In [1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from xgboost import XGBRegressor
from sklearn import metrics
```

In [2]:
```python
Boston_df = pd.read_csv("HousingData.csv")
```

In [3]:
```python
print(Boston_df.head())
```

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

        B  LSTAT  MEDV  
0  396.90   4.98  24.0  
1  396.90   9.14  21.6  
2  392.83   4.03  34.7  
3  394.63   2.94  33.4  
4  396.90    NaN  36.2  
```

In [4]:
```python
print(Boston_df.shape)
```

```
(506, 14)
```

Data preprocessing

In [5]:
```python
Boston_df.isnull().sum()
```

Out[5]:
```
CRIM       20
ZN         20
INDUS      20
CHAS       20
NOX         0
RM          0
AGE        20
DIS         0
RAD         0
TAX         0
PTRATIO     0
B           0
LSTAT      20
MEDV        0
dtype: int64
```

In [6]:
```python
fig, ax = plt.subplots(figsize=(8,8))
sns.distplot(Boston_df.CRIM)
```

```
plt.show()

fig, ax = plt.subplots(figsize=(8,8))
sns.distplot(Boston_df.ZN)
plt.show()

fig, ax = plt.subplots(figsize=(8,8))
sns.distplot(Boston_df.INDUS)
plt.show()

fig, ax = plt.subplots(figsize=(8,8))
sns.distplot(Boston_df.CHAS)
plt.show()

fig, ax = plt.subplots(figsize=(8,8))
sns.distplot(Boston_df.AGE)
plt.show()

fig, ax = plt.subplots(figsize=(8,8))
sns.distplot(Boston_df.LSTAT)
plt.show()
```

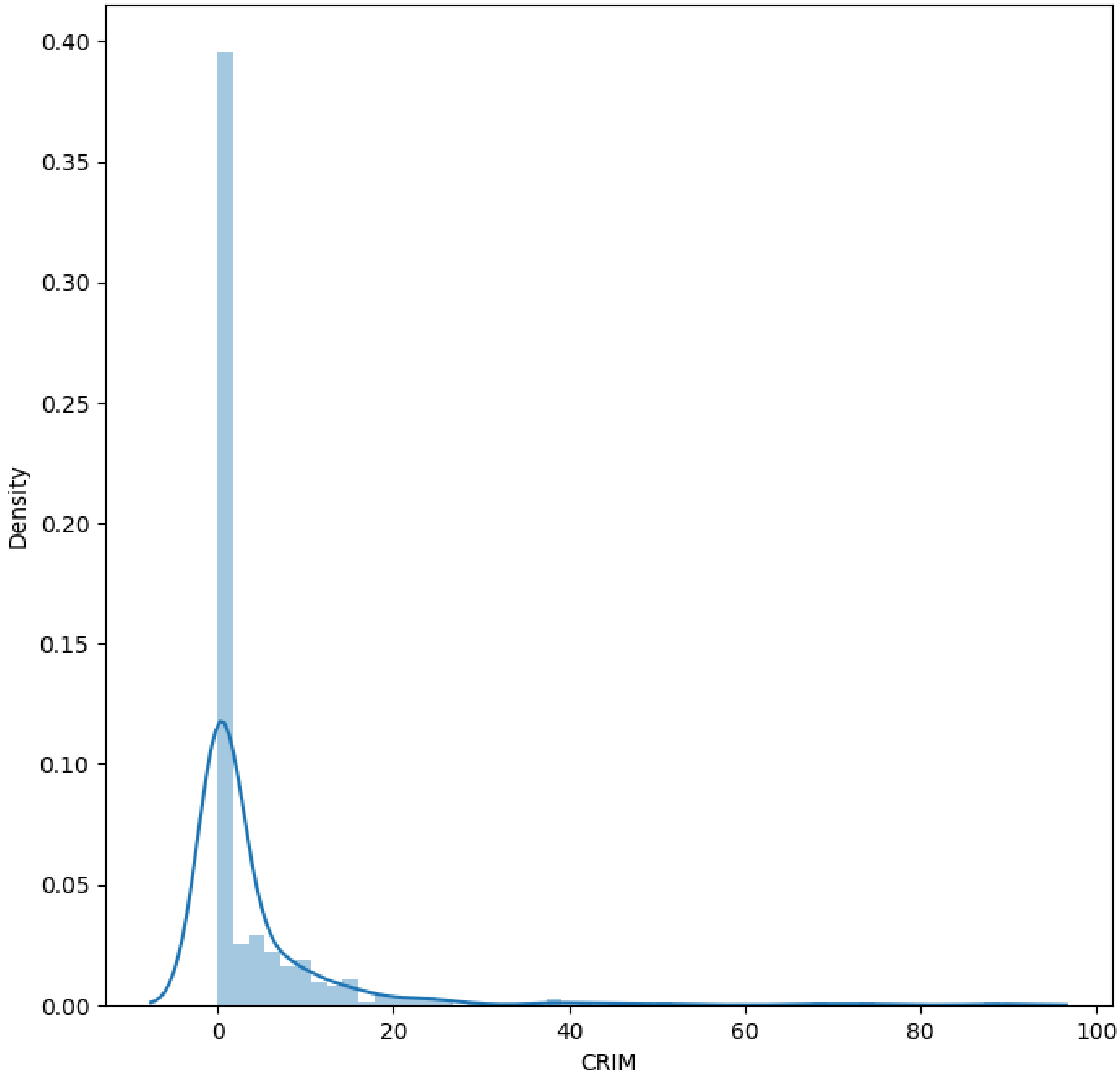
```
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\1919813569.py:2: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(Boston_df.CRIM)
```



