Step 1: Reading and understanding the data

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_boston
```

In [2]:

```
boston = load_boston()
```

In [3]:

```
boston.keys()
```

Out[3]:

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])
```

- Print column names

In [4]:

```
print(boston.feature_names)
```

```
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

In [5]:

```
bos_data = pd.DataFrame(data = boston.data , columns=boston.feature_names )
```

- Now,
  - we have a pandas DataFrame called bos_data containing all the data we want to use to predict Boston Housing prices.
  - Let's create a variable called **PRICE** which will contain the prices. This information is contained in the **target data**.

In [6]:

```
bos_data['PRICE'] = boston.target
bos_data.head()
```

Out[6]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LST.
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.

Step 2: `Data Description`

- A story of what data is all about and the features present in the data

Data Set Characteristics:

- **Number of Instances:** 506
- **Number of Attributes:** 13 numeric/categorical predictive
- **Median Value** (attribute 14) is usually the target
    - `Attribute Information (in order) :`
        - **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 oxides concentration (parts per 10 million)
        - **RM** : average number of rooms per dwelling
        - **AGE** : proportion of owner-occupied units built prior to 1940
        - **DIS** : weighted distances to five Boston employment centres
        - **RAD** : index of accessibility to radial highways
        - **PTRATIO** : pupil-teacher ratio by town
        - **B** : 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - **LSTAT** : % lower status of the population
        - **MEDV** : Median value of owner-occupied homes in $1000's
        - Missing Attribute Values: None

In [7]:

```
bos_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 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    float64
 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    float64
 9   TAX      506 non-null    float64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  PRICE    506 non-null    float64
dtypes: float64(14)
memory usage: 55.5 KB
```

In [8]:

```
bos_data.shape
```

Out[8]:

```
(506, 14)
```

In [9]:

```
bos_data.isnull().sum()
```

Out[9]:

```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
PRICE      0
dtype: int64
```

- Data does not contain any null value
- so, we good to go with next steps.

Step 3: `Performing both Statistical and Graphical Data Analysis`

`Implementation` : Calculate Statistics

- We will calculate descriptive statistics about the Boston housing prices. Since numpy has already been imported, we use this library for perform the necessary calculations. - - These statistics will be extremely important later on to analyze various prediction results from the constructed model.
- In the code cell below, we will need to implement the following:
- Calculate the **minimum, maximum, mean, median, and standard deviation of 'MEDV', which is stored in `PRICE`**. Store each calculation in their respective variable.

In [10]:

```
min_price = np.min(bos_data['PRICE'])
max_price = np.max(bos_data['PRICE'])
mean_price = np.mean(bos_data['PRICE'])
median_price = np.median(bos_data['PRICE'])
std_price = np.std(bos_data['PRICE'])

# Show the calculated statistics
print("Statistics for Boston housing dataset: ")
print("Minimum price    : ${:,.2f}".format(min_price))
print("Maximum price    : ${:,.2f}".format(max_price))
print("Mean price       : ${:,.2f}".format(mean_price))
print("Median price     : ${:,.2f}".format(median_price))
print("Std. Deviation   : ${:,.2f}".format(std_price))
```

```
Statistics for Boston housing dataset:
Minimum price   : $5.00
Maximum price   : $50.00
Mean price      : $22.53
Median price    : $21.20
Std. Deviation  : $9.19
```

`Pearson's Coeffecient` 'r' can be used to justify the above correlations:

- r > 0 => Positive Correlation
- r = 0 => No Correlation
- r < 0 => Negative Correlation

In [11]:

```
bos_data.corr()
```

Out[11]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS
CRIM	1.000000	-0.200469	0.406583	-0.055892	0.420972	-0.219247	0.352734	-0.379670
ZN	-0.200469	1.000000	-0.533828	-0.042697	-0.516604	0.311991	-0.569537	0.664408
INDUS	0.406583	-0.533828	1.000000	0.062938	0.763651	-0.391676	0.644779	-0.708027
CHAS	-0.055892	-0.042697	0.062938	1.000000	0.091203	0.091251	0.086518	-0.099176
NOX	0.420972	-0.516604	0.763651	0.091203	1.000000	-0.302188	0.731470	-0.769230
RM	-0.219247	0.311991	-0.391676	0.091251	-0.302188	1.000000	-0.240265	0.205246
AGE	0.352734	-0.569537	0.644779	0.086518	0.731470	-0.240265	1.000000	-0.747881
DIS	-0.379670	0.664408	-0.708027	-0.099176	-0.769230	0.205246	-0.747881	1.000000
RAD	0.625505	-0.311948	0.595129	-0.007368	0.611441	-0.209847	0.456022	-0.494588
TAX	0.582764	-0.314563	0.720760	-0.035587	0.668023	-0.292048	0.506456	-0.534432
PTRATIO	0.289946	-0.391679	0.383248	-0.121515	0.188933	-0.355501	0.261515	-0.232471
B	-0.385064	0.175520	-0.356977	0.048788	-0.380051	0.128069	-0.273534	0.291512
LSTAT	0.455621	-0.412995	0.603800	-0.053929	0.590879	-0.613808	0.602339	-0.496996
PRICE	-0.388305	0.360445	-0.483725	0.175260	-0.427321	0.695360	-0.376955	0.249929

`Visualising the Data`

- EDA and Summary Statistics
  - Let's explore this data set. First we use `describe()` to get basic **summary statistics for each of the columns**.

In [12]:

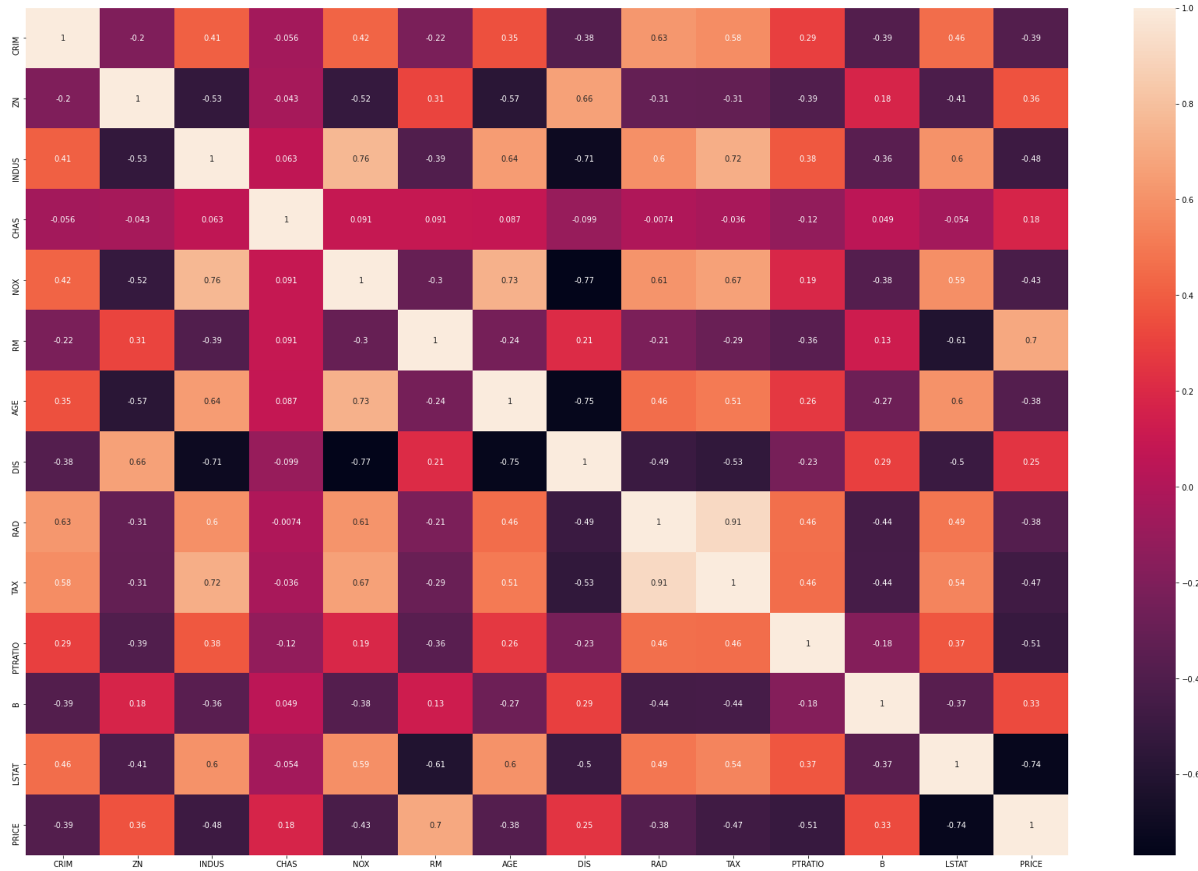
```
bos_data.describe()
```

Out[12]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12

In [13]:

```
plt.figure(figsize=(30, 20))
sns.heatmap(data = bos_data.corr(), annot = True)
plt.show()
```



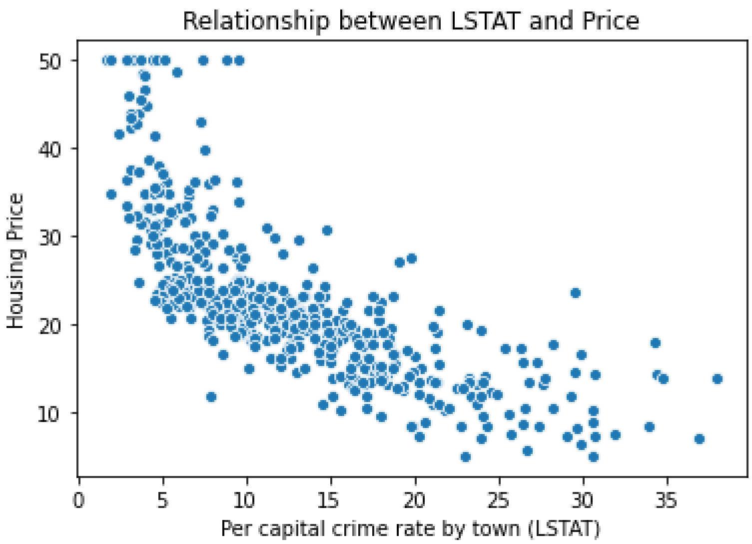
- Since the main goal of this project is to construct a working model which has the capability of predicting the value of houses, we will need to separate the dataset into features and the target variable.
- The features, `'RM'` , `'LSTAT'` , and `'PTRATIO'` , give us quantitative information about each data point.
- The target variable, `'MEDV'` , will be the variable we seek to predict. **These are stored in features and prices, respectively.**

Scatter plots

- Let's look at some **scatter plots** for three variables: 'LSTAT', 'RM' and 'PTRATIO'.

In [14]:

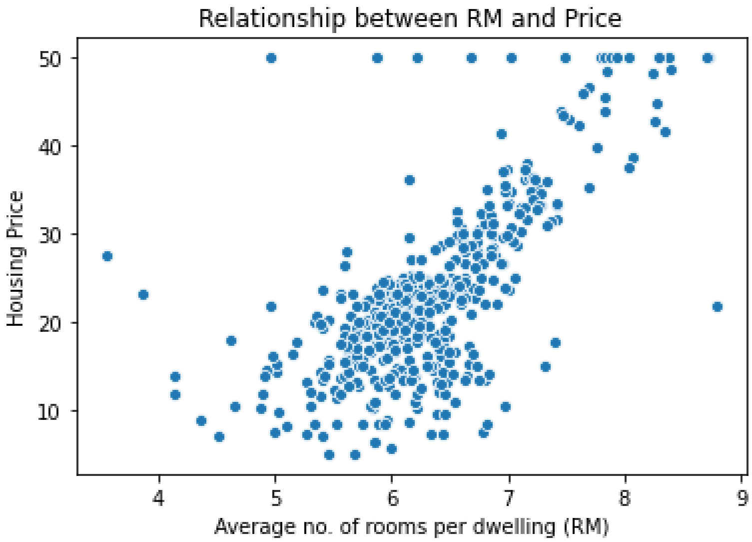
```
sns.scatterplot(x = bos_data.LSTAT, y = bos_data.PRICE)
plt.xlabel('Per capital crime rate by town (LSTAT)')
plt.ylabel('Housing Price')
plt.title("Relationship between LSTAT and Price")
plt.show()
```



- `'LSTAT'` is the percentage of homeowners in the neighborhood considered "lower class" (working poor).
  - An increase in 'LSTAT' indicates that more number of lower class owners live in the neighbourhood indicating cheaper/lower house prices i.e. 'MEDV'. **Hence 'LSTAT' and 'PRICE' are negatively correlated**.

In [15]:

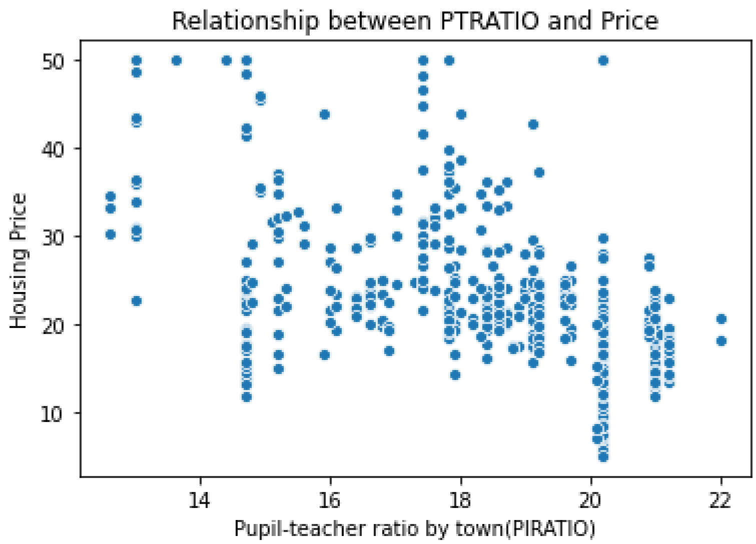
```
sns.scatterplot(x = bos_data.RM, y = bos_data.PRICE)
plt.xlabel('Average no. of rooms per dwelling (RM)')
plt.ylabel('Housing Price')
plt.title("Relationship between RM and Price")
plt.show()
```



- `'RM'` is the average number of rooms among homes in the neighborhood.
  - An increase in 'RM' indicates more number or rooms or more space which increase the price i.e. 'MEDV'. **Hence 'LSTAT' and 'PRICE' are positively correlated.**

In [16]:

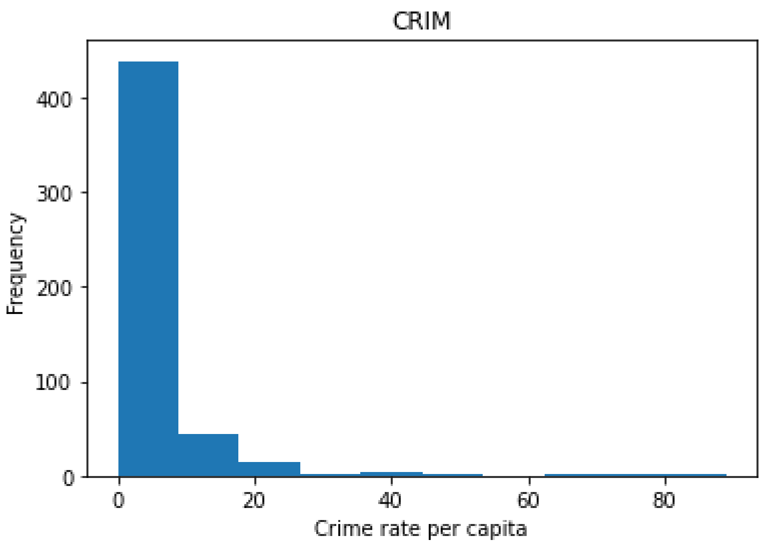
```
sns.scatterplot(x = bos_data.PTRATIO, y = bos_data.PRICE)
plt.xlabel('Pupil-teacher ratio by town(PIRATIO)')
plt.ylabel('Housing Price')
plt.title("Relationship between PTRATIO and Price")
plt.show()
```



- `'PTRATIO'` is the ratio of students to teachers in primary and secondary schools in the neighborhood.
  - An increase in 'PTRATIO' indicates that there are more number students than teachers and the teachers will not be able to pay more attention to each of students thus lowering the quality of education which will decrease the price i.e. 'MEDV'. **Hence 'PTRATIO' and 'PRICE' are negatively correlated.**

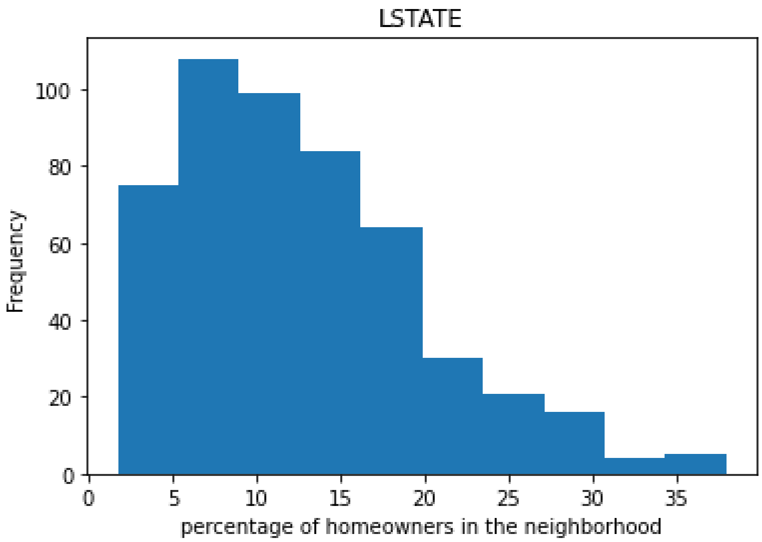
In [17]:

```
plt.hist(bos_data.CRIM)
plt.title("CRIM")
plt.xlabel("Crime rate per capita")
plt.ylabel("Frequency")
plt.show()
```



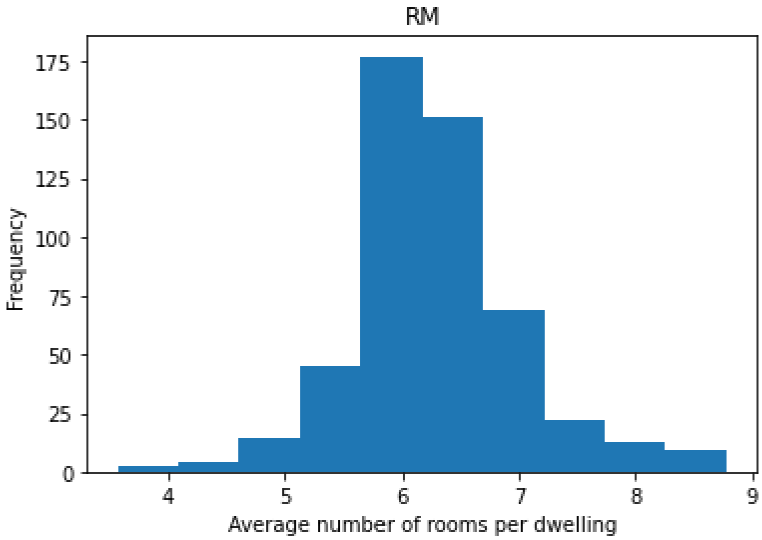
In [18]:

```
plt.hist(bos_data.LSTAT)
plt.title("LSTATE")
plt.xlabel("percentage of homeowners in the neighborhood")
plt.ylabel("Frequency")
plt.show()
```



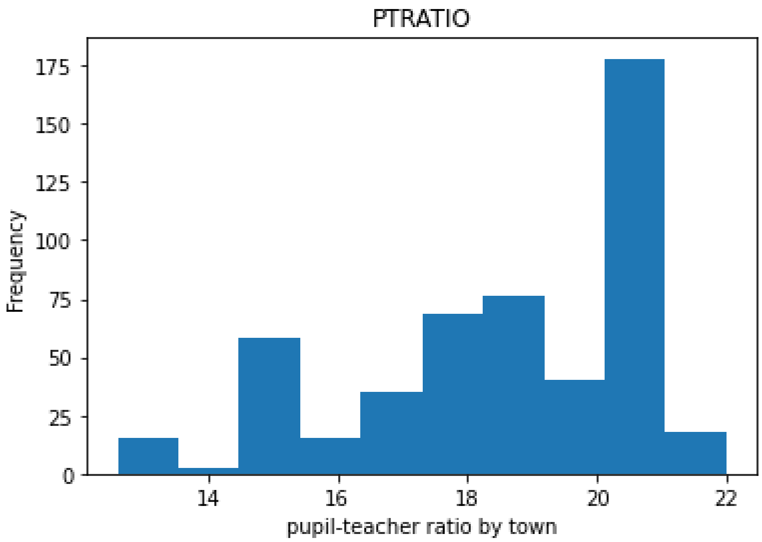
In [19]:

```
plt.hist(bos_data.RM)
plt.title("RM")
plt.xlabel("Average number of rooms per dwelling")
plt.ylabel("Frequency")
plt.show()
```



In [20]:

```
plt.hist(bos_data.PTRATIO)
plt.title("PTRATIO")
plt.xlabel("pupil-teacher ratio by town")
plt.ylabel("Frequency")
plt.show()
```

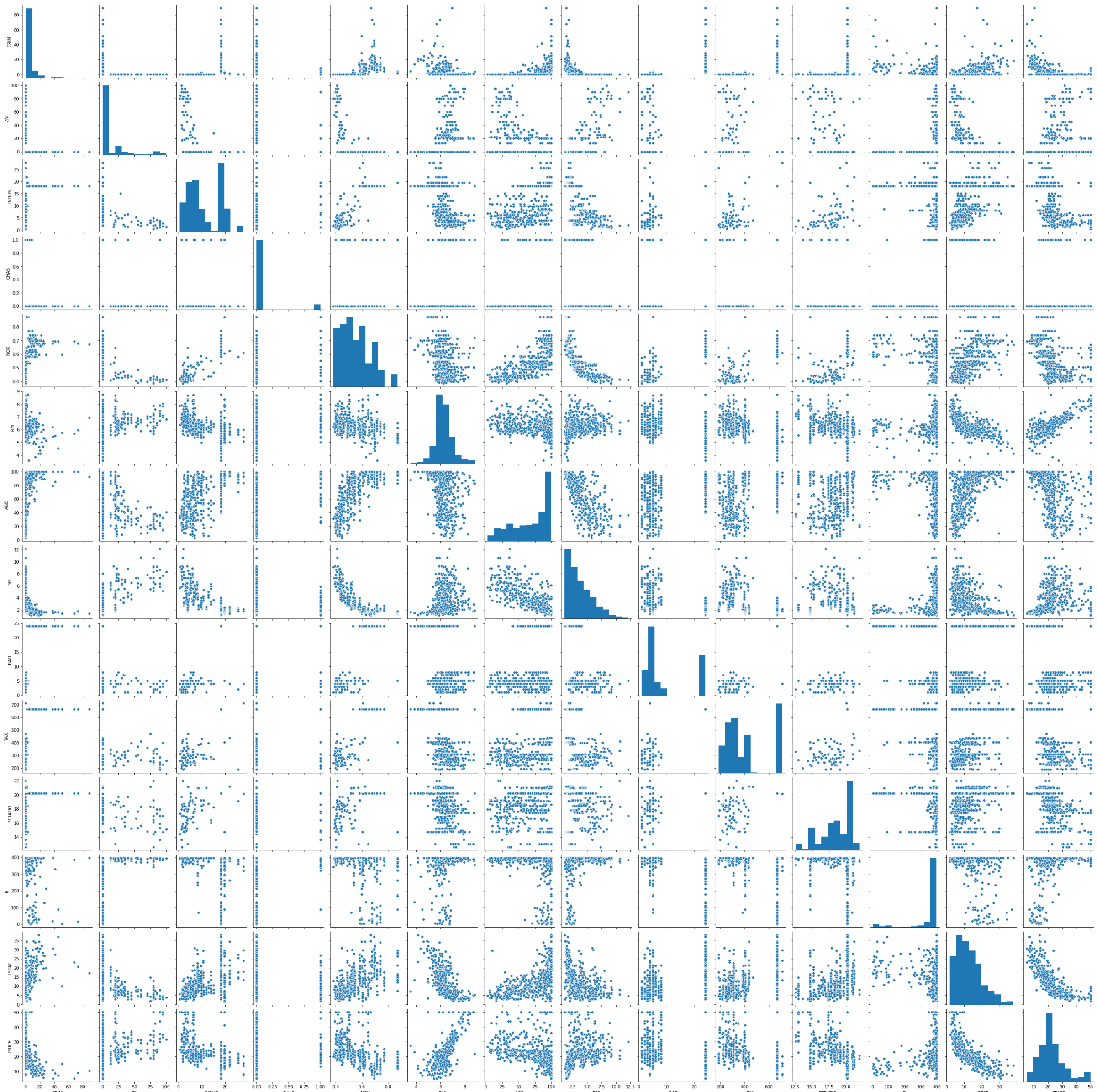


- Let's see the **pair plot**

In [21]:

```
plt.figure(figsize = (20, 20))
sns.pairplot(bos_data)
plt.show()
```

<Figure size 1440x1440 with 0 Axes>



Step 4: Data Standardization and Normalization

Rescaling the Features

- It is extremely important to rescale the variables so that varaibles have a comparable scale.
- if we don't have comparable scales, then some of the coeff. as obtained by fitting the regression model might be very large or very small as compared to the other coeff.
- this might be very annoying at the time of model evaluation.

as we know, there are two common ways to rescaling

1. Min-Max Scaling
2. Standardisation (mean-0, sigma-1)

- rescale all the numeric data in between 0 to 1

We use StandardScaler Scaling

In [22]:

```
from sklearn.preprocessing import StandardScaler
```

In [23]:

```
var_list = list(bos_data.columns)
```

In [24]:

```
var_list.remove('PRICE')
```

In [25]:

```
def min_max_scale(col, bos_data):
    scale = StandardScaler()
    bos_data[col] = scale.fit_transform(bos_data[[col]])
    return bos_data
```

In [26]:

```
for col in var_list:
    bos_data = min_max_scale(col, bos_data)
```

In [27]:

```
bos_data.head()
```

Out[27]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD
0	-0.419782	0.284830	-1.287909	-0.272599	-0.144217	0.413672	-0.120013	0.140214	-0.98284
1	-0.417339	-0.487722	-0.593381	-0.272599	-0.740262	0.194274	0.367166	0.557160	-0.86788
2	-0.417342	-0.487722	-0.593381	-0.272599	-0.740262	1.282714	-0.265812	0.557160	-0.86788
3	-0.416750	-0.487722	-1.306878	-0.272599	-0.835284	1.016303	-0.809889	1.077737	-0.75292
4	-0.412482	-0.487722	-1.306878	-0.272599	-0.835284	1.228577	-0.511180	1.077737	-0.75292

Step 5: Creation of Train and Test data sets using optimum parameters

In [28]:

```
from sklearn.model_selection import train_test_split

df_train, df_test = train_test_split(bos_data, test_size = 0.2, random_state = 0)

df_train.shape, df_test.shape
```

Out[28]:

```
((404, 14), (102, 14))
```

In [29]:

```
y_train = df_train.pop('PRICE')
X_train = df_train
```

Step 6: Model Training using the ML Algorithm tested above

6.1 : Building Linear Regression by statsmodel.api

- As wee might noticed, 'PRICE' seems to the correlated to 'RM' the most.
- so we pick 'RM' as the first variable and we'll try to fit a regression line.

In [30]:

```
# build model with statsmodel.api

import statsmodels.api as sm
```

- **Dividing into X and Y sets for the model building**
- ***Linear model with variable 'RM' as we above picked it*.**

In [31]:

```
# add constant
X_train_lm = X_train[['RM']]
X_train_lm = sm.add_constant(X_train_lm)
```

In [32]:

```
# creating a first fitted model

lr = sm.OLS(y_train, X_train_lm).fit()
```

In [33]:

```
lr.params
```

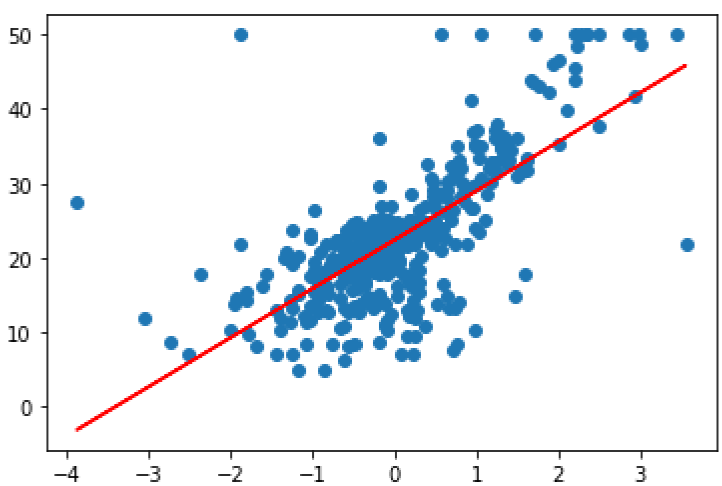
Out[33]:

```
const    22.450958
RM        6.581495
dtype: float64
```

In [34]:

```
# scatter plot and the fitted regression line

plt.scatter(X_train_lm.iloc[:,1], y_train)
plt.plot(X_train_lm.iloc[:,1],22.450958 + 6.581495 * X_train_lm.iloc[:,1],'r')
plt.show()
```



In [35]:

```
lr.summary()
```

Out[35]:

OLS Regression Results

Dep. Variable:	PRICE	**R-squared:**	0.497
Model:	OLS	**Adj. R-squared:**	0.496
Method:	Least Squares	**F-statistic:**	397.3
Date:	Wed, 20 Jan 2021	**Prob (F-statistic):**	5.64e-62
Time:	17:09:20	**Log-Likelihood:**	-1332.2
No. Observations:	404	**AIC:**	2668.
Df Residuals:	402	**BIC:**	2676.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	22.4510	0.326	68.768	0.000	21.809	23.093
RM	6.5815	0.330	19.933	0.000	5.932	7.231

Omnibus:	66.615	**Durbin-Watson:**	1.892
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	456.894
Skew:	0.458	**Prob(JB):**	6.12e-100
Kurtosis:	8.129	**Cond. No.**	1.03

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

`Adding another variable`

- the R sq. value obtained is 0.497. since we have so many variables,
- we can clearly do better than this.
- so let's go ahead and add the second variable.
- i.e `CRIME` . (choice of variable is random)

In [36]:

```
# build a linear model again with two varaibles

X_train_lm = X_train[['RM','CRIM']]

# add constant
X_train_lm = sm.add_constant(X_train_lm)
```

In [37]:

```
# build model

lr = sm.OLS(y_train, X_train_lm).fit()
```

In [38]:

```
lr.params
```

Out[38]:

```
const    22.390549
RM        6.061324
CRIM     -2.765795
dtype: float64
```

In [39]:

```
lr.summary()
```

Out[39]:

OLS Regression Results

Dep. Variable:	PRICE	**R-squared:**	0.574
Model:	OLS	**Adj. R-squared:**	0.572
Method:	Least Squares	**F-statistic:**	270.5
Date:	Wed, 20 Jan 2021	**Prob (F-statistic):**	4.21e-75
Time:	17:09:21	**Log-Likelihood:**	-1298.5
No. Observations:	404	**AIC:**	2603.
Df Residuals:	401	**BIC:**	2615.
Df Model:	2		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	22.3905	0.301	74.436	0.000	21.799	22.982
RM	6.0613	0.310	19.541	0.000	5.452	6.671
CRIM	-2.7658	0.324	-8.532	0.000	-3.403	-2.129

Omnibus:	126.742	**Durbin-Watson:**	1.906
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	807.488
Skew:	1.172	**Prob(JB):**	4.53e-176
Kurtosis:	9.517	**Cond. No.**	1.23

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

- we have clearly imporved the model as the value of adjusted R- sq.
- as its value has gone up to `0.574` from `0.497` .

`Adding another variable`

- let's go ahead and add another variable **`ZN`**

In [40]:

```
X_train_lm = X_train[['RM','CRIM','ZN']]
```

In [41]:

```
# add constant

X_train_lm = sm.add_constant(X_train_lm)
```

In [42]:

```
# build the model

lr = sm.OLS(y_train, X_train_lm).fit()
```

In [43]:

```
lr.params
```

Out[43]:

```
const    22.381875
RM        5.622117
CRIM     -2.565376
ZN        1.336155
dtype: float64
```

In [44]:

```
lr.summary()
```

Out[44]:

OLS Regression Results

Dep. Variable:	PRICE	**R-squared:**	0.593
Model:	OLS	**Adj. R-squared:**	0.590
Method:	Least Squares	**F-statistic:**	194.2
Date:	Wed, 20 Jan 2021	**Prob (F-statistic):**	1.09e-77
Time:	17:09:21	**Log-Likelihood:**	-1289.5
No. Observations:	404	**AIC:**	2587.
Df Residuals:	400	**BIC:**	2603.
Df Model:	3		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	22.3819	0.295	75.983	0.000	21.803	22.961
RM	5.6221	0.321	17.529	0.000	4.992	6.253
CRIM	-2.5654	0.321	-7.995	0.000	-3.196	-1.935
ZN	1.3362	0.313	4.265	0.000	0.720	1.952

Omnibus:	144.339	**Durbin-Watson:**	1.913
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	941.398
Skew:	1.361	**Prob(JB):**	3.79e-205
Kurtosis:	9.965	**Cond. No.**	1.51

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

- we have improved the adjusted R-sq. again.
- now let's go ahead and add all the feature variables

Adding all the variables to the model

In [45]:

```
# check all the columns of the dataframe

X_train.columns
```

Out[45]:

```
Index(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TA
X',
       'PTRATIO', 'B', 'LSTAT'],
      dtype='object')
```

In [46]:

```
# add a constant

X_train_lm = sm.add_constant(X_train)
```

In [47]:

```
# build the model

lr = sm.OLS(y_train, X_train_lm).fit()
```

In [48]:

```
lr.params
```

Out[48]:

```
const      22.480353
CRIM       -1.026382
ZN          1.043346
INDUS       0.037594
CHAS        0.593962
NOX        -1.866519
RM          2.603226
AGE        -0.087768
DIS        -2.916465
RAD         2.124022
TAX        -1.850331
PTRATIO    -2.262124
B           0.739679
LSTAT      -3.515584
dtype: float64
```

In [49]:

```
lr.summary()
```

Out[49]:

OLS Regression Results

Dep. Variable:	PRICE	**R-squared:**	0.773
Model:	OLS	**Adj. R-squared:**	0.765
Method:	Least Squares	**F-statistic:**	102.2
Date:	Wed, 20 Jan 2021	**Prob (F-statistic):**	9.64e-117
Time:	17:09:22	**Log-Likelihood:**	-1171.5
No. Observations:	404	**AIC:**	2371.
Df Residuals:	390	**BIC:**	2427.
Df Model:	13		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	22.4804	0.224	100.452	0.000	22.040	22.920
CRIM	-1.0264	0.315	-3.257	0.001	-1.646	-0.407
ZN	1.0433	0.336	3.102	0.002	0.382	1.705
INDUS	0.0376	0.435	0.087	0.931	-0.817	0.892
CHAS	0.5940	0.229	2.595	0.010	0.144	1.044
NOX	-1.8665	0.488	-3.828	0.000	-2.825	-0.908
RM	2.6032	0.321	8.106	0.000	1.972	3.235
AGE	-0.0878	0.403	-0.218	0.828	-0.880	0.704
DIS	-2.9165	0.450	-6.480	0.000	-3.801	-2.032
RAD	2.1240	0.610	3.481	0.001	0.924	3.324
TAX	-1.8503	0.656	-2.819	0.005	-3.141	-0.560
PTRATIO	-2.2621	0.296	-7.636	0.000	-2.845	-1.680
B	0.7397	0.269	2.749	0.006	0.211	1.269
LSTAT	-3.5156	0.387	-9.086	0.000	-4.276	-2.755

Omnibus:	141.494	**Durbin-Watson:**	1.996
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	629.882
Skew:	1.470	**Prob(JB):**	1.67e-137
Kurtosis:	8.365	**Cond. No.**	9.75

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

- looking at the p-value,
- it look like some of the variables **aren't really significant** (in the presence of other variables).
  - we could simply drop the variable with the `highest p-value` (non-significant).
  - A better way would be to supplement this with the `VIF information`.

`Checking VIF`

- Variance Inflation Factor (VIF), gives a basic quantitative idea ablut how much the feautre variables are correlated with each other.
- **it is an extremely important parameter to test our linear model.**

In [50]:

```
# importing library for VIF

from statsmodels.stats.outliers_influence import variance_inflation_factor
```

In [51]:

```
# create a dataframe to store data

vif = pd.DataFrame()
```

In [52]:

```
# creating 1st column which contain all the column name in vif DataFrame

vif['Features'] = X_train.columns
```

In [53]:

```
# creating 2nd column which contain value of VIF

vif['VIF'] = [variance_inflation_factor(X_train.values, i) for i in range(X_train.shape[1
])]
vif['VIF'] = round(vif['VIF'],2)  # round values upto 2 decimal
```

In [54]:

```
# sorting value of VIF by decending order

vif = vif.sort_values(by = 'VIF', ascending=False)
vif
```

Out[54]:

	Features	VIF
9	TAX	8.90
8	RAD	7.43
4	NOX	4.74
7	DIS	3.99
2	INDUS	3.96
6	AGE	3.27
12	LSTAT	3.15
1	ZN	2.34
5	RM	2.03
10	PTRATIO	1.82
0	CRIM	1.79
11	B	1.38
3	CHAS	1.06

- we generally want a VIF theat is less than 5
- so, there are clearly some variable we need to drop

Dropping the variables and updating the model

now as we can see from the 'Summary' and 'VIF' dataframe. some variable are still indignificant.

- Here first we drop variable which have **high p-value(non-significant) because when we drop this and rebuild the model then value of VIF will modify and reduce**
- so, `INDUS` as it a very high p-value of `0.931`

`Drop 'INDUS' variable and rebuild the model`

In [55]:

```
# dropping variable

X = X_train.drop('INDUS',1)
```

In [56]:

```
# create model
# adding constant

X_train_lm = sm.add_constant(X)
```

In [57]:

```
lr_2 = sm.OLS(y_train, X_train_lm).fit()
```

In [58]:

```
lr_2.summary()
```

Out[58]:

OLS Regression Results

Dep. Variable:	PRICE	**R-squared:**	0.773
Model:	OLS	**Adj. R-squared:**	0.766
Method:	Least Squares	**F-statistic:**	111.0
Date:	Wed, 20 Jan 2021	**Prob (F-statistic):**	8.98e-118
Time:	17:09:23	**Log-Likelihood:**	-1171.5
No. Observations:	404	**AIC:**	2369.
Df Residuals:	391	**BIC:**	2421.
Df Model:	12		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	22.4807	0.223	100.597	0.000	22.041	22.920
CRIM	-1.0274	0.314	-3.267	0.001	-1.646	-0.409
ZN	1.0409	0.335	3.110	0.002	0.383	1.699
CHAS	0.5957	0.228	2.616	0.009	0.148	1.043
NOX	-1.8556	0.470	-3.944	0.000	-2.781	-0.931
RM	2.6007	0.319	8.142	0.000	1.973	3.229
AGE	-0.0885	0.402	-0.220	0.826	-0.880	0.703
DIS	-2.9250	0.438	-6.671	0.000	-3.787	-2.063
RAD	2.1073	0.578	3.646	0.000	0.971	3.244
TAX	-1.8242	0.582	-3.133	0.002	-2.969	-0.680
PTRATIO	-2.2581	0.292	-7.728	0.000	-2.833	-1.684
B	0.7390	0.269	2.751	0.006	0.211	1.267
LSTAT	-3.5130	0.385	-9.118	0.000	-4.270	-2.756

Omnibus:	141.572	**Durbin-Watson:**	1.997
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	630.874
Skew:	1.470	**Prob(JB):**	1.02e-137
Kurtosis:	8.370	**Cond. No.**	8.15

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

In [59]:

```
# calculating VIF again new model

vif = pd.DataFrame()
vif['Features'] = X.columns
vif['VIF'] = [variance_inflation_factor(X.values,i) for i in range(X.shape[1])]
vif['VIF'] = round(vif['VIF'],2)
vif = vif.sort_values(by='VIF', ascending=False)
vif
```

Out[59]:

	Features	VIF
8	TAX	7.02
7	RAD	6.68
3	NOX	4.42
6	DIS	3.79
5	AGE	3.26
11	LSTAT	3.13
1	ZN	2.32
4	RM	2.02
0	CRIM	1.79
9	PTRATIO	1.77
10	B	1.38
2	CHAS	1.05

as we can notice some of the variable have high VIF values as well as high p-values.

- the variable `TAX` has a high `VIF (7.02)` and `p-value (0.002)` .
- hence, this variable isn't of much use and should be dropped

`Again drop another variable 'TAX' and rebuild the model`

In [60]:

```
# dropping variable 'bedroom'

X = X.drop('TAX', axis=1)  # 1 for column
```

In [61]:

```
# add constant

X_train_lm = sm.add_constant(X)
```

In [62]:

```
# build model

lr_3 = sm.OLS(y_train, X_train_lm).fit()
```

In [63]:

```
lr_3.summary()
```

Out[63]:

OLS Regression Results

Dep. Variable:	PRICE	**R-squared:**	0.767
Model:	OLS	**Adj. R-squared:**	0.761
Method:	Least Squares	**F-statistic:**	117.5
Date:	Wed, 20 Jan 2021	**Prob (F-statistic):**	9.94e-117
Time:	17:09:24	**Log-Likelihood:**	-1176.5
No. Observations:	404	**AIC:**	2377.
Df Residuals:	392	**BIC:**	2425.
Df Model:	11		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	22.5234	0.226	99.860	0.000	22.080	22.967
CRIM	-0.9952	0.318	-3.131	0.002	-1.620	-0.370
ZN	0.8302	0.332	2.504	0.013	0.178	1.482
CHAS	0.6568	0.229	2.863	0.004	0.206	1.108
NOX	-2.1729	0.465	-4.677	0.000	-3.086	-1.260
RM	2.7341	0.320	8.542	0.000	2.105	3.363
AGE	-0.1081	0.407	-0.266	0.791	-0.908	0.692
DIS	-2.7613	0.440	-6.272	0.000	-3.627	-1.896
RAD	0.7462	0.386	1.935	0.054	-0.012	1.504
PTRATIO	-2.4121	0.291	-8.282	0.000	-2.985	-1.840
B	0.7608	0.272	2.801	0.005	0.227	1.295
LSTAT	-3.5375	0.390	-9.082	0.000	-4.303	-2.772

Omnibus:	133.243	**Durbin-Watson:**	1.998
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	562.909
Skew:	1.393	**Prob(JB):**	5.83e-123
Kurtosis:	8.067	**Cond. No.**	5.63

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

In [64]:

```
# calculating VIF again in new model

vif = pd.DataFrame()
vif['Features'] = X.columns
vif['VIF'] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
vif['VIF'] = round(vif['VIF'],2)
vif = vif.sort_values(by='VIF', ascending=False)
vif
```

Out[64]:

	Features	VIF
3	NOX	4.22
6	DIS	3.74
5	AGE	3.26
10	LSTAT	3.13
7	RAD	2.91
1	ZN	2.23
4	RM	1.98
0	CRIM	1.79
8	PTRATIO	1.72
9	B	1.38
2	CHAS	1.04

- now we can see the VIF and p-value both are within an **acceptable range**. so we go ahead and make our predictions using this model only

Residual Analysis of the train data

- now, to check if the error terms are also normally distributed (which is infact, one of the major assumptions of linear regression).
- let us plot the histogram of the error terms and see what it looks like. with final model

In [65]:

```
y_train_pred  = lr_3.predict(X_train_lm)
y_train_pred
```

Out[65]:

```
220    32.340189
71     22.214052
240    27.581276
6      23.673391
417     6.386207
         ...
323    19.072939
192    34.652608
117    24.688279
47     18.101102
172    22.906744
Length: 404, dtype: float64
```

In [66]:

```
# error term RSS

res = y_train - y_train_pred
res
```

Out[66]:

```
220   -5.640189
71    -0.514052
240   -5.581276
6     -0.773391
417    4.013793
         ...
323   -0.572939
192    1.747392
117   -5.488279
47    -1.501102
172    0.193256
Length: 404, dtype: float64
```

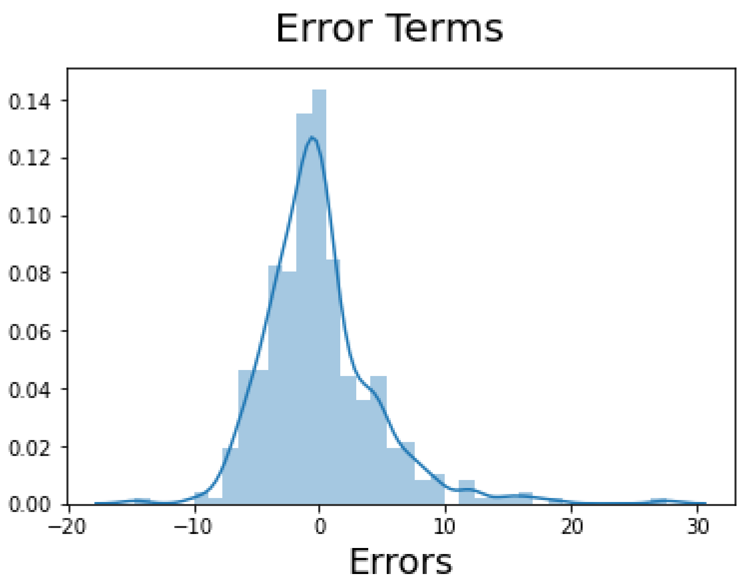
In [67]:

```
# plot histogram plot
import seaborn as sns

fig = plt.figure()
sns.distplot(res)
fig.suptitle('Error Terms', fontsize = 20)
plt.xlabel('Errors', fontsize = 18)
```

Out[67]:

```
Text(0.5, 0, 'Errors')
```



Making prediction on Test set using Final model

In [68]:

```
# this is our test dataset
df_test.head()
```

Out[68]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	
329	-0.412692	-0.487722	-1.152214	-0.272599	-0.818007	0.068904	-1.826921	0.674814	-0.63
371	0.653875	-0.487722	1.015999	-0.272599	0.659147	-0.097781	1.117494	-1.248292	1.66
219	-0.407222	-0.487722	0.401721	3.668398	-0.040557	0.125891	0.847234	-0.205237	-0.52
403	2.465737	-0.487722	1.015999	-0.272599	1.194724	-1.332960	0.975252	-0.994588	1.66
78	-0.413947	-0.487722	0.247057	-0.272599	-1.016689	-0.074986	-0.528960	0.579502	-0.52

- Dividing into X_test ant y_test

In [69]:

```
y_test = df_test.pop('PRICE')
X_test = df_test
```

In [70]:

```
# adding constant variable to test dataset

X_test_lm = sm.add_constant(X_test)
```

In [71]:

```
# as we know the final model so droppping unwanted
# variable from test dataset

X_test_lm = X_test_lm.drop(['INDUS','TAX'], axis = 1)
```

making prediction using final model which is lr_3

In [72]:

```
y_test_pred = lr_3.predict(X_test_lm)
```

calculating R sq.

In [73]:

```
from sklearn.metrics import r2_score
```

In [74]:

```
r2 = r2_score(y_test, y_test_pred)
```

In [75]:

```
# R sq. for test data set
r2
```

Out[75]:

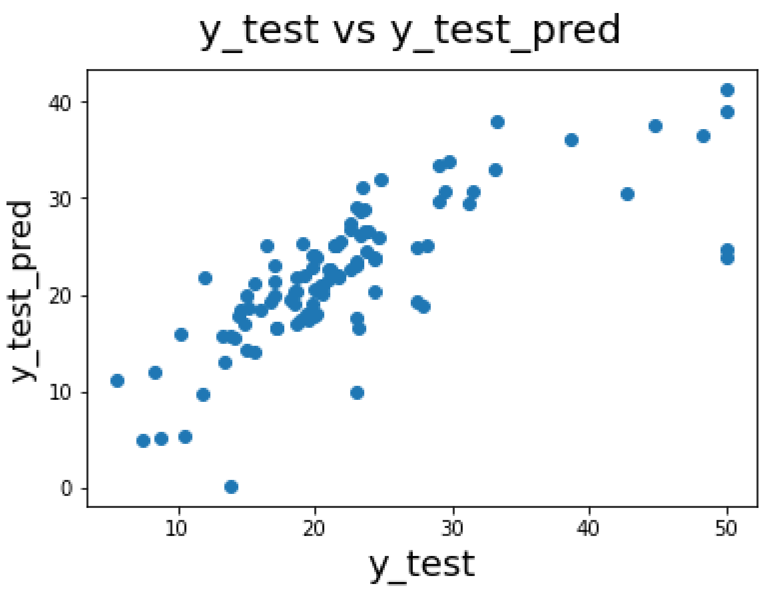
```
0.5786416823996582
```

- Model Evaluation

In [76]:

```
# plotting y_test and y_test_pred to understand the spread

fig = plt.figure()
plt.scatter(y_test, y_test_pred)
fig.suptitle("y_test vs y_test_pred", fontsize = 20)
plt.xlabel('y_test', fontsize = 18)
plt.ylabel('y_test_pred', fontsize = 16)
plt.show()
```



In [77]:

```
res = y_test - y_test_pred
res
```

Out[77]:

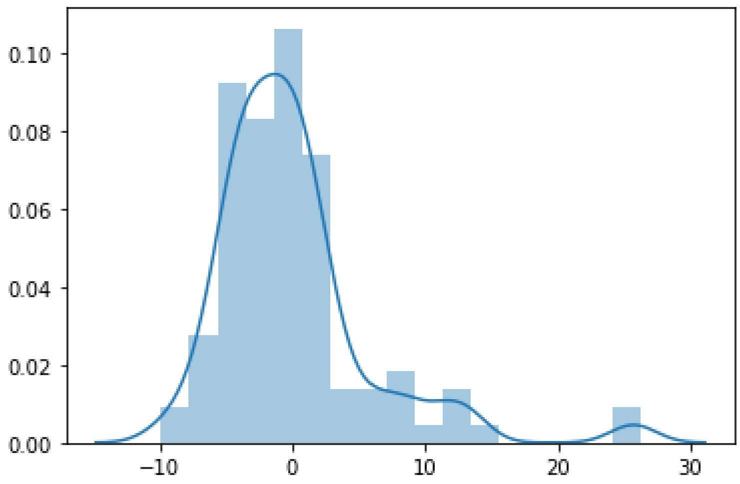
```
329    -4.098243
371    26.163080
219    -6.119655
403    -3.686488
78     -1.352907
         ...
56     -1.221177
455    -1.415425
60      1.617532
213     3.002182
108    -3.005420
Length: 102, dtype: float64
```

In [78]:

```
sns.distplot(res)  # error plot for test data set
```

Out[78]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x29c0640e460>
```



6.2 : Building Linear Regression by `Sklearn`

In [79]:

```
data = bos_data.copy()
```

In [80]:

```
bos_data.columns
```

Out[80]:

```
Index(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TA
X',
       'PTRATIO', 'B', 'LSTAT', 'PRICE'],
      dtype='object')
```

In [81]:

```
data.drop(columns = ['INDUS','TAX'], inplace = True)
```

In [82]:

```
y = data.pop('PRICE')
X = data
```

Splitting the Data into Training and Testing Sets

In [83]:

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state =
0)

x_train.shape, x_test.shape
```

Out[83]:

```
((404, 11), (102, 11))
```

Model Training using ML algo tested above

In [84]:

```
from sklearn.linear_model import LinearRegression
```

In [85]:

```
lr = LinearRegression()
```

In [86]:

```
lr.fit(x_train, y_train)
```

Out[86]:

```
LinearRegression()
```

- **Predicted values of `test data`**

In [87]:

```
y_pred_test = lr.predict(x_test)
```

- **Predicted values of `train data`**

In [88]:

```
y_pred_train = lr.predict(x_train)
```

Calculation of Model Accuracy:

- **Both Training and test Accuracies**

In [89]:

```
print("Test data Accuracy  : ", round(r2_score(y_test, y_pred_test), 3))
print("Train data Accuracy : ", round(r2_score(y_train, y_pred_train), 3))
```

```
Test data Accuracy  :  0.579
Train data Accuracy :  0.767
```

In [90]:

```
from sklearn.metrics import mean_absolute_error, mean_squared_error
```

- **Mean Absolute Error**

In [92]:

```
print('MAE for Test Data  : ' , mean_absolute_error(y_test, y_pred_test))
print('MAE for Train Data : ' , mean_absolute_error(y_train, y_pred_train))
```

```
MAE for Test Data  :  3.9498411059954015
MAE for Train Data :  3.132515673402906
```

- **Mean Square Error**

In [94]:

```
print('MSE for Test Data  : ' , mean_squared_error(y_test, y_pred_test))
print('MSE for Train Data : ' , mean_squared_error(y_train, y_pred_train))
```

```
MSE for Test Data  :  34.31055008794417
MSE for Train Data :  19.812144707771346
```

- **So it looks like our model train data score is less on the test data**. #### It's mean our model is `overfitted` . let's check by using `Regularization`

Regularization

In [95]:

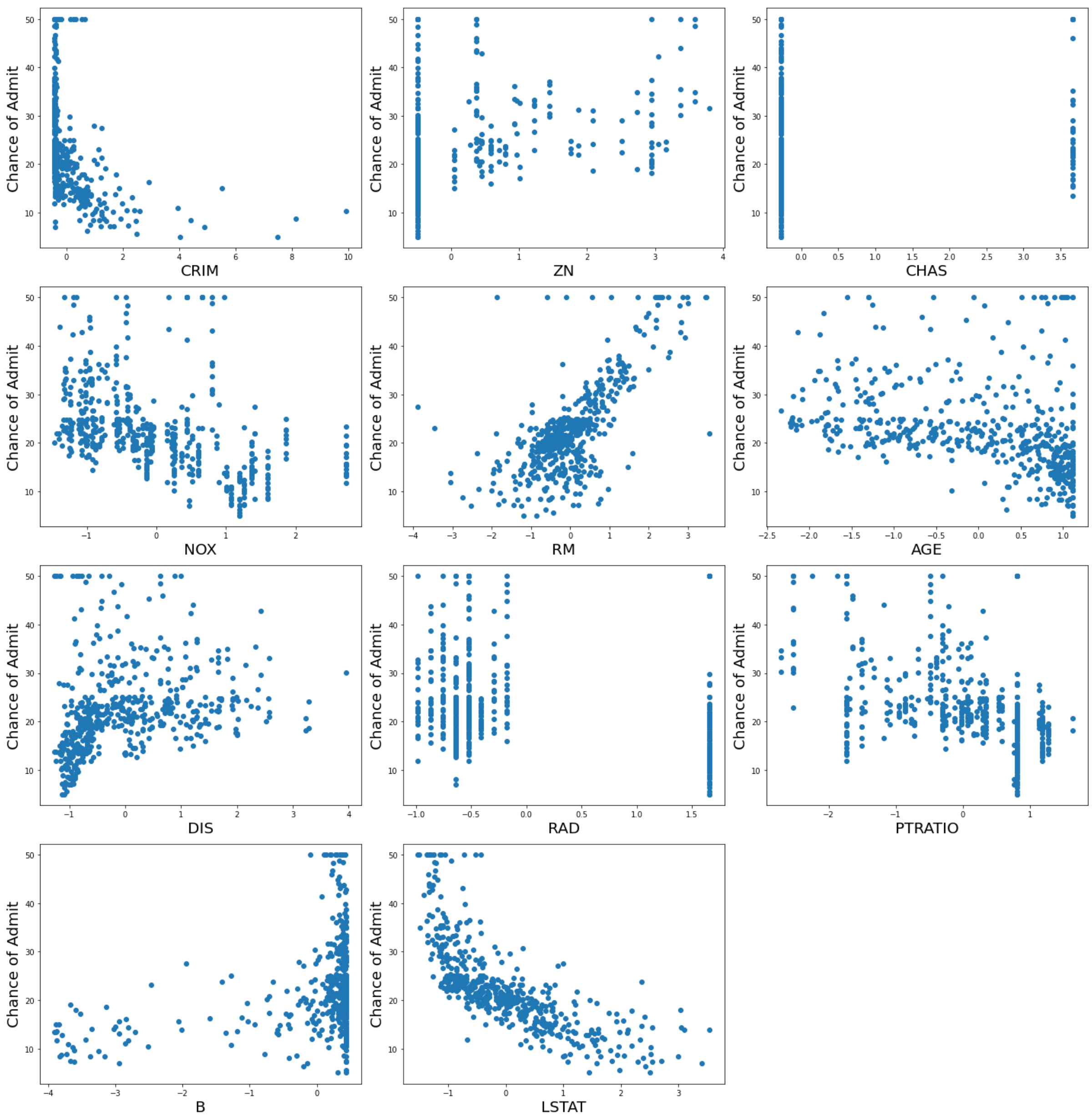
```
from sklearn.linear_model import Ridge, RidgeCV
from sklearn.linear_model import Lasso, LassoCV
from sklearn.linear_model import ElasticNet, ElasticNetCV
```

In [96]:

```
# let's see how data is distributed for every column

plt.figure(figsize=(20,25))
plotno = 1

for col in X:
    if plotno <= 15:
        ax = plt.subplot(5,3, plotno)
        plt.scatter(X[col], y)
        plt.xlabel(col, fontsize = 20)
        plt.ylabel('Chance of Admit', fontsize = 20)
    plotno += 1
plt.tight_layout()
```



- The relationship between the dependent and independent variable look `fairly linear` , thus our linearity assumption is `satisfied` .
- let's move ahead and `check for multicolinearity` .

Data Standardization

In [97]:

```
scale = StandardScaler()
```

In [98]:

```
X_scale = scale.fit_transform(X)
```

Split data into Train and Test

In [99]:

```
x_train, x_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.20, random_s
tate = 42)    #355
```

In [100]:

```
x_train.shape, x_test.shape, y_train.shape
```

Out[100]:

```
((404, 11), (102, 11), (404,))
```

Regularization

- 1). **`Lasso (L1)`**
- 2). **`Ridge (L2)`**
- 3). **`Elastic Net`**
- Let's see if our model is `overfitting` our training data or not. ## 1. **`Lasso Regression`**
- **LassoCV** will **return best alpha** and **coeff.** after performing 10 CV

In [101]:

```
lasscv = LassoCV(alphas = None, cv = 10, max_iter = 1000, normalize = True)
lasscv.fit(x_train, y_train)
```

Out[101]:

```
LassoCV(cv=10, normalize=True)
```

In [102]:

```
# best alpha parameter
```

In [103]:

```
alpha = lasscv.alpha_
alpha
```

Out[103]:

```
0.0009725797661664984
```

- now that we have `best parameter`
- let's use **lasso Regression** and see how well our data had fitted before

In [104]:

```
lass_reg = Lasso(alpha)
lass_reg.fit(x_train, y_train)
```

Out[104]:

```
Lasso(alpha=0.0009725797661664984)
```

`Checking Lasso Score`

- **Score for `train`**

In [105]:

```
lass_reg.score(x_train, y_train)
```

Out[105]:

```
0.7467562339777498
```

- **Score for `test`**

In [106]:

```
lass_reg.score(x_test, y_test)
```

Out[106]:

```
0.6510131983061462
```

- our score for **test data `(65.101%)`** comes not same as before using regularization. So, it is fair to say our OLS model **is overfit** the data.

2. Ridge Regression

- **RidgeCV** will **return best alpha** and **coeff.** after performing 10 CV
- We will pass an array of random numbers for ridgeCV to select best alpha from them

In [107]:

```
alpha = np.random.uniform(low = 0, high = 10, size = (50,))
ridgecv = RidgeCV(alphas= alpha, cv = 10 , normalize=True)
ridgecv.fit(x_train, y_train)
```

Out[107]:

```
RidgeCV(alphas=array([4.19088048, 6.14813811, 5.19071174, 6.84022679, 2.26593
818,
       5.36465011, 8.72132029, 0.72594771, 1.11476736, 0.7853579 ,
       5.03328249, 7.88620301, 1.60918344, 9.24227043, 1.09190458,
       3.01634306, 1.97601308, 5.50824773, 5.68656898, 1.23472488,
       4.60143808, 1.67381429, 6.18341398, 5.62388918, 0.5561337 ,
       8.55533135, 4.83049296, 0.7944039 , 1.05916495, 3.85257315,
       0.08492379, 2.08790489, 5.95047575, 4.5642833 , 3.38369793,
       8.77198575, 5.52381267, 7.49788995, 0.06664861, 0.50303636,
       0.22974858, 4.30523044, 7.67907489, 4.18128492, 5.95243093,
       0.65179207, 8.85504238, 6.12729838, 4.25638262, 2.08896906]),
        cv=10, normalize=True)
```

- **best alpha parameter**

In [108]:

```
alpha = ridgecv.alpha_
alpha
```

Out[108]:

```
0.06664860689511443
```

- now that we have `best parameter`
- let's use **Ridge Regression** and see how well our data had fitted before

In [109]:

```
ridge_reg = Ridge(alpha = alpha)
ridge_reg.fit(x_train, y_train)
```

Out[109]:

```
Ridge(alpha=0.06664860689511443)
```

Checking Ridge Score

- **Score for `train`**

In [110]:

```
ridge_reg.score(x_train, y_train)
```

Out[110]:

```
0.7467565140286906
```

- **Score for `test`**

In [111]:

```
ridge_reg.score(x_test, y_test)
```

Out[111]:

```
0.6510377206836205
```

- we got the **same score using Ridge reg** as well as we got in linear reg so it's safe to say there is **`overfitting`**

3. Elatic Net

- **Elatic Net** will **return best alpha** and **coeff.** after performing 10 CV
- We will pass an array of random numbers for ridgeCV to select best alpha from them

In [112]:

```
elaticv = ElasticNetCV(alphas = None, cv = 10)
elaticv.fit(x_train, y_train)
```

Out[112]:

```
ElasticNetCV(cv=10)
```

- **best alpha parameter**

In [113]:

```
alpha = elaticv.alpha_
alpha
```

Out[113]:

```
0.027456948333396687
```

- **l1_ratio** gives how close the model is to L1 regulatization below value indicates we are `giving equal prefernce to L1 and L2`

In [114]:

```
ratio = elaticv.l1_ratio
```

- now that we have `best parameter`
- let's use **Elastic Net** and see how well our data had fitted before

In [115]:

```
elastic_reg = ElasticNet(alpha=alpha, l1_ratio = ratio)
elastic_reg.fit(x_train, y_train)
```

Out[115]:

```
ElasticNet(alpha=0.027456948333396687)
```

`Checking Elastic Net Score`

- **Score for `train`**

In [116]:

```
elastic_reg.score(x_train, y_train)
```

Out[116]:

```
0.7463361031829615
```

- **Score for `test`**

In [117]:

```
elastic_reg.score(x_test, y_test)
```

Out[117]:

```
0.6504229437929967
```

- So, we can see by using different type of regularization, we still are getting the `same score` .
- That means our **OLS model has been well trained over the training data** and there is **`overfitting`** .

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```