Data Mining

다중 선형 회귀 (Multiple Linear Regression)

학습 목표

- 다중 회귀 분석 모델의 개념과 구현을 알아보고 부트스트랩, 계수의 표준 오차, 정규화에 대해서 살펴보자.

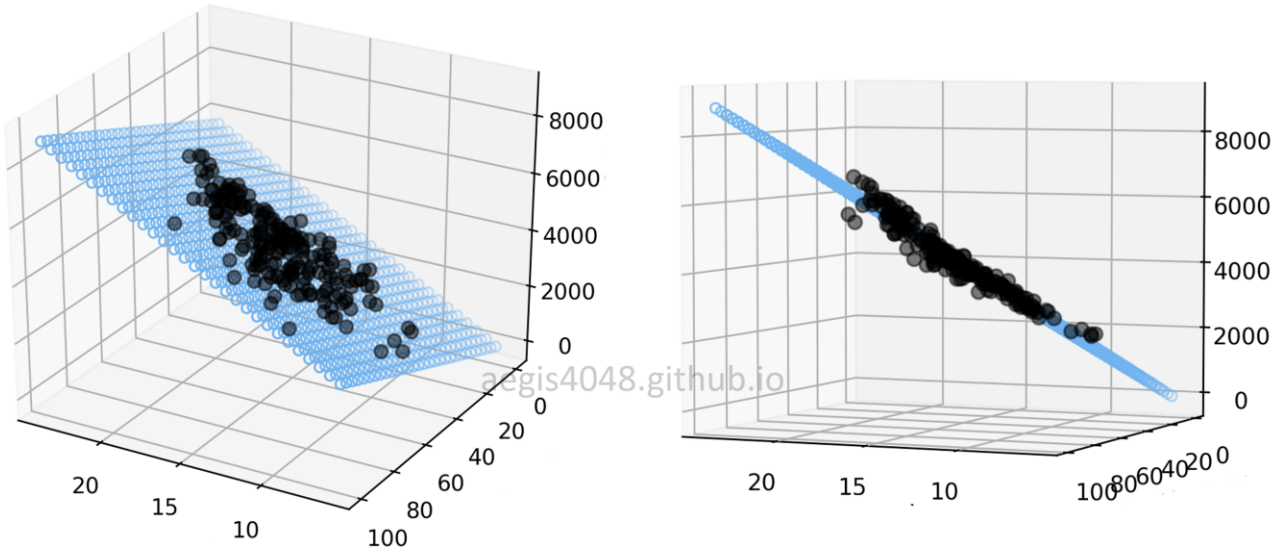
주요 내용

1. 다중 회귀 분석
2. 경사 하강법으로 학습
3. 부트스트랩
4. 계수의 표준 오차
5. 정규화

1. 다중 선형 회귀

다중 선형 회귀 (Multiple Linear Regression)

여러 독립 변수와 하나의 종속 변수 사이에 선형 관계를 선형 함수로 모델링 하는 기법



$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + \varepsilon$$

독립변수 여러개

관측 오차 $\varepsilon \sim \mathcal{N}(\epsilon \mid 0, \sigma^2)$

정규 분포를 따르며 평균은 0이고 분산이 σ^2

다중 선형 회귀의 예측

다차원 데이터의 벡터 표현

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + \varepsilon \qquad \varepsilon \sim \mathcal{N}(\epsilon \mid 0, \sigma^2)$$

상수 항은 입력이 1인 항으로 간주

$$\boldsymbol{x} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_k \end{pmatrix} \qquad \boldsymbol{\beta} = \begin{pmatrix} \alpha \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_k \end{pmatrix} \qquad y = \boldsymbol{\beta} \cdot \boldsymbol{x} + \varepsilon$$

벡터의 내적으로 이해가능

k 차원의 벡터로 표현

모델 예측

```python
from scratch.linear_algebra import dot, Vector

def predict(x: Vector, beta: Vector) -> float:
    """assumes that the first element of x is 1"""
    return dot(x, beta)
```

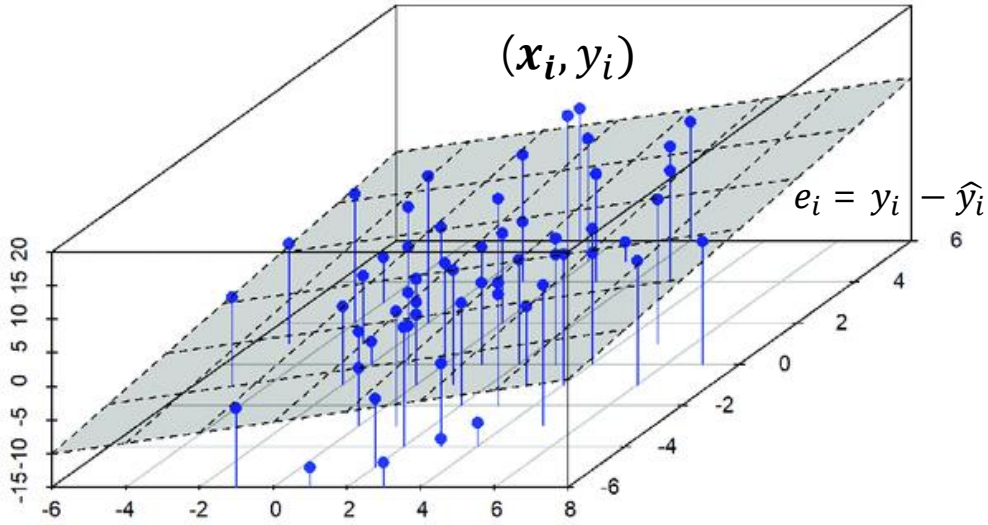
- x와 beta의 내적

모델 파라미터

잔차(residual)를 최소화 하는 다중 선형 모델의 $\boldsymbol{\beta}$를 구해보자!

