텐서플로우를 이용한 선형회귀 이해

[9주/1차시 학습내용]: 텐서플로우의 추상화를 통해 선형회귀를 이해한다

국민대학교 소프트웨어학부

손실함수 이해하기

기울기와 손실함수값에 대한 데이터 시각화를
통해 손실함수를 이해한다

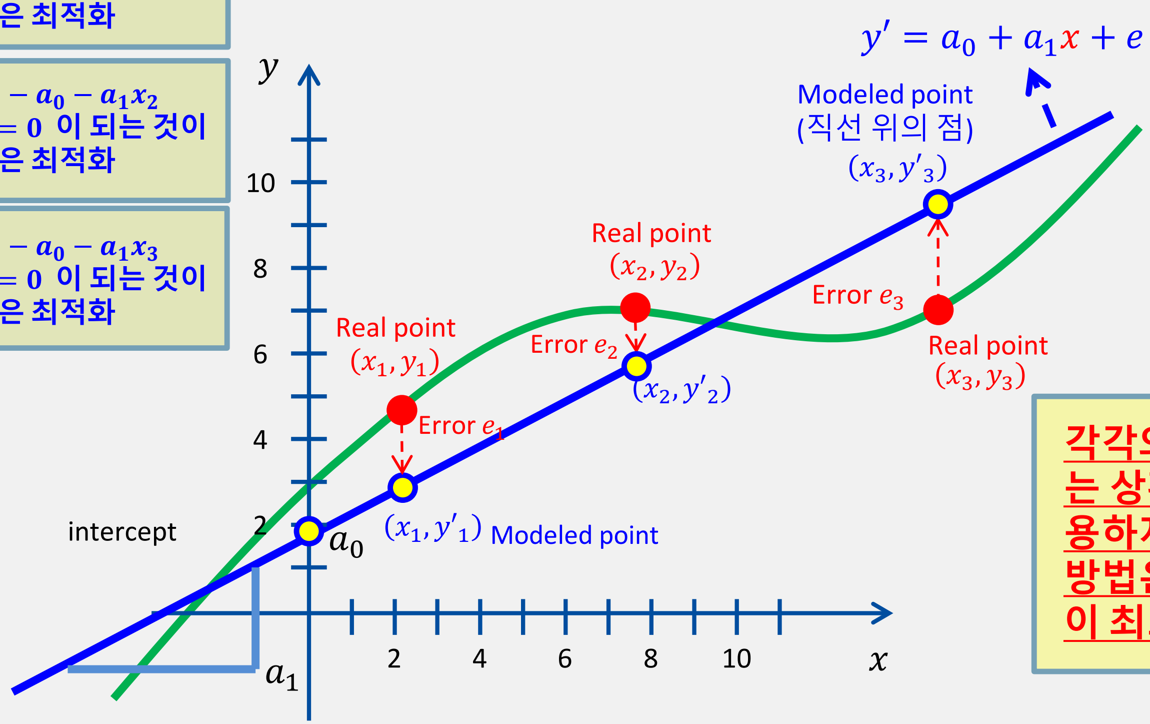
국민대학교 소프트웨어학부

Minimize the sum of errors (차선책)

$e_1 = y_1 - a_0 - a_1x_1$
즉, $e_1 = 0$ 이 되는 것이 제일 좋은 최적화

$e_2 = y_2 - a_0 - a_1x_2$
즉, $e_2 = 0$ 이 되는 것이 제일 좋은 최적화

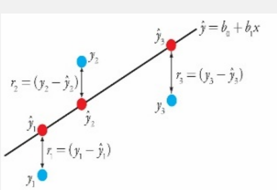
$e_3 = y_3 - a_0 - a_1x_3$
즉, $e_3 = 0$ 이 되는 것이 제일 좋은 최적화



$$e_1 = y_1 - a_0 - a_1x_1$$
$$e_2 = y_2 - a_0 - a_1x_2$$
$$e_3 = y_3 - a_0 - a_1x_3$$

<u>각각의 모든 요구 조건을 수용해야 하는 상황이지만, 그렇게 하지 못한다. 수용하지 못하는 상황에서 최선을 다하는 방법은 조금씩 양보를 해서 에러의 합이 최소화되도록 하는 것이다.</u>

Y 축 상의 에러



$$minimize\ the\ sum\ of\ error\ (e_1 + e_2 + \cdots + e_n) = min \sum_{i=1}^{n} e_i$$
$$= min \sum_{i=1}^{n} (y_i - a_0 - a_1x_i)$$

선형 회귀 (X와 Y는 실제점)

- hypothesis는 곡선접합에 의해 기울기를 가지는 직선
- hypothesis-Y는 모델점과 실제점 사이의 오차($\boldsymbol{e_1}, \boldsymbol{e_2}$, … , $\boldsymbol{e_n}$)
- cost는 오차(hypothesis-Y)를 제곱 한 값, 최소 제곱 회귀

```
import tensorflow as tf
import matplotlib.pyplot as plt

X = [1., 2., 3.]
Y = [1., 2., 3.]
m = len(X)
W = tf.placeholder(tf.float32)

#hypothesis = tf.mul(W, X)
hypothesis = W*X
cost = tf.reduce_sum(tf.pow(hypothesis-Y, 2)) / m

init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/loss_function.py

최소 제곱 회귀 : W_val와 cost의 관계는?

- 기울기(W_val) 값에 따라 오차(hypothesis-Y)를 제곱 한 값(cost)의 값이 어떤 모양으로 바뀌는 지에 대해 알아보고자 함

```
# 그래프로 표시하기 위해 데이터를 누적할 리스트
W_val, cost_val = [], []

# 0.1 단위로 증가할 수 없어서 -30부터 시작. 그래프에는 -3에서 5까지 표시됨.
for i in range(-30, 50):
    xPos = i*0.1                                   # x 좌표. -3에서 5까지 0.1씩 증가
    yPos = sess.run(cost, feed_dict={W: xPos})     # x 좌표에 따른 y 값
    print('{:3.1f}, {:3.1f}'.format(xPos, yPos))
    # 그래프에 표시할 데이터 누적. 단순히 리스트에 갯수를 늘려나감
    W_val.append(xPos)
    cost_val.append(yPos)
sess.close()
```

GUI 디버깅

```
#hypothesis = tf.mul(W, X)
hypothesis = W*X   hypothesis: Tensor("mul:0", dtype=float32)
cost = tf.reduce_sum(tf.pow(hypothesis-Y, 2)) / m   cost: Tensor("truediv:0", shape=(), dtype=float32

init = tf.initialize_all_variables()   init: name: "init"₩nop: "NoOp"₩n
sess = tf.Session()   sess: <tensorflow.python.client.session.Session object at 0x00000271564CB940>
sess.run(init)

# 그래프로 표시하기 위해 데이터를 누적할 리스트
W_val, cost_val = [], []

# 0.1 단위로 증가할 수 없어서 -30부터 시작. 그래프에는 -3에서 5까지 표시됨.
```

Variables

```
Special Variables
W = {Tensor} Tensor("Placeholder:0", dtype=float32)
X = {list} <class 'list'>: [1.0, 2.0, 3.0]
Y = {list} <class 'list'>: [1.0, 2.0, 3.0]
cost = {Tensor} Tensor("truediv:0", shape=(), dtype=float32)
hypothesis = {Tensor} Tensor("mul:0", dtype=float32)
init = {Operation} name: "init"₩nop: "NoOp"₩n
m = {int} 3
sess = {Session} <tensorflow.python.client.session.Session object at 0x00000271564CB940>
```

GUI 디버깅

- xPos값이 -3일 때, yPos 값이 74.66667이 나오게 된다

```python
# 그래프로 표시하기 위해 데이터를 누적할 리스트
W_val, cost_val = [], []  W_val: <class 'list'>: []  cost_val: <class 'list'>: []

# 0.1 단위로 증가할 수 없어서 -30부터 시작. 그래프에는 -3에서 5까지 표시됨.
for i in range(-30, 50):  i: -30
    xPos = i*0.1                                # x 좌표. -3에서 5까지 0.1씩 증가  xPos: -3.0
    yPos = sess.run(cost, feed_dict={W: xPos})  # x 좌표에 따른 y 값  yPos: 74.66667
    print('{:3.1f}, {:3.1f}'.format(xPos, yPos))
    # 그래프에 표시할 데이터 누적. 단순히 리스트에 갯수를 늘려나감
    W_val.append(xPos)
    cost_val.append(yPos)
```

for i in range(-30, 50)

Variables

```
Special Variables
W = {Tensor} Tensor("Placeholder:0", dtype=float32)
W_val = {list} <class 'list'>: []
X = {list} <class 'list'>: [1.0, 2.0, 3.0]
Y = {list} <class 'list'>: [1.0, 2.0, 3.0]
cost = {Tensor} Tensor("truediv:0", shape=(), dtype=float32)
cost_val = {list} <class 'list'>: []
hypothesis = {Tensor} Tensor("mul:0", dtype=float32)
i = {int} -30
init = {Operation} name: "init"₩nop: "NoOp"₩n
m = {int} 3
sess = {Session} <tensorflow.python.client.session.Session object at 0x00000271564CB940>
xPos = {float} -3.0
yPos = {float32} 74.66667
```

GUI 디버깅

- 코드에서 hypothesis = W*X 이다.
- 코드에서 cost = tf.reduce_sum(tf.pow(hypothesis-Y, 2)) / m 로 정의되었다.

```python
import tensorflow as tf
import matplotlib.pyplot as plt

X = [1., 2., 3.]
Y = [1., 2., 3.]
m = len(X)
W = tf.placeholder(tf.float32)

#hypothesis = tf.mul(W, X)
hypothesis = W*X
cost = tf.reduce_sum(tf.pow(hypothesis-Y, 2)) / m

init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

- 코드에서 i=-30일 때,
- xPos = i*0.1 에 의해 xPos = -3이 된다
- yPos = sess.run(cost, feed_dict={W: xPos})에 의해, W 키에 xPos (-3)의 값이 할당된다.
- cost = tf.reduce_sum(tf.pow(W*X-Y, 2)) / m 안의 W =-3(xPos) 이 된다.
-

```python
# 그래프로 표시하기 위해 데이터를 누적할 리스트
W_val, cost_val = [], []

# 0.1 단위로 증가할 수 없어서 -30부터 시작. 그래프에는 -3에서 5까지 표시됨.
for i in range(-30, 50):
    xPos = i*0.1                                    # x 좌표. -3에서 5까지 0.1씩 증가
    yPos = sess.run(cost, feed_dict={W: xPos})      # x 좌표에 따른 y 값
    print('{:3.1f}, {:3.1f}'.format(xPos, yPos))
    # 그래프에 표시할 데이터 누적. 단순히 리스트에 갯수를 늘려나감
    W_val.append(xPos)
    cost_val.append(yPos)
sess.close()
```

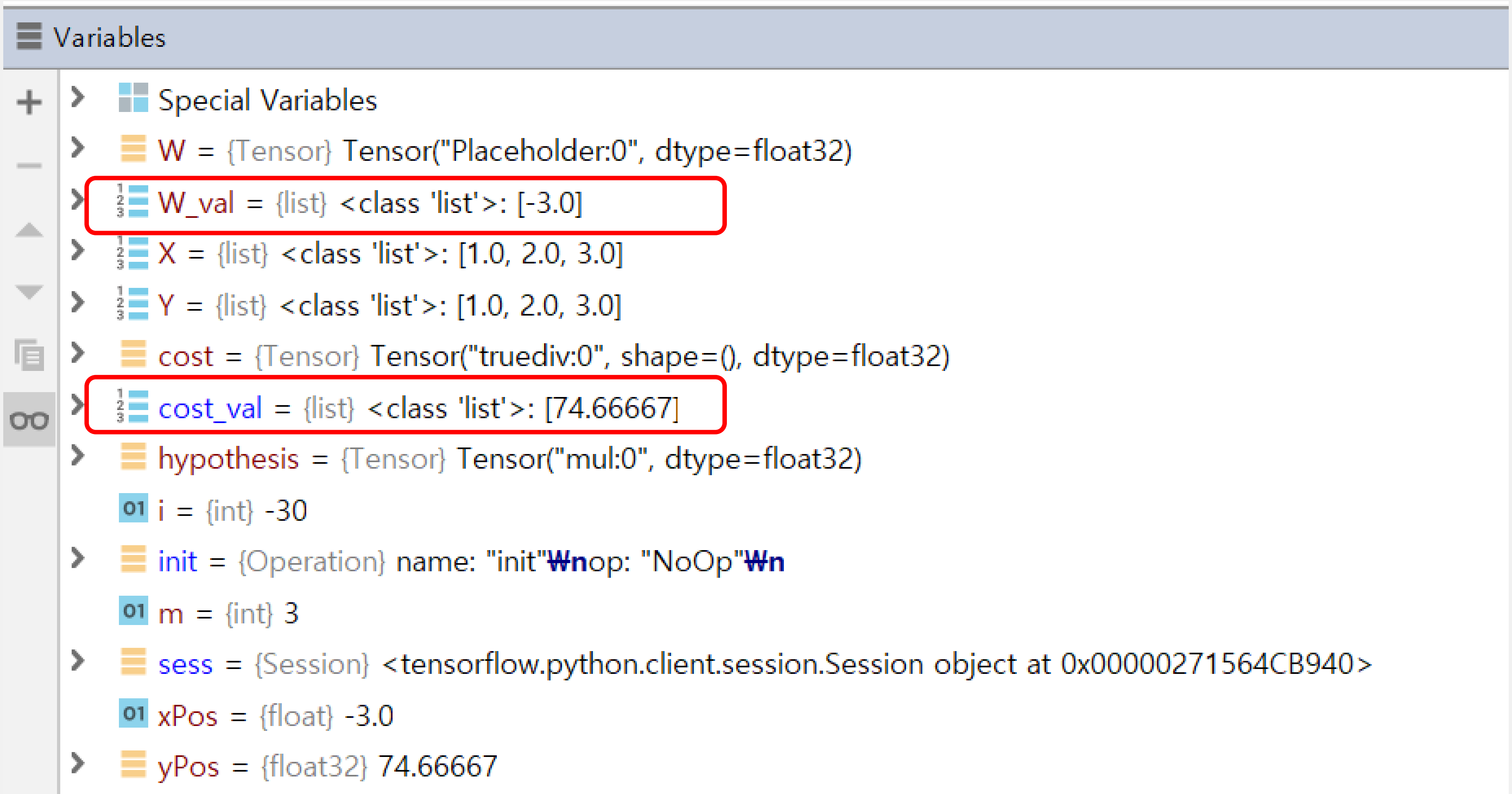
디버깅 (Hypothesis를 W*X로 대체해 본다)

- cost = tf.reduce_sum(tf.pow(hypothesis-Y, 2)) / m
- Hypothesis 가 W*X로 정의 되어 있다.
- i=-30일 때, xPos = i*0.1= -30*0.1=-3 에 의해 xPos = -3이 된다
- yPos = sess.run(cost, feed_dict={W: xPos})에 의해, W 키에 xPos =-3의 값이 할당된다.
- cost = tf.reduce_sum(tf.pow(W*X-Y, 2)) / m 안의 W =-3(xPos) 이 된다.

디버깅 (실제로 W*X-Y 를 계산해본다)

- W=-3일 때, 실제로 W*X-Y 를 계산해본다
- X = [1., 2., 3.], Y = [1., 2., 3.] 로 주어져 있다.
- 실제로 W*X-Y 를 계산해 보면,
  – W*X-Y= -3 *[1., 2., 3.] - [1., 2., 3.]=[-3, -6, -9] - [1,2,3]=[-4,-8,-12] 이 된다.
- 이 값을 tf.pow(W*X-Y, 2)에 대입한다.
  – tf.pow(W*X-Y, 2) = tf.pow([-4,-8,-12], 2) = (-4)^2+(-8)^2+(12)^2=224 값이 나오게 된다.
- tf.pow(W*X-Y, 2)/m 을 계산한다.
  – tf.pow(W*X-Y, 2)/3=224 /3= 74.66
- 기울기가 W=-3일 때, 손실값이 74.66 이 나오게 된다.

기울기가 W=-3일 때, 손실값이 74.66 확인



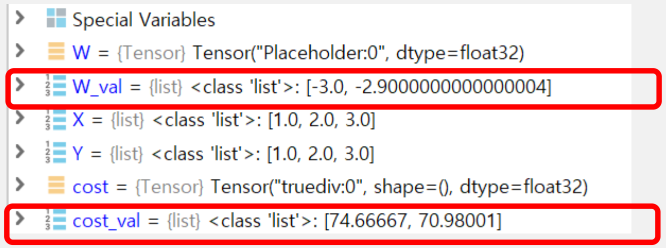
기울기가 W=-2.9일 때, 손실값 확인

- i=-29일 때, xPos = i*0.1= -29*0.1=-2.9 에 의해 xPos = -2.9 가 된다
- yPos = sess.run(cost, feed_dict={W: xPos})에 의해, W 키에 xPos =-2.9의 값이 할당된다.
- cost = tf.reduce_sum(tf.pow(W*X-Y, 2)) / m 안의 W =-2.9(xPos) 이 된다.

```
# 0.1 단위로 증가할 수 없어서 -30부터 시작. 그래프에는 -3에서 5까지 표시됨.
for i in range(-30, 50):   i: -29
    xPos = i*0.1                                 # x 좌표. -3에서 5까지 0.1씩 증가   xPos: -2.9000
    yPos = sess.run(cost, feed_dict={W: xPos})   # x 좌표에 따른 y 값   yPos: 70.98001
    print('{:3.1f}, {:3.1f}'.format(xPos, yPos))
```

기울기가 W=-2.9일 때, 손실값 70.98 확인

- i=-29 → xPos=-2.9일 때, 실제로 W*X-Y 를 계산해본다
- X = [1., 2., 3.], Y = [1., 2., 3.] 로 주어져 있다.
- 실제로 W*X-Y 를 계산해 보면,
  - W*X-Y= -2.9 *[1., 2., 3.] - [1., 2., 3.]=[-2.9, -5.8, -8.7] - [1,2,3]=[**-3.9,-7.8,-11.7**] 이 된다.
- 이 값을 tf.pow(W*X-Y, 2)에 대입한다.
  - tf.pow(W*X-Y, 2) = tf.pow([**-3.9,-7.8,-11.7**], 2) = (**-3.9**)^2+(**-7.8**)^2+(**-11.7**)^2=**212.94** 값이 나오게 된다.
- tf.pow(W*X-Y, 2)/m 을 계산한다.
  - tf.pow(W*X-Y, 2)/3= **212.94** /3= **70.98**
- 기울기가 W=-2.9일 때, 손실값이 70.98 이 나오게 된다.



