AnalyzingBostonHousing

0.1 Importing necessary libraries

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import random
from sklearn import linear_model
from sklearn.metrics import mean_squared_error
```

0.1.1 Importing the data for analysis

The dataset used here is about housing prices in the Boston area, including suburbs, as drawn from the Boston Standard Metropolitan Statistical Area (SMSA). Data was collected in 1970, and it is a commonly used dataset for practicing regression.

[2]:
```
df = pd.read_csv('boston_house_prices.csv', header=1)
df
```

[2]:
```
        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]
```

0.2 Creating a dataframe for columns and what they mean for easiness of access

Note that for regression, the MEDV feature is used as the target. Other features serve the purpose to analyse how much these variables affect the housing prices.

[3]:
```
legends = ['per capita crime rate by town','proportion of residential land zoned␣
↪for lots over 25,000 sq.ft','proportion of non-retail business acres per town',
          'Charles River dummy variable (= 1 if tract bounds river; 0␣
↪otherwise)','nitric oxides concentration (parts per 10 million)','average␣
↪number of rooms per dwelling',
          'proportion of owner-occupied units built prior to 1940','weighted␣
↪distances to five Boston employment centers','index of accessibility to radial␣
↪highways',
          'full-value property-tax rate per $10,000','pupil-teacher ratio by␣
↪town','1000(Bk-0.63)2 where Bk is the proportion of blacks by town', '% lower␣
↪status of the population',
          'Median value of owner-occupied homes in $1000s']
pd.set_option('display.max_colwidth', None)
pd.DataFrame({'Columns':df.columns, 'Legends': legends})
```

[3]:
```
    Columns  \
0      CRIM
1        ZN
2     INDUS
3      CHAS
4       NOX
5        RM
6       AGE
7       DIS
8       RAD
9       TAX
10  PTRATIO
11        B
12    LSTAT
13     MEDV

                                                  Legends
0                                per capita crime rate by town
```

```
1          proportion of residential land zoned for lots over 25,000 sq.ft
2                          proportion of non-retail business acres per town
3     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
4                        nitric oxides concentration (parts per 10 million)
5                                     average number of rooms per dwelling
