Machine Learning & Statistics - Assessment 2019

Justin Rutherford (13/12/19)

Foreword

An assessment of the 'Boston House Price' dataset contained in the jupyter notebook below is the assignment submission for the 4th semester 5 credit module - **Machine Learning & Statistics**, part of the course entitled *Higher Diploma in Science - Computing (Data Analytics)*, submitted to Dr. Ian McLoughlin, lecturer and Programme Administrator at GMIT. The analyses below are as per the assignment direction and are answered in the order in which they were presented. A number of sources of reference material was used and reviewed in this analysis, and a list of referenced sites is found at the end of the page.

The Assement Instructions

1. Describe - Use descriptive statistics and plots to describe the Boston House Price dataset.
2. Infer - Use inferential statistics to analyse whether there is a significant difference between median house prices for houses along the Charles River and those that are not.
3. Predict - Create a neural network that can predict the median house price based on the other variables in the dataset.

Executive Report

Section 1 - Describe.

Use descriptive statistics and plots to describe the Boston House Price dataset.

The Boston House Price dataset was compiled by Harrison and Rubinfed in 1978 for the purpose establishing whether or not clean air influenced the value of houses in Boston. They collected and analysed data under 14 categories. For convenience we have listed the categories below.

1. CRIM - per capita crime rate per town
2. ZN - proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS - proportion of non-retail business acres per town
4. CHAS - Charles River dummy variable (1 if tract bounds river; else 0)
5. NOX – nitric oxides concentration (parts per 10 million)
6. RM – average number of rooms per dwelling
7. AGE – proportion of owner-occupied units built prior to 1940
8. DIS – weighted distances to five Boston employment centres
9. RAD – index of accessibility to radial highways
10. TAX – full-value property-tax rate per per 10,000
11. PT - pupil teacher ratio by town
12. B – 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
13. LSTAT - % lower status of the population
14. MEDV – Median value of owner-occupied homes in 1000’s.

We downloaded the dataset from the following resource (https://github.com/selva86/datasets/blob/master/BostonHousing.csv).

We then viewed the dataset by importing it as a dataframe using the pandas package for data analytics. With the benefit of having named the categories in line with the instructions, we did preliminary checks to see that the dataset we had obtained was complete and intact.

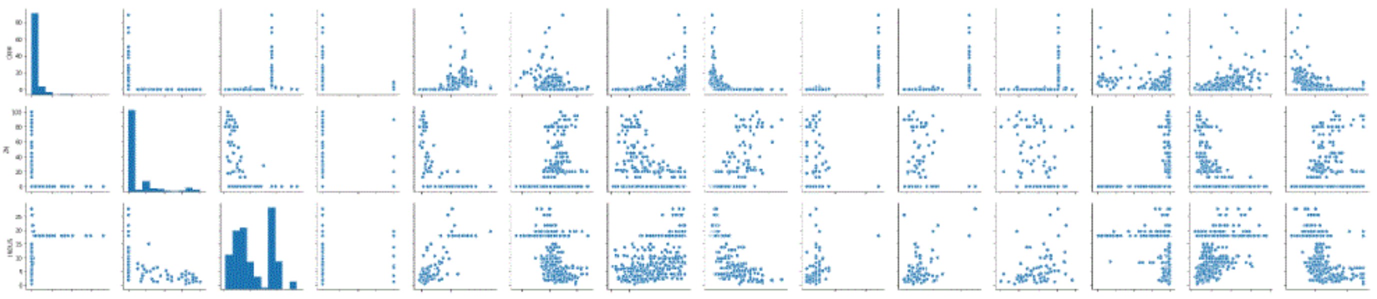
We confirmed this using the dataframe 'info' function.

```
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
CRIM       506 non-null float64
ZN         506 non-null float64
INDUS      506 non-null float64
CHAS       506 non-null int64
NOX        506 non-null float64
RM         506 non-null float64
AGE        506 non-null float64
DIS        506 non-null float64
RAD        506 non-null int64
TAX        506 non-null int64
PTRATIO    506 non-null float64
B          506 non-null float64
LSTAT      506 non-null float64
MEDV       506 non-null float64
dtypes: float64(11), int64(3)
```

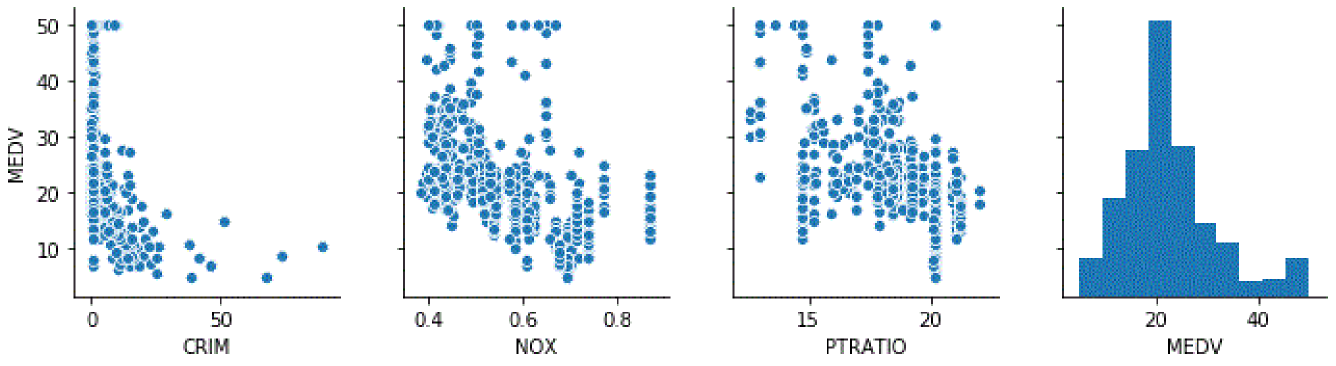
We then checked for duplicates and run some decriptive analyitics on the data to get a view of the distributions etc.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000
75%	3.677082	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000

We used the matplotlib and seaborn libraries to illustrate plots of all data for comparison using pairplot. However given the 14 categories of data this was useful to pick out some categories that we could investigate futher.



Having viewed full set of plots, we selected a subset of the data which we thought was interesting to see if we could detect correlations from graphs.



We can clearly see from the plots the MEDV category is higly influenced by factors such as crime, but not so by NOX values and less so by pupil-teacher ratios. To investigate the dataset further we analysed the maximum and minimum values for each of 'MEDV', 'PTRATIO', and 'CRIM'.

```
min_max_values('MEDV')
```

	161	398
CRIM	1.46336	38.3518
ZN	0.00000	0.0000
INDUS	19.58000	18.1000
CHAS	0.00000	0.0000
NOX	0.60500	0.6930
RM	7.48900	5.4530
AGE	90.80000	100.0000
DIS	1.97090	1.4896
RAD	5.00000	24.0000
TAX	403.00000	666.0000
PTRATIO	14.70000	20.2000
B	374.43000	396.9000
LSTAT	1.73000	30.5900
MEDV	50.00000	5.0000

```
min_max_values('PTRATIO')
```

	354	196
CRIM	0.04301	0.04011
ZN	80.00000	80.00000
INDUS	1.91000	1.52000
CHAS	0.00000	0.00000
NOX	0.41300	0.40400
RM	5.66300	7.28700
AGE	21.90000	34.10000
DIS	10.58570	7.30900
RAD	4.00000	2.00000
TAX	334.00000	329.00000
PTRATIO	22.00000	12.60000
B	382.80000	396.90000
LSTAT	8.05000	4.08000
MEDV	18.20000	33.30000

```
min_max_values('CRIM')
```

	380	0
CRIM	88.9762	0.00632
ZN	0.0000	18.00000
INDUS	18.1000	2.31000
CHAS	0.0000	0.00000
NOX	0.6710	0.53800
RM	6.9680	6.57500
AGE	91.9000	65.20000
DIS	1.4165	4.09000
RAD	24.0000	1.00000
TAX	666.0000	296.00000
PTRATIO	20.2000	15.30000
B	396.9000	396.90000
LSTAT	17.2100	4.98000
MEDV	10.4000	24.00000

From the analysis above we made the following observations;

1. The max MEDV occurs where crime is very low and the contrary value holds true.
2. The tax rate is lower for the highest MEDV than the lowest MEDV.
3. PTRATIO max and min values have equal crime rates, NOx emmissions, and Tax values.
4. For crime min and max values, high levels of industry and retail businesses reflect high crime rates, and access to highways.
5. The Charles River variable was 'zero' for all max and min values in the subset of data we looked at.

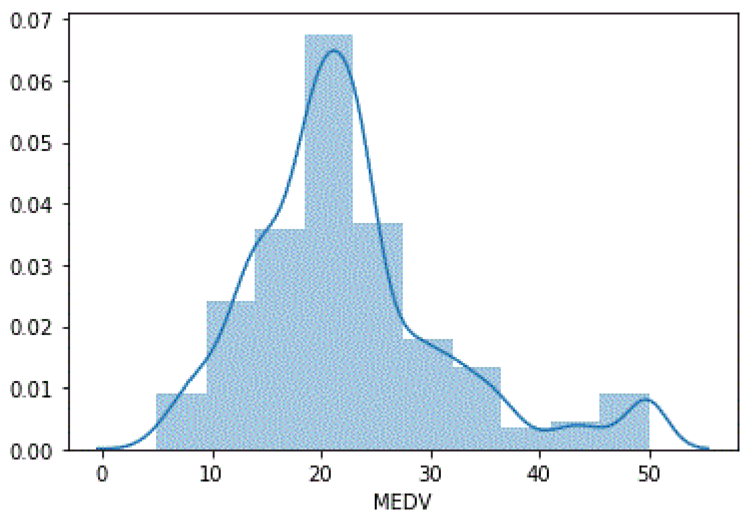
We then sorted the data selecting the top values for MEDV to see if we could see a trend in the data.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
283	0.01501	90.0	1.21	1	0.401	7.923	24.8	5.8850	1	198	13.6	395.52	3.16	50.0
225	0.52693	0.0	6.20	0	0.504	8.725	83.0	2.8944	8	307	17.4	382.00	4.63	50.0
369	5.66998	0.0	18.10	1	0.631	6.683	96.8	1.3567	24	666	20.2	375.33	3.73	50.0
370	6.53876	0.0	18.10	1	0.631	7.016	97.5	1.2024	24	666	20.2	392.05	2.96	50.0
371	9.23230	0.0	18.10	0	0.631	6.216	100.0	1.1691	24	666	20.2	366.15	9.53	50.0

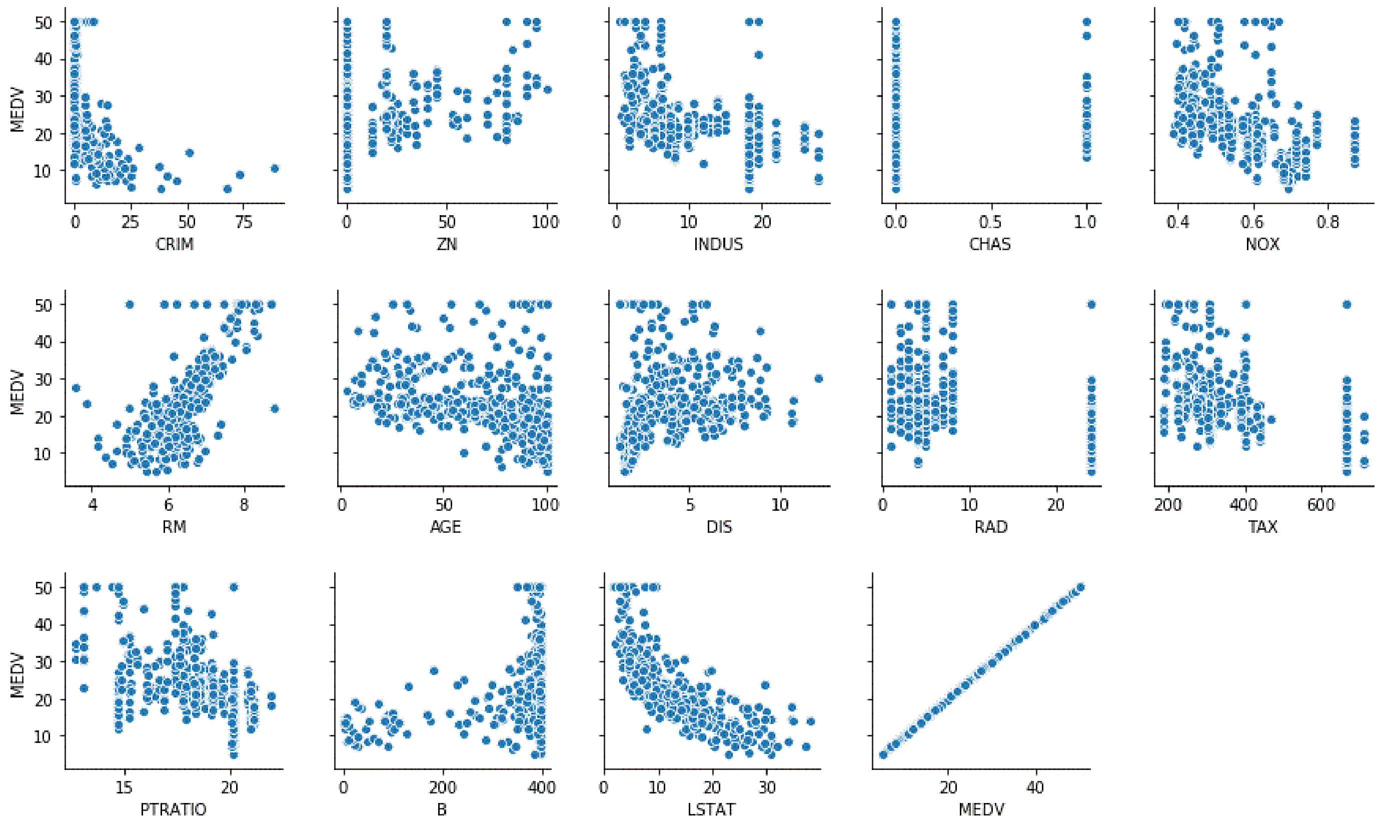
This was inconclusive as we see variances in the data for the other variables, other than the crime variable which is only consistenly low for high MEDV values. We sorted the dataset on the Charles River variable to see what was returned where the variable indicated proximity to the river.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
160	1.27346	0.0	19.58	1	0.605	6.250	92.6	1.7984	5	403	14.7	338.92	5.50	27.0
358	5.20177	0.0	18.10	1	0.770	6.127	83.4	2.7227	24	666	20.2	395.43	11.48	22.7
210	0.17446	0.0	10.59	1	0.489	5.960	92.1	3.8771	4	277	18.6	393.25	17.27	21.7
236	0.52058	0.0	6.20	1	0.507	6.631	76.5	4.1480	8	307	17.4	388.45	9.54	25.1
152	1.12658	0.0	19.58	1	0.871	5.012	88.0	1.6102	5	403	14.7	343.28	12.12	15.3

We note a relative spread of values and in particular the MEDV values are close to the mean of the distribution. We then analysed this futher by plotting the distribution of the MEDV data.



To view the relationship between the MEDV value and the other 13 categories we plotted the values for the complete data, so as to not skew the values (as would be inferred from our discrete selections above e.g. max/min).

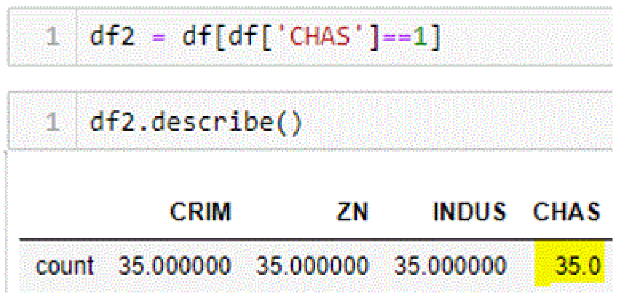


The correlation between MEDV and CRIM and LSTAT, are noted by the downward trending graphs indicating a negative correlation. Other variables such as NOX, RAD, AGE do not indicate any correlation.

