Machine Learning

Linear Regression

Quan Minh Phan & Ngoc Hoang Luong

University of Information Technology

-

Vietnam National University Ho Chi Minh City

October 7, 2022

New Packages

- numpy → very frequently used in ML (python)

  Link: https://numpy.org/doc/stable/user/index.html#user

```
>> import numpy as np
```

- matplotlib → for visualization

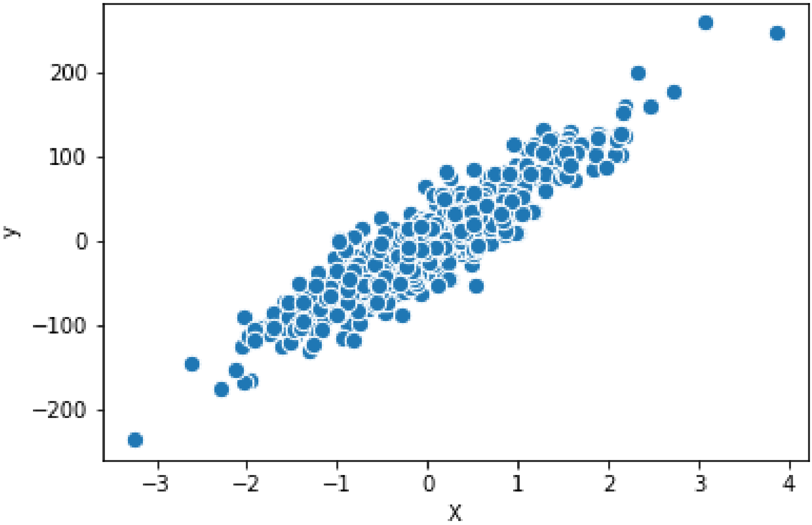
  Link: https://matplotlib.org/stable/tutorials/index.html

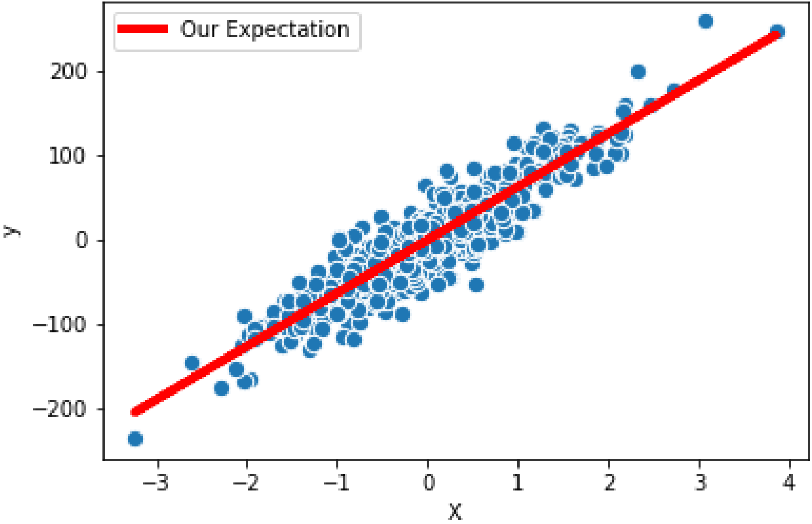
```
>> import matplotlib.pyplot as plt
```

Generate A Regression Problem

```
>> from sklearn.datasets import make_regression
>> X, y = make_regression(n_samples=500, n_features=1,
   n_informative=1, noise=25, random_state=42)
```

```
>> plt.scatter(X, y, facecolor='tab:blue', edgecolor='white', s=70)
   plt.xlabel('X')
   plt.ylabel('y')
   plt.show()
```


200
100
0
−100
−200
y
−3
−2
−1
0
1
2
3
4
X


Our Expectation
200
100
0
−100
−200
y
−3
−2
−1
0
1
2
3
4
X

Recall (Linear Regression)