SSE (Sum of Squared Error) = $\sum_{i=1}^{N} {e_i}^2 = \sum_{i=1}^{N} (y_i - (\boldsymbol{\beta} \cdot \boldsymbol{x_i}))^2$

scala



$e_i = y_i - \hat{y}_i$ 잔차 (residual) 관측 값과 예측 값의 차이

$\hat{y}_i = \boldsymbol{\beta} \cdot \boldsymbol{x_i}$ 예측 값

관측 데이터 $\mathcal{D} = \{(\boldsymbol{x_i}, \mathrm{y}_i) : i = 1 \ \ldots \ N\}$ $\boldsymbol{x_i} = (1, x_{i1}, x_{i2}, \ldots, x_{\mathrm{ik}})^{\mathrm{T}}$

2. 경사 하강법으로 학습

SSE (Sum of Squared Error)

SSE (Sum of Squared Error) = $\sum_{i=1}^{N} {e_i}^2 = \sum_{i=1}^{N} (y_i - (\boldsymbol{\beta} \cdot \boldsymbol{x_i}))^2$ $\quad e_i = y_i - \hat{y_i}$

잔차

```
from typing import List

def error(x: Vector, y: float, beta: Vector) -> float:
    return predict(x, beta) - y
```

- 잔차 정의 $e_i = y_i - \hat{y_i}$와 순서가 반대

24-30

~~SSE~~ squared_error

```
def squared_error(x: Vector, y: float, beta: Vector) -> float:
    return error(x, y, beta) ** 2
```

(error)² = 36.

검증

```
x = [1, 2, 3]
y = 30
beta = [4, 4, 4]  # so prediction = 4 + 8 + 12 = 24

assert error(x, y, beta) == -6
assert squared_error(x, y, beta) == 36
```

경사 하강법 그래디언트 계산

SSE를 손실 함수로 정의하고 최소화하는 문제

$$\text{SSE} = \sum_{i=1}^{N} (y_i - \boldsymbol{\beta} \cdot \boldsymbol{x_i})^2$$

$$\frac{\partial \text{SSE}}{\partial \boldsymbol{\beta}} = \sum_{i=1}^{N} -2(y_i - \boldsymbol{\beta} \cdot \boldsymbol{x_i})\boldsymbol{x_i}$$

구현에서 음수 부호는 error 함수에서 y_i와 $\hat{y}_i$의 순서가 바뀌어서 없어짐

β에 대해 그래디언트 계산

```
def sqerror_gradient(x: Vector, y: float, beta: Vector) -> Vector:
    err = error(x, y, beta)
    return [2 * err * x_i for x_i in x]
```

검증

```
assert sqerror_gradient(x, y, beta) == [-12, -24, -36]
```

경사 하강법 최소자승법

파라미터 초기화

```
import random
import tqdm
from scratch.linear_algebra import vector_mean
from scratch.gradient_descent import gradient_step


def least_squares_fit(xs: List[Vector],
                      ys: List[float],
                      learning_rate: float = 0.001,
                      num_steps: int = 1000,
                      batch_size: int = 1) -> Vector:
    """
    Find the beta that minimizes the sum of squared errors
    assuming the model y = dot(x, beta).
    """
    # Start with a random guess
    guess = [random.random() for _ in xs[0]]
```

- guess : $\boldsymbol{\beta}$ 파라미터 벡터

경사 하강법 최소자승법

num_steps 에폭 반복

```
    for _ in tqdm.trange(num_steps, desc="least squares fit"):
```

미니 배치 생성

```
        for start in range(0, len(xs), batch_size):
            batch_xs = xs[start:start+batch_size]
            batch_ys = ys[start:start+batch_size]
```

그래디언트 계산

```
            gradient = vector_mean([sqerror_gradient(x, y, guess)
                                    for x, y in zip(batch_xs, batch_ys)])
```

- 그래디언트 평균 (손실 함수를 SSE가 아닌 MSE로 사용한 경우)

그래디언트 스텝 실행

```
            guess = gradient_step(guess, gradient, -learning_rate)

    return guess
```