Section 2 - Infer.

Use inferential statistics to analyse whether there is a significant difference between median house prices for houses along the Charles River and those that are not.

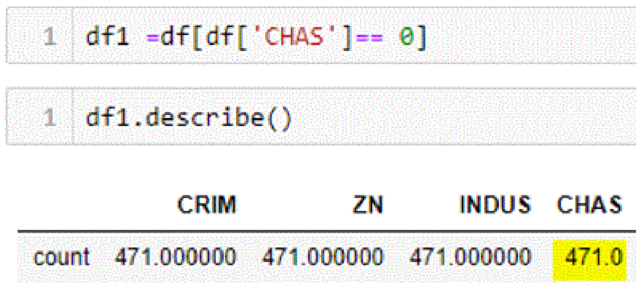
We analysed the Charles River data and found that the data was somewhat skewed in favour of properties not bounding the river. We note that the instances of houses bounding the river numbers 35, whereas the number of properities not bounding the river in the dataset is 471. Therefore the sample dataset has a bias as the population quantum for median house prices is over 13 times larger for properties not bounding the Charles River than those that are.



```
df2 = df[df['CHAS']==1]
```

```
df2.describe()
```

	CRIM	ZN	INDUS	CHAS
count	35.000000	35.000000	35.000000	35.0



```
df1 =df[df['CHAS']== 0]
```

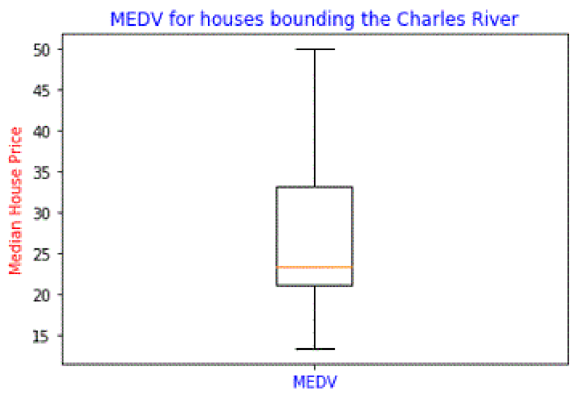
```
df1.describe()
```

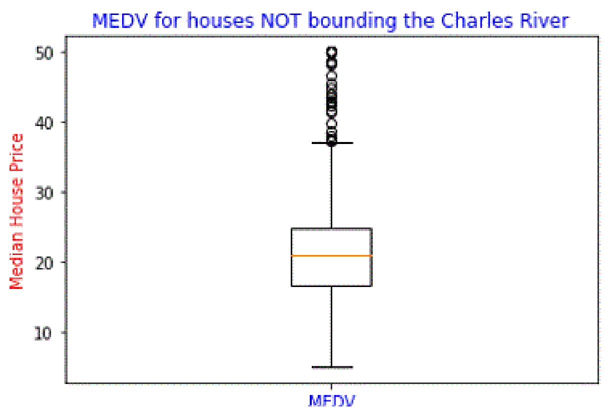
	CRIM	ZN	INDUS	CHAS
count	471.000000	471.000000	471.000000	471.0

Now that we understand that there is a bias in terms of the population size, and that the relative size of the sub-groups may influence a determination, we looked at the distributions of the data sub-categorised by the Charles River dummy variable. To visualise this we used a box-plot to illustrate the middle 50 percent of the data values, also known as the interquartile range. The median of the values is depicted by the line splitting the box in half. The IQR illustrates the variability in a set of values. A large IQR indicates a large spread in values, while a smaller IQR indicates most values fall near the center. Box plots also illustrate the minimum and maximum data values through whiskers extending from the box, and outliers as points extending beyond the whiskers.[Ref (https://pro.arcgis.com/en/pro-app/help/analysis/geoprocessing/charts/box-plot.htm)]

Outliers are plotted as individual data points. These are individual data points that are beyond the max or min values are could be considered as erroneous values, subject to further investigation.

The box-and-whisker plot is useful for revealing the central tendency and variability of a data set, the distribution (particularly symmetry or skewness) of the data, and the presence of outliers. It is also a powerful graphical technique for comparing samples from two or more different treatments or populations.



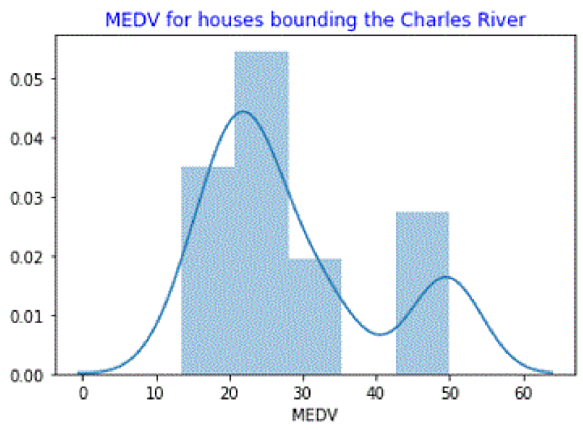


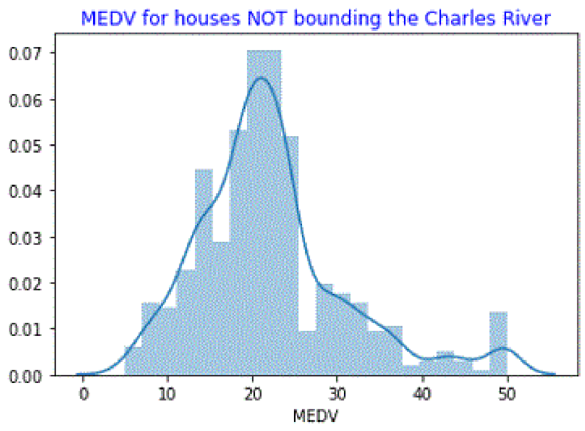
```
count    35.000000
mean     28.440000
std      11.816643
min      13.400000
25%      21.100000
50%      23.300000
75%      33.150000
max      50.000000
Name: MEDV, dtype: float64
```

```
count    471.000000
mean      22.093843
std        8.831362
min        5.000000
25%       16.600000
50%       20.900000
75%       24.800000
max       50.000000
Name: MEDV, dtype: float64
```

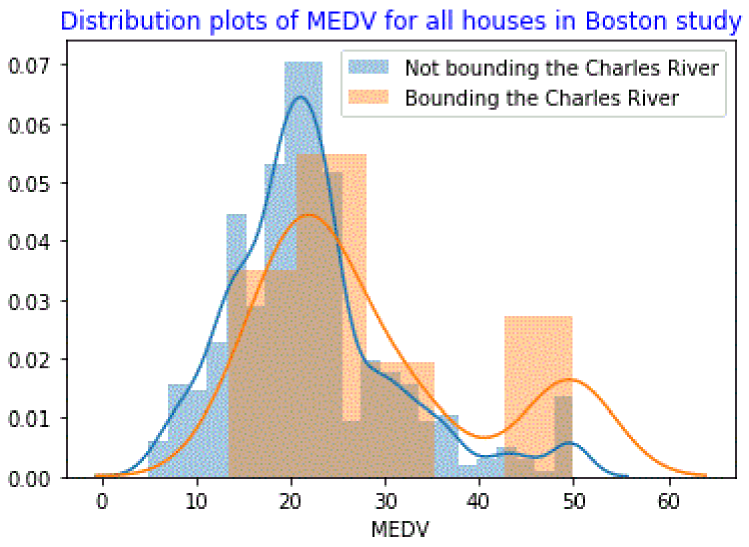
From a comparison of the two boxplots above, we note that the median values (MEDV) are relatively similar, but that we see a greater variablility in the interquartile range for houses bounding the Charles River. However, this is due to the inherent assumption in the box-plot that the house values above approximately $36,000 are to be considered as outliers. Given that we consider this dataset to be a factual respresenation of data collected we cannot therefore exclude the 'outliers'.

To further analyse this we will need to plot the distributions of the subsets using a histogram to provide a more accurate representation of the distribution, without exclusions.





The histogram's provide a much more accurate respresention of the subsets. We can clearly see that within both populations, we have a range house price values that are similar. In both populations we can see that the overall distribution is similar and that within each we can identify a 'bin' of house prices we roughly estimate between $43,000 and $50,000. Due to the relative size of the populations, the influence of a single bin of the high MEDV houses is more profound in the dataset for houses bounding the Charles River.



When we overlay the histograms, we see that the distribution and probability density functions are relatively similar for house bounding the Charles river and those that do not.

We ran a 'T-Test' to confirm our obsverations.

```python
import scipy.stats as ss
ss.ttest_ind(df1['MEDV'], df2['MEDV'])
```

```
Ttest_indResult(statistic=-3.996437466090509, pvalue=7.390623170519905e-05)
```

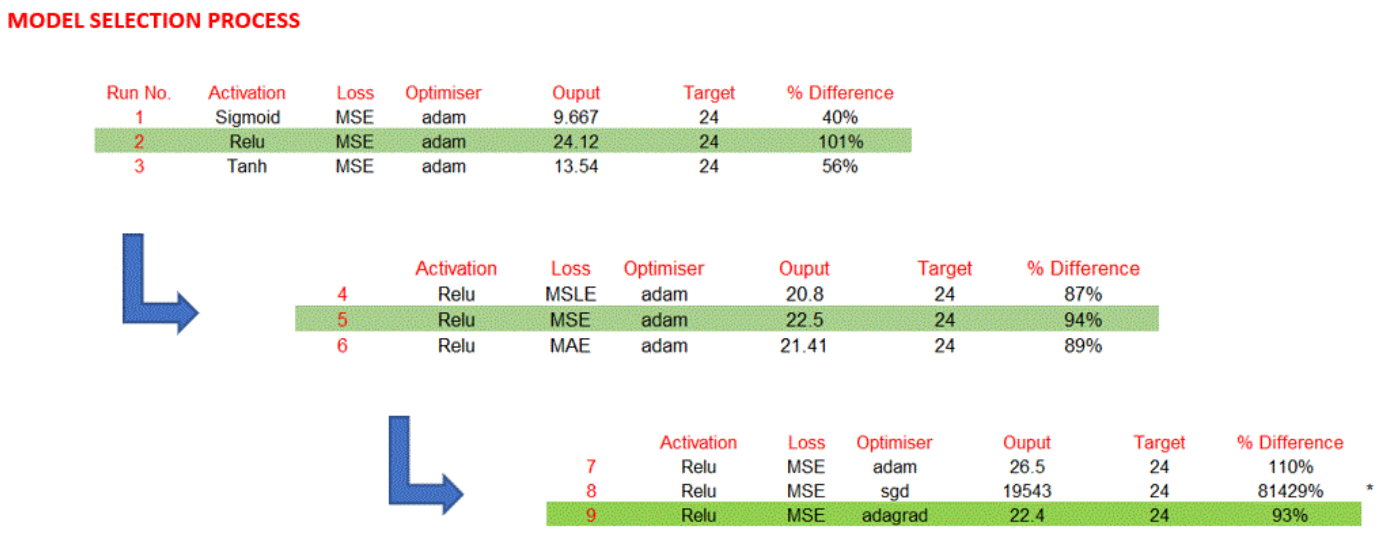
The resultant value was below our predetermined level to consider the Null Hypothesis as void, although this would need further investigation which is beyond the scope of this report, it assists us in making our determination.

In conclusion we find that there is not a significant difference in median house prices for houses along the Charles River and those that are not.

Section 3 - Predict.

Create a neural network that can predict the median house price based on the other variables in the dataset.

For the purpose of efficient building, training and testing of multiple neural networks we imported the Keras library. Due to its structure, we can build a neural network or multiple versions of a network with various selections of cost functions, optimisers, activation functions, and as many internal and hidden layers as we choose. For the purpose of this report we will design our neural network model by running scenarios where we change one of the following inputs for each run; Activation Function, Loss Function, and Optimiser. Having selected the Input dataset (i.e. all categories except the MEDV) and the Output dataset (i.e the MEDV category) we then follow the following process to select the functions which give us the least error, defined by percentage difference over our target (known) value.



MODEL SELECTION PROCESS

Run No.	Activation	Loss	Optimiser	Ouput	Target	% Difference
1	Sigmoid	MSE	adam	9.667	24	40%
2	Relu	MSE	adam	24.12	24	101%
3	Tanh	MSE	adam	13.54	24	56%

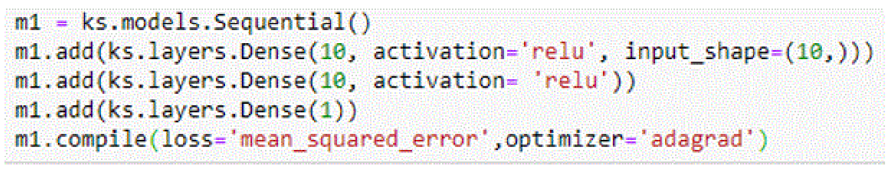
	Activation	Loss	Optimiser	Ouput	Target	% Difference
4	Relu	MSLE	adam	20.8	24	87%
5	Relu	MSE	adam	22.5	24	94%
6	Relu	MAE	adam	21.41	24	89%

	Activation	Loss	Optimiser	Ouput	Target	% Difference	
7	Relu	MSE	adam	26.5	24	110%	
8	Relu	MSE	sgd	19543	24	81429%	*
9	Relu	MSE	adagrad	22.4	24	93%	

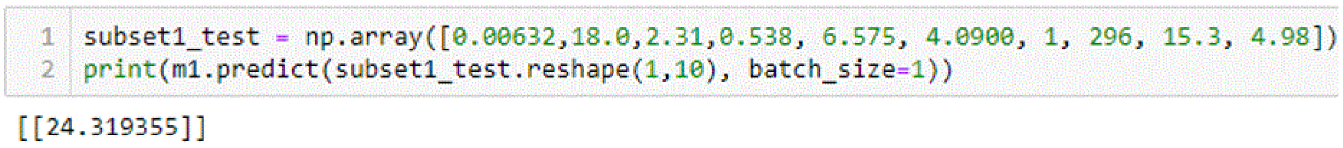
We use the following acronyms aoove; Mean Square Logarithmic Error (MSLE), Mean Square Error (MSE), Mean Absolute Error (MAE), Stochastic Gradient Descent (SGD), Rectified Linear Unit (ReLU). Based on the results obtained from multiple runs of the model, we choose 'Relu' as the most appropriate activation function, Mean Square Error for the loss function and adagrad as the Optimiser.

Based on the selected design and from multiple runs of the neural network, the predicted value error was between 1% and 7%.