6                     proportion of owner-occupied units built prior to 1940
7                        weighted distances to five Boston employment centers
8                                 index of accessibility to radial highways
9                                  full-value property-tax rate per $10,000
10                                              pupil-teacher ratio by town
11              1000(Bk-0.63)2 where Bk is the proportion of blacks by town
12                                          % lower status of the population
13                            Median value of owner-occupied homes in $1000s
```

0.3 Exploring the data.

0.3.1 Table is all numeric and presents no missing values. The presence of noise is more visible with the boxplot and histogram in the visualization.

[4]:
```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    int64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    int64
 9   TAX      506 non-null    int64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  MEDV     506 non-null    float64
dtypes: float64(11), int64(3)
memory usage: 55.5 KB
```

0.4 Identifying Statistics

[5]:
```python
np.round(df.describe(),2)
```

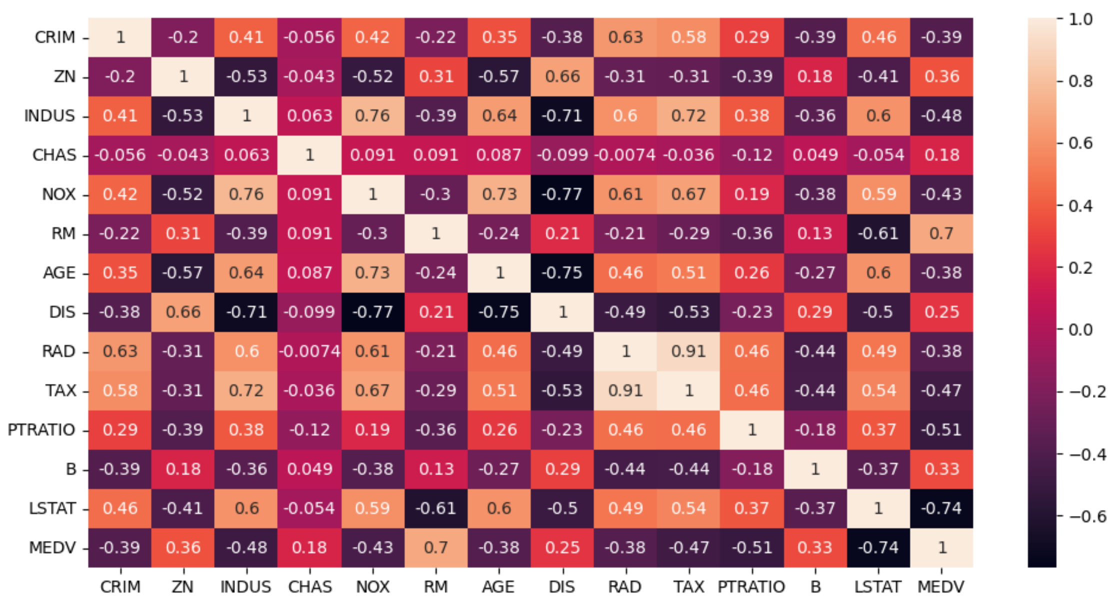
[5]:
```
        CRIM      ZN   INDUS    CHAS     NOX      RM     AGE     DIS     RAD  \
count 506.00  506.00  506.00  506.00  506.00  506.00  506.00  506.00  506.00
mean    3.61   11.36   11.14    0.07    0.55    6.28   68.57    3.80    9.55
std     8.60   23.32    6.86    0.25    0.12    0.70   28.15    2.11    8.71
min     0.01    0.00    0.46    0.00    0.38    3.56    2.90    1.13    1.00
25%     0.08    0.00    5.19    0.00    0.45    5.89   45.02    2.10    4.00
50%     0.26    0.00    9.69    0.00    0.54    6.21   77.50    3.21    5.00
75%     3.68   12.50   18.10    0.00    0.62    6.62   94.07    5.19   24.00
max    88.98  100.00   27.74    1.00    0.87    8.78  100.00   12.13   24.00

         TAX  PTRATIO       B   LSTAT    MEDV
count 506.00   506.00  506.00  506.00  506.00
mean  408.24    18.46  356.67   12.65   22.53
std   168.54     2.16   91.29    7.14    9.20
min   187.00    12.60    0.32    1.73    5.00
25%   279.00    17.40  375.38    6.95   17.02
50%   330.00    19.05  391.44   11.36   21.20
75%   666.00    20.20  396.22   16.96   25.00
max   711.00    22.00  396.90   37.97   50.00
```

0.5 Visualizing Statistics and Correlation

Creating a heatmap with correlation matrix to see which columns would probably suit better together for analysis or not.