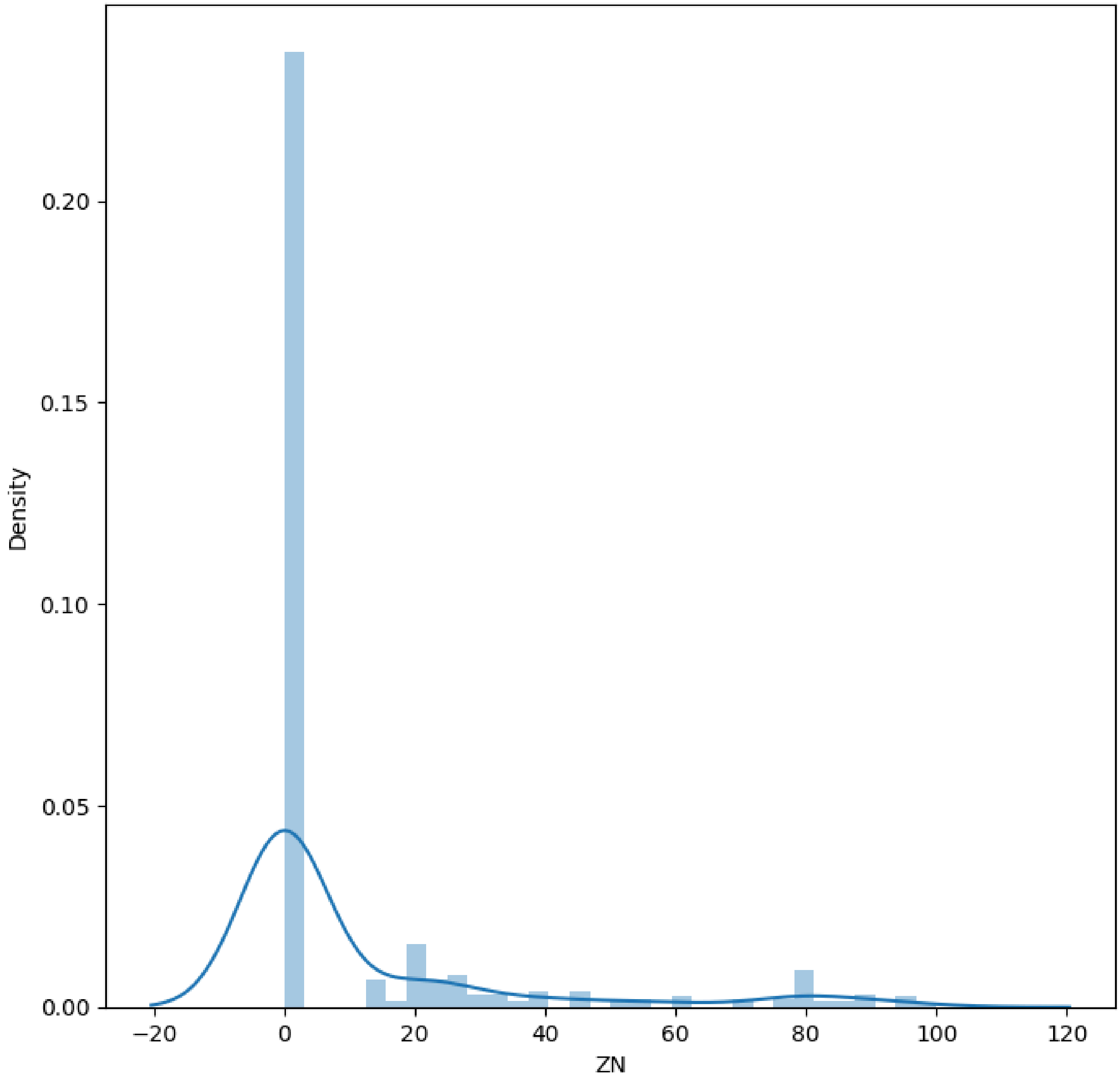

```
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\1919813569.py:6: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(Boston_df.ZN)
```



