



Multivariable Linear regression

Simple Linear Regression



$$H(x) = Wx + b$$

하나의 정보로부터 하나의 결론을 짓는 모델

Multivariate Linear Regression



$$H(x) = Wx + b$$

Multivariable Linear regression : 복수의 정보가 존재할 때 하나의 결론을 짓는 모델

$$H(x) = w_1x_1 + w_2x_2 + w_3x_3 + b$$

단순하게는 이렇게 표현할 수 있지만 x 의 개수가 많아지면 불가능

$$H(x) = Wx + b$$

```
hypothesis = x_train.matmul(w) + b
```

- **matmul()** 로 한번에 계산
 - 더 간결함
 - x 의 길이가 바뀌어도 코드를 바꿀 필요가 없음
 - 속도가 더 빠름

Cost Function

$$cost(W) = \frac{1}{m} \sum_{i=1}^m \left(\underset{\text{Mean}}{H(x^{(i)})} - \underset{\text{Prediction}}{y^{(i)}} \right)^2$$

Mean
Prediction
Target

기존 Simple Linear Regression과 동일한 공식

학습방식

Gradient Descent with torch.optim

$$\nabla W = \frac{\partial cost}{\partial W} = \frac{2}{m} \sum_{i=1}^m (W x^{(i)} - y^{(i)}) x^{(i)}$$

$$W := W - \alpha \nabla W$$

Optimizer을 설정한 후 Cost를 구할때마다 Optimizer의 Gradient에 저장한 후 Gradient Descent를 실행한다.

```
#데이터
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])

#모델 초기화
W = torch.zeros((3, 1), requires_grad = True)
b = torch.zeros(1, requires_grad = True)

#optimizer 설정
optimizer = optim.SGD([W, b], 1r = 1e-5)
```

Simple Linear Regression과 달라진 점은 데이터와 W를 정의하는 부분밖에 없다.

```

nb_epochs = 20
for epoch in range(nb_epochs + 1):

    hypothesis = x_train.matmul(W) + b

    cost = torch.mean((hypothesis - y_train) ** 2)

    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    print('Epoch {:4d}/{:} Batch {:}/{:} Cost: {:.6f}'.format(
        epoch, nb_epochs, hypothesis.squeeze().detach(),
        cost_item()
    ))

```

학습 부분은 동일하다.

Results

```

Epoch 0/20 hypothesis: tensor([0., 0., 0., 0., 0.]) Cost: 29661.800781
Epoch 1/20 hypothesis: tensor([67.2578, 80.8397, 79.6523, 86.7394, 61.6605]) Cost: 9298.520508
Epoch 2/20 hypothesis: tensor([104.9128, 126.0990, 124.2466, 135.3015, 96.1821]) Cost: 2915.713135
Epoch 3/20 hypothesis: tensor([125.9942, 151.4381, 149.2133, 162.4896, 115.5097]) Cost: 915.040527
Epoch 4/20 hypothesis: tensor([137.7968, 165.6247, 163.1911, 177.7112, 126.3307]) Cost: 287.936005
Epoch 5/20 hypothesis: tensor([144.4044, 173.5674, 171.0168, 186.2332, 132.3891]) Cost: 91.371017
Epoch 6/20 hypothesis: tensor([148.1035, 178.0144, 175.3980, 191.0042, 135.7812]) Cost: 29.758139
Epoch 7/20 hypothesis: tensor([150.1744, 180.5042, 177.8508, 193.6753, 137.6805]) Cost: 10.445305
Epoch 8/20 hypothesis: tensor([151.3336, 181.8983, 179.2240, 195.1707, 138.7440]) Cost: 4.391228
Epoch 9/20 hypothesis: tensor([151.9824, 182.6789, 179.9928, 196.0079, 139.3396]) Cost: 2.493135
Epoch 10/20 hypothesis: tensor([152.3454, 183.1161, 180.4231, 196.4765, 139.6732]) Cost: 1.897688
Epoch 11/20 hypothesis: tensor([152.5485, 183.3610, 180.6640, 196.7389, 139.8602]) Cost: 1.710541
Epoch 12/20 hypothesis: tensor([152.6620, 183.4982, 180.7988, 196.8857, 139.9651]) Cost: 1.651413
Epoch 13/20 hypothesis: tensor([152.7253, 183.5752, 180.8742, 196.9678, 140.0240]) Cost: 1.632387
Epoch 14/20 hypothesis: tensor([152.7606, 183.6184, 180.9164, 197.0138, 140.0571]) Cost: 1.625923
Epoch 15/20 hypothesis: tensor([152.7802, 183.6427, 180.9399, 197.0395, 140.0759]) Cost: 1.623412
Epoch 16/20 hypothesis: tensor([152.7909, 183.6565, 180.9530, 197.0538, 140.0865]) Cost: 1.622141
Epoch 17/20 hypothesis: tensor([152.7968, 183.6643, 180.9603, 197.0618, 140.0927]) Cost: 1.621253
Epoch 18/20 hypothesis: tensor([152.7999, 183.6688, 180.9644, 197.0662, 140.0963]) Cost: 1.620500
Epoch 19/20 hypothesis: tensor([152.8014, 183.6715, 180.9666, 197.0686, 140.0985]) Cost: 1.619770
Epoch 20/20 hypothesis: tensor([152.8020, 183.6731, 180.9677, 197.0699, 140.1000]) Cost: 1.619033

```

Final (y)
152
185
180
196
142

- 점점 작아지는 Cost
- 점점 y 에 가까워지는 $H(x)$
- Learning rate 에 따라 발산할수도!

nn.Module

```

#모델 초기화
W = torch.zeros((3, 1), requires_grad = True)
b = torch.zeros(1, requires_grad = True)

# H(x) 계산
hypothesis = x_train.matmul(W) + b

```

```
import torch.nn as nn

class MultivariateLinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3, 1)

    def forward(self, x):
        return self.linear(x)

hypothesis = model(x_train)
```

- nn.Module 을 상속해서 모델 생성
- nn.Linear(3, 1)
 - 입력 차원: 3
 - 출력 차원: 1
- Hypothesis 계산은 forward() 에서!
- Gradient 계산은 Pytorch가 알아서 해준다 - backward()

F.mse_loss

```
cost = torch.mean((hypothesis - y_train) ** 2)

import torch.nn.functional as F
cost = F.mse_loss(prediction, y_train)
```

- torch.nn.functional 에서 제공하는 loss function 사용
- 쉽게 다른 loss 와 교체 가능! (l1_loss, smooth_l1_loss 등..)