Real Pricing

March 13, 2020

1 Pricing - Example with Demodata by sklearn

The Task

For that example a dataset by sklearn is used: the Boston House Dataset. The goal is to be able to make a price prediction of a house and to determine the factors on which the price depends.

First things first: The Result:

Dependencies in this case: 'INDUS', 'NOX', 'RM', 'TAX', 'PTRATIO', 'LSTAT' (The description of that is below)

Here are 5 Samples:

PREDICTION: 25.17 // REAL: 23.8 // DIFFERENCE: 1.37

PREDICTION: 22.31 // REAL: 19.3 // DIFFERENCE: 3.01

PREDICTION: 12.51 // REAL: 7.2 // DIFFERENCE: 5.31

PREDICTION: 32.52 // REAL: 33.1 // DIFFERENCE: -0.58

PREDICTION: 23.62 // REAL: 24.0 // DIFFERENCE: -0.38

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

2 Boston Housing Dataset

In this example we will evaluate the price of a house in dependencies with some parameters. Therefor the boston dataset of sklearn is using. That are real, but old data.

Before the price gets evaluate, the data will be read as a dataframe.

[16]:
```
from sklearn.datasets import load_boston
boston = load_boston()
```

2.1 Get Informations about the data

The dataset is loaded as an object with some methods. At the following we will take a look at the given informations and will create a pandas dataframe.

DESCR - Gives some detail information about the dataset

```
[22]: print(boston.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value
(attribute 14) is usually the target.

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

    :Missing Attribute Values: None

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

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


This dataset was taken from the StatLib library which is maintained at Carnegie
```

```
Mellon University.

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

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

.. topic:: References

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

feature_names gives Informations about the columnnames

[19]:
```
boston.feature_names
```

[19]:
```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')
```

[ ]:
```
boston.
```

In this section the data convert into a pandas dataframe In real project this would normaly be the entrypoint.

[26]:
```
df = pd.DataFrame(boston.data, columns=boston.feature_names)
```

2.2 Pandas Dataframe

Take a quick look at the dataframe with the head method.

[21]:
```
df.head()
```

[21]:
```
      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  1.0  296.0
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0
4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0
```

```
   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   5.33
```

The House prices are missing at this dataframe. They are the target of the boston dataframe. We need to add them to the dataframe and take a quick look to the data after it.

[29]:
```python
df['MEDV'] = boston.target
```

[30]:
```python
df.head()
```

[30]:
```
      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  1.0  296.0
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0
4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0

   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
```

2.2.1 Quality Check

Now we need to know something about the quality of that data, most important to check is if there are null-values or different types of values in one column.

With the info method we will have a quick overview about the data

[31]:
```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    float64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
```

```
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    float64
 9   TAX      506 non-null    float64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  MEDV     506 non-null    float64
dtypes: float64(14)
memory usage: 55.5 KB
```

This quick check shows, that they are no null-values and no type conflicts. All column types are floats.

[32]:
```
df.describe()
```

[32]:
```
             CRIM          ZN       INDUS        CHAS         NOX          RM  \
count  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
std      8.601545   23.322453    6.860353    0.253994    0.115878    0.702617
min      0.006320    0.000000    0.460000    0.000000    0.385000    3.561000
25%      0.082045    0.000000    5.190000    0.000000    0.449000    5.885500
50%      0.256510    0.000000    9.690000    0.000000    0.538000    6.208500
75%      3.677083   12.500000   18.100000    0.000000    0.624000    6.623500
max     88.976200  100.000000   27.740000    1.000000    0.871000    8.780000

              AGE         DIS         RAD         TAX     PTRATIO           B  \
count  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean    68.574901    3.795043    9.549407  408.237154   18.455534  356.674032
std     28.148861    2.105710    8.707259  168.537116    2.164946   91.294864
min      2.900000    1.129600    1.000000  187.000000   12.600000    0.320000
25%     45.025000    2.100175    4.000000  279.000000   17.400000  375.377500
50%     77.500000    3.207450    5.000000  330.000000   19.050000  391.440000
75%     94.075000    5.188425   24.000000  666.000000   20.200000  396.225000
max    100.000000   12.126500   24.000000  711.000000   22.000000  396.900000

