Linear Regression

A Lecture on Programming in Python
By Dr. Akinul Islam Jony

- A linear regression is one of the common statistical models in machine learning.
- The term “linearity” in algebra refers to a linear relationship between two or more variables.
- Linear regression performs the task to predict a dependent variable value (y) based on a given independent variable (x). So, this regression technique finds out a linear relationship between x (input) and y(output). Hence, the name is Linear Regression. If we plot the independent variable (x) on the x-axis and dependent variable (y) on the y-axis, linear regression gives us a straight line that best fits the data points.
- In a nutshell, it is used to show the linear relationship between a dependent variable and one or more independent variables.

Imports and Settings

In [44]:

```
import numpy as np
import pandas as pd

#Visualization Libraries
import matplotlib.pyplot as plt
import seaborn as sns

#To plot the graph embedded in the notebook
%matplotlib inline
```

Load Dataset: Boston

- we will load the Boston housing data from the scikit-learn library
- The Boston Housing dataset contains information about various houses in Boston through different parameters.
- There are 506 samples and 13 feature variables in this dataset. The objective is to predict the value of prices of the house using the given features.

In [2]:
```
from sklearn import datasets

#loading the dataset direclty from sklearn
boston = datasets.load_boston()
```

Explore the Dataset

- sklearn returns Dictionary-like object

In [4]:
```
# Let's print the keys of 'boston' dictionary object
boston.keys()
```

Out[4]:
```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])
```

- data: contains the information for various houses (feature matrix)
- target: prices of the house (target array/vector)
- feature_names: names of the features
- DESCR: describes the dataset
- filename: the physical location of boston csv dataset

In [8]:
```
# Let's print the description of boston dataset
print(boston.DESCR)
```

```
.. _boston_dataset:
```

Boston house prices dataset

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
- TAX full-value property-tax rate per $10,000
- PTRATIO pupil-teacher ratio by town
- B 1000(Bk - 0.63)^2 where Bk is the proportion of black people by town
- LSTAT % lower status of the population
- MEDV Median value of owner-occupied homes in $1000's

:Missing Attribute Values: None

:Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/

This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978. Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980. N.B. Various transformations are used in the table on
pages 244-261 of the latter.

```
The Boston house-price data has been used in many machine learning papers that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wi
ley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth Internation
al Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
```

In [11]:
```
# Let's print the data (features matrix) of boston dataset
print(boston.data)
```

```
[[6.3200e-03 1.8000e+01 2.3100e+00 ... 1.5300e+01 3.9690e+02 4.9800e+00]
 [2.7310e-02 0.0000e+00 7.0700e+00 ... 1.7800e+01 3.9690e+02 9.1400e+00]
 [2.7290e-02 0.0000e+00 7.0700e+00 ... 1.7800e+01 3.9283e+02 4.0300e+00]
 ...
 [6.0760e-02 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9690e+02 5.6400e+00]
 [1.0959e-01 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9345e+02 6.4800e+00]
 [4.7410e-02 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9690e+02 7.8800e+00]]
```

In [13]:
```
# Let's check the shape of features matrix
print(boston.data.shape)
```

```
(506, 13)
```