[6]:
```python
plt.figure(figsize=(12,6))
sns.heatmap(df.corr(), annot=True)
```

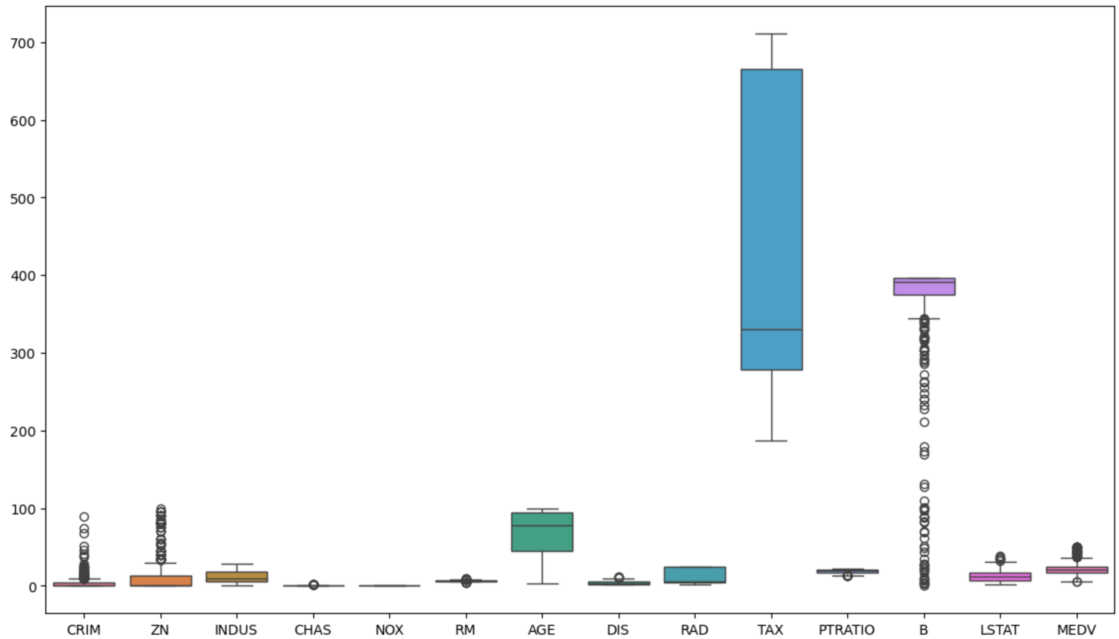
[6]: <Axes: >



First creating a general boxplot to investigate the presence of outliers. As it can be seen, outliers are highly present in multiple features. Since the tax and B features are distorting the rest of the data, another boxplot is created without it to better see other features.

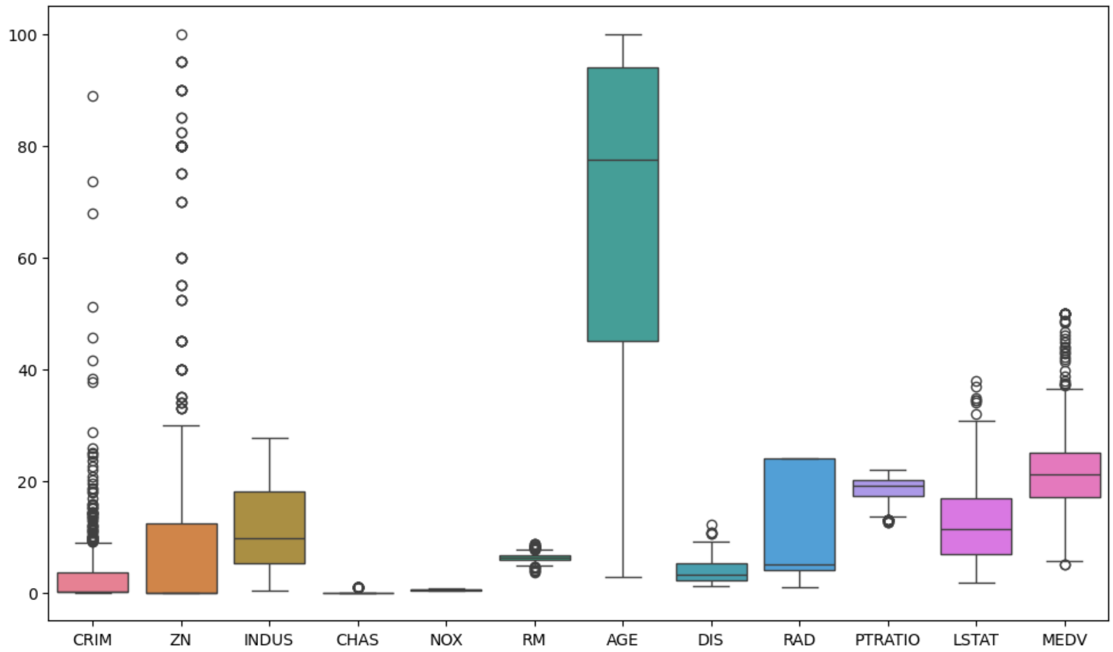
[7]:
```
plt.figure(figsize=(14,8))
sns.boxplot(df)
```

[7]: <Axes: >



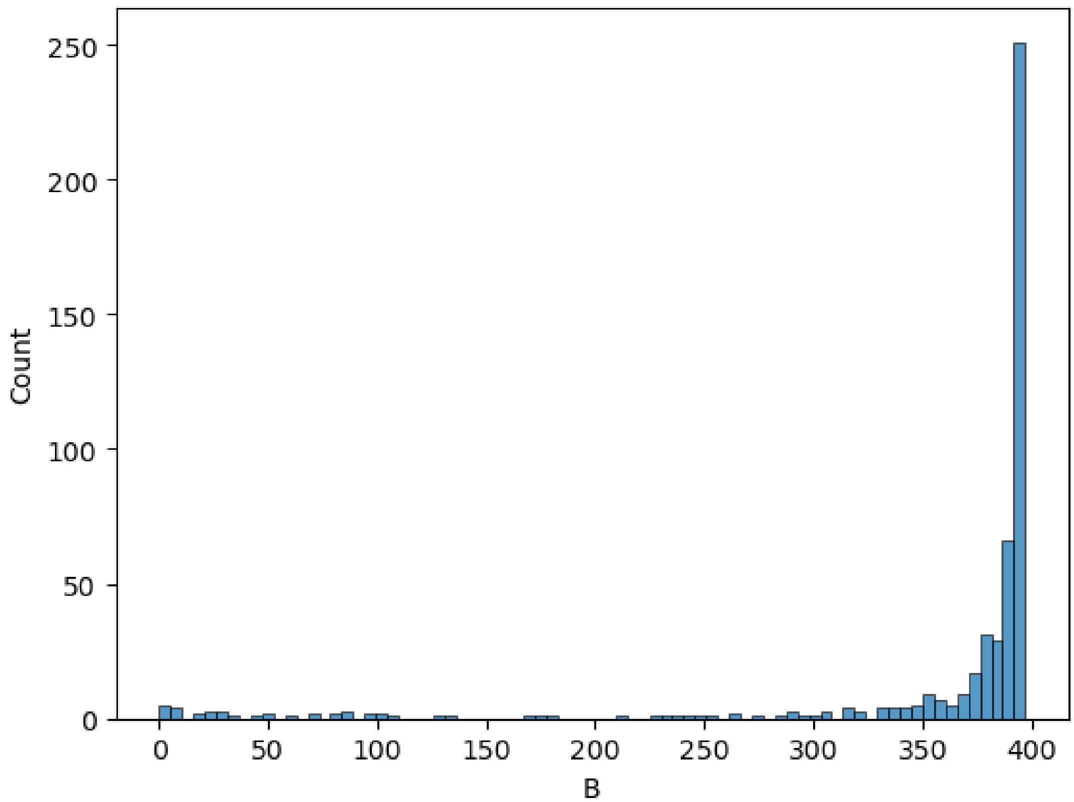
[8]:
```
df2 = df.drop(['TAX', 'B'], axis=1)
plt.figure(figsize=(12,7))
sns.boxplot(df2)
```

[8]: <Axes: >



[9]:
```
sns.histplot(df['B'])
```

[9]: <Axes: xlabel='B', ylabel='Count'>



Since B feature means the proportion of black people by town, without further specification, it can be seen that the reason for the strong outliers is due to a high concentration in specific towns with minorities spread in other areas. That results in outliers and a lot of distortion. Only further analysis would allow to determine how to react towards the outliers.