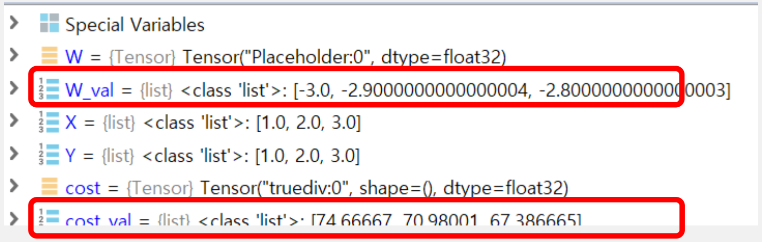
기울기가 W=-2.8일 때, 손실값 확인

- i=-28일 때, xPos = i*0.1= -28*0.1=-2.8 에 의해 xPos = -2.8 가 된다
- yPos = sess.run(cost, feed_dict={W: xPos})에 의해, W 키에 xPos =-2.8의 값이 할당된다.
- cost = tf.reduce_sum(tf.pow(W*X-Y, 2)) / m 안의 W =-2.8(xPos) 이 된다.

```
# 0.1 단위로 증가할 수 없어서 -30부터 시작. 그래프에는 -3에서 5까지 표시됨.
for i in range(-30, 50):   i: -28
    xPos = i*0.1                                # x 좌표. -3에서 5까지 0.1씩 증가   xPos: -2.80
    yPos = sess.run(cost, feed_dict={W: xPos})  # x 좌표에 따른 y 값   yPos: 67.386665
    print('{:3.1f}, {:3.1f}'.format(xPos, yPos))
```

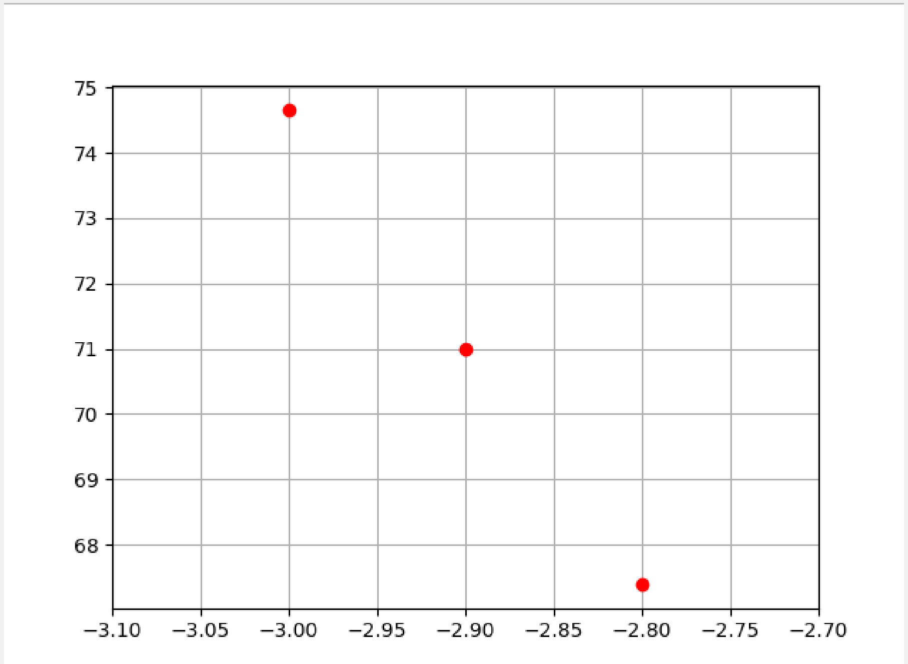
기울기가 W=-2.8일 때, 손실값 67.39 확인

- i=-28 → xPos=-2.8일 때, 실제로 W*X-Y 를 계산해본다
- X = [1., 2., 3.], Y = [1., 2., 3.] 로 주어져 있다.
- 실제로 W*X-Y 를 계산해 보면,
  - W*X-Y= -2.8 *[1., 2., 3.] - [1., 2., 3.]=[-2.8, -5.6, -8.4] - [1,2,3]=[**-3.8,-7.6,-11.4**] 이 된다.
- 이 값을 tf.pow(W*X-Y, 2)에 대입한다.
  - tf.pow(W*X-Y, 2) = tf.pow([]=[**-3.8,-7.6,-11.4**], 2) = (**-3.8**)^2+(**-7.6**)^2+(**-11.4**)^2=**202.16** 값이 나오게 된다.
- tf.pow(W*X-Y, 2)/m 을 계산한다.
  - tf.pow(W*X-Y, 2)/3= **202.16** /3= **67.39**
- 기울기가 W=-2.8일 때, 손실값이 67.39 가 나오게 된다.



기울기와 손실함수값에 대한 데이터 시각화

- 기울기(W_val)가 W=-3일 때, 손실값(cost_val) 74.66 확인
- 기울기(W_val)가 W=-2.9일 때, 손실값(cost_val) 70.98 확인
- 기울기(W_val)가 W=-2.8일 때, 손실값(cost_val) 67.39 확인
- 기울기가 증가함에 따라, 손실값이 감소하는 것으로 파악된다



파이썬으로 손실값 곡선형태 그리기

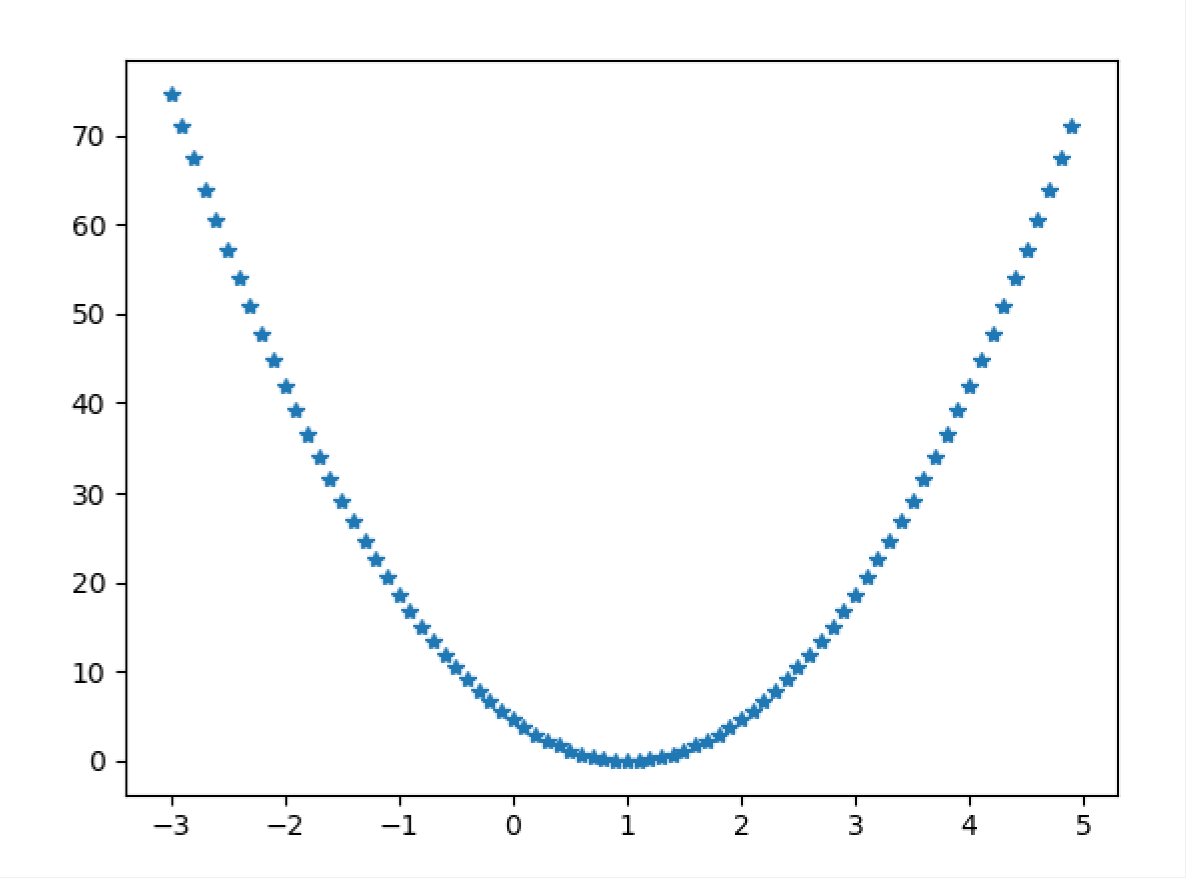
- 텐서플로우 대신 파이썬으로 계산할 수 있다.
- https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/linear_envelope.py

```
import numpy as np
import matplotlib.pyplot as plt

def linear_enveope(x,y):
    Cost=[]
    Slope=[]
    for i in np.arange(-3, 5, 0.1): #np.linspace(-3, 5, 80):
        Ct=(sum((i * x - y) ** 2)) / np.size(x)
        Cost.append(Ct)
        Slope.append(i)
    return Cost, Slope

def draw(Cost,Slope):
    plt.plot(Slope, Cost, '*')
    plt.show()
    plt.grid()
    plt.xlabel('Slope')
    plt.ylabel('Cost')
```

```
if __name__ == '__main__':
    x=np.array([1,2,3])
    y=np.array([1,2,3])
    Cost, Slope=linear_enveope(x,y)
    zipped=list(zip(Slope, Cost))
    draw(Cost, Slope)
```

파이썬으로 손실값 곡선형태 그리기

- 파이썬으로 손실값 곡선형태 그리기



파이썬으로 손실값 곡선형태 그리기

- 데이터 값을 확인해보면, 기울기가 -3에서 1로 증가하면서 손실값이 작아진다.

- [(-3.0, 74.66666666666667),
- (-2.9, 70.98),
- (-2.8, 67.38666666666666),
- (-2.6999999999999997, 63.886666666666656),

- (0.8000000000000034, 0.18666666666666035),
- (0.9000000000000035, 0.046666666666663435),

- 데이터 값을 확인해보면, 기울기가1에서 5로 증가하면서 손실값이 커진다

- (1.0000000000000036, 5.890161425650222e-29),
- (1.1000000000000032, 0.04666666666666965),
- (1.2000000000000037, 0.18666666666667364),

- (4.600000000000007, 60.48000000000224),
- (4.700000000000006, 63.88666666666688),
- (4.800000000000007, 67.3866666666669),
- (4.9000000000000075, 70.98000000000027)]

기울기와 손실함수값에 대한 데이터 시각화

```
# 0.1 단위로 증가할 수 없어서 -30부터 시작. 그래프에는 -3에서 5까지 표시됨.
for i in range(-30, 50):
    xPos = i*0.1                                    # x 좌표. -3에서 5까지 0.1씩 
증가
    yPos = sess.run(cost, feed_dict={W: xPos})      # x 좌표에 따른 y 값
    print('{:3.1f}, {:3.1f}'.format(xPos, yPos))
    # 그래프에 표시할 데이터 누적. 단순히 리스트에 갯수를 늘려나감
    W_val.append(xPos)
    cost_val.append(yPos)
sess.close()
# ------------------------------------------ #
print('size(W_val=)', np.size(W_val))
print('W_val=', W_val)
print('cost_val=', cost_val)
plt.plot(W_val, cost_val, 'ro')
plt.ylabel('Cost')
plt.xlabel('W')
plt.grid()
plt.show()
```

기울기와 손실함수값에 대한 데이터 시각화

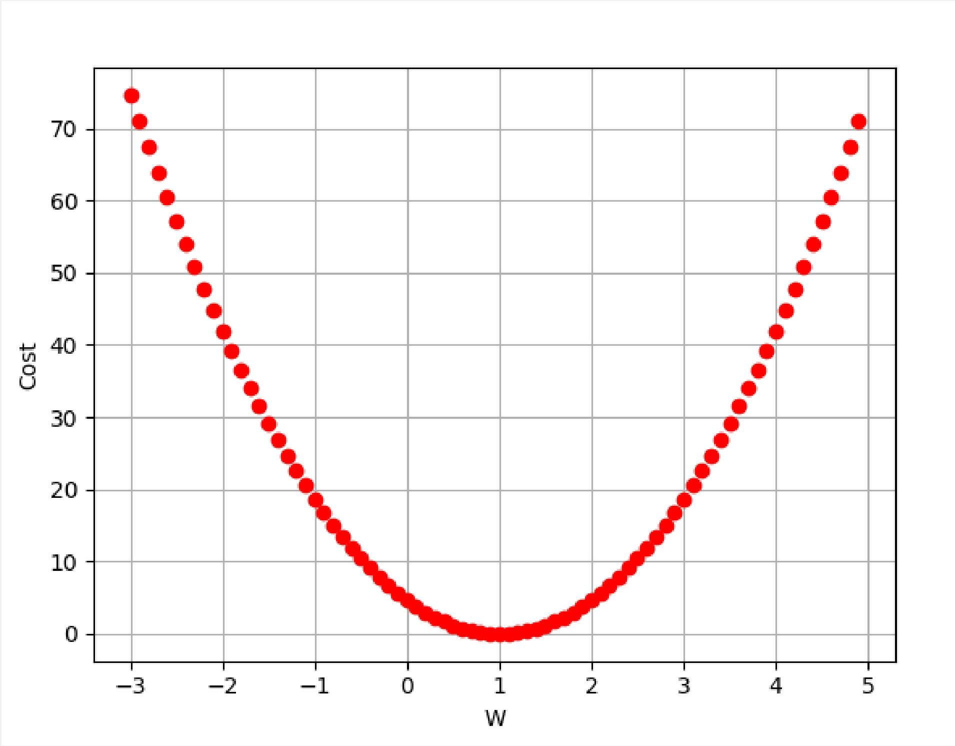
- size(W_val=) 80
- **W_val**= [-3.0, -2.9, -2.8, -2.7, -2.6, -2.5, -2.4, -2.3, -2.2, -2.1, -2.0, -1.9, -1.8, -1.7, -1.6, -1.5, -1.4, -1.3, -1.2, -1.1, -1.0, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0.0, **0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7**, **0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8,** 1.9, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9]

기울기와 손실함수값에 대한 데이터 시각화

- **cost_val**= [74.66667, 70.98001, 67.386665, 63.88667, 60.479992, 57.166668, 53.94668, 50.82, 47.786674, 44.84666, 42.0, 39.246666, 36.586662, 34.020004, 31.546667, 29.166668, 26.880001, 24.686666, 22.58667, 20.58, 18.666668, 16.846666, 15.120001, 13.486667, 11.946669, 10.5, 9.146666, 7.886667, 6.7200003, 5.6466665, 4.666667, **3.7800002, 2.9866672, 2.2866664, 1.6800001, 1.1666667, 0.7466666, 0.42000008, 0.18666664, 0.04666671, 0.0, 0.04666671, 0.18666676, 0.4199999, 0.74666655, 1.1666667, 1.6800003, 2.2866673, 2.9866662,** 3.7799995, 4.666667, 5.6466665, 6.720001, 7.8866653, 9.146668, 10.5, 11.946666, 13.48667, 15.119998, 16.84667, 18.666668, 20.579998, 22.58667, 24.686666, 26.880005, 29.166668, 31.546661, 34.020004, 36.586662, 39.246674, 42.0, 44.84666, 47.786663, 50.820007, 53.94668, 57.166668, 60.479992, 63.886658, 67.38667, 70.98001]

기울기와 손실함수값에 대한 데이터 시각화

- W_val가 1에 가까이 갈수록 cost_val 값이 0에 가까이 가는 것을 확인해 보자.
- W_val가 -3에서 5사이에 총 80개의 점이 있다. (-3, 5, 0.1)