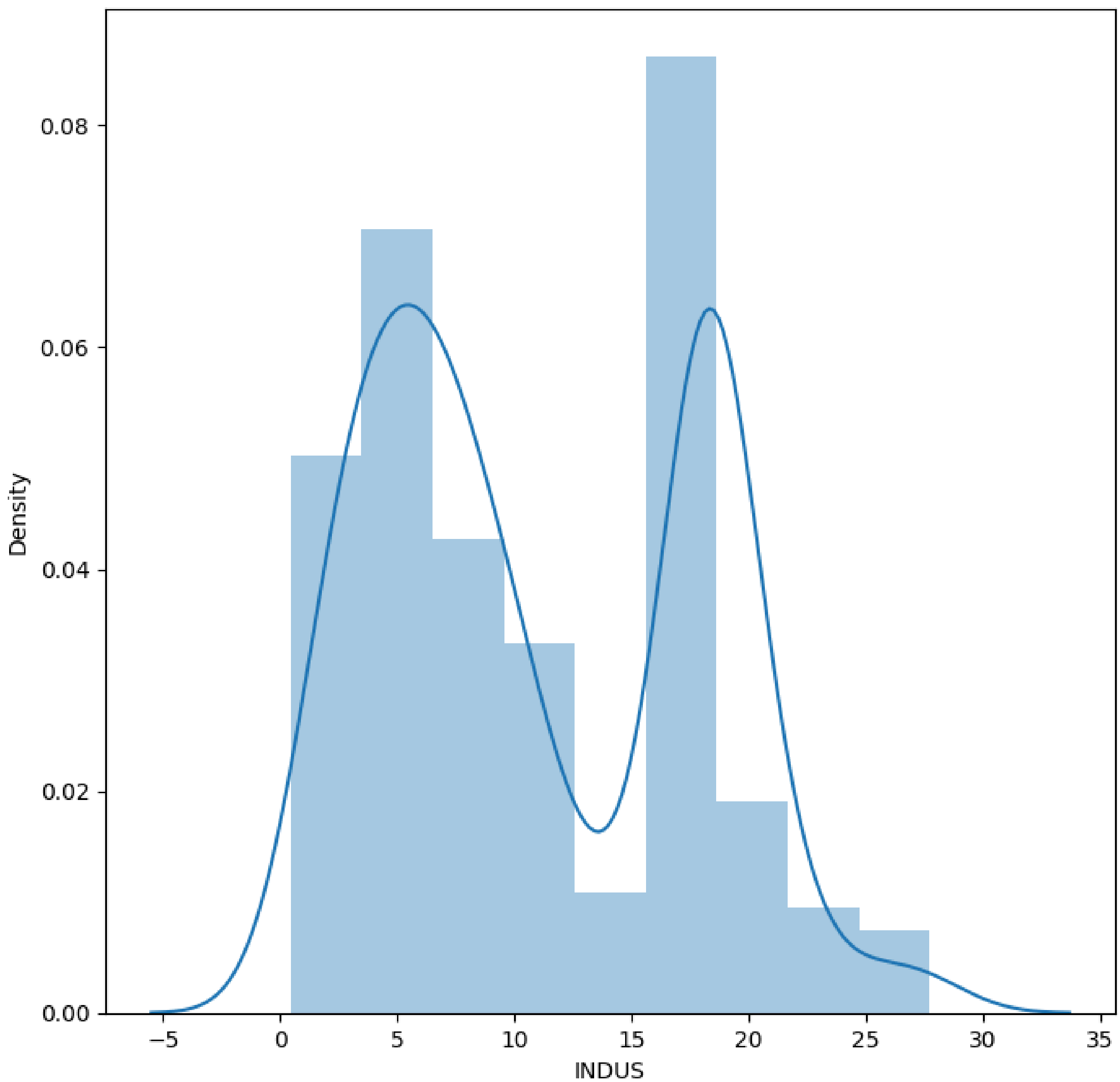

```
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\1919813569.py:10: UserWarnin
g:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(Boston_df.INDUS)
```



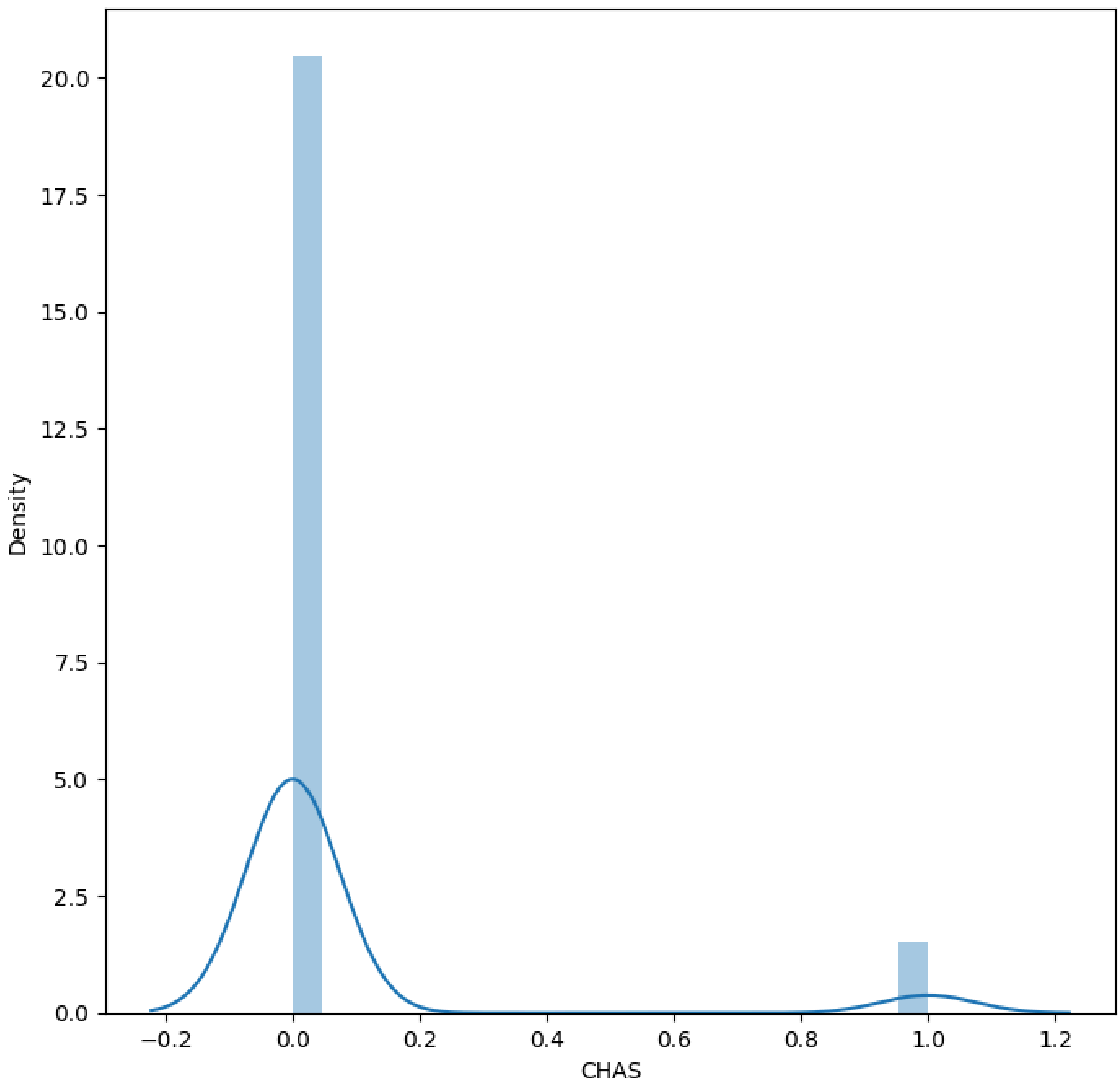

```
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\1919813569.py:14: UserWarnin
g:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(Boston_df.CHAS)
```