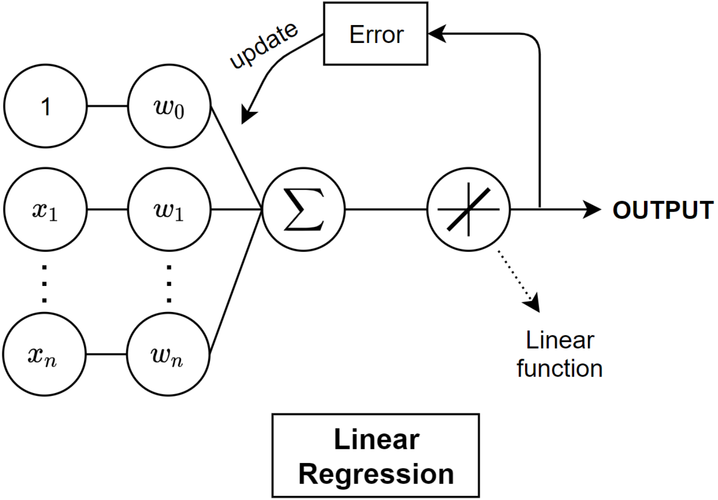


Figure: The general concept of Linear Regression

Minimizing cost function with gradient descent

Cost function (Squared Error):

$$J(w) = \frac{1}{2}\sum_i (y^{(i)} - \hat{y}^{(i)})^2 \tag{1}$$

Update the weights:

$$w_{t+1} := w_t + \Delta w \tag{2}$$

$$\Delta w = -\eta \nabla J(w) \tag{3}$$

$$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)} \tag{4}$$

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)} \tag{5}$$

Minimizing cost function with gradient descent (cont.)

$$w_j = \begin{cases} w_j + \eta * sum(y - \hat{y}) & j = 0 \\ w_j + \eta * \sum_i (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)} & j \in [1, \dots, n] \end{cases}$$

Pseudocode of the Training Process

Algorithm 1 Gradient Descent

```
Initialize the weights, w
while Stopping Criteria is not satisfied do
    Compute the output value, ŷ
    Updates the weights
        Compute the difference between y and ŷ
        Update the intercept
        Update the coefficients
end while
```

Components

Hyperparameters

- eta (float): the initial learning rate
- max_iter (int): the maximum number of iterations
- random_state (int)

Parameters

- w (list/array): the weight values
- costs (list/array): the list containing the cost values over iterations

Methods

- fit(X, y)
- predict(X)

Implement (code from scratch)

```
class LinearRegression_GD:
    def __init__(self, eta = 0.001, max_iter = 20, random_state = 42):
        self.eta = eta
        self.max_iter = max_iter
        self.random_state = random_state
        self.w = None
        self.costs = [ ]
```

```
    def predict(self, X):
        return np.dot(X, self.w[1:]) + self.w[0]
```

'fit' method

```
def fit(self, X, y):
    rgen = np.random.RandomState(self.random_state)
    self.w = rgen.normal(loc = 0.0, scale = 0.01, size = 1 + X.shape[1])
    self.costs = [ ]
    for n_iters in range(self.max_iter):
        y_pred = self.predict(X)
        diff = y - y_pred
        self.w[0] += self.eta * np.sum(diff)
        for j in range(X.shape[1]):            // j ← [0, 1, ..., X.shape[1]]
            delta = 0.0
            for i in range(X.shape[0]):        // i ← [0, 1, ..., X.shape[0]]
                delta += self.eta * diff[i] * X[i][j]
            self.w[j + 1] += delta
        cost = np.sum(diff ** 2) / 2
        self.costs.append(cost)
```

```
def fit(self, X, y):
    rgen = np.random.RandomState(self.random_state)
    self.w = rgen.normal(loc = 0.0, scale = 0.01, size = 1 + X.shape[1])
    self.costs = [ ]
    for n_iters in range (self.max_iter):
        y_pred = self.predict(X)
        diff = y - y_pred
        self.w[0] += self.eta * np.sum(diff)
        self.w[1:] += self.eta * np.dot(X.T, diff)
        cost = np.sum(diff ** 2) / 2
        self.costs.append(cost)
```

