선형회귀 문제를 pytorch를 활용하여 풀어보기 ¶

학습 목표

- pytorch로 구현하는 딥러닝 기본에 대해 알아본다.
- pytorch를 이용하여 기본 선형회귀 모델을 만들어본다.

목차

01 데이터 확인

목차로 이동하기

데이터

- 다섯명의 사람
- 신장과 체중은 다음과 같다.
  - (166, 58.7), (176, 75.7), (171, 62.1), (173, 70.4), (169, 60.1)
- 주어진 신장으로부터 체중을 예측하는 머신러닝 모델을 만들어보기

In [1]:

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import torch
```

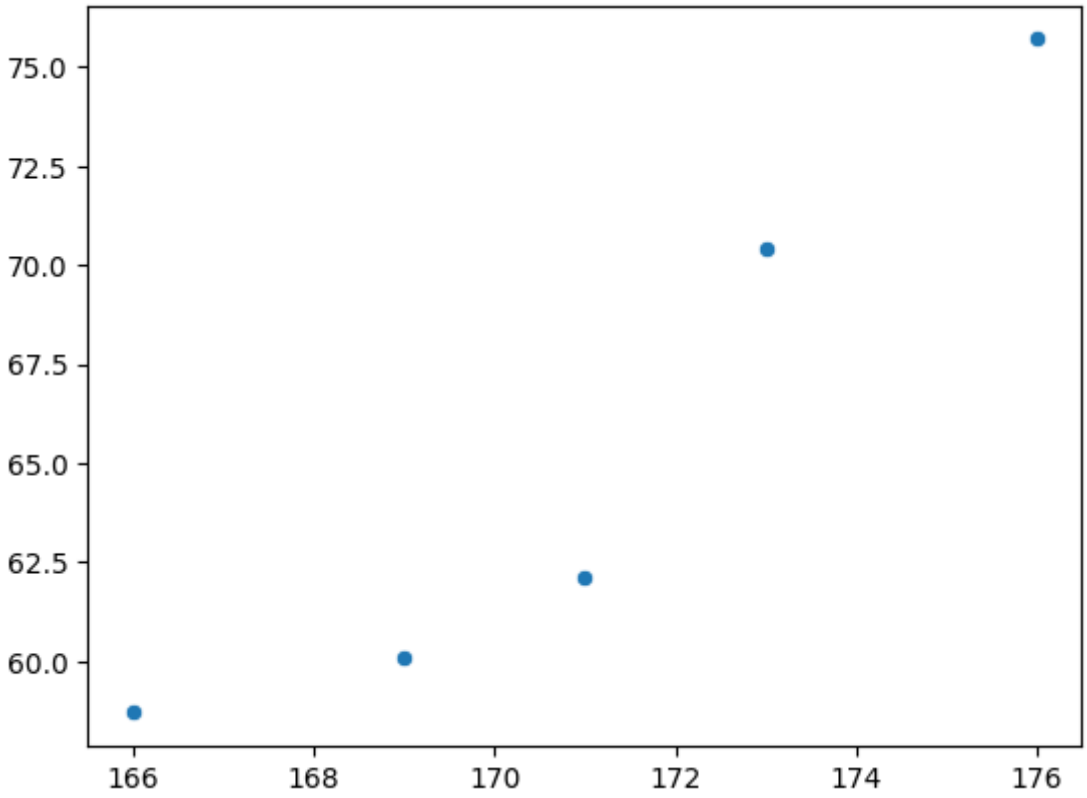
In [2]:

```
height = [166,176, 171, 173, 169]
weight = [58.7, 75.7, 62.1, 70.4, 60.1]

sns.scatterplot(x=height, y=weight)
```

Out[2]:

<AxesSubplot: >



선형 회귀 구하기

- 산포도는 신장과 체중사이에서 일정한 관계가 있다.
- 다음과 같이 표현된 것은 1차 함수로 근사할 수 있을 것이다.
- 다섯개의 점과 가장 가까운 1차 함수를 구하는 것을 우리는 선형회귀라 말할 수 있다.

경사하강법

- 딥러닝 프로그래밍은 수학적으로 '경사하강법'으로 불리는 알고리즘에 기반하게 된다.
- 편미분, 선형대수와 같은 수학적 지식이 필요.
- 꼭대기에서 제일 낮은 지점까지 이동하는 방법.
- 경사하강법은 가장 작은 손실(오차)값을 구하는 문제 '최솟값을 구하는 문제'로 변경
- 예측 함수를 이용하여 출력 값을 구하면 원래 정답과 비교하여 손실(loss)을 계산한다. 이를 토대로 경사를 계산하여 경사값 변경하고, 파라미터(W)와 B(바이어스)를 조금씩 수정해 간다.
    - 경사값에 작은 정수(학습률) lr를 곱해, 그 값만큼 W, B를 줄여나간다.

- 손실함수(loss)는 예측 함수에 따라 적절한 것을 고른다.
    - 회귀 모델의 경우, 평균제곱오차(MSE)를 선택했다.

In [3]:

```
dat1 = np.array(height)
dat2 = np.array(weight)
dat1, dat2
```

Out[3]:

```
(array([166, 176, 171, 173, 169]), array([58.7, 75.7, 62.1, 70.4, 60.1]))
```

In [4]:

```
dat_all = np.dstack([dat1, dat2])
dat_all
```

Out[4]:

```
array([[[166. ,  58.7],
        [176. ,  75.7],
        [171. ,  62.1],
        [173. ,  70.4],
        [169. ,  60.1]]])
```

In [5]:

```
### 데이터를 평균값을 0으로 되도로 변환해 준다.
X = dat1 - dat1.mean()
Y = dat2 - dat2.mean()
```

X와 Y를 텐서 변수로 변환하기

In [6]:

```
X = torch.tensor(X).float()
Y = torch.tensor(Y).float()

# 결과확인
print(X)
print(Y)
```

```
tensor([-5.,  5.,  0.,  2., -2.])
tensor([-6.7000, 10.3000, -3.3000,  5.0000, -5.3000])
```

예측 함수 : Yp = W * X + B

- W와 B의 정의
- W와 B는 경사 계산을 위해 requires_grad=True로 설정.
  - requires_grad = True를 통해 미분 대상의 텐서가 됨.

In [7]:

```
W = torch.tensor(1.0, requires_grad = True).float()
B = torch.tensor(1.0, requires_grad = True).float()
```

- 두 변수의 초깃값을 1.0으로 설정.

예측값 Yp의 계산

In [9]:

```
def pred(X):
    return W * X + B
```

In [10]:

```
# 예측 값 계산
Yp = pred(X)

# 결과 확인
print(Yp)
```

```
tensor([-4.,  6.,  1.,  3., -1.], grad_fn=<AddBackward0>)
```

손실함수 계산

In [12]:

```
def mse(Yp, Y):
    loss = ( (Yp - Y) ** 2 ).mean()
    return loss
```

손실 계산

In [13]:

```
loss = mse(Yp, Y)

print(loss)
```

```
tensor(13.3520, grad_fn=<MeanBackward0>)
```

02 경사 계산 및 파라미터 수정

목차로 이동하기

In [14]:

```
# 경사값 확인
print(W.grad)
print(B.grad)
```

None
None

In [15]:

```
# 경사 계산은 backward()로 호출로 가능
loss.backward()
```

In [16]:

```
# 경사값 확인
print(W.grad)
print(B.grad)
```

tensor(-19.0400)
tensor(2.0000)

파라미터 수정

- 경사 계산 후, 그 값에 일정한 학습률 lr(0.01이나 0.001과 같은 값으로 설정)을 곱하여, 원래의 파라미터에서 빼준다.

In [17]:

```
# 학습률 정의
lr = 0.001

# 경사를 기반으로 파라미터 수정
W = W - lr * W.grad
B = B - lr * B.grad
```

In [18]:

```
print(W)
print(B)
```

tensor(1.0190, grad_fn=<SubBackward0>)
tensor(0.9980, grad_fn=<SubBackward0>)

- 초깃값이 W, B가 모두 1이었다. W는 증가하는 방향으로, B는 감소하는 방향으로 조금씩 변화했다.
- 이 계산을 반복해서 W와 B를 최적의 값으로 수렴시키는 것이 경사 하강법이다.

03. 실전 모델 만들어보기 - 선형회귀 구현하기

목차로 이동하기

- 가중치(Weight)와 편향(bias) 초기화
- 경사 하강법 구현
  - SGD는 경사 하강법의 일정, lr은 학습률(learning rate)를 의미)
- SGD - 확률적 경사 하강법(Stochastic Gradient Descent)
  - 점진적 학습의 대표적 알고리즘
  - 학습용 데이터 세트에서 샘플 하나씩 꺼내(랜덤) 손실 함수의 경사를 따라 최적의 모델을 찾는 알고리즘

투자한 시간과 게임 캐릭터 능력치 향상. 상관관계 예측

- 4시간 투자한다면 게임 캐릭터 능력 향상은 어떻게 될까?

In [19]:

```
# 도구 임포트
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

In [20]:

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

In [27]:

```
# optimizer 설정
optimizer = optim.SGD([W, b], lr=0.01)

# 경사 하강법 반복
num_epochs = 200

# 학습 기록을 위한 배열 초기화
hist = np.zeros( (0, 2))
```

- 1차 함수의 계수와 정수항 W와 B는 초기상태로 변경.
- 반복 계산하는 횟수는 num_epochs라는 이름의 변수로 총 500회 설정.
- 학습률은 lr = 0.001

시간(hours)	시간당능력치향상(p)
1	200
2	400
3	600

In [28]:

```
# 데이터
X = torch.FloatTensor([[1], [2], [3]])
Y = torch.FloatTensor([[200], [400], [600]])
```

In [29]:

```
X, Y
```

Out[29]:

```
(tensor([[1.],
         [2.],
         [3.]]),
 tensor([[200.],
         [400.],
         [600.]]))
```

In [30]:

```
def mse(Yp, Y):
    loss = ( (Yp - Y) ** 2 ).mean()
    return loss
```

In [31]:

```
# 루프 처리
for epoch in range(num_epochs + 1):

    # 예측 계산
    Yp = W * X + B

    # 손실 계산
    loss = mse(Yp, Y)

    # Gradient 초기화
    optimizer.zero_grad()
    # 경사 계산 - 역전파를 통한 기울기 계산
    loss.backward()
    # 파라미터 업데이트
    optimizer.step()

    # 손실 기록
    if (epoch % 10 == 0):
        item = np.array([epoch, loss.item()])
        hist = np.vstack( ( hist, item ))
        print(f"epoch = {epoch}  loss = {loss:.4f}")
```

```
epoch = 0  loss = 152792.8281
epoch = 10  loss = 21530.7207
epoch = 20  loss = 3034.0964
epoch = 30  loss = 427.6685
epoch = 40  loss = 60.3867
epoch = 50  loss = 8.6317
epoch = 60  loss = 1.3385
epoch = 70  loss = 0.3108
epoch = 80  loss = 0.1661
epoch = 90  loss = 0.1456
epoch = 100  loss = 0.1428
epoch = 110  loss = 0.1423
epoch = 120  loss = 0.1423
epoch = 130  loss = 0.1423
epoch = 140  loss = 0.1423
epoch = 150  loss = 0.1423
epoch = 160  loss = 0.1423
epoch = 170  loss = 0.1423
epoch = 180  loss = 0.1423
epoch = 190  loss = 0.1423
epoch = 200  loss = 0.1423
```

04. 최종 학습 결과 확인

목차로 이동하기

In [32]:

```
hist
```

Out[32]:

```
array([[0.00000000e+00, 1.52792828e+05],
       [1.00000000e+01, 2.15307207e+04],
       [2.00000000e+01, 3.03409644e+03],
       [3.00000000e+01, 4.27668457e+02],
       [4.00000000e+01, 6.03866997e+01],
       [5.00000000e+01, 8.63168621e+00],
       [6.00000000e+01, 1.33850479e+00],
       [7.00000000e+01, 3.10840249e-01],
       [8.00000000e+01, 1.66051805e-01],
       [9.00000000e+01, 1.45644516e-01],
       [1.00000000e+02, 1.42757967e-01],
       [1.10000000e+02, 1.42348647e-01],
       [1.20000000e+02, 1.42294422e-01],
       [1.30000000e+02, 1.42292261e-01],
       [1.40000000e+02, 1.42288253e-01],
       [1.50000000e+02, 1.42285168e-01],
       [1.60000000e+02, 1.42285168e-01],
       [1.70000000e+02, 1.42285168e-01],
       [1.80000000e+02, 1.42285168e-01],
       [1.90000000e+02, 1.42285168e-01],
       [2.00000000e+02, 1.42285168e-01]])
```