In [17]:
```
# Let's print the feature names
print(boston.feature_names)
```

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

In [18]:
```
# Let's print the target vetor of boston dataset
print(boston.target)
```

```
[24.  21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 15.  18.9 21.7 20.4
 18.2 19.9 23.1 17.5 20.2 18.2 13.6 19.6 15.2 14.5 15.6 13.9 16.6 14.8
 18.4 21.  12.7 14.5 13.2 13.1 13.5 18.9 20.  21.  24.7 30.8 34.9 26.6
 25.3 24.7 21.2 19.3 20.  16.6 14.4 19.4 19.7 20.5 25.  23.4 18.9 35.4
 24.7 31.6 23.3 19.6 18.7 16.  22.2 25.  33.  23.5 19.4 22.  17.4 20.9
 24.2 21.7 22.8 23.4 24.1 21.4 20.  20.8 21.2 20.3 28.  23.9 24.8 22.9
 23.9 26.6 22.5 22.2 23.6 28.7 22.6 22.  22.9 25.  20.6 28.4 21.4 38.7
 43.8 33.2 27.5 26.5 18.6 19.3 20.1 19.5 19.5 20.4 19.8 19.4 21.7 22.8
 18.8 18.7 18.5 18.3 21.2 19.2 20.4 19.3 22.  20.3 20.5 17.3 18.8 21.4
 15.7 16.2 18.  14.3 19.2 19.6 23.  18.4 15.6 18.1 17.4 17.1 13.3 17.8
 14.  14.4 13.4 15.6 11.8 13.8 15.6 14.6 17.8 15.4 21.5 19.6 15.3 19.4
 17.  15.6 13.1 41.3 24.3 23.3 27.  50.  50.  50.  22.7 25.  50.  23.8
 23.8 22.3 17.4 19.1 23.1 23.6 22.6 29.4 23.2 24.6 29.9 37.2 39.8 36.2
 37.9 32.5 26.4 29.6 50.  32.  29.8 34.9 37.  30.5 36.4 31.1 29.1 50.
 33.3 30.3 34.6 34.9 32.9 24.1 42.3 48.5 50.  22.6 24.4 22.5 24.4 20.
 21.7 19.3 22.4 28.1 23.7 25.  23.3 28.7 21.5 23.  26.7 21.7 27.5 30.1
 44.8 50.  37.6 31.6 46.7 31.5 24.3 31.7 41.7 48.3 29.  24.  25.1 31.5
 23.7 23.3 22.  20.1 22.2 23.7 17.6 18.5 24.3 20.5 24.5 26.2 24.4 24.8
 29.6 42.8 21.9 20.9 44.  50.  36.  30.1 33.8 43.1 48.8 31.  36.5 22.8
 30.7 50.  43.5 20.7 21.1 25.2 24.4 35.2 32.4 32.  33.2 33.1 29.1 35.1
 45.4 35.4 46.  50.  32.2 22.  20.1 23.2 22.3 24.8 28.5 37.3 27.9 23.9
 21.7 28.6 27.1 20.3 22.5 29.  24.8 22.  26.4 33.1 36.1 28.4 33.4 28.2
 22.8 20.3 16.1 22.1 19.4 21.6 23.8 16.2 17.8 19.8 23.1 21.  23.8 23.1
 20.4 18.5 25.  24.6 23.  22.2 19.3 22.6 19.8 17.1 19.4 22.2 20.7 21.1
 19.5 18.5 20.6 19.  18.7 32.7 16.5 23.9 31.2 17.5 17.2 23.1 24.5 26.6
 22.9 24.1 18.6 30.1 18.2 20.6 17.8 21.7 22.7 22.6 25.  19.9 20.8 16.8
 21.9 27.5 21.9 23.1 50.  50.  50.  50.  50.  13.8 13.8 15.  13.9 13.3
 13.1 10.2 10.4 10.9 11.3 12.3  8.8  7.2 10.5  7.4 10.2 11.5 15.1 23.2
  9.7 13.8 12.7 13.1 12.5  8.5  5.   6.3  5.6  7.2 12.1  8.3  8.5  5.
 11.9 27.9 17.2 27.5 15.  17.2 17.9 16.3  7.   7.2  7.5 10.4  8.8  8.4
 16.7 14.2 20.8 13.4 11.7  8.3 10.2 10.9 11.   9.5 14.5 14.1 16.1 14.3
 11.7 13.4  9.6  8.7  8.4 12.8 10.5 17.1 18.4 15.4 10.8 11.8 14.9 12.6
 14.1 13.  13.4 15.2 16.1 17.8 14.9 14.1 12.7 13.5 14.9 20.  16.4 17.7
 19.5 20.2 21.4 19.9 19.  19.1 19.1 20.1 19.9 19.6 23.2 29.8 13.8 13.3
 16.7 12.  14.6 21.4 23.  23.7 25.  21.8 20.6 21.2 19.1 20.6 15.2  7.
  8.1 13.6 20.1 21.8 24.5 23.1 19.7 18.3 21.2 17.5 16.8 22.4 20.6 23.9
 22.  11.9]
```

In [19]:

```
# Let's check the shape of target
print(boston.target.shape)
```

```
(506,)
```

- The prices of the house indicated by the variable `MEDV` is our target variable and the remaining are the feature variables based on which we will predict the value of a house.

In [25]:
```
# Let's load the data (features matrix) into pandas DataFrame

boston_df = pd.DataFrame(boston.data, columns=boston.feature_names)

# Print the DataFrame
boston_df
```

Out[25]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
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	4.98
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	9.14
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	4.03
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	2.94
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	5.33
...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0.0	0.573	6.593	69.1	2.4786	1.0	273.0	21.0	391.99	9.67
502	0.04527	0.0	11.93	0.0	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	396.90	9.08
503	0.06076	0.0	11.93	0.0	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	396.90	5.64
504	0.10959	0.0	11.93	0.0	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	393.45	6.48
505	0.04741	0.0	11.93	0.0	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	396.90	7.88

506 rows × 13 columns

- We can see that the target label `MEDV` is missing in the DataFrame (that is alright).
- Now, create a new column for target label and add it to the datafram

In [26]:
```
# Let's add target label into pandas DataFrame

boston_df['MEDV'] = boston.target

# Print the DataFrame
boston_df
```

Out[26]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
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	4.98	24.0
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	9.14	21.6
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	4.03	34.7
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	2.94	33.4
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	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0.0	0.573	6.593	69.1	2.4786	1.0	273.0	21.0	391.99	9.67	22.4
502	0.04527	0.0	11.93	0.0	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	396.90	9.08	20.6
503	0.06076	0.0	11.93	0.0	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	396.90	5.64	23.9
504	0.10959	0.0	11.93	0.0	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	393.45	6.48	22.0
505	0.04741	0.0	11.93	0.0	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	396.90	7.88	11.9

506 rows × 14 columns

Data preprocessing

- After loading the data, it's a good practice to see if there are any missing values in the data.
- We count the number of missing values for each feature using `isnull()`.

In [27]:
```
# Return number of missing values for each column
boston_df.isnull().sum()
```

Out[27]:
```
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
MEDV       0
dtype: int64
```

- As we can see, it doesn't show null values. But when we look at the `boston_df` from above, we can see that there are values of 0 which can also be missing values.
- For good measure, we'll turn the 0 values into `np.nan` where we can see what is missing.
- A rule of thumb to drop columns that are missing 70-75% of their data. If it consists of 20-25%, then there may be some hope and opportunity to finagle with filling the values in.