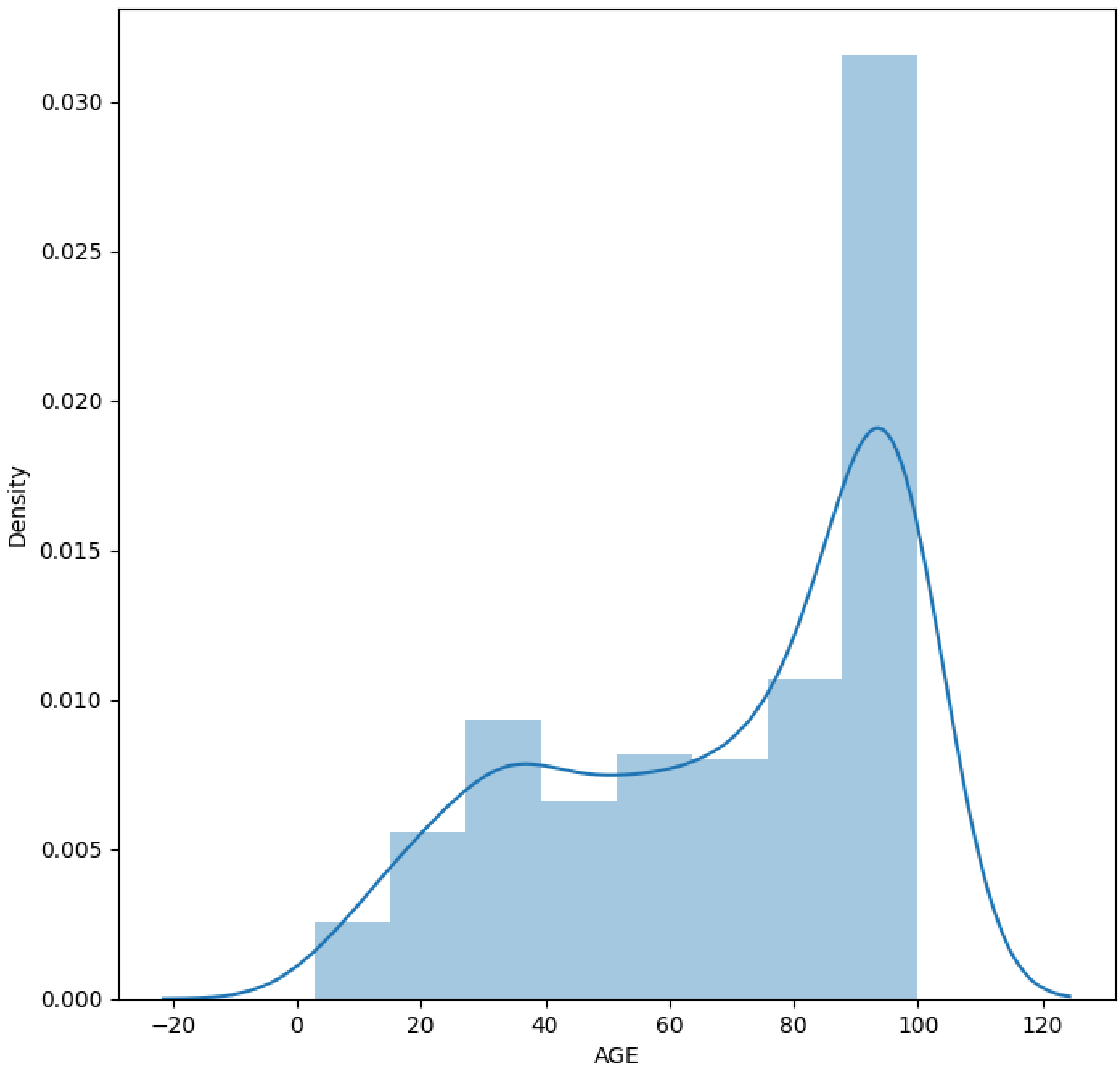

```
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\1919813569.py:18: UserWarnin
g:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(Boston_df.AGE)
```



