Linear Regression with Python

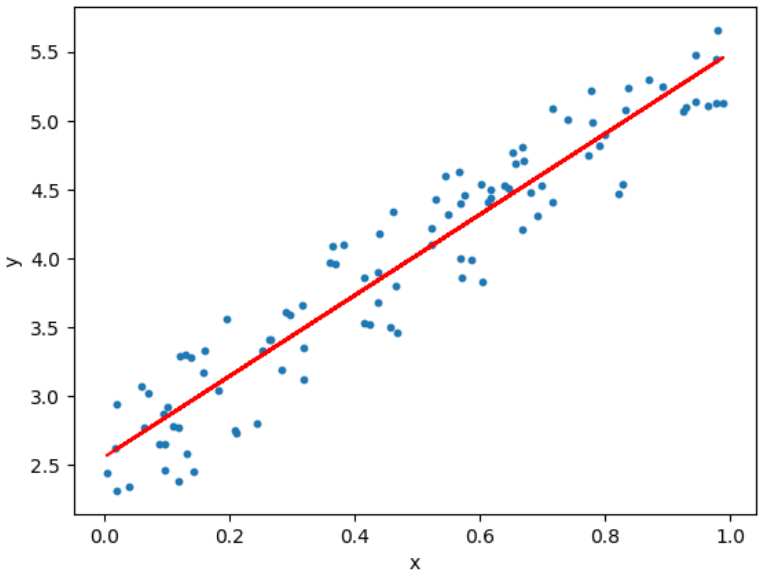
House Price Prediction on Boston Housing Dataset

You are a real estate agent and you want to predict the house price. It would be great if you can make some kind of automated system which predict price of a house based on various input which is known as feature.

Supervised Machine learning algorithms needs some data to train its model before making a prediction

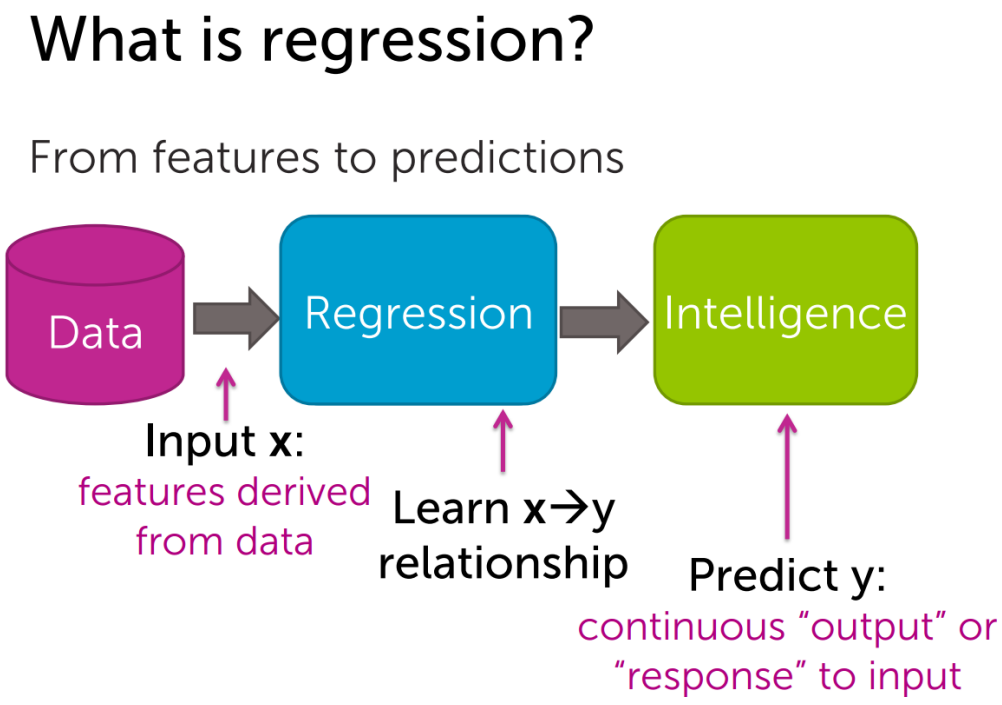
We have a Boston Dataset.

Where can Linear Regression be used?



It is a very powerful technique and can be used to understand the factors that influence profitability. It can be used to forecast sales in the coming months by analyzing the sales data for previous months. It can also be used to gain various insights about customer behaviour.

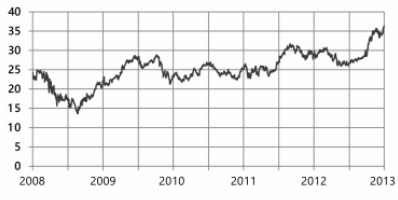
What is Linear Regression



Regression Examples

Stock prediction

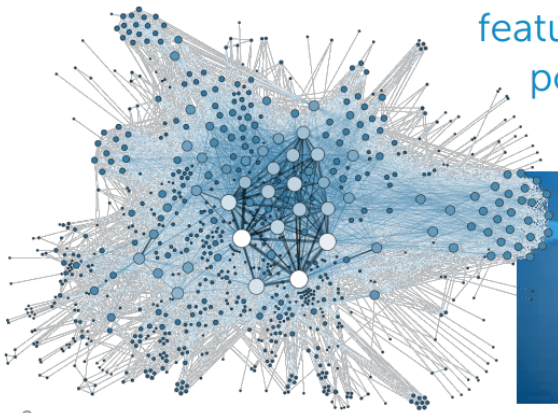
- Predict the price of a stock (y)
- Depends on **x** =
  - Recent history of stock price
  - News events
  - Related commodities



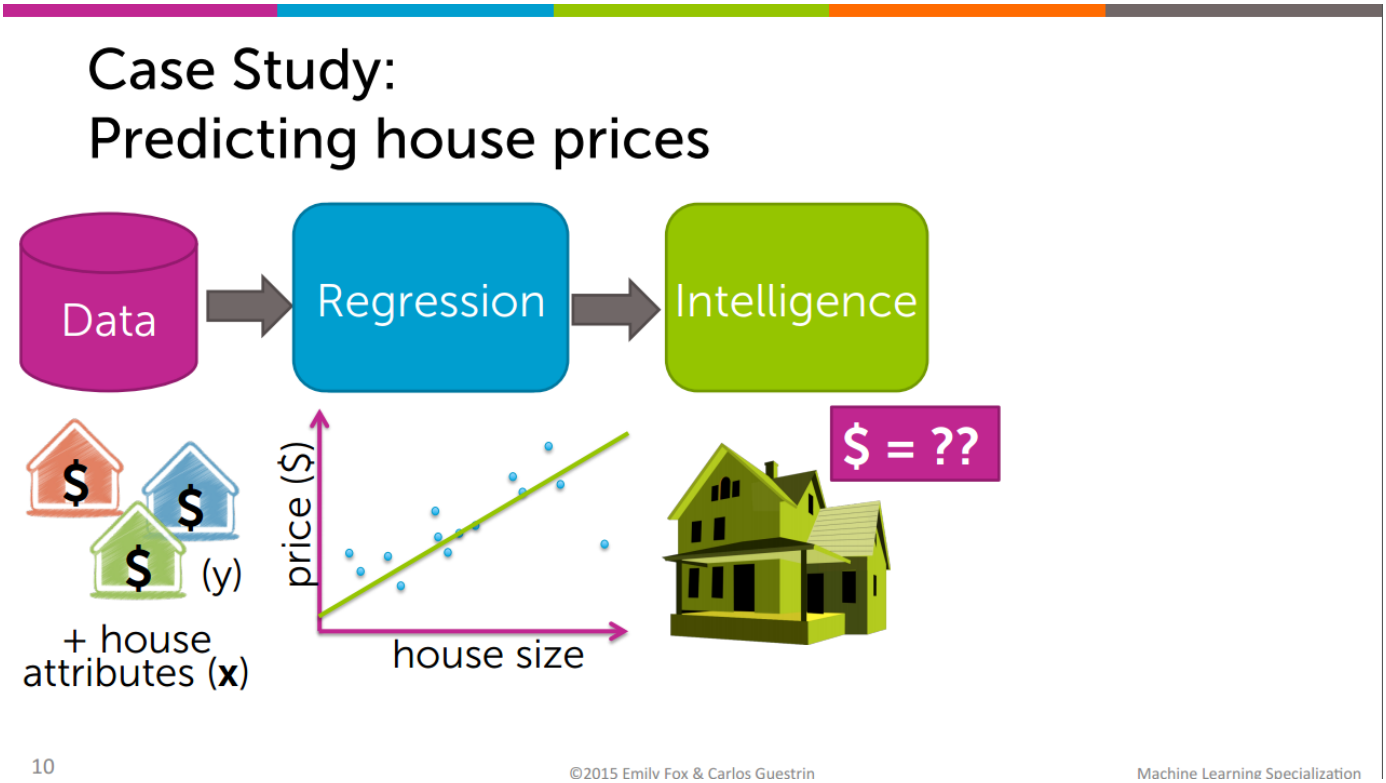
7 ©2015 Emily Fox & Carlos Guestrin Machine Learning Specialization

Tweet popularity

- How many people will retweet your tweet? (y)
- Depends on **x** = # followers,
  # of followers of followers,
  features of text tweeted,
  popularity of hashtag,
  # of past retweets,...