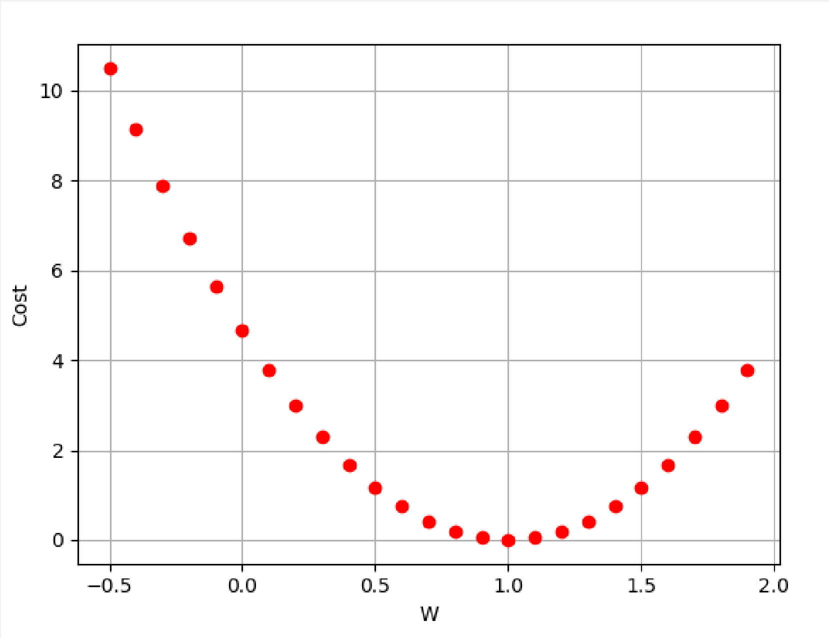
기울기 1과 손실 함수값 0 부분의 확대

- -5에서 20 사이, -0.5에서 2 사이 0.1씩 증가하면 총 25개의 점 확인

```
# -5에서 20 사이, -0.5에서 2 사이 0.1씩 증가하면 총 25개의 점 확인
for i in range(-5, 20): # -30, 50
    xPos = i*0.1                                    # x 좌표. -3에서 5까지 0.1씩 증가
    yPos = sess.run(cost, feed_dict={W: xPos})      # x 좌표에 따른 y 값
    print('{:3.1f}, {:3.1f}'.format(xPos, yPos))
    # 그래프에 표시할 데이터 누적. 단순히 리스트에 갯수를 늘려나감
    W_val.append(xPos)
    cost_val.append(yPos)
sess.close()
```

기울기 1과 손실함수값 0 부분의 확대

- **W_val**= array([-0.5, -0.4, -0.3, -0.2, -0.1, 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, **1.** , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
- **cost_val**= array([10.5, 9.1, 7.9, 6.7, 5.6, 4.7, 3.8, 2.9,   2.3, 1.7, 1.2, 0.7, 0.4, 0.19, 0.04, **0.** , 0.04, 0.19, 0.4 , 0.7])



np.where(W_val==1.)
(array([15], dtype=int64),)
np.where(cost_val==0.)
(array([15], dtype=int64),)

기울기와 손실함수값 관계

- 기울기가 1일 때, 손실함수값이 0을 가진다.
- X=[1,2,3], Y=[1,2,3]을 지나는 직선의 방정식은 즉, **hypothesis(가설, 또는 곡선접합, 선형회귀)**은 W*X의 형태를 가지게 되며, 이 때, W(기울기)가 1일 때, **가설(곡선접합, 선형회귀)** W*X와 실제값 Y의 에러 차이(손실값)는 0이다

선형 회귀 (X와 Y는 실제점)

- hypothesis는 곡선접합에 의해 기울기를 가지는 직선
- hypothesis-Y는 모델점과 실제점 사이의 오차(e_1, e_2 , … , e_n)
- cost는 오차(hypothesis-Y)를 제곱 한 값, 최소 제곱 회귀

```python
import tensorflow as tf
import matplotlib.pyplot as plt

X = [1., 2., 3.]
Y = [1., 2., 3.]
m = len(X)
W = tf.placeholder(tf.float32)

#hypothesis = tf.mul(W, X)
hypothesis = W*X
cost = tf.reduce_sum(tf.pow(hypothesis-Y, 2)) / m

init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/loss_function.py

GUI 디버깅

- 코드에서 hypothesis = W*X 이다.
- 코드에서 cost = tf.reduce_sum(tf.pow(hypothesis-Y, 2)) / m 로 정의되었다.

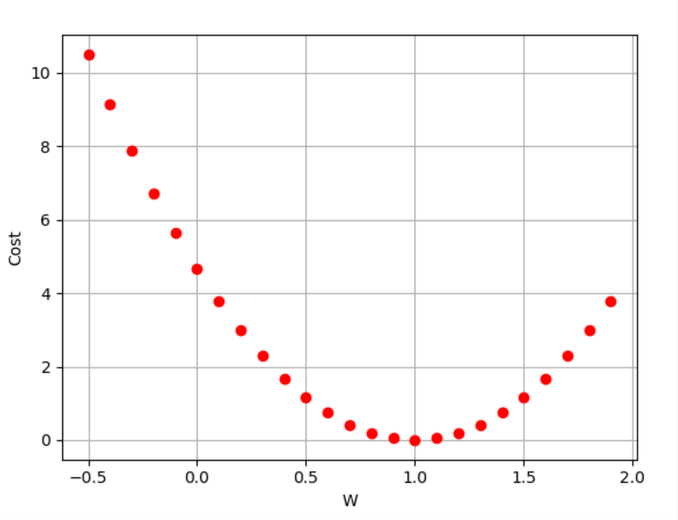
```python
import tensorflow as tf
import matplotlib.pyplot as plt

X = [1., 2., 3.]
Y = [1., 2., 3.]
m = len(X)
W = tf.placeholder(tf.float32)

#hypothesis = tf.mul(W, X)
hypothesis = W*X
cost = tf.reduce_sum(tf.pow(hypothesis-Y, 2)) / m

init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

- 코드에서 i=-30일 때,
- xPos = i*0.1 에 의해 xPos = -3이 된다
- yPos = sess.run(cost, feed_dict={W: xPos})에 의해, W 키에 xPos (-3)의 값이 할당된다.
- cost = tf.reduce_sum(tf.pow(W*X-Y, 2)) / m 안의 W =-3(xPos) 이 된다.
-

```python
# 그래프로 표시하기 위해 데이터를 누적할 리스트
W_val, cost_val = [], []

# 0.1 단위로 증가할 수 없어서 -30부터 시작. 그래프에는 -3에서 5까지 표시됨.
for i in range(-30, 50):
    xPos = i*0.1                                    # x 좌표. -3에서 5까지 0.1씩 증가
    yPos = sess.run(cost, feed_dict={W: xPos})      # x 좌표에 따른 y 값
    print('{:3.1f}, {:3.1f}'.format(xPos, yPos))
    # 그래프에 표시할 데이터 누적. 단순히 리스트에 갯수를 늘려나감
    W_val.append(xPos)
    cost_val.append(yPos)
sess.close()
```

손실함수와 경사하강법 (Gradient Decent)

- 초기 weight (W (기울기) 또는 b(절편)) 에서 경사를 하강 시켜 가면서 가장 최소의 값을 찾는 것이다.
- y축이 오차이기 때문에 y가 가장 낮을 수록 가장 적절한 값이기 때문이다.
- 2차 함수는 비용 함수인데 가장 낮은 W 는 무엇인가?
- 그렇다. 미분해서 0 이 나오는 값, 즉 최소값이다.(최적화)



손실함수와 경사하강법 (Gradient Decent)

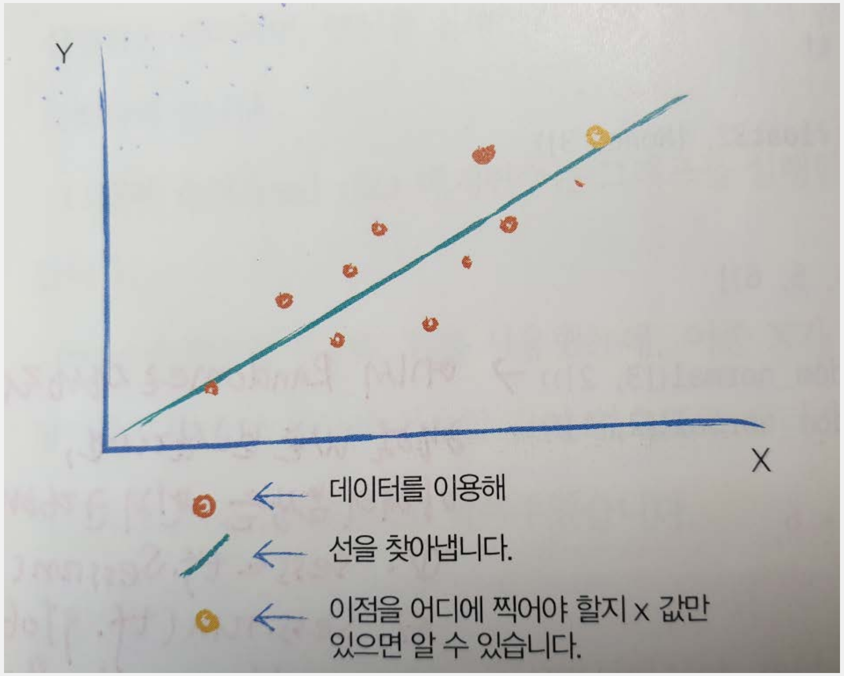
- W (기울기)와 b(절편) 은 각각 따로 편미분을 통해서 최소값을 구하게 된다.
- 당연히 기울기(m) 을 가장 적절하게 바꿔가면서, 전반적인 높, 낮이 (b) 를 바꿔야지 가장 최적의 직선이 될 것 아닌가?

np.polyfit()으로 선형회귀 구현해보기

np.polyfit() 추상화를 통해 선형회귀를 이해한다

선형회귀 구현해보기

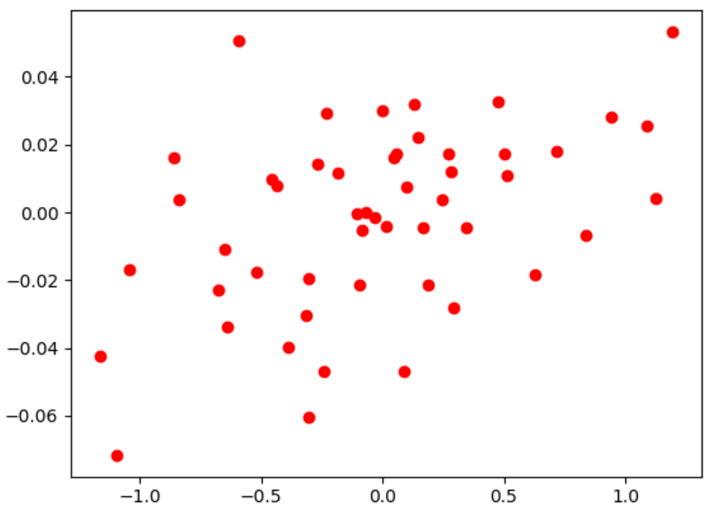
- 선형회귀란 주어진 x와 y값을 가지고 x와 y 간의 관계를 파악하는 것이다.
- 이 관계를 알고 나면, 새로운 x 값이 주어졌을 때, y값을 쉽게 알 수 있다.
- 어떤 입력에 대한 출력을 예측하는 것, 이것이 머신러닝의 기본이다.



데이터 생성

- 랜덤 데이터 생성시 np.random.normal() 메소드 또는 random.randn() 메소드 를 사용한다.

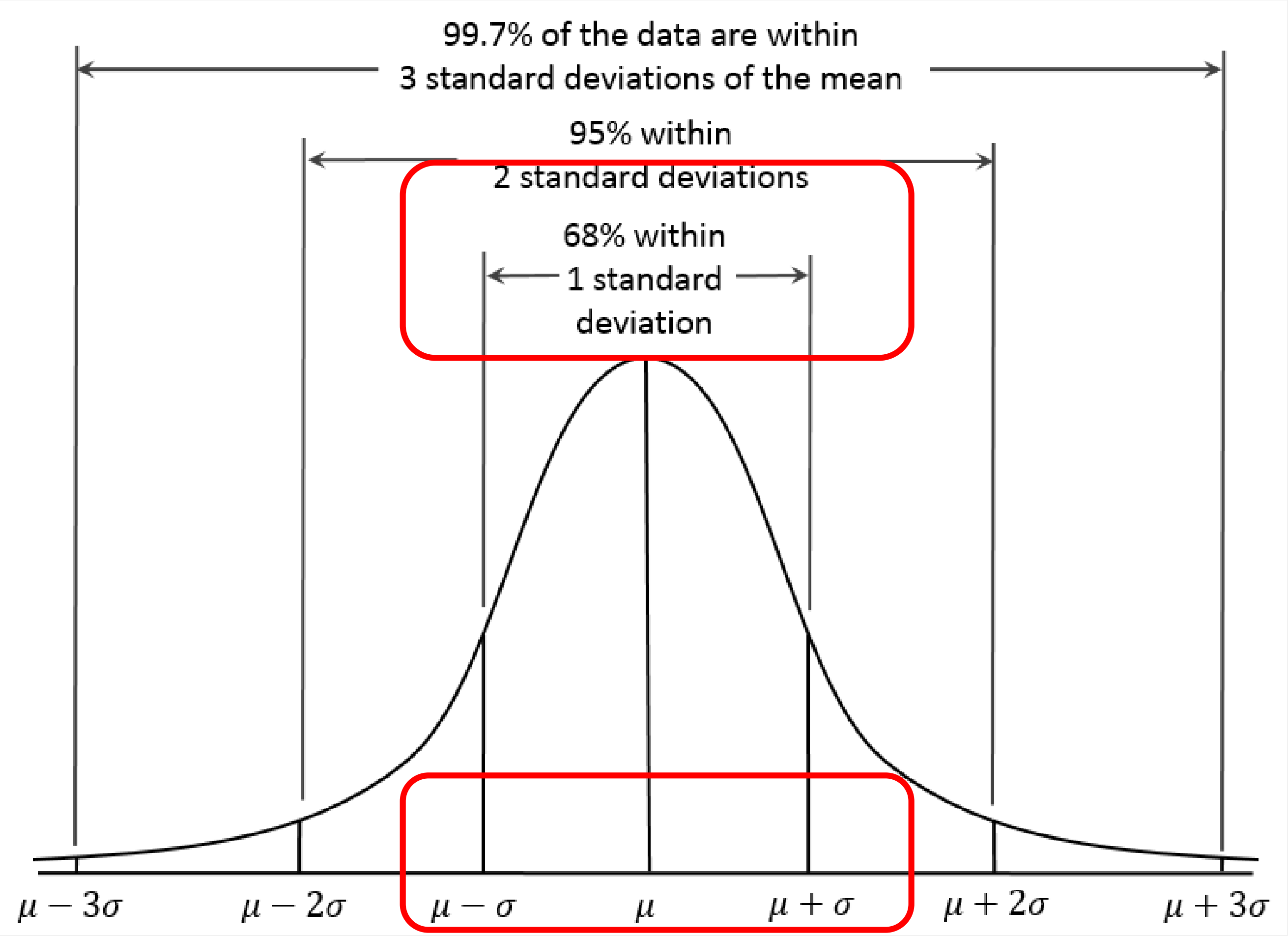
```
import numpy as np
import matplotlib.pyplot as plt

num_points=50
vectors_set=[]

for i in np.arange(num_points):
    x1=np.random.normal(0.0, 0.55)
    y1=x1*0.1*0.3+np.random.normal(0.0, 0.03)
    vectors_set.append([x1, y1])

x_data=[v[0] for v in vectors_set]
y_data=[v[1] for v in vectors_set]

plt.plot(x_data, y_data, 'ro')
plt.show()
```



https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/data_gen.py

잠깐!! Normal Distribution (정규 분포)란?

- 분포의 68%가 1 시그마의 위치에 분포한다



random.normal() 메소드와 random.randn() 메소드

- **np.random.normal(size= (10, 10)) 와 np.random.randn(10, 10) 은 같다.**
- **평균(mu) 0과 표준편차(sigma) 1을 가지는 [10 10] 행렬을 출력으로 준다.**
- np.random.normal() 메소드
  - np.random.normal (loc= 0, scale= 1, size=(n,m)) 의 형태를 따른다.
- mu + sigma * np.random.randn(n,n)
  - np.random.normal(loc= mu, scale= sigma, size=(n,n)) 와 동일한 내용을 가진다.
- 1 + 3 * np.random.randn(10,10)
  - np.random.normal(loc= 1, scale= 3, size=(10,10))은 같은 결과를 가진다.

