Boston House Price Prediction

Objective

The problem at hand is to **predict the housing prices of a town or a suburb based on the features of the locality provided to us**. In the process, we need to **identify the most important features affecting the price of the house**. We need to employ techniques of data preprocessing and build a linear regression model that predicts the prices for the unseen data. We use linear regression because this is supervised learning and the target variable is a continuous variable.

Dataset

Each record in the database describes a house in Boston suburb or town. The data was drawn from the Boston Standard Metropolitan Statistical Area (SMSA) in 1970. Detailed attribute information can be found below:

Attribute Information:

- **CRIM:** Per capita crime rate by town
- **ZN:** Proportion of residential land zoned for lots over 25,000 sq.ft.
- **INDUS:** Proportion of non-retail business acres per town
- **CHAS:** Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- **NOX:** Nitric Oxide concentration (parts per 10 million)
- **RM:** The average number of rooms per dwelling
- **AGE:** Proportion of owner-occupied units built before 1940
- **DIS:** Weighted distances to five Boston employment centers
- **RAD:** Index of accessibility to radial highways
- **TAX:** Full-value property-tax rate per 10,000 dollars
- **PTRATIO:** Pupil-teacher ratio by town
- **LSTAT:** % lower status of the population
- **MEDV:** Median value of owner-occupied homes in 1000 dollars

Importing the necessary libraries

In [1]:
```
# Import libraries for data manipulation
import pandas as pd
```

```
import numpy as np

# Import libraries for data visualization
import matplotlib.pyplot as plt

import seaborn as sns



from statsmodels.graphics.gofplots import ProbPlot #Q-Q plot of the quantiles c

# Import libraries for building linear regression model
from statsmodels.formula.api import ols #ols = Ordinary Least Squares

#statsmodels is a Python module that provides classes and functions for the est
#of many different statistical models, as well as for conducting statistical te
#statistical data exploration. An extensive list of result statistics are avai
#each estimator. The results are tested against existing statistical packages t
#that they are correct.

import statsmodels.api as sm

from sklearn.linear_model import LinearRegression #Ordinary least squares Linea

# Import library for preparing data
from sklearn.model_selection import train_test_split

# Import library for data preprocessing
from sklearn.preprocessing import MinMaxScaler

import warnings
warnings.filterwarnings("ignore")
```

Loading the data

In [2]:
```
df = pd.read_csv("Boston.csv")

df.head()
```

Out[2]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	LSTAT	MEI
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	4.98	24
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14	2
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	4.03	34
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	2.94	33
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	5.33	36

Observation:

- The price of the house indicated by the variable MEDV is the target variable and the rest of the variables are independent variables based on which we will predict the house price (MEDV).

Checking data info

In [3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 13 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    int64  
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    int64  
 9   TAX      506 non-null    int64  
 10  PTRATIO  506 non-null    float64
 11  LSTAT    506 non-null    float64
 12  MEDV     506 non-null    float64
dtypes: float64(10), int64(3)
memory usage: 51.5 KB
```

Observations:

- There are a total of **506 non-null observations in each of the columns**. This indicates that there are **no missing values** in the data.
- There are **13 columns** in the dataset and **every column is of numeric data type**.

Checking unique values

In [4]:
```
df.nunique()
```

Out[4]:
```
CRIM       504
ZN          26
INDUS       76
CHAS         2
NOX         81
RM         446
AGE        356
DIS        412
RAD          9
TAX         66
PTRATIO     46
LSTAT      455
MEDV       229
dtype: int64
```

Splitting the dataset into test and train sets

Let's split the data into the dependent and independent variables and further split it into train and test set in a ratio of 70:30 for train and test sets.

In [5]:
```
#The dependent variable is sightly skewed. Hence we will apply a log transforma

#df['MEDV_log'] = np.log(df['MEDV'])
#sns.histplot(data = df, x = 'MEDV_log', kde = True)
# Separate the dependent variable and indepedent variables
#Y = df['MEDV_log']

Y = df['MEDV']
X = df.drop(columns = {'MEDV'})

# Add the intercept term
X = sm.add_constant(X)
```

In [6]:
```
# splitting the data in 70:30 ratio of train to test data
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.30, rar
```

Check the training data

In [7]:
```
X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 354 entries, 13 to 37
Data columns (total 13 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   const    354 non-null    float64
 1   CRIM     354 non-null    float64
 2   ZN       354 non-null    float64
 3   INDUS    354 non-null    float64
 4   CHAS     354 non-null    int64
 5   NOX      354 non-null    float64
 6   RM       354 non-null    float64
 7   AGE      354 non-null    float64
 8   DIS      354 non-null    float64
 9   RAD      354 non-null    int64
 10  TAX      354 non-null    int64
 11  PTRATIO  354 non-null    float64
 12  LSTAT    354 non-null    float64
dtypes: float64(10), int64(3)
memory usage: 38.7 KB
```

Observations:

- The train dataset has **354 observations and 11 columns**.
- None of the independent features have missing values.
- All the variables are numerical.

In [8]:
```
y_train.info()
```