경사 하강법 데이터 적용

[상수 항, 친구 수, 하루 근무 시간, 박사 학위 취득 여부]로 특징을 추가해서 "사이트에 머무르는 시간"을 예측해보자.

```
from typing import List

inputs: List[List[float]] = [[1.,49,4,0],[1,41,9,0],[1,40,8,0],[1,25,6,0],[1,21,
1,0],[1,21,0,0],[1,19,3,0],[1,19,0,0],[1,18,9,0],[1,18,8,0],[1,16,4,0],[1,15,3,0
],[1,15,0,0],[1,15,2,0],[1,15,7,0],[1,14,0,0],[1,14,1,0],[1,13,1,0],[1,13,7,0],[
1,13,4,0],[1,13,2,0],[1,12,5,0],[1,12,0,0],[1,11,9,0],[1,10,9,0],[1,10,1,0],[1,1
0,1,0],[1,10,7,0],[1,10,9,0],[1,10,1,0],[1,10,6,0],[1,10,6,0],[1,10,8,0],[1,10,1
0,0],[1,10,6,0],[1,10,0,0],[1,10,5,0],[1,10,3,0],[1,10,4,0],[1,9,9,0],[1,9,9,0],
[1,9,0,0],[1,9,0,0],[1,9,6,0],[1,9,10,0],[1,9,8,0],[1,9,5,0],[1,9,2,0],[1,9,9,0]
,[1,9,10,0],[1,9,7,0],[1,9,2,0],[1,9,0,0],[1,9,4,0],[1,9,6,0],[1,9,4,0],[1,9,7,0
],[1,8,3,0],[1,8,2,0],[1,8,4,0],[1,8,9,0],[1,8,2,0],[1,8,3,0],[1,8,5,0],[1,8,8,0
],[1,8,0,0],[1,8,9,0],[1,8,10,0],[1,8,5,0],[1,8,5,0],[1,7,5,0],[1,7,5,0],[1,7,0,
0],[1,7,2,0],[1,7,8,0],[1,7,10,0],[1,7,5,0],[1,7,3,0],[1,7,3,0],[1,7,6,0],[1,7,7
,0],[1,7,7,0],[1,7,9,0],[1,7,3,0],[1,7,8,0],[1,6,4,0],[1,6,6,0],[1,6,4,0],[1,6,9
,0],[1,6,0,0],[1,6,1,0],[1,6,4,0],[1,6,1,0],[1,6,0,0],[1,6,7,0],[1,6,0,0],[1,6,8
,0],[1,6,4,0],[1,6,2,1],[1,6,1,1],[1,6,3,1],[1,6,6,1],[1,6,4,1],[1,6,4,1],[1,6,1
,1],[1,6,3,1],[1,6,4,1],[1,5,1,1],[1,5,9,1],[1,5,4,1],[1,5,6,1],[1,5,4,1],[1,5,4
,1],[1,5,10,1],[1,5,5,1], …, [1,1,5,1]]
```

경사 하강법 데이터 적용

친구 수와 사이트 머무는 시간 예제 검증

```
from scratch.statistics import daily_minutes_good

random.seed(0)
# I used trial and error to choose niters and step_size.
# This will run for a while.
learning_rate = 0.001

beta = least_squares_fit(inputs, daily_minutes_good, learning_rate, 5000, 25)
assert 30.50 < beta[0] < 30.70  # constant
assert  0.96 < beta[1] <  1.00  # num friends
assert -1.89 < beta[2] < -1.85  # work hours per day
assert  0.91 < beta[3] <  0.94  # has PhD
```

사이트에서 보내는 시간 (분) = 30.58 + 0.972 [친구 수] - 1.87 [근무 시간] + 0.923 [박사 학위 취득 여부]

모델 적합도 (goodness-of-fit)

$$\left\{ R^2 = \frac{SSR}{SST} \right\} = 1 - \frac{SSE}{SST}$$

- **SST** : Y의 전체 변동 $\sum(y_i - \bar{y}_i)^2$
- **SSE** : 모형에 의해 설명되는 변동 $\sum(y_i - \hat{y}_i)^2$
- **SSR** :모형에 의해 설명되지 않는 변동 $\sum(y_i - \hat{y}_i)^2$

결정계수

```
from scratch.simple_linear_regression import total_sum_of_squares

def multiple_r_squared(xs: List[Vector], ys: Vector, beta: Vector) -> float:
    sum_of_squared_errors = sum(error(x, y, beta) ** 2
                                for x, y in zip(xs, ys))
    return 1.0 - sum_of_squared_errors / total_sum_of_squares(ys)
```

친구수와 사이트 머무는 시간 예제 검증

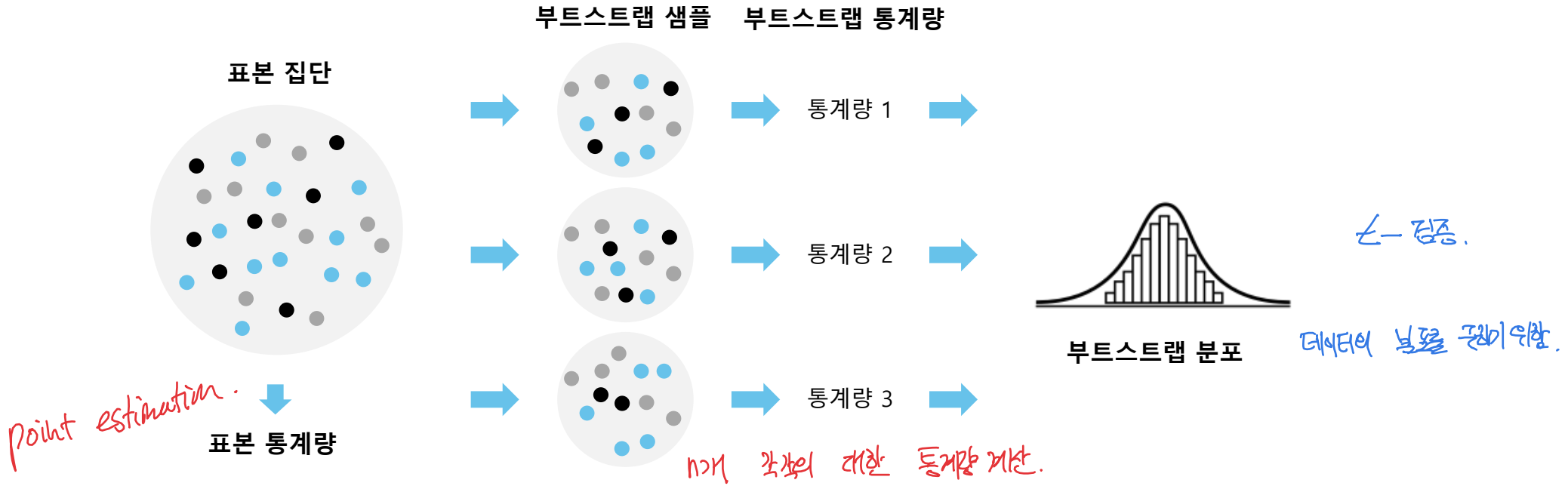
```
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta) < 0.68
```

결정계수가 단순 회귀 분석의 0.329에서 0.69로 올라간 것을 알 수 있다.