            LSTAT        MEDV
count  506.000000  506.000000
mean    12.653063   22.532806
std      7.141062    9.197104
min      1.730000    5.000000
25%      6.950000   17.025000
50%     11.360000   21.200000
75%     16.955000   25.000000
max     37.970000   50.000000
```

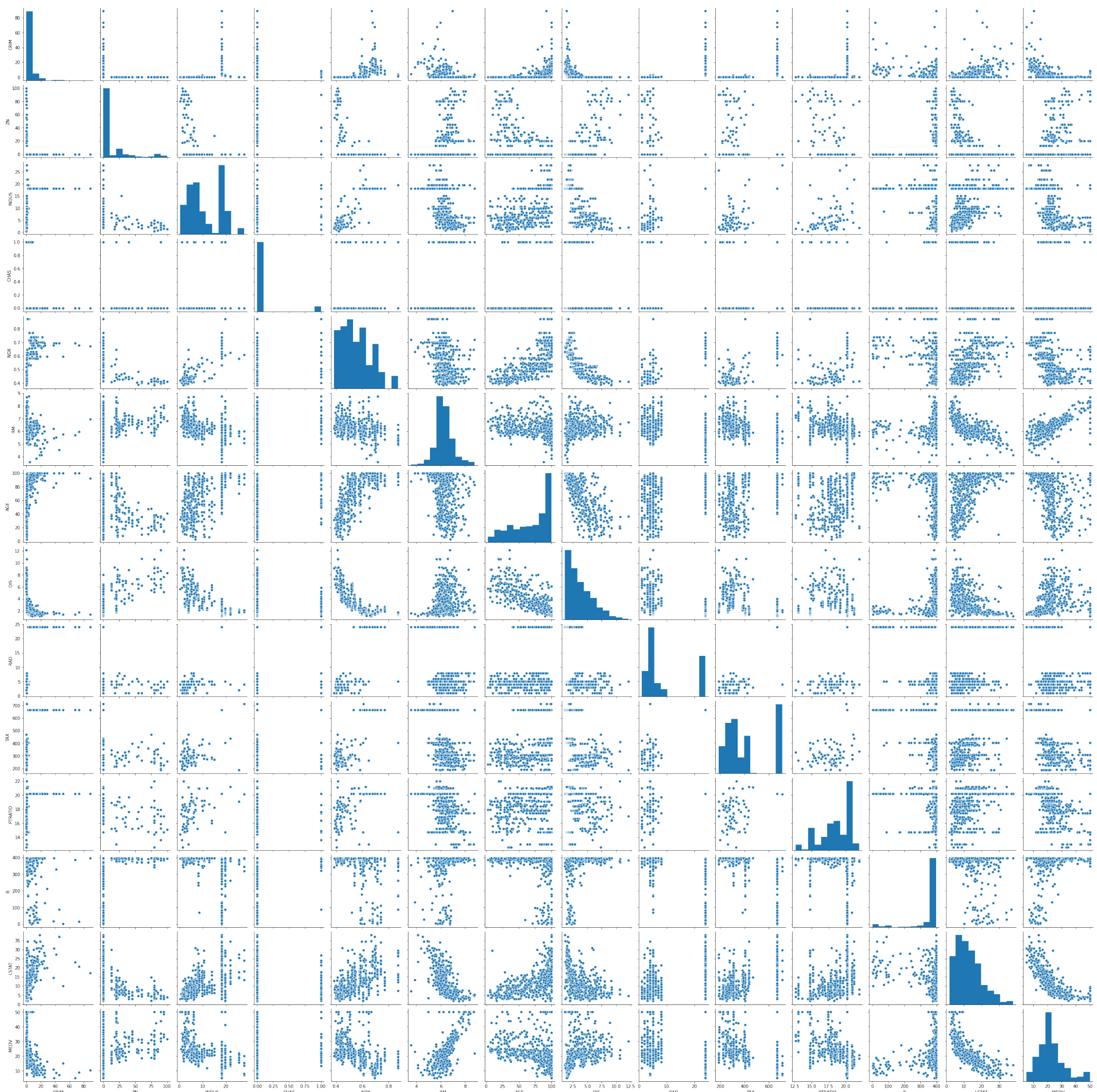
2.2.2 Visualization