Normal Distribution (정규 분포)

- np.random.randn () 메소드 사용하여 정규 분포를 만들어 보자
- np.random.randn(발생하는 수의 최소값이 없다, 발생하는 수의 최대값이 없다, A, B)
- Normal Distribution은 발생하는 수의 최소값, 최대값을 결정할 수 없다.
- 왜냐면, 평균 0과 표준 편차 1에 따르는 발생하는 수가 -1에서 1사이에 65%가 발생하기 때문이다.

```
test=np.random.randn(2, 2)
array([[ 0.45917798, -1.21059427],
      [-0.1110276 ,  0.18611427]])
```

Normal Distribution (정규 분포)

- 정규분포의 평균과 분산을 찾아보자
- 정규분포는 평균 0과 표준 편차 1에 따르는 발생하는 수가 -1에서 1사이에 65%가 발생하기 때문에, 평균은 0을 편차는 1을 가진다.

```
import numpy as np
test=np.random.randn(2, 2)
print('test=', test)
nd=np.random.randn(10)
print('mean=', nd.mean())
print('var=', nd.var())
```

```
nd= [ 0.35179773  0.58658899  1.11691527  0.7170146  -0.46622098 -0.47576145
 -1.00675362 -0.11350472  0.36946904 -0.37405974]
mean= 0.07054851045100212
var= 0.39262622138735803

nd= [ 0.36053446  1.20064185 -1.3655847   0.01744486 -0.65374724 -1.89743099
 -1.36894345  0.65940963 -0.40730065  0.87351696]
mean= -0.2581459261123235
var= 1.0035641400305582
```

Normal Distribution (정규 분포)

- 데이터 셋의 수가 작기 때문에 평균과 표준편차의 결과값이 0과 1에 수렴(Converge) 하지 않는다.

```
nd= [ 0.35179773  0.58658899  1.11691527  0.7170146  -0.46622098 -0.47576145
 -1.00675362 -0.11350472  0.36946904 -0.37405974]
mean= 0.07054851045100212
var= 0.3926262213873580
```

```
nd= [ 0.36053446  1.20064185 -1.3655847   0.01744486 -0.65374724 -1.89743099
 -1.36894345  0.65940963 -0.40730065  0.87351696]
mean= -0.2581459261123235
var= 1.0035641400305582
```

데이터 생성 후 디버깅 분석

- x1=np.random.normal(0.0, 0.55)
  - 평균 0과 편차 0.55 사이에서, 1개의 랜덤 숫자를 발생시킴
  - -0.55에서 0.55 사이에 68%의 랜덤 숫자가 발생된다.
- y1=x1*0.1*0.3+np.random.normal(0.0, 0.03)
  - 발생된 x1에 0.1과 0.3을 곱한다. (0.1과 0.3 (=0.03이 기울기 개념)
  - **이 기울기값을 나중에 선형회귀(polyfit())로 찾아내야 한다**.
  - 그리고, 평균 0과 편차 0.03 사이에서, 1개의 랜덤 숫자를 발생시켜서 더한다. (노이즈 개념 또는 y 절편 개념, **이 절편 값을 찾아내야 함**)
- vectors_set.append([x1, y1])
  - 발생된 x1 과 y1을 vectors_set에 저장한다.
- **for** i **in** np.arange(num_points):
  - x1, y1 생성 및 vectors_set 저장 작업을 num_points 횟수만큼 반복

데이터 생성 후 디버깅 분석

- vectors_set

```
vectors_set
[[-0.059537647254096854, 0.021051839490324006],
 [-0.2980240539226151, -0.060478110162352 67],
 [0.1827084754707148, 0.0050051771985430 54],
 [-0.5556608793816922, 0.0099608886800562 28],
 [0.5626040881330123, -0.0369659192047194 4],
 [-0.24989383901693324, -0.020916248646793437],
```

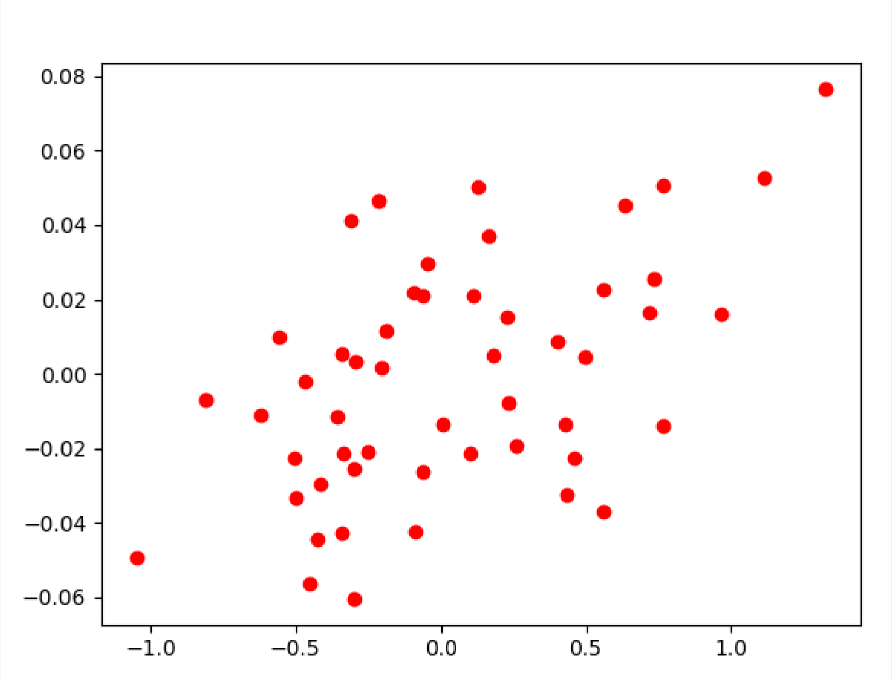
- x_data=[v[0] for v in vectors_set]
  - vectors_set의 첫 번째 요소를 x_data 리스트로 따로 추출함

```
x_data
[-0.059537647254096854,
 -0.2980240539226151,
 0.1827084754707148,
 -0.5556608793816922,
 0.5626040881330123,
```

데이터 생성 후 디버깅 분석

- y_data=[v[1] for v in vectors_set]
  - vectors_set의 두 번째 요소를 y_data 리스트로 따로 추출함

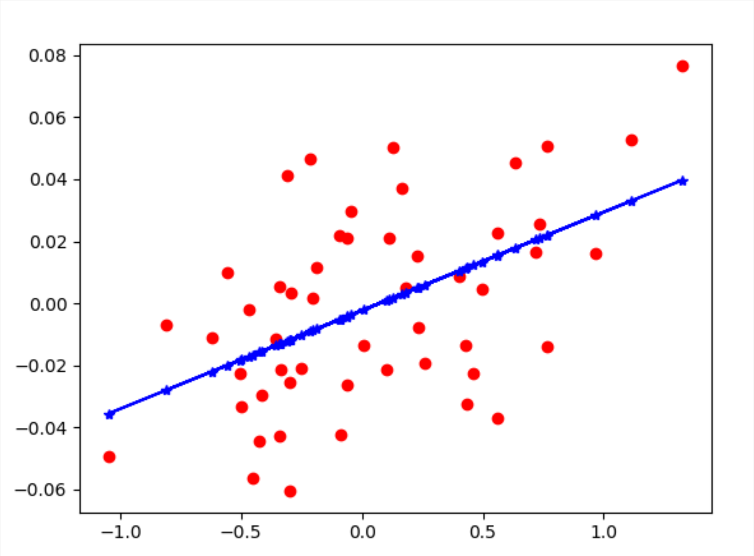
```
y_data
[0.021051839490324006,
 -0.06047811016235267,
 0.005005177198543054,
 0.009960888680056228,
 -0.03696591920471944,
```



- plt.plot(x_data, y_data, 'ro')
  - 추출된 점을 시각화함

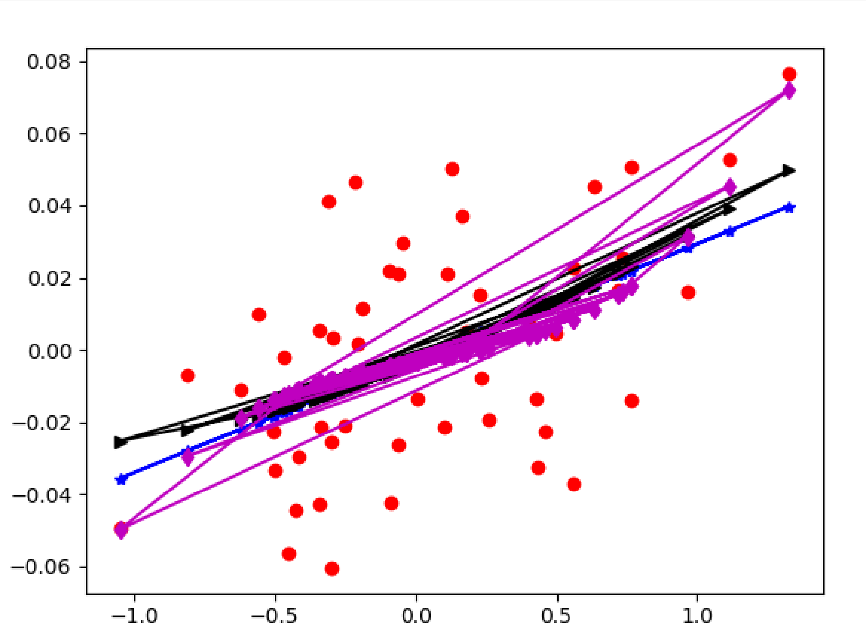
선형회귀 분석 np.polyfit()과 np.polyval() 사용

- **p1=np.polyfit(x_data, y_data, 1) 으로 기울기와 y절편을 계산**
  - **array([0.03172169, -0.00232394])**
- 기울기 값이 0.03임
  - **x1에 0.1과 0.3을 곱해서 0.03이 기울기 개념이 된다는 분석이 합당함**
  - y1=x1*0.1*0.3+np.random.normal(0.0, 0.03)
- plt.plot(x_data, np.polyval(p1, x_data), 'b*-') 으로 선형회귀를 실시함



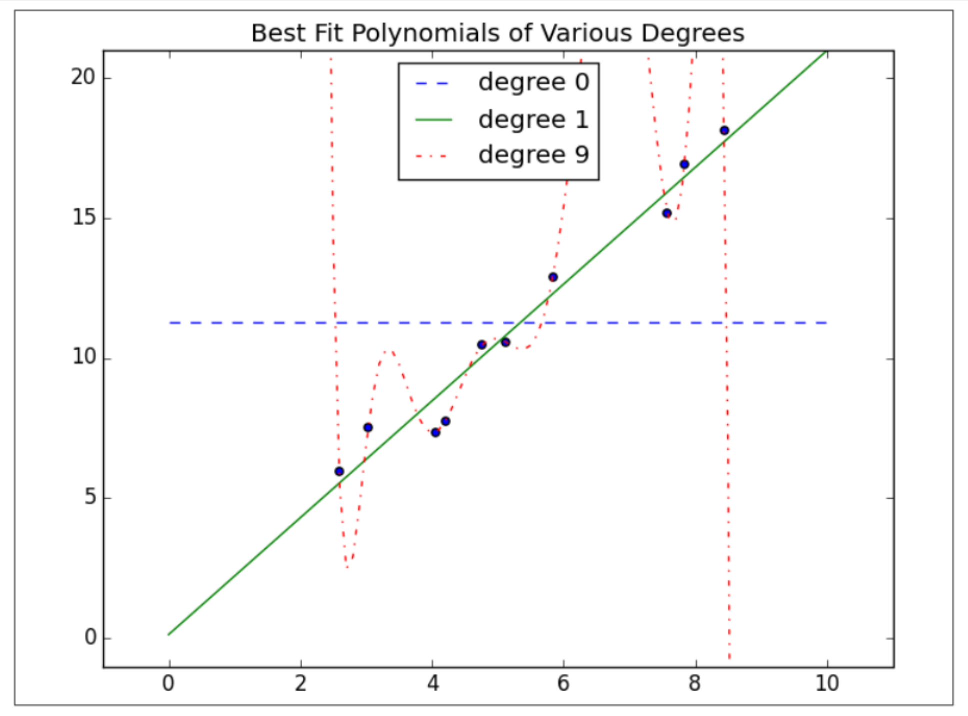
다항 회귀 분석 np.polyfit()과 np.polyval() 사용

- p2=np.polyfit(x_data, y_data, 2)
- plt.plot(x_data, np.polyval(p2, x_data), 'k>-')
- p2 = array([0.00889878, 0.02913994, -0.00452123])
- p3=np.polyfit(x_data, y_data, 3)
- plt.plot(x_data, np.polyval(p3, x_data), 'md-')
- p3=array([0.02595519, -0.00193393, 0.01370987, -0.00327114])
- **p2, p3에 의한 다항 회귀는 그래프상에서 거의 대부분의 training data를 지나고 있다, overfitting 하고 있음, 적절하지 않음**



적합도 (Goodness of Fit, 훈련 방법)

- **degree 0 (i.e., constant) polynomial**
  - **한 개의 훈련 데이터 점도 지나지 않는 언더 피팅을 하고 있음**
- **degree 9 (i.e., 10-parameter) polynomial**
  - **모든 훈련 데이터 점을 지나는 오버 피팅을 하고 있음**



텐서플로우로 선형회귀 구현해 보기

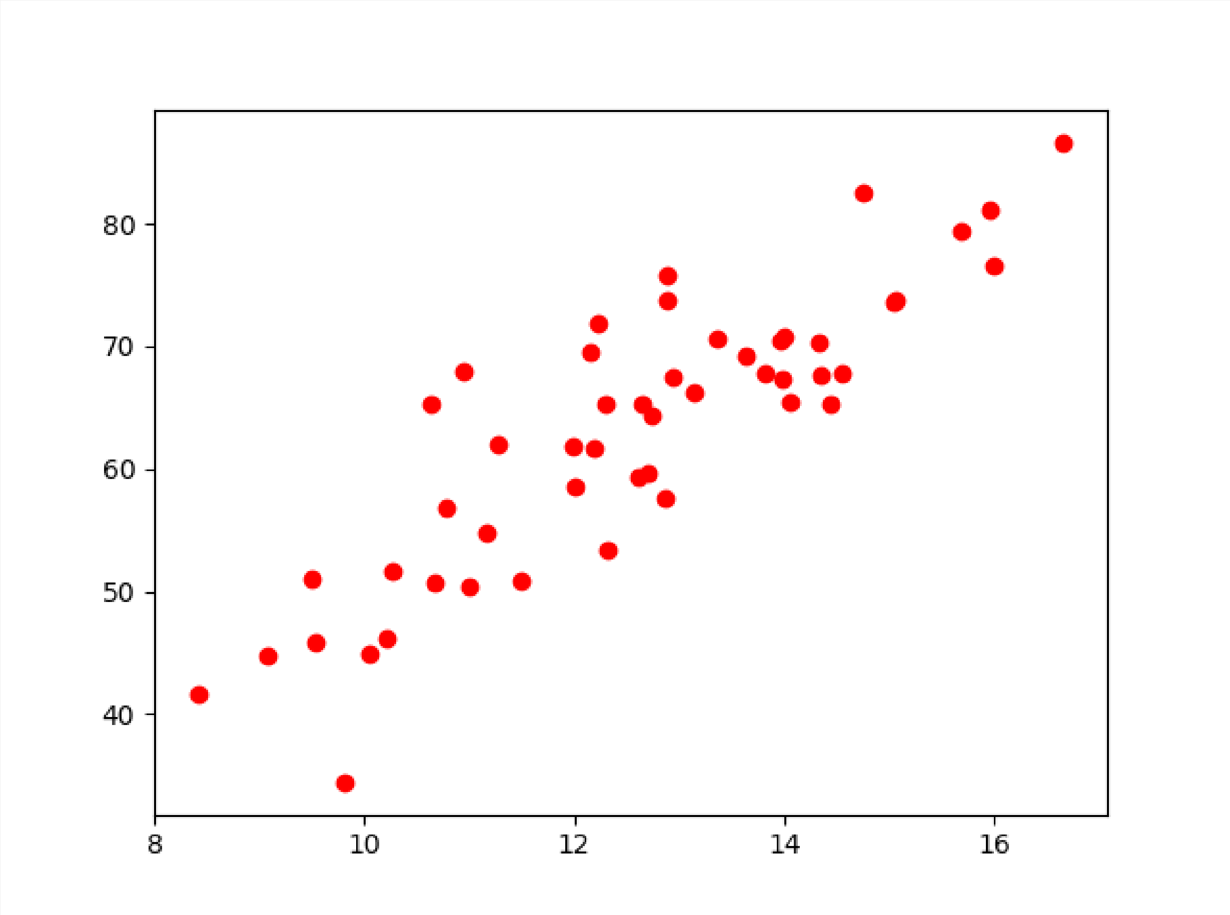
텐서플로우의 추상화를 통해 선형회귀를 이해한다

https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/regression_class.py

국민대학교 소프트웨어학부

데이터의 분포와 생성

- 아래와 같은 데이터 분포를 가지는 데이터를 생성해보자



데이터 생성

- Data_Generation() 메소드를 이용한다
- x와 y의 값을 랜덤 발생시켜 50개의 데이터를 생성한다.
- 난수 발생의 특성 상, 각 개인의 랜덤 값이 다르게 나타난다.

```
def Data_Genearion(num_points):
    # num_points = 50
    vectors_set = []
    for i in np.arange(num_points):
        x = np.random.normal(2, 2) + 10
        y = x * 5 + (np.random.normal(0, 3)) * 2
        vectors_set.append([x, y])

    x_data = [v[0] for v in vectors_set]
    y_data = [v[1] for v in vectors_set]

    return  x_data, y_data
```

Data Visualization (데이터 시각화)

- 생성된 50개의 점을 그리는 메소드이다.
- 난수 발생의 특성 상, 각 개인의 데이터 시각화는 다르게 나타난다.

```
def Data_Draw(x_data, y_data):
    plt.figure(100)
    plt.plot(x_data, y_data,'ro')
    plt.ylim([0,100])
    plt.xlim([0,25])
    plt.xlabel('x')
    plt.ylabel('y')
    #plt.legend()
    plt.show()
```

