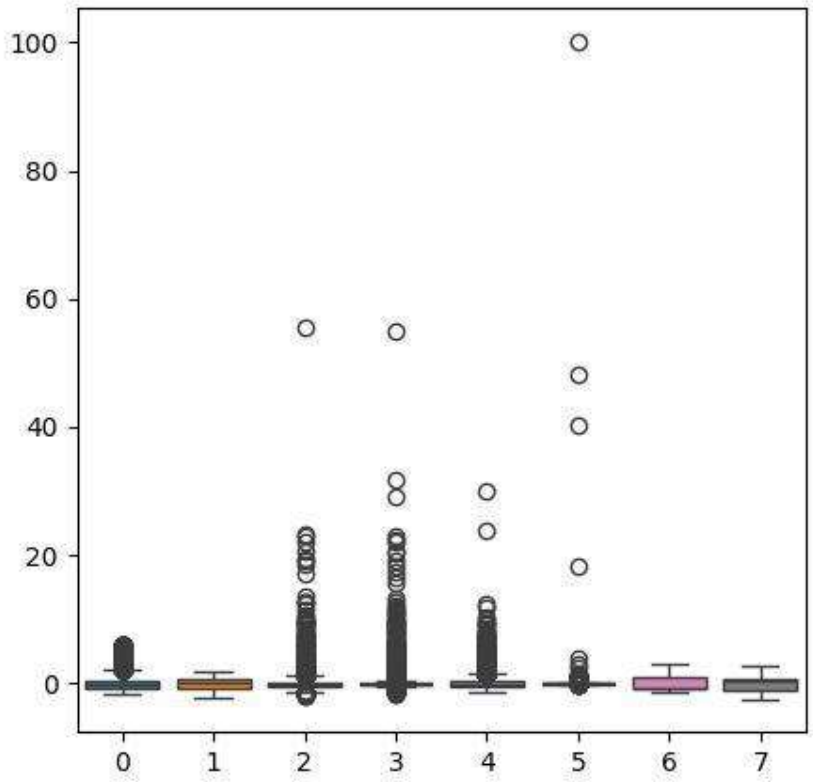
Centers the data around zero and scales it based on the standard deviation.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_train_norm = scaler.fit_transform(X_train)

Now, when you boxplot the ***standardized_data***, you will see the datapoints will come in particular range:




Boxplot after Normalization

Similarly, we can normalize the test_data, and see the boxplot:

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_test_norm = scaler.transform(X_test)

In this example:

- **scaler.fit_transform(X_train):** This fits the scaler to the training data and transforms the training data accordingly.
- **scaler.transform(X_test):** This uses the same scaler to transform the test data. The scaler applies the transformation learned from the training data to maintain consistency.

Consistent scaling between the training and test sets is crucial for the model to make accurate predictions on new, unseen data. It ensures that the scaling characteristics learned from the training set are applied uniformly to the test set.