tu18_LinearRegression_II

April 6, 2023

1 Linear Regression II

Learning objectives:

Learn basic measures of quality of fit and variance explained: - Mean Squared Errors - R2 - Total Sum of Squares - Residual Sum of Squares - Explained Sum of Squares - Linear Regression with multiple parameters

---

1.1 Linear regression *recap.*

In the last tutorial we went over linear regression using *numpy*'s *polyval* and *polyfit*.

Linear regression is used to model the relationship between a dependent variable and one or more independent variables. It assumes that there is a linear relationship between the independent variable(s) and the dependent variable.

The goal of linear regression is to find the best-fit model that minimizes the difference between the predicted values and the actual values. Linear regression is important because it can help us to understand and predict the relationship between two or more variables. The term **linear** in Linear regression does not refer only to a line, but it applies to any polynomial! Linear here is referred to the parameters of the model and not the model *per se*!

In the previous tutorial we have learned how to fit models via linear regression by using `numpy`. Here we will learn a little bit more about fitting linear regression models.

[32]:
```
import numpy as np
import matplotlib.pyplot as plt

# Generate random data
np.random.seed(123)
x = np.random.rand(50)
y = 2*x + 0.5*np.random.randn(50)

# Fit a linear regression model
coeffs = np.polyfit(x, y, 1)
y_pred = np.polyval(coeffs, x)

# Print the coefficients
print("Intercept:", coeffs[1])
```

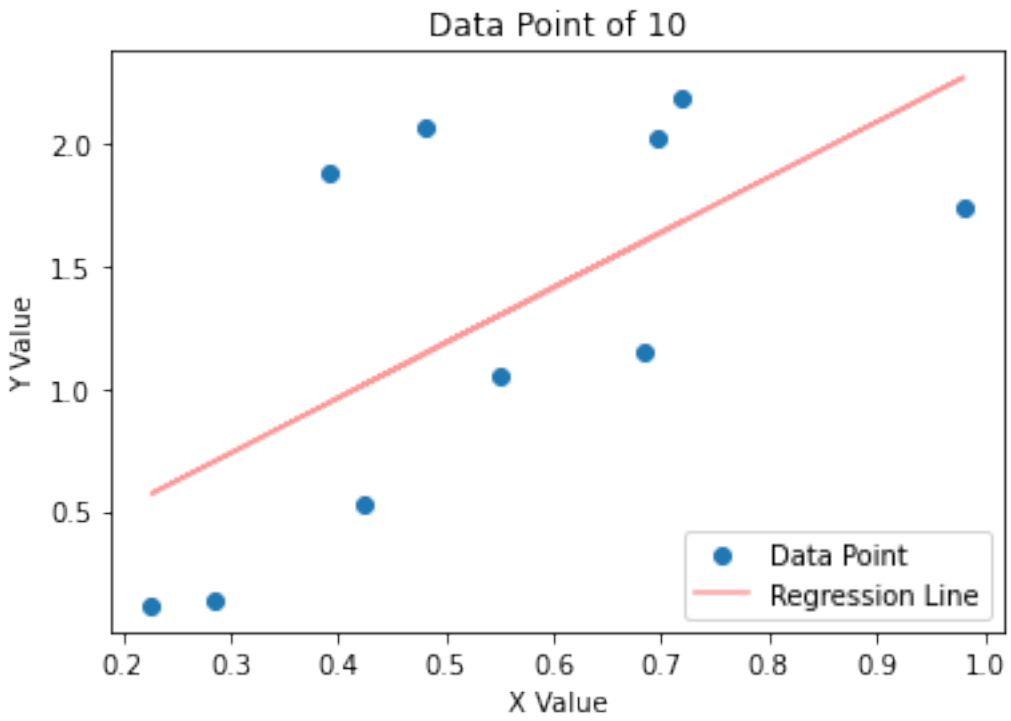
```
print("Coefficient:", coeffs[0])

# Compute SSE
sse = sum((y-y_pred)**2)
print("SSE:", sse)
```

```
Intercept: 0.13583178849897345
Coefficient: 1.8342321813863356
SSE: 15.356374565056809
```

1.1.1 Exercise

- Generate your own data of shape `(10,)`.
- Fit a line and estimate the SSE
- Make a single figure and plot, data, and line in different colors

[15]:
```
# Generate Data
np.random.seed(123)
x_10 = np.random.rand(10)
y_10 = 2*x_10 + 0.5*np.random.randn(10)

# Fir a linear regression model
coeffs_10 = np.polyfit(x_10,y_10,1)
y_10_fit = np.polyval(coeffs_10, x_10)

# Compute SSE
sse_10 = sum((y_10-y_10_fit)**2)

plt.scatter(x_10,y_10, label = 'Data Point')
plt.plot(x_10,y_10_fit, label = 'Regression Line', color = 'red', alpha = 0.4)
plt.xlabel('X Value')
plt.ylabel('Y Value')
plt.title('Data Point of 10')
plt.legend()
plt.show()
```



1.2 Linear regression using scikit-learn

The same operations of fitting and evaluating the fit of a regression model can also be implemented using a much more powerful set of toosl implemented in the machine learning library `scikit-learn`. `scikit-learn` has module dedicated to linear regression models called `LinearRegression`.