In [76]:
```
# First replace the 0 values with np.nan values
boston_df.replace(0, np.nan, inplace=True)
boston_df
```

Out[76]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	NaN	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	NaN	7.07	NaN	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	NaN	7.07	NaN	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	NaN	2.18	NaN	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	NaN	2.18	NaN	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	NaN	11.93	NaN	0.573	6.593	69.1	2.4786	1.0	273.0	21.0	391.99	9.67	22.4
502	0.04527	NaN	11.93	NaN	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	396.90	9.08	20.6
503	0.06076	NaN	11.93	NaN	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	396.90	5.64	23.9
504	0.10959	NaN	11.93	NaN	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	393.45	6.48	22.0
505	0.04741	NaN	11.93	NaN	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	396.90	7.88	11.9

506 rows × 14 columns

In [78]:

```
# Check what percentage of each column's data is missing
boston_df.isnull().sum()/len(boston_df)
```

Out[78]:
```
CRIM       0.000000
ZN         0.735178
INDUS      0.000000
CHAS       0.930830
NOX        0.000000
RM         0.000000
AGE        0.000000
DIS        0.000000
RAD        0.000000
TAX        0.000000
PTRATIO    0.000000
B          0.000000
LSTAT      0.000000
MEDV       0.000000
dtype: float64
```

- As we can see that `73%` of the `ZN` feature and `93%` of `CHAS` feature are missing.
- S, we will leave them out from our features variables to test as they do not give us enough information for our regression model to interpret.

In [79]:
```
# Drop ZN and CHAS columns, as they contain too many missing values
boston_df = boston_df.drop('ZN', axis=1)
boston_df = boston_df.drop('CHAS', axis=1)

# let's see the DataFrame now
boston_df
```

Out[79]:

	CRIM	INDUS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	2.31	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	7.07	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	7.07	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	2.18	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	2.18	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	11.93	0.573	6.593	69.1	2.4786	1.0	273.0	21.0	391.99	9.67	22.4
502	0.04527	11.93	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	396.90	9.08	20.6
503	0.06076	11.93	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	396.90	5.64	23.9
504	0.10959	11.93	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	393.45	6.48	22.0
505	0.04741	11.93	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	396.90	7.88	11.9

506 rows × 12 columns

- Now, let's check some basic statitstics using `describe()`.

In [80]:

```
# Return basic statistics summary of the dataset
print(boston_df.describe())
```

```
              CRIM       INDUS         NOX          RM         AGE          DIS  \
count   506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean      3.613524   11.136779    0.554695    6.284634   68.574901    3.795043
std       8.601545    6.860353    0.115878    0.702617   28.148861    2.105710
min       0.006320    0.460000    0.385000    3.561000    2.900000    1.129600
25%       0.082045    5.190000    0.449000    5.885500   45.025000    2.100175
50%       0.256510    9.690000    0.538000    6.208500   77.500000    3.207450
75%       3.677083   18.100000    0.624000    6.623500   94.075000    5.188425
max      88.976200   27.740000    0.871000    8.780000  100.000000   12.126500

              RAD         TAX     PTRATIO           B       LSTAT        MEDV
count   506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean      9.549407  408.237154   18.455534  356.674032   12.653063   22.532806
std       8.707259  168.537116    2.164946   91.294864    7.141062    9.197104
min       1.000000  187.000000   12.600000    0.320000    1.730000    5.000000
25%       4.000000  279.000000   17.400000  375.377500    6.950000   17.025000
50%       5.000000  330.000000   19.050000  391.440000   11.360000   21.200000
75%      24.000000  666.000000   20.200000  396.225000   16.955000   25.000000
max      24.000000  711.000000   22.000000  396.900000   37.970000   50.000000
```

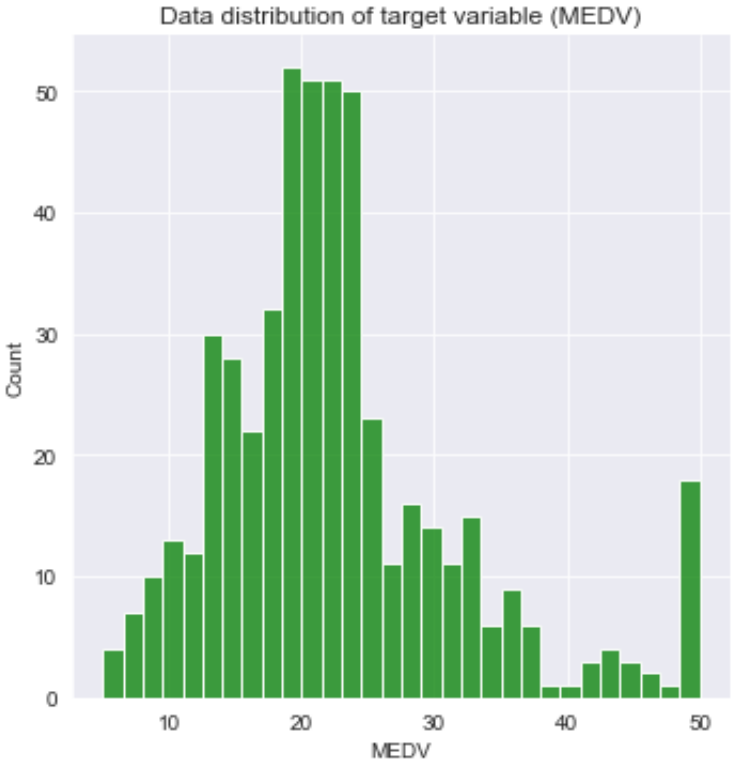
- As we can see, All the variables are Numerical including the target.