```
<class 'pandas.core.series.Series'>
Int64Index: 354 entries, 13 to 37
Series name: MEDV
Non-Null Count  Dtype
--------------  -----
354 non-null    float64
dtypes: float64(1)
memory usage: 5.5 KB
```

Observations:

- The dependent/outcome variable (MEDV) in the train dataset has **354 observations.
- None of the values are missing.
- The is a numerical feature.
- We will predict MEDV using regression.

Exploratory Data Analysis

Now that we have an understanding of the problem we want to solve, and we have loaded the datasets, the next step to follow is to have a better understanding of the dataset, i.e., what is the distribution of the variables, what are different relationships that exist between variables, etc. If there are any data anomalies like missing values or outliers, how do we treat them to prepare the dataset for building the predictive model?

Summary Statistics of this Dataset

In [9]:
```
df.describe(include = "all").T
```

Out[9]:

	count	mean	std	min	25%	50%	75%	
CRIM	506.0	3.613524	8.601545	0.00632	0.082045	0.25651	3.677083	88.9
ZN	506.0	11.363636	23.322453	0.00000	0.000000	0.00000	12.500000	100.0
INDUS	506.0	11.136779	6.860353	0.46000	5.190000	9.69000	18.100000	27.7
CHAS	506.0	0.069170	0.253994	0.00000	0.000000	0.00000	0.000000	1.0
NOX	506.0	0.554695	0.115878	0.38500	0.449000	0.53800	0.624000	0.8
RM	506.0	6.284634	0.702617	3.56100	5.885500	6.20850	6.623500	8.7
AGE	506.0	68.574901	28.148861	2.90000	45.025000	77.50000	94.075000	100.0
DIS	506.0	3.795043	2.105710	1.12960	2.100175	3.20745	5.188425	12.1
RAD	506.0	9.549407	8.707259	1.00000	4.000000	5.00000	24.000000	24.0
TAX	506.0	408.237154	168.537116	187.00000	279.000000	330.00000	666.000000	711.0
PTRATIO	506.0	18.455534	2.164946	12.60000	17.400000	19.05000	20.200000	22.0
LSTAT	506.0	12.653063	7.141062	1.73000	6.950000	11.36000	16.955000	37.9
MEDV	506.0	22.532806	9.197104	5.00000	17.025000	21.20000	25.000000	50.0

Observations:

1. MEDV (Median value of owner-occupied homes in 1000 dollars)

    - The mean of the median value of owner-occupied homes in Boston Standard Metropolitan Statistical Area in 1970 was $22.532806K.
    - The range is ($5K, \$50K). The standard deviaiton is $9.197104K. This means that 99.73\% of all the median value of owner-occupied homes lies in the range

(\$13.335702K, $31.72991K).

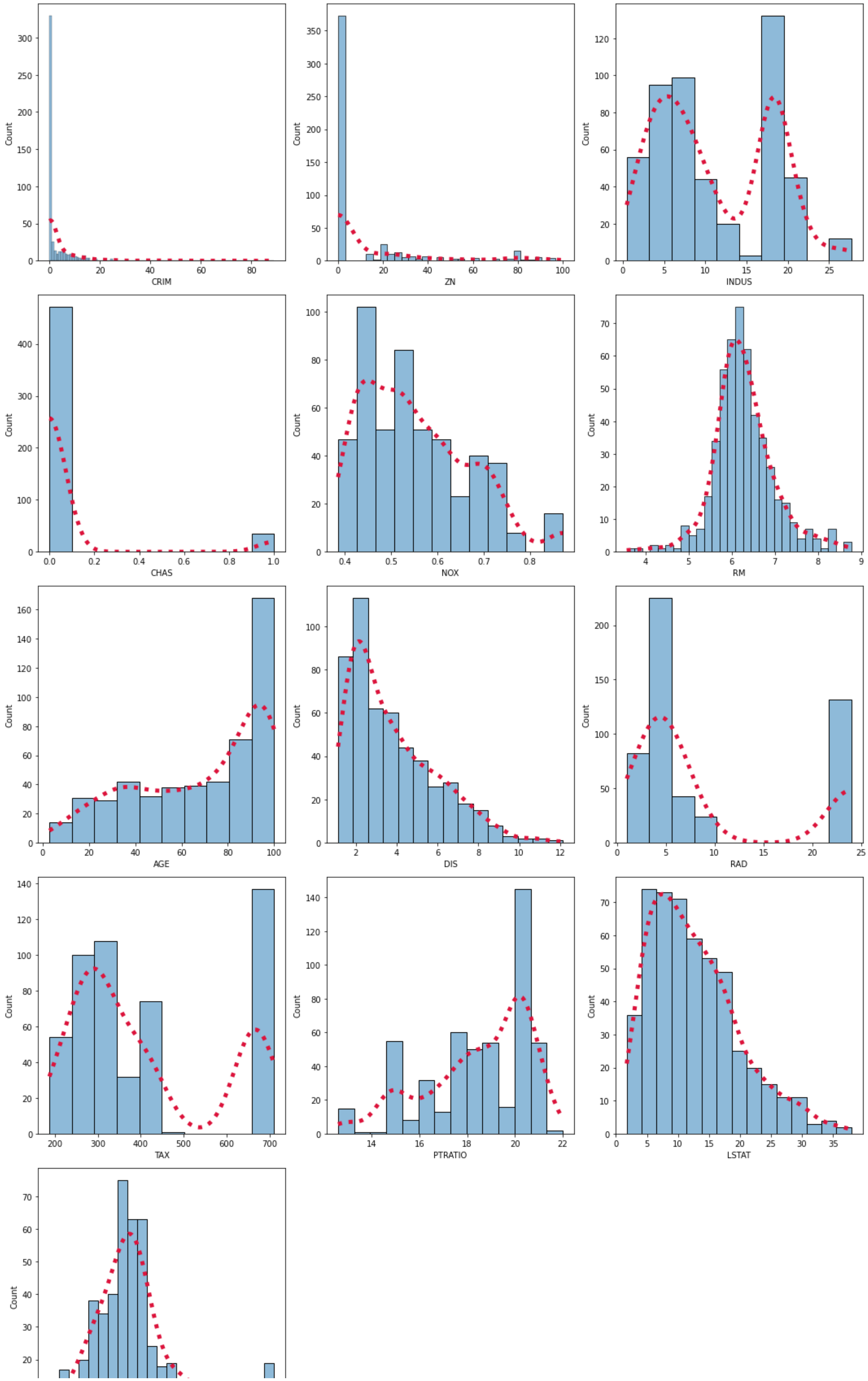
- The median corresponds to 50 percentile, i.e., 50\% of the data lies below this point. Hence, the median MEDV for this sample set is $21.20000K. 25\% of the data lie below \$17.025000K. 75\% of the data lie below $25K.

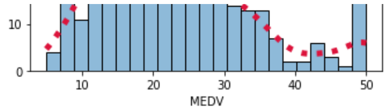
2. CRIM (Per capita crime rate by town)

- The mean of the per capita crime rate by town in Boston Standard Metropolitan Statistical Area in 1970 was \.
- The range is (,). The standard deviaiton is . This means that \% of all the per capita crime rate by town lies in the range (,).
- The median corresponds to 50 percentile, i.e., 50\% of the data lies below this point. Hence, the median CRIM for this sample set is . 25\% of the data lie below . 75\% of the data lie below .

Univariate Analysis

In [10]:
```
cols = 3
rows= 5
num_cols = df.select_dtypes(exclude='object').columns
fig = plt.figure(figsize= (cols*5, rows*5))
for i, col in enumerate(num_cols):

    ax=fig.add_subplot(rows,cols,i+1)

    sns.histplot(x = df[col], ax = ax, kde = True, line_kws={'lw': 5, 'ls': ':'
    ax.lines[0].set_color('crimson')

fig.tight_layout()
plt.show()
```


Count
CRIM
ZN
INDUS
CHAS
NOX
RM
AGE
DIS
RAD
TAX
PTRATIO
LSTAT



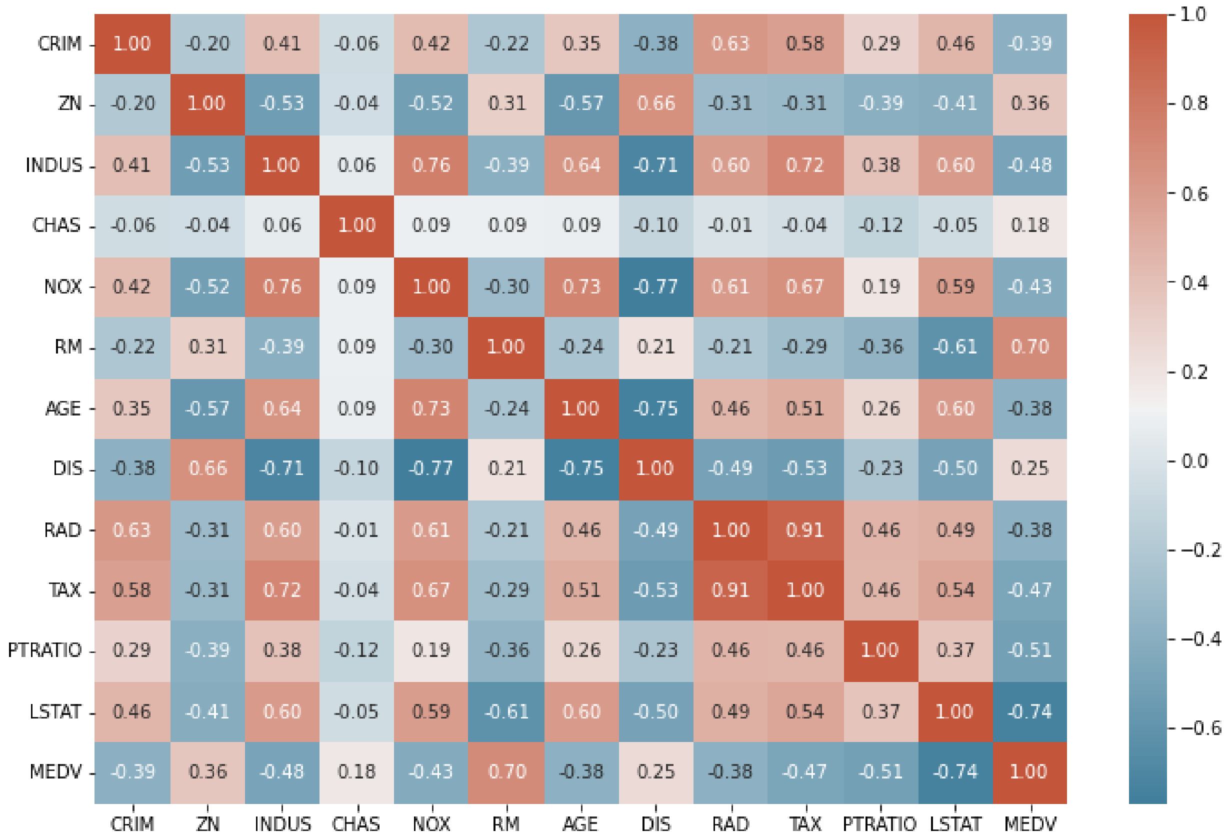
Observations:

- The feature **RM** is approx uniformly distributed.
- The featurs **CRIM**, **ZN**, **NOX**, **DIS**, **LSTAT** have a right skew.
- The features **CHAS**, **INDUS**, **RAD**, **TAX** have bimodal distributions.

Bivariate Analysis

- ***Correlation matrix based on heatmap:***

In [11]:
```
plt.figure(figsize = (12, 8))

cmap = sns.diverging_palette(230, 20, as_cmap = True)

sns.heatmap(df.corr(), annot = True, fmt = '.2f', cmap = cmap)