Let's import it:

[16]:
```
from sklearn.linear_model import LinearRegression
```

We can implement the operations shown about using `polyval` and `polyfit` using `LinearRegression` using the following lines:

[17]:
```
# Fit a linear model
model = LinearRegression()

# The fit method in LinearRegression only acceps predictors (x) as matrices.
# So we need to reshape our array:
X = np.array(x).reshape(-1, 1)
model.fit(X, y)
y_pred = model.predict(X)

# The coefficients can be extracted from the fit model as follows:
```

```
print("Intercept:", model.intercept_)
print("Coefficient:", model.coef_[0])

# Compute SSE
sse = sum((y-y_pred)**2)
print("SSE:", sse)
```

```
Intercept: 0.13583178849897382
Coefficient: 1.8342321813863351
SSE: 15.356374565056813
```

OK, besides the ideosyncracy of how `LinearRegression` accepts `x`, that was not very different. Instead of using `polyfit` and `polyval`, we used `model.fit` and `model.predict` and the results (parameers and MSE) were identical. Good.

Now, `LinearRegression` might seem a little bit more complicated because, oh well, it is more complicated but also much more powerful!

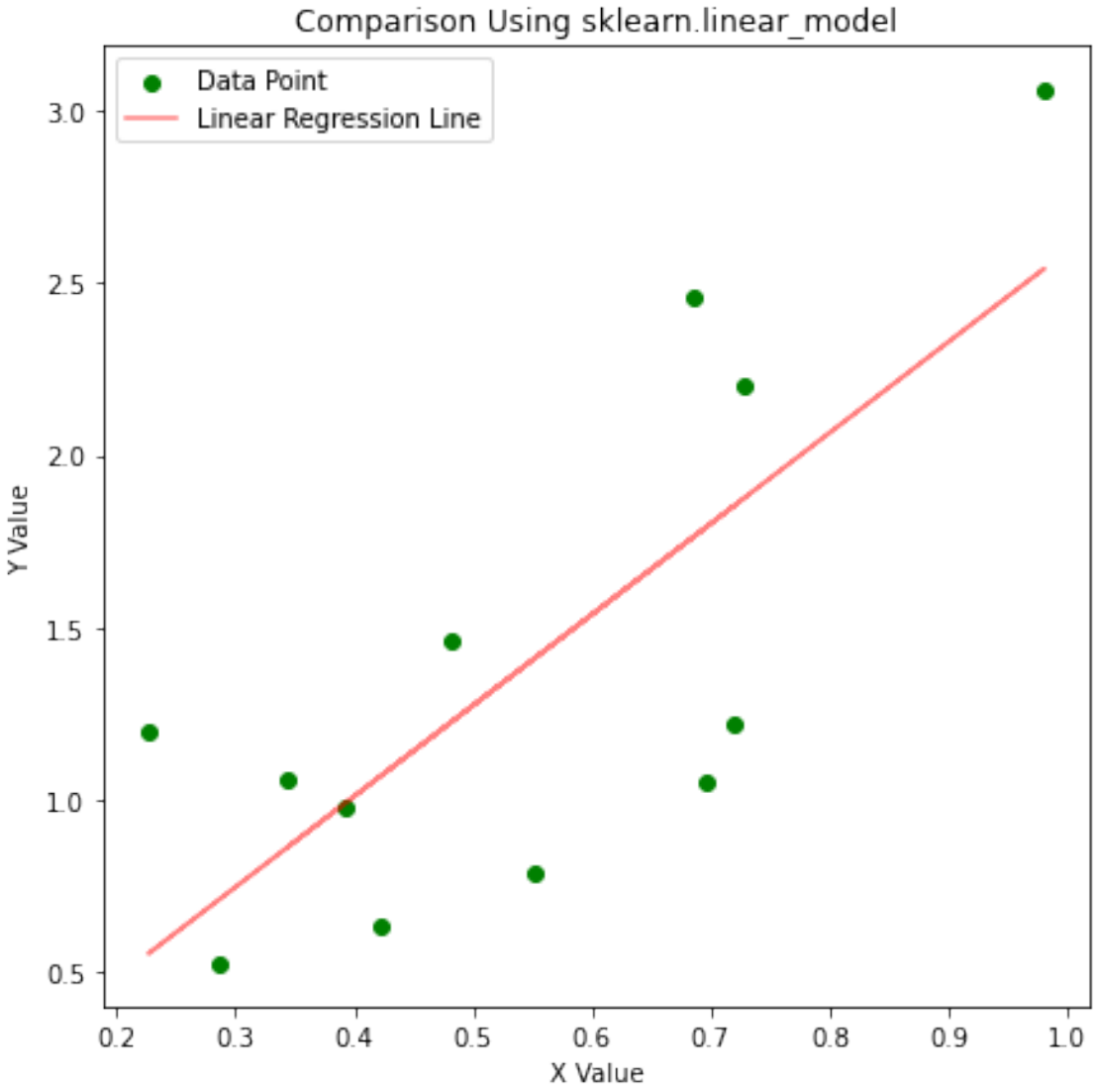
1.2.1 Exercise

- Generate a new data set of shape `(12,)`.
- Fit a line using `scikit-learn.linear_model` and estimate the SSE
- Make a single figure and plot, data, and line in different colors