데이터 시각화

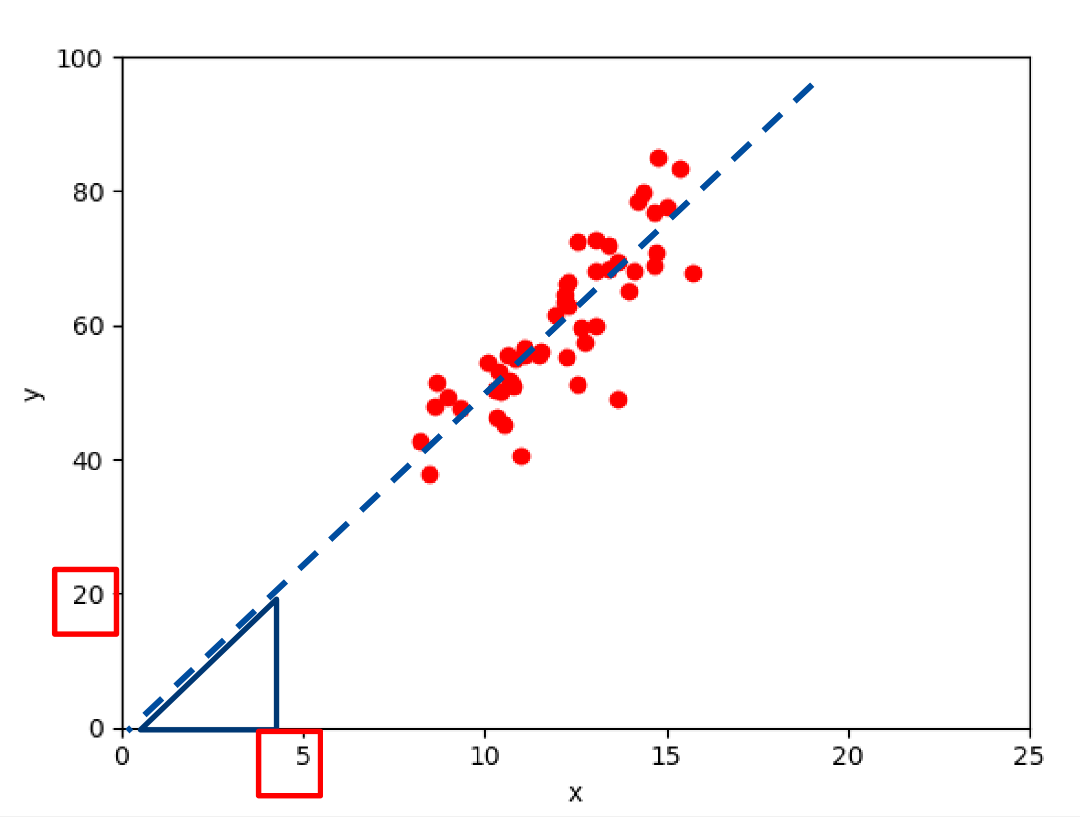
데이터 생성 후 그래프 분석 을 해 본다

https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/regression_class.py

국민대학교 소프트웨어학부

데이터 시각화를 통한 Graphical Method

- 생성된 데이터 50개 점에 직선을 그려 보면, 기울기가 **대략 4~6 정도 된다.** 절편은 **어림잡아 -10에서 10 사이의 값이** 예상 된다. **정확한 값이 아니라, 눈대중으로 읽었다.**



데이터 생성 후 디버깅 분석

- x = np.random.normal(2, 2) + 10
  - **평균 2과 편차 2 사이 [0,4]** 에서, 1개의 난수를 발생시켜서 10을 더함
  - **[0, 4] @ 68% +10 → [10, 14] @ 68% 의 난수 발생.**
- y = x * 5 + (np.random.normal(0, 3)) * 2
  - **발생된 난수 x에 5를 곱한다. (5가 기울기 개념)**
  - **이 기울기값을 나중에 선형회귀(텐서플로우)로 찾아내야 한다**.
  - **평균 0과 편차 3 사이 [-3, 3] 에서, 1개의 난수를 발생시켜 2를 곱한다. (노이즈 개념 또는 y 절편 개념, 이 절편 값을 찾아내야 함)**
  - **[-3, 3] @ 68% *2 = [-6, 6] @ 68%**
- y = x * 5 + (np.random.normal(0, 3)) * 2
  - y 는 **[10, 14] @ 68% *5=[50, 70] @ 68% + [-6, 6] @ 68% = [44, 76] @ 68% 의 난수 발생**

데이터 생성 후 디버깅 분석

- vectors_set.append([x1, y1])
  - 발생된 x 와 y을 vectors_set에 저장한다.
- **for** i **in** np.arange(num_points):
  - x, y 생성 및 vectors_set 저장 작업을 num_points 횟수만큼 반복

난수 발생 범위 디버깅 분석

- x_data : **[10, 14] @ 68% 의 난수 발생**

- [11.353811375962337,
- 11.689379764438412,
- 10.923584614257171,
- 13.281504414253567,
- 12.885320072037343,
- 14.2138185713159,
- 10.312947592147651,
- 14.096639379239809,
- 10.639137335477688,
- 10.503629956441367,
- 10.029899423753488,

- y_data: **[44, 76] @ 68% 의 난수 발생**

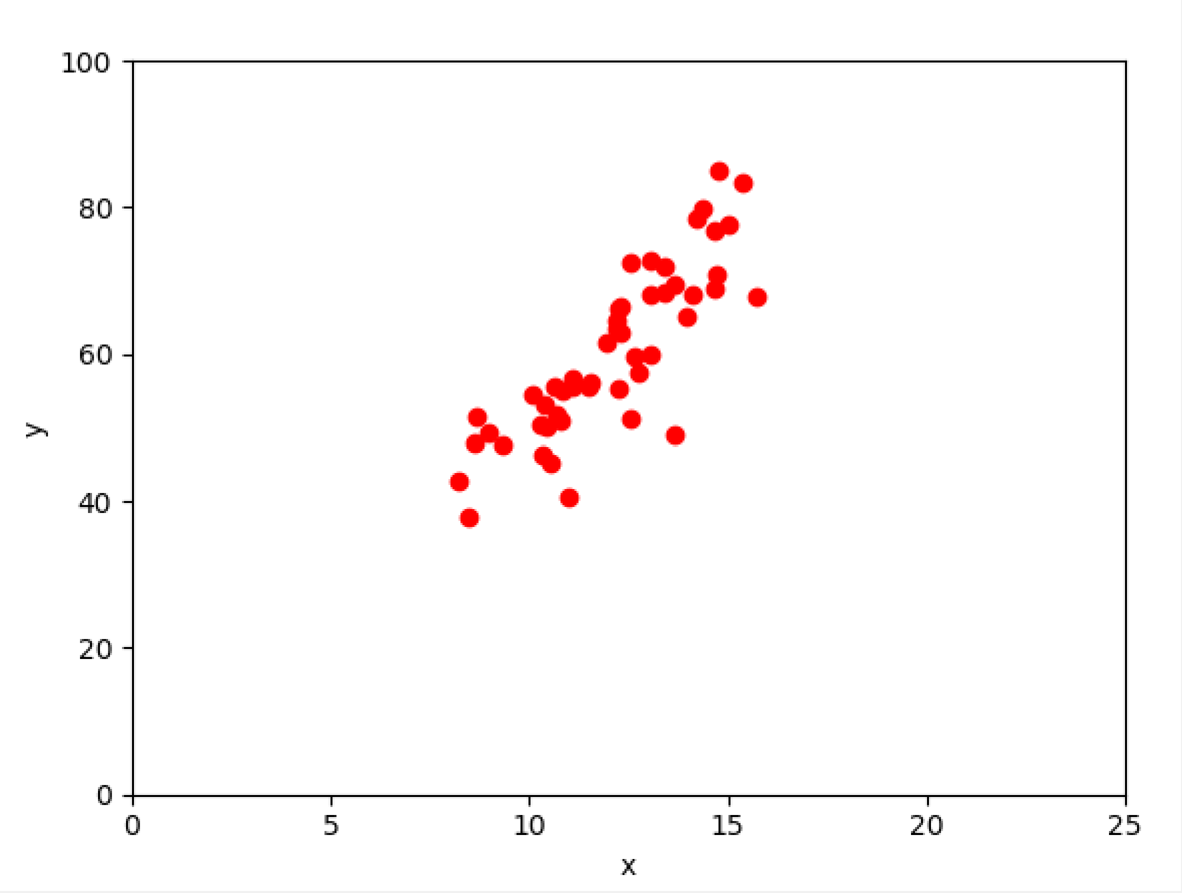
- [57.406450453005746,
- 56.95610239967386,
- 58.11110324939592,
- 74.35952771524798,
- 72.8046600926817,
- 86.89776182326789,
- 51.587319883271135,
- 74.2625854757124,
- 54.43205800066196,
- 54.07464208506491,
- 51.41882135773061,

난수 발생 범위 디버깅 분석

- **x_data: [10, 14] @ 68% 의 난수 발생**
  - x_data=np.array(x_data)
  - np.size(np.where((x_data>=10) & (x_data<=14)))
  - 30
  - num_points가 50개중 30개가 [10, 14] 사이에 발생, 60% 발생
  - np.size(np.where((x_data>=10) & (x_data<=14)))/num_points*100
- **y_data: [44, 76] @ 68% 의 난수 발생**
  - y_data=np.array(y_data)
  - np.size(np.where((y_data>=44) & (y_data>=76)))
  - 37
  - num_points가 50개중 37개가 [44, 76] 사이에 발생, 74% 발생
  - np.size(np.where((y_data>=44) & (y_data<=76)))/num_points*100

데이터시각화

- plt.xlim([0,25]): x_data가 [10, 14] @ 68% 의 난수 발생됨
- plt.ylim([0,100]): y_data가 [44, 76] @ 68% 의 난수 발생됨

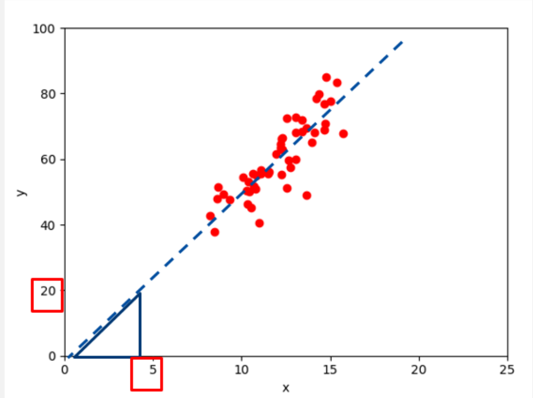


polyfit() 메소드를 통한 비교

- p1=np.polyfit(x_data, y_data, 1)
- array([5.26760467, -3.80478759])
- 즉, 기울기가 5.26 이고, y절편이 -3.8인 직선의 기울기를 가진 선형회귀로 모델링될 수 있다.
- 눈대중으로 읽은 값과 비교해 본다.

데이터 시각화를 통한 Graphical Method

- 생성된 데이터 50개 점에 직선을 그려 보면, 기울기가 **대략 4~6 정도 된다.** 절편은 **어림잡아 -10에서 10 사이의 값이** 예상 된다. **정확한 값이 아니라, 눈대중으로 읽었다.**



국민대학교 소프트웨어학부

학습 (Data_Learning)

텐서플로우를 이용하여 학습을 수행한다.

https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/regression_class.py

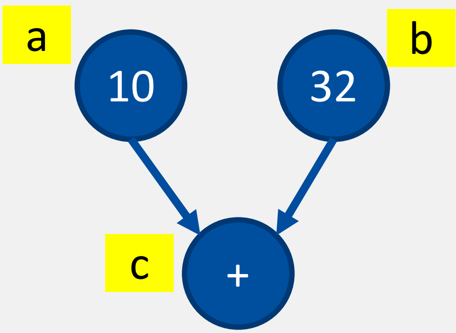
텐서 연산

- print(c) 하면, 42가 나올 것으로 생각할 수 있으나, 텐서의 형태로 출력한다
- 그 이유는 텐서플로의 프로그램 구조가 **1. 그래프 생성, 2. 그래프 실행**의 두 가지로 분리되어 있음

```
a=tf.constant(10)
<tf.Tensor 'Const_1:0' shape=() dtype=int32>
```

```
b=tf.constant(32)
<tf.Tensor 'Const_2:0' shape=() dtype=int32>
```

```
c=tf.add(a,b)
<tf.Tensor 'Add:0' shape=() dtype=int32>
```



그래프 생성

```
print(sess.run(hello))
b'Hello, TensorFlow!'
```

```
print(sess.run([a, b, c]))
[10, 32, 42]
```

b'Hello, TensorFlow!'

10+32, 42

그래프 실행

연산을 실제로 수행하는 시점

그래프와 지연 실행(Lazy Evaluation)

- 그래프: 텐서들의 연산 모음
- 텐서와 텐서의 연산들을 먼저 정의하여 그래프를 만듦
- 이후 필요할 때 연산을 실행하는 코드를 넣어 '원하는 시점'에 실제 연산을 수행함
- 지연 실행(Lazy Evaluation)이라 함
- 그래프의 실행은 Session 안에서 이루어져야 함
- Session 객체와 run 메서드를 이용함

```
sess=tf.Session()
print(sess.run(hello))
b'Hello, TensorFlow!'

print(sess.run([a, b, c]))
[10, 32, 42]
sess.close()
```

hello() 함수

```
import tensorflow as tf
def hello():
    a = tf.constant('hello, tensorflow!')
    print(a)  # Tensor("Const:0", shape=(), dtype=string)

    sess = tf.Session()
    result = sess.run(a)

    # 2.x 버전에서는 문자열로 출력되지만, 3.x 버전에서는 byte 자료형
   # 문자열로 변환하기 위해 decode 함수로 변환
   print(result)  # b'hello, tensorflow!'
    print(type(result))  # <class 'bytes'>
    print(result.decode(encoding='utf-8'))  # hello, tensorflow!
    print(type(result.decode(encoding='utf-8')))  # <class 'str'>

    # 세션 닫기
   sess.close()

if '__name__' == '__main__':
    hello()

Tensor("Const_5:0", shape=(), dtype=string)
b'hello, tensorflow!'
<class 'bytes'>
hello, tensorflow!
<class 'str'>
```

https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/tf_type.py

학습 Data_Learning

- Data_Learning() 메소드를 통하여 학습한다.
- **주어진 x_data와 y_data의 50개 점들에 대해 최소의 에러값(손실값)을 제공하는 기울기 (W)와 y절편 (b)값을 계산한다.**
- 최소자승법을 이용한다.
- GradientDecentOptimizer() 메소드를 이용한다.

```
def Data_Learning(x_data, y_data):
    W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
    b = tf.Variable(tf.zeros([1]))
    y = W * x_data + b
    loss = tf.reduce_mean(tf.square(y - y_data))
    #lo.append(loss)
    optimizer = tf.train.GradientDescentOptimizer(0.0015) # 0.1, 0.01 0.001 0.0015
    train = optimizer.minimize(loss)
    init = tf.initialize_all_variables()
    sess = tf.Session()
    sess.run(init)
```

텐서플로우 변수 초기화와 학습율

- 생성된 50개의 x, y 좌표의 상관관계를 설명하기 위한 변수들인 **W를 -1.0부터 1.0사의의 균등분포를 가진 무작위 값으로 초기화하고, b를 0으로 초기화한다.**
- GradientDecentOptimizer() 메소드의 학습율을 0.1, 0.01, 0.001 등으로 변화시켜 최적의 학습율을 지정한다.
  – 최적의 학습율을 제공하는 코드를 작성한다

```
def Data_Learning(x_data, y_data):
    W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
    b = tf.Variable(tf.zeros([1]))
    y = W * x_data + b
    loss = tf.reduce_mean(tf.square(y - y_data))
    #lo.append(loss)
    optimizer = tf.train.GradientDescentOptimizer(0.0015) # 0.1, 0.01 0.001 0.0015
    train = optimizer.minimize(loss)
    init = tf.initialize_all_variables()
    sess = tf.Session()
    sess.run(init)
```

변수 초기화 내용 보기

- 변수 초기화 내용을 보기 원한다면 아래와 같이 수행한다.
- **W를 -1.0부터 1.0사의의 균등분포를 가진 무작위 값으로 초기화하고, b를 0으로 초기화한다.**
- sess = tf.Session()
- init = tf.initialize_all_variables()
- sess.run(init)
- sess.run(W)
- array([-0.8621006], dtype=float32)
- sess.run(b)
- array([0.], dtype=float32)

균등분포, tf.random_uniform 메소드 ?