3. 부트스트랩

부트스트랩 (Bootstrap)

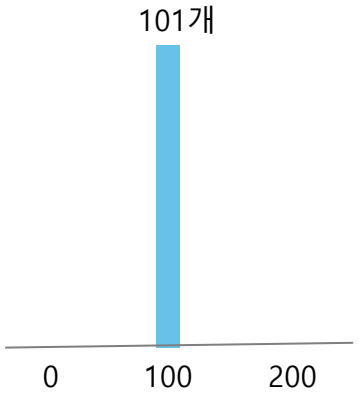
표본 데이터로 모집단의 통계량을 추정할 때,
통계량을 여러 번 측정해서 오차 및 신뢰 구간을 추정하는 방식

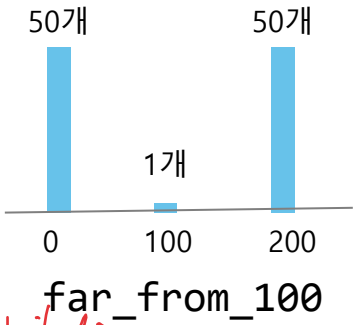


- **부트스트랩 샘플** : 통계량을 측정할 때 원래의 표본 데이터에서 복원 추출 방식으로 새롭게 표본 데이터를 구성
- **부트스트랩 통계량** : 각 부트스트랩 샘플에 대해 통계량을 측정
- **부트스트랩 분포** : 여러 통계량이 생기므로 통계량의 오차와 신뢰 구간을 구할 수 있음

부트스트랩 (Bootstrap)

부트스트랩 통계량을 구해서 두 샘플의 분포를 확인해보자!





검증용 데이터 생성

uniform distribution.

```
# 101 points all very close to 100
close_to_100 = [99.5 + random.random() for _ in range(101)]

# 101 points, 50 of them near 0, 50 of them near 200
far_from_100 = ([99.5 + random.random()] +
                [random.random() for _ in range(50)] +
                [200 + random.random() for _ in range(50)])
```

- 99.5를 중심으로 100에 가까운 101개의 샘플 생성
- [0에 가까운 샘플 50개, 99.5에 가까운 샘플 1개, 200에 가까운 샘플 50개] 총 101개 샘플 생성

부트스트랩 (Bootstrap)

표본 및 통계량의 데이터 타입

```
from typing import TypeVar, Callable

X = TypeVar('X')         # Generic type for data
Stat = TypeVar('Stat')   # Generic type for "statistic"
```

- 임의의 데이터 타입 허용

부트스트랩 샘플 생성

```
def bootstrap_sample(data: List[X]) -> List[X]:
    """randomly samples len(data) elements with replacement"""
    return [random.choice(data) for _ in data]
```

데이터 갯수만큼 복원추출

- data의 크기만큼 부트스트랩 샘플을 복원 추출

부트스트랩 통계

```
def bootstrap_statistic(data: List[X],
                        stats_fn: Callable[[List[X]], Stat],
                        num_samples: int) -> List[Stat]:
    """evaluates stats_fn on num_samples bootstrap samples from data"""
    return [stats_fn(bootstrap_sample(data)) for _ in range(num_samples)]
```

- 부트스트랩 샘플을 num_samples 만큼 생성하고 통계량을 계산
- 통계량을 계산할 때는 전달 함수 stats_fun을 실행
- 부트스트랩 통계량은 리스트 형태로 반환

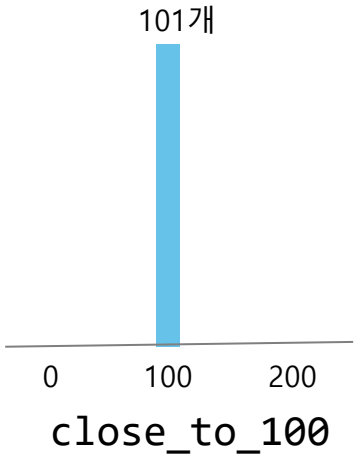
각각의 부트스트랩회함

부트스트랩 (Bootstrap)

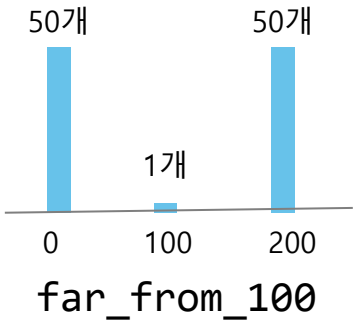
부트스트랩 (중앙값 100번 측정)

```
from scratch.statistics import median, standard_deviation

medians_close = bootstrap_statistic(close_to_100, median, 100)
medians_far = bootstrap_statistic(far_from_100, median, 100)

assert standard_deviation(medians_close) < 1
assert standard_deviation(medians_far) > 90
```

표준 편차 1 미만



표준 편차 90 이상



4. β의 표준 오차 검증

$\hat{\beta}_i$의 표준 오차

부트스트랩을 사용해서 $\hat{\beta}_i$의 표준 오차를 추정해보자.

(변동이 적으면 신뢰할 만한 추정량이고 변동량이 크면 신뢰하기 힘든 추정량이다.)