plt.show()
```



Observations:

Now, we will visualize the relationship between the pairs of features having significant linear correlations.

1. Strong negative correlations with MEDV:
   - More the proportion of population that is lower status, lesser is the median value of the house.
2. Strong positive correlations with MEDV:
   - Median value of the house increases with the average number of rooms per dwelling (RM)
3. Moderate negative correlation with MEDV:
   - There is a moderate negative correlation between the median value of a house and pupil-teacher ratio (PTRATIO) in the town. The regions with higher median house price have schools with lesser students assigned to a single teacher and hence ideally have a better learning experience.
   - Full-value property-tax rate (TAX) has a moderate negative correlation with the MEDV.
   - MEDV increases with decreasing Nitric Oxide concentration (NOX).
   - Proportion of non-retail business acres per town (INDUS)
4. Weak negative correlation with MEDV:
   - Index of accessibility to radial highways (RAD)
   - Proportion of owner-occupied units built before 1940 (AGE)
   - Per capita crime rate by town (CRIM)
5. Weak positive correlation with MEDV:
   - Proportion of residential land zoned for lots over 25,000 sq.ft. (ZN)
   - Weighted distances to five Boston employment centers (DIS)
6. No correlation with MEDV:
   - Charles River dummy variable (CHAS)
7. Other strong postive correlations:
   - TAX, CRIM with RAD.
   - AGE and NOX
   - DIS and ZN
   - LSTAT and INDUS correlate positively and strongly with AGE.
   - TAX, RAD and INDUS correlate positively and strongly with NOX.
   - TAX and INDUS correlate positively and strongly.

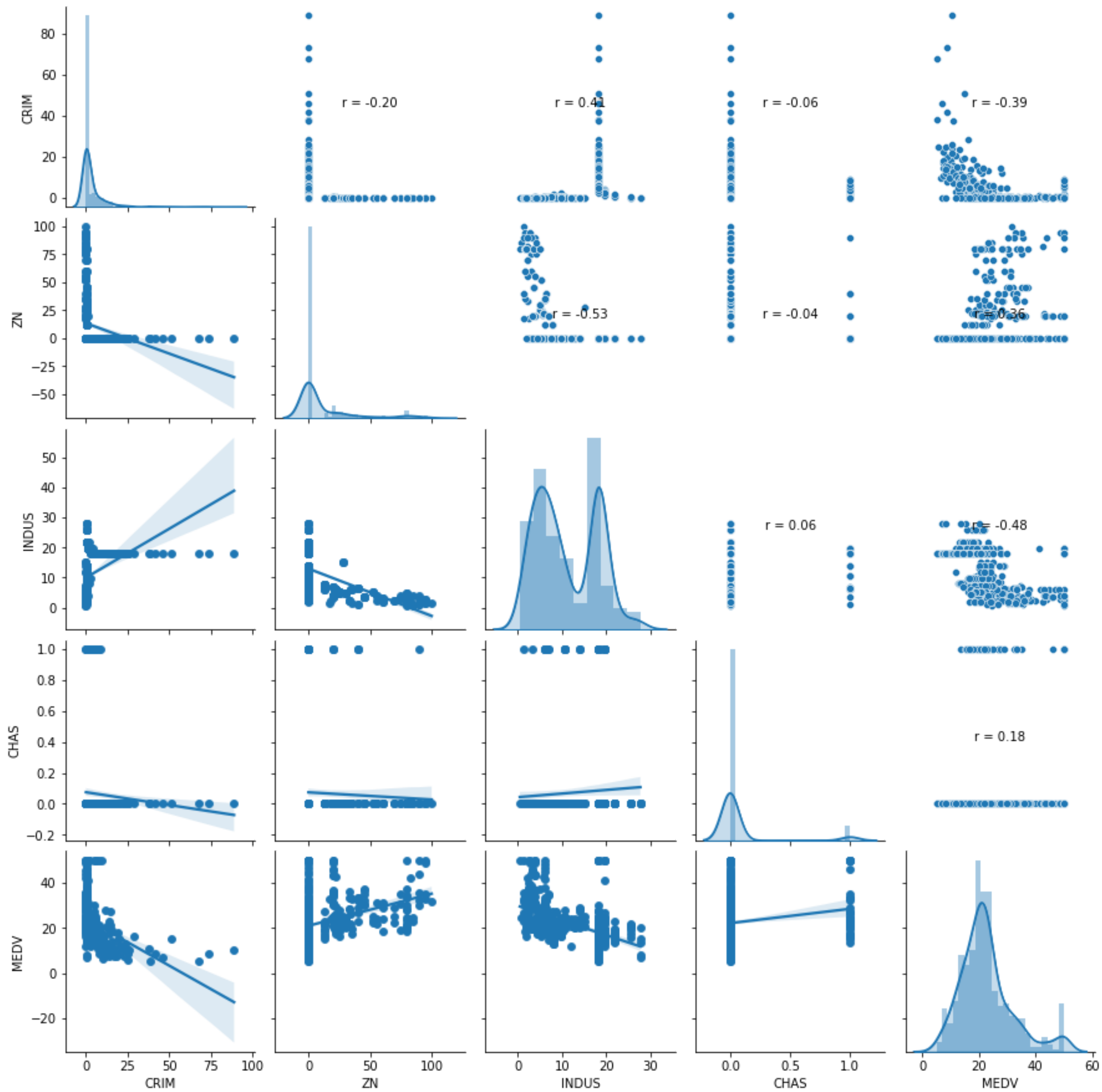
1. Other strong negative correlations:
   - DIS with INDUS, NOX and AGE
   - LSTAT with RM

- ***Could any of the strong/weak correlations be due to outliers?***

In [12]:
```
from scipy.stats import pearsonr
```

```
def reg_coef(x,y,label=None,color=None,**kwargs):
    ax = plt.gca()
    r,p = pearsonr(x,y)
    ax.annotate('r = {:.2f}'.format(r), xy=(0.5,0.5), xycoords='axes fraction',
    ax.set_axis_off()


cols_to_plot = df.columns[0:4].tolist() + ['MEDV'] # explicitly add the column
g = sns.pairplot(df[cols_to_plot],kind='scatter', diag_kind='kde')
#for i, j in zip(*np.triu_indices_from(g.axes, 1)):
 #   g.axes[i, j].set_visible(False)
g.map_diag(sns.distplot)
g.map_lower(sns.regplot)
g.map_upper(reg_coef)
```

Out[12]: <seaborn.axisgrid.PairGrid at 0x7fb00a630fd0>

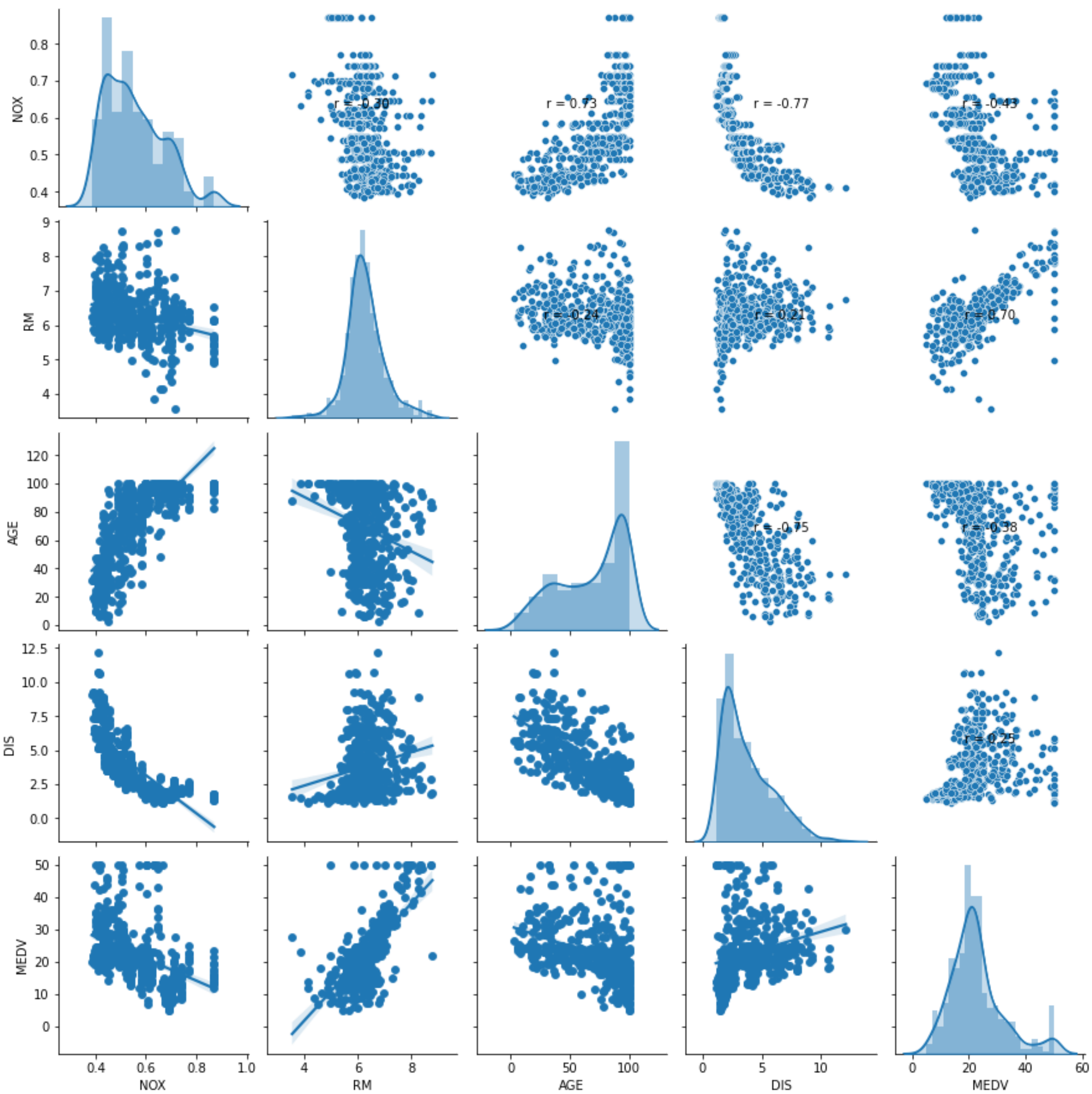


Observations:

- There isn't an appreciable correlation between the pairs of features shown above.

In [13]:
```
#pair plot
from scipy.stats import pearsonr
def reg_coef(x,y,label=None,color=None,**kwargs):
    ax = plt.gca()
    r,p = pearsonr(x,y)
    ax.annotate('r = {:.2f}'.format(r), xy=(0.5,0.5), xycoords='axes fraction',
    ax.set_axis_off()