For the model building exercise, we incorporated the full dataset in selecting the most suitable functions. However, to investigate whether or not a subset of the dataset would produce a more accurate result, we removed 'CHAS', 'AGE', 'B', from the input dataset.



```
m1 = ks.models.Sequential()
m1.add(ks.layers.Dense(10, activation='relu', input_shape=(10,)))
m1.add(ks.layers.Dense(10, activation= 'relu'))
m1.add(ks.layers.Dense(1))
m1.compile(loss='mean_squared_error',optimizer='adagrad')
```

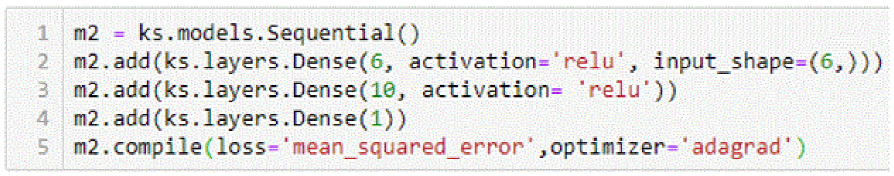


```
subset1_test = np.array([0.00632,18.0,2.31,0.538, 6.575, 4.0900, 1, 296, 15.3, 4.98])
print(m1.predict(subset1_test.reshape(1,10), batch_size=1))
```

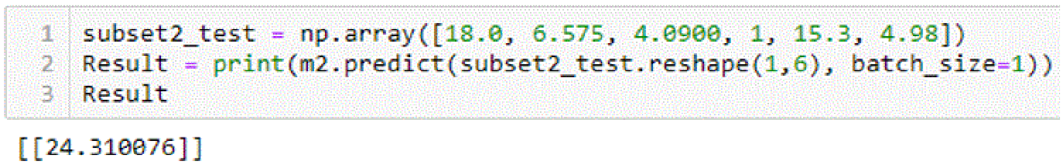
```
[[24.319355]]
```

The result obtained from modelling the subset of data was an improved result (i.e. <1%) over the prediction error obtained when using the full dataset. We therefore conclude that this dataset can be optimised by removing unnecessary categories.

We then reduced the dataset further by additionally removing the following categories: 'CRIM', 'INDUS', 'NOX'.



```
m2 = ks.models.Sequential()
m2.add(ks.layers.Dense(6, activation='relu', input_shape=(6,)))
m2.add(ks.layers.Dense(10, activation= 'relu'))
m2.add(ks.layers.Dense(1))
m2.compile(loss='mean_squared_error',optimizer='adagrad')
```



```
subset2_test = np.array([18.0, 6.575, 4.0900, 1, 15.3, 4.98])
Result = print(m2.predict(subset2_test.reshape(1,6), batch_size=1))
Result
```

```
[[24.310076]]
```

The reduced subset returned a negligable improvement over the previous iteration of the model run and we consider that the datset reduction is sufficient.

However, although the results obtained above had a very small level of error, we considered whether or not a pre-processed dataset would improve the result. For this we normalised, scaled and used data whitening on the full dataset. The purpose of whitening was to remove the influenece of correlation. For this task we imported the pre-processing libary from SciKit-Learn. We re-ran the model using the same input/output functions as adopted earlier,however we elected to let the model use 20% of the data for self training purposes.

```
# We aim to make our neural net convert the 'dfa' values into a 'MEDV' value.
model = ks.models.Sequential()
model.add(ks.layers.Dense(13,activation='relu', input_shape = (13,)))
model.add(ks.layers.Dense(13, activation = 'relu'))
model.add(ks.layers.Dense(1))
model.compile(loss = 'mean_squared_error',optimizer='adagrad', metrics=['accuracy'])
```

```
#Here's where we deviate from earlier work, by modelling the full dataset and then splitting the data.
import sklearn.model_selection as mod
```

```
# split the dataframe inputs and outputs into training and test sets.
inputs_train, inputs_test, outputs_train, outputs_test = mod.train_test_split(dfa, dfb, test_size = 0.2)
```

```
# Train the neural network
model.fit(inputs_train, outputs_train, epochs =50, batch_size=10)
```

```
#To test our model we will use the 1st row of the dataset as our testcase (without the MEDV value).
#As we know what the actual data is we will then be able to compare the actual with the model output.
testn_data = np.array([0.00632,18.0,2.31, 0,0.538, 6.575, 65.2, 4.0900, 1, 296, 15.3, 396.90, 4.98])
print(model.predict(testn_data.reshape(1,13), batch_size=1))
```

```
[[29.031414]]
```

However, the accuracy of our neural network decreased with the application of pre-processing and or self-training.

As a confirmation of the findings obtained from the latest run of the model (i.e. pre-processed), we then did a model run to predict the top 5 and bottom 5 MEDV values, by list order as we considered this equivalent to a random selection of variables.

```
top_5_Out = df.iloc[0:5,13:]
top_5_Out = np.array(top_5_Out)
top_5_Out
```

```
array([[24. ],
       [21.6],
       [34.7],
       [33.4],
       [36.2]])
```

```
print(model.predict(top_5_In.reshape(5,13), batch_size=5))
```

```
[[27.679256]
 [26.698673]
 [28.387997]
 [31.411617]
 [30.638226]]
```

```
bot_5_Out=df.iloc[501:506,13:]
bot_5_Out= np.array(bot_5_Out)
print(bot_5_Out)
type(bot_5_Out)
```

```
[[22.4]
 [20.6]
 [23.9]
 [22. ]
 [11.9]]
```

```
numpy.ndarray
```

```
bot_5_predict = print(model.predict(bot_5_In.reshape(5,13), batch_size=5))
bot_5_predict = np.array(bot_5_predict)
type(bot_5_predict)
```

```
[[24.955082]
 [25.248653]
 [26.037075]
 [25.514858]
 [25.478313]]
```

However, the results obtained were less accurate than for the single layered output. A further investigation of the dataset would be required to establish the optimum parameters for multi layered output scenarios.

In conclusion, we found that the dataset did not require preprocessing to achieve the best prediction result. The optimum prediction was generated consistently from a sub-set of data. The level of accuracy using a simplified algorithm to select the input parameters proved to be high and consistent when we targeted one a layered output. The level of accuracy deteriorated for greater numbers of output layers, however this would need to be analysed further to confirm whether or not our input parameters are suitable for this approach.

Section 1 - Describe

In [1]:

```
import pandas as pd
import numpy as np
```

In [2]:

```
# Read in the csv file and select the columns we are interested in;
df = pd.read_csv("Boston_.csv", header=None, delimiter = ",")
```

In [3]:

```
df.head()
```

Out[3]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

In [4]:

```
df.tail()

```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273	21.0	391.99	9.67	22.4
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273	21.0	396.90	9.08	20.6
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273	21.0	396.90	5.64	23.9
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273	21.0	393.45	6.48	22.0
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273	21.0	396.90	7.88	11.9

Ok, the data set complex.

In [5]:

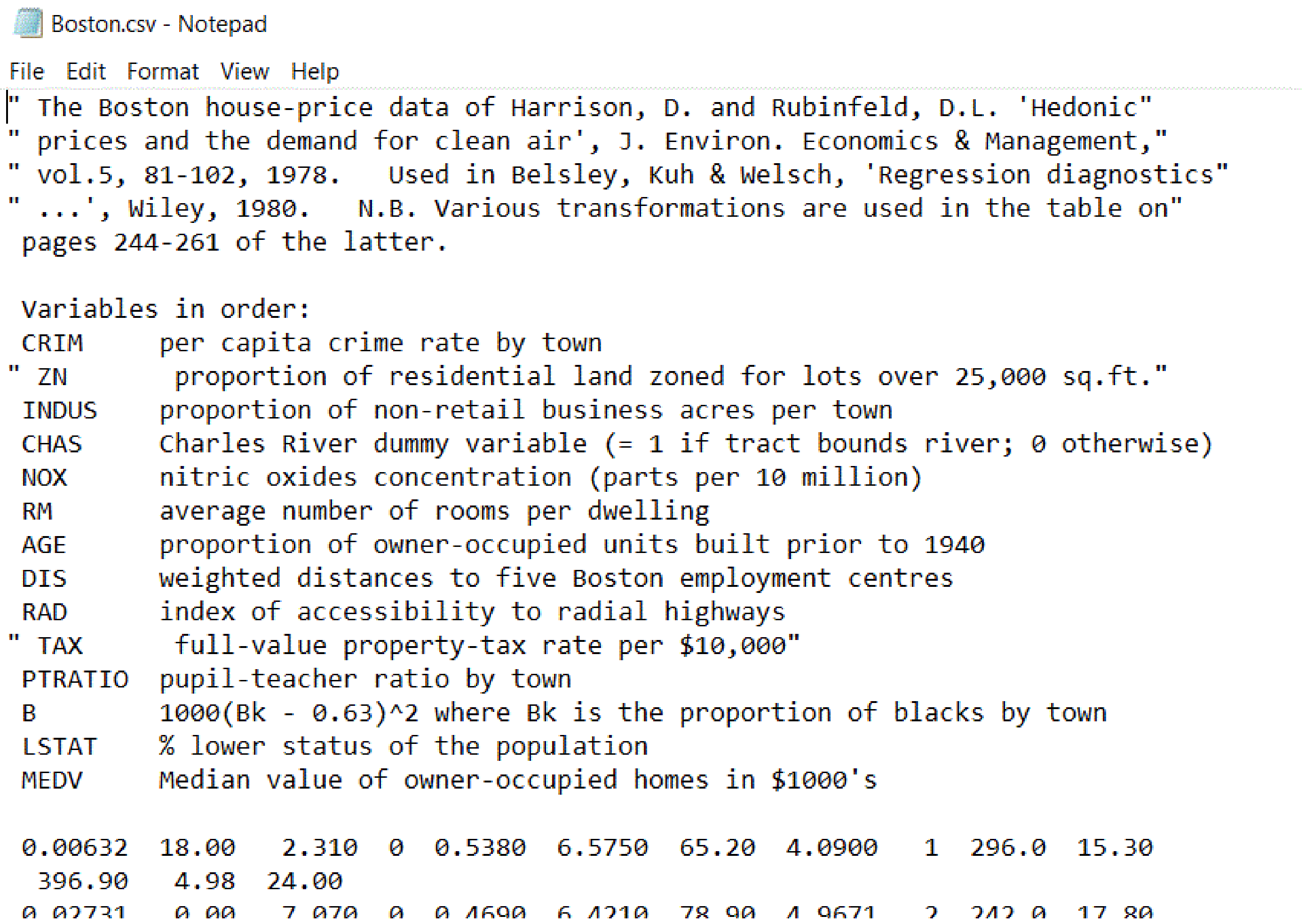
```
df
```

Out[5]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273	21.0	391.99	9.67	22.4
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273	21.0	396.90	9.08	20.6
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273	21.0	396.90	5.64	23.9
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273	21.0	393.45	6.48	22.0
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273	21.0	396.90	7.88	11.9

506 rows × 14 columns

Ok, we see that all the data is in 1 column, therefore we conclude that the data needs to be cleaned up.



```
" 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.

 Variables 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 blacks by town
 LSTAT    % lower status of the population
 MEDV     Median value of owner-occupied homes in $1000's

 0.00632  18.00   2.310  0  0.5380  6.5750  65.20  4.0900   1  296.0  15.30
  396.90   4.98  24.00
 0.02731   0.00   7.070  0  0.4690  6.4210  78.90  4.9671   2  242.0  17.80
```

In [6]:

```
#We name the columns according to the the text data extracted from the dataset.
col_name = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'P
```

In [7]:

```
df.columns = col_name
```

In [8]:

```
#We use the info method to view the data types for each variable we are dealing with.
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
CRIM       506 non-null float64
ZN         506 non-null float64
INDUS      506 non-null float64
CHAS       506 non-null int64
NOX        506 non-null float64
RM         506 non-null float64
AGE        506 non-null float64
DIS        506 non-null float64
RAD        506 non-null int64
TAX        506 non-null int64
PTRATIO    506 non-null float64
B          506 non-null float64
LSTAT      506 non-null float64
MEDV       506 non-null float64
dtypes: float64(11), int64(3)
memory usage: 55.5 KB
```

Ok. We appear to have 14 columns matching 14 datsets and the 'count' is equal across the columns. Now to provide some insight into the dataset we will do some basic data analysis.

In [9]:

```
#We call the function describe to provide useful insight into the dataset.  We can qui
#Does our dataset contain missing values?
```

In [10]:

```python
df.describe()
```

Out[10]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000
75%	3.677082	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000

In [11]:

```
# from the describe function we see that the dataset is complete and some of the categ
# We can get a feel for the distribution of the data, whether we have extreme values, 
# We will check to see if our dataset has duplicates, and if so we will remove.
dup = df.drop_duplicates
print(dup)
```

```
<bound method DataFrame.drop_duplicates of          CRIM    ZN  INDUS  CHAS
NOX     RM   AGE     DIS  RAD  TAX  \
0    0.00632  18.0   2.31     0  0.538  6.575  65.2  4.0900    1  296
1    0.02731   0.0   7.07     0  0.469  6.421  78.9  4.9671    2  242
2    0.02729   0.0   7.07     0  0.469  7.185  61.1  4.9671    2  242
3    0.03237   0.0   2.18     0  0.458  6.998  45.8  6.0622    3  222
4    0.06905   0.0   2.18     0  0.458  7.147  54.2  6.0622    3  222
..       ...   ...    ...   ...    ...    ...   ...     ...  ...  ...
501  0.06263   0.0  11.93     0  0.573  6.593  69.1  2.4786    1  273
502  0.04527   0.0  11.93     0  0.573  6.120  76.7  2.2875    1  273
503  0.06076   0.0  11.93     0  0.573  6.976  91.0  2.1675    1  273
504  0.10959   0.0  11.93     0  0.573  6.794  89.3  2.3889    1  273
505  0.04741   0.0  11.93     0  0.573  6.030  80.8  2.5050    1  273

     PTRATIO       B  LSTAT  MEDV
0       15.3  396.90   4.98  24.0
1       17.8  396.90   9.14  21.6
2       17.8  392.83   4.03  34.7
3       18.7  394.63   2.94  33.4
4       18.7  396.90   5.33  36.2
..       ...     ...    ...   ...
501     21.0  391.99   9.67  22.4
502     21.0  396.90   9.08  20.6
503     21.0  396.90   5.64  23.9
504     21.0  393.45   6.48  22.0
505     21.0  396.90   7.88  11.9

[506 rows x 14 columns]>
```

In [12]:

```
#We note that the no. of rows and colums is the same before and after we call the funt
```

In [13]:

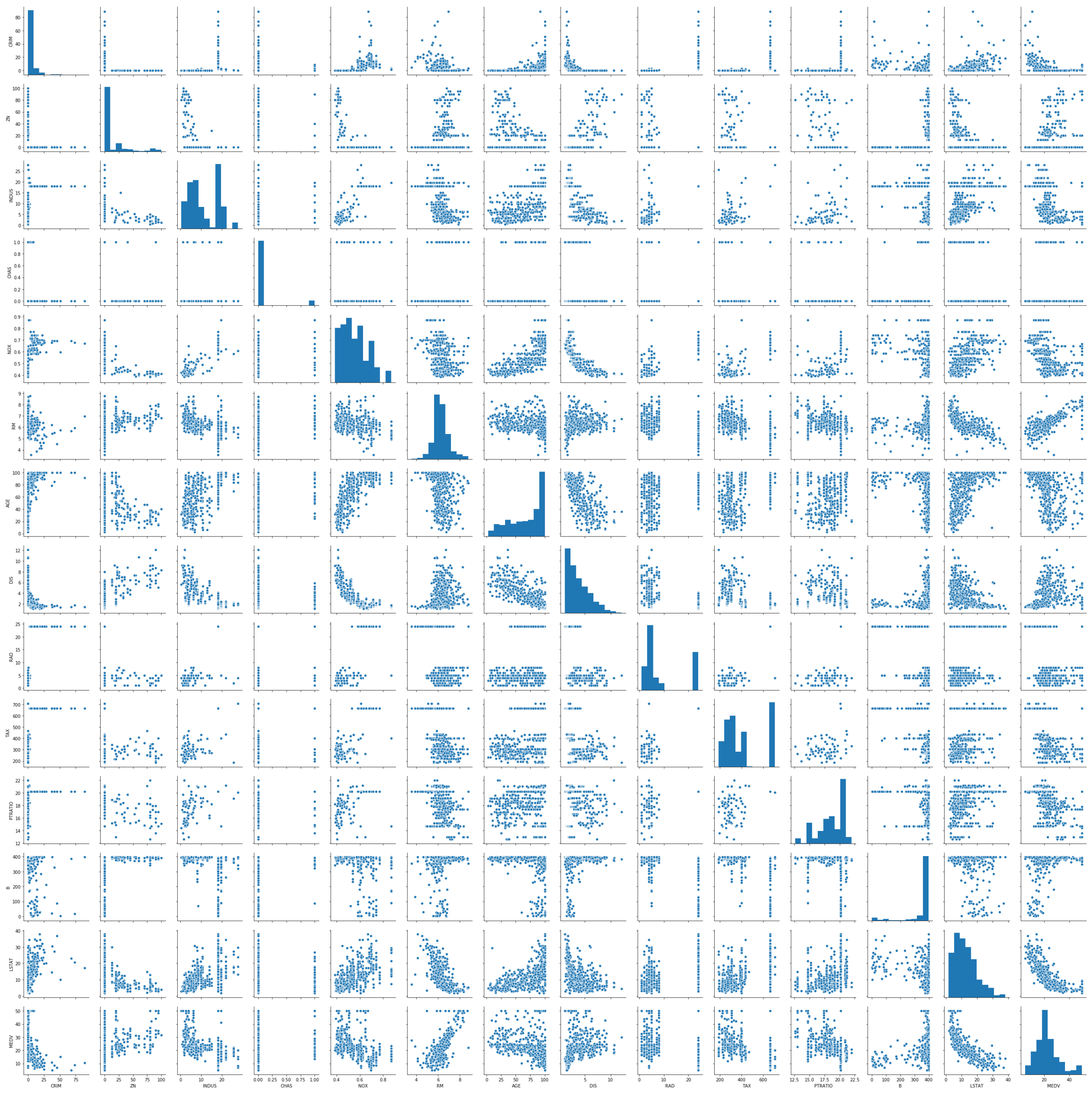
```
# We use the matplotlib and seaborn libraries to illustrate plots of our data for comp
import matplotlib.pyplot as plt
import seaborn as sns
```

In [14]:

```
sns.pairplot(df)
```

Out[14]:

```
<seaborn.axisgrid.PairGrid at 0x1c6610868c8>
```



There is surplus data in the dataset to 'decribe' it, therefore we will use selected data for visualisation.