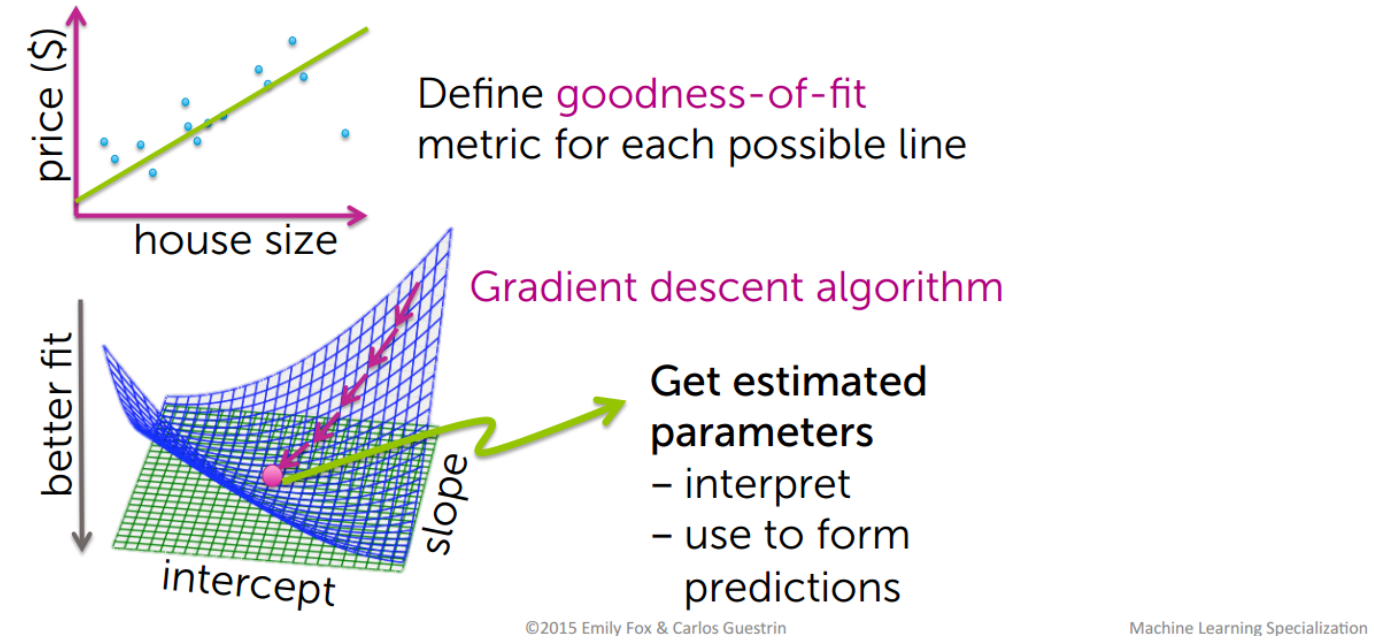


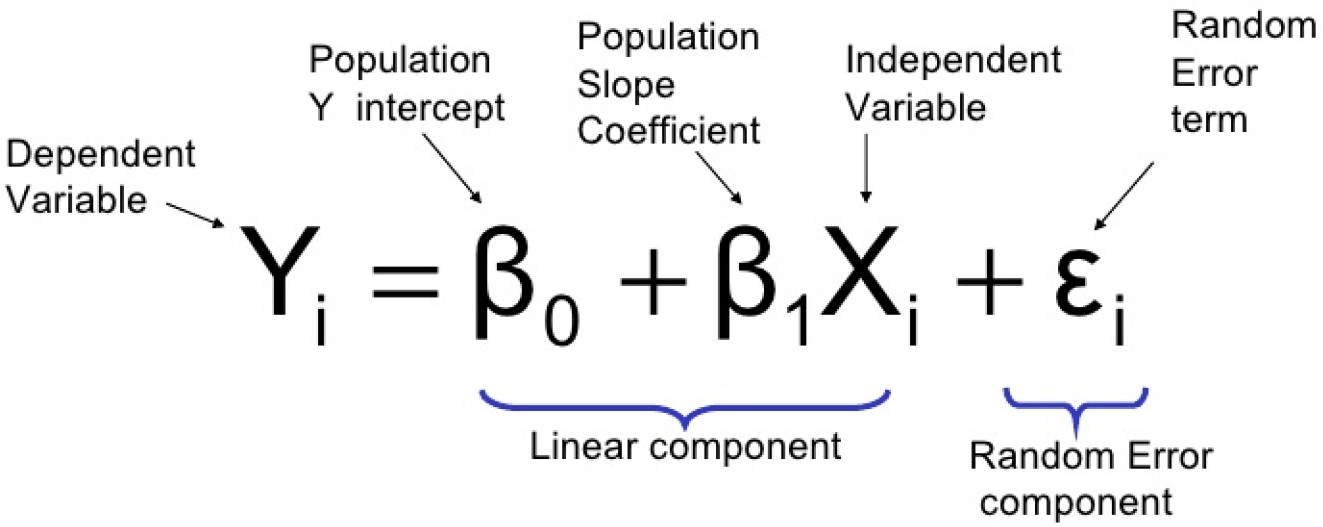
8 ©2015 Emily Fox & Carlos Guestrin Machine Learning Specialization



