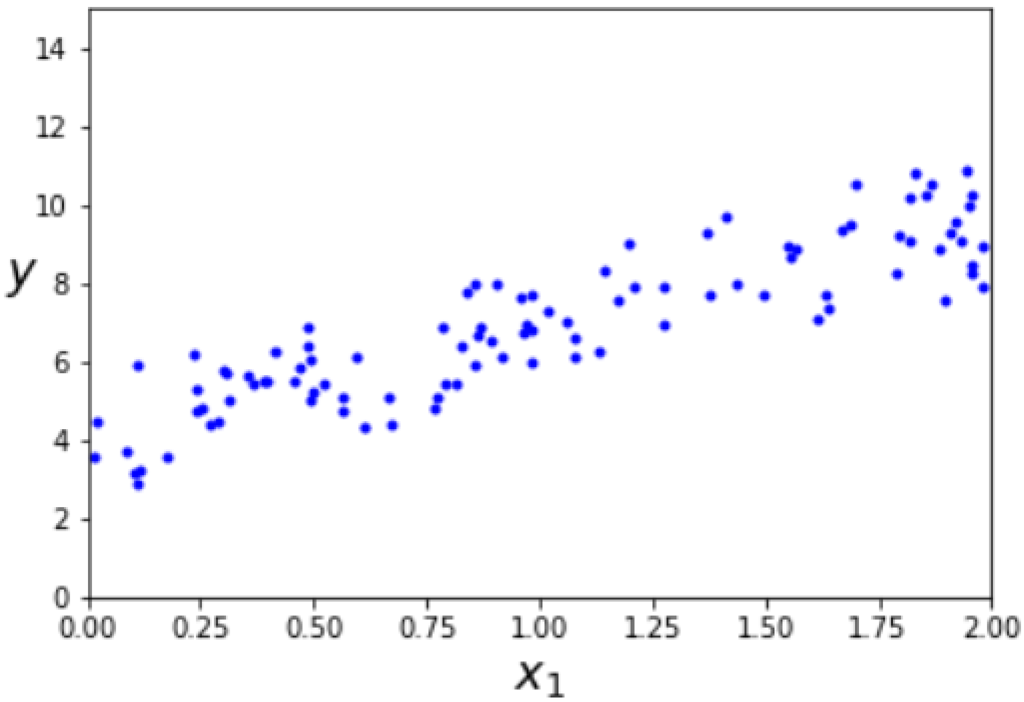
인공지능 과제1

20130940 장용훈

1-1 데이터셋 생성

```
import numpy as np
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
#x와 y의 데이터셋 생성
x = 2*np.random.rand(100,1)
y = 4 + 3 *x + np.random.randn(100,1)
plt.plot(x,y,"b. ")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation = 0, fontsize =18)
plt.axis([0,2,0,15])
#1. 데이터셋의 화면출력
plt.show()
```



1-2 정규방정식 활용(비용 함수를 최소화 하기위한 방법)

```
x_b = np.c_[np.ones((100,1)),x]#x_b 에 [1,x]형태의 열을 추가해줌(r_행)
#비용함수를 최소화 하기 위한 정규방정식 활용 inv로  역행렬을 구하고 dot으로 행렬곱셈
theta_best = np.linalg.inv(x_b.T.dot(x_b)).dot(x_b.T).dot(y)
print(theta_best)
```

```
[[ 4.07351305]
 [ 2.83016319]]
```

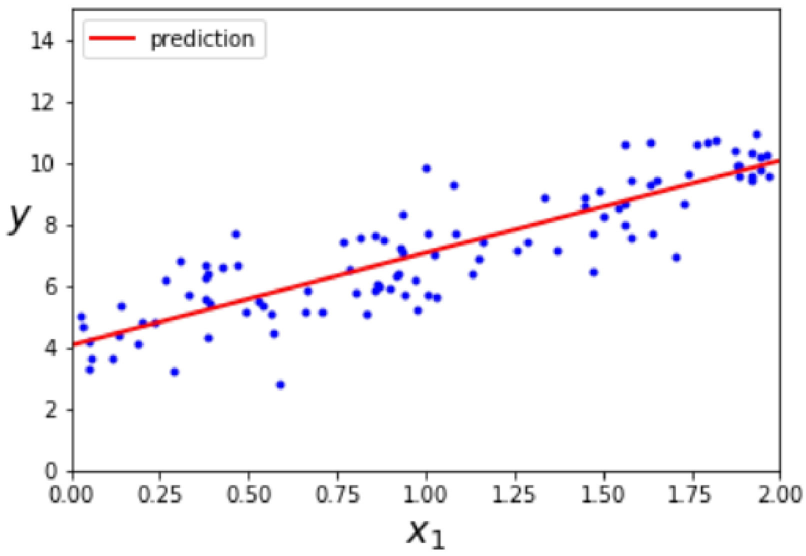
1-3 y_predict 값 생성

```
x_new = np.array([[0], [2]])
x_new_b = np.c_[np.ones((2, 1)), x_new] #모든 샘플에 x0=1 을 추가
y_predict = x_new_b.dot(theta_best)
print(y_predict)
```

```
[[ 4.07351305]
 [ 9.73383943]]
```

1-4 출력확인

```
#기존 데이터셋이 있는 plot 에 예측한 모형을 출력
plt.plot(x_new,y_predict,"r-",linewidth=2,label="prediction")
plt.legend(loc="upper left")
plt.show()
```



1-5 같은 문제를 sklearn 의 LinearReggression활용

```python
from sklearn.linear_model import LinearRegression
#사이킷 런을 활용하여 같은 문제를 다른 방식으로 코딩
lin_reg = LinearRegression()
lin_reg.fit(x,y)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

1-6 intercept와 coef 출력 확인

```python
#가중치와 편향 값
print(lin_reg.intercept_,  lin_reg.coef_)
```

```
[ 3.90228591] [[ 3.23193604]]
```

1-7,8 theta를 출력

```python
#최적의 theta값 출력 최소제곱법을 이용 한다.
theta_best_svd, residuals, rank ,s = np.linalg.lstsq(x_b,y, rcond=1e-6)
print(theta_best_svd)
print(np.linalg.pinv(x_b).dot(y))
```