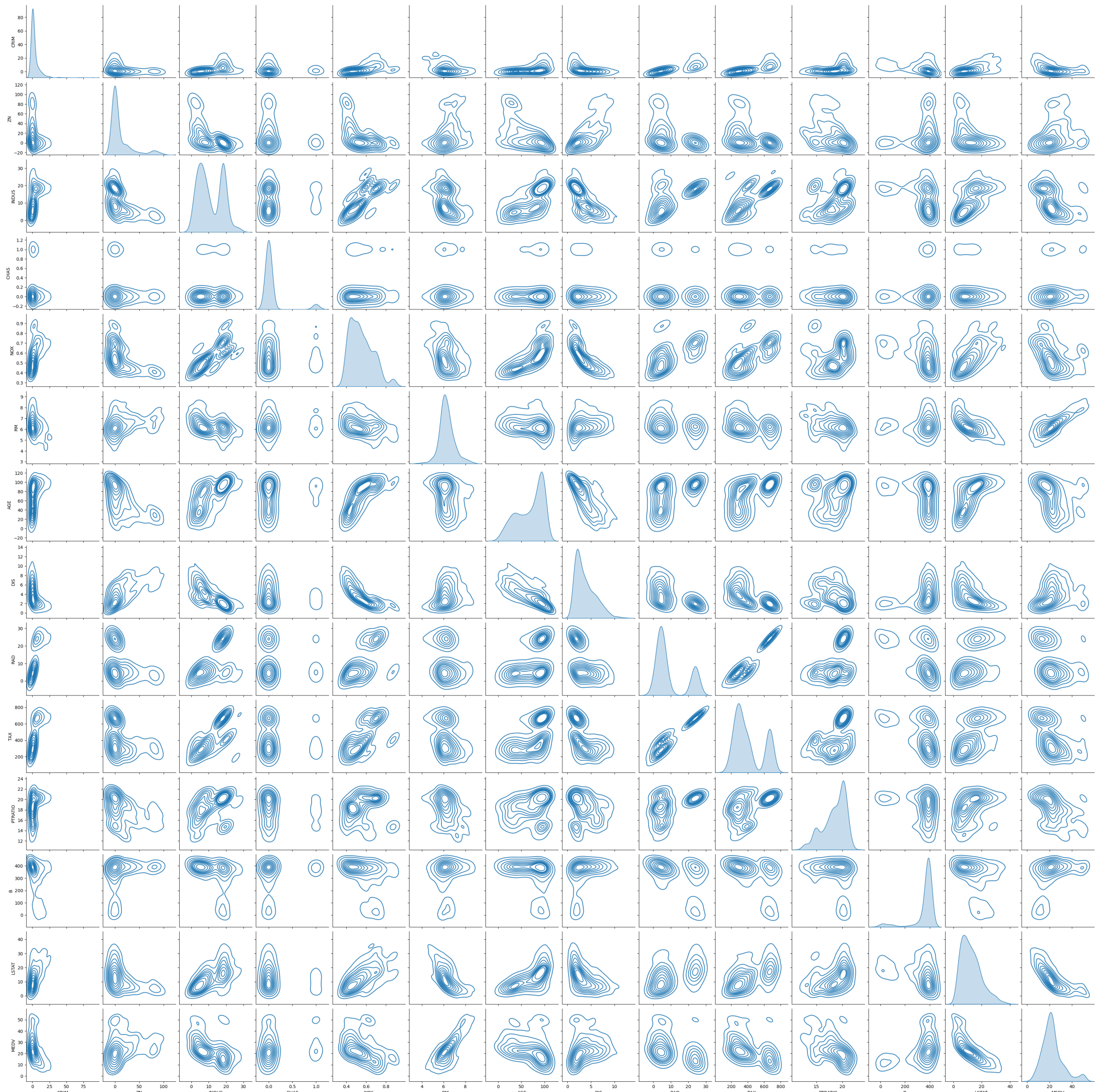
For the purpose of Regression Analysis, the biggest focus lies in finding whether there is a relationship between features, especially in relationship to the target feature, and whether there is multicollinearity among features. It isn't interesting to eliminate outliers altogether as there presence can be important for the analysis. However, in some cases, some outliers can be eliminated in order to find a better slope for the regression the fits better the rest of the data.

First the features need to be kept as it is. For a regression model, first it has to be decided whether a feature should be kept in the model, then a version with outliers and without could be compared to see how it affects the final prediction.