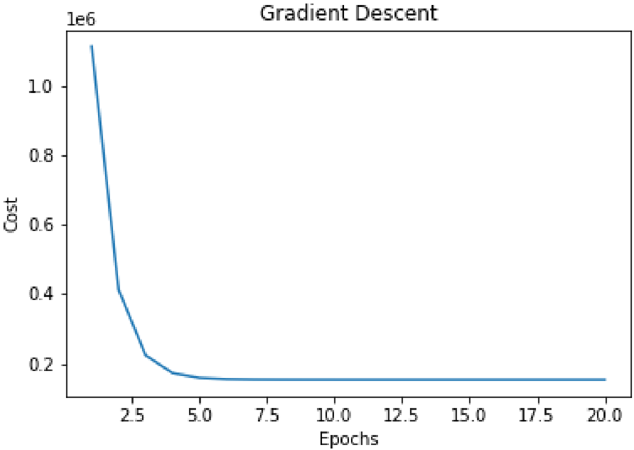
Train Model

Gradient Descent

```
>> reg_GD = LinearRegression_GD(eta=0.001, max_iter=20,
   random_state=42)
   reg_GD.fit(X, y)
```

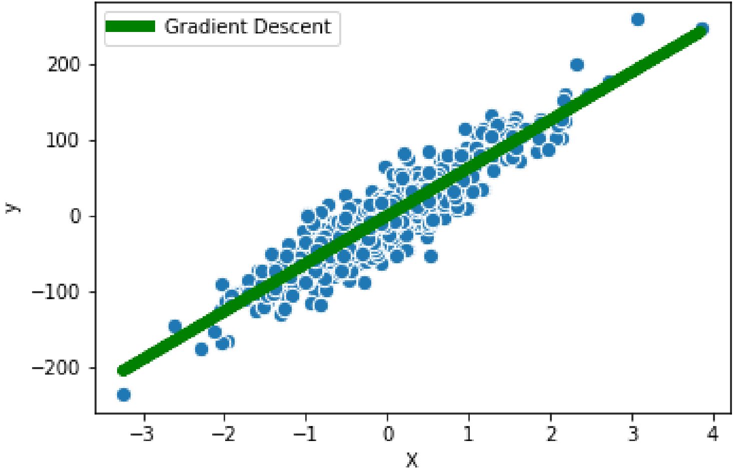
Visualize the trend in the cost values (Gradient Descent)

```
>> plt.plot(range(1, len(reg_GD.costs) + 1), reg_GD.costs)
   plt.xlabel('Epochs')
   plt.ylabel('Cost')
   plt.title('Gradient Descent')
   plt.show()
```


Gradient Descent
1e6
1.0
0.8
0.6
0.4
0.2
Cost
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs

Visualize on Data

```
>> plt.scatter(X, y, facecolor='tab:blue', edgecolor='white', s=70)
   plt.plot(X, reg_GD.predict(X), color='green', lw=6, label='Gradient
   Descent')
   plt.xlabel('X')
   plt.ylabel('y')
   plt.legend()
   plt.show()
```


Gradient Descent
200
100
0
−100
−200
y
−3
−2
−1
0
1
2
3
4
X

Weight values

```
>> w_GD = reg_GD.w
   w_GD
>> [-0.9794002,   63.18592509]
```

Implement (package)

Stochastic Gradient Descent

```
from sklearn.linear_model import SGDRegressor
```

Hyperparameters

- eta0
- max_iter
- random_state

Parameters

- intercept_
- coef_

Methods

- fit(X, y)
- predict(X)

Implement (package) (cont.)

Normal Equation

```python
from sklearn.linear_model import LinearRegression
```

Parameters

- intercept_
- coef_

Methods

- fit(X, y)
- predict(X)

Differences

Gradient Descent

- $w := w + \Delta w$
- $\Delta w = \eta \sum_i (y^{(i)} - \hat{y}^{(i)}) x^i$

Stochastic Gradient Descent

- $w := w + \Delta w$
- $\Delta w = \eta (y^{(i)} - \hat{y}^{(i)}) x^i$

Normal Equation

- $w = (X^T X)^{-1} \, X^T y$

Practice (cont.)