Exploratory Data Analysis

- Exploratory Data Analysis is a very important step before training the model.
- we will use some visualizations to understand the relationship of the target variable with other feature variables.
- Let's first plot the distribution of the target variable `MEDV` (price of the house).

In [96]:

```
# Let's use the displot function from the seaborn
# to visualise data distribution of the target variable.

with sns.axes_style('darkgrid'):
    sns.displot(boston_df['MEDV'], bins=30, color='green')
    plt.title("Data distribution of target variable (MEDV)");
```

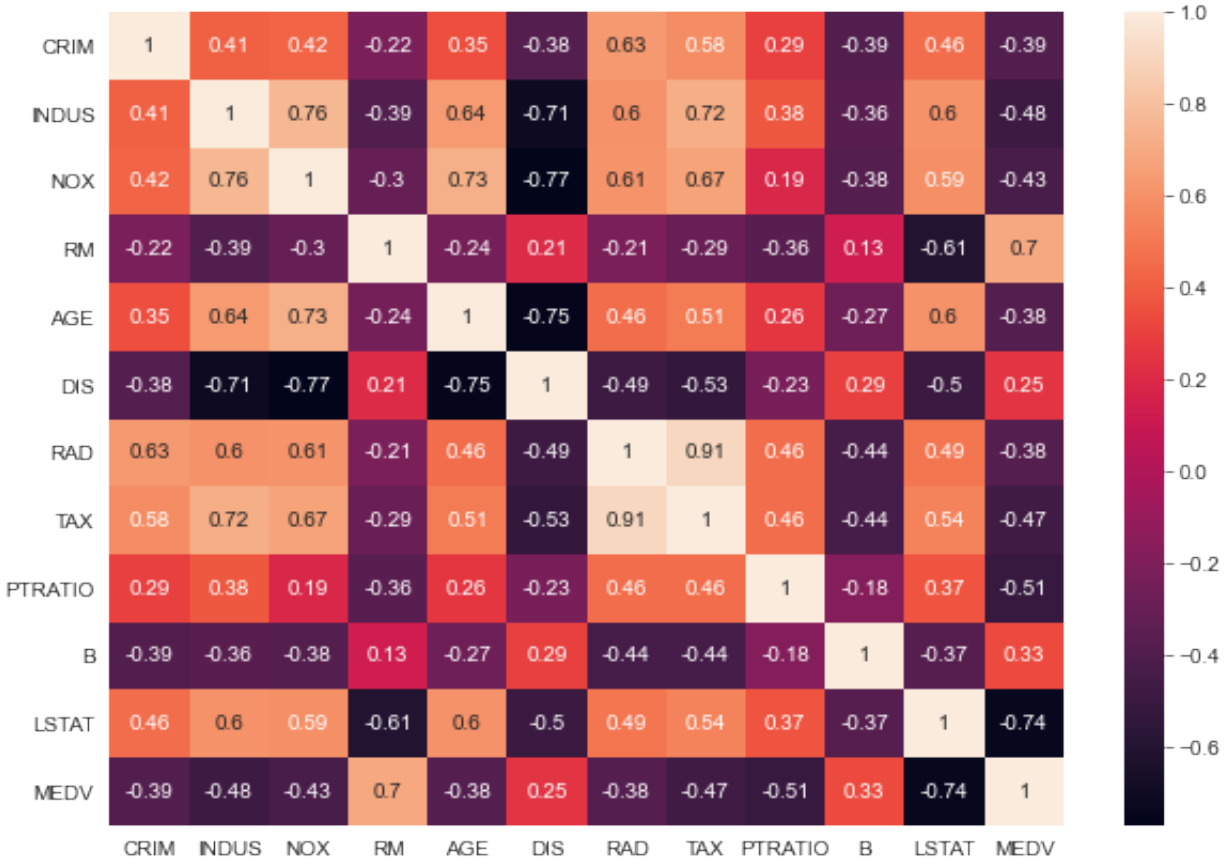


- As we can see, the values of `MEDV` are distributed normally with few outliers.
- Most of the house prices are around 20–24 range.

- Next, we create a correlation matrix that measures the linear relationships between the variables.
- The correlation matrix can be formed by using the `corr` function from the pandas library.
- The correlation coefficient ranges from `−1` to `1`. If the value is close to `1`, it means that there is a *strong positive correlation* between the two variables. When it is close to `−1`, the variables have a *strong negative correlation*.
- Then, we will use the `heatmap` function from the seaborn library to plot the correlation matrix.