cols_to_plot = df.columns[4:8].tolist() + ['MEDV'] # explicitly add the column
g = sns.pairplot(df[cols_to_plot],kind='scatter', diag_kind='kde')
#for i, j in zip(*np.triu_indices_from(g.axes, 1)):
 #   g.axes[i, j].set_visible(False)
g.map_diag(sns.distplot)
g.map_lower(sns.regplot)
g.map_upper(reg_coef)
```

Out[13]:
```
<seaborn.axisgrid.PairGrid at 0x7fb00a016df0>
```

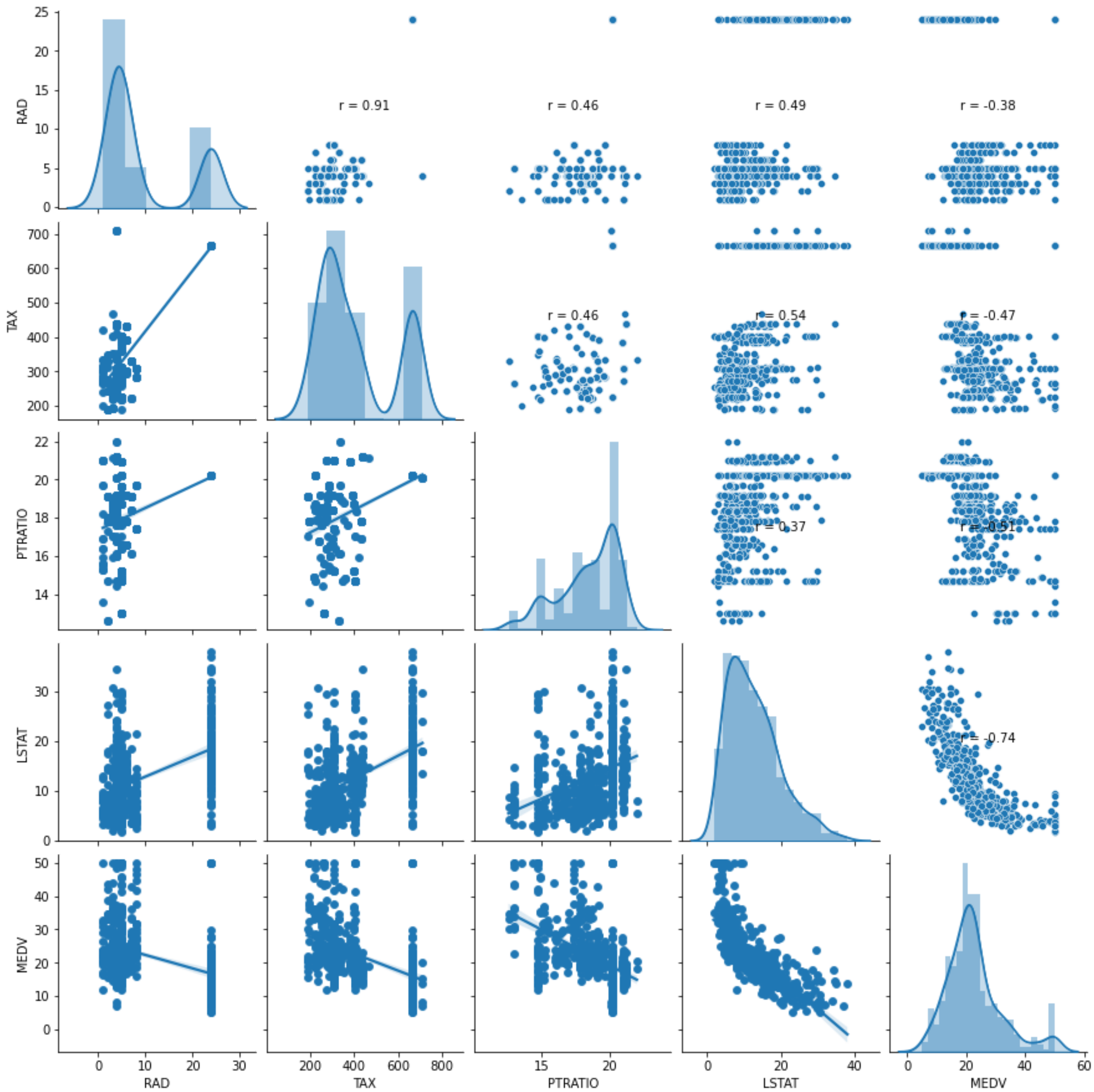


Observations:

- There seems to be an appreciable correlation between the folowing:
  - AGE and DIS
    - The distance of the houses to the Boston employment centers appears to decrease moderately as the the proportion of the old houses increase in the town. It is possible that the Boston employment centers are located in the established towns where proportion of owner-occupied units built prior to 1940 is comparatively high.
  - RM and MEDV
    - As expected, median value increases with number of rooms in the house.
    - - There are a few outliers in a horizontal line as the MEDV value seems to be capped at 50.
  - NOX and DIST
  - NOX and AGE

In [14]:
```
#pair plot
from scipy.stats import pearsonr
def reg_coef(x,y,label=None,color=None,**kwargs):
    ax = plt.gca()
    r,p = pearsonr(x,y)
    ax.annotate('r = {:.2f}'.format(r), xy=(0.5,0.5), xycoords='axes fraction',
    ax.set_axis_off()


cols_to_plot = df.columns[8:12].tolist() + ['MEDV'] # explicitly add the columr
g = sns.pairplot(df[cols_to_plot],kind='scatter', diag_kind='kde')
#for i, j in zip(*np.triu_indices_from(g.axes, 1)):
 #   g.axes[i, j].set_visible(False)
g.map_diag(sns.distplot)
g.map_lower(sns.regplot)
g.map_upper(reg_coef)
```

Out[14]:
```
<seaborn.axisgrid.PairGrid at 0x7fb00bafe430>
```



Observations:

- A strong negative correlation between proportion of population that is lower status (LSTAT) and median value of the house (MEDV) clearly stands out in the scatterplot.
- A strong positive correlation between average number of rooms per dwelling (RM) and median value of the house (MEDV) also stands out in the scatterplot.
- Strong correlation between TAX and RAD seems to arise from outliers.

Checking correlation of TAX and RAD after removing outliers:

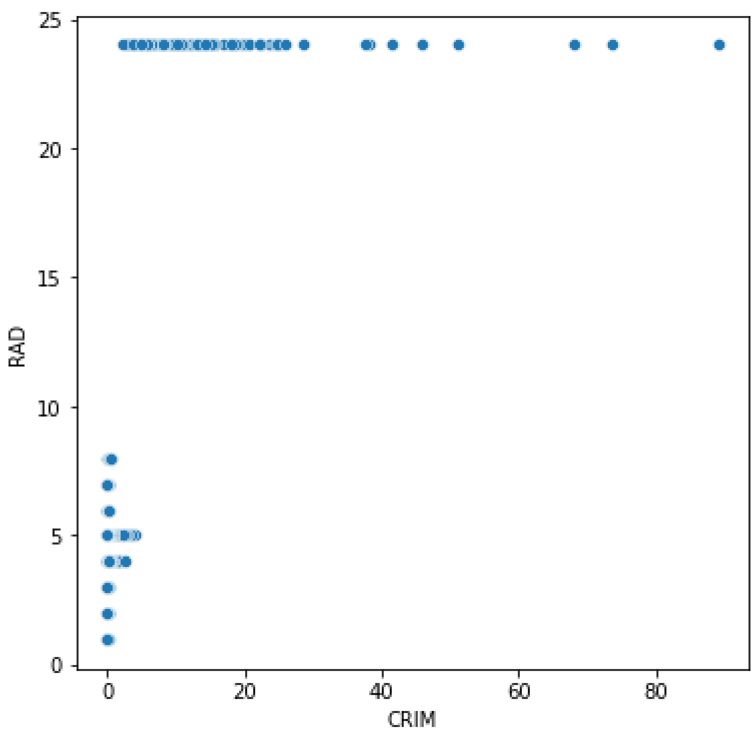
In [15]:
```
# Remove the data corresponding to high tax rate
df1 = df[df['TAX'] < 600]