[10]:
```
plt.figure(figsize=(12,12))
sns.pairplot(df, height= 2.5, kind='kde')
```

[10]: <seaborn.axisgrid.PairGrid at 0x297c8af3950>

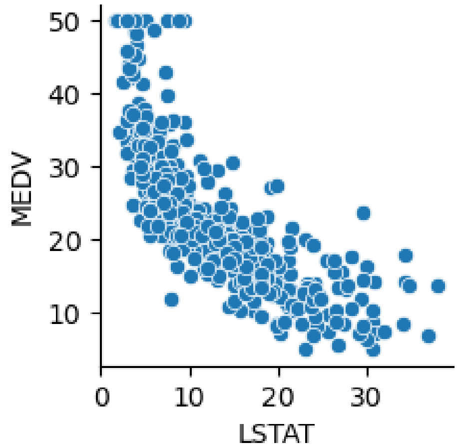
<Figure size 1200x1200 with 0 Axes>

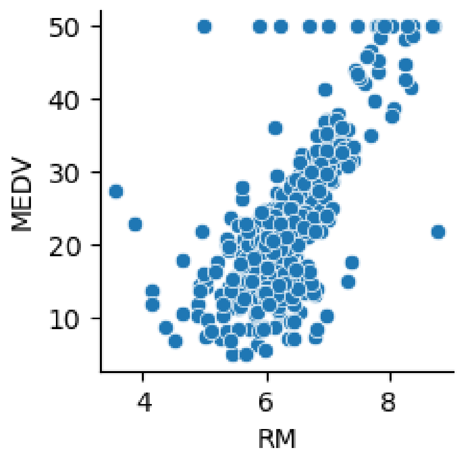


The pairplot creates a correlation matrix of features that allows to see better which features might be correlated or not. Most examples do not show a clear correlation between them. Regarding the target feature MEDV, there is a clear correlation with RM only. Correlation is also seen with the LSTAT column, but it might be non-linear.

The B feature appears to share multicollinearity with many features with some distortion caused by outliers. Its slope does not show any increase or decrease however. It is most likely that the analysis would need to proceed without this feature.

[11]:
```
sns.pairplot(x_vars='LSTAT', y_vars = 'MEDV', data=df)
sns.pairplot(x_vars='RM', y_vars ='MEDV', data=df)
plt.show()
```





Here the two scatterplots showing correlation with the target feature are analyzed in isolation from the other ones. Although the LSTAT is non-linear, the correlation is strong, a further analysis beyond the scope here would continue the investigation through the two features. The linearity is clear regarding the RM feature, however the differences in the values and the fact that the RM values vary so little might also be affecting the plot.

0.6 Dividing the data into test, validation and training

Shuffling data with pandas

[12]:
```
df2 = df.sample(frac=1)
X = df2['MEDV']
y = df2.drop('MEDV',axis=1)
```

[13]:
```
X
```

[13]:
```
18     20.2
243    23.7
205    22.6
68     17.4
181    36.2
       ...
26     16.6
156    13.1
503    23.9
255    20.9
278    29.1
Name: MEDV, Length: 506, dtype: float64
```

[14]:
```
y
```

[14]:
```
        CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD  TAX  \
18   0.80271   0.0   8.14     0  0.538  5.456  36.6  3.7965    4  307
243  0.12757  30.0   4.93     0  0.428  6.393   7.8  7.0355    6  300
205  0.13642   0.0  10.59     0  0.489  5.891  22.3  3.9454    4  277
68   0.13554  12.5   6.07     0  0.409  5.594  36.8  6.4980    4  345
181  0.06888   0.0   2.46     0  0.488  6.144  62.2  2.5979    3  193
..       ...   ...    ...   ...    ...    ...   ...     ...  ...  ...