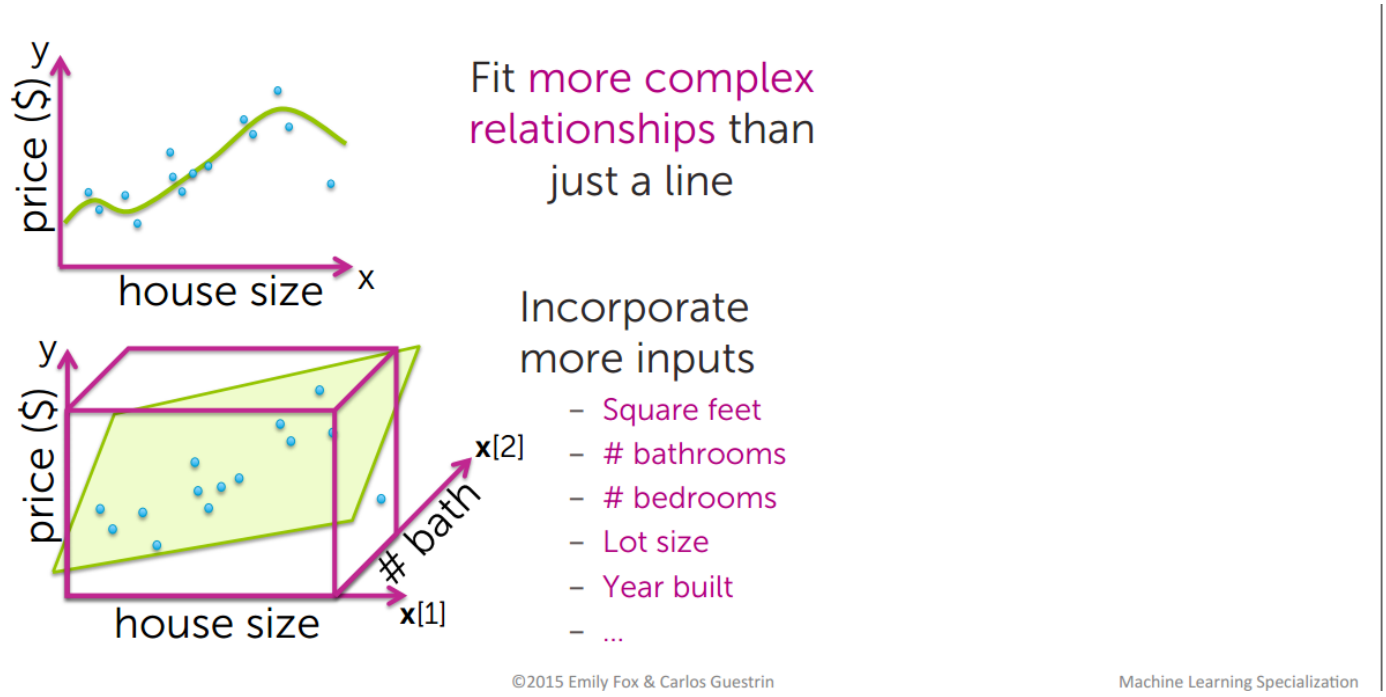
Regression Types

Simple Linear Regression

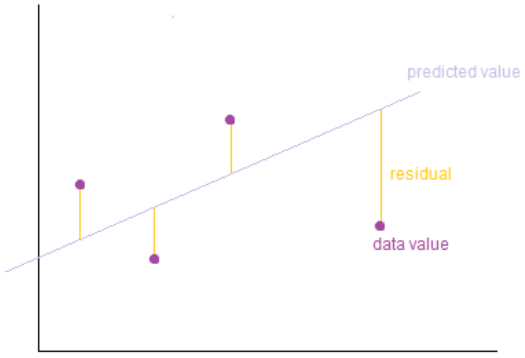




Multiple Linear Regression

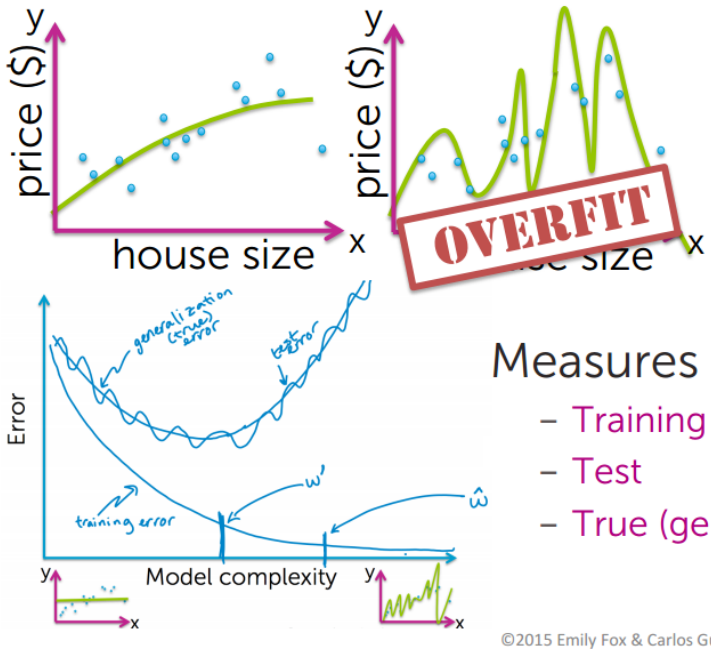


Assessing the performance of the model

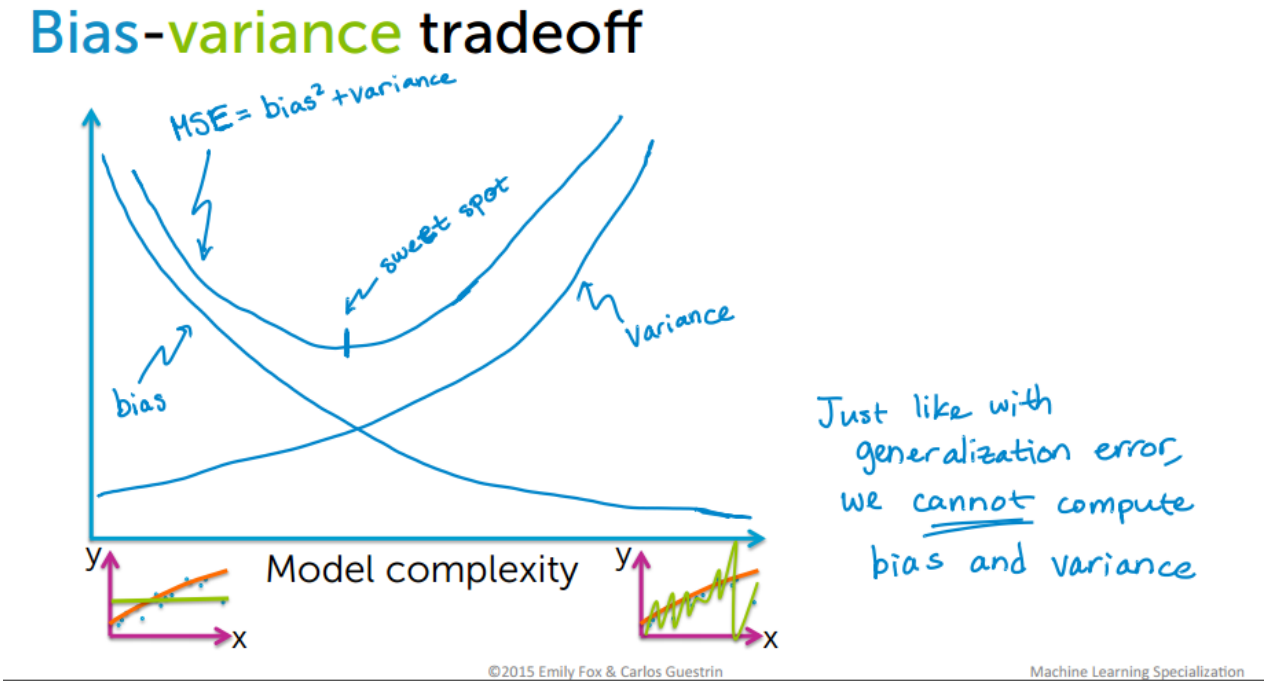


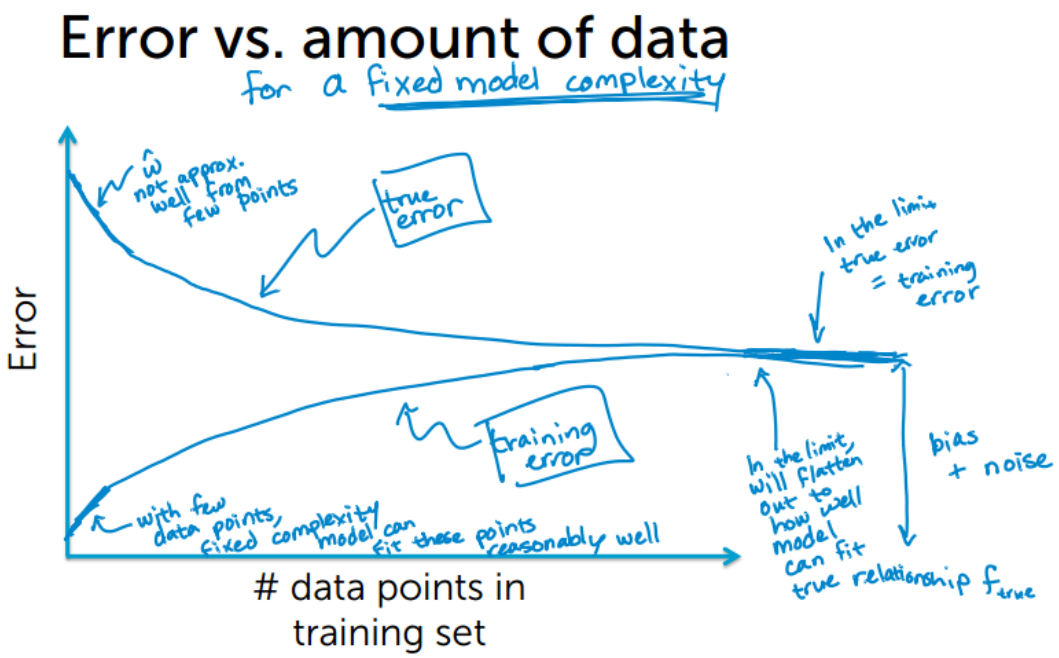
How do we determine the best fit line?

The line for which the the error between the predicted values and the observed values is minimum is called the best fit line or the regression line. These errors are also called as residuals. The residuals can be visualized by the vertical lines from the observed data value to the regression line.



Bias-Variance tradeoff





How to determine error

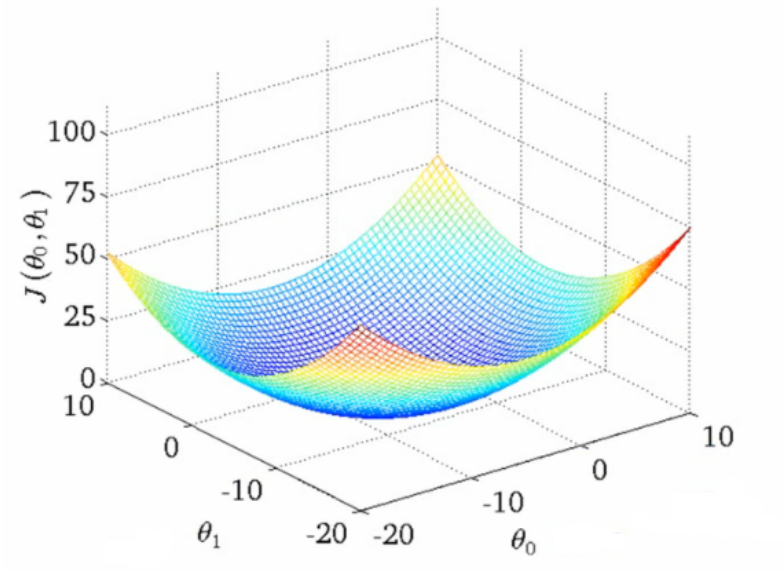
Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

Parameters: θ_0, θ_1

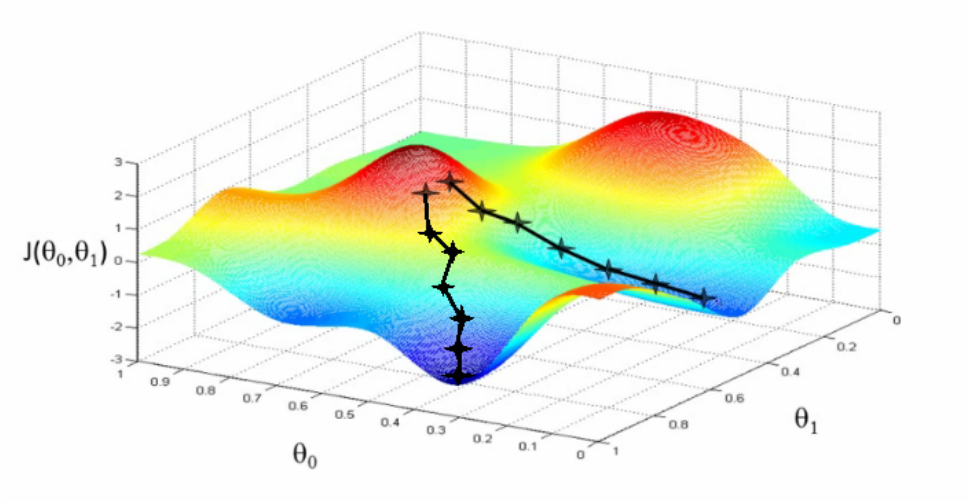
Cost Function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$

Goal: $\underset{\theta_0,\theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