# Import the required function
from scipy.stats import pearsonr
```

```
# Calculate the correlation
print('The correlation between TAX and RAD is', pearsonr(df1['TAX'], df1['RAD']
```

```
The correlation between TAX and RAD is 0.249757313314292
```

The correlation between TAX and RAD is very weak after removing the outliers. Hence, the high correlation between TAX and RAD was due to the outliers. The tax rate for some properties could be higher due to other reasons.

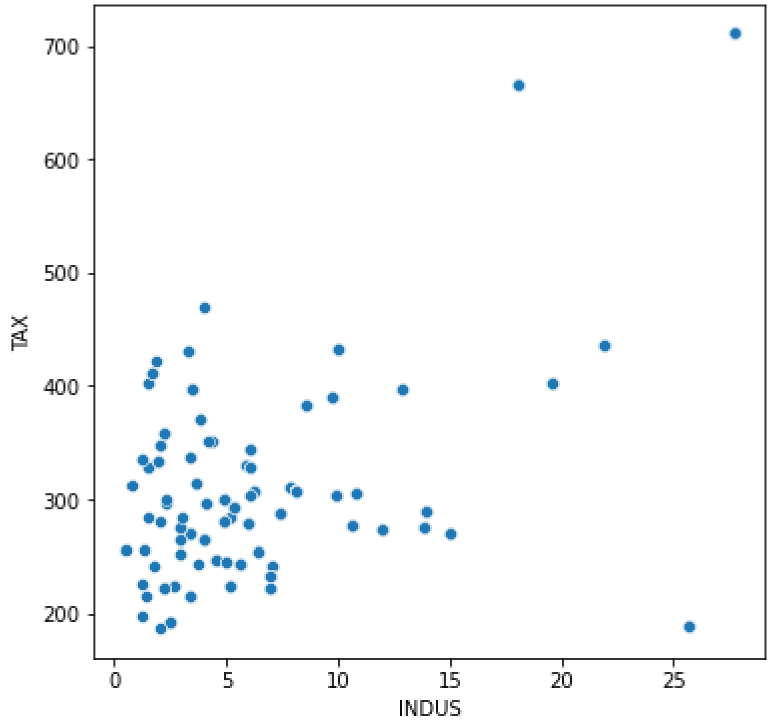
In [16]:
```
# Scatterplot to visualize the relationship between AGE and DIS
plt.figure(figsize = (6, 6))

sns.scatterplot(x = 'CRIM', y = 'RAD', data = df)


plt.show()
```



Observations:

- CRIM and RAD doesn't have a strong correlation as is clear from the scatterplot above.

In [17]:
```
# Scatterplot to visualize the relationship between INDUS and TAX
plt.figure(figsize = (6, 6))

sns.scatterplot(x = 'INDUS', y = 'TAX', data = df)

plt.show()
```



- The tax rate appears to increase with an increase in the proportion of non-retail business acres per town.

Observations:

We have seen that the variables LSTAT and RM have a linear relationship with the dependent variable MEDV. Also, there are significant relationships among few independent variables, which is not desirable for a linear regression model. Let's first split the dataset.

Regression model

In [18]:
```
# Create the model
model0 = sm.OLS(y_train, X_train)
# Fitting the Model
ols_res_0 = model0.fit()

# Get the model summary
ols_res_0.summary()
```

Out[18]:

OLS Regression Results

Dep. Variable:	MEDV	**R-squared:**	0.707
Model:	OLS	**Adj. R-squared:**	0.697
Method:	Least Squares	**F-statistic:**	68.69
Date:	Thu, 11 Aug 2022	**Prob (F-statistic):**	2.38e-83
Time:	23:28:04	**Log-Likelihood:**	-1063.0
No. Observations:	354	**AIC:**	2152.
Df Residuals:	341	**BIC:**	2202.
Df Model:	12		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	49.8852	6.107	8.168	0.000	37.872	61.898
CRIM	-0.1138	0.043	-2.647	0.009	-0.198	-0.029
ZN	0.0612	0.019	3.288	0.001	0.025	0.098
INDUS	0.0541	0.077	0.702	0.483	-0.097	0.206
CHAS	2.5175	0.983	2.560	0.011	0.583	4.452
NOX	-22.2485	4.696	-4.738	0.000	-31.485	-13.012
RM	2.6984	0.521	5.183	0.000	1.674	3.722
AGE	0.0048	0.017	0.291	0.771	-0.028	0.037
DIS	-1.5343	0.258	-5.944	0.000	-2.042	-1.027
RAD	0.2988	0.087	3.445	0.001	0.128	0.469
TAX	-0.0114	0.005	-2.302	0.022	-0.021	-0.002
PTRATIO	-0.9889	0.172	-5.762	0.000	-1.326	-0.651
LSTAT	-0.5861	0.061	-9.540	0.000	-0.707	-0.465

Omnibus:	134.560	**Durbin-Watson:**	1.847
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	545.280
Skew:	1.626	**Prob(JB):**	3.93e-119
Kurtosis:	8.137	**Cond. No.**	1.17e+04

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.17e+04. This might indicate that there are
strong multicollinearity or other numerical problems.

Observations:

- We can see that the **R-squared** for the model is **0.707**. Which means, 70.7% of the data can be explained by the model.
- Not all the variables are statistically significant to predict the outcome variable. To check which variables are statistically significant or have predictive power to predict the target variable, we need to check the **p-value** against all the independent variables.

Interpreting the Regression Results:

1. **Adj. R-squared**: It reflects the fit of the model.

    - Adjusted R-squared values range from 0 to 1, where a higher value generally indicates a better fit, assuming certain conditions are met.
    - In our case, the value for Adjusted R-squared is **0.697**.
2. **coeff**: It represents the change in the output Y due to a change of one unit in the independent variable (everything else held constant).

3. **std err**: It reflects the level of accuracy of the coefficients.
    - The lower it is, the more accurate the coefficients are.
4. **P >|t|**: It is the p-value.

    - Pr(>|t|) : For each independent feature, there is a null hypothesis and alternate hypothesis.

      Ho : Independent feature is not significant.

      Ha : Independent feature is significant.

    - The p-value of less than 0.05 is considered to be statistically significant with a confidence level of 95%.

1. **Confidence Interval**: It represents the range in which our coefficients are likely to fall (with a likelihood of 95%).

Check for Multicollinearity

- Multicollinearity occurs when predictor variables in a regression model are correlated. This correlation is a problem because predictor variables should be independent. If the correlation between independent variables is high, it can cause problems when we fit the model and interpret the results. When we have multicollinearity in the linear model, the coefficients that the model suggests are unreliable.

- There are different ways of detecting (or testing) multicollinearity. One such way is the Variation Inflation Factor.- We will use the Variance Inflation Factor (VIF), to check if there is multicollinearity in the data.
- **Variance Inflation factor**: Variance inflation factor measures the inflation in the variances of the regression parameter estimates due to collinearities that exist among the predictors. It is a measure of how much the variance of the estimated regression

coefficient βk is “inflated” by the existence of correlation among the predictor variables in the model.

- General Rule of thumb: If VIF is 1, then there is no correlation between the kth predictor and the remaining predictor variables, and hence the variance of β̂k is not inflated at all. Whereas, if VIF exceeds 5 or is close to exceeding 5, we say there is moderate VIF and if it is 10 or exceeds 10, it shows signs of high multicollinearity.

  - The algorithm runs a hypothesis testing for each feature. If the P(t) < 0.005, then the feature is significant in the prediction of MEDV. If P(t) > 0.005 then the multicollinearity must be removed.
  - VIF = 1/ (1/R^2).
  - As R^2 increases, VIF increases. Low VIF means, no correlation.
  - Features having a VIF score > 5 have very high correlation. They will be dropped/treated till all the features have a VIF score < 5.
  - Each time we drop/treat a feature, we must check the value of R^2.
  - Ideally we want to have the best R^2 (i.e., explainability of our model) with minimum number of columns (least amount of complexity in the model).

In [19]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor

# Function to check VIF
def checking_vif(train):
    vif = pd.DataFrame()
    vif["feature"] = train.columns

    # Calculating VIF for each feature
    vif["VIF"] = [
        variance_inflation_factor(train.values, i) for i in range(len(train.col
    ]
    return vif


print(checking_vif(X_train))
```

```
    feature         VIF
0     const  535.372593
1      CRIM    1.924114
2        ZN    2.743574
3     INDUS    3.999538
4      CHAS    1.076564
5       NOX    4.396157
6        RM    1.860950
7       AGE    3.150170
8       DIS    4.355469
9       RAD    8.345247
10      TAX   10.191941
11  PTRATIO    1.943409
12    LSTAT    2.861881
```

Observations:

- There are two variables with a high VIF - RAD and TAX (greater than 5).