```
[[ 3.90228591]
 [ 3.23193604]]
[[ 3.90228591]
 [ 3.23193604]]
```

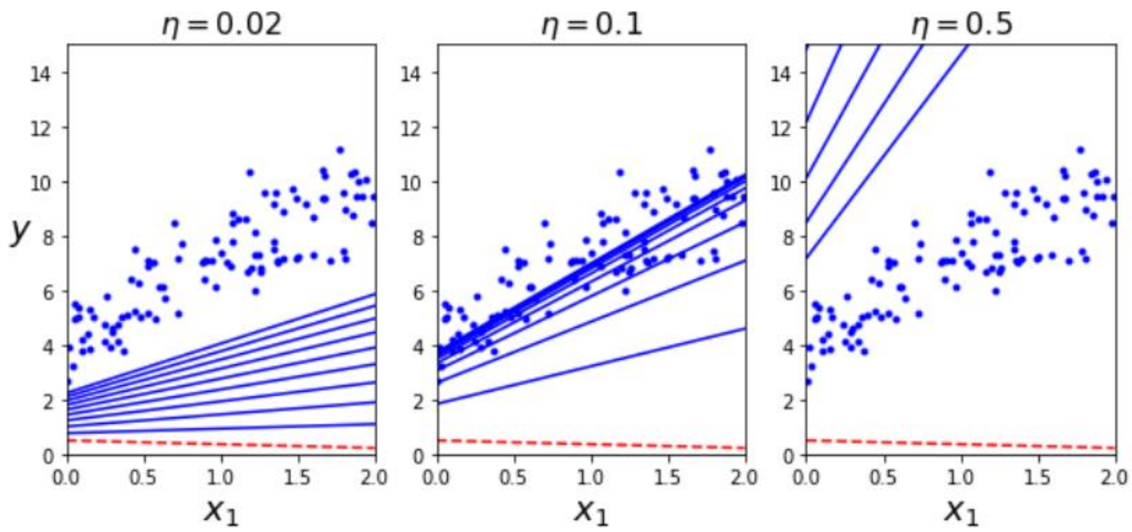
1-9,10 경사하강법으로 선형회귀

```python
eta = 0.1 #학습률
n_iterations = 1000 #반복횟수
m = 100
theta = np.random.randn(2,1) #랜덤하게 초기화
for iteration in range(n_iterations):
    gradients = 2/m * x_b.T.dot(x_b.dot(theta)-y)
    theta = theta-eta * gradients
print(theta,"\n")#정규방정식으로 찾은 theta 와 값이 동일
print(x_new_b.dot(theta)) #테스트 셋에 대한 예측 결과
```

```
[[ 4.29417216]
 [ 2.79842459]]

[[ 4.29417216]
 [ 9.89102134]]
```

1-11 여러가지 학습률에 따른 비교

```python
theta_path_bgd = []
#여러가지 학습률에대한 경사 하강법
def plot_gradient_descent(theta, eta, theta_path=None):
    m = len(x_b)
    plt.plot(x, y, "b.")
    n_iterations = 1000
    for iteration in range(n_iterations):
        if iteration < 10:
            y_predict = x_new_b.dot(theta)
            style = "b-" if iteration > 0 else "r--"
            plt.plot(x_new, y_predict, style)
        gradients = 2/m * x_b.T.dot(x_b.dot(theta) - y)
        theta = theta - eta * gradients
        if theta_path is not None:
            theta_path.append(theta)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 2, 0, 15])
    plt.title(r"$\eta = {}$".format(eta), fontsize=16)

np.random.seed(42)
theta = np.random.randn(2,1)  # random initialization

plt.figure(figsize=(10,4))
plt.subplot(131); plot_gradient_descent(theta, eta=0.02)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(132); plot_gradient_descent(theta, eta=0.1, theta_path=theta_path_bgd)
plt.subplot(133); plot_gradient_descent(theta, eta=0.5)
plt.show()
```

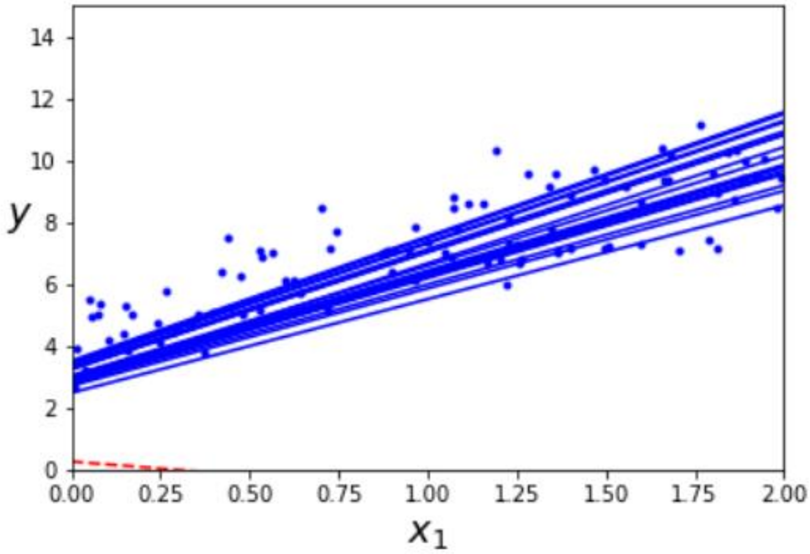


학습률에따라 예측의 변화에 대한 양상이 조금씩 다르다. 제일 왼쪽은 학습률이 최적치를 찾아가는데 시간이 오래걸렸다. 가장 오른쪽의 학습률은 학습률이 너무 높아서 문제이다. 가운데의 학습률은 적당한 학습률로 몇번 만에 최적치를 찾는 모습이다.

1-12,13 스톤캐스틱 경사하강법

```
theta_path_sgd = []
n_iterations = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

np.random.seed(43)
theta = np.random.randn(2,1)  # random initialization
#학습스케줄 _ 매 반복에서 학습률을 결정해 준다.
def learning_schedule(t):
    return t0 / (t + t1)

m = len(x_b)
#무작위 샘플링으로 gradient를 구해준다. 무작위기 때문에 지역값이 갖히지 않는다.
#뿐만아니라 전체 샘플에대해 진행이 되지 않고, 몇개의 샘플을 확인 하기 때문에 속도가 더 빠르다.
for epoch in range(n_iterations):
    for i in range(m):
        if epoch == 0 and i < 20:
            y_predict = x_new_b.dot(theta)
            style = "b-" if i > 0 else "r--"
            plt.plot(x_new, y_predict, style)
        random_index = np.random.randint(m)
        xi = x_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
        theta_path_sgd.append(theta)
```

```python
plt.plot(x, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
print(theta)
```




```
[[ 4.29289682]
 [ 2.74421779]]
```

위에서 스톤캐스틱의 흐름에 대한 양상을 보여준다. 뿐만 아니라 최적치일 때의 theta값을 출력했다. 앞서 다른 방식으로 구한값과 오차가 그게 다르지 않음이 보인다.

1-14 SGD REGRESSOR활용