Stochastic Gradient Descent

```
>> from sklearn.linear_model import SGDRegressor
>> reg_SGD = SGDRegressor(eta0=0.001, max_iter=20,
   random_state=42, learning_rate='constant')
   reg_SGD.fit(X, y)
```

Normal Equation

```
>> from sklearn.linear_model import LinearRegression
>> reg_NE = LinearRegression()
   reg_NE.fit(X, y)
```

Weight Values Comparisons

Gradient Descent (ours)

```
>> w_GD = reg_GD.w
   w_GD
>> [-0.9794002,   63.18592509]
```

Stochastic Gradient Descent

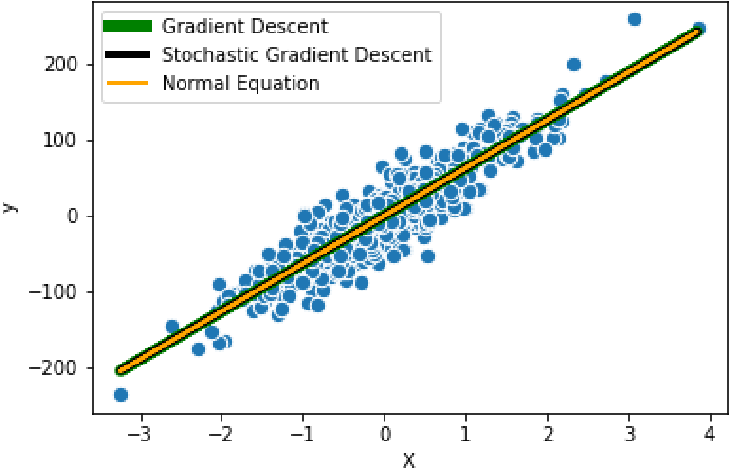
```
>> w_SGD = np.append(reg_SGD.intercept_, reg_SGD.coef_)
   w_SGD
>> [-1.02681553,   63.08630288]
```

Normal Equation

```
>> w_NE = np.append(reg_NE.intercept_, reg_NE.coef_)
   w_NE
>> [-0.97941333,   63.18605572]
```

Visualize on Data (all)

```
>> plt.scatter(X, y, facecolor='tab:blue', edgecolor='white', s=70)
   plt.plot(X, reg_GD.predict(X), color='green', lw=6, label='Gradient
   Descent')
   plt.plot(X, reg_SGD.predict(X), color='black', lw=4,
   label='Stochastic Gradient Descent')
   plt.plot(X, reg_NE.predict(X), color='orange', lw=2, label='Normal
   Equation')
   plt.xlabel('X')
   plt.ylabel('y')
   plt.legend()
   plt.show()
```



Gradient Descent
Stochastic Gradient Descent
Normal Equation
200
100
0
−100
−200
y
−3
−2
−1
0
1
2
3
4
X

Performance Evaluation

Mean Absolute Error (MAE)

$$MAE(y, \hat{y}) = \frac{1}{n} \sum_{i} |y^{(i)} - \hat{y}^{(i)}| \tag{6}$$

Mean Squared Error (MSE)

$$MSE(y, \hat{y}) = \frac{1}{n} \sum_{i} (y^{(i)} - \hat{y}^{(i)})^2 \tag{7}$$

R-Squared (R2)

$$R^2(y, \hat{y}) = 1 - \frac{\sum_i (y^{(i)} - \hat{y}^{(i)})^2}{\sum_i (y^{(i)} - \overline{y})^2} \tag{8}$$

Performance Evaluation

```
>> from sklearn.metrics import mean_absolute_error as MAE
   from sklearn.metrics import mean_squared_error as MSE
   from sklearn.metrics import r2_score as R2
```

```
>> y_pred_GD = reg_GD.predict(X)
```

```
>> y_pred_SGD = reg_SGD.predict(X)
```

```
>> y_pred_NE = reg_NE.predict(X)
```

Performance Evaluation (cont.)

Mean Absolute Error

```
>> print('MAE of GD:', round(MAE(y, y_pred_GD), 6))
   print('MAE of SGD:', round(MAE(y, y_pred_SGD), 6))
   print('MAE of NE:', round(MAE(y, y_pred_NE), 6))
```