- Let's remove TAX as it has the highest VIF values and check the multicollinearity again.

Modelling after removing multicollinearity

In [20]:
```
X_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 152 entries, 307 to 23
Data columns (total 13 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   const    152 non-null    float64
 1   CRIM     152 non-null    float64
 2   ZN       152 non-null    float64
 3   INDUS    152 non-null    float64
 4   CHAS     152 non-null    int64  
 5   NOX      152 non-null    float64
 6   RM       152 non-null    float64
 7   AGE      152 non-null    float64
 8   DIS      152 non-null    float64
 9   RAD      152 non-null    int64  
 10  TAX      152 non-null    int64  
 11  PTRATIO  152 non-null    float64
 12  LSTAT    152 non-null    float64
dtypes: float64(10), int64(3)
memory usage: 16.6 KB
```

In [21]:
```
#X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.30, ra
# Create the model after dropping TAX
X_train.head()
X_train1 = X_train.drop('TAX', axis = 1)
X_test1 = X_test.drop('TAX', axis = 1)

#X_train.head()
# Check for VIF
print(checking_vif(X_train1))
```

```
    feature         VIF
0     const  532.025529
1      CRIM    1.923159
2        ZN    2.483399
3     INDUS    3.270983
4      CHAS    1.050708
5       NOX    4.361847
6        RM    1.857918
7       AGE    3.149005
8       DIS    4.333734
9       RAD    2.942862
10  PTRATIO    1.909750
11    LSTAT    2.860251
```

Observations:

- All the independent features now have a VIF value < 5. We can assume that the multicollinearity has been removed. Now, we will create a linear regression model.

In [22]:
```
# Create the model
```

```
model1 = sm.OLS(y_train, X_train1)
# Fitting the Model
ols_res_1 = model1.fit()

# Get the model summary
ols_res_1.summary()
```

Out[22]:

OLS Regression Results

Dep. Variable:	MEDV	**R-squared:**	0.703
Model:	OLS	**Adj. R-squared:**	0.693
Method:	Least Squares	**F-statistic:**	73.53
Date:	Thu, 11 Aug 2022	**Prob (F-statistic):**	3.64e-83
Time:	23:28:04	**Log-Likelihood:**	-1065.8
No. Observations:	354	**AIC:**	2156.
Df Residuals:	342	**BIC:**	2202.
Df Model:	11		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	48.7735	6.126	7.961	0.000	36.723	60.824
CRIM	-0.1116	0.043	-2.580	0.010	-0.197	-0.027
ZN	0.0480	0.018	2.694	0.007	0.013	0.083
INDUS	-0.0216	0.070	-0.308	0.758	-0.160	0.116
CHAS	2.8684	0.978	2.934	0.004	0.946	4.791
NOX	-23.2036	4.707	-4.930	0.000	-32.461	-13.946
RM	2.7468	0.523	5.248	0.000	1.717	3.776
AGE	0.0041	0.017	0.246	0.806	-0.029	0.037
DIS	-1.4923	0.259	-5.760	0.000	-2.002	-0.983
RAD	0.1381	0.052	2.665	0.008	0.036	0.240
PTRATIO	-1.0409	0.171	-6.080	0.000	-1.378	-0.704
LSTAT	-0.5828	0.062	-9.429	0.000	-0.704	-0.461

Omnibus:	129.924	**Durbin-Watson:**	1.805
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	505.090
Skew:	1.580	**Prob(JB):**	2.09e-110
Kurtosis:	7.925	**Cond. No.**	2.09e+03

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.09e+03. This might indicate that there are
strong multicollinearity or other numerical problems.

Observation:

- The model hasn't improved.

- We will now drop the features that have P(t) > 0.05.

Dropping features with P(t) > 0.05

In [23]:
```
X_train2 = X_train1.drop(columns=['INDUS','AGE','ZN'], axis = 1)
X_train2
X_test2 = X_test1.drop(columns=['INDUS','AGE','ZN'], axis = 1)
X_test2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 152 entries, 307 to 23
Data columns (total 9 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   const    152 non-null    float64
 1   CRIM     152 non-null    float64
 2   CHAS     152 non-null    int64
 3   NOX      152 non-null    float64
 4   RM       152 non-null    float64
 5   DIS      152 non-null    float64
 6   RAD      152 non-null    int64
 7   PTRATIO  152 non-null    float64
 8   LSTAT    152 non-null    float64
dtypes: float64(7), int64(2)
memory usage: 11.9 KB
```

In [24]:
```
# Create the model
model2 = sm.OLS(y_train, X_train2)
# Fitting the Model
ols_res_2 = model2.fit()

# Get the model summary
ols_res_2.summary()
```

Out[24]:

OLS Regression Results

Dep. Variable:	MEDV	**R-squared:**	0.696
Model:	OLS	**Adj. R-squared:**	0.689
Method:	Least Squares	**F-statistic:**	98.93
Date:	Thu, 11 Aug 2022	**Prob (F-statistic):**	1.48e-84
Time:	23:28:04	**Log-Likelihood:**	-1069.5
No. Observations:	354	**AIC:**	2157.
Df Residuals:	345	**BIC:**	2192.
Df Model:	8		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	49.7954	6.136	8.116	0.000	37.727	61.863
CRIM	-0.0977	0.043	-2.262	0.024	-0.183	-0.013
CHAS	2.8594	0.983	2.908	0.004	0.926	4.793
NOX	-23.8071	4.260	-5.589	0.000	-32.186	-15.429
RM	2.9636	0.511	5.800	0.000	1.959	3.969
DIS	-1.1364	0.201	-5.647	0.000	-1.532	-0.741
RAD	0.1527	0.051	2.980	0.003	0.052	0.253
PTRATIO	-1.2101	0.157	-7.694	0.000	-1.519	-0.901
LSTAT	-0.5745	0.057	-10.006	0.000	-0.687	-0.462

Omnibus:	133.454	**Durbin-Watson:**	1.822
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	535.588
Skew:	1.615	**Prob(JB):**	4.99e-117
Kurtosis:	8.087	**Cond. No.**	690.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Observations:

- The model hasn't improved even after dropping the insignificant features.
- Since the median value is slightly left skewed, perhaps taking a log of the dependent value might help.

Checking for the assumptions and rebuilding the model

In this step, we will check whether the below assumptions hold true or not for the model. In case there is an issue, we will rebuild the model after fixing those issues.

1. **Mean of residuals should be 0**
2. **No Heteroscedasticity**
3. **Linearity of variables**
4. **Normality of error terms**

Mean of residuals should be 0 and normality of error terms

In [25]:
```
# Residuals
residual2 = ols_res_2.resid
residual2.mean()
```

Out[25]:
```
-1.3668915340469724e-14
```

Observation:

- The mean of residuals is very close to 0. Hence, the corresponding assumption is satisfied.

Tests for Normality

What is the test?

- Error terms/Residuals should be normally distributed.
- If the error terms are non-normally distributed, confidence intervals may become too wide or narrow. Once the confidence interval becomes unstable, it leads to difficulty in estimating coefficients based on the minimization of least squares.

What does non-normality indicate?

- It suggests that there are a few unusual data points that must be studied closely to make a better model.