We will make a quick visualization for the data for seeing any dependencies and to look to the price-distribution

Quick overview First we make a quick overview for seeing the basic distribution and seeing the distribiutions and take a look at the distribution of each column.

[169]:
```
sns.pairplot(df)
```

[169]: <seaborn.axisgrid.PairGrid at 0x12ffdf4d0>



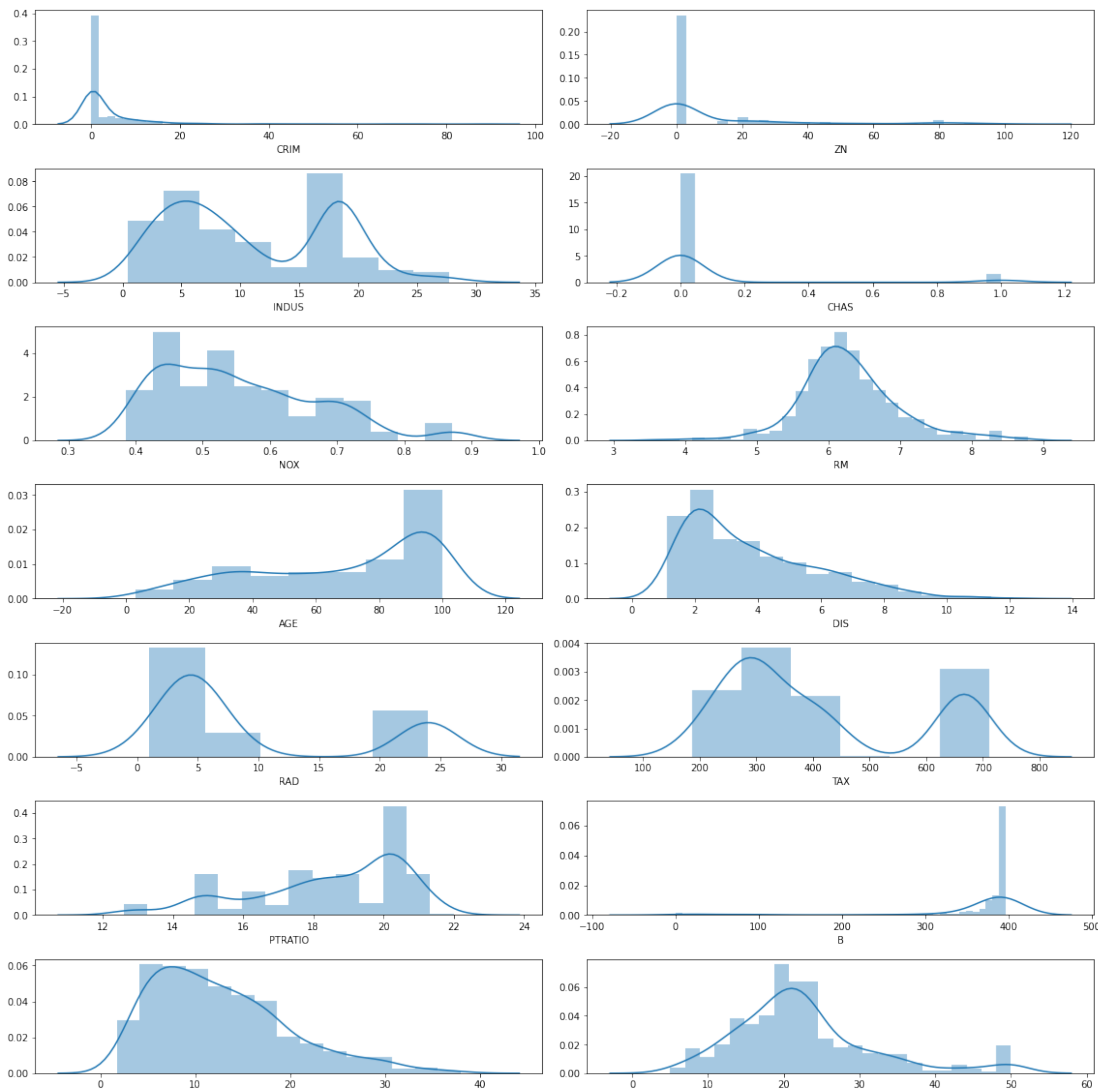
[65]:
```
rows = 7
cols = 2
```