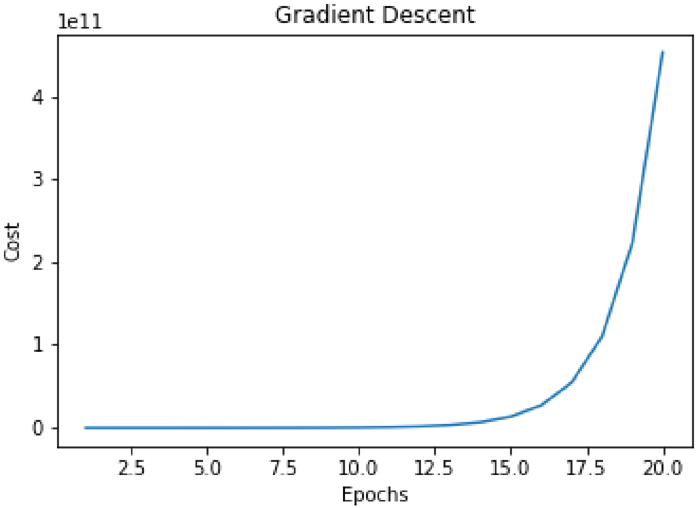
Mean Squared Error

```
>> print('MSE of GD:', round(MSE(y, y_pred_GD), 6))
   print('MSE of SGD:', round(MSE(y, y_pred_SGD), 6))
   print('MSE of NE:', round(MSE(y, y_pred_NE), 6))
```

R^2 score

```
>> print('R2 of GD:', round(R2(y, y_pred_GD), 6))
   print('R2 of SGD:', round(R2(y, y_pred_SGD), 6))
   print('R2 of NE:', round(R2(y, y_pred_NE), 6))
```

Run Gradient Descent with lr = 0.005



Polynominal Regression

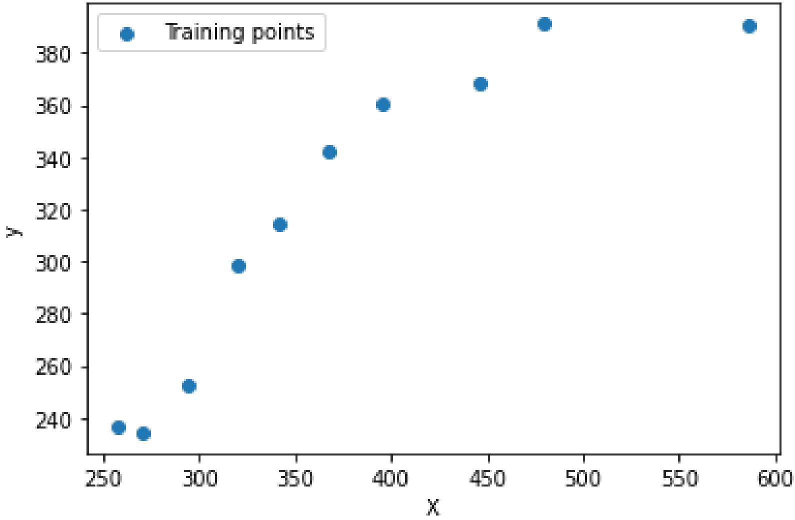
Example

X = [258.0, 270.0, 294.0, 320.0, 342.0, 368.0, 396.0, 446.0, 480.0, 586.0]
y = [236.4, 234.4, 252.8, 298.6, 314.2, 342.2, 360.8, 368.0, 391.2, 390.8]

```
>> X = np.array([258.0, 270.0, 294.0, 320.0, 342.0, 368.0, 396.0, 446.0,
   480.0, 586.0])[:, np.newaxis]
   y = np.array([236.4, 234.4, 252.8, 298.6, 314.2, 342.2, 360.8, 368.0,
   391.2, 390.8])
```

```
>> plt.scatter(X, y, label='Training points')
   plt.xlabel('X')
   plt.ylabel('y')
   plt.legend()
   plt.show()
```

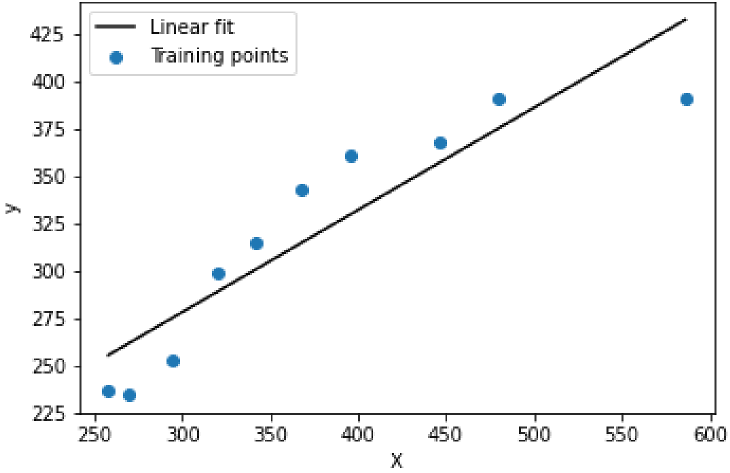
Visualize data



Experiment with Linear Regression

```
>> from sklearn.linear_model import LinearRegression
   lr = LinearRegression()
   lr.fit(X, y)
```

Experiment with Linear Regression (cont.)