- W = tf.Variable(tf.random_uniform([2], -1.0, 1.0))
- -1.0과 1.0사이에서만 100% 발생되는 난수 2개를 발생시킴
- **정규분포는 평균과 1 편차사이에서 난수 68% 발생**
- **균등분포는 정해진 범위사이에서 난수 100% 발생**
- 결과 확인은 아래의 코드를 실행하면 된다.
- sess = tf.Session()
- init = tf.initialize_all_variables()
- sess.run(init)
- sess.run(W)
- array([-0.7264285 , 0.40802813], dtype=float32)

y = W * x_data + b 의 의미

- **W와 x_data 의 곱과 b와의 합을 통해 x_data (x가 아님)와 y_data (y가 아님) 의 관계를 설명하겠다는 뜻이다**
- **x_data 가 주어졌을 때, y_data 를 만들어 낼 수 있는 W와 b를 찾아내겠다는 의미이다.**
- W:가중치 (weight), b: 편향(bias)

```
def Data_Learning(x_data, y_data):
    W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
    b = tf.Variable(tf.zeros([1]))
    y = W * x_data + b
    loss = tf.reduce_mean(tf.square(y - y_data))
    #lo.append(loss)
    optimizer = tf.train.GradientDescentOptimizer(0.0015) # 0.1, 0.01 0.001 0.0015
    train = optimizer.minimize(loss)
    init = tf.initialize_all_variables()
    sess = tf.Session()
    sess.run(init)
```

y = W * x_data + b 의 데이터 타입

- y = W * x_data + b
- <tf.Tensor 'add_5:0' shape=(50,) dtype=float32>
- W , b가 텐서플로우 그래프임으로, x_data 가 리스트 타입에도 불구하고, **y는 텐서플로우 그래프임**
- 참고로, **y_data는 리스트 타입임**
- **W와 x_data 의 곱과 b와의 합을 통해 x_data (x가 아님)와 y_data (y가 아님) 의 관계를 설명하겠다는 뜻이다**
- **y는 텐서플로우 그래프이고, y_data는 리스트 타입임**

loss = tf.reduce_mean(tf.square(y - y_data))

- W , b, **y가 텐서플로우 그래프임**으로, **loss** **는 텐서플로우 그래프임**
- <tf.Tensor 'Mean_6:0' shape=() dtype=float32>
- sess = tf.Session(); init = tf.initialize_all_variables(); sess.run(init); sess.run(tf.square(y - y_data))을 통해 tf.square(y - y_data)의 값을 알아보자
- array([4466.662 , 4443.7363, 4513.225 , 7290.8813, 6972.7266, 9741.495 , 3617.995 , 7390.239 , 4002.5022, 3943.2634, 3569.6172, 5960.148 , 1696.7238, 4028.826 , 6603.5884, 3380.1748, 4495.2427, 4864.013 , 5929.98 , 2830.981 , 6873.971 , 3916.5583, 3053.5615, 7159.072 , 4283.92 , 4221.1597,], dtype=float32)

sess.run(y-y_data) 디버깅 분석

- sess.run(y)의 값을 알아보자
- array([-9.426642 , -9.705253 , -9.069442 , -11.027134 , -10.698196 , -11.801251 , -8.562452 , -11.70391 , -8.833276 , -8.720769, ..., dtype=float32)
- y_data의 값을 알아보자
- [57.40, 56.96, 58.11, 74.34, 72.80, 86.89, ...]
- sess.run(y-y_data) 의 값을 알아보자
- array([-66.83309 , -66.661354, -67.18054 , -85.38666 , -83.50285 ,
- -98.69901 , -60.149773, -85.9665 , -63.26533 , -62.79541 ,

sess.run(tf.square(y - y_data)) 디버깅 분석

- sess.run(tf.square(y - y_data)) 의 값을 알아보자
- array([4466.662 , 4443.7363, 4513.225 , 7290.8813, 6972.7266, 9741.495 , 3617.995 , 7390.239 , 4002.5022, 3943.2634,
- sess.run(tf.square(y[0]-y_data[0])) 의 값을 알아보자
- 4466.662

loss = tf.reduce_mean(tf.square(y - y_data))

- loss = tf.reduce_mean(tf.square(y - y_data)) 3개의 항목에 대해서만 디버깅 분석을 통해 보면, 아래와 같다.
- **tf.reduce_mean () 메소드는 제곱한 에러 손실값의 평균을 구하는 메소드임이 밝혀졌다**
- sess.run(tf.square(y[0:3]-y_data[0:3])) 의 값을 알아보자
- array([4466.662 , 4443.7363, 4513.225], dtype=float32)
- sess.run(tf.reduce_mean(tf.square(y[0:3]-y_data[0:3]))) 의 값을 알아보자
- 4474.541
- np.mean([4466.662 , 4443.7363, 4513.225]) 의 값을 알아보자
- 4474.5411

손실(loss) 또는 비용(cost) 함수

- **손실함수(loss function)는 한 쌍(x_data, y_data)의 데이터에 대한 예측값 (y)과의 손실값을 계산하는 함수이다**
- **손실값이란 실제값(y_data)과 모델로 예측한 값(y)이 얼마나 차이가 나는가를 나타내는 값이다.**
- **손실값이 작을수록 그 모델(y) 이 x_data 와 y_data 의 관계를 잘 설명하고 있다는 뜻이며, 주어진 x_data 값에 대한 y_data 값을 정확하게 예측할 수 있다는 뜻이다.**
- **이 손실을 전체 데이터에 대해 구한 경우 비용(cost)라고 한다**
- 비용 함수는 tf.reduce_mean () 이용한 loss 또는 cost 로 구현
- loss = tf.reduce_mean(tf.square(y - y_data))
- cost=tf.reduce_mean(tf.square(hypothesis-Y))

손실(loss) 코드 리뷰

- sess = tf.Session(); init = tf.initialize_all_variables(); sess.run(init); sess.run(y) 의 값을 알아보자
- array([10.564473 , 10.876712 , 10.164157 , 12.35815 , 11.989508 , 13.225705 , 9.595971 , 13.116614 , 9.899485, ..
- loss = tf.reduce_mean(tf.square(y - y_data)) 의 값을 알아보자
- sess.run(loss)
- 2419.6118

학습과 손실함수 관계

- 학습이란, 변수들(W:가중치 (weight), b: 편향(bias))의 값을 다양하게 넣어 계산해보면서 이 손실값을 최소화하는 W 와 b의 값을 구하는 것이다.
- 손실값으로는 '예측값과 실제값의 거리'를 가장 많이 사용한다.
- 손실값은 예측값에서 실제값을 뺀 뒤 제곱하여 구하며, 그리고, 비용은 모든 데이터에 대한 손실값의 평균을 내어 구한다

텐서플로우 그래프 생성

학습에서 텐서플로우의 그래프 생성 과정을 이해한다.

https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/regression_class.py

텐서플로우 그래프 생성단계

- 학습에 필요한 텐서플로우 그래프들을 생성했다.
- W, b 가 텐서플로우 그래프임으로, W, b에 의해 만들어진 y도 텐서플로우 그래프이다.
- 랜덤 발생된 x_data 변수의 값은 확인할 수 있지만, 텐서플로우 그래프인 W, b, y는 텐서플로우 그래프 생성 단계에 있음으로 아직 콘솔 디버깅에서 그 값을 확인할 수 없다. W, b, y 값은 초기화 되어 있다.

```
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b
loss = tf.reduce_mean(tf.square(y - y_data))
#lo.append(loss)
optimizer = tf.train.GradientDescentOptimizer(0.0015) # 0.1, 0.01 0.001 0.0015
train = optimizer.minimize(loss)
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

텐서플로우 그래프 실행

텐서플로우의 그래프 실행 과정을 이해한다.

https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/regression_class.py

텐서플로우 그래프 실행

- **머신러닝에서는 구동시켜 보기 전에는, 현재 데이터가 무엇인지 판단할 수 없는 상황이 매우 많다.**
- run 함수를 호출하기 전에는 값을 알 수 없기 때문에 일관되게 처리하기 위해서는 **모든 텐서 객체에 대해 자신이 누구인지만 알려주는 요약본을 출력하는 것이 맞다.**
- W: <tf.Variable 'Variable:0' shape=(1,) dtype=float32_ref>
- b: <tf.Variable 'Variable_1:0' shape=(1,) dtype=float32_ref>
- loss: <tf.Tensor 'Mean:0' shape=() dtype=float32>
- **텐서플로우 프로그램을 구동하기 위해서는 세션이 필요하다.**
- **텐서플로우 구동은 세션에 포함된 run 함수를 호출하면 된다.**

텐서플로우 그래프 실행

- 그래프 실행은 Session 안에서 이루어져야 한다.
- 그래프 실행은 Session 객체와 run 메서드를 이용한다

```
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
train_set = []  ###
for step in np.arange(20):
    sess.run(train)
    print(step, sess.run(W), sess.run(b))
    print(step, sess.run(loss))
    train_set.append([sess.run(W), sess.run(b), sess.run(loss)])  ###
    plt.figure(step)
    plt.plot(x_data, y_data, 'ro')
    plt.plot(x_data, sess.run(W) * x_data + sess.run(b))
    plt.xlabel('x')
    plt.ylabel('y')
    #plt.legend()
    plt.show()
W_data = [t[0] for t in train_set]
v_data = [t[1] for t in train_set]
Loss_data= [t[2] for t in train_set]
return W_data,v_data, Loss_data
```

텐서플로우 변수 초기화 수행

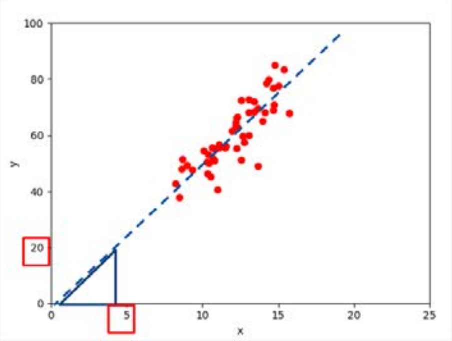
- tf.initialize_all_variables() 메소드를 통해 변수 초기화를 위한 init 그래프 객체를 생성한다.
  - init = tf.initialize_all_variables()
  - <tf.Operation 'init_1' type=NoOp>
- tf.Session() 메소드로 sess 그래프 객체를 만든다.
  - sess = tf.Session()
  - <tensorflow.python.client.session.Session at 0x22502e08fd0>
- init 그래프 객체를 run 메서드의 파라미터 값으로 주어, sess.run(init) 메소드로 텐서플로우에 사용되는 변수 초기화를 수행한다
  - sess.run(init)

텐서플로우 그래프 실행 디버깅

- **sess.run(train)**
  - Session 객체의 run 메서드를 통해, train 객체를 수행한다
  - **loss** = tf.reduce_mean(tf.square(y - y_data))
  - **train** = optimizer.minimize(**loss**)
- train 객체는 loss 객체를 실행한 후에 step 0번째 학습 결과인 W와 b 값을 출력한다.
- print(step, sess.run(W), sess.run(b))
  - 0 [3.553465] [0.31483752]
  - W값은 3.553465 이며, (개인마다 다름)
  - b 값은 0.31483752이다. (개인마다 다름)
  - Graphical Method에서 기울기 4정도
  - 절편은 -10과 10사이의 값 예상에 부합

데이터 시각화를 통한 Graphical Method

- 생성된 데이터 50개 점에 직선을 그려 보면, 기울기가 **대략 4~6 정도 된다.** 절편은 **어림잡아 -10에서 10 사이의 값이** 예상 된다. **정확한 값이 아니라, 눈대중으로 읽었다.**



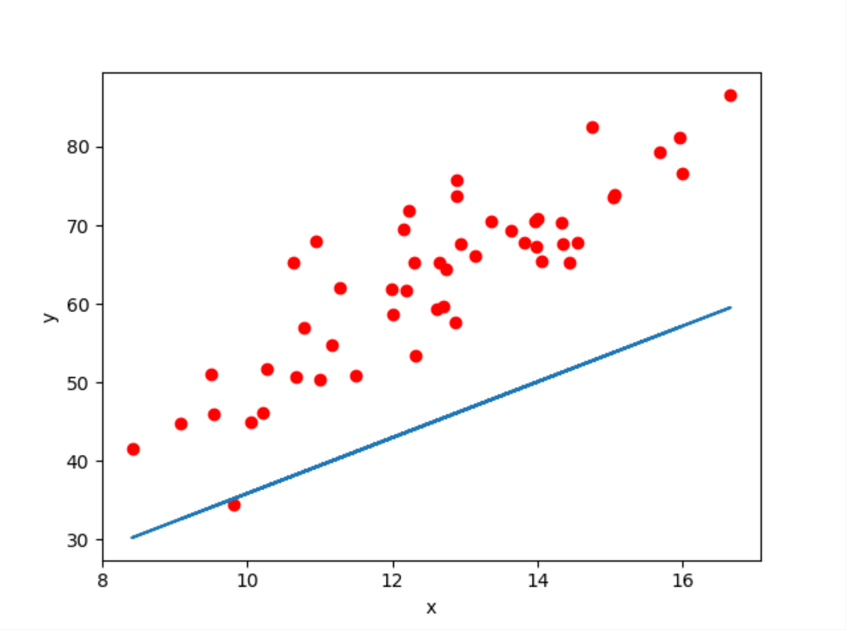
텐서플로우 그래프 실행 디버깅

- print(step, sess.run(loss))
  - Step 0 에서의 loss 값을 출력한다.
  - 0   367.4683
  - loss 값은 367.46이다.
- train_set.append([sess.run(W), sess.run(b), sess.run(loss)])
  - train_set 에 Step 0 에서의 W, b, loss 값을 저장한다.
  - train_set
    - [[array([3.553465], dtype=float32),   array([0.31483752], dtype=float32), 367.4683]]

Step 0(첫 번째 학습)에서의 loss 값 콘솔 디버깅

- Step 0 첫 번째 학습 결과인 W, b, loss 값을 계산하여 출력한다.

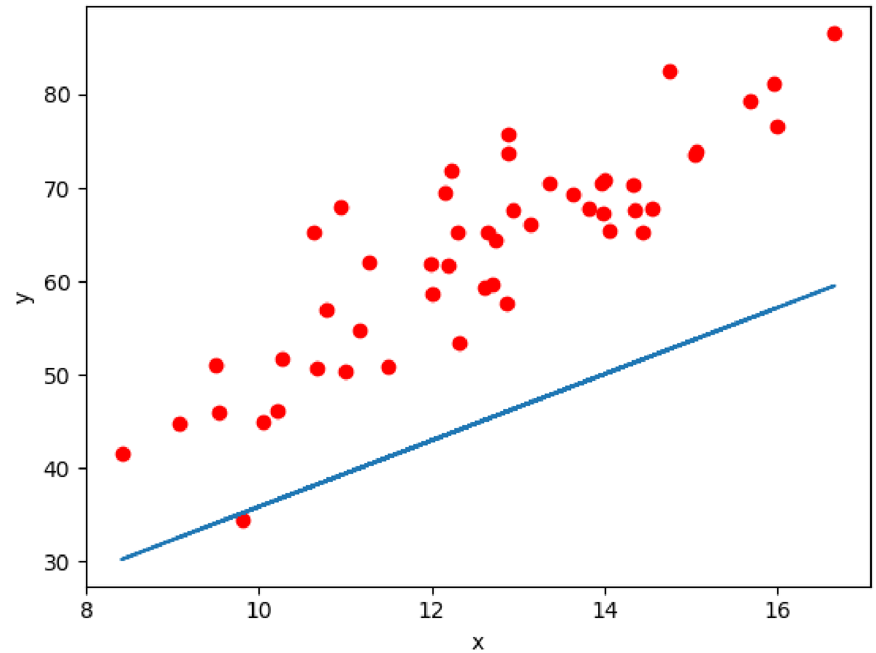
```
sess.run(train)
print(step, sess.run(W), sess.run(b))
print(step, sess.run(loss))
train_set.append([sess.run(W), sess.run(b), sess.run(loss)])  ###
```



Step 0(첫 번째 학습)에서의 loss 값 콘솔 디버깅

- Step 0 첫 번째 학습 결과인 W와 b 값을 이용하여, 예측 직선 (y)을 x_data, y_data 좌표 위에 출력한다.

```
plt.figure(step)
plt.plot(x_data, y_data, 'ro')
plt.plot(x_data, sess.run(W) * x_data + sess.run(b))
plt.xlabel('x')
plt.ylabel('y')
#plt.legend()
plt.show()
```



Step 0(첫 번째 학습)에서의 loss 값 콘솔 디버깅

- y=x_data*W+b
  - <tf.Tensor 'add_2:0' shape=(50,) dtype=float32>
- sess.run(y)를 수행하면, y 그래프 값을 확인할 수 있다
- y=sess.run(y)를 수행하면, y 변수에 y값을 저장한다.

```
array([42.92975 , 40.00953 , 46.08135 , 45.447483, 49.918053, 51.624714,
       34.05071 , 52.764053, 57.15245 , 42.98017 , 51.312218, 57.04682 ,
       43.72928 , 41.14911 , 43.657482, 39.395763, 44.090137, 38.637604,
       45.117508, 46.29372 , 51.215588, 44.00339 , 50.25373 , 36.61917 ,
       38.0987  , 36.795113, 45.256386, 34.173832, 35.144196, 45.60984 ,
       53.855278, 46.03917 , 40.39592 , 32.584106, 53.79027 , 49.385914,
       50.0337  , 47.026936, 36.007835, 52.042915, 39.187145, 48.742886,
       46.10148 , 30.20313 , 43.508713, 47.80955 , 59.52558 , 56.052826,
       38.24937 , 50.064648], dtype=float32)
```

Step 0(첫 번째 학습)에서의 loss 값 콘솔 디버깅

- tf.square(y - y_data)을 수행한다.
  - Out[97]: <tf.Tensor 'Square_5:0' shape=(50,) dtype=float64>
- tf.square(y - y_data)의 결과값을 확인하려면 sess.run(tf.square(y - y_data)) 을 수행한다.