β의 추정 샘플을 생성

```
from typing import Tuple
import datetime

def estimate_sample_beta(pairs: List[Tuple[Vector, float]]):
    x_sample = [x for x, _ in pairs]
    y_sample = [y for _, y in pairs]
    beta = least_squares_fit(x_sample, y_sample, learning_rate, 5000, 25)
    print("bootstrap sample", beta)
    return beta
```

- 데이터 pairs를 이용해서 최소자승법으로 β의 추정 샘플을 생성

$\hat{\beta}_i$의 표준 오차

β 추정 부트스트랩 (100번)

```
random.seed(0) # so that you get the same results as me

# This will take a couple of minutes!
bootstrap_betas = bootstrap_statistic(list(zip(inputs, daily_minutes_good)),
                                      estimate_sample_beta,
                                      100)
```

부트스트랩 100번.
결과에 beta에 대해서
List. → bootstrap_betas

부트스트랩 추정치의 표준 편차 계산

```
bootstrap_standard_errors = [
    standard_deviation([beta[i] for beta in bootstrap_betas])
    for i in range(4)]
print(bootstrap_standard_errors)

# [1.272,    # constant term, actual error = 1.19
#  0.103,    # num_friends,   actual error = 0.080
#  0.155,    # work_hours,    actual error = 0.127
#  1.249]    # phd,           actual error = 0.998
```

$x\beta_0$
β_1 특징은 3개.
β_2
β_3 → 틸확실함.

- 4개 변수에 대해서 각기 표준 편차 계산

[1.2715078186272781, 0.10318410116073963, 0.15510591689663628, 1.2490975248051257]

↳ 표준편차가 적음 계수로서 정확한 측정임.

표준 편차가 크면 계수의 신뢰도를 의심할 수 있다.

$\hat{\beta}_i$의 t-검정

$f(x) = \beta \cdot x \Rightarrow \beta$

선형 모델의 파라미터 β_i가 회귀 모델에 적합할까?

귀무 가설
(null hypothesis)

$H_0 : \beta_i = 0$

대립 가설
(alternative hypothesis)

$H_1 : \beta_i \neq 0$

양측 검정

귀무 가설은 모분포의 평균인 β_i가 0인지 추정하는 문제로
$n-k$ 자유도를 지닌 스튜던트 t 분포를 따른다.

↳ 회귀 분석이 될 수 있다.

$$t_i = \left(\frac{\hat{\beta}_i - \beta_i}{\hat{\sigma}_i} \right) \sim t(n-k)$$

n : 샘플 수, k : β_i의 개수

$\hat{\sigma}_i$: $\hat{\beta}_i$의 표준 오차

자유도 $n-k$가 클 경우 표준정규분포로 추정할 수 있다.

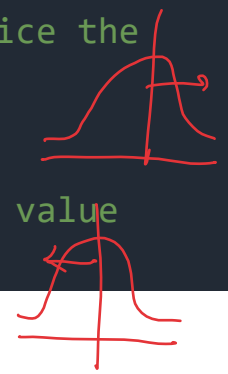
$\beta_i = 0$ 이므로 $t_i = \frac{\hat{\beta}_i}{\hat{\sigma}_i}$

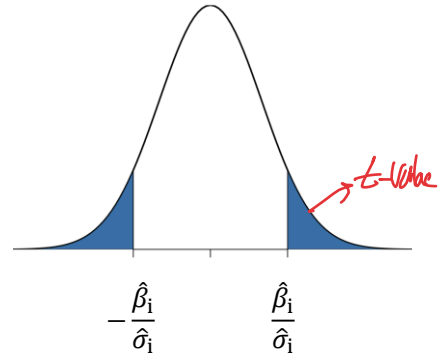
t분포에서 샘플수 많아지면 정규분포와 비슷해지므로 정규분포로 추정함.

$\hat{\beta}_i$의 t-검정

P-value 계산

```
from scratch.probability import normal_cdf

def p_value(beta_hat_j: float, sigma_hat_j: float) -> float:
    if beta_hat_j > 0:
        # if the coefficient is positive, we need to compute twice the
        # probability of seeing an even *larger* value
        return 2 * (1 - normal_cdf(beta_hat_j / sigma_hat_j))
    else:
        # otherwise twice the probability of seeing a *smaller* value
        return 2 * normal_cdf(beta_hat_j / sigma_hat_j)
```





- 양측 검정을 해야 하므로 계산된 p-value를 2배 해준다.