```python
fig, ax = plt.subplots(nrows= rows, ncols= cols, figsize = (16,16))

col = df.columns
index = 0

for i in range(rows):
    for j in range(cols):
        sns.distplot(df[col[index]], ax = ax[i][j])
        index = index + 1

plt.tight_layout()
```

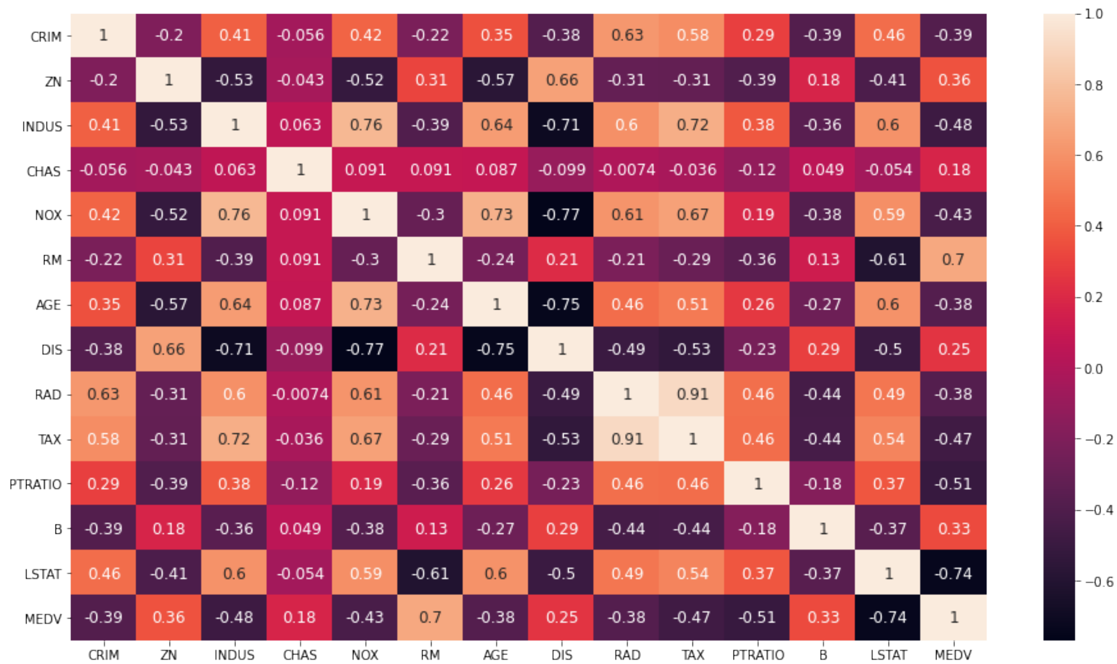


Correlations Next we check the correlations of and summarize the relationships between the variables.

Remember: a correlation factor near 1 or -1 is wanted. 0 means that there is no linear relation between that columns and they will sabotage the linear regression model

[98]:
```python
fig, ax = plt.subplots(figsize = (16, 9))
sns.heatmap(df.corr(), annot = True, annot_kws={'size': 12})
```

[98]: <matplotlib.axes._subplots.AxesSubplot at 0x12e595290>



Specially the correlations ad the MEDV row are interesting for us. For a linear regression method we need nearly high correlations. in that case we need to define a threshold filter

[69]:
```python
def getCorrelatedFeature(corrdata, threshold):
    feature = []
    value = []