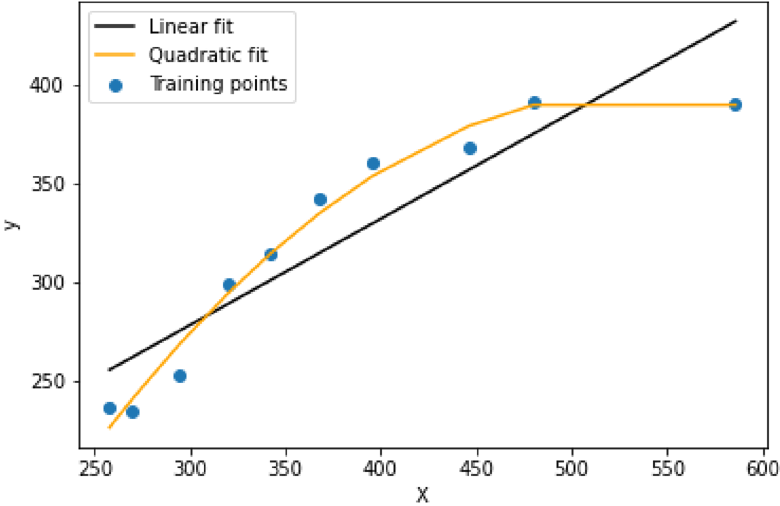
Experiment with Polynominal Regression

Syntax

```
from sklearn.preprocessing import PolynomialFeatures
```

```
>> from sklearn.preprocessing import PolynomialFeatures
   quadratic = PolynomialFeatures(degree=2)
   X_quad = quadratic.fit_transform(X)
   pr = LinearRegression()
   pr.fit(X_quad, y)
```

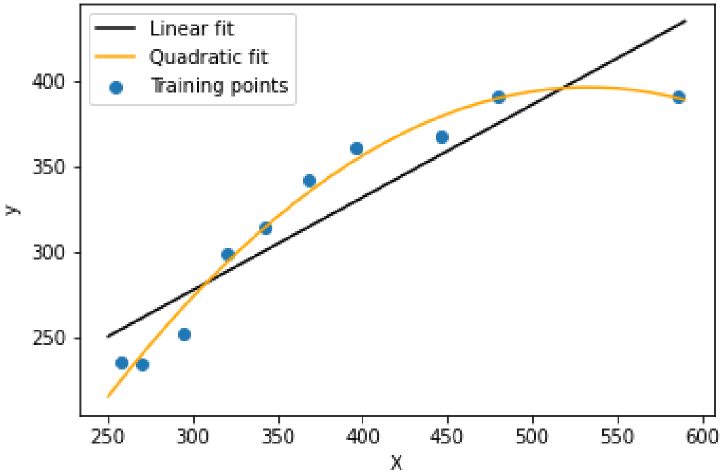
Experiment with Polynominal Regression (cont.)



```
>> X_test = np.arange(250, 600, 10)[:, np.newaxis]
```

```
>> y_pred_linear = lr.predict(X_test)
   y_pred_quad = pr.predict(quadratic.fit_transform(X_test))
```

```
>> plt.scatter(X, y, label='Training points')
   plt.xlabel('X')
   plt.ylabel('y')
   plt.plot(X_test, y_pred_linear, label='Linear fit', c='black')
   plt.plot(X_test, y_pred_quad, label='Quadratic fit', c='orange')
   plt.legend()
   plt.show()
```


Linear fit
Quadratic fit
Training points
400
350
300
250
y
250
300
350
400
450
500
550
600
X

Practice

- Dataset: 'Boston Housing' (housing.csv) (14 attributes: 13 independent variables + 1 target variable)

  - 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

- File: boston_housing.iypnb