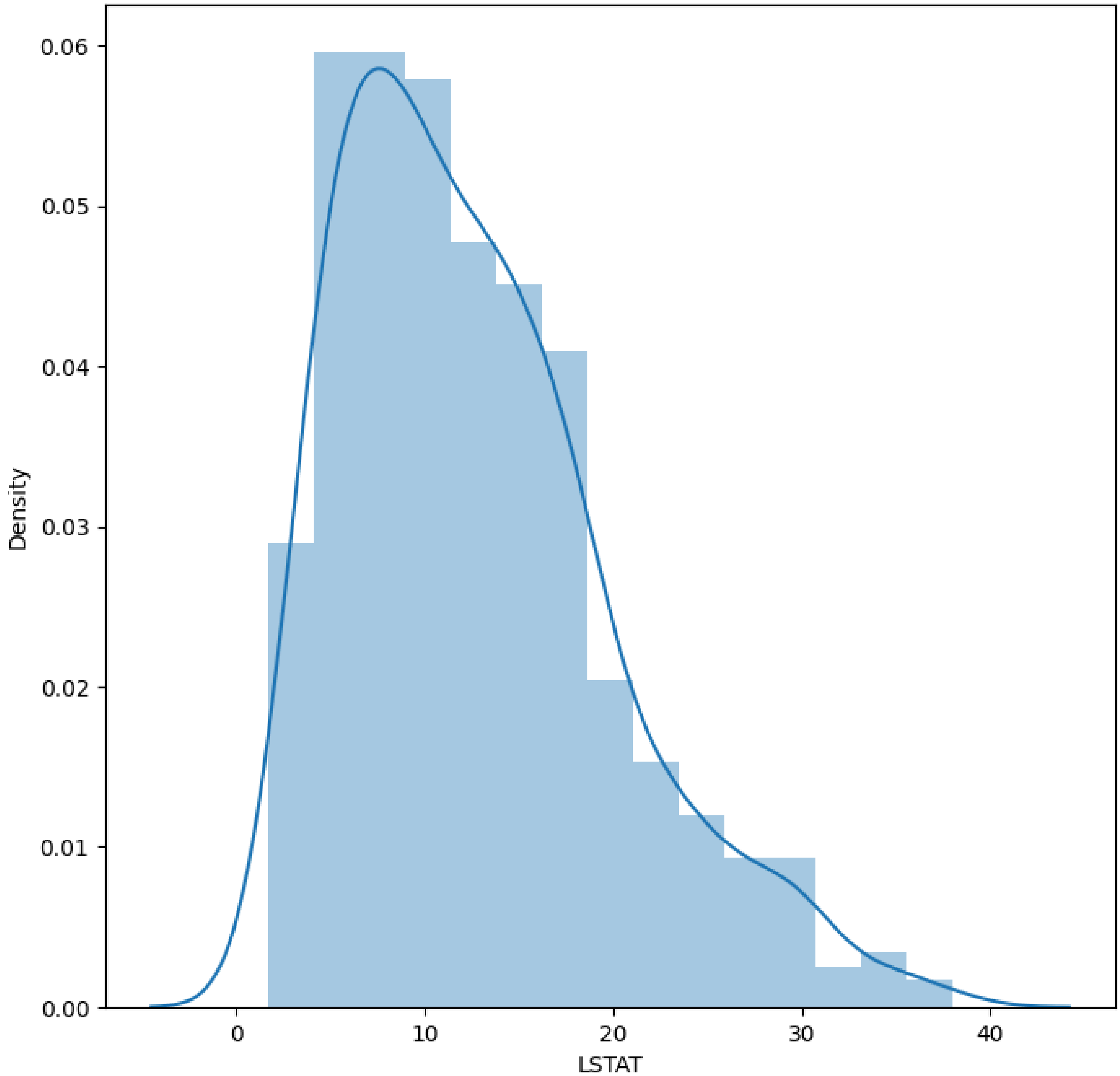

```
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\1919813569.py:22: UserWarnin
g:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(Boston_df.LSTAT)
```