```python
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(x, y.ravel())
```

```
SGDRegressor(alpha=0.0001, average=False, epsilon=0.1, eta0=0.1,
       fit_intercept=True, l1_ratio=0.15, learning_rate='invscaling',
       loss='squared_loss', n_iter=50, penalty=None, power_t=0.25,
       random_state=None, shuffle=True, verbose=0, warm_start=False)
```

반복은 50번으로 그리고 이때 학습률은 0.1로 설정을 해준 모습이다.

1-15 intercept 와 coef 출력

```
sgd_reg.intercept_, sgd_reg.coef_
```

```
(array([ 4.34531949]), array([ 2.85419327]))
```

역시 비슷한 bias 와 weight 을 구했음을 알 수있다.

1-16 미니배치 경사법

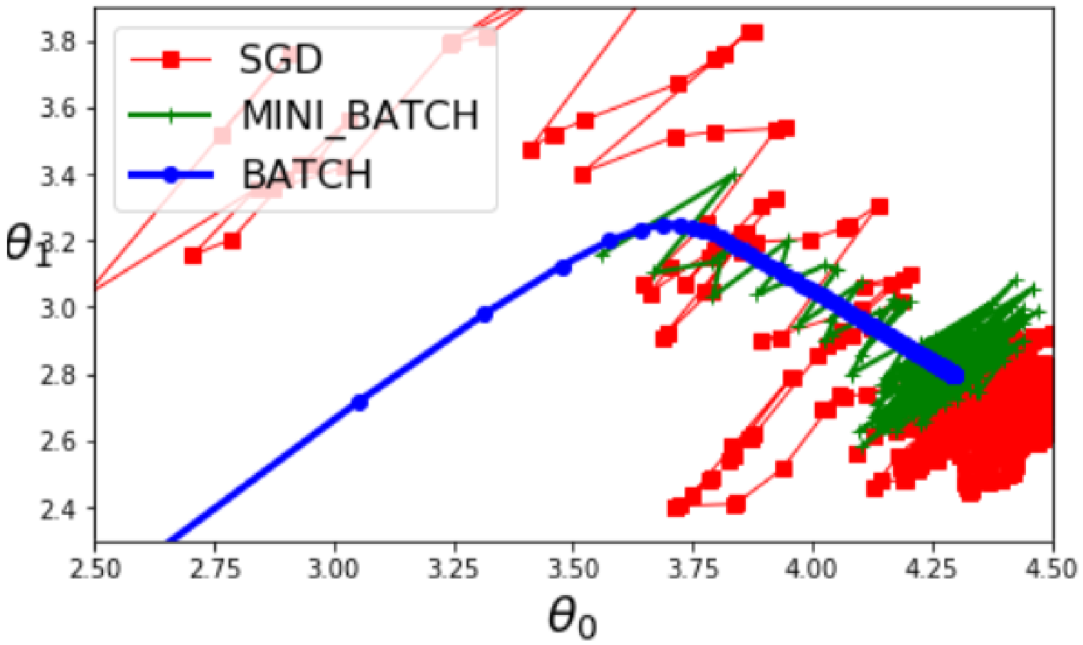
```
theta_path_mgd = []
n_iterations = 50
minibatch_size = 20
np.random.seed(43)
theta = np.random.randn(2,1)

t0,t1 = 200, 1000
def learning_schedule(t):
    return t0 / (t+t1)
t = 0
for epoch in range(n_iterations):
    shuffled_indices = np.random.permutation(m)
    x_b_shuffled = x_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    for i in range(0, m , minibatch_size):
        t += 1
        xi = x_b_shuffled[i:i+minibatch_size]
        yi = y_shuffled[i:i+minibatch_size]
        gradients =2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(t)
        theta = theta - eta * gradients
        theta_path_mgd.append(theta)
print(theta)
```

```
[[ 4.34861295]
 [ 2.85507338]]
```

임의의 작은 샘플을 기준으로 배치알고리즘을 진행 한다. 그렇기에 배치 알고리즘에 비하여 속도가 뛰어나다.

1-17 세 알고리즘의 비교

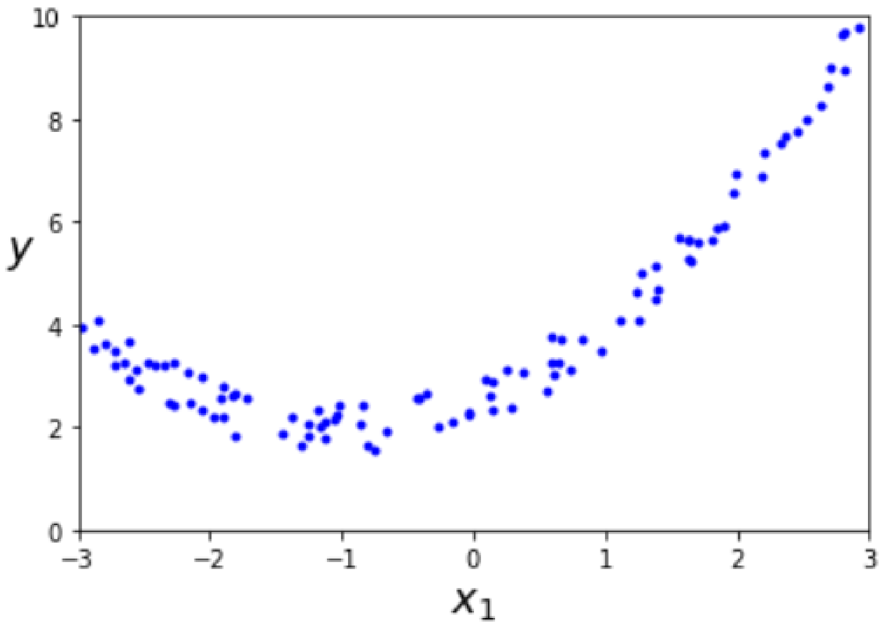
```
#배치 , 스톤캐스틱, 미니배치의 theta를 저장한다. 이때, 각각의 랜덤으로 생성한 수들은 동일한 seed를 사용했기에 데이터셋은 동일하다.
theta_path_bgd = np.array(theta_path_bgd)
theta_path_sgd = np.array(theta_path_sgd)
theta_path_mgd = np.array(theta_path_mgd)
plt.figure(figsize=(7,4))
plt.plot(theta_path_sgd[:,0], theta_path_sgd[:,1],"r-s",linewidth=1, label="SGD")
plt.plot(theta_path_mgd[:,0], theta_path_mgd[:,1],"g-+",linewidth=2, label="MINI_BATCH")
plt.plot(theta_path_bgd[:,0], theta_path_bgd[:,1],"b-o",linewidth=3, label="BATCH")
plt.legend(loc="upper left", fontsize=16)
plt.xlabel(r"$\theta_0$",fontsize=20)
plt.ylabel(r"$\theta_1$", fontsize=20, rotation=0)
plt.axis([2.5, 4.5, 2.3, 3.9])
plt.show()
```



앞서 정규방정식까지 총 4개의 알고리즘으로 선형회귀를 수행했다. 수행 값에대한 결과치는 크게 다르지는 않았지만, 걸리는 시간의 차이는 있었다. 특히 배치 알고리즘의 경우에서 시간이 가장 오래걸렸다.

2-1 다차항회귀