[29]:
```
# Generate Data
np.random.seed(123)
x_12 = np.random.rand(12)
y_12 = 2*x_12 + 0.5*np.random.randn(12)

# The fit method in LinearRegression only acceps predictors (x) as matrices.
# So we need to reshape our array:
X_12 = np.array(x_12).reshape(-1, 1)
model.fit(X_12, y_12)
y_pred_12 = model.predict(X_12)

# The coefficients can be extracted from the fit model as follows:
print("Intercept:", model.intercept_)
print("Coefficient:", model.coef_[0])

# Compute SSE
sse_12 = sum((y_12-y_pred_12)**2)
print("SSE:", sse_12)

plt.figure(figsize=(7,7))
plt.scatter(x_12,y_12, color = 'green', label = 'Data Point')
plt.plot(x_12,y_pred_12, color = 'red', label = 'Linear Regression Line', alpha␣
↪= 0.5)
```

```
plt.xlabel('X Value')
plt.ylabel('Y Value')
plt.title('Comparison Using sklearn.linear_model')
plt.legend()
plt.show()
```

```
Intercept: -0.047910090056658694
Coefficient: 2.641784964208517
SSE: 2.955680135917802
```



1.3 Quality of fit metrics

Linear regession is more generally referred to as *Ordinary Linear Square Regression* or *OLS* Regression.

This is because the approach in regression is to minimise the sum of square errors (SSEs) between the data and the prediction of a model. The parameters of the model are adjusted so as to reduce the SSE and eventually minimize it.

We have seen before how to compute the SSE.

[22]:
```
print('SSE:',sum((y-y_pred)**2))
```

```
SSE: 15.356374565056813
```

In addition to SSE there are other measures of error important to learn about.

When fitting OLS regression models we attempt to explain some proportion of the variability in the data with a model. More specifically, we try to explain some proportion fo the *variance* in the data using the model. So models are generally judged by the proportion of variance in the data that they can explain.

The proportion of variance explained is a measure that describes the amount of variation in the dependent variable (y) that can be explained by the independent variable(s) (x) in a statistical model, such as a linear regression model.

When we fit a regression model, we are trying to find a line (or curve) that best represents the relationship between the independent variable(s) and the dependent variable. The amount of variation in the dependent variable that can be explained by the independent variable(s) is determined by the fit of the regression line to the data points.

In the context of linear regression, the total sum of squares (TSS) can be decomposed into two components: * the explained sum of squares (ESS) and * the residual sum of squares (RSS or as called until now, the sum of squared error, SSE).

The explained sum of squares (ESS) is the sum of squares of the difference between the predicted values of the dependent variable and the mean of the dependent variable. It represents the amount of variability in the dependent variable that is explained by the independent variable(s) in the model.

$ESS = \Sigma(i -)^2$

[23]:
```
ESS = sum((y_pred - np.mean(y))**2)
print(ESS)
```

```
9.091390913253534
```

The residual sum of squares (RSS, a.k.a., SSE) is the sum of squares of the difference between the predicted values of the dependent variable and the actual values of the dependent variable. It represents the amount of variability in the dependent variable that is not explained by the independent variable(s) in the model.

$RSS = \Sigma(yi - i)^2$

[24]:
```
RSS = sum((y - y_pred)**2) #a.k.a. SSE
print(RSS)
```

```
15.356374565056813
```

The total sum of squares (TSS) is the sum of squares of the difference between the actual values of the dependent variable and the mean of the dependent variable. It represents the total amount of variability in the dependent variable.

$TSS = \Sigma(yi -)^2$

[25]:
```
TSS = sum((y - np.mean(y))**2)
print(TSS)
```

```
24.447765478310348
```

where y_i is the actual value of the dependent variable, $_i$ is the predicted value of the dependent variable, and is the mean of the dependent variable.

Note that $TSS = \Sigma(RSS + ESS)$

[26]:
```
print([TSS, RSS+ESS])
```

```
[24.447765478310348, 24.447765478310345]
```

So far we have used only the SSE to copute the quality of fit of a model. There are several alternatives to RSS (a.k.a., SSE) that can be used to estimate the quality of fit of a model. A few commonly used ones are:

- Mean squared error (MSE): MSE is calculated as SSE divided by the number of degrees of freedom in the model. It is a measure of the average squared difference between the predicted values of the dependent variable and the actual values, and is often used as a measure of the overall goodness of fit of a model.
- Root mean squared error (RMSE): RMSE is the square root of MSE and is often used as a more interpretable measure of the overall goodness of fit of a model. RMSE has the same units as the dependent variable and is more easily interpretable than MSE.
- Mean absolute error (MAE): MAE is a measure of the average absolute difference between the predicted values of the dependent variable and the actual values. It is less sensitive to outliers than SSE and can be more robust in the presence of extreme values.
- Coefficient of determination (R^2): R^2 is a measure of the proportion of variance in the dependent variable that is explained by the independent variables in the model. It ranges from 0 to 1, with higher values indicating a better fit between the model and the observed data. $R^2 = 1 - (SSE/TSS)$

Each one of these metrics is useful in different situations. Others also exist such as the K-L Divergence or Akaiake Information Criteria (AIK) or Baeysian Information Cirteria (BIC), we will cover some of these only in the future.