Gradient Descent Algorithm to reduce the cost function



You might not end up in global minimum



Implimentation with sklearn

scikit-learn

Machine Learning in Python

Simple and efficient tools for data mining and data analysis

Accessible to everybody, and reusable in various contexts

Built on NumPy, SciPy, and matplotlib

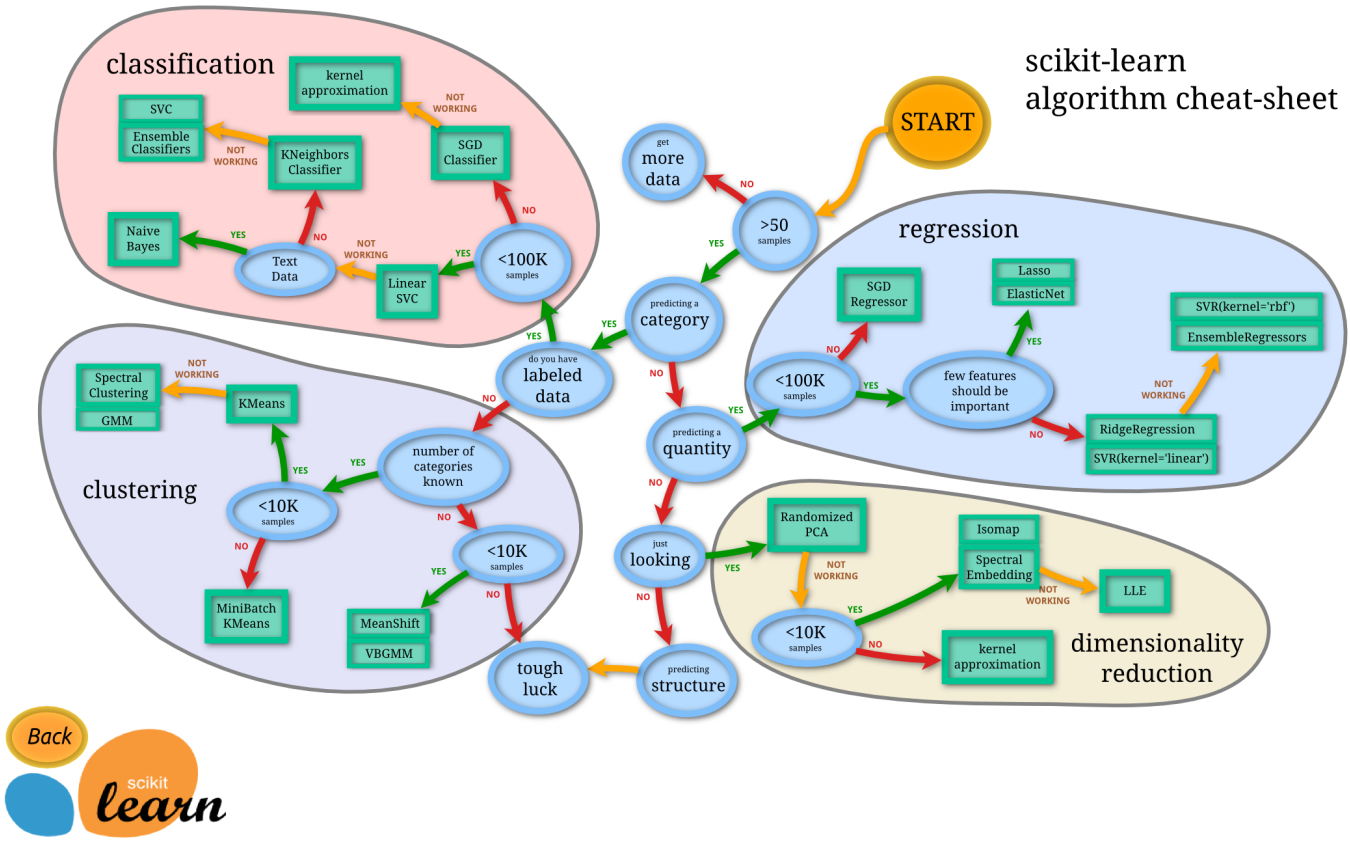
Open source, commercially usable - BSD license

Learn more here: https://scikit-learn.org/stable/ (https://scikit-learn.org/stable/)

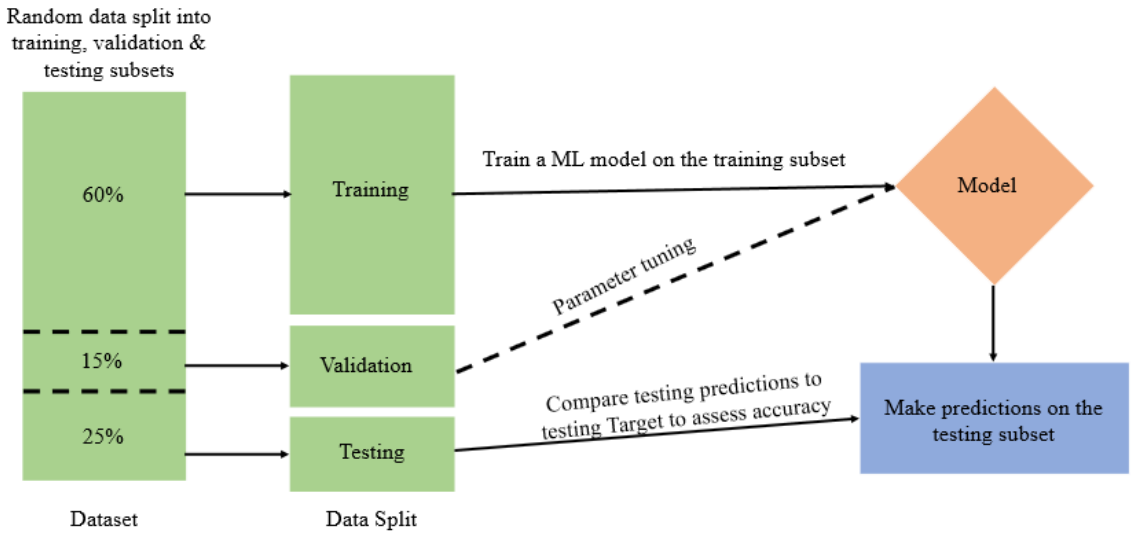
In [ ]:

```
! pip install -U scikit-learn
```

Image Source: https://cdn-images-1.medium.com/max/2400/1*2NR51X0FDjLB13u4WdYc4g.png (https://cdn-images-1.medium.com/max/2400/1*2NR51X0FDjLB13u4WdYc4g.png)



Training and testing splitting



Lets get started

In [1]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

In [3]:

```
from sklearn.datasets import load_boston

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error
```

In [4]:

```
boston = load_boston()
```

In [5]:

```
type(boston)
```

Out[5]:

```
sklearn.utils.Bunch
```

In [6]:

```
boston.keys()
```

Out[6]:

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

In [7]:

```
boston.DESCR
```

Out[7]:

```
".. _boston_dataset:\n\nBoston house prices dataset\n---------------------------\n\n**Data Set Characteristics:**  \n
\n    :Number of Instances: 506 \n\n    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attri
bute 14) is usually the target.\n\n    :Attribute Information (in order):\n        - CRIM     per capita crime rate b
y town\n        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.\n        - INDUS    propo
rtion of non-retail business acres per town\n        - CHAS     Charles River dummy variable (= 1 if tract bounds riv
er; 0 otherwise)\n        - NOX      nitric oxides concentration (parts per 10 million)\n        - RM       average n
umber of rooms per dwelling\n        - AGE      proportion of owner-occupied units built prior to 1940\n        - DIS
weighted distances to five Boston employment centres\n        - RAD      index of accessibility to radial highways\n
- TAX      full-value property-tax rate per $10,000\n        - PTRATIO  pupil-teacher ratio by town\n        - B
1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town\n        - LSTAT    % lower status of the population\n
- MEDV     Median value of owner-occupied homes in $1000's\n\n    :Missing Attribute Values: None\n\n    :Creator: Ha
rrison, D. and Rubinfeld, D.L.\n\nThis is a copy of UCI ML housing dataset.\nhttps://archive.ics.uci.edu/ml/machine-l
earning-databases/housing/\n\n\nThis dataset was taken from the StatLib library which is maintained at Carnegie Mello
n University.\n\nThe Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic\nprices and the demand for
clean air', J. Environ. Economics & Management,\nvol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression di
agnostics\n...', Wiley, 1980.   N.B. Various transformations are used in the table on\npages 244-261 of the latter.\n
\nThe Boston house-price data has been used in many machine learning papers that address regression\nproblems.   \n
\n.. topic:: References\n\n   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sour
ces of Collinearity', Wiley, 1980. 244-261.\n   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learnin
g. In Proceedings on the Tenth International Conference of Machine Learning, 236-243, University of Massachusetts, Am
herst. Morgan Kaufmann.\n"
```

The dataset used in this project comes from the UCI Machine Learning Repository. This data was collected in 1978 and each of the 506 entries represents aggregate information about 14 features of homes from various suburbs located in Boston. The features can be summarized as follows: CRIM: This is the per capita crime rate by town ZN: This is the proportion of residential land zoned for lots larger than 25,000 sq.ft. INDUS: This is the proportion of non-retail business acres per town. CHAS: This is the Charles River dummy variable (this is equal to 1 if tract bounds river; 0 otherwise) NOX: This is the nitric oxides concentration (parts per 10 million) RM: This is the average number of rooms per dwelling AGE: This is the proportion of owner-occupied units built prior to 1940 DIS: This is the weighted distances to five Boston employment centers RAD: This is the index of accessibility to radial highways TAX: This is the full-value property-tax rate per $10,000 PTRATIO: This is the pupil − teacher ratio by town B: This is calculated as 1000(Bk — 0.63)², 1000s where Bk is the proportion of people of African American descent by town LSTAT: This is the percentage lower status of the population MEDV: This is the median value of owner − occupied homes in

In [8]:

```
boston.feature_names
```

Out[8]:

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

In [9]:

```
boston.target
```

Out[9]:

```
array([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])
```

In [10]:

```
data = boston.data
type(data)
```

Out[10]:

```
numpy.ndarray
```

In [11]:

```
data.shape
```

Out[11]:

```
(506, 13)
```

The dataset used in this project comes from the UCI Machine Learning Repository. This data was collected in 1978 and each of the 506 entries represents aggregate information about 14 features of homes from various suburbs located in Boston. The features can be summarized as follows: CRIM: This is the per capita crime rate by town ZN: This is the proportion of residential land zoned for lots larger than 25,000 sq.ft. INDUS: This is the proportion of non-retail business acres per town. CHAS: This is the Charles River dummy variable (this is equal to 1 if tract bounds river; 0 otherwise) NOX: This is the nitric oxides concentration (parts per 10 million) RM: This is the average number of rooms per dwelling AGE: This is the proportion of owner-occupied units built prior to 1940 DIS: This is the weighted distances to five Boston employment centers RAD: This is the index of accessibility to radial highways TAX: This is the full-value property-tax rate per $10,000 PTRATIO : This is the pupil − teacher ratio by town B : This is calculated as 1000(Bk — 0.63)^2,$ 1000s
$where Bk is the proportion of people of African American descent by town LSTAT : This is the percentage lower status of the population MEDV$
$: This is the median value of owner − occupied homes in$

In [12]:

```
data = pd.DataFrame(data = data, columns= boston.feature_names)
data.head()
```

Out[12]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

In [13]:

```
data['Price'] = boston.target
data.head()
```

Out[13]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	Price
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Understand your data

In [14]:

```
data.describe()
```

Out[14]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.