표준 오차에 대한 검증

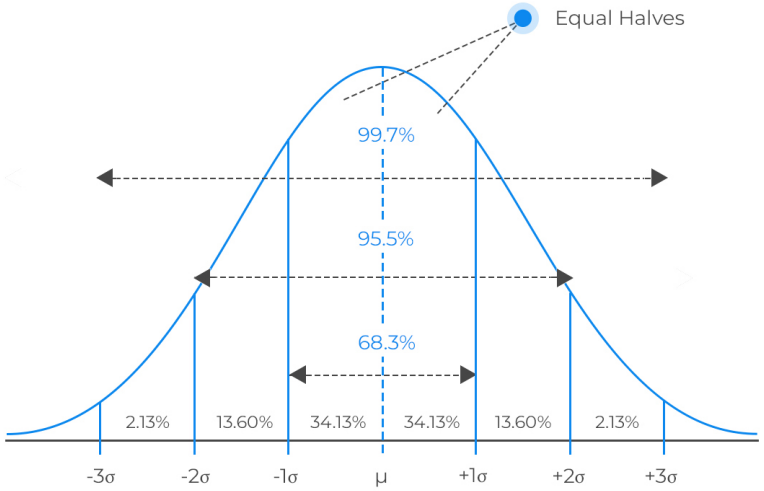
유의수준

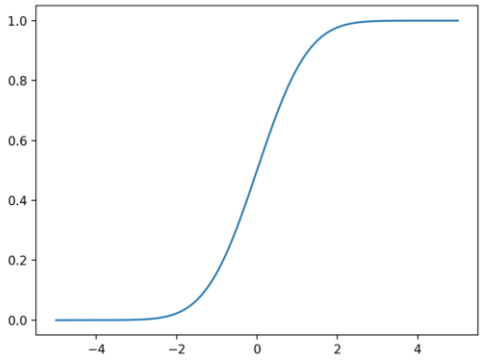
```
assert p_value(30.58, 1.27)   < 0.001  # constant term
assert p_value(0.972, 0.103)  < 0.001  # num_friends
assert p_value(-1.865, 0.155) < 0.001  # work_hours
assert p_value(0.923, 1.249)  > 0.4    # phd
```

p value가 아주 작기 때문에 귀무 가설이 기각되어 β는 0이 아니라고 말할 수 있다.
단, 박사학위 취득 여부의 계수는 0.5에 가깝기 때문에 주의 깊게 봐야 한다.

6장 확률 정규 분포 (Gaussian Distribution)

정규 분포 (Gaussian Distribution) 양방향으로 대칭인 종모양의 분포





확률 밀도 함수 (probability density function) pdf

$$\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

μ: 평균, σ^2 : 분산

누적 분포 함수 (cumulative density function) cdf

$$\frac{1}{2}\left(1 + \operatorname{erf}\frac{x-\mu}{\sigma\sqrt{2}}\right)$$

erf: 오차 함수 (error function)

6장 확률 정규 분포 (Gaussian Distribution)

정규 분포 (확률 밀도 함수)

```python
import math
SQRT_TWO_PI = math.sqrt(2 * math.pi)

def normal_pdf(x: float, mu: float = 0, sigma: float = 1) -> float:
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (SQRT_TWO_PI * sigma))
```

정규 분포 (누적 분포 함수)

```python
def normal_cdf(x: float, mu: float = 0, sigma: float = 1) -> float:
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

확률 밀도 함수

$$\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

누적 분포 함수

$$\frac{1}{2}\left(1 + \operatorname{erf}\frac{x-\mu}{\sigma\sqrt{2}}\right)$$

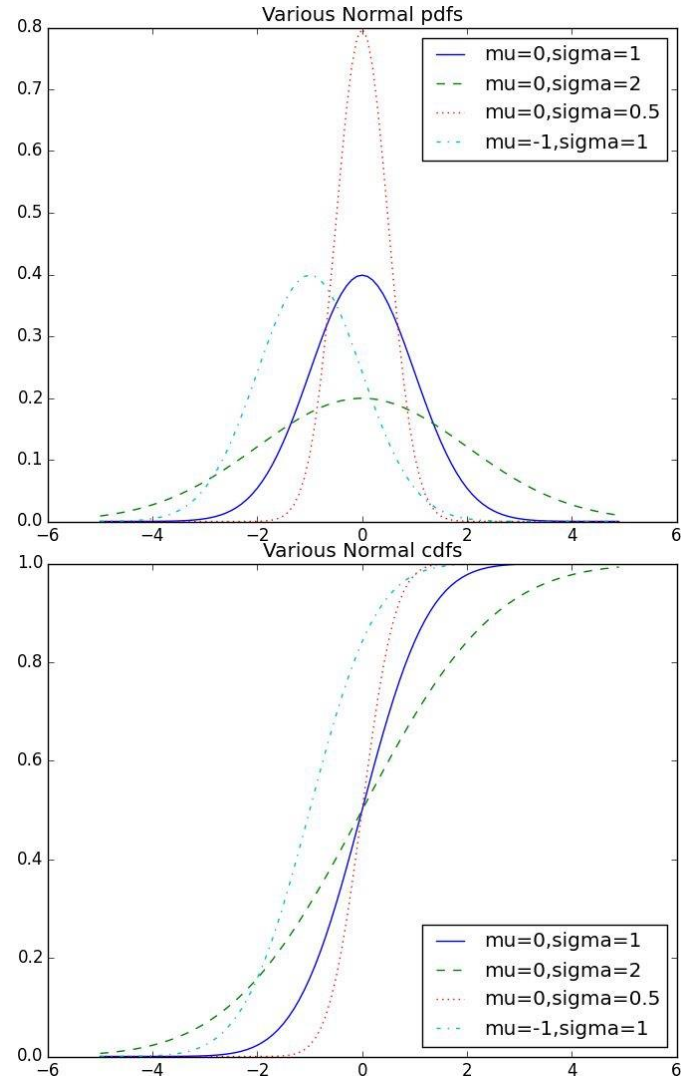
6장 확률 정규 분포 (Gaussian Distribution)

다양한 정규 분포