```python
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
np.random.seed(42)
m = 100
x = 6 * np.random.rand(m,1)-3
#2차 방정식으로 비선형 데이터 생성
y = 0.5 * x**2 + x + 2 + np.random.rand(m,1)
plt.plot(x,y,"b.")
plt.xlabel("$x_1$",fontsize=18)
plt.ylabel("$y$",rotation=0,fontsize=18)
plt.axis([-3,3,0,10])
plt.show()
```



2-2, 3 x,x_poly 출력

```python
#PolynomialFeatures을 이용하여 훈련데이터 변화. 훈련데이터의 각 특성을 제곱하여 추가한다.
poly_features = PolynomialFeatures(degree=2, include_bias=False)
x_poly = poly_features.fit_transform(x)
print(x[0])
print(x_poly[0])
```

```
[-0.75275929]
[-0.75275929  0.56664654]
```

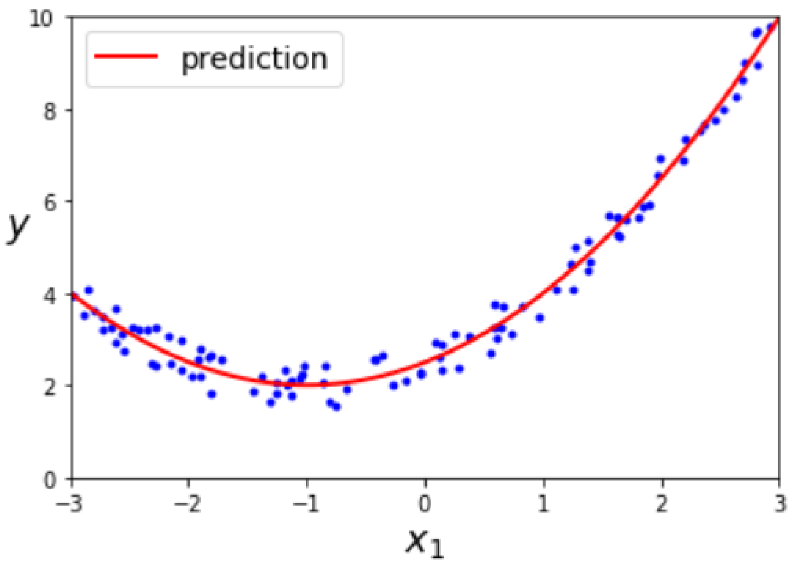
2-4 LinearGression 수행

```
#앞서 만든 데이터 셋에 대한 선형회귀
lin_reg = LinearRegression()
lin_reg.fit(x_poly,y)
print(lin_reg.intercept_, lin_reg.coef_)
```

```
[ 2.49786712] [[ 0.9943591   0.49967213]]
```

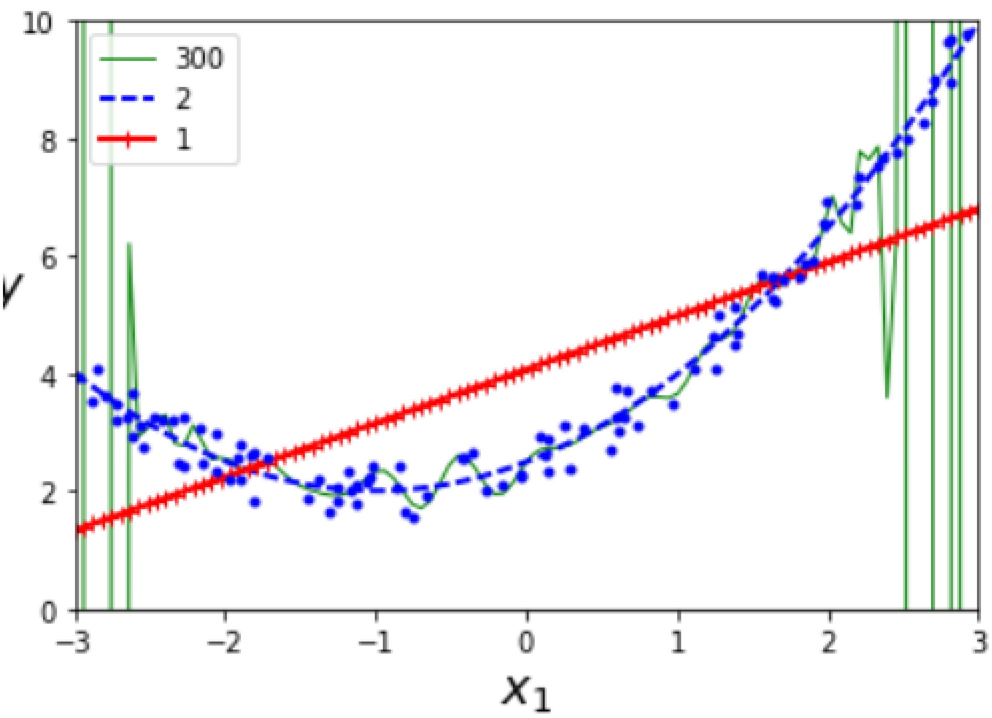
2-5 예측결과와 데이터셋을 표현