    for i, index in enumerate(corrdata.index):
        if abs(corrdata[index])> threshold:
            feature.append(index)
            value.append(corrdata[index])

    df = pd.DataFrame(data = value, index = feature, columns=['Corr Value'])
    return df
```

[151]:
```
threshold = 0.4
corr_value = getCorrelatedFeature(df.corr()['MEDV'], threshold)
```

[114]:
```
corr_value.index.values
```

[114]:
```
array(['INDUS', 'NOX', 'RM', 'TAX', 'PTRATIO', 'LSTAT', 'MEDV'],
      dtype=object)
```

[115]:
```
correlated_data = df[corr_value.index]
correlated_data.head()
```

[115]:

	INDUS	NOX	RM	TAX	PTRATIO	LSTAT	MEDV
0	2.31	0.538	6.575	296.0	15.3	4.98	24.0
1	7.07	0.469	6.421	242.0	17.8	9.14	21.6
2	7.07	0.469	7.185	242.0	17.8	4.03	34.7
3	2.18	0.458	6.998	222.0	18.7	2.94	33.4
4	2.18	0.458	7.147	222.0	18.7	5.33	36.2

2.3 Linear Regression

we will split the given data into a training and testing dataset.

[116]:
```
X = correlated_data.drop(labels=['MEDV'], axis = 1)
y = correlated_data['MEDV']
```

[117]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,␣
↪random_state=1)
```

[118]:
```
from sklearn.linear_model import LinearRegression
lm = LinearRegression()
```

[119]:
```
lm.fit(X_train,y_train)
```

[119]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

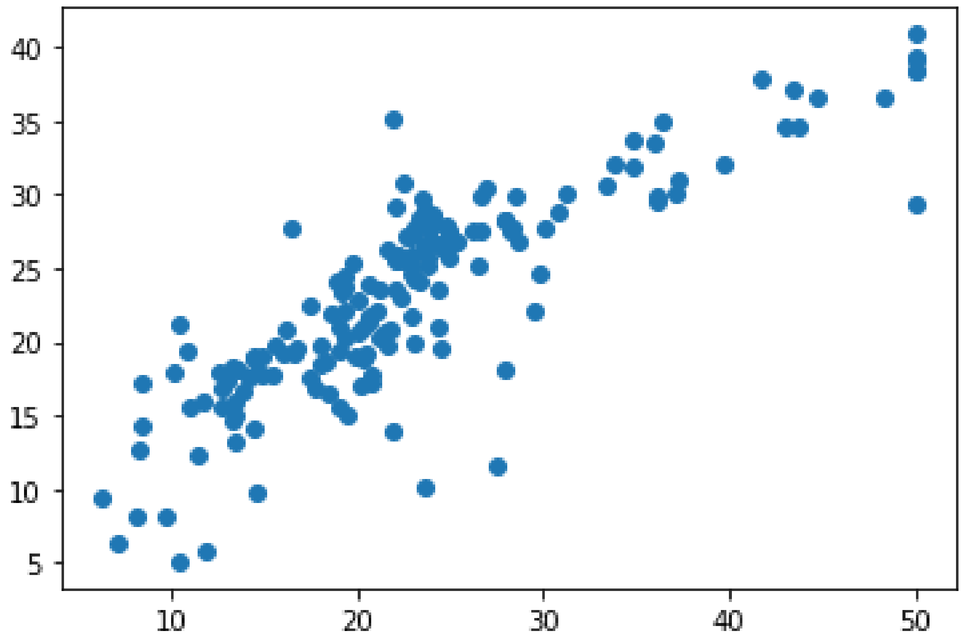
2.3.1 Result in a mathematical / visual way

We want to have a perfect linear relation of the points (or nearly linear). The larger the distribution of points, the greater the inaccuracy of the model.