```
import matplotlib.pyplot as plt
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_pdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_pdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_pdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_pdf(x,mu=-1)   for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend()
plt.title("Various Normal pdfs")
plt.show()
```

누적 분포

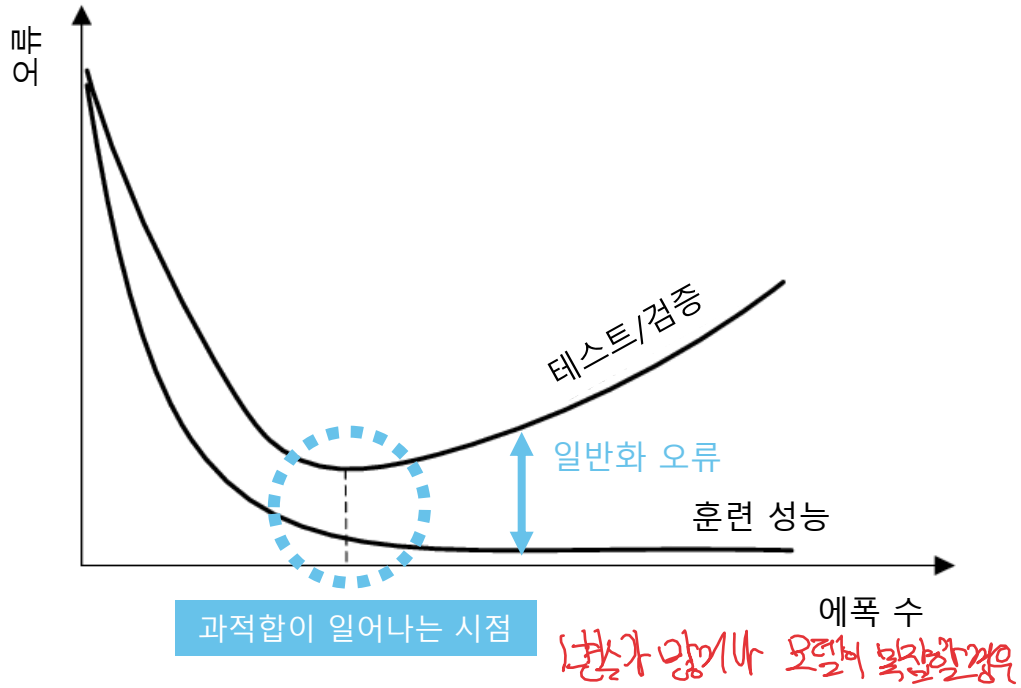
```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend(loc=4) # bottom right
plt.title("Various Normal cdfs")
plt.show()
```



5. 정규화

정규화 (Regularization)

정규화 (Regularization)



- 최적화를 할 때 최적해를 잘 찾도록 정보를 추가하는 기법
- 일반화 오류를 최소화 시키는 것이 목표
- 일반화 오류 : 훈련 성능과 테스트/검증 성능의 차이

단순한 모델이 왜 좋을까?

1 모델에 파라미터 많으면 과적합 되기 쉽다.

- 과적합이 일어나면 모델 성능이 떨어진다.

2 변수가 많으면 모델 해석이 어려워진다.

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k$$

- 각 계수는 변수의 영향력을 설명
- 변수가 많으면 이런 해석이 어려워 짐

손실 함수 정규화 항

변동성이 높은 변수의 계수 β_i를 작게 만들자!

$\lambda\uparrow$ $\beta\downarrow$
$\lambda\downarrow$ $\beta\uparrow$

$$\tilde{J}(\boldsymbol{\beta}) = J(\boldsymbol{\beta}) + \lambda R(\boldsymbol{\beta})$$

데이터 손실 패널티 항 (또는 정규화 항)

λ : 정규화 상수

(정규화 상수가 커지면 패널티가 커져서 β_i가 더 작아짐)

손실 함수에 패널티 항을 추가하면 손실이 최소화 되는 동시에 β_i도 작아지게 됨

리지 회귀 (Ridge)

$$\tilde{J}(\boldsymbol{\beta}) = J(\boldsymbol{\beta}) + \lambda \| \boldsymbol{\beta} \|_2^2$$

L_2 정규화

라소 회귀 (Lasso)

$$\tilde{J}(\boldsymbol{\beta}) = J(\boldsymbol{\beta}) + \lambda \| \boldsymbol{\beta} \|_1$$

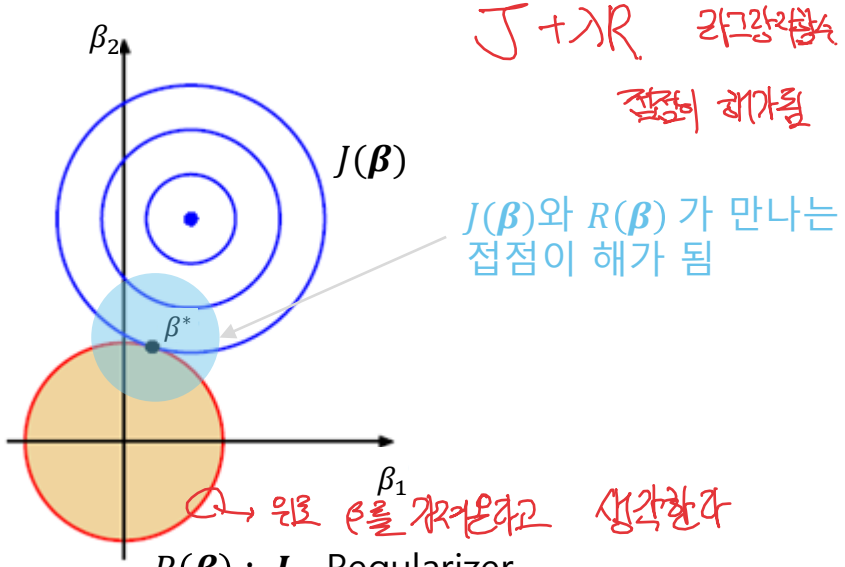
L_1 정규화

β를 작게 만드는것은 같음

손실 함수 정규화 효과

β가 대해 정규화를 하면 β가 작아지면서 원점 근처에서 값이 정해진다.

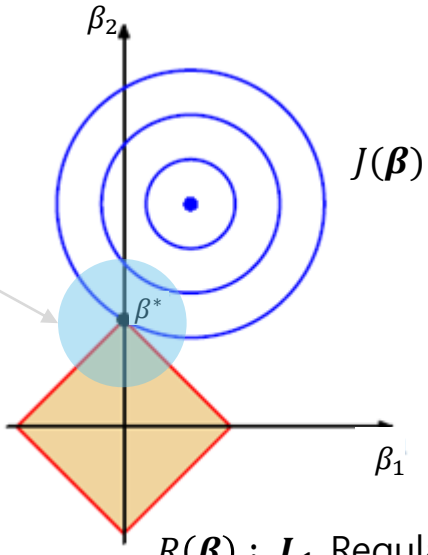
리지 회귀 (Ridge)



$J(\beta)$와 $R(\beta)$ 가 만나는 접점이 해가 됨

$R(\beta)$: L_2 Regularizer

라소 회귀 (Lasso)



$R(\beta)$: L_1 Regularizer

라소 회귀는 β를 0으로 만들어서 유효 파라미터 수를 줄이고 모델을 작게 만든다.

(특징 선택(Feature Selection)**을 하는 효과가 생김)**

리지 회귀 (Ridge Regression)

리지 회귀 (Ridge) $\tilde{J}(\boldsymbol{\beta}) = J(\boldsymbol{\beta}) + \lambda \| \boldsymbol{\beta} \|_2^2$ $J(\boldsymbol{\beta}) = \sum_{i=1}^{N} {e_i}^2 = \sum_{i=1}^{N} (y_i - (\boldsymbol{\beta} \cdot \boldsymbol{x_i}))^2$

패널티 항 정의