In [15]:

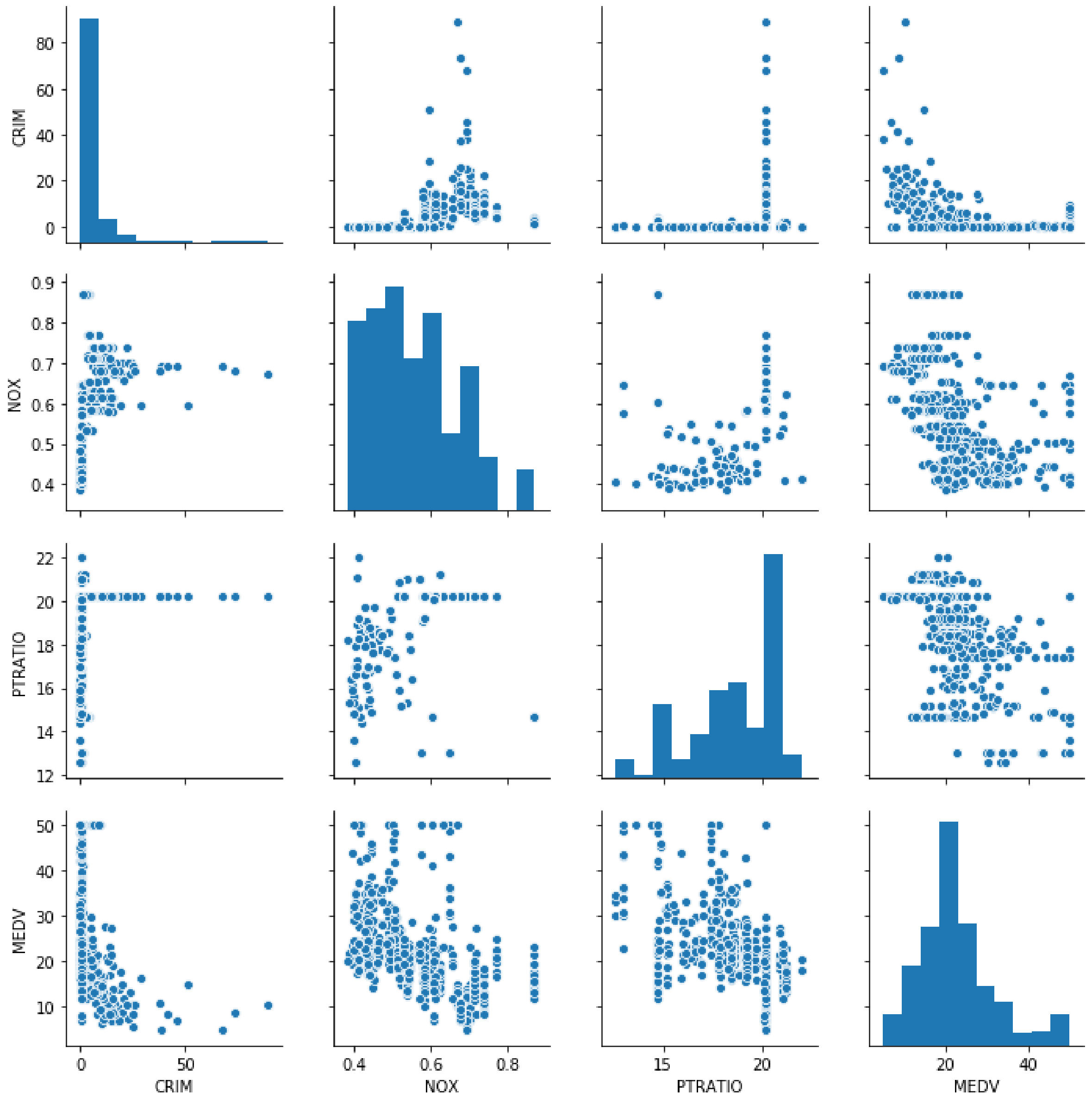
```python
short = ['CRIM', 'NOX', 'PTRATIO','MEDV']
```

In [16]:

```python
sns.pairplot(df[short])
plt.show
```

Out[16]:

```
<function matplotlib.pyplot.show(*args, **kw)>
```



In [17]:

```python
def min_max_values(col):
    top = df[col].idxmax()
    top_obs = pd.DataFrame(df.loc[top])
    bottom = df[col].idxmin()
    bottom_obs = pd.DataFrame(df.loc[bottom])
    min_max_obs = pd.concat([top_obs, bottom_obs], axis=1)
    return min_max_obs
```

Type *Markdown* and LaTeX: α^2

In [18]:

```
min_max_values('MEDV')
```

Out[18]:

	161	398
CRIM	1.46336	38.3518
ZN	0.00000	0.0000
INDUS	19.58000	18.1000
CHAS	0.00000	0.0000
NOX	0.60500	0.6930
RM	7.48900	5.4530
AGE	90.80000	100.0000
DIS	1.97090	1.4896
RAD	5.00000	24.0000
TAX	403.00000	666.0000
PTRATIO	14.70000	20.2000
B	374.43000	396.9000
LSTAT	1.73000	30.5900
MEDV	50.00000	5.0000

In [19]:

```
min_max_values('PTRATIO')
```

Out[19]:

	354	196
CRIM	0.04301	0.04011
ZN	80.00000	80.00000
INDUS	1.91000	1.52000
CHAS	0.00000	0.00000
NOX	0.41300	0.40400
RM	5.66300	7.28700
AGE	21.90000	34.10000
DIS	10.58570	7.30900
RAD	4.00000	2.00000
TAX	334.00000	329.00000
PTRATIO	22.00000	12.60000
B	382.80000	396.90000
LSTAT	8.05000	4.08000
MEDV	18.20000	33.30000

In [20]:

```
min_max_values('CRIM')
```

Out[20]:

	380	0
CRIM	88.9762	0.00632
ZN	0.0000	18.00000
INDUS	18.1000	2.31000
CHAS	0.0000	0.00000
NOX	0.6710	0.53800
RM	6.9680	6.57500
AGE	91.9000	65.20000
DIS	1.4165	4.09000
RAD	24.0000	1.00000
TAX	666.0000	296.00000
PTRATIO	20.2000	15.30000
B	396.9000	396.90000
LSTAT	17.2100	4.98000
MEDV	10.4000	24.00000

In [21]:

```
df_sort = df.sort_values(by = 'MEDV', ascending=False).head()
df_sort.head()
```

Out[21]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	L
283	0.01501	90.0	1.21	1	0.401	7.923	24.8	5.8850	1	198	13.6	395.52	
225	0.52693	0.0	6.20	0	0.504	8.725	83.0	2.8944	8	307	17.4	382.00	
369	5.66998	0.0	18.10	1	0.631	6.683	96.8	1.3567	24	666	20.2	375.33	
370	6.53876	0.0	18.10	1	0.631	7.016	97.5	1.2024	24	666	20.2	392.05	
371	9.23230	0.0	18.10	0	0.631	6.216	100.0	1.1691	24	666	20.2	366.15	

In [22]:

```python
df_sort = df.sort_values(by = 'CHAS', ascending=False).head()
df_sort.head()
```

Out[22]:

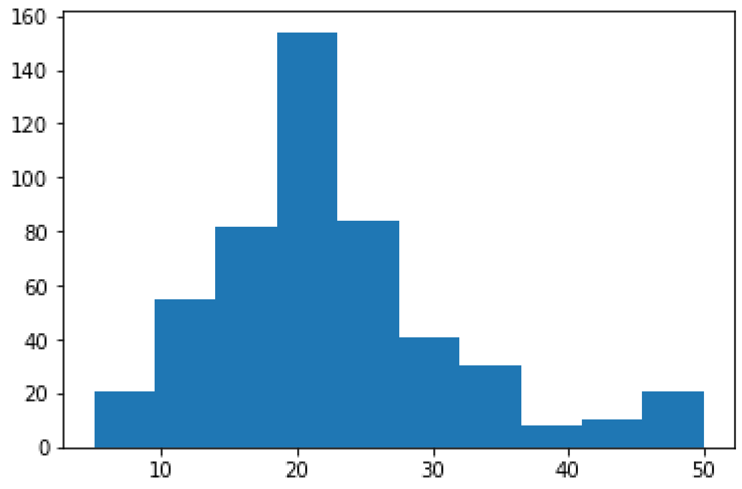
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LS
160	1.27346	0.0	19.58	1	0.605	6.250	92.6	1.7984	5	403	14.7	338.92	
358	5.20177	0.0	18.10	1	0.770	6.127	83.4	2.7227	24	666	20.2	395.43	1
210	0.17446	0.0	10.59	1	0.489	5.960	92.1	3.8771	4	277	18.6	393.25	1
236	0.52058	0.0	6.20	1	0.507	6.631	76.5	4.1480	8	307	17.4	388.45	
152	1.12658	0.0	19.58	1	0.871	5.012	88.0	1.6102	5	403	14.7	343.28	1

In [23]:

```python
num_bins=10
plt.hist(df['MEDV'], num_bins)
```

Out[23]:

```
(array([ 21.,  55.,  82., 154.,  84.,  41.,  30.,   8.,  10.,  21.]),
 array([ 5. ,  9.5, 14. , 18.5, 23. , 27.5, 32. , 36.5, 41. , 45.5, 50. ]),
 <a list of 10 Patch objects>)
```

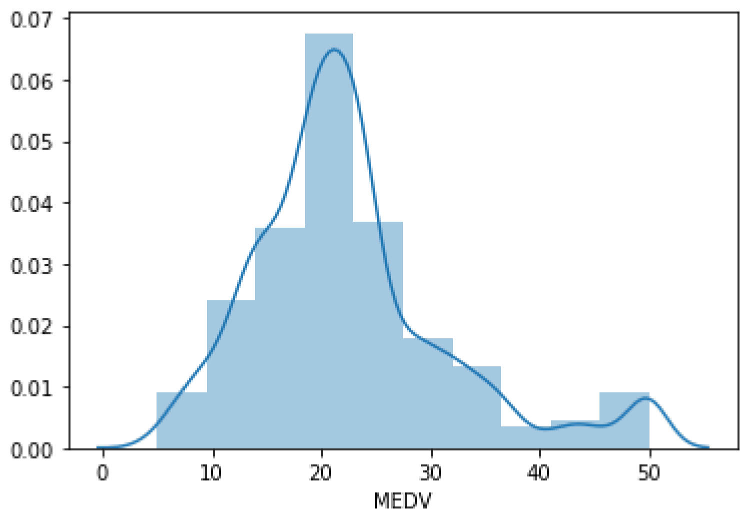


In [24]:

```
sns.distplot(df['MEDV'], num_bins)
```

Out[24]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c66c641808>
```



In [25]:

```
make_dist = df.groupby('MEDV').size()
```

In [26]:

```
make_dist
```

Out[26]:

```
MEDV
5.0      2
5.6      1
6.3      1
7.0      2
7.2      3
        ..
46.7     1
48.3     1
48.5     1
48.8     1
50.0    16
Length: 229, dtype: int64
```

In [27]:

```
df_num = df.select_dtypes(include=['float64', 'int64'])
df_num.head()
```

Out[27]:

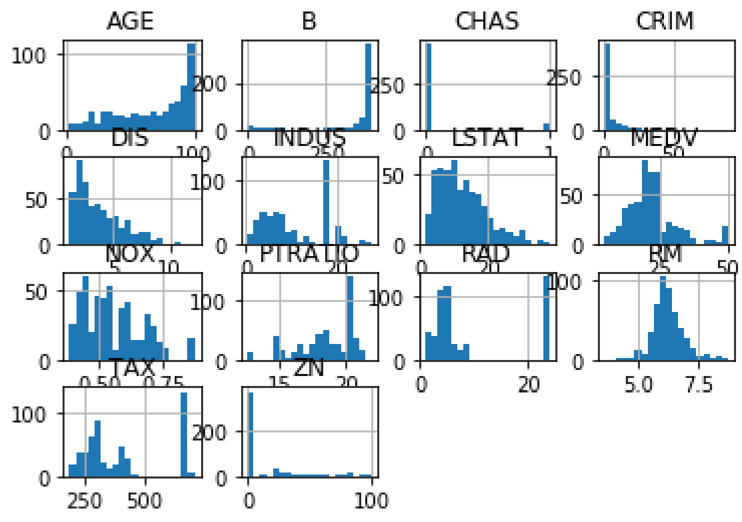
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LST
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.

In [28]:

```
df_num.hist(bins = 20)
```

Out[28]:

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C3CFFC8
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C6F3AC8
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C72D308
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C764E08
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C79CF08
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C7D8088
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C8100C8
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C8491C8
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C84ED88
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C887F88
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C8F3548
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C92C588
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C9636C8
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C99C7C8
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66C9D58C8
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001C66CA0BAC8
>]],
      dtype=object)
```



In [29]:

```
#correlation with the variable of interest
df_corr = df_num.corr()['CRIM'][:-1]
```

In [30]:

```
df_corr
```

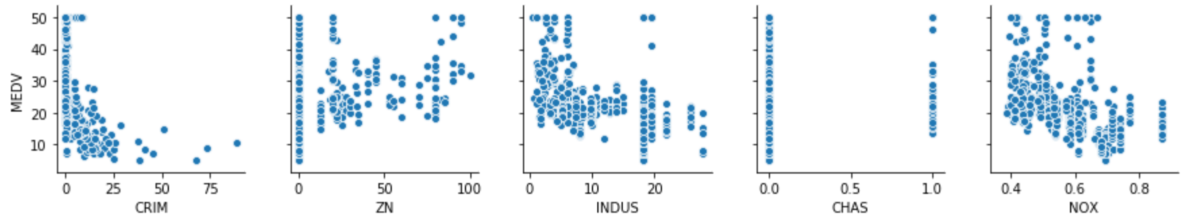
Out[30]:

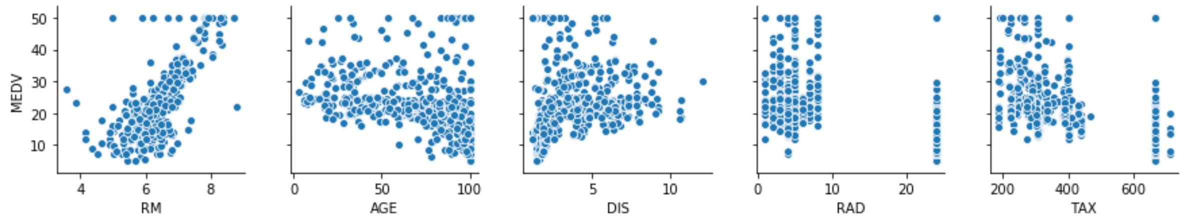
```
CRIM       1.000000
ZN        -0.200469
INDUS      0.406583
CHAS      -0.055892
NOX        0.420972
RM        -0.219247
AGE        0.352734
DIS       -0.379670
RAD        0.625505
TAX        0.582764
PTRATIO    0.289946
B         -0.385064
LSTAT      0.455621
Name: CRIM, dtype: float64
```

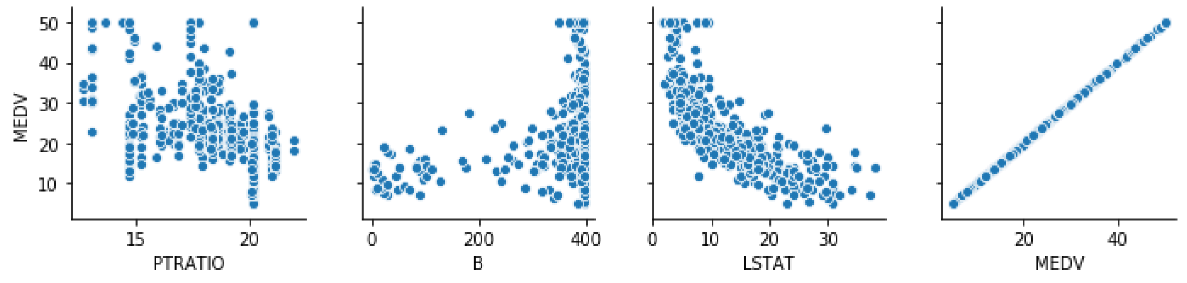
Where we have a negative sign, that means a negative correlation, so where one variable increases the other decreases and vice versa for the postive sign

In [31]:

```
for i in range(0, len(df_num.columns),5):
    sns.pairplot(df_num, y_vars=['MEDV'], x_vars= df_num.columns[i:i+5])
```



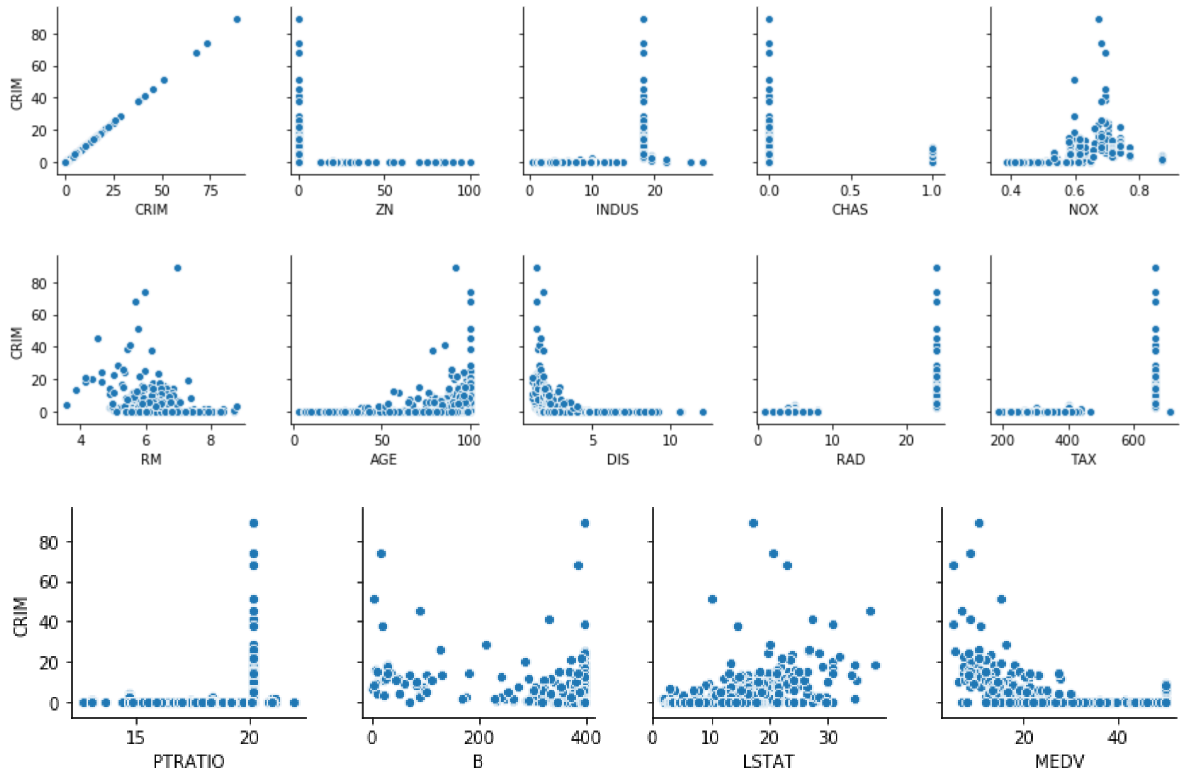




Downward sloping is a negative relationship e.g. CRIM, LSTAT

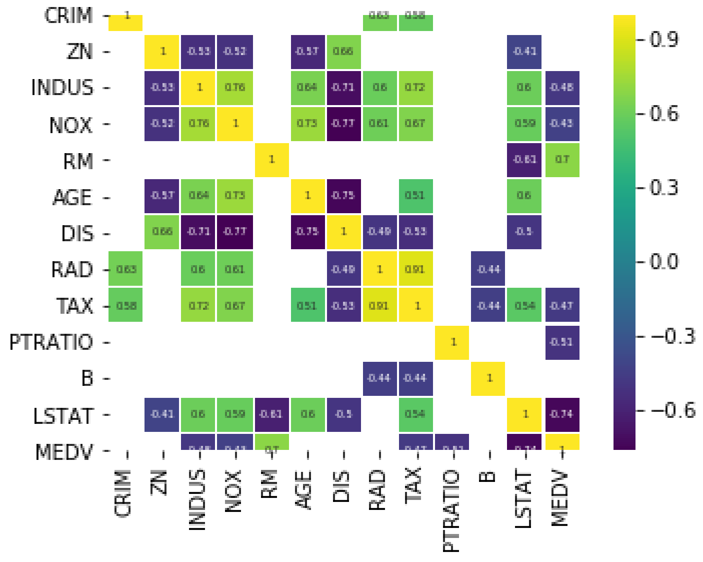
In [32]:

```
for i in range(0, len(df_num.columns),5):
    sns.pairplot(df_num, y_vars=['CRIM'], x_vars= df_num.columns[i:i+5])
```



In [33]:

```
#plotting significant correlation in a heatmap
corr = df_num.drop('CHAS', axis=1).corr()
sns.heatmap(corr[(corr>=0.5) | (corr <=-0.4)], cmap='viridis', vmax=1.0, linewidths=0.
```



Section 2 - Infer

Requirement - use inferential statistics to analyse whether there is a significant difference in median house prices between houses that are along the Charles river and those that aren’t.

Plan - we now need to split the dataset into two based on the 'CHAS' parameter (Charles River Dummy Variable =1 if track bounds the river; otherwise = 0).

In [34]:

```
df.head()
```

Out[34]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LST
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.

In [35]:

```
df1 =df[df['CHAS']== 0]
```

In [36]:

```
df1.describe()
```

Out[36]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	
count	471.000000	471.000000	471.000000	471.0	471.000000	471.000000	471.000000	471.000
mean	3.744447	11.634820	11.019193	0.0	0.551817	6.267174	67.911677	3.851
std	8.876818	23.617979	6.913850	0.0	0.113102	0.685895	28.458924	2.145
min	0.006320	0.000000	0.460000	0.0	0.385000	3.561000	2.900000	1.137
25%	0.079640	0.000000	5.040000	0.0	0.448000	5.882000	42.500000	2.105
50%	0.245220	0.000000	8.560000	0.0	0.538000	6.202000	76.500000	3.215
75%	3.695030	12.500000	18.100000	0.0	0.624000	6.594000	94.100000	5.287
max	88.976200	100.000000	27.740000	0.0	0.871000	8.725000	100.000000	12.126

In [37]:

```
df2 = df[df['CHAS']==1]
```

In [38]:

```
df2.describe()
```

Out[38]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	
count	35.000000	35.000000	35.000000	35.0	35.000000	35.000000	35.00000	35.000000	35.(
mean	1.851670	7.714286	12.719143	1.0	0.593426	6.519600	77.50000	3.029709	9.:
std	2.494072	18.800143	5.957623	0.0	0.144736	0.876416	22.02134	1.254728	8.:
min	0.015010	0.000000	1.210000	1.0	0.401000	5.012000	24.80000	1.129600	1.(
25%	0.125060	0.000000	6.410000	1.0	0.489000	5.935500	60.30000	1.904700	4.(
50%	0.447910	0.000000	13.890000	1.0	0.550000	6.250000	88.50000	3.048000	5.(
75%	3.397665	0.000000	18.100000	1.0	0.693000	6.915000	93.20000	3.897300	8.(
max	8.982960	90.000000	19.580000	1.0	0.871000	8.780000	100.00000	5.885000	24.(

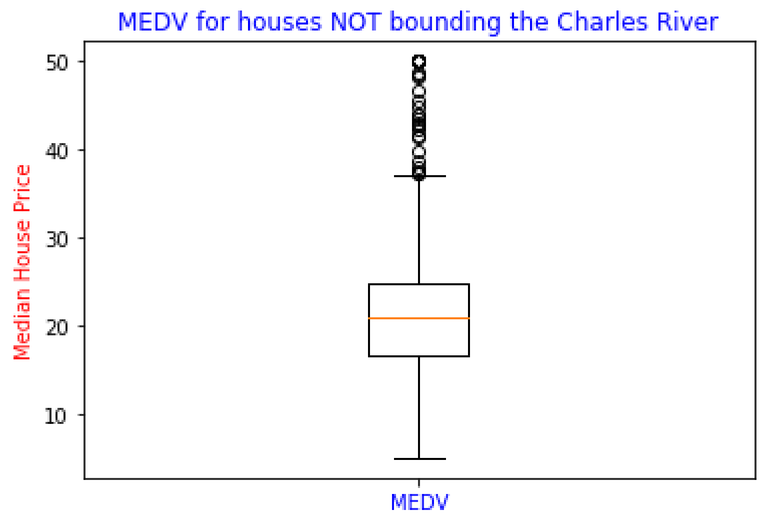
Note the results of the division based on the CHAS binary indicator. We see that the count is skewed. We have a population of 471 vs a population of 35, in analysising whether the median house prices are different for houses along the Charles River (35 samples) or not (471 samples). Therefore the sample dataset has a bias as the sample for median house prices away from the Charles River is over 13 times larger than the corresponding dataset.

In [39]:

```
A = plt.boxplot(df1['MEDV'])
plt.xticks([1], ['MEDV'], color='b')
plt.ylabel('Median House Price',color='r')
plt.title("MEDV for houses NOT bounding the Charles River", fontdict=None, loc='center
```

Out[39]:

```
Text(0.5, 1.0, 'MEDV for houses NOT bounding the Charles River')
```



In [40]:

```
stats1 = df1['MEDV'].describe()
stats1
```

Out[40]:

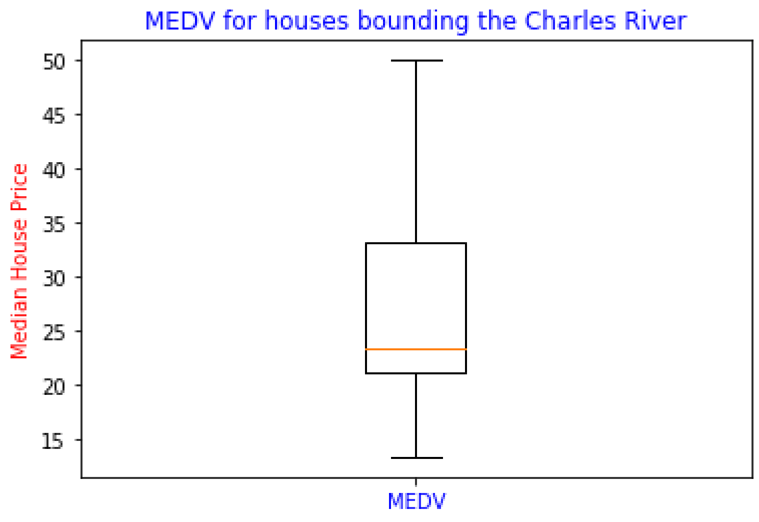
```
count    471.000000
mean      22.093843
std        8.831362
min        5.000000
25%       16.600000
50%       20.900000
75%       24.800000
max       50.000000
Name: MEDV, dtype: float64
```

In [41]:

```
B = plt.boxplot(df2['MEDV'])
plt.xticks([1], ['MEDV'], color='b')
plt.ylabel('Median House Price',color='r')
plt.title("MEDV for houses bounding the Charles River", fontdict=None, loc='center', c
```

Out[41]:

```
Text(0.5, 1.0, 'MEDV for houses bounding the Charles River')
```



In [42]:

```
stats2 = df2['MEDV'].describe()
```

In [43]:

```
stats2
```

Out[43]:

```
count    35.000000
mean     28.440000
std      11.816643
min      13.400000
25%      21.100000
50%      23.300000
75%      33.150000
max      50.000000
Name: MEDV, dtype: float64
```

Independent T-test

In order to determine if there is a significant difference between the mean of the two groups we have carried out an independent t-test. If the mean of df1['MEDV'] minus the mean of df2['MEDV'] returns a value of zero, we conculde that the means of each dataset are equal, otherwise known as the 'null hypothesis'. Else, the means are not equal (or return a value above our predetermined propability value) then we conclude that the alternative hypothesis has been proven. For this analysis we will determine a P-value of 0.05, which is a widely used reference value.Ref (https://pythonfordatascience.org/independent-t-test-python/)

Boxplot (https://en.wikipedia.org/wiki/Box_plot)

In [44]:

```
import scipy.stats as ss
ss.ttest_ind(df1['MEDV'], df2['MEDV'])
```

Out[44]:

```
Ttest_indResult(statistic=-3.996437466090509, pvalue=7.390623170519905e-05)
```

In [45]:

```
import statsmodels.stats.weightstats as ws
ws.ttest_ind(df1['MEDV'], df2['MEDV'])
```

Out[45]:

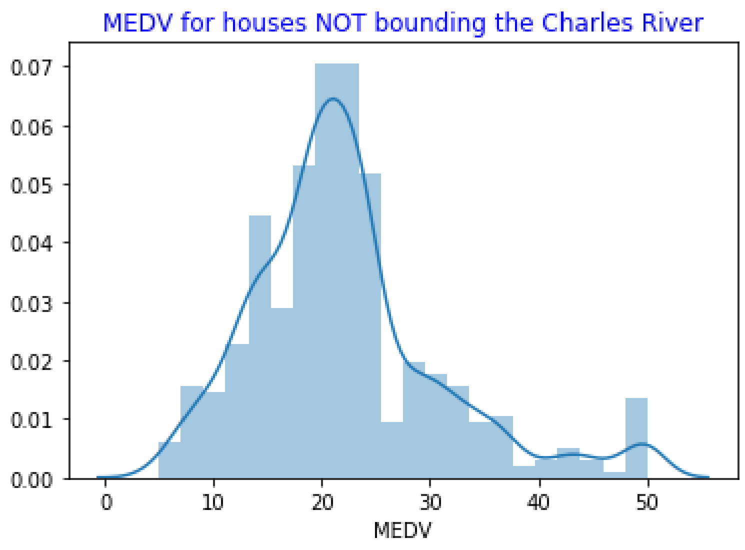
```
(-3.9964374660905095, 7.390623170519883e-05, 504.0)
```

In [46]:

```
sns.distplot(df1['MEDV'])
plt.title("MEDV for houses NOT bounding the Charles River", fontdict=None, loc='center
```

Out[46]:

```
Text(0.5, 1.0, 'MEDV for houses NOT bounding the Charles River')
```

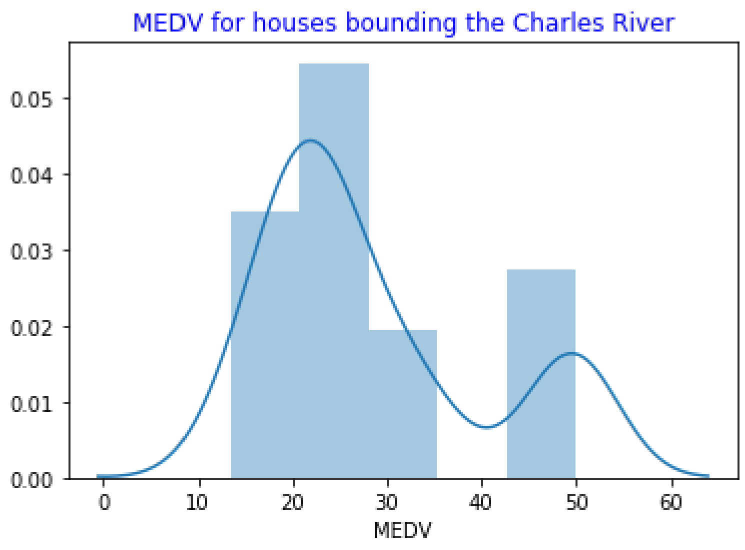


In [47]:

```
sns.distplot(df2['MEDV'])
plt.title("MEDV for houses bounding the Charles River", fontdict=None, loc='center', c
```

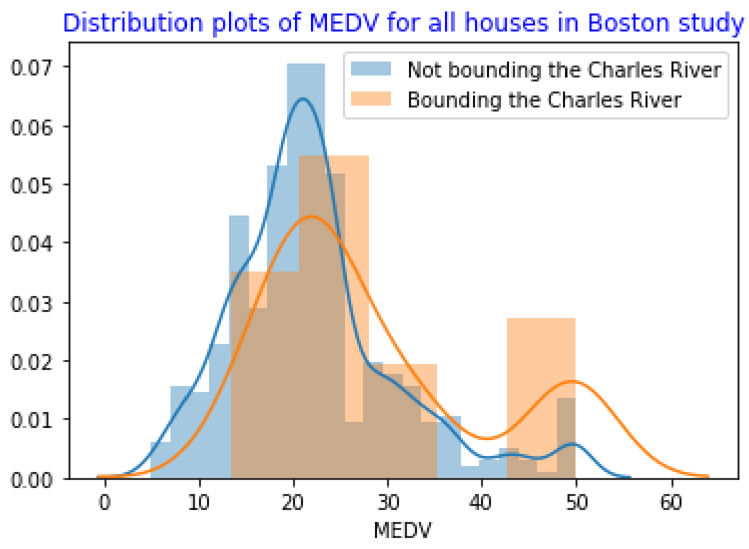
Out[47]:

```
Text(0.5, 1.0, 'MEDV for houses bounding the Charles River')
```



In [48]:

```
sns.distplot(df1['MEDV'], label = "Not bounding the Charles River")
sns.distplot(df2['MEDV'], label = "Bounding the Charles River")
plt.title("Distribution plots of MEDV for all houses in Boston study", fontdict=None, 
plt.legend()
plt.show()
```



In [ ]:

```

```

Section 3 - Predict.

Create a neural network that can predict the median house price based on the other variables in the dataset.

In [49]:

```
#Refresh.
df.head()
```

Out[49]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LST
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.

In [50]:

```
Input = df.drop(columns = ["MEDV"])
Output = df[["MEDV"]]
```

In [51]:

```
#For create a neural network
import keras as ks
```

```
Using TensorFlow backend.
```

In [52]:

```
# define model parameter
m = ks.models.Sequential()
```

BUILD 1

In [116]:

```
#Add neurons.
m.add(ks.layers.Dense(13, activation='relu', input_shape = (13,)))
# We use '13' layers in reference to the number of categories we have for input (but tl
# In this model run we are using the Rectified Linear Unit activation function (relu).
#postive, otherwise it will output zero.
#We use the number of columns for the input shape value.
m.add(ks.layers.Dense(13, activation= 'relu'))
#We add a 2nd layer without the input shape.
m.add(ks.layers.Dense(1))
# We only want 1 output from the model which is equal to the MEDV value.
#Complile the model.
m.compile(loss='mean_squared_error',optimizer='adagrad')
#Compiled using 'adam' as the optimiser.  The optimiser attempts to minimise our error
```

TRAIN 1

In [117]:

```python
# Train the neural network by fiting the inputs to the outputs
m.fit(Input, Output, epochs=50)
#callbacks=[ks.callbacks.EarlyStopping(patience=5)]
```

```
Epoch 1/50
506/506 [==============================] - 0s 950us/step - loss: 409.2867
Epoch 2/50
506/506 [==============================] - 0s 51us/step - loss: 236.5017
Epoch 3/50
506/506 [==============================] - 0s 69us/step - loss: 156.8924
Epoch 4/50
506/506 [==============================] - 0s 59us/step - loss: 116.7358
Epoch 5/50
506/506 [==============================] - 0s 59us/step - loss: 98.5560
Epoch 6/50
506/506 [==============================] - 0s 63us/step - loss: 90.2837
Epoch 7/50
506/506 [==============================] - 0s 51us/step - loss: 86.8363
Epoch 8/50
506/506 [==============================] - 0s 61us/step - loss: 85.3893
Epoch 9/50
506/506 [==============================] - 0s 51us/step - loss: 84.7993
Epoch 10/50
506/506 [==============================] - 0s 75us/step - loss: 84.5664
Epoch 11/50
506/506 [==============================] - 0s 51us/step - loss: 84.5582
Epoch 12/50
506/506 [==============================] - 0s 59us/step - loss: 84.4850
Epoch 13/50
506/506 [==============================] - 0s 53us/step - loss: 84.4583
Epoch 14/50
506/506 [==============================] - 0s 61us/step - loss: 84.5109
Epoch 15/50
506/506 [==============================] - 0s 53us/step - loss: 84.4813
Epoch 16/50
506/506 [==============================] - 0s 59us/step - loss: 84.4922
Epoch 17/50
506/506 [==============================] - 0s 51us/step - loss: 84.4879
Epoch 18/50
506/506 [==============================] - 0s 63us/step - loss: 84.5056
Epoch 19/50
506/506 [==============================] - 0s 55us/step - loss: 84.5333
Epoch 20/50
506/506 [==============================] - 0s 61us/step - loss: 84.5181
Epoch 21/50
506/506 [==============================] - 0s 55us/step - loss: 84.4805
Epoch 22/50
506/506 [==============================] - 0s 57us/step - loss: 84.4680
Epoch 23/50
506/506 [==============================] - 0s 55us/step - loss: 84.5083
Epoch 24/50
506/506 [==============================] - 0s 53us/step - loss: 84.5254
Epoch 25/50
506/506 [==============================] - 0s 65us/step - loss: 84.4600
Epoch 26/50
506/506 [==============================] - 0s 55us/step - loss: 84.4707
Epoch 27/50
506/506 [==============================] - 0s 63us/step - loss: 84.4949
Epoch 28/50
```

```
506/506 [==============================] - 0s 55us/step - loss: 84.4919
Epoch 29/50
506/506 [==============================] - 0s 63us/step - loss: 84.5331
Epoch 30/50
506/506 [==============================] - 0s 53us/step - loss: 84.4585
Epoch 31/50
506/506 [==============================] - 0s 63us/step - loss: 84.5344
Epoch 32/50
506/506 [==============================] - 0s 55us/step - loss: 84.4809
Epoch 33/50
506/506 [==============================] - 0s 59us/step - loss: 84.5079
Epoch 34/50
506/506 [==============================] - 0s 59us/step - loss: 84.4662
Epoch 35/50
506/506 [==============================] - 0s 59us/step - loss: 84.4821
Epoch 36/50
506/506 [==============================] - 0s 85us/step - loss: 84.4977
Epoch 37/50
506/506 [==============================] - 0s 63us/step - loss: 84.4567
Epoch 38/50
506/506 [==============================] - 0s 51us/step - loss: 84.4792
Epoch 39/50
506/506 [==============================] - 0s 53us/step - loss: 84.4964
Epoch 40/50
506/506 [==============================] - 0s 61us/step - loss: 84.4647
Epoch 41/50
506/506 [==============================] - 0s 55us/step - loss: 84.4893
Epoch 42/50
506/506 [==============================] - 0s 65us/step - loss: 84.4981
Epoch 43/50
506/506 [==============================] - 0s 51us/step - loss: 84.4752
Epoch 44/50
506/506 [==============================] - 0s 59us/step - loss: 84.4720
Epoch 45/50
506/506 [==============================] - 0s 57us/step - loss: 84.4787
Epoch 46/50
506/506 [==============================] - 0s 53us/step - loss: 84.4787
Epoch 47/50
506/506 [==============================] - 0s 59us/step - loss: 84.5116
Epoch 48/50
506/506 [==============================] - 0s 53us/step - loss: 84.4590
Epoch 49/50
506/506 [==============================] - 0s 63us/step - loss: 84.4925
Epoch 50/50
506/506 [==============================] - 0s 59us/step - loss: 84.4518
```

```
Out[117]:
```

```
<keras.callbacks.callbacks.History at 0x1c677648848>
```

TEST 1

In [118]:

```
#To test our model we will use the 1st row of the dataset as our testcase (without the
#As we know what the actual data is we will then be able to compare the actual with the
#The actual price is $24,000
test1_data = np.array([0.00632,18.0,2.31, 0,0.538, 6.575, 65.2, 4.0900, 1, 296, 15.3,
print(m.predict(test1_data.reshape(1,13), batch_size=1))
```

```
[[22.489676]]
```

In [56]:

```
type(m.predict(Input))
```

Out[56]:

```
numpy.ndarray
```

In [57]:

```
Output.as_matrix()
```

```
C:\Users\justi\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: Future
Warning: Method .as_matrix will be removed in a future version. Use .value
s instead.
  """Entry point for launching an IPython kernel.
```

In [58]:

```
np.sqrt(np.sum((m.predict(Input) - Output.as_matrix())**2))
```

```
C:\Users\justi\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: FutureWa
rning: Method .as_matrix will be removed in a future version. Use .values in
stead.
  """Entry point for launching an IPython kernel.
```

Out[58]:

```
439106.27383970155
```

RUN OTHER SENARIOS

Let's use a sub-set of the dataset to see if we can improve upon our results.

In [60]:

```
subIn = df.drop(columns=['CHAS','AGE', 'B', 'MEDV'])
```

In [61]:

```
subIn.head()
```

Out[61]:

	CRIM	ZN	INDUS	NOX	RM	DIS	RAD	TAX	PTRATIO	LSTAT
0	0.00632	18.0	2.31	0.538	6.575	4.0900	1	296	15.3	4.98
1	0.02731	0.0	7.07	0.469	6.421	4.9671	2	242	17.8	9.14
2	0.02729	0.0	7.07	0.469	7.185	4.9671	2	242	17.8	4.03
3	0.03237	0.0	2.18	0.458	6.998	6.0622	3	222	18.7	2.94
4	0.06905	0.0	2.18	0.458	7.147	6.0622	3	222	18.7	5.33

In [62]:

```
subOut = df[["MEDV"]]
```

In [119]:

```
m1 = ks.models.Sequential()
m1.add(ks.layers.Dense(10, activation='relu', input_shape=(10,)))
m1.add(ks.layers.Dense(10, activation= 'relu'))
m1.add(ks.layers.Dense(1))
m1.compile(loss='mean_squared_error',optimizer='adagrad')
```

In [120]:

```
m1.fit(subIn, subOut, epochs=50,callbacks=[ks.callbacks.EarlyStopping(monitor='val_los

# early stopping from the Keras https://keras.io/callbacks/

```

```
Epoch 1/50
506/506 [==============================] - 0s 566us/step - loss: 178.7130
Epoch 2/50
506/506 [==============================] - 0s 43us/step - loss: 144.2409
Epoch 3/50
506/506 [==============================] - 0s 49us/step - loss: 136.5151
Epoch 4/50
506/506 [==============================] - 0s 45us/step - loss: 125.5370
Epoch 5/50
506/506 [==============================] - 0s 41us/step - loss: 117.7323
Epoch 6/50
506/506 [==============================] - 0s 59us/step - loss: 112.9192
Epoch 7/50
506/506 [==============================] - 0s 45us/step - loss: 110.7013
Epoch 8/50
 32/506 [>.............................] - ETA: 0s - loss: 117.6526
```

In [ ]:

```

```

In [121]:

```
#Now we will run a test on the subset to see if reducing the input variables by select
#Can we get a better model result (knowing the acutal output = 24)?
```

In [122]:

```
subset1_test = np.array([0.00632,18.0,2.31,0.538, 6.575, 4.0900, 1, 296, 15.3, 4.98])
print(m1.predict(subset1_test.reshape(1,10), batch_size=1))
```

```
[[24.319355]]
```

If we reduce the dataset further will we get an even better result?