```
x_new = np.linspace(-3,3,100).reshape(100,1)
x_new_poly = poly_features.transform(x_new)
y_new = lin_reg.predict(x_new_poly)
plt.plot(x,y,"b.")
plt.plot(x_new,y_new,"r-",linewidth=2,label="prediction")
plt.xlabel("$x_1$",fontsize=18)
plt.ylabel("$y$",rotation=0,fontsize=18)
plt.legend(loc="upper left",fontsize=14)
plt.axis([-3,3,0,10])
plt.show()
```



2-6 출력확인

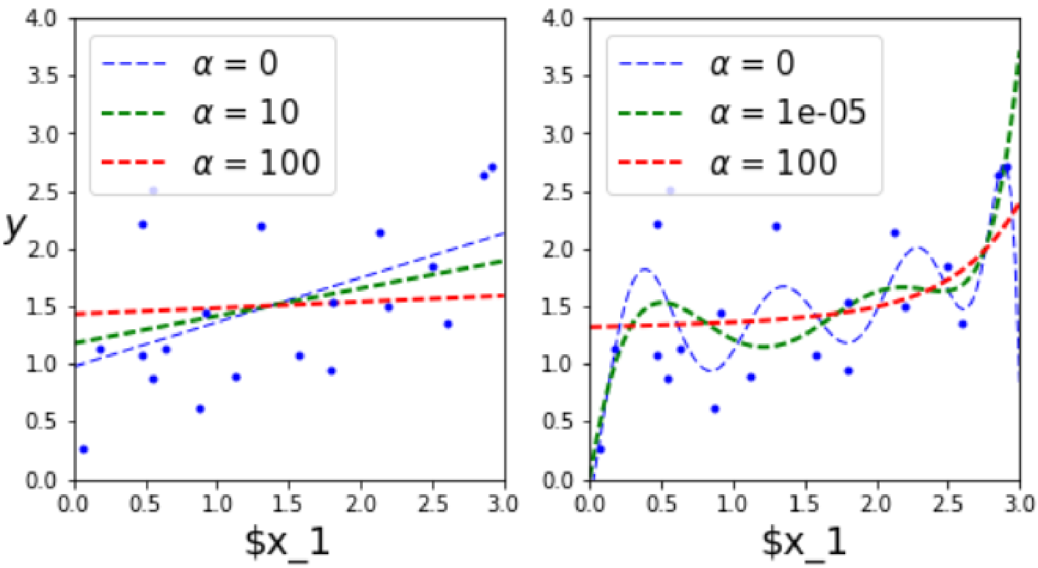
```
for style, width, degree in (("g-",1,300),("b--",2,2),("r-+",2,1)):
    polybig_features = PolynomialFeatures(degree=degree,include_bias=Fals
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline([
        ("poly_features",polybig_features),
        ("std_scaler",std_scaler),
        ("lin_reg",lin_reg)
    ])
    polynomial_regression.fit(x,y)
    y_newbig = polynomial_regression.predict(x_new)
    plt.plot(x_new,y_newbig,style,label=str(degree), linewidth=width)
plt.plot(x,y,"b.",linewidth=3)
plt.legend(loc="upper left")
plt.xlabel("$x_1$",fontsize=18)
plt.ylabel("$y$",rotation=0,fontsize=18)
plt.axis([-3,3,0,10])
plt.show()
```



차수가 1,2, 300 에따른 회귀 모델을 보여준다. 선형회귀는 데이터셋을 잘 예측 하지 못함으로 언더피팅이다. 차수가 300인 경우는 심하게 오버피팅 되어있다. 2차인 경우는 결과를 잘 맞춘 적절한 값이다.

3. 규제가 추가된 선형회귀

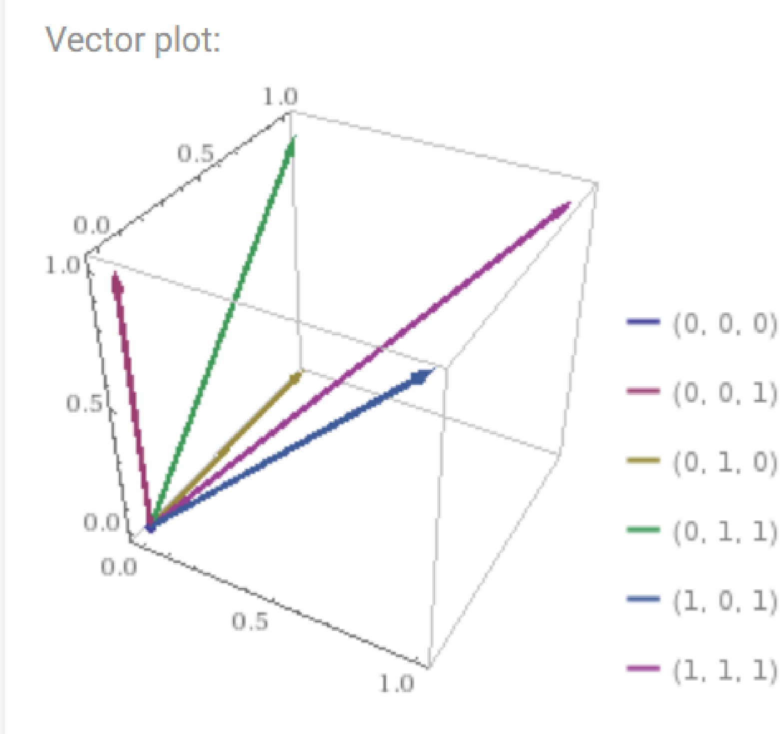
```
np.random.seed(42)
m =20
x = 3 * np.random.rand(m,1)
y = 1+ 0.5 *x +np.random.randn(m,1) /1.5
x_new = np.linspace(0,3,100).reshape(100,1)
def plot_model(model_class,polynomial,alphas,**model_kargs):
    for alpha, style in zip(alphas,("b--","g--","r--")):
        model = model_class(alpha, **model_kargs) if alpha >0 else LinearRegression()
        if polynomial:
            model = Pipeline([("poly_features", PolynomialFeatures(degree=10, include_bias=Fal
                             ("std_scaler",StandardScaler()),
                             ("regul_reg",model)
                             ])
        model.fit(x,y)
        y_new_regul = model.predict(x_new)
        lw = 2 if alpha > 0 else 1
        plt.plot(x_new,y_new_regul,style,linewidth=lw,label=r"$\alpha$ = {}".format(alpha))
    plt.plot(x,y,"b.",linewidth=3)
    plt.legend(loc="upper left",fontsize=15)
    plt.xlabel("$x_1",fontsize=18)
    plt.axis([0,3,0,4])
plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Ridge,polynomial=False,alphas=(0,10,100),random_state=42)
plt.ylabel("$y$",rotation=0,fontsize=18)
plt.subplot(122)
plot_model(Ridge,polynomial=True,alphas=(0,10**-5,100),random_state=42)
plt.show()
```



이때 모델에 alpha를 얼마나 할당하는지에 따라 가중치가 달라진다. 만약 alpha가 0이라면 일반적인 선형회귀와 동일해진다. Alpha가 아주 크면 가중치가 0에 가까워 지고, 데이터의 평균을 지나는 직선에 가까워진다. 뿐만 아니라 왼쪽과 오른 쪽 그림에서 의 차이는 polynomial 의 true false의 유무에 달려있다. 왼쪽은 일반적인 릿지 모델이지만, 오른쪽은 polynomialFeature(degree=10)을 사용하여 데이터를 확장하고, StandardScaler를 사용하여 스케일을 조정한후 릿지 회귀 를 수행 하였다. 즉 데이터의 스케일을 조정하여 나타낸 것이다. 덧 붙이자면 true일경우는 다항회귀를 수행한 것이고, false라면 일반적인 선형회귀를 진행했기에 이러한

모양의 차이가 나타난다.

4. 선형분리



선형분리가 불가능하다. 그림으로 그렸을 때 직관적으로 선형분리가 불가능이 보일 뿐만이 아니라, 벡터들이 선형독립이기 때문에, 동일한 공간에 속하다고 여겨 질 수 없다. 만약 선형으로 분리가 되려면 이 벡터들이 두개의 공간에 선형종속인 경우에 가능하다. 이를 분리하려면 다층으로 나누어서 분리하여야 한다.

5. 행렬에 관한 계산들