In [15]:

```
data.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 float64
NOX        506 non-null float64
RM         506 non-null float64
AGE        506 non-null float64
DIS        506 non-null float64
RAD        506 non-null float64
TAX        506 non-null float64
PTRATIO    506 non-null float64
B          506 non-null float64
LSTAT      506 non-null float64
Price      506 non-null float64
dtypes: float64(14)
memory usage: 55.4 KB
```

In [17]:

```
data.isnull().sum()
```

Out[17]:

```
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
Price      0
dtype: int64
```

Data Visualization

We will start by creating a scatterplot matrix that will allow us to visualize the pair-wise relationships and correlations between the different features.

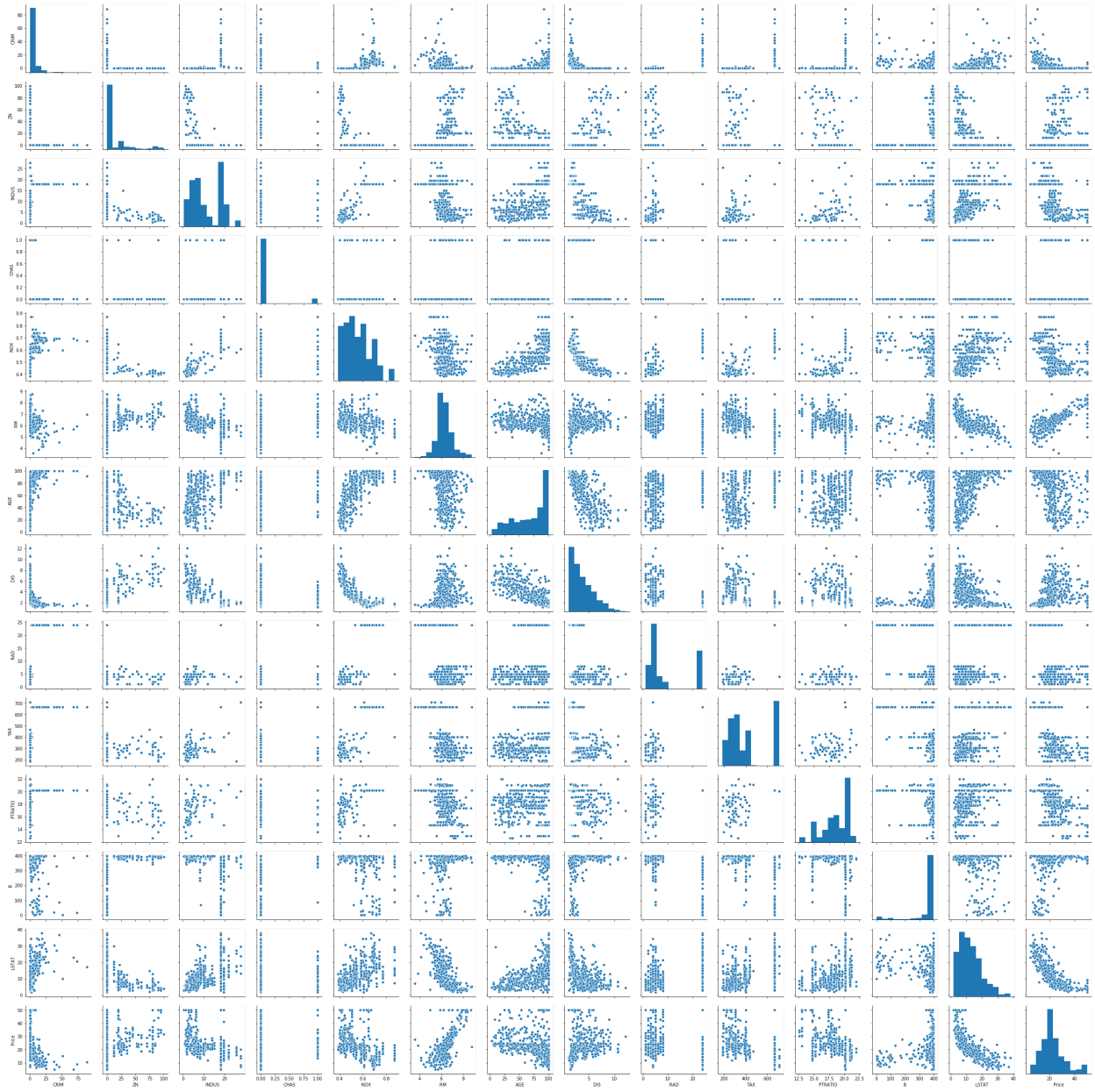
It is also quite useful to have a quick overview of how the data is distributed and wheter it cointains or not outliers.

In [18]:

```
sns.pairplot(data)
```

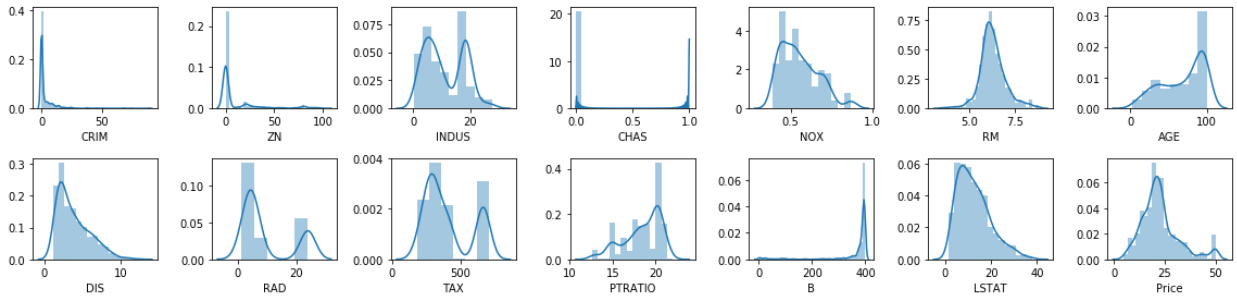
Out[18]:

```
<seaborn.axisgrid.PairGrid at 0x1ac5195c4e0>
```


CRIM
ZN
INDUS
CHAS
NOX
RM
AGE
DIS
RAD
TAX
PTRATIO
B
LSTAT
Price

In [22]:

```
rows = 2
cols = 7

fig, ax = plt.subplots(nrows= rows, ncols= cols, figsize = (16,4))

col = data.columns
index = 0

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

plt.tight_layout()
```



We are going to create now a correlation matrix to quantify and summarize the relationships between the variables.

This correlation matrix is closely related witn covariance matrix, in fact it is a rescaled version of the covariance matrix, computed from standardize features.

It is a square matrix (with the same number of columns and rows) that contains the Person's r correlation coefficient.

In [23]:

```
corrmat = data.corr()
corrmat
```

Out[23]:

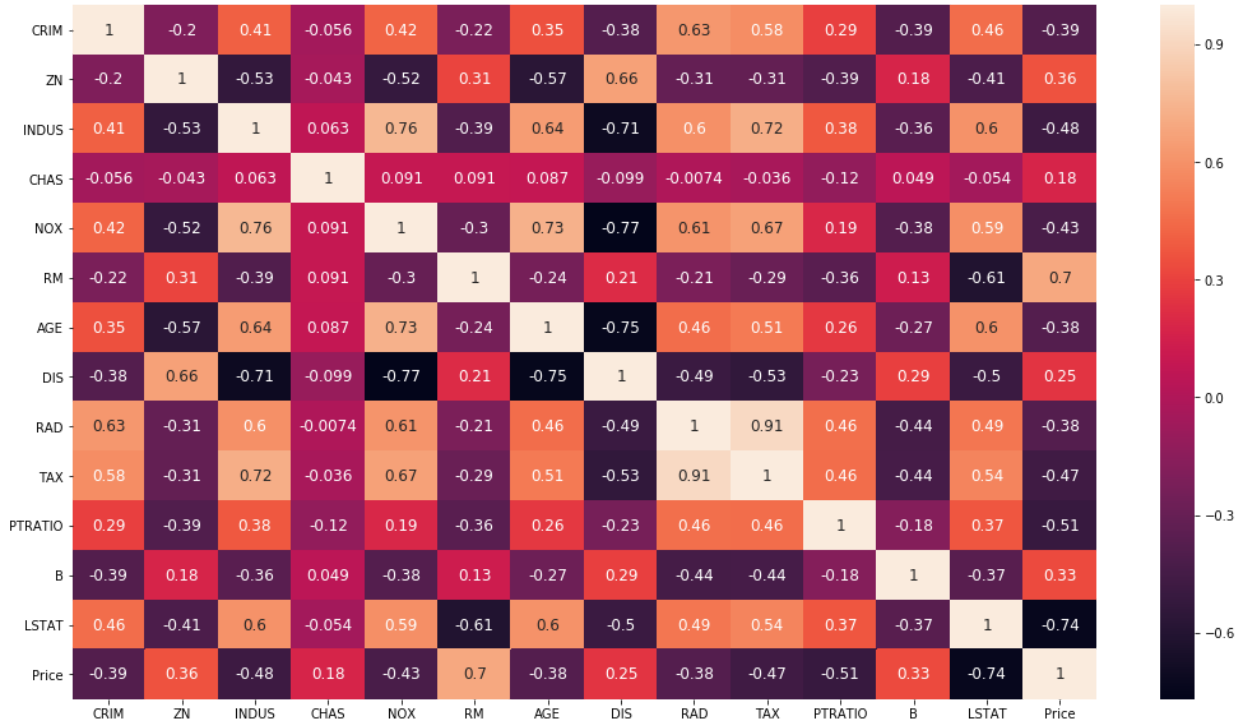
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LS
CRIM	1.000000	-0.200469	0.406583	-0.055892	0.420972	-0.219247	0.352734	-0.379670	0.625505	0.582764	0.289946	-0.385064	0.455
ZN	-0.200469	1.000000	-0.533828	-0.042697	-0.516604	0.311991	-0.569537	0.664408	-0.311948	-0.314563	-0.391679	0.175520	-0.412
INDUS	0.406583	-0.533828	1.000000	0.062938	0.763651	-0.391676	0.644779	-0.708027	0.595129	0.720760	0.383248	-0.356977	0.603
CHAS	-0.055892	-0.042697	0.062938	1.000000	0.091203	0.091251	0.086518	-0.099176	-0.007368	-0.035587	-0.121515	0.048788	-0.053
NOX	0.420972	-0.516604	0.763651	0.091203	1.000000	-0.302188	0.731470	-0.769230	0.611441	0.668023	0.188933	-0.380051	0.590
RM	-0.219247	0.311991	-0.391676	0.091251	-0.302188	1.000000	-0.240265	0.205246	-0.209847	-0.292048	-0.355501	0.128069	-0.613
AGE	0.352734	-0.569537	0.644779	0.086518	0.731470	-0.240265	1.000000	-0.747881	0.456022	0.506456	0.261515	-0.273534	0.602
DIS	-0.379670	0.664408	-0.708027	-0.099176	-0.769230	0.205246	-0.747881	1.000000	-0.494588	-0.534432	-0.232471	0.291512	-0.496
RAD	0.625505	-0.311948	0.595129	-0.007368	0.611441	-0.209847	0.456022	-0.494588	1.000000	0.910228	0.464741	-0.444413	0.488
TAX	0.582764	-0.314563	0.720760	-0.035587	0.668023	-0.292048	0.506456	-0.534432	0.910228	1.000000	0.460853	-0.441808	0.543
PTRATIO	0.289946	-0.391679	0.383248	-0.121515	0.188933	-0.355501	0.261515	-0.232471	0.464741	0.460853	1.000000	-0.177383	0.374
B	-0.385064	0.175520	-0.356977	0.048788	-0.380051	0.128069	-0.273534	0.291512	-0.444413	-0.441808	-0.177383	1.000000	-0.366
LSTAT	0.455621	-0.412995	0.603800	-0.053929	0.590879	-0.613808	0.602339	-0.496996	0.488676	0.543993	0.374044	-0.366087	1.000
Price	-0.388305	0.360445	-0.483725	0.175260	-0.427321	0.695360	-0.376955	0.249929	-0.381626	-0.468536	-0.507787	0.333461	-0.737