`scikit-learn` provides a convenient way to compute several goodness of fit metrics to evaluate model performance. The module `sklearn.metrics` can be imported and submodules within it

contain estimators of the goodness of fit of models:

[30]:
```
from sklearn.metrics import mean_absolute_error,mean_squared_error, r2_score

mae  = mean_absolute_error(y_true=y,y_pred=y_pred)
mse  = mean_squared_error(y_true=y,y_pred=y_pred) #squared=True
rmse = mean_squared_error(y_true=y,y_pred=y_pred,squared=False)
r2   = r2_score(y_true=y,y_pred=y_pred)

print("Mean Absolute Error (MAE):",mae)
print("Mean Squared Error (MSE):",mse)
print("Root-Mean Squared Error (RMSE):",rmse)
print("Coefficient of Determination (R²):",r2)
```

```
Mean Absolute Error (MAE): 0.4859694020486803
Mean Squared Error (MSE): 0.30712749130113615
Root-Mean Squared Error (RMSE): 0.5541908437543299
Coefficient of Determination (R²): 0.37187001492301086
```

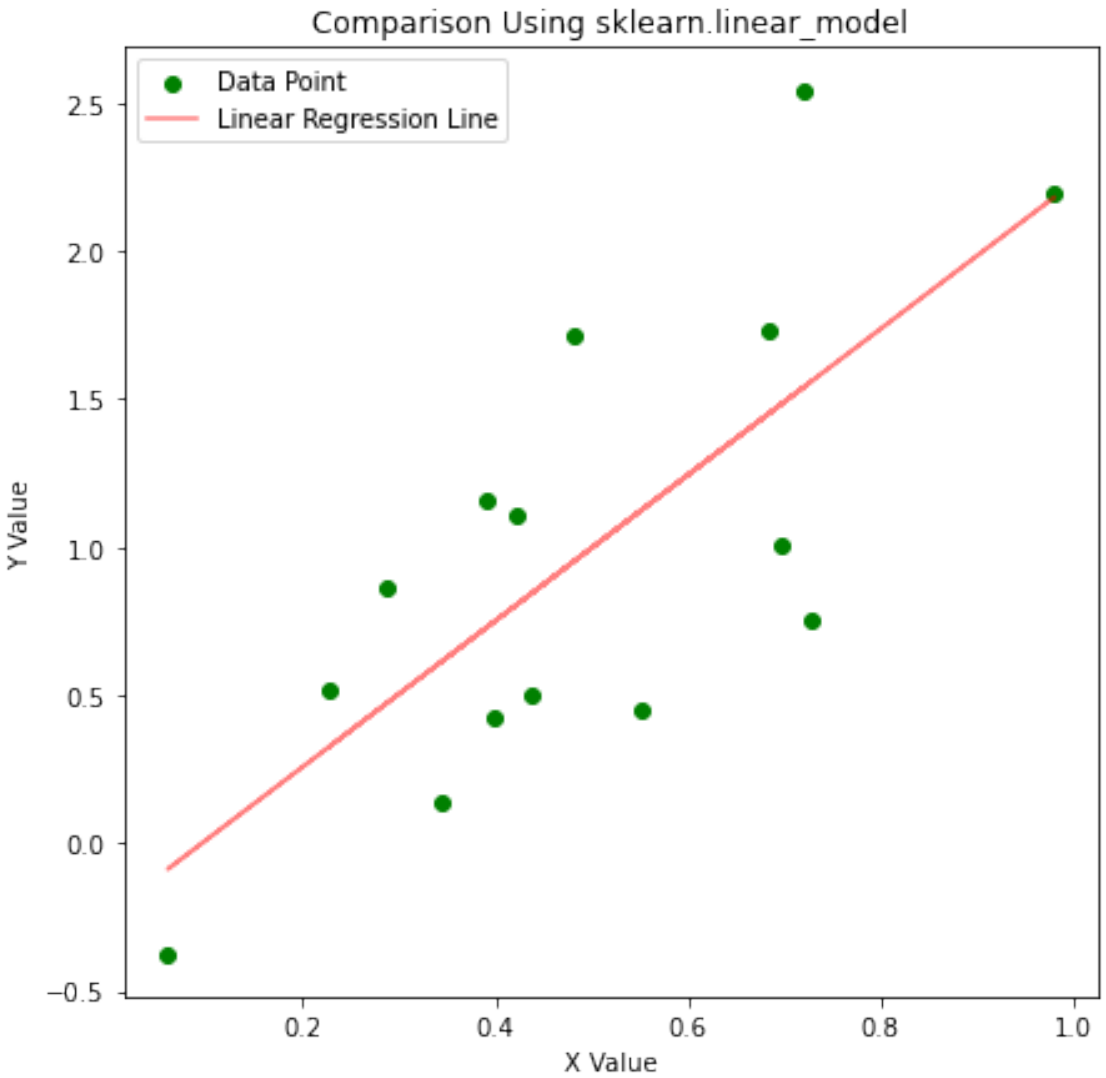
1.3.1 Exercise

- Generate a new data set of shape `(15,)`.
- Fit a line using `scikit-learn.linear_model` and estimate the SSE
- Use `scikit-learn.metrics` to estimate R^2