```python
A = np.array([[1,-2,3,5],[2,2,-1,0],[3,0,1,2],[1,0,2,0]])
print("A출력")
print(A,"\n")

print("A의 2A")
print(2*A,"\n")

print("A의 Transporse")
print(A.T,"\n")

print("A의 역행렬")
print(linalg.inv(A),"\n")

print("A의 계수")
print(np.linalg.matrix_rank(A),"\n")

print("A의 행렬식")
print(np.linalg.det(A),"\n")

v1, v2=np.linalg.eig(A)
print("A의 고유값 분해")
print("A의 고유값")
print(v1)

print("A의 고유벡터")
print(v2,"\n")

print("A의 특이값 분해")
print(np.linalg.svd(A, full_matrices = True))
```

```
A출력
[[ 1 -2  3  5]
 [ 2  2 -1  0]
 [ 3  0  1  2]
 [ 1  0  2  0]]

A의 2A
[[ 2 -4  6 10]
 [ 4  4 -2  0]
 [ 6  0  2  4]
 [ 2  0  4  0]]

A의 Transporse
[[ 1  2  3  1]
 [-2  2  0  0]
 [ 3 -1  1  2]
 [ 5  0  2  0]]

A의 역행렬
[[-0.23529412 -0.23529412  0.58823529 -0.05882353]
 [ 0.29411765  0.79411765 -0.73529412  0.32352941]
 [ 0.11764706  0.11764706 -0.29411765  0.52941176]
 [ 0.29411765  0.29411765 -0.23529412 -0.17647059]]

A의 계수
4
```

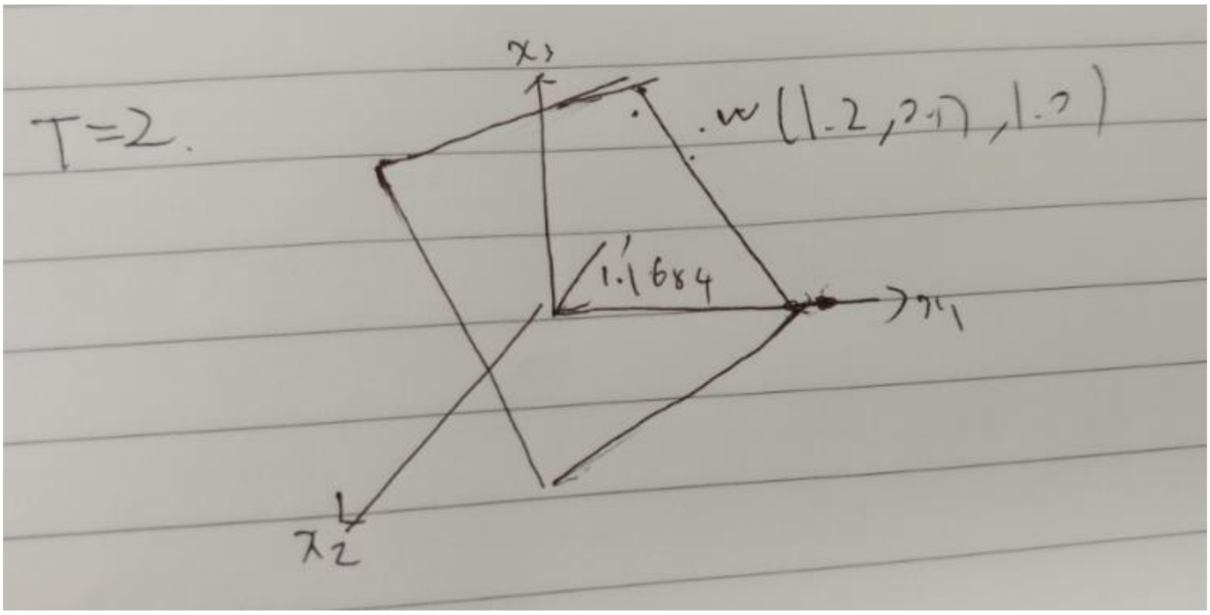
```
A의 행렬식
34.0

A의 고유값 분해
A의 고유값
[ 5.52552524+0.j         -1.52204833+1.31733645j -1.52204833-1.31733645j
  1.51857142+0.j        ]
A의 고유벡터
[[-0.68465996+0.j          0.61993500+0.j          0.61993500-0.j
   0.11199539+0.j        ]
 [-0.21666012+0.j         -0.34875983-0.24694874j -0.34875983+0.24694874j
  -0.93998928+0.j        ]
 [-0.60547918+0.j         -0.31379356-0.41033136j -0.31379356+0.41033136j
  -0.22854692+0.j        ]
 [-0.34306572+0.j         -0.26392900+0.31075191j -0.26392900-0.31075191j
  -0.22725204+0.j        ]]

A의 특이값 분해
(array([[-0.8819521 ,  0.25706354, -0.24178316, -0.31244156],
       [ 0.05743106, -0.76320224, -0.26028922, -0.58862003],
       [-0.42722253, -0.58124436,  0.04756433,  0.69092224],
       [-0.19063859, -0.11660051,  0.93355667, -0.28023773]]), array([ 6.97059301,  3.8068416 ,  1.85521105,  0.69063894]), arr
ay([[-0.32126336,  0.26952747, -0.50380033, -0.75520197],
       [-0.8221184 , -0.53601693,  0.18911937,  0.03226532],
       [ 0.16919264, -0.01995035,  0.78137601, -0.60035603],
       [ 0.43850318, -0.7997767 , -0.31602313, -0.2611543 ]]))
```

6. 놈을 계산