26   0.67191   0.0   8.14     0  0.538  5.813  90.3  4.6820    4  307
156  2.44668   0.0  19.58     0  0.871  5.272  94.0  1.7364    5  403
503  0.06076   0.0  11.93     0  0.573  6.976  91.0  2.1675    1  273
255  0.03548  80.0   3.64     0  0.392  5.876  19.1  9.2203    1  315
278  0.07978  40.0   6.41     0  0.447  6.482  32.1  4.1403    4  254

     PTRATIO       B  LSTAT
18      21.0  288.99  11.69
243     16.6  374.71   5.19
205     18.6  396.90  10.87
68      18.9  396.90  13.09
181     17.8  396.90   9.45
..       ...     ...    ...
26      21.0  376.88  14.81
156     14.7   88.63  16.14
503     21.0  396.90   5.64
255     16.4  395.18   9.25
278     17.6  396.90   7.19

[506 rows x 13 columns]
```

[15]:
```
N = len(X)
X = (X.to_numpy()).reshape(-1,1)
y = y.to_numpy()
X_train,X_val,X_test = X[:3*N//5],X[3*N//5:4*N//5], X[4*N//5:]
```

```
y_train,y_val,y_test = y[:3*N//5],y[3*N//5:4*N//5], y[4*N//5:]
```

[16]:
```
len(X_train), len(X_val), len(X_test)
```

[16]: (303, 101, 102)

[17]:
```
506*0.6
```

[17]: 303.59999999999997

[18]:
```
506*0.2
```

[18]: 101.2

[19]:
```
y_train
```

[19]:
```
array([[8.0271e-01, 0.0000e+00, 8.1400e+00, ..., 2.1000e+01, 2.8899e+02,
         1.1690e+01],
        [1.2757e-01, 3.0000e+01, 4.9300e+00, ..., 1.6600e+01, 3.7471e+02,
         5.1900e+00],
        [1.3642e-01, 0.0000e+00, 1.0590e+01, ..., 1.8600e+01, 3.9690e+02,
         1.0870e+01],
        ...,
        [4.1238e-01, 0.0000e+00, 6.2000e+00, ..., 1.7400e+01, 3.7208e+02,
         6.3600e+00],
        [9.0650e-02, 2.0000e+01, 6.9600e+00, ..., 1.8600e+01, 3.9134e+02,
         1.3650e+01],
        [2.4980e-01, 0.0000e+00, 2.1890e+01, ..., 2.1200e+01, 3.9204e+02,
         2.1320e+01]])
```

0.7 Using Ridge Model

Creating a mean squared error function to be used separately

[20]:
```
def MSE(model,X,y):
    preds = model.predict(X)
    diffs = [(a-b)**2 for (a,b) in zip(preds,y)]
    return sum(sum(diffs)/len(diffs))/len(sum(diffs)/len(diffs))
```

Using the function for mean squared error from sklearn with all the features

[21]:
```
y_pred = linear_model.Ridge(0.001, fit_intercept=False).fit(X_train,y_train)
y_pred = y_pred.predict(X_val)
print(mean_squared_error(y_val, y_pred))
y_pred = linear_model.Ridge(0.001, fit_intercept=False).fit(X_train,y_train)
y_pred = y_pred.predict(X_test)
mean_squared_error(y_test,y_pred)
```

```
8008.645331464965
```

[21]: 7061.927127520113

The high results for validation and test show that even with the penalization done by the Ridge model, the full set of features does not present linearity and therefore cannot be used also in Ridge Regression. The chosen alpha present the best results in rounded values.