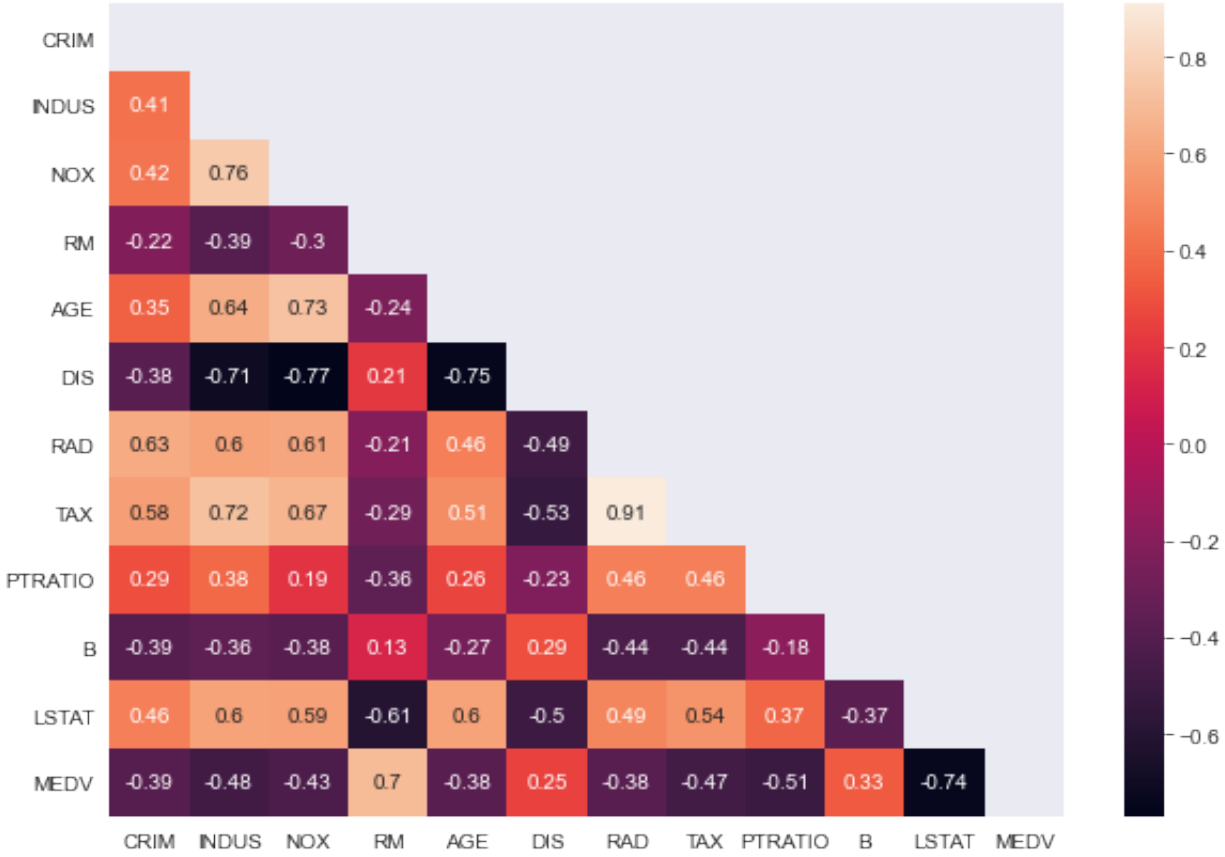
In [108…

```
# corr() to calculate the correlation between variables
correlation_matrix = boston_df.corr().round(2)

# changing the figure size
plt.figure(figsize = (10, 7))
# "annot = True" to print the values inside the square
sns.heatmap(data=correlation_matrix, annot=True);
```



In [110…

```
# Steps to remove redundant values

# Return a array filled with zeros
mask = np.zeros_like(correlation_matrix)
# Return the indices for the upper-triangle of array
mask[np.triu_indices_from(mask)] = True

# changing the figure size
plt.figure(figsize = (10, 7))
# "annot = True" to print the values inside the square
sns.heatmap(data=correlation_matrix, annot=True, mask=mask);
```



- An important point in selecting features for a linear regression model is to check for *Multicollinearity* (It's helpful to see which features increase/decrease together. Features that correlate together may make interpretability of their effectiveness difficult).
- From the heatmap, if we set a cut off for high correlation to be `+- .75`, we can see that:
  - `TAX` and `RAD` have a high correlation of `0.91`
  - `NOX` and `INDUS` have a high correlation of `0.76`
  - `DIS` and `NOX` have a high correlation of `-0.77`
  - `DIS` and `AGE` have a high correlation of `-0.75`
- Therefore, these feature pairs, (`TAX` and `RAD`), (`NOX` and `INDUS`), (`DIS` and `NOX`), and (`DIS` and `AGE`) are strongly correlated to each other. Thus, we should not select these features for training the model. So, we will drop all of these for better accuracy.
- To fit a linear regression model, we select those features which have a high correlation with our target variable `MEDV`.
- By looking at the correlation matrix we can see that `RM` has a strong positive correlation with `MEDV` (0.7) where as `LSTAT` has a high negative correlation with `MEDV` (-0.74).