In [67]:

```
subIn2 = df.drop(columns=['CRIM', 'INDUS','CHAS', 'NOX','AGE','TAX', 'B', 'MEDV',])
subIn2.head()
```

Out[67]:

	ZN	RM	DIS	RAD	PTRATIO	LSTAT
0	18.0	6.575	4.0900	1	15.3	4.98
1	0.0	6.421	4.9671	2	17.8	9.14
2	0.0	7.185	4.9671	2	17.8	4.03
3	0.0	6.998	6.0622	3	18.7	2.94
4	0.0	7.147	6.0622	3	18.7	5.33

In [68]:

```
subOut = df[["MEDV"]]
```

In [128]:

```
m2 = ks.models.Sequential()
m2.add(ks.layers.Dense(6, activation='relu', input_shape=(6,)))
m2.add(ks.layers.Dense(10, activation= 'relu'))
m2.add(ks.layers.Dense(1))
m2.compile(loss='mean_squared_error',optimizer='adagrad')
```

In [129]:

```
m2.fit(subIn2, subOut, epochs=50,callbacks=[ks.callbacks.EarlyStopping(monitor='val_lo
```

```
Epoch 1/50
506/506 [==============================] - 0s 566us/step - loss: 647.4539
Epoch 2/50
506/506 [==============================] - 0s 51us/step - loss: 395.0789
Epoch 3/50
506/506 [==============================] - 0s 57us/step - loss: 271.4004
Epoch 4/50
506/506 [==============================] - 0s 51us/step - loss: 217.2746
Epoch 5/50
506/506 [==============================] - 0s 49us/step - loss: 192.0805
Epoch 6/50
506/506 [==============================] - 0s 61us/step - loss: 178.7924
Epoch 7/50
506/506 [==============================] - 0s 49us/step - loss: 170.0395
Epoch 8/50
506/506 [==============================] - 0s 55us/step - loss: 163.3518
Epoch 9/50
506/506 [==============================] - 0s 61us/step - loss: 157.6553
Epoch 10/50
506/506 [==============================] - 0s 59us/step - loss: 152.7911
Epoch 11/50
506/506 [==============================] - 0s 53us/step - loss: 148.1099
Epoch 12/50
506/506 [==============================] - 0s 47us/step - loss: 144.0706
Epoch 13/50
506/506 [==============================] - 0s 51us/step - loss: 140.1751
Epoch 14/50
506/506 [==============================] - 0s 49us/step - loss: 136.5960
Epoch 15/50
506/506 [==============================] - 0s 57us/step - loss: 133.0792
Epoch 16/50
506/506 [==============================] - 0s 49us/step - loss: 129.8354
Epoch 17/50
506/506 [==============================] - 0s 57us/step - loss: 126.6848
Epoch 18/50
506/506 [==============================] - 0s 53us/step - loss: 123.6753
Epoch 19/50
506/506 [==============================] - 0s 61us/step - loss: 120.7631
Epoch 20/50
506/506 [==============================] - 0s 61us/step - loss: 117.9567
Epoch 21/50
506/506 [==============================] - 0s 61us/step - loss: 115.2304
Epoch 22/50
506/506 [==============================] - 0s 59us/step - loss: 112.4682
Epoch 23/50
506/506 [==============================] - 0s 51us/step - loss: 109.9512
Epoch 24/50
506/506 [==============================] - 0s 59us/step - loss: 107.4598
Epoch 25/50
506/506 [==============================] - 0s 51us/step - loss: 105.0844
Epoch 26/50
506/506 [==============================] - 0s 59us/step - loss: 102.8316
Epoch 27/50
506/506 [==============================] - 0s 55us/step - loss: 100.5996
Epoch 28/50
506/506 [==============================] - 0s 65us/step - loss: 98.5009
```

```
Epoch 29/50
506/506 [==============================] - 0s 59us/step - loss: 96.1603
Epoch 30/50
506/506 [==============================] - 0s 51us/step - loss: 92.7222
Epoch 31/50
506/506 [==============================] - 0s 57us/step - loss: 89.5669
Epoch 32/50
506/506 [==============================] - 0s 51us/step - loss: 86.9573
Epoch 33/50
506/506 [==============================] - 0s 69us/step - loss: 84.5068
Epoch 34/50
506/506 [==============================] - 0s 67us/step - loss: 82.1699
Epoch 35/50
506/506 [==============================] - 0s 53us/step - loss: 80.0550
Epoch 36/50
506/506 [==============================] - 0s 59us/step - loss: 78.0924
Epoch 37/50
506/506 [==============================] - 0s 49us/step - loss: 76.3332
Epoch 38/50
506/506 [==============================] - 0s 61us/step - loss: 74.6853
Epoch 39/50
506/506 [==============================] - 0s 53us/step - loss: 73.1529
Epoch 40/50
506/506 [==============================] - 0s 53us/step - loss: 71.7277
Epoch 41/50
506/506 [==============================] - 0s 55us/step - loss: 70.4601
Epoch 42/50
506/506 [==============================] - 0s 53us/step - loss: 69.2743
Epoch 43/50
506/506 [==============================] - 0s 61us/step - loss: 67.9983
Epoch 44/50
506/506 [==============================] - 0s 51us/step - loss: 66.9990
Epoch 45/50
506/506 [==============================] - 0s 61us/step - loss: 65.9917
Epoch 46/50
506/506 [==============================] - 0s 51us/step - loss: 64.9384
Epoch 47/50
506/506 [==============================] - 0s 65us/step - loss: 63.8823
Epoch 48/50
506/506 [==============================] - 0s 53us/step - loss: 62.9379
Epoch 49/50
506/506 [==============================] - 0s 55us/step - loss: 61.8970
Epoch 50/50
506/506 [==============================] - 0s 53us/step - loss: 60.9638
```

Out[129]:

```
<keras.callbacks.callbacks.History at 0x1c67baee788>
```

In [130]:

```python
subset2_test = np.array([18.0, 6.575, 4.0900, 1, 15.3, 4.98])
Result = print(m2.predict(subset2_test.reshape(1,6), batch_size=1))
Result
```

```
[[24.310076]]
```

In [72]:

```python
#Let's remind ourselves of the dataset.
df.describe()
```

Out[72]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000
75%	3.677082	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000

In [ ]:

```python

```

Let's do some preprocessing.

In [73]:

```python
import sklearn.preprocessing as pre
# We do some preprocessing in order to normalise the data.  Preprocessing enables downs
```

In [74]:

```python
df.head()
```

Out[74]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LST
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.

In [75]:

```
pre.scale(df)
```

Out[75]:

```
array([[-0.41978194,  0.28482986, -1.2879095 , ...,  0.44105193,
        -1.0755623 ,  0.15968566],
       [-0.41733926, -0.48772236, -0.59338101, ...,  0.44105193,
        -0.49243937, -0.10152429],
       [-0.41734159, -0.48772236, -0.59338101, ...,  0.39642699,
        -1.2087274 ,  1.32424667],
       ...,
       [-0.41344658, -0.48772236,  0.11573841, ...,  0.44105193,
        -0.98304761,  0.14880191],
       [-0.40776407, -0.48772236,  0.11573841, ...,  0.4032249 ,
        -0.86530163, -0.0579893 ],
       [-0.41500016, -0.48772236,  0.11573841, ...,  0.44105193,
        -0.66905833, -1.15724782]])
```

In [76]:

```
xscale = pd.DataFrame(pre.scale(df), columns=df.columns)
xscale
```

Out[76]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	
0	-0.419782	0.284830	-1.287909	-0.272599	-0.144217	0.413672	-0.120013	0.140214	-0.9
1	-0.417339	-0.487722	-0.593381	-0.272599	-0.740262	0.194274	0.367166	0.557160	-0.8
2	-0.417342	-0.487722	-0.593381	-0.272599	-0.740262	1.282714	-0.265812	0.557160	-0.8
3	-0.416750	-0.487722	-1.306878	-0.272599	-0.835284	1.016303	-0.809889	1.077737	-0.7
4	-0.412482	-0.487722	-1.306878	-0.272599	-0.835284	1.228577	-0.511180	1.077737	-0.7
...	...	...	...	...	...	...	...	...	
501	-0.413229	-0.487722	0.115738	-0.272599	0.158124	0.439316	0.018673	-0.625796	-0.9
502	-0.415249	-0.487722	0.115738	-0.272599	0.158124	-0.234548	0.288933	-0.716639	-0.9
503	-0.413447	-0.487722	0.115738	-0.272599	0.158124	0.984960	0.797449	-0.773684	-0.9
504	-0.407764	-0.487722	0.115738	-0.272599	0.158124	0.725672	0.736996	-0.668437	-0.9
505	-0.415000	-0.487722	0.115738	-0.272599	0.158124	-0.362767	0.434732	-0.613246	-0.9

506 rows × 14 columns

In [77]:

```
xscale.describe()
```

Out[77]:

	CRIM	ZN	INDUS	CHAS	NOX	RM
count	5.060000e+02	5.060000e+02	5.060000e+02	5.060000e+02	5.060000e+02	5.060000e+02
mean	-8.688702e-17	3.306534e-16	2.804081e-16	-3.100287e-16	-8.071058e-16	-5.978968e-17
std	1.000990e+00	1.000990e+00	1.000990e+00	1.000990e+00	1.000990e+00	1.000990e+00
min	-4.197819e-01	-4.877224e-01	-1.557842e+00	-2.725986e-01	-1.465882e+00	-3.880249e+00
25%	-4.109696e-01	-4.877224e-01	-8.676906e-01	-2.725986e-01	-9.130288e-01	-5.686303e-01
50%	-3.906665e-01	-4.877224e-01	-2.110985e-01	-2.725986e-01	-1.442174e-01	-1.084655e-01
75%	7.396560e-03	4.877224e-02	1.015999e+00	-2.725986e-01	5.986790e-01	4.827678e-01
max	9.933931e+00	3.804234e+00	2.422565e+00	3.668398e+00	2.732346e+00	3.555044e+00

In [78]:

```
xscale.mean() < 0.00001
```

Out[78]:

```
CRIM       True
ZN         True
INDUS      True
CHAS       True
NOX        True
RM         True
AGE        True
DIS        True
RAD        True
TAX        True
PTRATIO    True
B          True
LSTAT      True
MEDV       True
dtype: bool
```

In [79]:

```
xscale.var()
```

Out[79]:

```
CRIM       1.00198
ZN         1.00198
INDUS      1.00198
CHAS       1.00198
NOX        1.00198
RM         1.00198
AGE        1.00198
DIS        1.00198
RAD        1.00198
TAX        1.00198
PTRATIO    1.00198
B          1.00198
LSTAT      1.00198
MEDV       1.00198
dtype: float64
```

In [80]:

```
scaler = pre.StandardScaler()
scaler.fit(xscale)
scaler.mean_,scaler.scale_
```

Out[80]:

```
(array([-1.12338772e-16,  7.89881994e-17,  2.10635198e-16, -3.51058664e-17,
        -2.80846931e-16, -4.56376263e-17, -1.47444639e-16, -8.42540793e-17,
        -1.12338772e-16,  0.00000000e+00, -4.21270397e-16, -7.44244367e-16,
        -3.08931624e-16, -5.19566823e-16]),
 array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]))
```

In [81]:

```
xscale.std()
```

Out[81]:

```
CRIM       1.00099
ZN         1.00099
INDUS      1.00099
CHAS       1.00099
NOX        1.00099
RM         1.00099
AGE        1.00099
DIS        1.00099
RAD        1.00099
TAX        1.00099
PTRATIO    1.00099
B          1.00099
LSTAT      1.00099
MEDV       1.00099
dtype: float64
```

In [82]:

```
# To confirm at we are more or less at zero for the mean (allowing for floating point
xscale.mean()
```

Out[82]:

```
CRIM      -8.688702e-17
ZN         3.306534e-16
INDUS      2.804081e-16
CHAS      -3.100287e-16
NOX       -8.071058e-16
RM        -5.978968e-17
AGE       -2.650493e-16
DIS        8.293761e-17
RAD        1.514379e-15
TAX       -9.934960e-16
PTRATIO    4.493551e-16
B         -1.451408e-16
LSTAT     -1.595123e-16
MEDV      -4.247810e-16
dtype: float64
```

In [83]:

```
#Check to confirm that Scaler is producing the same result as above with 'xscale'
#Xscaler = pd.DataFrame(scaler.transform(xscale), columns=xscale.columns)
#Xscaler
```

In [84]:

```
test3 = df.iloc[0,:]
test3 = np.array([test3])
test3
```

Out[84]:

```
array([[6.320e-03, 1.800e+01, 2.310e+00, 0.000e+00, 5.380e-01, 6.575e+00,
        6.520e+01, 4.090e+00, 1.000e+00, 2.960e+02, 1.530e+01, 3.969e+02,
        4.980e+00, 2.400e+01]])
```

In [ ]:

```

```

In [ ]:

```

```

In [85]:

```
# test our transform on some of the original dataset to confirm.

scaler.transform(test3)

```

Out[85]:

```
array([[6.32000000e-03, 1.80000000e+01, 2.31000000e+00, 3.51058664e-17,
        5.38000000e-01, 6.57500000e+00, 6.52000000e+01, 4.09000000e+00,
        1.00000000e+00, 2.96000000e+02, 1.53000000e+01, 3.96900000e+02,
        4.98000000e+00, 2.40000000e+01]])
```

Whitening the Data

In [86]:

```
df.head()
```

Out[86]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LST.
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.

In [87]:

```
df.corr()
```

Out[87]:

	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
MEDV	-0.388305	0.360445	-0.483725	0.175260	-0.427321	0.695360	-0.376955	0.249929

In [88]:

```
import sklearn.decomposition as dec
```

In [89]:

```
pca = dec.PCA(n_components=14, whiten=True)
```

In [90]:

```
pca.fit(df)
```

Out[90]:

```
PCA(copy=True, iterated_power='auto', n_components=14, random_state=None,
    svd_solver='auto', tol=0.0, whiten=True)
```

In [91]:

```python
xwhite = pd.DataFrame(pca.transform(df), columns=df.columns)
xwhite
```

Out[91]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	
0	-0.681508	-0.070059	-0.113950	0.320981	0.186845	-0.030093	-1.195454	-1.554510	1.2
1	-0.960167	0.128689	-1.083597	0.073223	-0.152216	0.083313	-0.712645	-0.884073	-0.1
2	-0.964469	0.177350	-0.562518	-0.593586	1.212336	0.604822	0.120108	-0.374935	0.2
3	-1.083226	0.230959	-0.212275	-1.175221	0.978605	0.776677	-0.842677	-0.216832	0.2
4	-1.083079	0.202770	-0.435360	-0.830369	1.184552	0.855421	-0.751467	0.457089	0.5
...	...	...	...	...	...	...	...	...	
501	-0.788613	0.073659	-0.739263	-0.353494	-0.139965	-0.127916	0.613094	-0.880804	-0.2
502	-0.792938	0.013891	-0.945938	-0.062415	-0.226083	-0.185941	0.353956	-1.315370	-0.4
503	-0.787062	0.012244	-1.301361	0.500821	0.393902	-0.039150	0.121196	-1.976862	-0.5
504	-0.781771	0.054364	-1.261120	0.429291	0.160436	-0.113558	0.081338	-1.964207	-0.5
505	-0.789888	0.014482	-1.071475	0.066325	-0.950501	-0.485126	-0.248381	-2.543759	-1.0

506 rows × 14 columns

In [92]:

```
xwhite.corr().round()
```

Out[92]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
CRIM	1.0	-0.0	-0.0	-0.0	0.0	-0.0	-0.0	0.0	-0.0	0.0	0.0	-0.0	0.0
ZN	-0.0	1.0	0.0	0.0	-0.0	0.0	-0.0	0.0	-0.0	0.0	-0.0	-0.0	-0.0
INDUS	-0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.0	0.0	-0.0	0.0
CHAS	-0.0	0.0	0.0	1.0	-0.0	-0.0	-0.0	0.0	0.0	-0.0	-0.0	-0.0	-0.0
NOX	0.0	-0.0	0.0	-0.0	1.0	-0.0	0.0	0.0	-0.0	0.0	-0.0	0.0	0.0
RM	-0.0	0.0	0.0	-0.0	-0.0	1.0	-0.0	0.0	-0.0	0.0	0.0	0.0	0.0
AGE	-0.0	-0.0	0.0	-0.0	0.0	-0.0	1.0	-0.0	0.0	0.0	0.0	0.0	0.0
DIS	0.0	0.0	0.0	0.0	0.0	0.0	-0.0	1.0	0.0	0.0	-0.0	-0.0	0.0
RAD	-0.0	-0.0	0.0	0.0	-0.0	-0.0	0.0	0.0	1.0	-0.0	-0.0	0.0	-0.0
TAX	0.0	0.0	-0.0	-0.0	0.0	0.0	0.0	0.0	-0.0	1.0	-0.0	0.0	0.0
PTRATIO	0.0	-0.0	0.0	-0.0	-0.0	0.0	0.0	-0.0	-0.0	-0.0	1.0	-0.0	-0.0
B	-0.0	-0.0	-0.0	-0.0	0.0	0.0	0.0	-0.0	0.0	0.0	-0.0	1.0	-0.0
LSTAT	0.0	-0.0	0.0	-0.0	0.0	0.0	0.0	0.0	-0.0	0.0	-0.0	-0.0	1.0
MEDV	-0.0	0.0	0.0	-0.0	0.0	0.0	-0.0	-0.0	0.0	0.0	0.0	0.0	-0.0

In [93]:

```
xwhite.mean().round()
```

Out[93]:

```
CRIM       0.0
ZN         0.0
INDUS      0.0
CHAS      -0.0
NOX       -0.0
RM        -0.0
AGE        0.0
DIS       -0.0
RAD       -0.0
TAX        0.0
PTRATIO   -0.0
B         -0.0
LSTAT      0.0
MEDV      -0.0
dtype: float64
```

In [ ]:

```

```

Let's run a new neural network with different inputs and outputs to predict the MEDV value.