In [25]:

```
fig, ax = plt.subplots(figsize = (18, 10))
sns.heatmap(corrmat, annot = True, annot_kws={'size': 12})
```

Out[25]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1ac5dc6b2b0>
```



In [27]:

```
corrmat.index.values
```

Out[27]:

```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT', 'Price'], dtype=object)
```

In [28]:

```
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
```

In [29]:

```
threshold = 0.50
corr_value = getCorrelatedFeature(corrmat['Price'], threshold)
corr_value
```

Out[29]:

	Corr Value
RM	0.695360
PTRATIO	-0.507787
LSTAT	-0.737663
Price	1.000000

In [30]:

```
corr_value.index.values
```

Out[30]:

```
array(['RM', 'PTRATIO', 'LSTAT', 'Price'], dtype=object)
```

In [31]:

```
correlated_data = data[corr_value.index]
correlated_data.head()
```

Out[31]:

	RM	PTRATIO	LSTAT	Price
0	6.575	15.3	4.98	24.0
1	6.421	17.8	9.14	21.6
2	7.185	17.8	4.03	34.7
3	6.998	18.7	2.94	33.4
4	7.147	18.7	5.33	36.2

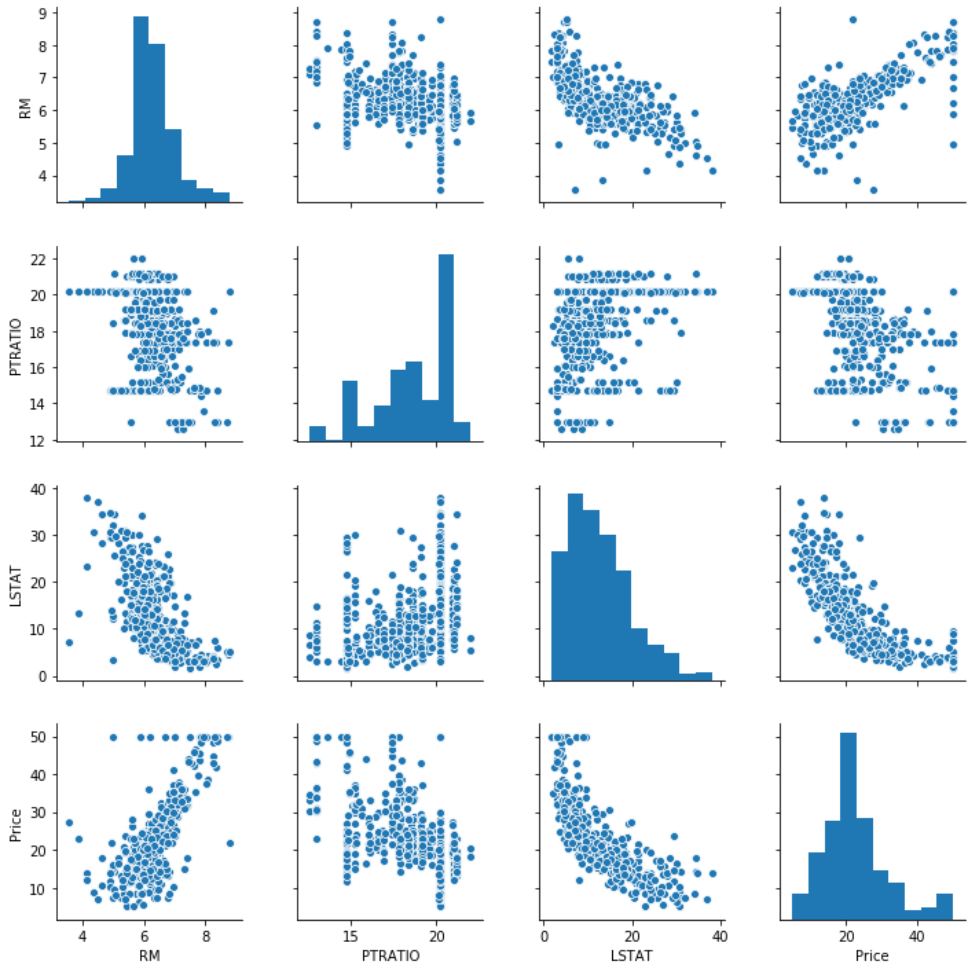
In [ ]:

```

```

Pairplot and Corrmat of correlated data

In [32]:

```
sns.pairplot(correlated_data)
plt.tight_layout()
```



In [33]:

```
sns.heatmap(correlated_data.corr(), annot=True, annot_kws={'size': 12})
```

Out[33]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1ac5fca6048>
```

Shuffle and Split Data

we will take the Boston housing dataset and split the data into training and testing subsets. Typically, the data is also shuffled into a random order when creating the training and testing subsets to remove any bias in the ordering of the dataset.

In [34]:

```
X = correlated_data.drop(labels=['Price'], axis = 1)
y = correlated_data['Price']
X.head()
```

Out[34]:

	RM	PTRATIO	LSTAT
0	6.575	15.3	4.98
1	6.421	17.8	9.14
2	7.185	17.8	4.03
3	6.998	18.7	2.94
4	7.147	18.7	5.33

In [35]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

In [37]:

```
X_train.shape, X_test.shape
```

Out[37]:

```
((404, 3), (102, 3))
```

Lets train the mode

In [38]:

```
model = LinearRegression()
model.fit(X_train, y_train)
```

Out[38]:

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

In [45]:

```
y_predict = model.predict(X_test)
```

In [47]:

```
df = pd.DataFrame(data = [y_predict, y_test])
df.T
```

Out[47]:

	0	1
0	27.609031	22.6
1	22.099034	50.0
2	26.529255	23.0
3	12.507986	8.3
4	22.254879	21.2
5	20.170639	19.9
6	19.667634	20.6
7	21.179452	18.7
8	17.053618	16.1
9	21.476452	18.6
10	14.611881	8.8
11	17.252031	17.2
12	17.878346	14.9
13	4.637631	10.5
14	39.493968	50.0
15	34.511718	29.0
16	21.513542	23.0
17	38.441143	33.3
18	30.015226	29.4
19	22.112007	21.0
20	25.017566	23.8
21	25.478218	19.1
22	18.299493	20.4
23	27.535253	29.1
24	22.044256	19.3
25	9.876631	23.1
26	17.644553	19.6
27	22.584952	19.4
28	35.564159	38.7
29	19.946952	18.7
...	...	...
72	29.871518	23.5
73	30.691957	31.2
74	25.406699	23.7
75	3.986648	7.4
76	37.154884	48.3
77	23.557903	24.4
78	26.098377	22.6
79	18.805641	18.3
80	29.097060	23.3
81	19.065406	17.1
82	18.009590	27.9
83	37.126569	44.8
84	38.802174	50.0
85	25.135660	23.0
86	23.331702	21.4
87	16.937407	10.2
88	30.835554	23.3
89	16.083255	23.2
90	16.644241	18.9
91	15.952794	13.4
92	27.660842	21.9
93	33.021234	24.8
94	21.346126	11.9
95	23.101565	24.3

	0	1
96	-3.131354	13.8
97	28.271228	24.7
98	18.467419	14.1
99	18.558070	18.7
100	24.681964	28.1
101	20.826879	19.8

102 rows × 2 columns

Defining performance metrics

It is difficult to measure the quality of a given model without quantifying its performance over training and testing. This is typically done using some type of performance metric, whether it is through calculating some type of error, the goodness of fit, or some other useful measurement. For this project, you will be calculating the coefficient of determination, R2, to quantify your model's performance. The coefficient of determination for a model is a useful statistic in regression analysis, as it often describes how "good" that model is at making predictions.