In [111…

```
# Thus, let's drop correlated varaibles

boston_df = boston_df.drop(['TAX', 'RAD', 'NOX', 'INDUS', 'DIS', 'AGE'], axis=1)

# Now, see the current DataFrame
boston_df
```

Out[111…

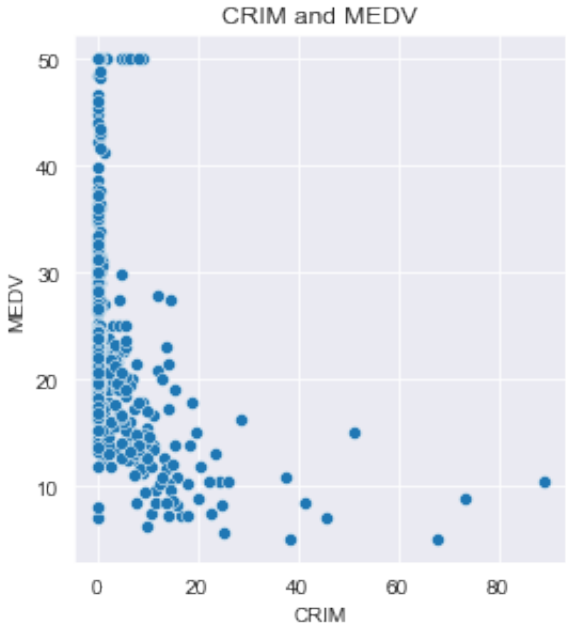
	CRIM	RM	PTRATIO	B	LSTAT	MEDV
0	0.00632	6.575	15.3	396.90	4.98	24.0
1	0.02731	6.421	17.8	396.90	9.14	21.6
2	0.02729	7.185	17.8	392.83	4.03	34.7
3	0.03237	6.998	18.7	394.63	2.94	33.4
4	0.06905	7.147	18.7	396.90	5.33	36.2
...	...	...	...	...	...	...
501	0.06263	6.593	21.0	391.99	9.67	22.4
502	0.04527	6.120	21.0	396.90	9.08	20.6
503	0.06076	6.976	21.0	396.90	5.64	23.9
504	0.10959	6.794	21.0	393.45	6.48	22.0
505	0.04741	6.030	21.0	396.90	7.88	11.9

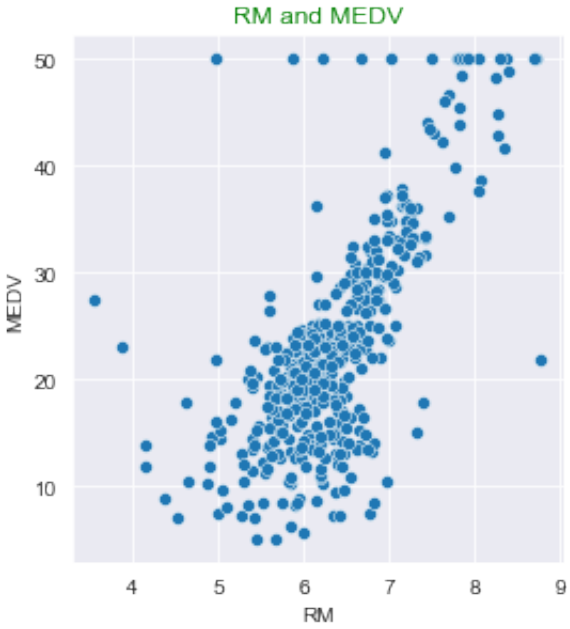
506 rows × 6 columns

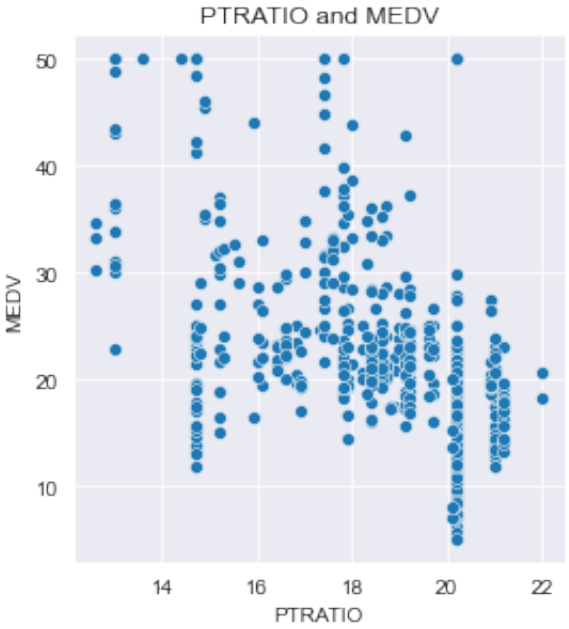
- Now, let's check the relationship between each of the remaining feature variables and the target variable `MEDV`.
- This wiil help us to see which features have linear relationships with the target variable.