```
array([3.55887660e+02, 2.19515568e+02, 7.65990154e+02, 2.00963266e+02,
       4.24603277e+02, 1.86404216e+02, 2.86510325e+02, 8.83120084e+02,
       3.75882401e+02, 2.44313768e+02, 2.68821511e+02, 5.83241280e+02,
       7.93462849e+02, 9.42321300e+01, 3.23208629e+02, 1.20379282e+02,
       8.60770028e+01, 3.33349419e+02, 2.02680634e+02, 4.52072610e+02,
       3.65716953e+02, 4.54534894e+02, 2.29317497e+02, 9.05798690e+01,
       7.37774252e+02, 2.20189581e+02, 4.01763774e+02, 1.37117623e+02,
       5.06256283e-01, 3.52435534e+02, 3.98363520e+02, 1.35701994e+02,
       4.68060278e+02, 1.47089562e+02, 3.93036451e+02, 3.36543665e+02,
       3.00211507e+02, 3.66502536e+02, 7.88853182e+01, 2.48265203e+02,
       8.26627877e+02, 4.21276920e+02, 8.77037917e+02, 1.29749703e+02,
       6.73178502e+02, 5.18197669e+02, 7.35094790e+02, 5.42157841e+02,
       1.54648405e+02, 4.32132662e+02])
```

Step 0(첫 번째 학습)에서의 loss 값 콘솔 디버깅

- tf.reduce_mean(tf.square(y - y_data))을 수행한다.
- tf.reduce_mean(tf.square(y - y_data))의 결과값을 확인하려면 sess.run(tf.reduce_mean(tf.square(y - y_data))) 을 수행한다.
- sess.run(tf.reduce_mean(tf.square(y - y_data)))
- Out[106]: **367.46829244768355**
- Step 0 에서의 loss 출력값을 콘솔 디버깅으로 확인했다.
  - print(step, sess.run(loss))
  - 0 367.4683
  - loss 값은 367.46이다.

Step 0(첫 번째 학습)에서의 loss 값 콘솔 디버깅

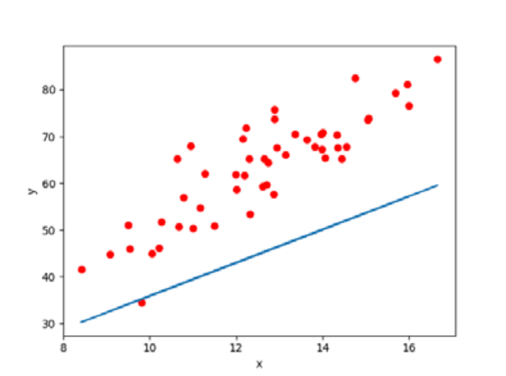
- Step 0 에서의 loss 값을 출력값, W, b, loss 을 아래의 코드를 통해 저장한다.
- W_data = [t[0] for t in train_set]
- b_data = [t[1] for t in train_set]
- Loss_data= [t[2] for t in train_set]

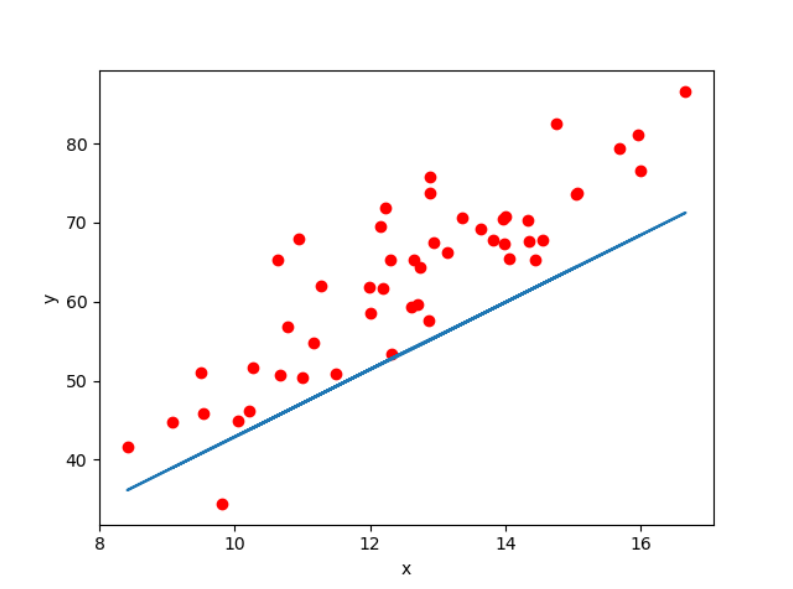
- W_data
  - Out[112]: [array([3.553465], dtype=float32)]
- b_data
  - Out[113]: [array([0.31483752], dtype=float32)]
- Loss_data
  - Out[114]: [367.4683]

Step 1(두 번째 학습)에서의 loss 값 콘솔 디버깅

- Step 1, 두 번째 학습 결과인 W, b, loss 값을 계산하여 출력한다.

```
sess.run(train)
print(step, sess.run(W), sess.run(b))
print(step, sess.run(loss))
train_set.append([sess.run(W), sess.run(b), sess.run(loss)])  ###
```





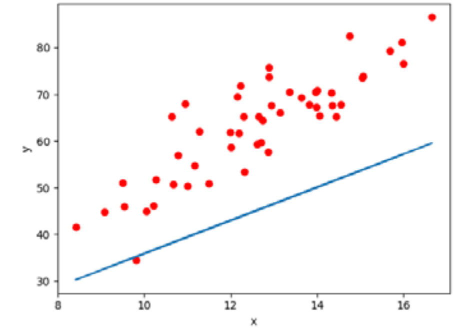
step=1
sess.run(train)
print(step, sess.run(W), sess.run(b))
1 [4.2541637] [0.3692001]
print(step, sess.run(loss))
1 118.703514

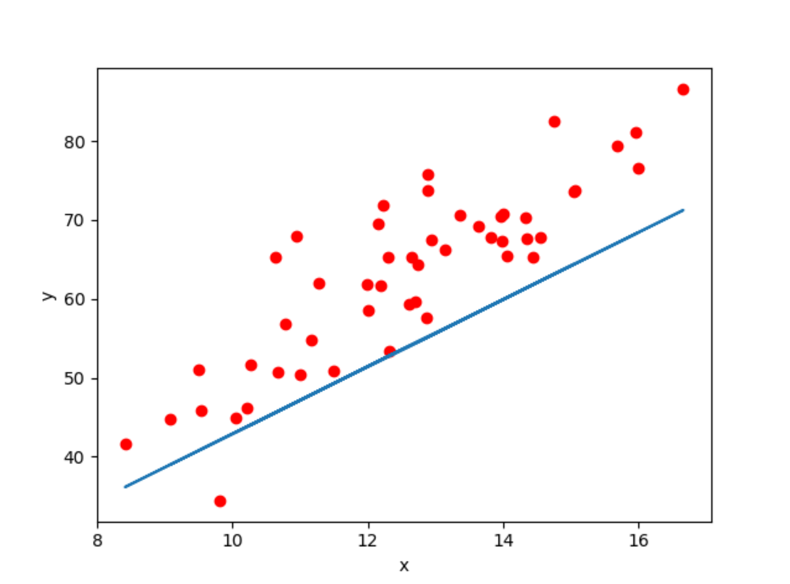
Step 1(두 번째 학습)에서의 loss 값 콘솔 디버깅

- Step 1, 두 번째 학습 결과인 W와 b 값을 이용하여, 예측 직선 (y)을 x_data, y_data 좌표 위에 출력한다.
- Step 0 번째 그림보다 직선이 더 데이터 쪽으로 움직였다.

```
plt.figure(step)
plt.plot(x_data, y_data, 'ro')
plt.plot(x_data, sess.run(W) * x_data + sess.run(b))
plt.xlabel('x')
plt.ylabel('y')
#plt.legend()
plt.show()
```

```
step=1
sess.run(train)
print(step, sess.run(W), sess.run(b))
1 [4.2541637] [0.3692001]
print(step, sess.run(loss))
1 118.703514
```





Step 1(두 번째 학습)에서의 loss 값 콘솔 디버깅

- y=x_data*W+b
  - <tf.Tensor 'add_2:0' shape=(50,) dtype=float32>
- sess.run(y)를 수행하면, y 그래프 값을 확인할 수 있다
- y=sess.run(y)를 수행하면, y 변수에 y값을 저장한다.

Step 0의 y값

```
array([42.92975 , 40.00953 , 46.08135 , 45.447483, 49.918053, 51.624714,
    34.05071 , 52.764053, 57.15245 , 42.98017 , 51.312218, 57.04682 ,
    43.72928 , 41.14911 , 43.657482, 39.395763, 44.090137, 38.637604,
    45.117508, 46.29372 , 51.215588, 44.00339 , 50.25373 , 36.61917 ,
    38.0987  , 36.795113, 45.256386, 34.173832, 35.144196, 45.60984 ,
    53.855278, 46.03917 , 40.39592 , 32.584106, 53.79027 , 49.385914,
    50.0337  , 47.026936, 36.007835, 52.042915, 39.187145, 48.742886,
    46.10148 , 30.20313 , 43.508713, 47.80955 , 59.52558 , 56.052826,
    38.24937 , 50.064648], dtype=float32)
```

Step 1의 y값

```
array([51.387238, 47.89119 , 55.160294, 54.401443, 59.75355 , 61.796745,
    40.757362, 63.160744, 68.41448 , 51.447605, 61.422626, 68.288025,
    52.344425, 49.25548 , 52.258472, 47.1564  , 52.776443, 46.248737,
    54.006397, 55.414543, 61.30694 , 52.672592, 60.155422, 43.832294,
    45.603573, 44.042927, 54.17266 , 40.904762, 42.066475, 54.595814,
    64.46715 , 55.109802, 48.35377 , 39.001564, 64.38932 , 59.116478,
    59.892   , 56.29234 , 43.10041 , 62.29741 , 46.906643, 58.346653,
    55.184395, 36.151093, 52.08037 , 57.22928 , 71.25556 , 67.09802 ,
    45.78395 , 59.92905 ], dtype=float32)
```

y_data값

```
array([61.79473357, 54.82558689, 73.75787555, 59.62363436, 70.52395661, 65.27770682, 50.97732466, 82.48138976,
76.54013741, 58.6107108 , 67.70799496, 81.19721042, 71.89775118, 50.85643241, 61.63548618, 50.36751265,
53.36790676, 56.89546287, 59.35410282, 67.55571944, 70.3393151 , 65.32321523, 65.39696349, 46.13651683,
65.26070163, 51.63389899, 65.30043166, 45.88355539, 34.43267867, 64.38310687, 73.81432409, 57.68828918,
62.03062059, 44.712155  , 73.61541869,  67.73104031, 67.36031172, 66.17119153, 44.8895761 , 67.79934904,
 67.9382822 , 69.26791715, 75.71630455, 41.59390346, 69.45439645, 70.57350672, 86.63821292, 79.33710907,
50.68514176, 70.85244846])
```

y_data=np.array(y_data)

Step 1(두 번째 학습)에서의 loss 값 콘솔 디버깅

- tf.square(y - y_data)을 수행한다.
  - Out[97]: <tf.Tensor 'Square_5:0' shape=(50,) dtype=float64>
- tf.square(y - y_data)의 결과값을 확인하려면 **sess.run(tf.square(y - y_data))** 을 수행한다.

Step 0의 y-y_data값

```
array([3.55887660e+02, 2.19515568e+02, 7.65990154e+02, 2.00963266e+02,
    4.24603277e+02, 1.86404216e+02, 2.86510325e+02, 8.83120084e+02,
    3.75882401e+02, 2.44313768e+02, 2.68821511e+02, 5.83241280e+02,
    7.93462849e+02, 9.42321300e+01, 3.23208629e+02, 1.20379282e+02,
    8.60770028e+01, 3.33349419e+02, 2.02680634e+02, 4.52072610e+02,
    3.65716953e+02, 4.54534894e+02, 2.29317497e+02, 9.05798690e+01,
    7.37774252e+02, 2.20189581e+02, 4.01763774e+02, 1.37117623e+02,
    5.06256283e-01, 3.52435534e+02, 3.98363520e+02, 1.35701994e+02,
    4.68060278e+02, 1.47089562e+02, 3.93036451e+02, 3.36543665e+02,
    3.00211507e+02, 3.66502536e+02, 7.88853182e+01, 2.48265203e+02,
    8.26627877e+02, 4.21276920e+02, 8.77037917e+02, 1.29749703e+02,
    6.73178502e+02, 5.18197669e+02, 7.35094790e+02, 5.42157841e+02,
    1.54648405e+02, 4.32132662e+02])
```

Step 1의 y-y_data값

```
array([1.08315973e+02, 4.80858661e+01, 3.45870055e+02, 2.72712776e+01,
    1.16001627e+02, 1.21170931e+01, 1.04447629e+02, 3.73287364e+02,
    6.60262740e+01, 5.13100828e+01, 3.95058567e+01, 1.66647071e+02,
    3.82332557e+02, 2.56304312e+00, 8.79283866e+01, 1.03112524e+01,
    3.49828809e-01, 1.13352765e+02, 2.85979549e+01, 1.47408162e+02,
    8.15838329e+01, 1.60038264e+02, 2.74737550e+01, 5.30944073e+00,
    3.86402712e+02, 5.76228590e+01, 1.23827284e+02, 2.47883809e+01,
    5.82748451e+01, 9.57911065e+01, 8.73697041e+01, 6.64859488e+00,
    1.87056209e+02, 3.26108493e+01, 8.51208901e+01, 7.42106844e+01,
    5.57757055e+01, 9.75917209e+01, 3.20111370e+00, 3.02713436e+01,
    4.42329851e+02, 1.19274011e+02, 4.21559316e+02, 2.96241908e+01,
    3.01856731e+02, 1.78068425e+02, 2.36625955e+02, 1.49795241e+02,
    2.40216727e+01, 1.19320624e+02])
```

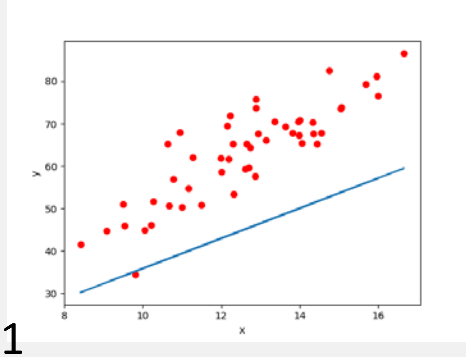
Step 1(두 번째 학습)에서의 loss 값 콘솔 디버깅

- tf.reduce_mean(tf.square(y - y_data))을 수행한다.
- tf.reduce_mean(tf.square(y - y_data))의 결과값을 확인하려면 **sess.run(tf.reduce_mean(tf.square(y - y_data)))** 을 수행한다.
- sess.run(tf.reduce_mean(tf.square(y - y_data)))
  - Out[106]: **367.46829244768355** Step 0의 tf.reduce_mean(tf.square(y - y_data)) 값
  - Out[131]: **118.70350861527437** Step 1의 tf.reduce_mean(tf.square(y - y_data)) 값
- Step 1 에서의 loss 값을 출력값을 콘솔 디버깅으로 확인했다.
  - print(step, sess.run(loss))
  - 0 367.4683
  - 1 118.703514
  - loss 값은 118.70 이다.

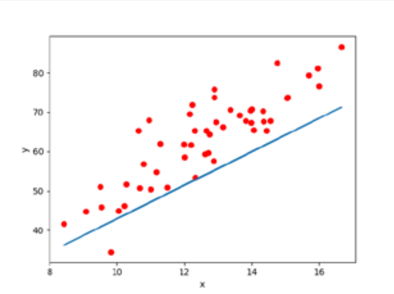
Step 2(세 번째 학습)에서의 loss 값 콘솔 디버깅

- Step 2, 세 번째 학습 결과인 W, b, loss 값을 계산하여 출력한다.

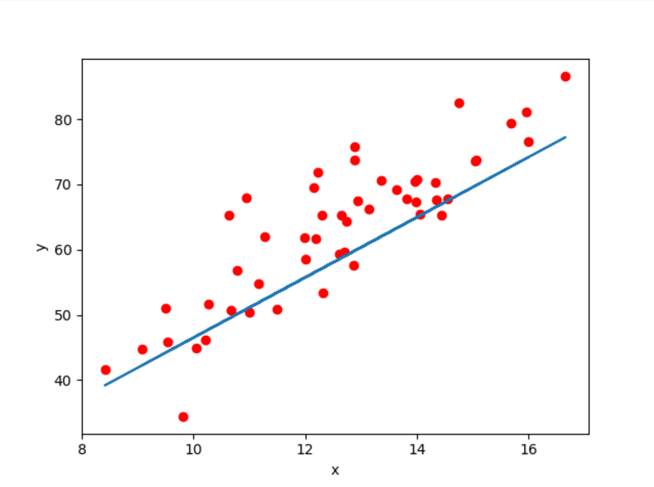
```
sess.run(train)
print(step, sess.run(W), sess.run(b))
print(step, sess.run(loss))
train_set.append([sess.run(W), sess.run(b), sess.run(loss)])   ###
```



step=1
sess.run(train)
print(step, sess.run(W), sess.run(b))
1 [4.2541637] [0.3692001]
print(step, sess.run(loss))
1 118.703514



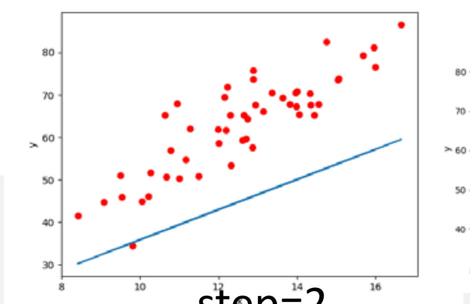
step=2
sess.run(train)
print(step, sess.run(W), sess.run(b))
2 [4.6121655] [0.39695176]
print(step, sess.run(loss))
2 53.766617

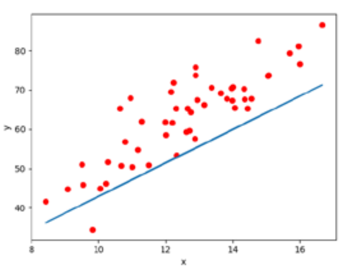


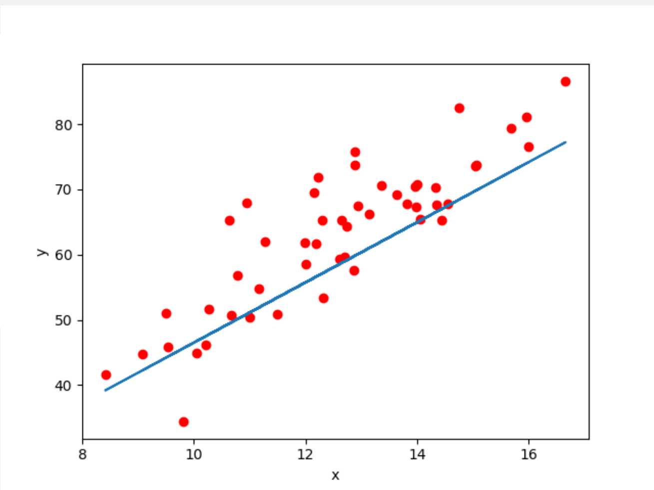
Step 2(세 번째 학습)에서의 loss 값 콘솔 디버깅

- Step 2번째 학습 결과인 W와 b 값을 이용하여, 예측 직선 (y)을 x_data, y_data 좌표 위에 출력한다.
- Step 1 번째 그림보다 직선이 더 데이터 쪽으로 움직였다.