[38]:
```
# Generate Data
np.random.seed(123)
x_15 = np.random.rand(15)
y_15 = 2*x_15 + 0.5*np.random.randn(15)

# The fit method in LinearRegression only acceps predictors (x) as matrices.
# So we need to reshape our array:
X_15 = np.array(x_15).reshape(-1, 1)
model.fit(X_15, y_15)
y_pred_15 = model.predict(X_15)

# The coefficients can be extracted from the fit model as follows:
print("Intercept:", model.intercept_)
print("Coefficient:", model.coef_[0])

# Compute R^2
mae  = mean_absolute_error(y_true = y_15, y_pred = y_pred_15)
mse  = mean_squared_error(y_true = y_15 , y_pred= y_pred_15)    #squared=True
rmse = mean_squared_error(y_true = y_15 , y_pred = y_pred_15 , squared=False)
r2   = r2_score(y_true = y_15 , y_pred = y_pred_15)
print("Coefficient of Determination (R²):",r2)
```

```
plt.figure(figsize=(7,7))
plt.scatter(x_15,y_15, color = 'green', label = 'Data Point')
plt.plot(x_15,y_pred_15, color = 'red', label = 'Linear Regression Line', alpha␣
↪= 0.5)
plt.xlabel('X Value')
plt.ylabel('Y Value')
plt.title('Comparison Using sklearn.linear_model')
plt.legend()
plt.show()
```

```
Intercept: -0.23579614335089316
Coefficient: 2.464002578325705
Coefficient of Determination (R²): 0.5423134147541951
```



1.3.2 Linear regression using scikit-learn (generalized linear regression)

So, far we have used `scikit-learn`'s `LinearRegression` uniquely to predict `n y` variables from `n x` variables.

Yet, in practice we can think situations where we might have multiple variables (say `n x m` variables) and we would like to use them to predict a single set of `n` variables.

For example imagine the case of `m` repeated measures of `n` values and wanting to predict corresponding `n` values of another variable.

`LinearRegression` allows us to set up this type of modelling. This is the reason why the `X` variables must alswyas be 2D and above we had to make sure it was a 2D array.

To work this example, we will use one of the datasets that come with `scikit learn`, the Boston Housing database:

[53]:
```
import pandas as pd
import seaborn as sns
from sklearn.datasets import load_boston
```

[54]:
```
#import pandas as pd
#import numpy as np

#data_url = "http://lib.stat.cmu.edu/datasets/boston"
#raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
#data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
#target = raw_df.values[1::2, 2]
```

[58]:
```
boston_dataset = load_boston()
```

```
/Users/jesadathavornfung/opt/anaconda3/lib/python3.9/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function load_boston is
deprecated; `load_boston` is deprecated in 1.0 and will be removed in 1.2.

    The Boston housing prices dataset has an ethical problem. You can refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use of this
    dataset unless the purpose of the code is to study and educate about
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np


        data_url = "http://lib.stat.cmu.edu/datasets/boston"
```

```
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames housing
    dataset. You can load the datasets as follows::

        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.

  warnings.warn(msg, category=FutureWarning)
```

To explore the dataset take a look at the Headers and Dictionary Keys. For example:

[59]:
```
print(boston_dataset.keys())
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename',
'data_module'])
```

Where:

- `data`: contains the information for various houses
- `target`: prices of the house
- `feature_names`: names of the features
- `DESCR`: describes the dataset

The dataset contains a series of attributes or features (variables) measured along different dimensions.

Take a look at:

[60]:
```
print(boston_dataset.DESCR)
```

```
.. _boston_dataset:

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

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value
```

```
(attribute 14) is usually the target.

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

    :Missing Attribute Values: None

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

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


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

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

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

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential
Data and Sources of Collinearity', 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.
```

The last variable `MEDV` (or median value) is our interest. It is the median value of homes in thousands of dollars.

The dataset contains `.data` and `.target`

[61]:
```
print(boston_dataset.data)
```

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

[62]:
```
print(boston_dataset.target)
```

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

```
 11.9 27.9 17.2 27.5 15.  17.2 17.9 16.3  7.   7.2  7.5 10.4  8.8  8.4
 16.7 14.2 20.8 13.4 11.7  8.3 10.2 10.9 11.   9.5 14.5 14.1 16.1 14.3
 11.7 13.4  9.6  8.7  8.4 12.8 10.5 17.1 18.4 15.4 10.8 11.8 14.9 12.6
 14.1 13.  13.4 15.2 16.1 17.8 14.9 14.1 12.7 13.5 14.9 20.  16.4 17.7
 19.5 20.2 21.4 19.9 19.  19.1 19.1 20.1 19.9 19.6 23.2 29.8 13.8 13.3
 16.7 12.  14.6 21.4 23.  23.7 25.  21.8 20.6 21.2 19.1 20.6 15.2  7.
  8.1 13.6 20.1 21.8 24.5 23.1 19.7 18.3 21.2 17.5 16.8 22.4 20.6 23.9
 22.  11.9]
```

For convenience we are going to create a smalled table of features:

[63]:
```
boston = pd.DataFrame(boston_dataset.data, columns=boston_dataset.feature_names)
boston.head()
```

[63]:

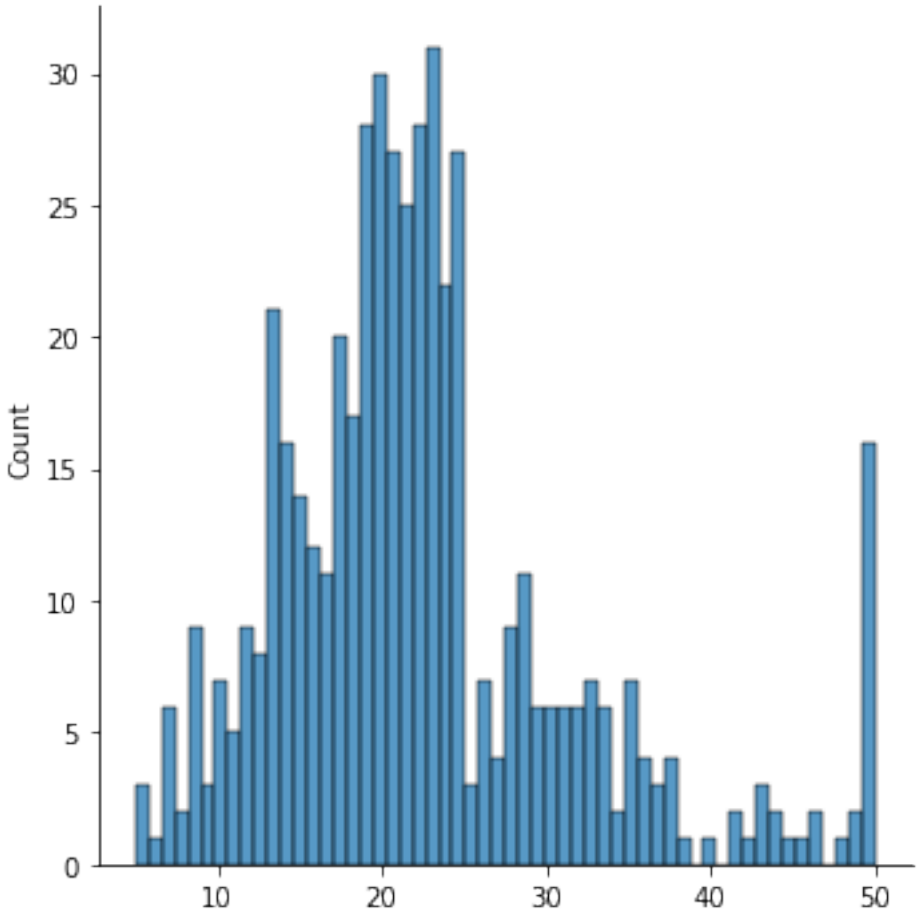
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	\
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	
4	0.06905	0.0	2.18	0.0	0.458	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

We can take a look at the median house value:

[64]:
```
sns.displot(boston_dataset.target, bins=56)
```

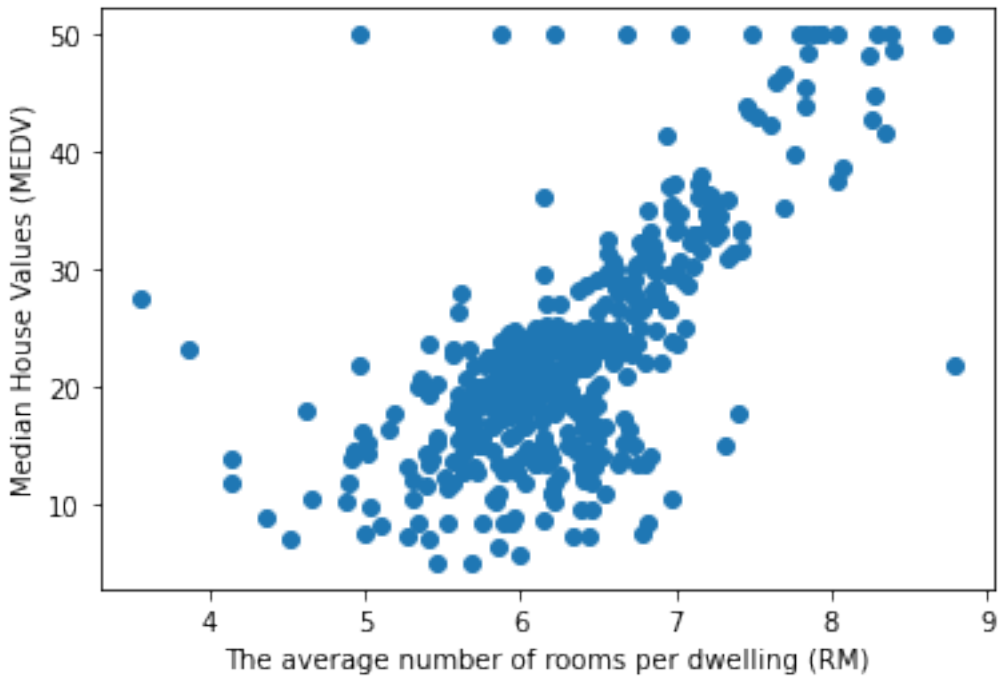
[64]:
```
<seaborn.axisgrid.FacetGrid at 0x7fc7c4a97e50>
```



We can explore the relationship between some of the features in the data and the target variable:

[65]:
```
x = boston['RM']
y = boston_dataset.target
plt.scatter(x, y, marker='o')
plt.xlabel('The average number of rooms per dwelling (RM)')
plt.ylabel('Median House Values (MEDV)')
```