In [94]:

```
df.head()
```

Out[94]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LST
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.

In [95]:

```
dfa = df.iloc[:, 0:13]
dfa.head()
```

Out[95]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LST
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.

In [96]:

```
dfb = df.iloc[:,13:]
dfb.head()
```

Out[96]:

	MEDV
0	24.0
1	21.6
2	34.7
3	33.4
4	36.2

In [ ]:

```

```

In [136]:

```
# We aim to make our neural net convert the 'dfa' values into a 'MEDV' value.
model = ks.models.Sequential()
model.add(ks.layers.Dense(13,activation='relu', input_shape = (13,)))
model.add(ks.layers.Dense(13, activation = 'relu'))
model.add(ks.layers.Dense(1))
model.compile(loss = 'mean_squared_error',optimizer='adagrad', metrics=['accuracy'])
```

In [137]:

```
#Here's where we deviate from earlier work, by modelling the full dataset and then spl
import sklearn.model_selection as mod
```

In [138]:

```
# split the dataframe inputs and outputs into training and test sets.
inputs_train, inputs_test, outputs_train, outputs_test = mod.train_test_split(dfa, dfb
```

In [139]:

```
# Train the neural network
model.fit(inputs_train, outputs_train, epochs =50, batch_size=10)
```

```
Epoch 1/50
404/404 [==============================] - 0s 864us/step - loss: 1626.0222 -
accuracy: 0.0000e+00
Epoch 2/50
404/404 [==============================] - 0s 147us/step - loss: 105.4196 -
accuracy: 0.0050
Epoch 3/50
404/404 [==============================] - 0s 156us/step - loss: 87.0988 - a
ccuracy: 0.0025
Epoch 4/50
404/404 [==============================] - 0s 158us/step - loss: 77.5383 - a
ccuracy: 0.0074
Epoch 5/50
404/404 [==============================] - 0s 156us/step - loss: 72.5729 - a
ccuracy: 0.0050
Epoch 6/50
404/404 [==============================] - 0s 165us/step - loss: 69.6049 - a
ccuracy: 0.0124
Epoch 7/50
404/404 [==============================] - 0s 173us/step - loss: 66.5320 - a
ccuracy: 0.0074
Epoch 8/50
404/404 [==============================] - 0s 165us/step - loss: 64.6521 - a
ccuracy: 0.0000e+00
Epoch 9/50
404/404 [==============================] - 0s 156us/step - loss: 63.1498 - a
ccuracy: 0.0099
Epoch 10/50
404/404 [==============================] - 0s 158us/step - loss: 62.0146 - a
ccuracy: 0.0074
Epoch 11/50
404/404 [==============================] - 0s 180us/step - loss: 60.8838 - a
ccuracy: 0.0074
Epoch 12/50
404/404 [==============================] - 0s 153us/step - loss: 60.2571 - a
ccuracy: 0.0025
Epoch 13/50
404/404 [==============================] - 0s 163us/step - loss: 58.8418 - a
ccuracy: 0.0124
Epoch 14/50
404/404 [==============================] - 0s 163us/step - loss: 58.7821 - a
ccuracy: 0.0050
Epoch 15/50
404/404 [==============================] - 0s 163us/step - loss: 57.1290 - a
ccuracy: 0.0050
Epoch 16/50
404/404 [==============================] - 0s 163us/step - loss: 57.1200 - a
ccuracy: 0.0099
Epoch 17/50
404/404 [==============================] - 0s 160us/step - loss: 55.7314 - a
ccuracy: 0.0050
Epoch 18/50
404/404 [==============================] - 0s 163us/step - loss: 54.8670 - a
ccuracy: 0.0050
Epoch 19/50
404/404 [==============================] - 0s 165us/step - loss: 54.4667 - a
```

```
ccuracy: 0.0074
Epoch 20/50
404/404 [==============================] - 0s 156us/step - loss: 53.1067 - a
ccuracy: 0.0050
Epoch 21/50
404/404 [==============================] - 0s 151us/step - loss: 52.5945 - a
ccuracy: 0.0149
Epoch 22/50
404/404 [==============================] - 0s 153us/step - loss: 51.2654 - a
ccuracy: 0.0124
Epoch 23/50
404/404 [==============================] - 0s 165us/step - loss: 51.6269 - a
ccuracy: 0.0074
Epoch 24/50
404/404 [==============================] - 0s 163us/step - loss: 50.5150 - a
ccuracy: 0.0050
Epoch 25/50
404/404 [==============================] - 0s 163us/step - loss: 50.3064 - a
ccuracy: 0.0124
Epoch 26/50
404/404 [==============================] - 0s 156us/step - loss: 49.5958 - a
ccuracy: 0.0124
Epoch 27/50
404/404 [==============================] - 0s 160us/step - loss: 49.6719 - a
ccuracy: 0.0074
Epoch 28/50
404/404 [==============================] - 0s 168us/step - loss: 49.1082 - a
ccuracy: 0.0050
Epoch 29/50
404/404 [==============================] - 0s 160us/step - loss: 49.2748 - a
ccuracy: 0.0050
Epoch 30/50
404/404 [==============================] - 0s 163us/step - loss: 48.2938 - a
ccuracy: 0.0025
Epoch 31/50
404/404 [==============================] - 0s 168us/step - loss: 48.5714 - a
ccuracy: 0.0074
Epoch 32/50
404/404 [==============================] - 0s 164us/step - loss: 47.2997 - a
ccuracy: 0.0074
Epoch 33/50
404/404 [==============================] - 0s 160us/step - loss: 47.6601 - a
ccuracy: 0.0074
Epoch 34/50
404/404 [==============================] - 0s 163us/step - loss: 47.2675 - a
ccuracy: 0.0124
Epoch 35/50
404/404 [==============================] - 0s 160us/step - loss: 46.9944 - a
ccuracy: 0.0074
Epoch 36/50
404/404 [==============================] - 0s 160us/step - loss: 47.1465 - a
ccuracy: 0.0124
Epoch 37/50
404/404 [==============================] - 0s 158us/step - loss: 46.3602 - a
ccuracy: 0.0025
Epoch 38/50
404/404 [==============================] - 0s 158us/step - loss: 46.6716 - a
ccuracy: 0.0025
Epoch 39/50
404/404 [==============================] - 0s 160us/step - loss: 45.8019 - a
ccuracy: 0.0074
```

```
Epoch 40/50
404/404 [==============================] - 0s 153us/step - loss: 45.5781 - a
ccuracy: 0.0050
Epoch 41/50
404/404 [==============================] - 0s 160us/step - loss: 44.8022 - a
ccuracy: 0.0124
Epoch 42/50
404/404 [==============================] - 0s 160us/step - loss: 44.9423 - a
ccuracy: 0.0099
Epoch 43/50
404/404 [==============================] - 0s 160us/step - loss: 44.4872 - a
ccuracy: 0.0124
Epoch 44/50
404/404 [==============================] - 0s 163us/step - loss: 44.5035 - a
ccuracy: 0.0099
Epoch 45/50
404/404 [==============================] - 0s 158us/step - loss: 44.2192 - a
ccuracy: 0.0025
Epoch 46/50
404/404 [==============================] - 0s 173us/step - loss: 44.2132 - a
ccuracy: 0.0074
Epoch 47/50
404/404 [==============================] - 0s 165us/step - loss: 44.2645 - a
ccuracy: 0.0099
Epoch 48/50
404/404 [==============================] - 0s 158us/step - loss: 43.7247 - a
ccuracy: 0.0099
Epoch 49/50
404/404 [==============================] - 0s 158us/step - loss: 43.1164 - a
ccuracy: 0.0074
Epoch 50/50
404/404 [==============================] - 0s 158us/step - loss: 43.1254 - a
ccuracy: 0.0124
```

Out[139]:

```
<keras.callbacks.callbacks.History at 0x1c67c635f48>
```

Evaluate the acuracy of the training set

In [140]:

```python
#To test our model we will use the 1st row of the dataset as our testcase (without the
#As we know what the actual data is we will then be able to compare the actual with the
testn_data = np.array([0.00632,18.0,2.31, 0,0.538, 6.575, 65.2, 4.0900, 1, 296, 15.3,
print(model.predict(testn_data.reshape(1,13), batch_size=1))
```

```
[[29.031414]]
```

In [ ]:

```python

```

In [102]:

```python
# Let's set the first 5 rows and the last 5 rows as our own test verification dataset
```

In [103]:

```
df.head()
```

Out[103]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LST
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.

In [104]:

```
top_5_In=df.iloc[0:5,0:13]
top_5_In = np.array(top_5_In)
top_5_In
```

Out[104]:

```
array([[6.3200e-03, 1.8000e+01, 2.3100e+00, 0.0000e+00, 5.3800e-01,
        6.5750e+00, 6.5200e+01, 4.0900e+00, 1.0000e+00, 2.9600e+02,
        1.5300e+01, 3.9690e+02, 4.9800e+00],
       [2.7310e-02, 0.0000e+00, 7.0700e+00, 0.0000e+00, 4.6900e-01,
        6.4210e+00, 7.8900e+01, 4.9671e+00, 2.0000e+00, 2.4200e+02,
        1.7800e+01, 3.9690e+02, 9.1400e+00],
       [2.7290e-02, 0.0000e+00, 7.0700e+00, 0.0000e+00, 4.6900e-01,
        7.1850e+00, 6.1100e+01, 4.9671e+00, 2.0000e+00, 2.4200e+02,
        1.7800e+01, 3.9283e+02, 4.0300e+00],
       [3.2370e-02, 0.0000e+00, 2.1800e+00, 0.0000e+00, 4.5800e-01,
        6.9980e+00, 4.5800e+01, 6.0622e+00, 3.0000e+00, 2.2200e+02,
        1.8700e+01, 3.9463e+02, 2.9400e+00],
       [6.9050e-02, 0.0000e+00, 2.1800e+00, 0.0000e+00, 4.5800e-01,
        7.1470e+00, 5.4200e+01, 6.0622e+00, 3.0000e+00, 2.2200e+02,
        1.8700e+01, 3.9690e+02, 5.3300e+00]])
```

In [105]:

```
top_5_Out = df.iloc[0:5,13:]
top_5_Out = np.array(top_5_Out)
top_5_Out
```

Out[105]:

```
array([[24. ],
       [21.6],
       [34.7],
       [33.4],
       [36.2]])
```

In [106]:

```
print(model.predict(top_5_In.reshape(5,13), batch_size=5))
```

```
[[27.679256]
 [26.698673]
 [28.387997]
 [31.411617]
 [30.638226]]
```

In [ ]:

```

```

In [107]:

```
df.tail()
```

Out[107]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LS
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273	21.0	391.99	9
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273	21.0	396.90	9
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273	21.0	396.90	5
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273	21.0	393.45	6
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273	21.0	396.90	7

In [108]:

```
bot_5_In=df.iloc[501:506,0:13]
bot_5_In = np.array(bot_5_In)
bot_5_In
```

Out[108]:

```
array([[6.2630e-02, 0.0000e+00, 1.1930e+01, 0.0000e+00, 5.7300e-01,
        6.5930e+00, 6.9100e+01, 2.4786e+00, 1.0000e+00, 2.7300e+02,
        2.1000e+01, 3.9199e+02, 9.6700e+00],
       [4.5270e-02, 0.0000e+00, 1.1930e+01, 0.0000e+00, 5.7300e-01,
        6.1200e+00, 7.6700e+01, 2.2875e+00, 1.0000e+00, 2.7300e+02,
        2.1000e+01, 3.9690e+02, 9.0800e+00],
       [6.0760e-02, 0.0000e+00, 1.1930e+01, 0.0000e+00, 5.7300e-01,
        6.9760e+00, 9.1000e+01, 2.1675e+00, 1.0000e+00, 2.7300e+02,
        2.1000e+01, 3.9690e+02, 5.6400e+00],
       [1.0959e-01, 0.0000e+00, 1.1930e+01, 0.0000e+00, 5.7300e-01,
        6.7940e+00, 8.9300e+01, 2.3889e+00, 1.0000e+00, 2.7300e+02,
        2.1000e+01, 3.9345e+02, 6.4800e+00],
       [4.7410e-02, 0.0000e+00, 1.1930e+01, 0.0000e+00, 5.7300e-01,
        6.0300e+00, 8.0800e+01, 2.5050e+00, 1.0000e+00, 2.7300e+02,
        2.1000e+01, 3.9690e+02, 7.8800e+00]])
```

In [109]:

```
bot_5_Out=df.iloc[501:506,13:]
bot_5_Out= np.array(bot_5_Out)
print(bot_5_Out)
type(bot_5_Out)
```

```
[[22.4]
 [20.6]
 [23.9]
 [22. ]
 [11.9]]
```

Out[109]:

```
numpy.ndarray
```

In [110]:

```
bot_5_predict = print(model.predict(bot_5_In.reshape(5,13), batch_size=5))
bot_5_predict = np.array(bot_5_predict)
type(bot_5_predict)
```

```
[[24.955082]
 [25.248653]
 [26.037075]
 [25.514858]
 [25.478313]]
```

Out[110]:

```
numpy.ndarray
```

In [ ]:

```

```

List of references used in this notebook.

https://en.wikipedia.org/wiki/Sampling_bias (https://en.wikipedia.org/wiki/Sampling_bias)
https://pro.arcgis.com/en/pro-app/help/analysis/geoprocessing/charts/box-plot.htm
(https://pro.arcgis.com/en/pro-app/help/analysis/geoprocessing/charts/box-plot.htm)
https://machinelearningmastery.com/ (https://machinelearningmastery.com/)
https://www.analyticsvidhya.com/machine-learning/ (https://www.analyticsvidhya.com/machine-learning/)
https://towardsdatascience.com/understanding-and-reducing-bias-in-machine-learning-6565e23900ac
(https://towardsdatascience.com/understanding-and-reducing-bias-in-machine-learning-6565e23900ac)
https://matplotlib.org/ (https://matplotlib.org/) https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html) https://stackoverflow.com/questions/21516089/difference-between-two-numpy-arrays-in-python/21516130 (https://stackoverflow.com/questions/21516089/difference-between-two-numpy-arrays-in-python/21516130) https://en.wikipedia.org/wiki/Sigmoid_function
(https://en.wikipedia.org/wiki/Sigmoid_function) https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7 (https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7)
https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
(https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6)
https://www.freecodecamp.org/news/machine-learning-mean-squared-error-regression-line-c7dde9a26b93/

(https://www.freecodecamp.org/news/machine-learning-mean-squared-error-regression-line-c7dde9a26b93/) https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-logarithmic-error (https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-logarithmic-error) https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ (https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/) https://en.wikipedia.org/wiki/Sigmoid_function (https://en.wikipedia.org/wiki/Sigmoid_function)

ENDS.