In [33]:

```
# 최종 파리미터 확인
print("W = ", W.data.numpy())
print("B = ", B.data.numpy())

# 손실 확인
print(f"초기 상태 : 손실 : {hist[0,1]:.4f}")
print(f"최종 상태 : 손실 : {hist[-1,1]:.4f}")
```

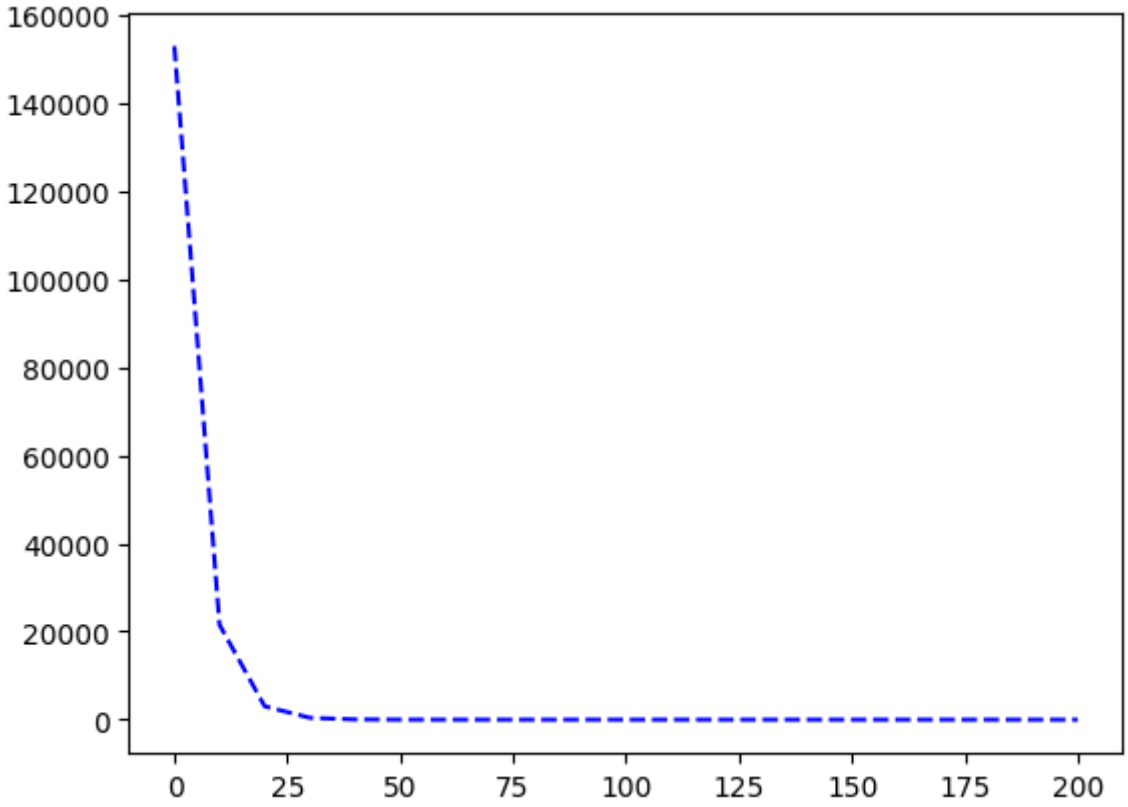
```
W =  [199.57222]
B =  0.998
초기 상태 : 손실 : 152792.8281
최종 상태 : 손실 : 0.1423
```

In [36]:

```
plt.plot(hist[ :, 0], hist[:, 1], '--b')
```

Out[36]:

```
[<matplotlib.lines.Line2D at 0x228583056a0>]
```



실습해 보기 1

- optim.SGD에서 momentum=0.9로 변경후, 확인해 보자.(hist는 hist2로 변경)
- 그래프를 그려보자.

실습해 보기 2

- 신장과 체중은 다음과 같다. (166, 58.7), (176, 75.7), (171, 62.1), (173, 70.4), (169, 60.1)
- 주어진 신장으로부터 체중을 예측하는 머신러닝 모델을 만들어보기

- SGD의 최적화 함수 클래스에서 momentum으로 불리는 것을 설정했을 때, 학습이 빠르게 되어있음을 확인할 수 있다.
  - 최적화 함수 또는 파라미터 변경으로 알고리즘 최적화 가능하다.