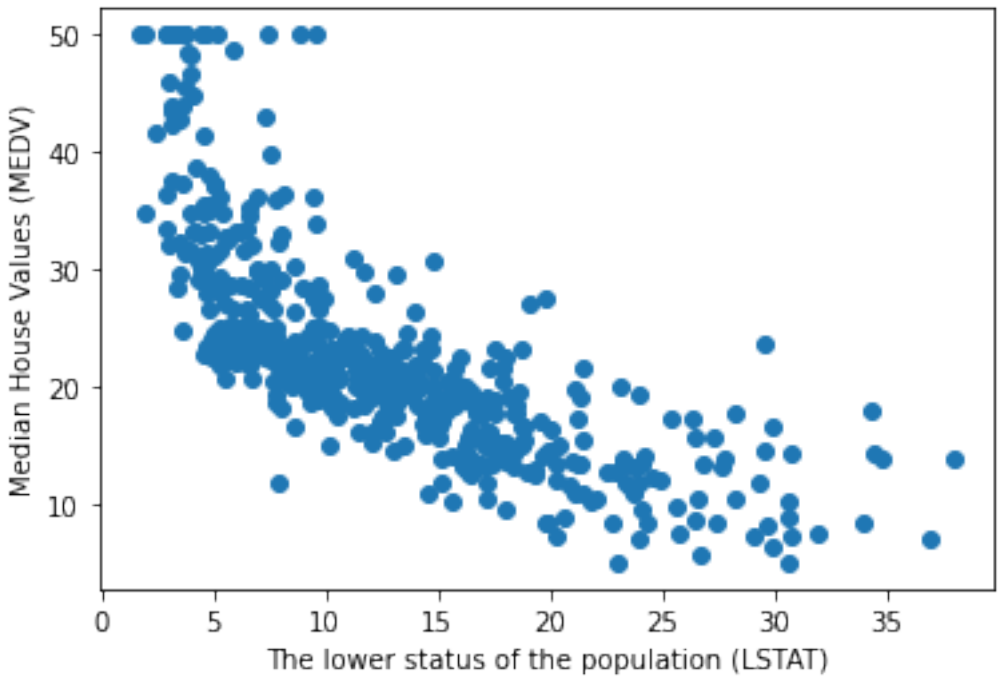
[65]:
```
Text(0, 0.5, 'Median House Values (MEDV)')
```



OK, it looks like there are features (like "RM") that have a relationship with the Median House Values. Let's try another feature:

[66]:
```
x = boston['LSTAT']
y = boston_dataset.target
plt.scatter(x, y, marker='o')
plt.xlabel('The lower status of the population (LSTAT)')
plt.ylabel('Median House Values (MEDV)')
```

[66]:
```
Text(0, 0.5, 'Median House Values (MEDV)')
```



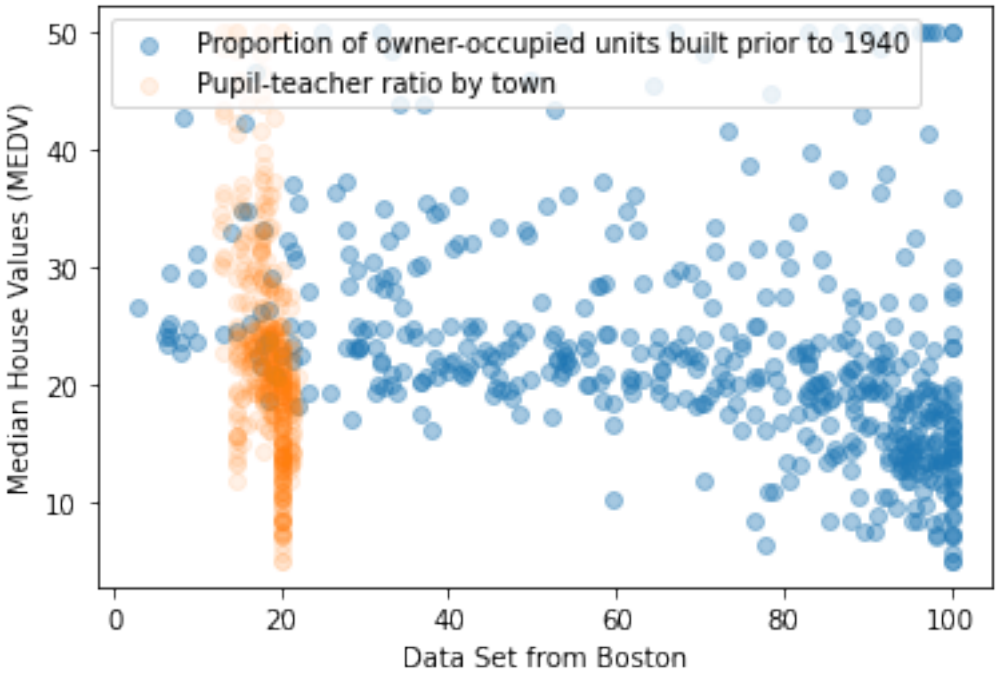
Also, a relationship. So, it looks like multiple features in the dataset have a relationship with the target variable (the median house value)

1.3.3 Exercise

- Explore the relationship between the target variable and two additional features of your choice. Make a plot.

[73]:
```
x = boston['AGE']
y = boston_dataset.target
plt.scatter(x, y, marker='o', alpha = 0.4, label = 'Proportion of␣
 ↪owner-occupied units built prior to 1940')
plt.scatter(boston['PTRATIO'], y, marker='o', alpha = 0.1, label =␣
 ↪'Pupil-teacher ratio by town')
plt.xlabel('Data Set from Boston')
plt.legend()
plt.ylabel('Median House Values (MEDV)')
```

[73]:
```
Text(0, 0.5, 'Median House Values (MEDV)')
```



It looks like multiple features have some relationship with the median house value in Boston.