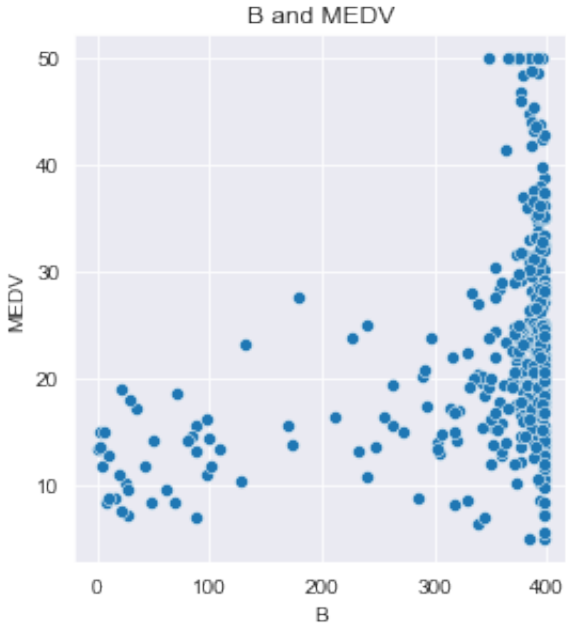
In [148…

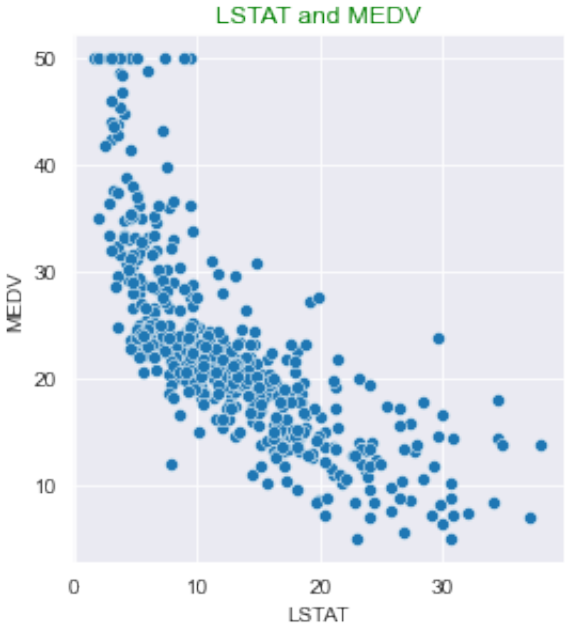
```
# Let's create plots to see the relationships
# between feature variables and target varable

plt.figure(figsize=(14,10))

# Relatiosnship with 'CRIM' and 'MEDV'
plt.subplot(2, 3, 1)
sns.scatterplot(x=boston_df['CRIM'], y=boston_df['MEDV'])
plt.title("CRIM and MEDV")

# Relatiosnship with 'RM' and 'MEDV'
plt.subplot(2, 3, 2)
sns.scatterplot(x=boston_df['RM'], y=boston_df['MEDV'])
plt.title("RM and MEDV", color='green')

# Relatiosnship with 'PTRATIO' and 'MEDV'
plt.subplot(2, 3, 3)
sns.scatterplot(x=boston_df['PTRATIO'], y=boston_df['MEDV'])
plt.title("PTRATIO and MEDV")

# Relatiosnship with 'B' and 'MEDV'
plt.subplot(2, 3, 4)
sns.scatterplot(x=boston_df['B'], y=boston_df['MEDV'])
plt.title("B and MEDV")

# Relatiosnship with 'LSTAT' and 'MEDV'
plt.subplot(2, 3, 5)
sns.scatterplot(x=boston_df['LSTAT'], y=boston_df['MEDV']);
plt.title("LSTAT and MEDV", color='green');
```


CRIM and MEDV
MEDV
CRIM
0
20
40
60
80
10
20
30
40
50


RM and MEDV
MEDV
RM
4
5
6
7
8
9
10
20
30
40
50


PTRATIO and MEDV
MEDV
PTRATIO
14
16
18
20
22
10
20
30
40
50


B and MEDV
MEDV
B
0
100
200
300
400
10
20
30
40
50


LSTAT and MEDV
MEDV
LSTAT
0
10
20
30
10
20
30
40
50

- As we can see, `RM` and `LSTAT` look like the only ones that have linear relationship with `MEDV`.
- That means,
  - The prices increase as the value of `RM` increases linearly.
  - The prices tend to decrease with an increase in `LSTAT`. Though it doesn't look to be following exactly a linear line.
- Just for reference:
  - MEDV = Median value of house
  - RM = Avg number of rooms per house
  - LSTAT = Neighborhood Affluence

Create Model

- First, create the features matrix and the target variable.

In [159…

```
# Feature matrix
X = boston_df[['RM', 'LSTAT']]
X
```

Out[159…

	RM	LSTAT
0	6.575	4.98
1	6.421	9.14
2	7.185	4.03
3	6.998	2.94
4	7.147	5.33
...	...	...
501	6.593	9.67
502	6.120	9.08
503	6.976	5.64
504	6.794	6.48
505	6.030	7.88

506 rows × 2 columns

In [160…

```
# Target variable
y = boston_df['MEDV']
y
```

Out[160…

```
0      24.0
1      21.6
2      34.7
3      33.4
4      36.2
       ...
501    22.4
502    20.6
503    23.9
504    22.0
505    11.9
Name: MEDV, Length: 506, dtype: float64
```

- Second, create training and testing set.

In [209…

```
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 = 2)

print("X_train shape: ", X_train.shape)
print("X_test shape: ", X_test.shape)
print("y_train shape: ", y_train.shape)
print("y_test shape: ", y_test.shape)
```

```
X_train shape:  (404, 2)
X_test shape:  (102, 2)
y_train shape:  (404,)
y_test shape:  (102,)
```

- Now we instantiate a Linear Regression object, fit the training data and then predict.

In [210…

```
from sklearn.linear_model import LinearRegression

# Create LinearRegression Instance
lrm = LinearRegression()

# Fit data on to the model
lrm.fit(X_train, y_train)

# Predict
y_predicted = lrm.predict(X_test)
```

Evaluate Model

- Now, let's evaluate how well our model did using metrics *r-squared* and *root mean squared error (rmse)*.
- The *r-squared* value shows how strong our features determined the target value.
- The *rmse* defines the difference between predicted and the test values.
- The higher the value of the *rmse*, the less accurate the model.