```
import numpy as np
x = np.array([3,-4,-1.2,0,2.3])
x = x.T
n1 = np.linalg.norm(x,axis=0,ord=1)
n2 =  np.linalg.norm(x,axis=0,ord=2)
n3 =  np.linalg.norm(x,axis=0,ord=2)
maxn =  np.linalg.norm(x,axis=0,ord=np.inf)
print("1차놈",n1)
print("2차놈",n2)
print("3차놈",n3)
print("max놈",maxn)
k = np.array([[2,1],[1,5],[4,1]])
print("프로베니우스 놈",np.linalg.norm(k,ord='fro'))
```

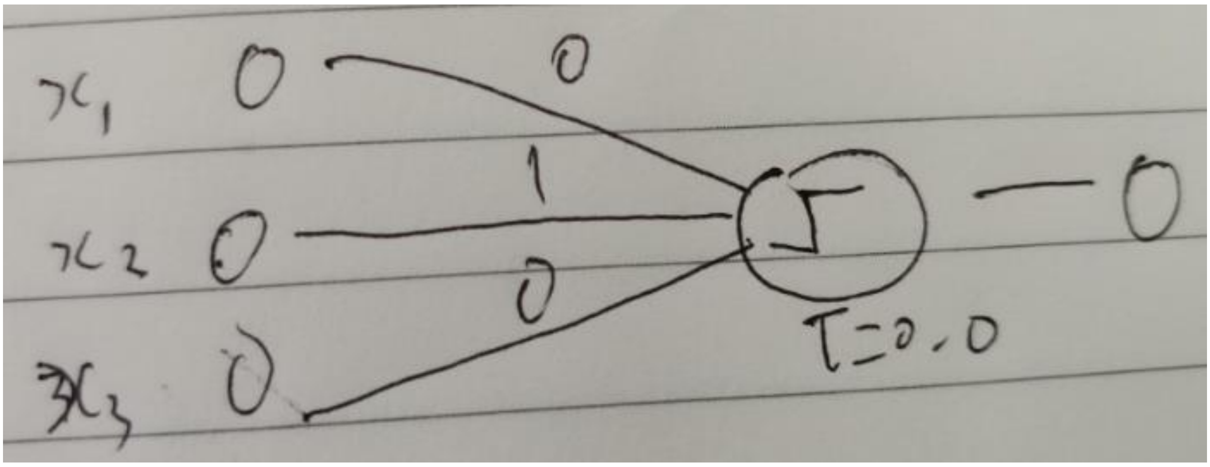
```
1차놈 10.5
2차놈 5.6329388422
3차놈 5.6329388422
max놈 4.0
프로베니우스 놈 6.92820323028
```

7. T 값의 변화



만약 T가 2라고 한다면 결정평면과 원점벡터사이의 거리만 증가할 뿐이다. 따라서 T값의 변화는 원점과 결정평면사이의 거리임으로 결정평면이 어디 위치하느냐를 나타내준다. 이는 곧 기울기임으로 이에 따라 동일한 입력값이라고 하더라도 출력값이 다르게 나타나게 된다.

8. 퍼셉트론 제시



W=(0,1,0) T=0을 가지는 벡터이다. 왜냐하면 x2에 의해서만 영향을 받고, 이에 따라 공간이 나누어 지기 때문이다.

9. 연쇄법칙 활용하기

$$f(x) = 2\left(\frac{1}{4}(1-2x)^2-1\right)^3 - 3\left(\frac{1}{4}(1-2x)^2-1\right)^2 - 3$$

$$\begin{cases} h(x) = \frac{1}{4}(1-2x)^2 - 1 \\ i(x) = 1-2x. \end{cases}$$

$$\Rightarrow f(x) = 2h(i(x))^3 - 3h(i(x))^2 - 3$$

$$f'(x) = 6h'(i(x))^2 \cdot i'(x) - 6h'(i(x)) \cdot i'(x)$$

$$= -6(1-2x)\left(\frac{1}{4}(1-2x)^2-1\right)^2 + 6(1-2x)\left(\frac{1}{4}(1-2x)^2-1\right)$$

$\therefore f'(0) = -\frac{63}{8}$ $\therefore f'(2.1) = 16.7731$

10.

1)최소점과 최소값 구하기

$$f(x_1, x_2) = 2x_1^2 + 3x_1x_2 + 2x_2^2 - 4x_1 + 2x_2 - 24$$

$$\left[\begin{array}{l} f'(x_1) = 4x_1 + 3x_2 - 4 = 0 \\ f'(x_2) = 4x_2 + 3x_1 + 4 = 0 \end{array}\right) \text{연립}$$

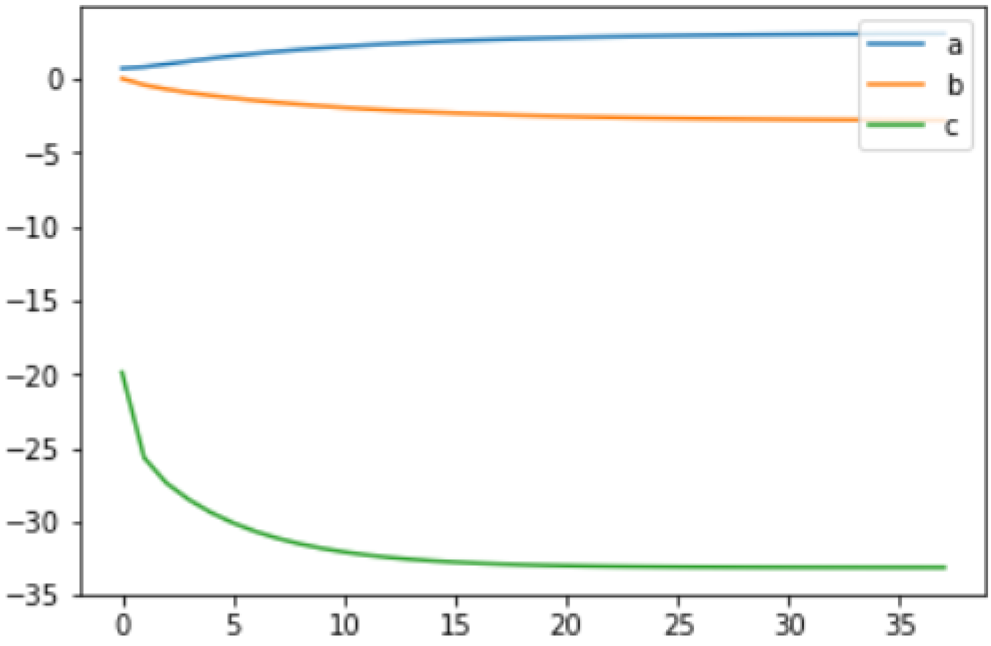
$$\Rightarrow \quad x_2 = -\frac{20}{7}, \quad x_1 = \frac{22}{7}$$

$$\therefore \; f\left(\frac{22}{7}, -\frac{20}{7}\right) = -33.143 \; (\text{최소 값})$$

2)