[22]:
```
bestModel = None
bestMSE = None
for lamb in [0.001,0.01, 1,10, 10000, 12000]:
    model = linear_model.Ridge(lamb, fit_intercept=False)
    model.fit(X_train,y_train)
    mseTrain = MSE(model, X_train,y_train)
    mseValid = MSE(model, X_val,y_val)
    mseTest = MSE(model, X_test,y_test)
    print("lambda = " + str(lamb) + " training/validation error = " +␣
 ↪str(mseTrain) + '/' + str(mseValid)
    + " test error = " + str(mseTest))
    if not bestModel or mseValid < bestMSE:
        bestMSE = mseValid
        bestModel = model
```

```
lambda = 0.001 training/validation error = 7213.3706058493535/8008.645331464965
test error = 7061.927127520113
lambda = 0.01 training/validation error = 7213.370605849412/8008.6452964211985
test error = 7061.927188867603
lambda = 1 training/validation error = 7213.370606432877/8008.641442260802 test
error = 7061.933937615167
lambda = 10 training/validation error = 7213.370664196011/8008.606463911042 test
error = 7061.995337410214
lambda = 10000 training/validation error = 7265.650272678696/8029.087767739629
test error = 7177.278329161023
lambda = 12000 training/validation error = 7287.06861152694/8045.658905986318
test error = 7210.188239567588
```

[23]:
```
bestMSE, bestModel
```

[23]: (8008.606463911042, Ridge(alpha=10, fit_intercept=False))

Multiple attempts showed that value of alpha is so similar, the best alpha is dependent on how the data is shuffled. It has alternated between 0.001 and 12000.

Repeating the process using only the features that showed linearity

[24]:
```
X = df2['MEDV']
y = df2[['RM', 'LSTAT']]
```

[25]:
```
N = len(X)
X = (X.to_numpy()).reshape(-1,1)
y = y.to_numpy()
```

```
X_train,X_val,X_test = X[:3*N//5],X[3*N//5:4*N//5], X[4*N//5:]
y_train,y_val,y_test = y[:3*N//5],y[3*N//5:4*N//5], y[4*N//5:]
```

[26]:
```
y_pred = linear_model.Ridge(0.001, fit_intercept=False).fit(X_train,y_train)
y_pred = y_pred.predict(X_val)
print(mean_squared_error(y_val, y_pred))
y_pred = linear_model.Ridge(0.001, fit_intercept=False).fit(X_train,y_train)
y_pred = y_pred.predict(X_test)
mean_squared_error(y_test,y_pred)
```

```
63.81448213675361
```

[26]:
```
60.909606197629294
```

The mean squared error reduced drastically, making it more suitable for regression.

[27]:
```
bestModel = None
bestMSE = None
for lamb in [0.001,0.01, 1,10, 10000, 12000]:
    model = linear_model.Ridge(lamb, fit_intercept=False)
    model.fit(X_train,y_train)
    mseTrain = MSE(model, X_train,y_train)
    mseValid = MSE(model, X_val,y_val)
    mseTest = MSE(model, X_test,y_test)
    print("lambda = " + str(lamb) + " training/validation error = " +␣
 ↪str(mseTrain) + '/' + str(mseValid)
    + " test error = " + str(mseTest))
    if not bestModel or mseValid < bestMSE:
        bestMSE = mseValid
        bestModel = model
```

```
lambda = 0.001 training/validation error = 58.66584036558601/63.81448213675361
test error = 60.909606197629294
lambda = 0.01 training/validation error = 58.66584036558624/63.81448170280746
test error = 60.90960627318088
lambda = 1 training/validation error = 58.66584036768764/63.81443397127413 test
error = 60.909614585832955
lambda = 10 training/validation error = 58.665840575729064/63.81400027980594
test error = 60.90969033484018
lambda = 10000 training/validation error = 58.854132339087094/63.564467363491204
test error = 61.172142182556705
lambda = 12000 training/validation error = 58.93127325324054/63.56353164138368
test error = 61.2620298402672
```

[28]:
```
bestMSE, bestModel
```

[28]:
```
(63.56353164138368, Ridge(alpha=12000, fit_intercept=False))
```

The best alpha still shows to be dependent on how the data is shuffled, resulting in values such as

0.001 or 10. The error has shown to be quite similar with different lambdas, which means in this context the lambda is not important. Here a multiple linear regression can be done without the use of Ridge.