In [7]:
```
# handling the missing values
# Impute missing values in the 'CRIM' column with the median
median_crim = Boston_df['CRIM'].median()
Boston_df['CRIM'].fillna(median_crim, inplace=True)

# Impute missing values in the 'ZN' column with the median
median_zn = Boston_df['ZN'].median()
Boston_df['ZN'].fillna(median_zn, inplace=True)

# Impute missing values in the 'INDUS' column with the median
median_indus = Boston_df['INDUS'].median()
Boston_df['INDUS'].fillna(median_indus, inplace=True)

# using the mode for imputation is a suitable approach. This will replace miss
# Impute missing values in the 'CHAS' column with the mode
mode_chas = Boston_df['CHAS'].mode()[0] # .mode() can return multiple values i
Boston_df['CHAS'].fillna(mode_chas, inplace=True)
```

```python

# Impute missing values in the 'AGE' column with the median
median_age = Boston_df['AGE'].median()
Boston_df['AGE'].fillna(median_age, inplace=True)

# Impute missing values in the 'LSTAT' column with the median
median_lstat = Boston_df['LSTAT'].median()
Boston_df['LSTAT'].fillna(median_lstat, inplace=True)
```

```
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\2845753325.py:4: FutureWarnin
g: A value is trying to be set on a copy of a DataFrame or Series through chain
ed assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work bec
ause the intermediate object on which we are setting values always behaves as a
copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.me
thod({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, t
o perform the operation inplace on the original object.


  Boston_df['CRIM'].fillna(median_crim, inplace=True)
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\2845753325.py:8: FutureWarnin
g: A value is trying to be set on a copy of a DataFrame or Series through chain
ed assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work bec
ause the intermediate object on which we are setting values always behaves as a
copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.me
thod({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, t
o perform the operation inplace on the original object.


  Boston_df['ZN'].fillna(median_zn, inplace=True)
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\2845753325.py:12: FutureWarni
ng: A value is trying to be set on a copy of a DataFrame or Series through chai
ned assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work bec
ause the intermediate object on which we are setting values always behaves as a
copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.me
thod({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, t
o perform the operation inplace on the original object.


  Boston_df['INDUS'].fillna(median_indus, inplace=True)
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\2845753325.py:17: FutureWarni
ng: A value is trying to be set on a copy of a DataFrame or Series through chai
ned assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work bec
ause the intermediate object on which we are setting values always behaves as a
copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.me
thod({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, t
o perform the operation inplace on the original object.


  Boston_df['CHAS'].fillna(mode_chas, inplace=True)
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\2845753325.py:21: FutureWarni
ng: A value is trying to be set on a copy of a DataFrame or Series through chai
```

```
ned assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work bec
ause the intermediate object on which we are setting values always behaves as a
copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.me
thod({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, t
o perform the operation inplace on the original object.


  Boston_df['AGE'].fillna(median_age, inplace=True)
C:\Users\erraj\AppData\Local\Temp\ipykernel_38320\2845753325.py:25: FutureWarni
ng: A value is trying to be set on a copy of a DataFrame or Series through chai
ned assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work bec
ause the intermediate object on which we are setting values always behaves as a
copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.me
thod({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, t
o perform the operation inplace on the original object.


  Boston_df['LSTAT'].fillna(median_lstat, inplace=True)
```

In [8]:
```
Boston_df.isnull().sum()
```

Out[8]:
```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
MEDV       0
dtype: int64
```

In [9]:
```
# all the stastistical method at once
Boston_df.describe()
```

Out[9]:

	CRIM	ZN	INDUS	CHAS	NOX	RM
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.479140	10.768775	11.028893	0.067194	0.554695	6.284634
std	8.570832	23.025124	6.704679	0.250605	0.115878	0.702617
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000
25%	0.083235	0.000000	5.190000	0.000000	0.449000	5.885500
50%	0.253715	0.000000	9.690000	0.000000	0.538000	6.208500
75%	2.808720	0.000000	18.100000	0.000000	0.624000	6.623500
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000

Cheacking of possitive and Negative correlation

In [10]:

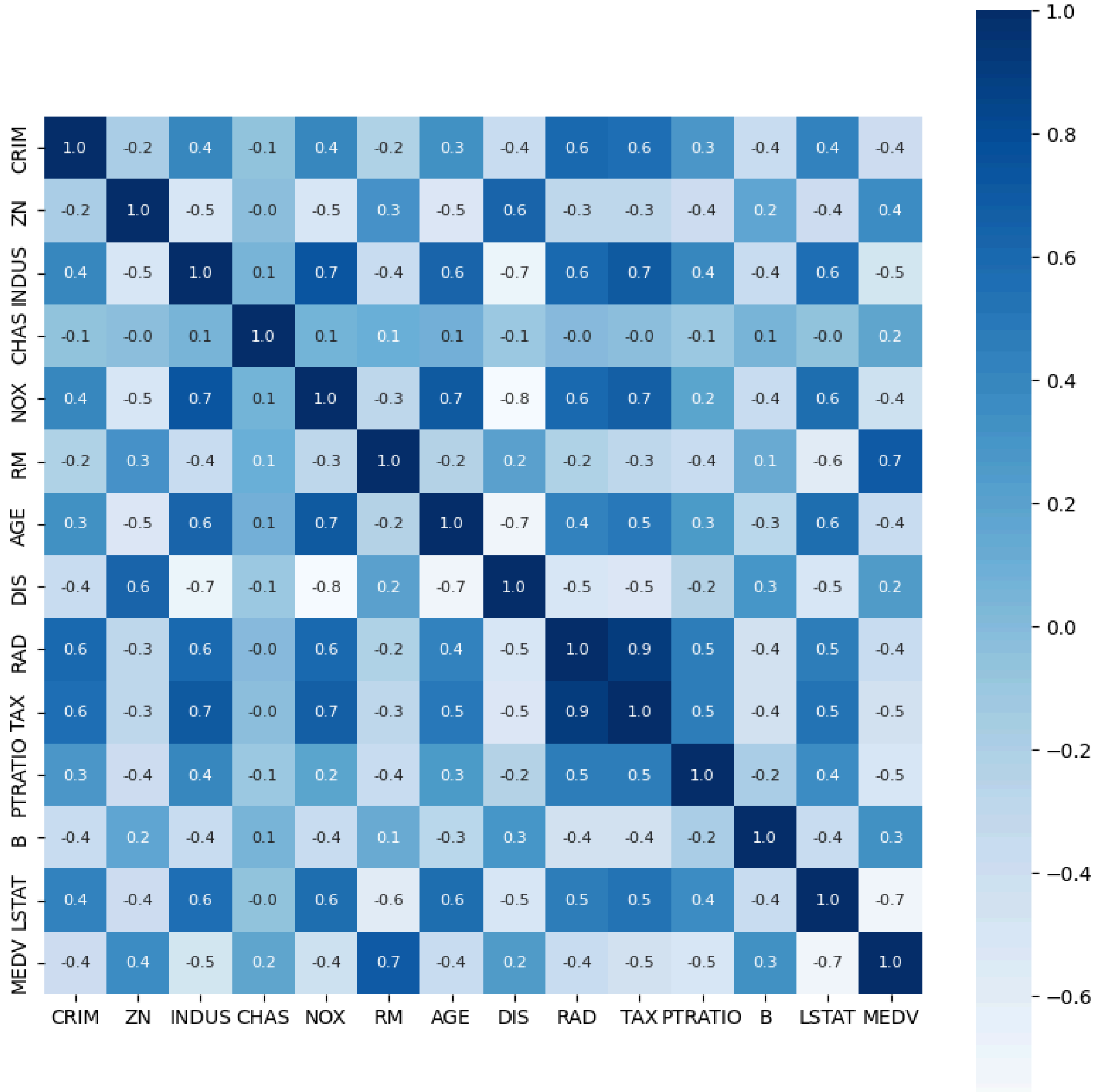
```
correlation = Boston_df.corr()
```

In [11]:

```
# constructing heatmap to understand the correlation
plt.figure(figsize=(10,10))
sns.heatmap(correlation, cbar=True, square=True, fmt='.1f', annot=True, annot_
```

Out[11]:

```
<Axes: >
```



splitting the data and Target

In [12]:
```
X = Boston_df.drop(['MEDV'], axis=1)    # droping column so axis = 1
Y = Boston_df['MEDV']
```

In [13]:
```
print(X)
print(Y)
```

```
        CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD  TAX  \
0    0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900    1  296
1    0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671    2  242
2    0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671    2  242
3    0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622    3  222
4    0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622    3  222
..       ...   ...    ...   ...    ...    ...   ...     ...  ...  ...
501  0.06263   0.0  11.93   0.0  0.573  6.593  69.1  2.4786    1  273
502  0.04527   0.0  11.93   0.0  0.573  6.120  76.7  2.2875    1  273
503  0.06076   0.0  11.93   0.0  0.573  6.976  91.0  2.1675    1  273
504  0.10959   0.0  11.93   0.0  0.573  6.794  89.3  2.3889    1  273
505  0.04741   0.0  11.93   0.0  0.573  6.030  76.8  2.5050    1  273

     PTRATIO       B  LSTAT
0       15.3  396.90   4.98
1       17.8  396.90   9.14
2       17.8  392.83   4.03
3       18.7  394.63   2.94
4       18.7  396.90  11.43
..       ...     ...    ...
501     21.0  391.99  11.43
502     21.0  396.90   9.08
503     21.0  396.90   5.64
504     21.0  393.45   6.48
505     21.0  396.90   7.88

[506 rows x 13 columns]
0      24.0
1      21.6
2      34.7
3      33.4
4      36.2
       ...
501    22.4
502    20.6
503    23.9
504    22.0
505    11.9
Name: MEDV, Length: 506, dtype: float64
```

SPLITTING DATA INTO Training and Testing data

In [14]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, rando
```

In [15]:
```
print(X.shape, X_train.shape, X_test.shape)
```

```
(506, 13) (404, 13) (102, 13)
```

MODEL Training XGBoost Regressor

In [16]:
```
# loading the model
model = XGBRegressor()
```

In [17]:
```
 # fitting is nothing but training of model
model.fit(X_train, Y_train)
```

Out[17]:

XGBRegressor

```
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None, early_stopping_round
s=None,
             enable_categorical=False, eval_metric=None, feature_type
s=None,
             feature_weights=None, gamma=None, grow_policy=None,
             importance_type=None, interaction_constraints=None,
             learning_rate=None, max_bin=None, max_cat_threshold=Non
e,
```

Evaluation

Prediction on training data

In [18]:
```
# accuracy for prediction on training data
training_data_prediction = model.predict(X_train)
```

In [19]:
```
print(training_data_prediction)
```