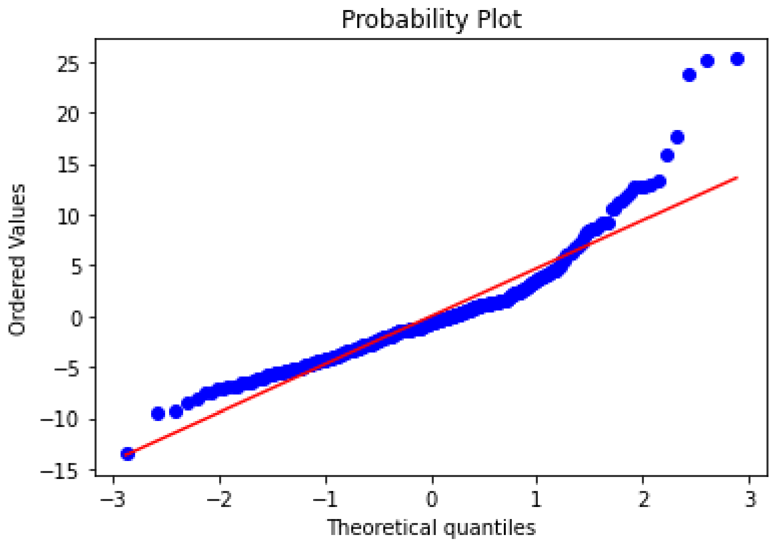
How to check the normality?

- We can plot the histogram of residuals and check the distribution visually.
- It can be checked via QQ Plot. Residuals following normal distribution will make a straight line plot otherwise not.
- Another test to check for normality: The Shapiro-Wilk test.

What if the residuals are not-normal?

- We can apply transformations like log, exponential, arcsinh, etc. as per our data.

In [26]:
```
# Plot histogram of residuals
#sns.histplot(residual2, kde = True)
import pylab

import scipy.stats as stats

stats.probplot(residual2, dist = "norm", plot = pylab)

plt.show()
```



Observations:

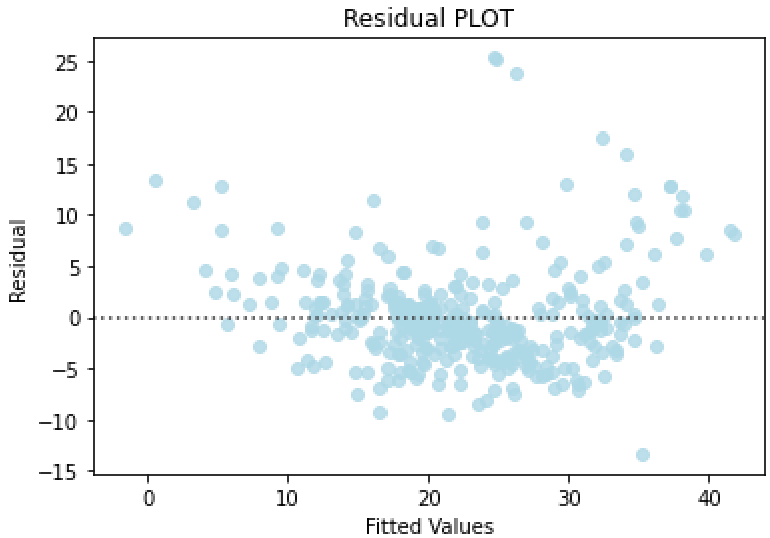
- We can see that the error terms are right skewed. The assumption of normality is **not** satisfied.
- We will address this issue by taking a log of the target variable.

Linearity of Variables

It states that the predictor variables must have a linear relation with the dependent variable.

To test this assumption, we'll plot the residuals and the fitted values and ensure that residuals do not form a strong pattern. They should be randomly and uniformly scattered on the x-axis.

In [27]:
```
# Predicted values
fitted = ols_res_2.fittedvalues

sns.residplot(x = fitted, y = residual2, color = "lightblue")

plt.xlabel("Fitted Values")

plt.ylabel("Residual")

plt.title("Residual PLOT")

plt.show()
```



Observation:

- The residuals are approximately randomly distributed.

Taking log of the target variable:

In [28]:
```
y_train = np.log(y_train)
y_test = np.log(y_test)
#sns.histplot(data = y_train, x = 'MEDV_log', kde = True)
y_train.head()
```

Out[28]:
```
13     3.015535
61     2.772589
377    2.587764
39     3.427515
365    3.314186
Name: MEDV, dtype: float64
```

In [29]:
```
X_train2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 354 entries, 13 to 37
Data columns (total 9 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   const    354 non-null    float64
 1   CRIM     354 non-null    float64
 2   CHAS     354 non-null    int64  
 3   NOX      354 non-null    float64
 4   RM       354 non-null    float64
 5   DIS      354 non-null    float64
 6   RAD      354 non-null    int64  
 7   PTRATIO  354 non-null    float64
 8   LSTAT    354 non-null    float64
dtypes: float64(7), int64(2)
memory usage: 27.7 KB
```

Model Summary:

In [30]:
```
# Create the model
model3 = sm.OLS(y_train, X_train2)
# Fitting the Model
ols_res_3 = model3.fit()

# Get the model summary
ols_res_3.summary()
```

Out[30]:

OLS Regression Results

Dep. Variable:	MEDV	**R-squared:**	0.767
Model:	OLS	**Adj. R-squared:**	0.762
Method:	Least Squares	**F-statistic:**	142.1
Date:	Thu, 11 Aug 2022	**Prob (F-statistic):**	2.61e-104
Time:	23:28:04	**Log-Likelihood:**	75.486
No. Observations:	354	**AIC:**	-133.0
Df Residuals:	345	**BIC:**	-98.15
Df Model:	8		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	4.6494	0.242	19.242	0.000	4.174	5.125
CRIM	-0.0125	0.002	-7.349	0.000	-0.016	-0.009
CHAS	0.1198	0.039	3.093	0.002	0.044	0.196
NOX	-1.0562	0.168	-6.296	0.000	-1.386	-0.726
RM	0.0589	0.020	2.928	0.004	0.019	0.098
DIS	-0.0441	0.008	-5.561	0.000	-0.060	-0.028
RAD	0.0078	0.002	3.890	0.000	0.004	0.012
PTRATIO	-0.0485	0.006	-7.832	0.000	-0.061	-0.036
LSTAT	-0.0293	0.002	-12.949	0.000	-0.034	-0.025

Omnibus:	32.514	**Durbin-Watson:**	1.925
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	87.354
Skew:	0.408	**Prob(JB):**	1.07e-19
Kurtosis:	5.293	**Cond. No.**	690.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

**Observations:

- Taking a log of the median value of the house in a locality has significantly improved the model.
- The explainability of the model has improved. R-squared is 0.768, whereas, adjusted R-squared is 0.762
- Hence, we have a better model when we assume a linear relationship between the log(Target variable) and the independent variable.

Checking for the assumptions for our model:

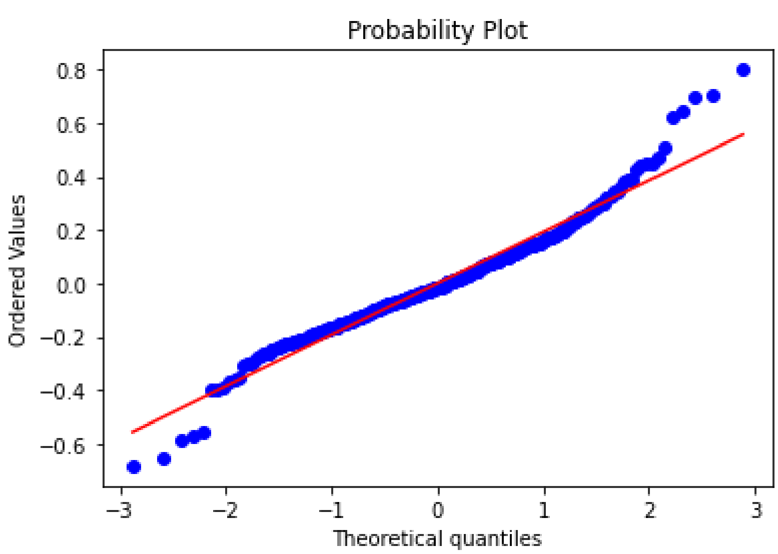
Mean of residuals:

In [31]:
```
# Residuals
residual3 = ols_res_3.resid
residual3.mean()
```

Out[31]:
```
-1.549921521948453e-15
```

Tests for Normality:

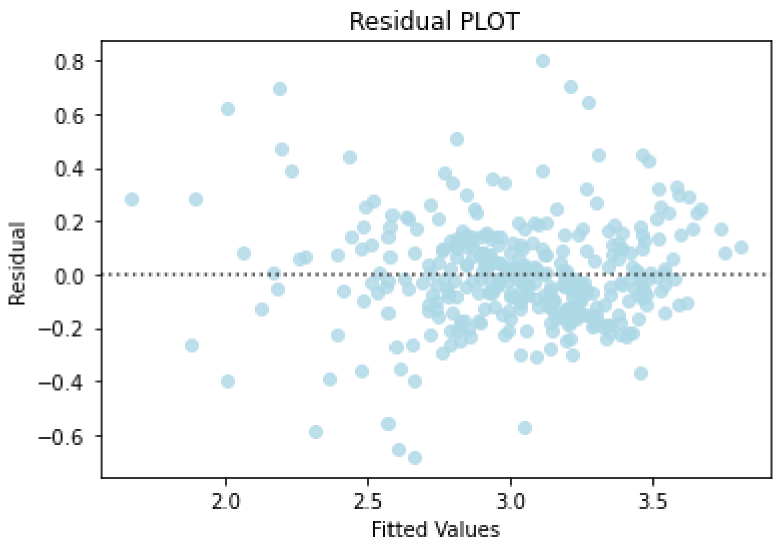
In [32]:
```
# Plot histogram of residuals
#sns.histplot(residual, kde = True)
import pylab

import scipy.stats as stats
stats.probplot(residual3, dist = "norm", plot = pylab)
plt.show()
```



Linearity of Variables

In [33]:
```
# Predicted values
fitted3 = ols_res_3.fittedvalues

sns.residplot(x = fitted3, y = residual3, color = "lightblue")
```

```
plt.xlabel("Fitted Values")

plt.ylabel("Residual")

plt.title("Residual PLOT")

plt.show()
```



Observations:

- The residual is very close to zero: Condition satisfied.
- Test of normality: Condition satisfied.
- Residual plot: Condition satisfied.

Heteroscedasticity

Test for Homoscedasticity

- **Homoscedasticity -** If the variance of the residuals are symmetrically distributed across the regression line, then the data is said to homoscedastic.
- **Heteroscedasticity -** If the variance is unequal for the residuals across the regression line, then the data is said to be heteroscedastic. In this case, the residuals can form an arrow shape or any other non symmetrical shape.
- We will use Goldfeld–Quandt test to check homoscedasticity.
  - Null hypothesis : Residuals are homoscedastic
  - Alternate hypothesis : Residuals are hetroscedastic

In [34]:
```
from statsmodels.stats.diagnostic import het_white

from statsmodels.compat import lzip

import statsmodels.stats.api as sms
```

In [35]:
```
name = ["F statistic", "p-value"]

test = sms.het_goldfeldquandt(y_train, X_train2)

lzip(name, test)
```

Out[35]:
```
[('F statistic', 1.083508292342528), ('p-value', 0.3019012006766869)]
```

Observation:

- As we observe from the above test, the p-value is greater than 0.05, so we fail to reject the null-hypothesis. That means the residuals are homoscedastic.

We have verified all the assumptions of the linear regression model. The final equation of the model is as follows:

$\log(\textbf{ MEDV }) = 4.6294 - 0.0128* \textbf{ CRIM}+0.001* \textbf{ ZN } +0.1202* \textbf{ CHAS}-1.0489* \textbf{ NOX } +0.0552*\textbf{RM}* - 0.0514*\textbf{DIS}+0.0075*\textbf{RAD}-0.0452*\textbf{PTRATIO}-0.0294*\textbf{LSTAT}$

In [36]:
```
#coef = model3.coef

#pd.DataFrame({'Feature' : coef.________, 'Coefs' : coef._______})
```

In [37]:
```
#Equation = "log (Price) = "

#print(Equation, end = '\t')

#for i in range(len(coef)):
 #   print('(', coef[i], ') * ', coef.index[i], '+', end = ' ')
```

Check the performance of the model on the train and test data set

In [38]:
```
from sklearn.metrics import r2_score, mean_absolute_percentage_error, mean_absc

# Model Performance on test and train data
def model_pref(olsmodel, x_train, x_test):

    # Insample Prediction
    y_pred_train = olsmodel.predict(x_train)
    y_observed_train = y_train

    # Prediction on test data
    y_pred_test = olsmodel.predict(x_test)
    y_observed_test = y_test

    print(
        pd.DataFrame(
            {
                "Data": ["Train", "Test"],
                "RMSE": [
                    np.sqrt(mean_squared_error(y_pred_train, y_observed_train))
                    np.sqrt(mean_squared_error(y_pred_test, y_observed_test)),
                ],
```

```
                "MAE": [
                    mean_absolute_error(y_pred_train, y_observed_train),
                    mean_absolute_error(y_pred_test, y_observed_test),
                ],

                "r2": [
                    r2_score(y_pred_train, y_observed_train),
                    r2_score(y_pred_test, y_observed_test),
                ],
            }
        )
    )
```

In [39]:

```
# Create the model
model3 = sm.OLS(y_train, X_train2).fit()

# Get the model summary
model3.summary()
```

Out[39]:

OLS Regression Results

Dep. Variable:	MEDV	**R-squared:**	0.767
Model:	OLS	**Adj. R-squared:**	0.762
Method:	Least Squares	**F-statistic:**	142.1
Date:	Thu, 11 Aug 2022	**Prob (F-statistic):**	2.61e-104
Time:	23:28:21	**Log-Likelihood:**	75.486
No. Observations:	354	**AIC:**	-133.0
Df Residuals:	345	**BIC:**	-98.15
Df Model:	8		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	4.6494	0.242	19.242	0.000	4.174	5.125
CRIM	-0.0125	0.002	-7.349	0.000	-0.016	-0.009
CHAS	0.1198	0.039	3.093	0.002	0.044	0.196
NOX	-1.0562	0.168	-6.296	0.000	-1.386	-0.726
RM	0.0589	0.020	2.928	0.004	0.019	0.098
DIS	-0.0441	0.008	-5.561	0.000	-0.060	-0.028
RAD	0.0078	0.002	3.890	0.000	0.004	0.012
PTRATIO	-0.0485	0.006	-7.832	0.000	-0.061	-0.036
LSTAT	-0.0293	0.002	-12.949	0.000	-0.034	-0.025

Omnibus:	32.514	**Durbin-Watson:**	1.925
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	87.354
Skew:	0.408	**Prob(JB):**	1.07e-19
Kurtosis:	5.293	**Cond. No.**	690.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [40]:

```
# Checking model1 performance
model_pref(model3, X_train2, X_test2)
```

```
    Data      RMSE       MAE        r2
0  Train  0.195504  0.143686  0.696514
1   Test  0.198045  0.151284  0.647196
```

In [41]:

```
# RMSE
def rmse(predictions, targets):
    return np.sqrt(((targets - predictions) ** 2).mean())
```

```
# MAPE
def mape(predictions, targets):
    return np.mean(np.abs((targets - predictions)) / targets) * 100


# MAE
def mae(predictions, targets):
    return np.mean(np.abs((targets - predictions)))


# Model Performance on test and train data
def model_pref(olsmodel, x_train, x_test):

    # In-sample Prediction
    y_pred_train = olsmodel.predict(x_train)
    y_observed_train = y_train

    # Prediction on test data
    y_pred_test = olsmodel.predict(x_test)
    y_observed_test = y_test

    print(
        pd.DataFrame(
            {
                "Data": ["Train", "Test"],
                "RMSE": [
                    rmse(y_pred_train, y_observed_train),
                    rmse(y_pred_test, y_observed_test),
                ],
                "MAE": [
                    mae(y_pred_train, y_observed_train),
                    mae(y_pred_test, y_observed_test),
                ],
                "MAPE": [
                    mape(y_pred_train, y_observed_train),
                    mape(y_pred_test, y_observed_test),
                ],
            }
        )
    )

model3 = sm.OLS(y_train, X_train2).fit()
# Checking model performance
model_pref(model3, X_train2, X_test2)
```

```
    Data      RMSE       MAE      MAPE
0  Train  0.195504  0.143686  4.981813
1   Test  0.198045  0.151284  5.257965
```

Observation:

- The R-Squared on the cross-validation is 0.198045 which is almost similar to the R-Squared on the training dataset.
- The MAE on cross-validation is 0.151284 which is almost similar to the MAE on the training dataset.

Conclusions and Recommendation

- We performed EDA, univariate and bivariate analysis, on all the variables in the dataset.
- We started the model building process with all the features.
- We removed multicollinearity from the data and analyzed the model summary report to drop insignificant features.
- We checked for different assumptions of linear regression and fixed the model iteratively if any assumptions did not hold true.
- Finally, we evaluated the model using different evaluation metrics.
- The linear model that we have developed is not capable of capturing non-linear patterns in the data. We may want to build more advanced regression model which can capture the non-linearities in the data and improve this model further.