[158]:
```
predictions = lm.predict(X_test)
```

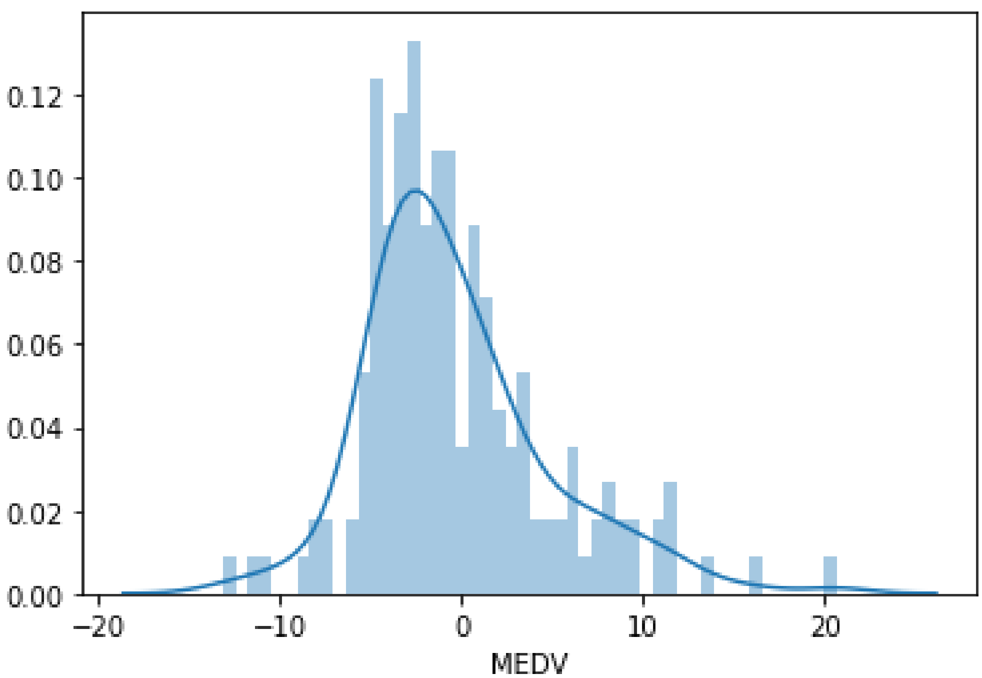
[159]:
```
plt.scatter(y_test,predictions)
```

[159]:
```
<matplotlib.collections.PathCollection at 0x12fca3f90>
```



[161]:
```
sns.distplot((y_test-predictions),bins=50)
```

[161]: <matplotlib.axes._subplots.AxesSubplot at 0x12fe795d0>



y-axis of a linear function

[122]:
```
lm.intercept_
```

[122]:
```
23.49923705354543
```

Coefficients of a linear regression function

[132]:
```
lm.coef_
```

[132]:
```
array([ 1.08016127e-01, -7.91996313e+00,  4.30011673e+00, -2.02636592e-04,
       -9.66142294e-01, -5.36384366e-01])
```

Define the linear regression funcion

[124]:
```
def lin_func(values, coefficients=lm.coef_, y_axis=lm.intercept_):
    return np.dot(values, coefficients) + y_axis
```

2.3.2 Samples

Lets see what we created. For that define random test data and make some samples

[168]:
```
from random import randint
for i in range(5):
    index = randint(0,len(df)-1)
    sample = df.iloc[index][corr_value.index.values].drop('MEDV')
    print(
            'PREDICTION: ', round(lin_func(sample),2),
            ' // REAL: ',df.iloc[index]['MEDV'],
            ' // DIFFERENCE: ', round(round(lin_func(sample),2) - df.
 ↪iloc[index]['MEDV'],2)
    )
```

```
PREDICTION:  25.17  // REAL:  23.8  // DIFFERENCE:  1.37
PREDICTION:  22.31  // REAL:  19.3  // DIFFERENCE:  3.01
PREDICTION:  12.51  // REAL:  7.2  // DIFFERENCE:  5.31
PREDICTION:  32.52  // REAL:  33.1  // DIFFERENCE:  -0.58
PREDICTION:  23.62  // REAL:  24.0  // DIFFERENCE:  -0.38
```

[ ]:
```

```