```
[23.124718  21.00754   20.102568  34.69276   13.904569  13.49714
 21.997927  15.19248   10.902376  22.7026    13.800668   5.5908513
 29.806072  50.005272  34.89682   20.596664  23.386295  19.18905
 32.691494  19.63137   26.9884     8.40349   46.001217  21.7111
 27.08755   19.365828  19.286129  24.817802  22.611925  31.707855
 18.541298   8.704057  17.404493  23.701723  13.304713  10.520918
 12.70769   24.98351   19.686928  14.899053  24.209797  24.994987
 14.897052  17.01417   15.603933  12.6952915 24.52194   15.007025
 49.999977  17.510012  21.203285  32.003624  15.595356  22.898546
 19.32731   18.687641  23.30319   37.200005  30.095251  33.104855
 20.992231  50.002266  13.401404   5.007679  16.5074     8.395711
 28.68154   19.493786  20.596518  45.400917  39.804905  33.41812
 19.840513  33.39644   25.271023  49.998375  12.517453  17.421158
 18.604883  22.601322  50.00689   23.780687  23.312428  23.099342
 41.69992   16.099009  31.596653  36.08056    6.999861  20.382881
 20.000896  11.997474  24.995806  49.99009   37.89816   23.101585
 41.27909   17.601559  16.308878  30.048021  22.85182   19.788818
 17.106504  18.901857  18.939293  22.594484  23.15195   33.202766
 14.999921  11.704477  18.80795   20.797966  17.995903  19.641762
 50.00197   17.195467  16.402473  17.50968   14.601552  33.097614
 14.495126  43.805664  34.900345  20.398077  14.625136   8.094603
 11.778059  11.81811   18.70215    6.2982407 23.978172  13.066449
 19.607876  49.999638  22.320127  18.919546  31.195917  20.702322
 32.199364  36.165913  14.21831   15.7038765 49.983604  20.426397
 16.174377  13.4097595 50.013714  31.606308  12.291674  19.217142
 29.799698  31.501135  22.799213  10.192999  24.081253  23.703596
 21.992378  13.785494  28.397635  33.19623   13.12277   19.048325
 26.577816  36.955544  30.789625  22.790836  10.191355  22.198246
 24.489767  36.18868   23.104813  20.114384  19.497326  10.80379
 22.674639  19.507032  20.12285    9.60392   42.800453  48.79566
 13.093994  20.288769  24.748339  14.103552  21.705412  22.232885
 33.000233  21.116417  25.001814  19.119667  32.39981   13.608207
 15.087661  23.093206  27.494678  19.37617   26.487434  27.501059
 28.713585  21.22839   18.686394  26.723293  14.007895  21.704782
 18.39718   43.114174  29.09505   20.300016  23.70995   18.291485
 17.193474  18.319866  24.398975  26.397472  19.099882  13.307144
 22.17664   22.189608   8.53707   18.89726   21.794674  19.346647
 18.196587   7.5117607 22.395323  20.004005  14.399867  22.498732
 28.510593  21.637846  13.804338  20.498287  21.900301  23.091843
 50.000694  16.212135  30.307535  49.994564  17.797531  19.062796
 10.398314  20.387184  16.501936  17.185501  16.736221  19.513783
 30.499746  29.00148   19.556606  23.182762  24.39962    9.502657
 23.892439  49.99509   21.199549  22.610487  19.990492  13.402443
 19.960081  17.11145   12.723276  22.999899  15.244268  20.579155
 26.206219  18.06881   24.096706  14.097596  21.700525  20.07267
 25.013454  27.908047  22.916878  18.493921  22.197021  24.003748
 14.810876  19.888756  24.399963  17.793882  24.5913    32.00859
 17.793465  23.326887  16.09375   13.007798  10.997188  24.316023
 15.592238  35.205746  19.599157  42.30329    8.796408  24.391514
 14.101722  15.392906  17.298933  22.118727  23.100609  44.811054
 17.803215  31.501413  22.80798   16.863302  23.907934  12.075618
 38.7021    21.406189  16.001757  23.913858  11.895336  24.956305
  7.200167  24.696964  18.20617   22.466045  23.026604  24.300848
 17.110569  17.80117   13.496556  27.07502   13.305881  21.898989
```

```
19.997826  15.361549  16.58387   22.302023  24.720388  21.399815
22.89386   29.597786  21.871275  19.895018  29.599123  23.39317
13.804395  24.44912   11.906414   7.2283416 20.504478   9.699921
48.29024   25.197203  11.686412  17.400322  14.49293   28.59803
19.38137   22.44903    7.014273  20.590538  22.981966  19.69853
23.695066  25.00565   28.003778  13.382997  14.525479  20.311686
19.308327  24.096216  14.900837  26.39965   33.302242  23.6273
24.596827  18.503511  20.899784  10.401823  23.296179  13.112332
24.671389  22.596157  20.502825  16.81155   10.204616  33.805656
18.616283  49.996334  23.796791  23.900902  21.182632  18.816933
 8.505743  21.498335  23.196657  21.019245  16.606592  28.09294
21.20949   28.39348   14.284869  50.002743  30.989561  24.997051
21.427883  19.001093  29.003275  15.20375   22.797543  21.770397
19.91061   23.77714  ]
```

In [20]:
```
# for knowing the error use R squared error
# i just calculate the difference btwn the value predicted by model and origin
score_1 = metrics.r2_score(Y_train, training_data_prediction)

# Mean Absolute Error
score_2 = metrics.mean_absolute_error(Y_train, training_data_prediction)

print("R squared error : ", score_1)  # this value shuld be near to 1 if it is
# if 0 then model preforming perfectly
print("Mean Absolute Error : ", score_2)
```

```
R squared error :  0.9999970846673867
Mean Absolute Error :  0.010642476601175762
```

In [21]:
```
# accuracy for prediction on testing data
test_data_prediction = model.predict(X_test)
```

visualizing the actual and predicted prices

In [25]:
```
plt.scatter(Y_train, training_data_prediction)
plt.xlabel("Actual Prices for teste data")
plt.ylabel("Predicted Prices test data")
plt.title("Actual Prices vs Predicted Prices")
plt.show()
```

In [22]:

```
plt.scatter(Y_test, test_data_prediction)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual Prices vs Predicted Prices")
plt.show()
```

Finding the error value for test data

In [ ]:
```
# for knowing the error use R squared error
# i just calculate the difference btwn the value predicted by model and origin
score_1 = metrics.r2_score(Y_test, test_data_prediction)

# Mean Absolute Error
score_2 = metrics.mean_absolute_error(Y_test, test_data_prediction)

print("R squared error : ", score_1)  # this value shuld be near to 1 if it is
# if 0 then model preforming perfectly
print("Mean Absolute Error : ", score_2)
```

```
R squared error :  0.8950119616414808
Mean Absolute Error :  2.2481449632083668
```