So, it makes sense to think that a linear combination of all these variables should predict in some way the median house value. This is a case in which `m` variables (features) predict alltogether a target variable.

We will use `LinearRegression` to experiment with fitting a linear model where `m` features predict a single variable.

First let's organize the data:

[74]:
```
# get dependent and independent variables from the data set
X = boston_dataset.data
y = boston_dataset.target
```

Second, let's fit the linear regression model.

[75]:
```
housing_linear_regression = LinearRegression()
housing_linear_regression.fit(X, y)
```

[75]:
```
LinearRegression()
```

Third, we will use the model to predict the data, the median house value:

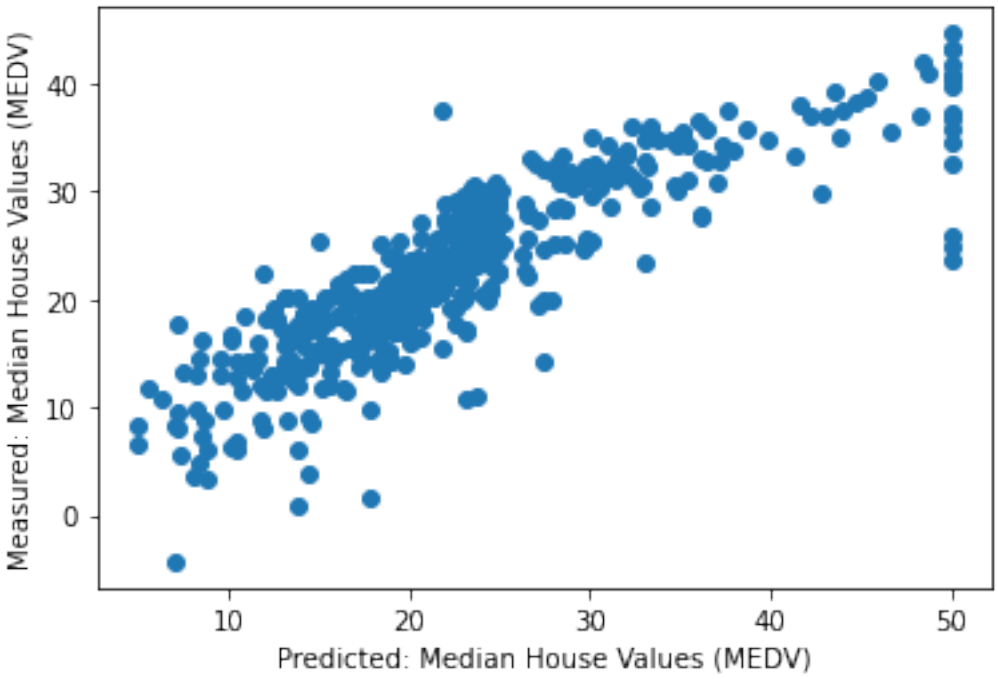
[76]:
```
y_pred = housing_linear_regression.predict(X)
```

Finally, we will compare using a plot the predicted and measured Median House Value

[77]:
```
y_data_array = np.array(y).reshape(-1, 1)
y_pred_array = np.array(y_pred).reshape(-1, 1)
```

[78]:
```
x = y_data_array
y = y_pred_array
plt.scatter(x, y, marker='o')
plt.xlabel('Predicted: Median House Values (MEDV)')
plt.ylabel('Measured: Median House Values (MEDV)')
```

[78]:
```
Text(0, 0.5, 'Measured: Median House Values (MEDV)')
```



1.3.4 Exercise

- Explain in your own words what you see in the previos Figure.
- Describe what the above experiment did
- How many features where in our model?
- How good was the quality of the fit (what was the R2 and MSE)?

  Based on the figure above, it appears that there is a postive relationship between the predicted *median house value (MEDV)* and the measured *median house value (MEDEV)*, which means that as the predicted values go up, the measured values tend to go up as well. Also, the predicted MEDV is only support value up to 50 because because even if

the measured MEDV goes up, the predicted MEDV never exceed 50.

Above experiment compared the between the predicted and measured MEDEV together using linear regression in which we use `LinearRegression` to experiment with fitting a linear model where `m` features predict a single variable.

In this case, the features are anything in `boston_dataset.data`, which could be 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 dollars), PTRATIO (pupil-teacher ratio by town), B (1000(Bk - 0.63)^2 where Bk is the proportion of black people by town), LSTAT (% lower status of the population), and **MEDV (Median value of owner-occupied homes in 1000's dollars)**.

[80]:
```
# Compute R^2
mae  = mean_absolute_error(y_true = y, y_pred = y_pred)
mse  = mean_squared_error(y_true = y , y_pred= y_pred)    #squared=True
rmse = mean_squared_error(y_true = y , y_pred = y_pred , squared=False)
r2   = r2_score(y_true = y , y_pred = y_pred)
print("Coefficient of Determination (R²):",r2)
```

```
Coefficient of Determination (R²): 1.0
```

The fit is good because the R^2 is 1.0, which means that the data fit well together.