```
a = 1.0
b = 0.9
p = 0.1
A = []
B = []
C = []
I = []
error = 0.001
for i in range(1000):
    c = 2*(a**2) + 3*a*b +2*(b**2) - 4*a + 2*b -24
    a2 = p*(4*a+3*b-4)
    b2 = p*(4*b + 3*a +2)
    a -= a2
    b -= b2
    A.append(a)
    B.append(b)
    C.append(c)
    I.append(i)
    if i>1:
        t = abs(C[i-1]-C[i])
        if t <= error:
            print("최소 값= ",c)
            break
plt.plot(I,A)
plt.plot(I,B)
plt.plot(I,C)
plt.legend(['a','b','c'], loc="upper right")
plt.show()
```

최소 값= -33.13927955975601



값의 변화에 따라 c(최소값)가 일정한 지점으로 수렴함을 알 수 있다. 그와 동시에 a(x1),b(x2)의 값도 일정한 값으로 수렴함을 보여주고 있다. 따라서 1번에서 구한 a 의값 22/7, b의 값 -20/7 일 때 최소값 -33.143은 타당함을 알 수 있다.

11. 베이즈 규칙

P(B) : 세 번째 문을 열 확률 사회자가 문을 열 확률

P(A1) : 첫 번째 문에 차가 있을 확률 = 1/3

P(B|A1) : 첫 번째 문에 차가 있을 때 세 번째 문을 열 확률 = 1/2

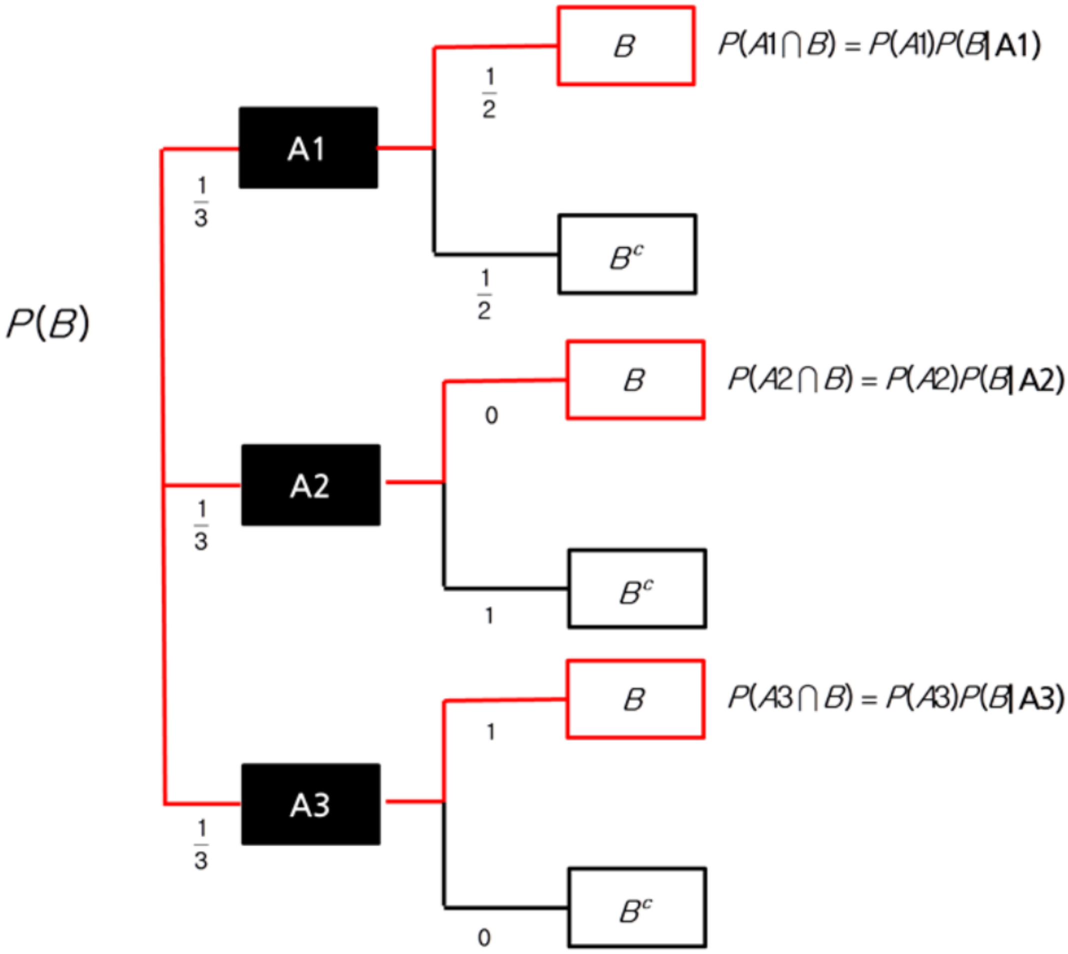
P(A2) : 두 번째 문에 차가 있을 확률 = 1/3

P(B|A2) : 두 번째 문에 차가 있을 때 세 번째 문을 열 확률 = 1

P(A3) : 세 번째 문에 차가 있을 확률 = 1/3

P(B|A3) : 세 번째 문에 차가 있을 때 세 번째 문을 열 확률 = 0

이때의 경우의 수는 다음과 같다. 사건이 벌어질 확률을 모두더한다면 1임으로 B의 경우로 상황을 전개해보고자 한다. 그 상황들은 다음과 같다.



이 notation을 이용하여 아래와 같은 확률을 구할 수 있다.

$$P(A1|B)$$

$$= \frac{P(A1 \cap B)}{P(B)}$$

$$= \frac{P(A1 \cap B)}{P(A1 \cap B) + P(A2 \cap B) + P(A3 \cap B)}$$

$$= \frac{P(A1)P(B|A1)}{P(A1)P(B|A1) + P(A2)P(A2|B) + P(A3)P(A3|B)}$$

$$= \frac{\frac{1}{3} \times \frac{1}{2}}{\frac{1}{3} \times \frac{1}{2} + \frac{1}{3} \times 0 + \frac{1}{3} \times 1} = \frac{1}{3}$$

따라서 선택을 고수하는 것 보다 선택을 바꾸는 것 (이때의 확률은 2/3)이기에 더 유리하다.