In [211…

```
from sklearn.metrics import mean_squared_error

# calculate r-squared values
r2 = lrm.score(X_test, y_test)

# calculate the difference between predicted and the test values.
rmse = (np.sqrt(mean_squared_error(y_test, y_predicted)))

# print the both values
print('-----------------------------------------')
print('r-squared: {}'.format(r2))
print('-----------------------------------------')
print('root mean squared error: {}'.format(rmse))
print('-----------------------------------------')
```

```
-----------------------------------------
r-squared: 0.7175039832076542
-----------------------------------------
root mean squared error: 4.861432596429275
-----------------------------------------
```

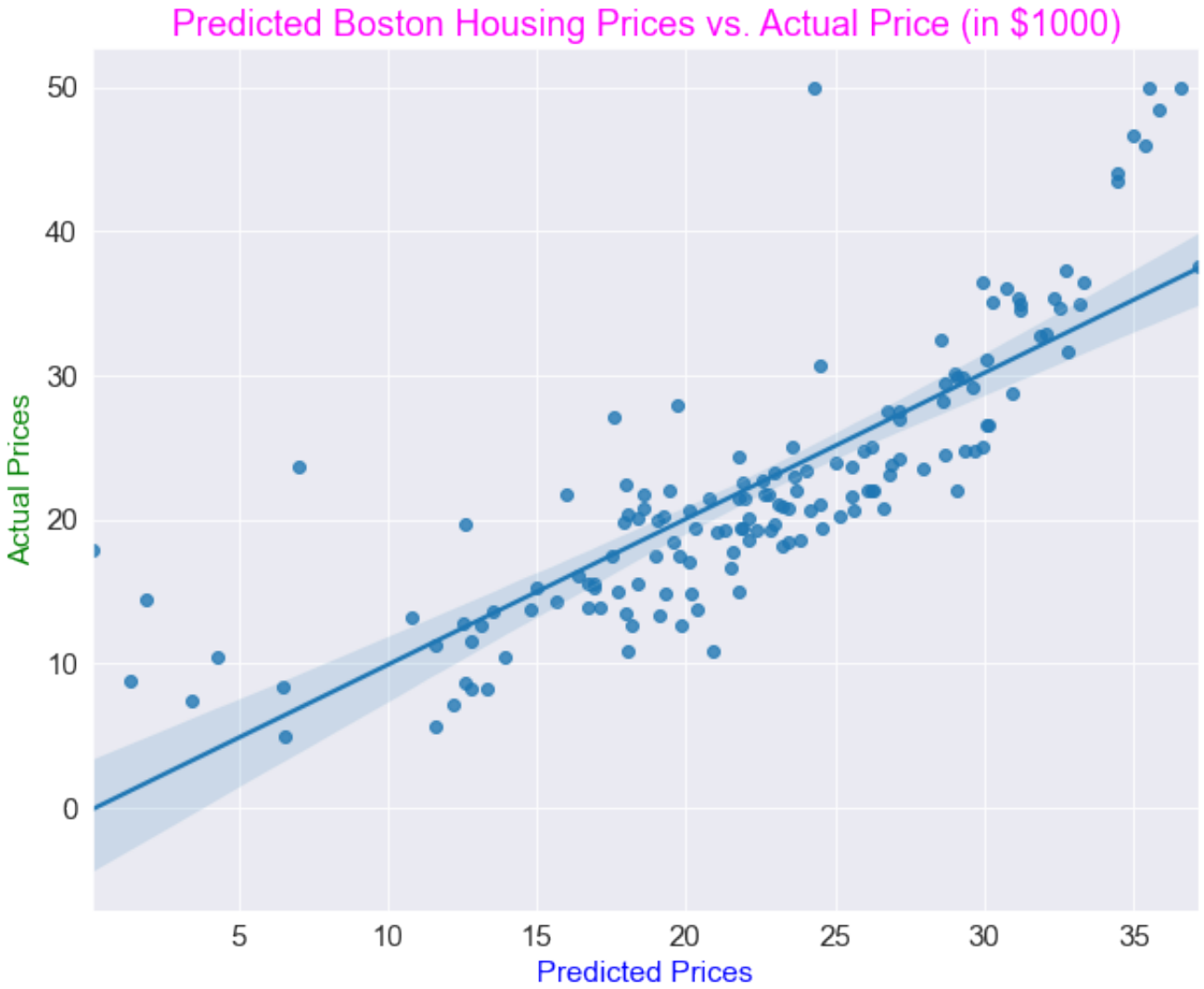
- With an *r-squared* value of `0.72`, the model is good enough but it's not perfect.
- This could be improved by:
  - bigger dataset
  - better features or maybe more features
  - remove outliers
- With an *root mean squared error (rmse)* value of `4.86`, we can interpret that on average we are **4.86k** dollars off the actual value.

Predicted Price vs Actual Price

- we can see how our model is predicting by plotting a scatter plot between the original house price and predicted house prices.

In [208…

```
# Let's plot predictions vs actual

plt.figure(figsize=(10,8))

sns.regplot(x=y_predicted, y=y_test)

plt.xlabel('Predicted Prices', fontsize=15, color='blue')
plt.ylabel('Actual Prices', fontsize=15, color='green')
plt.title("Predicted Boston Housing Prices vs. Actual Price (in $1000)",
          fontsize=18, color='magenta');
```



- The closer we can get the points to be at the 0 line, the more accurate the model is at predicting the prices.