```
plt.figure(step)
plt.plot(x_data, y_data, 'ro')
plt.plot(x_data, sess.run(W) * x_data + sess.run(b))
plt.xlabel('x')
plt.ylabel('y')
#plt.legend()
plt.show()
```





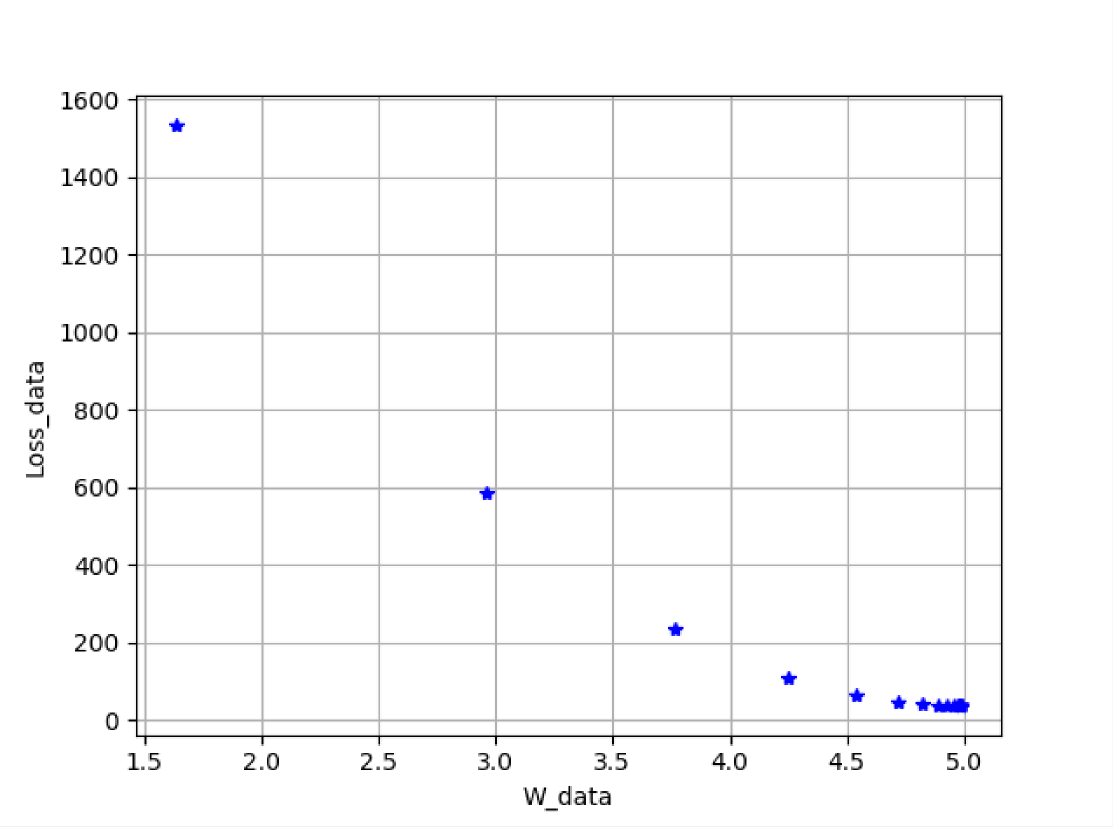


step=1
sess.run(train)
print(step, sess.run(W), sess.run(b))
1 [4.2541637] [0.3692001]
print(step, sess.run(loss))
1 118.703514

step=2
sess.run(train)
print(step, sess.run(W), sess.run(b))
2 [4.6121655] [0.39695176]
print(step, sess.run(loss))
2 53.766617

W 에 따른 loss 데이터 시각화

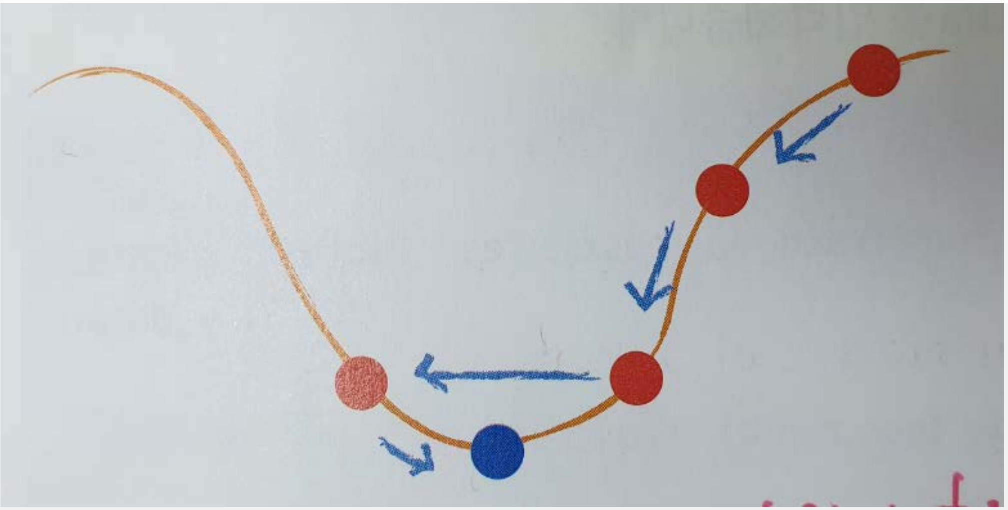
- Step 0(첫 번째 학습), Step 1(두 번째 학습), Step 2(세 번째 학습)에서의 W 에 따른 loss 값의 변화를 살펴보자
- **학습을 거듭할 때마다 다른 W값에 따라 loss 값이 줄어든다.**



경사하강법(Gradient Descent) 최적화 메소드

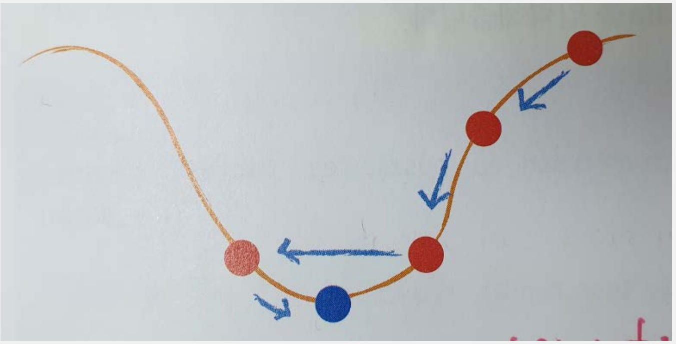
- Step 0(첫 번째 학습), Step 1(두 번째 학습), Step 2(세 번째 학습)에서의 W에 따른 loss 값의 변화를 살펴본 결과, **학습을 거듭할 때마다 다른 W값에 따라 loss 값이 줄어듦을 확인하였다.**
- **이는 텐서플로가 제공하는 경사하강법(Gradient Descent) 최적화 메소드가 손실, loss값을 최소화해주기 때문이다.**

```
optimizer=tf.train.GradientDescentOptimizer(learning_rate=0.1)
train_op=optimizer.minimize(cost)
```



최적화(Optimization)와 경사하강법

- 최적화 함수란 가중치(W)와 편향(b) 값을 변경해가면서 손실 loss값을 최소화하는 가장 최적화된 가중치(W)와 편향(b)값을 찾아주는 함수이다
- **가중치(W)와 편향(b)값을 무작위로 변경하면 시간이 너무 오래 걸리고, 학습 시간도 예측하기 어렵다**
- **빠르게 최적화하기 위한 방법 중의 하나가 경사하강법이다.**
- **경사하강법은 최적화 방법 중 가장 기본적인 알고리즘으로 음의 경사 방향으로 계속 이동하면서 최적의 값을 찾아 나가는 방법이다**

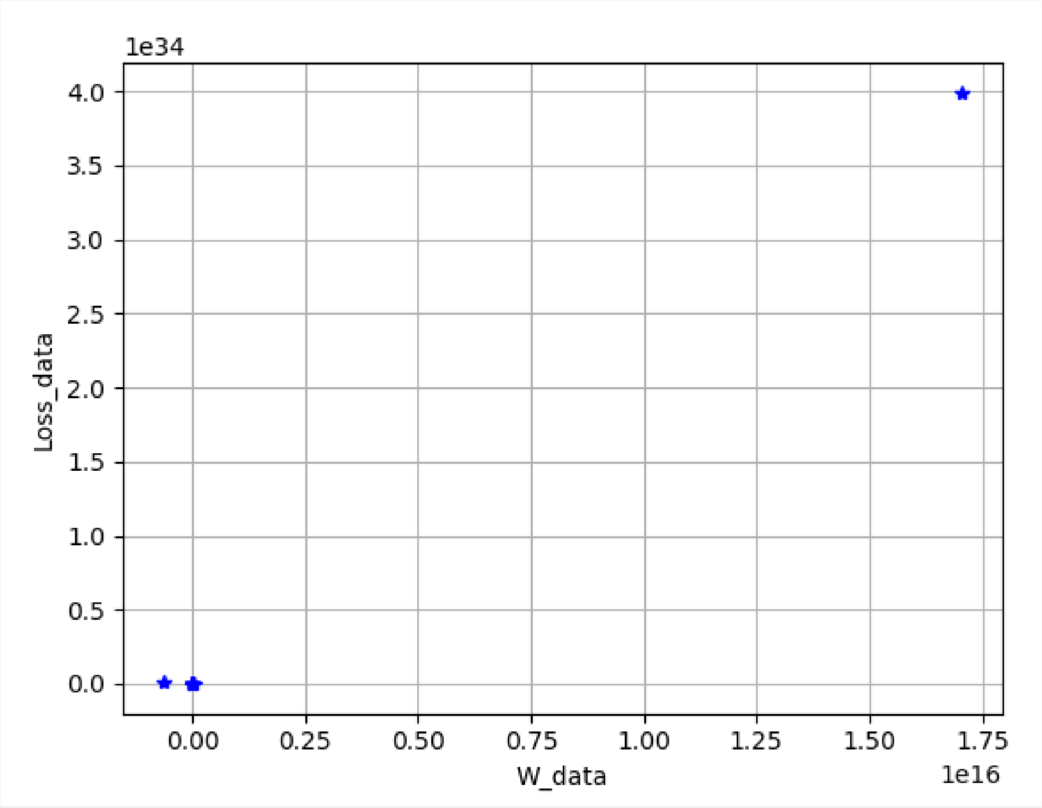


학습률: 하이퍼파라미터(hyperparameter)

- 학습률은 학습을 얼마나 급하게 할 것인가를 설정하는 값이다.
- **학습률이 너무 크면 최적의 손실값을 찾지 못하고 지나치게 되고, 값이 너무 작으면 학습 속도가 매우 느려진다.**
- 학습 진행에 영향을 주는 변수를 하이퍼파라미터(hyperparameter)라 하면, 이 값에 따라 학습 속도나 신경망 성능이 크게 달라진다.
- **머신러닝에서는 학습률, 하이퍼파라미터를 잘 튜닝하는 것이 큰 과제이다.**

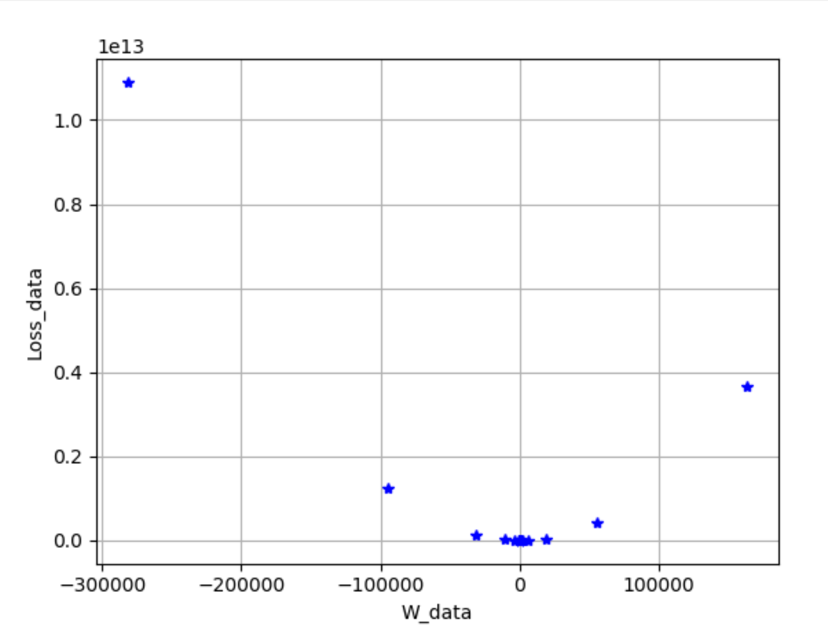
경사하강법의 Learning Rate(학습률)와 손실함수

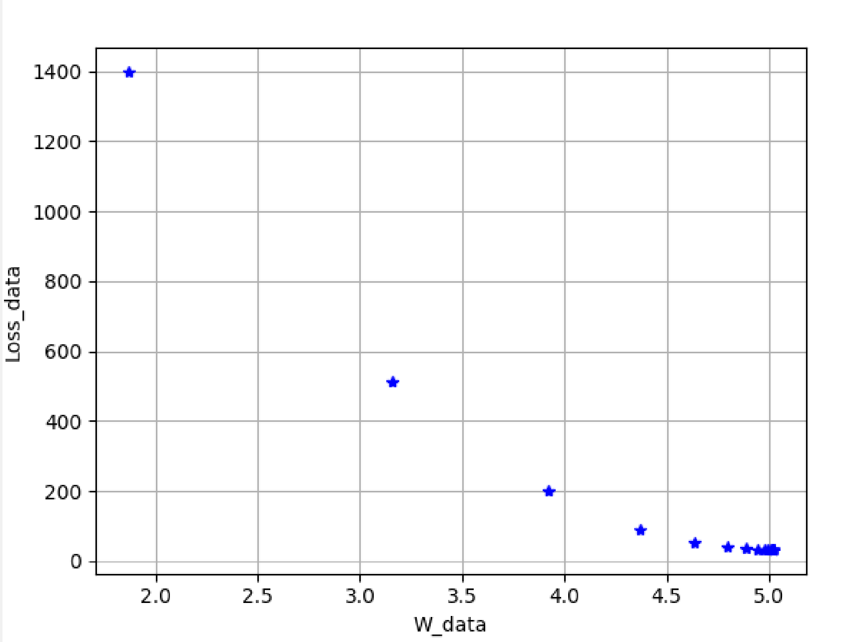
- **경사하강법에서 음의 경사로 계속 이동할 때의 이동 간격을 학습률이라 한다**
- 본 예제에서 학습률이 0.1일 때, 손실함수가 0이 되는 기울기를 찾기가 매우 어렵다



학습률 0.01, 0.0015일 때 손실함수의 그래프

- 학습률이 0.01일 때는 손실함수가 0이 되는 기울기를 찾기가 매우 어렵다
- 학습률이 0.0015일 때 손실함수가 0이 되는 기울기를 찾기가 매우 쉽다.





학습 수행

- 선형회귀모델을 다 만들었으니, 그래프를 실행해 학습을 시키고, 결과를 확인하자
- 파이썬 with 기능을 이용해 세션 블록을 만들고, 세션 종료를 자동으로 처리하자
- 최적화를 수행하는 그래프인 train_op를 실행하고, 실행 시마다 변화하는 손실값을 출력한다
- 학습은 100번 수행하며, feed_dict 매개변수를 통해, 상관관계를 알아내고자 하는 데이터인 x_data와 y_data를 입력한다.

학습 수행

- 최적화가 완료된 모델에 테스트 값을 넣고 결과가 잘 나오는지 확인해봅니다.

```
with tf.Session() as sess:

    sess.run(tf.global_variables_initializer())

    for step in range(100):
        _, cost_val=sess.run([train_op, cost], feed_dict={X:
x_data, Y: y_data})

        print(step, cost_val, sess.run(W), sess.run(b))

    print("\n=== Test ===")
    print("X: 5, Y:", sess.run(hypothesis, feed_dict={X: 5}))
    print("X: 2.5, Y:", sess.run(hypothesis, feed_dict={X:
2.5}))
```

학습 진행 상황 출력 (손실값과 변수들의 변화 확인)

- 스텝, 손실값, [W 값], [b 값]
- 0 0.869386 [0.9130715] [0.3043009]
- 1 0.02205333 [0.87248445] [0.27821213]
- 2 0.011377501 [0.88021415] [0.27357593]
- 3 0.010722056 [0.8825839] [0.26677507]
- 4 0.010211366 [0.8854622] [0.2603865]
- 5 0.009726319 [0.8882096] [0.2541243]
- 6 0.009264297 [0.8908976] [0.24801561]
- 7 0.008824232 [0.89352024] [0.24205346]
- 8 0.008405092 [0.89607996] [0.23623468]
- 9 0.008005846 [0.8985781] [0.23055576]
- 10 0.00762555 [0.90101624] [0.22501338]

학습 테스트

- 스텝, 손실값, [W 값], [b 값]
- 99 0.000100282195 [0.98864883] [0.02580387]
- 학습에 의해 x=[1,2,3], y=[1,2,3]을 만드는 W 값은 1에 가까운 0.988이 정해졌고, b 값은 0에 가까운 0.025가 정해졌다.
- Test에서는 정해진 [W 값], [b 값] 을 가지고, X: 5가 들어오면 Y 값을 예측하는 데, Y: [4.969048]로 예측하였다. (5에 가까움)
- X: 2.5가 들어오면 Y 값을 예측하는 데, Y: [2.4974258]로 예측하였다. (2.5에 가까움)
- === Test ===
- X: 5, Y: [4.969048]
- X: 2.5, Y: [2.4974258]