```
# alpha is a *hyperparameter* controlling how harsh the penalty is
# sometimes it's called "lambda" but that already means something in Python
def ridge_penalty(beta: Vector, alpha: float) -> float:
    return alpha * dot(beta[1:], beta[1:])
```

- 상수항은 제외 시킴

손실 함수 정의

```
def squared_error_ridge(x: Vector,
                        y: float,
                        beta: Vector,
                        alpha: float) -> float:
    """estimate error plus ridge penalty on beta"""
    return error(x, y, beta) ** 2 + ridge_penalty(beta, alpha)
```

리지 회귀 (Ridge Regression)

패널티 항 그래디언트

```python
from scratch.linear_algebra import add

def ridge_penalty_gradient(beta: Vector, alpha: float) -> Vector:
    """gradient of just the ridge penalty"""
    return [0.] + [2 * alpha * beta_j for beta_j in beta[1:]]
```

$$\frac{\partial}{\partial \boldsymbol{\beta}} \lambda \| \boldsymbol{\beta} \|_2^2 = 2\lambda \boldsymbol{\beta}$$

- 일반적으로 상수항은 정규화에 포함 시키지 않음
- 상수항은 미분 값을 0으로 처리

전체 그래디언트

```python
def sqerror_ridge_gradient(x: Vector,
                           y: float,
                           beta: Vector,
                           alpha: float) -> Vector:
    """
    the gradient corresponding to the ith squared error term
    including the ridge penalty
    """
    return add(sqerror_gradient(x, y, beta),
               ridge_penalty_gradient(beta, alpha))
```

- 손실 함수의 그래디언트에 패널티 항의 그래디언트를 더한 형태

리지 회귀 (Ridge Regression)

경사 하강법 적용

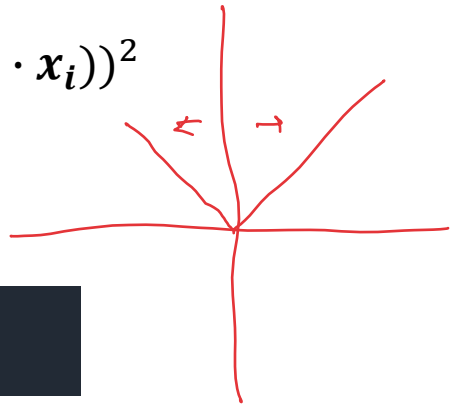
```
from scratch.statistics import daily_minutes_good
from scratch.gradient_descent import gradient_step

learning_rate = 0.001

def least_squares_fit_ridge(xs: List[Vector],
                            ys: List[float],
                            alpha: float,
                            learning_rate: float,
                            num_steps: int,
                            batch_size: int = 1) -> Vector:
    # Start guess with mean
    guess = [random.random() for _ in xs[0]]

    for i in range(num_steps):
        for start in range(0, len(xs), batch_size):
            batch_xs = xs[start:start+batch_size]
            batch_ys = ys[start:start+batch_size]

            gradient = vector_mean([sqerror_ridge_gradient(x, y, guess, alpha)
                                    for x, y in zip(batch_xs, batch_ys)])
            guess = gradient_step(guess, gradient, -learning_rate)

    return guess
```

그래디언트 계산 부분만 변화

라소 회귀 (Lasso Regression)

리지 회귀와 같은 방식으로 구현

라소 회귀 (Lasso) $\tilde{J}(\boldsymbol{\beta}) = J(\boldsymbol{\beta}) + \lambda \| \boldsymbol{\beta} \|_1$ $J(\boldsymbol{\beta}) = \sum_{i=1}^{N} {e_i}^2 = \sum_{i=1}^{N} (y_i - (\boldsymbol{\beta} \cdot \boldsymbol{x_i}))^2$



라소 패널티

```
def lasso_penalty(beta, alpha):
    return alpha * sum(abs(beta_i) for beta_i in beta[1:])
```

경사 하강법을 적용하려면 양수 구간과 음수 구간을 나눠서 그래디언트를 구해야 함

Thank you!