The values for R2 range from 0 to 1, which captures the percentage of squared correlation between the predicted and actual values of the target variable. A model with an R2 of 0 always fails to predict the target variable, whereas a model with an R2 of 1 perfectly predicts the target variable. Any value between 0 and 1 indicates what percentage of the target variable, using this model, can be explained by the features. A model can be given a negative R2 as well, which indicates that the model is no better than one that naively predicts the mean of the target variable.

For the performance_metric function in the code cell below, you will need to implement the following:

Use r2_score from sklearn.metrics to perform a performance calculation between y_true and y_predict. Assign the performance score to the score variable.

Now we will find R^2 which is defined as follows,

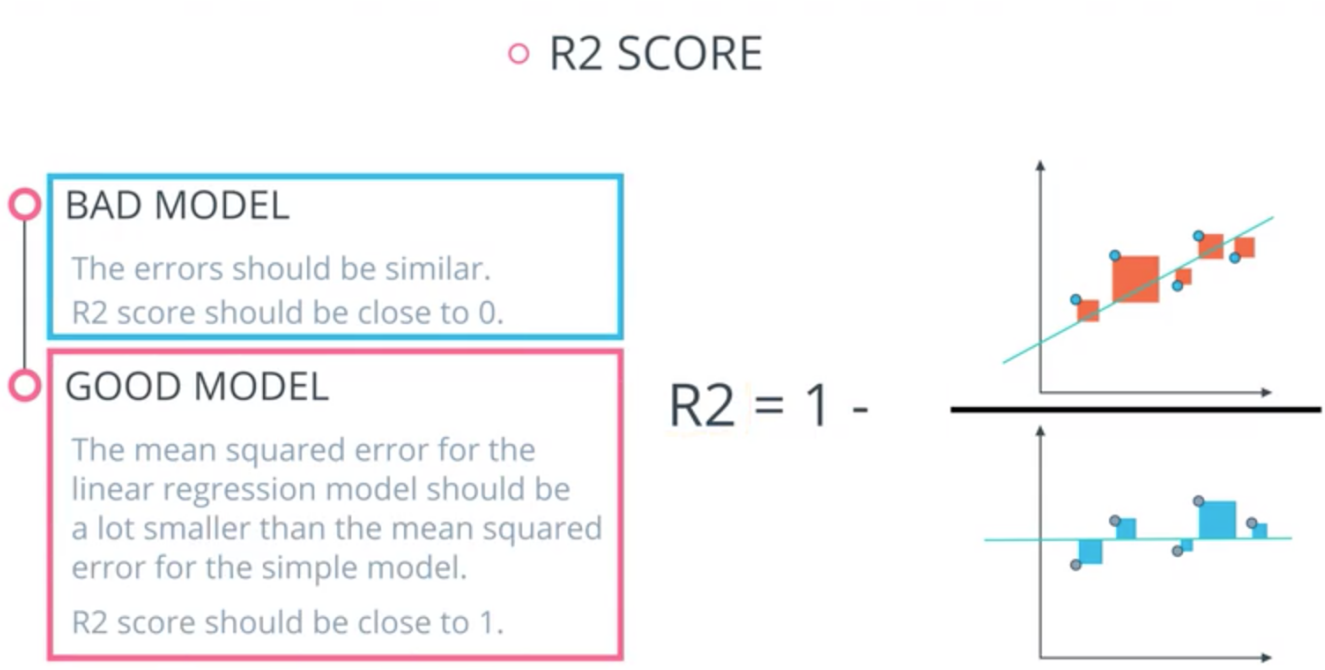
$$SS_t = \sum_{i=1}^{m} (y_i - \bar{y})^2$$

$$SS_r = \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

$$R^2 = 1 - \frac{SS_r}{SS_t}$$

SS_t is the total sum of squares and SS_r is the total sum of squares of residuals. R^2 Score usually range from 0 to 1. It will also become negative if the model is completely wrong.

Now we will find R^2 Score.



Regression Evaluation Metrics

Here are three common evaluation metrics for regression problems:

Mean Absolute Error (MAE) is the mean of the absolute value of the errors:

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Mean Squared Error (MSE) is the mean of the squared errors:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Root Mean Squared Error (RMSE) is the square root of the mean of the squared errors:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Comparing these metrics:

- **MAE** is the easiest to understand, because it's the average error.
- **MSE** is more popular than MAE, because MSE "punishes" larger errors, which tends to be useful in the real world.
- **RMSE** is even more popular than MSE, because RMSE is interpretable in the "y" units.

All of these are **loss functions**, because we want to minimize them.

In [63]:

```
from sklearn.metrics import r2_score
```

In [64]:

```
correlated_data.columns
```

Out[64]:

```
Index(['RM', 'PTRATIO', 'LSTAT', 'Price'], dtype='object')
```

In [65]:

```
score = r2_score(y_test, y_predict)
mae = mean_absolute_error(y_test, y_predict)
mse = mean_squared_error(y_test, y_predict)

print('r2_score: ', score)
print('mae: ', mae)
print('mse: ', mse)
```

```
r2_score:  0.4881642015692508
mae:  4.404434993909257
mse:  41.67799012221682
```

Store feature performance

In [73]:

```
total_features = []
total_features_name = []
selected_correlation_value = []
r2_scores = []
mae_value = []
mse_value = []
```

In [76]:

```
def performance_metrics(features, th, y_true, y_pred):
    score = r2_score(y_true, y_pred)
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)

    total_features.append(len(features)-1)
    total_features_name.append(str(features))
    selected_correlation_value.append(th)
    r2_scores.append(score)
    mae_value.append(mae)
    mse_value.append(mse)

    metrics_dataframe = pd.DataFrame(data= [total_features_name, total_features, selected_correlation_value, r2_scores, mae_value,
mse_value],
                                     index = ['features name', '#feature', 'corr_value', 'r2_score', 'MAE', 'MSE'])
    return metrics_dataframe.T
```

In [77]:

```
performance_metrics(correlated_data.columns.values, threshold, y_test, y_predict)
```

Out[77]:

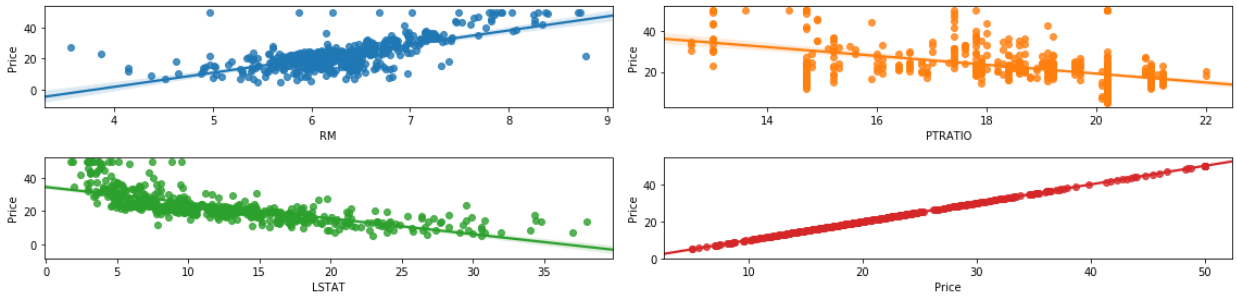
	features name	#feature	corr_value	r2_score	MAE	MSE
0	['RM' 'PTRATIO' 'LSTAT' 'Price']	3	0.5	0.488164	4.40443	41.678

regression plot of the features correlated with the House Price

In [78]:

```
rows = 2
cols = 2
fig, ax = plt.subplots(nrows=rows, ncols=cols, figsize = (16, 4))

col = correlated_data.columns
index = 0

for i in range(rows):
    for j in range(cols):
        sns.regplot(x = correlated_data[col[index]], y = correlated_data['Price'], ax = ax[i][j])
        index = index + 1
fig.tight_layout()
```

```
C:\ProgramData\Anaconda3\lib\site-packages\scipy\stats\stats.py:1713: FutureWarning: Using a non-tuple sequence for m
ultidimensional indexing is deprecated; use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be inter
preted as an array index, `arr[np.array(seq)]`, which will result either in an error or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



In [ ]:

In [ ]:

Let's find out other combination of columns to get better accuracy with >60%

In [79]:

```
corrmat['Price']
```

Out[79]:

```
CRIM      -0.388305
ZN         0.360445
INDUS     -0.483725
CHAS       0.175260
NOX       -0.427321
RM         0.695360
AGE       -0.376955
DIS        0.249929
RAD       -0.381626
TAX       -0.468536
PTRATIO   -0.507787
B          0.333461
LSTAT     -0.737663
Price      1.000000
Name: Price, dtype: float64
```

In [80]:

```
threshold = 0.60
corr_value = getCorrelatedFeature(corrmat['Price'], threshold)
corr_value
```

Out[80]:

	Corr Value
RM	0.695360
LSTAT	-0.737663
Price	1.000000

In [81]:

```
correlated_data = data[corr_value.index]
correlated_data.head()
```

Out[81]:

	RM	LSTAT	Price
0	6.575	4.98	24.0
1	6.421	9.14	21.6
2	7.185	4.03	34.7
3	6.998	2.94	33.4
4	7.147	5.33	36.2

In [82]:

```
def get_y_predict(corr_data):
    X = corr_data.drop(labels = ['Price'], axis = 1)
    y = corr_data['Price']

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
    model = LinearRegression()
    model.fit(X_train, y_train)
    y_predict = model.predict(X_test)
    return y_predict
```

In [83]:

```
y_predict = get_y_predict(correlated_data)
```

In [84]:

```
performance_metrics(correlated_data.columns.values, threshold, y_test, y_predict)
```

Out[84]:

	features name	#feature	corr_value	r2_score	MAE	MSE
0	['RM' 'PTRATIO' 'LSTAT' 'Price']	3	0.5	0.488164	4.40443	41.678
1	['RM' 'LSTAT' 'Price']	2	0.6	0.540908	4.14244	37.3831

Let's find out other combination of columns to get better accuracy > 70%

In [85]:

```
corrmat['Price']
```

Out[85]:

```
CRIM      -0.388305
ZN         0.360445
INDUS     -0.483725
CHAS       0.175260
NOX       -0.427321
RM         0.695360
AGE       -0.376955
DIS        0.249929
RAD       -0.381626
TAX       -0.468536
PTRATIO   -0.507787
B          0.333461
LSTAT     -0.737663
Price      1.000000
Name: Price, dtype: float64
```

In [86]:

```
threshold = 0.70
corr_value = getCorrelatedFeature(corrmat['Price'], threshold)
corr_value
```

Out[86]:

	Corr Value
LSTAT	-0.737663
Price	1.000000

In [87]:

```
correlated_data = data[corr_value.index]
correlated_data.head()
```

Out[87]:

	LSTAT	Price
0	4.98	24.0
1	9.14	21.6
2	4.03	34.7
3	2.94	33.4
4	5.33	36.2

In [88]:

```
y_predict = get_y_predict(correlated_data)
performance_metrics(correlated_data.columns.values, threshold, y_test, y_predict)
```

Out[88]:

	features name	#feature	corr_value	r2_score	MAE	MSE
0	['RM' 'PTRATIO' 'LSTAT' 'Price']	3	0.5	0.488164	4.40443	41.678
1	['RM' 'LSTAT' 'Price']	2	0.6	0.540908	4.14244	37.3831
2	['LSTAT' 'Price']	1	0.7	0.430957	4.86401	46.3363

Let's go ahead and select only RM feature

In [90]:

```
correlated_data = data[['RM', 'Price']]
correlated_data.head()
```

Out[90]:

	RM	Price
0	6.575	24.0
1	6.421	21.6
2	7.185	34.7
3	6.998	33.4
4	7.147	36.2

In [91]:

```
y_predict = get_y_predict(correlated_data)
performance_metrics(correlated_data.columns.values, threshold, y_test, y_predict)
```

Out[91]:

	features name	#feature	corr_value	r2_score	MAE	MSE
0	['RM' 'PTRATIO' 'LSTAT' 'Price']	3	0.5	0.488164	4.40443	41.678
1	['RM' 'LSTAT' 'Price']	2	0.6	0.540908	4.14244	37.3831
2	['LSTAT' 'Price']	1	0.7	0.430957	4.86401	46.3363
3	['RM' 'Price']	1	0.7	0.423944	4.32474	46.9074

Let's find out other combination of columns to get better accuracy > 40%

In [92]:

```
threshold = 0.40
corr_value = getCorrelatedFeature(corrmat['Price'], threshold)
corr_value
```

Out[92]:

	Corr Value
INDUS	-0.483725
NOX	-0.427321
RM	0.695360
TAX	-0.468536
PTRATIO	-0.507787
LSTAT	-0.737663
Price	1.000000

In [93]:

```
correlated_data = data[corr_value.index]
correlated_data.head()
```

Out[93]:

	INDUS	NOX	RM	TAX	PTRATIO	LSTAT	Price
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

In [94]:

```
y_predict = get_y_predict(correlated_data)
performance_metrics(correlated_data.columns.values, threshold, y_test, y_predict)
```

Out[94]:

	features name	#feature	corr_value	r2_score	MAE	MSE
0	['RM' 'PTRATIO' 'LSTAT' 'Price']	3	0.5	0.488164	4.40443	41.678
1	['RM' 'LSTAT' 'Price']	2	0.6	0.540908	4.14244	37.3831
2	['LSTAT' 'Price']	1	0.7	0.430957	4.86401	46.3363
3	['RM' 'Price']	1	0.7	0.423944	4.32474	46.9074
4	['INDUS' 'NOX' 'RM' 'TAX' 'PTRATIO' 'LSTAT' 'P...	6	0.4	0.476203	4.3945	42.6519

In [ ]:

```

```

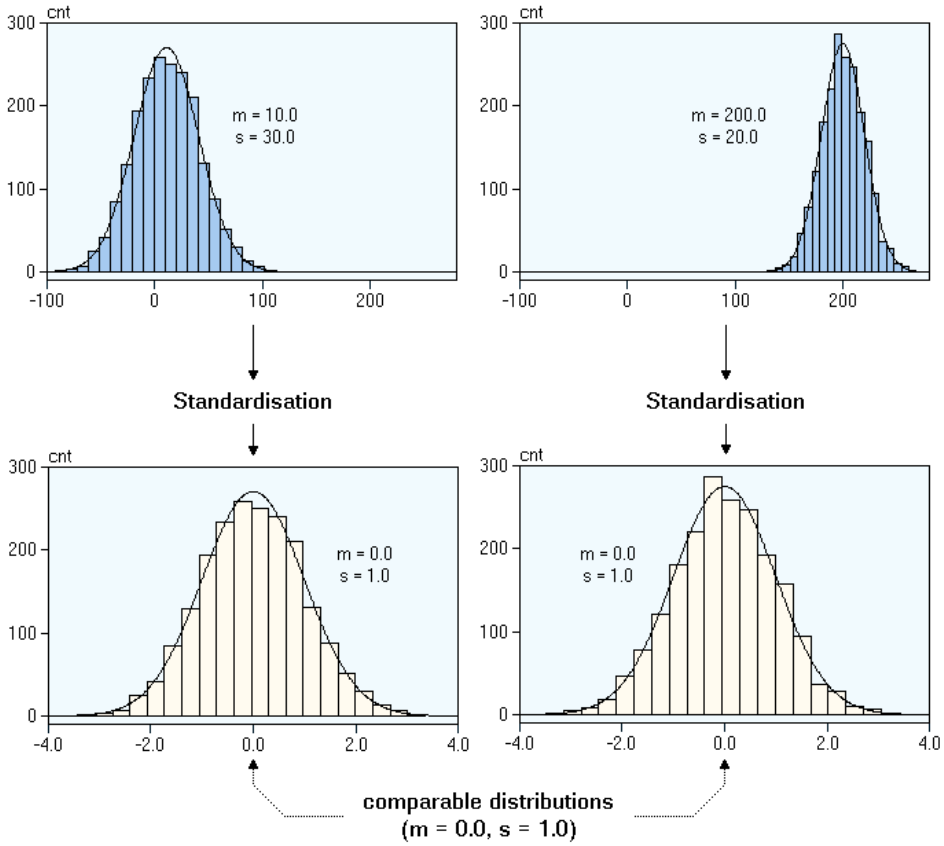
Now lets go ahead and understand what is Normalization and Standardization

In [ ]:

```

```

Standardization

Standardization of datasets is a common requirement for many machine learning estimators implemented in scikit-learn; they might behave badly if the individual features do not more or less look like standard normally distributed data: Gaussian with zero mean and unit variance.



Normalization

Normalization is the process of scaling individual samples to have unit norm. This process can be useful if you plan to use a quadratic form such as the dot-product or any other kernel to quantify the similarity of any pair of samples.

This assumption is the base of the Vector Space Model often used in text classification and clustering contexts.

Name	Sklearn_Class
StandardScaler	StandardScaler
MinMaxScaler	MinMaxScaler
MaxAbsScaler	MaxAbsScaler
RobustScaler	RobustScaler
QuantileTransformer-Normal	QuantileTransformer(output_distribution='normal')
QuantileTransformer-Uniform	QuantileTransformer(output_distribution='uniform')
PowerTransformer-Yeo-Johnson	PowerTransformer(method='yeo-johnson')
Normalizer	Normalizer

In [97]:

```
model = LinearRegression(normalize=True)
model.fit(X_train, y_train)
```

Out[97]:

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=True)
```

In [99]:

```
y_predict = model.predict(X_test)
r2_score(y_test, y_predict)
```

Out[99]:

```
0.4881642015692508
```

In [ ]:

```
```

In [ ]:

```
```

Defining performance metrics

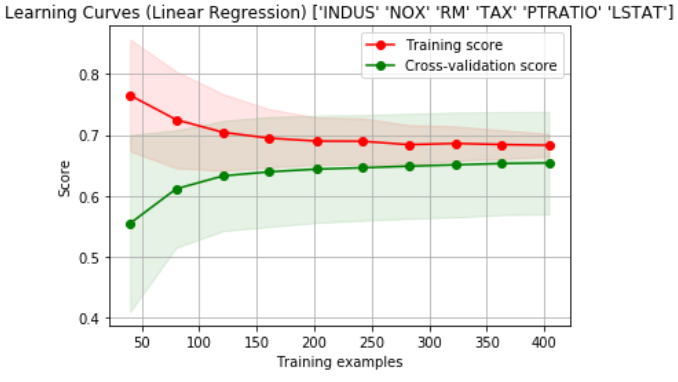
Plotting Learning Curves

In [103]:

```
from sklearn.model_selection import learning_curve, ShuffleSplit
```

In [104]:

```
def plot_learning_curve(estimator, title, X, y, ylim=None, cv=None,
                        n_jobs=None, train_sizes=np.linspace(.1, 1.0, 10)):

    plt.figure()
    plt.title(title)
    plt.xlabel("Training examples")
    plt.ylabel("Score")

    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.grid()

    plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="r")
    plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="g")

    plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Training score")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Cross-validation score")

    plt.legend(loc="best")
    return plt

X = correlated_data.drop(labels = ['Price'], axis = 1)
y = correlated_data['Price']

title = "Learning Curves (Linear Regression) " + str(X.columns.values)

cv = ShuffleSplit(n_splits=100, test_size=0.2, random_state=0)

estimator = LinearRegression()
plot_learning_curve(estimator, title, X, y, ylim=(0.7, 1.01), cv=cv, n_jobs=-1)

plt.show()
```



In [ ]:

```
```