조 이름: 1조 (김예진, 김지웅, 박효정, 홍소은, 편민우)

2주차 과제

1-1) 최적의 람다값 구하기

```python
import numpy as np

def find_optimal_lambda(data):
    n = len(data)
    optimal_lambda = n / np.sum(data)
    return optimal_lambda

# 데이터
data = np.array([3, 5, 7, 18, 43, 85, 91, 98, 100, 130, 230, 487])

# 최적의 람다 계산
optimal_lambda = find_optimal_lambda(data)

print("1.1 최적의 람다:", optimal_lambda)
```

```
1.1 최적의 람다: 0.009252120277563608
```

- `find_optimal_lambda` 함수에서는 입력으로 주어진 데이터 배열에 대해 최적의 람다 값을 계산하는 함수로서, 최적의 람다는 데이터 총합을 개수로 나눈 값으로 계산이 된다. (쉽게 말해 데이터의 갯수 n을 전체 데이터의 합으로 나눈 값을 반환하는 함수이다. → 이를 통해 지수 분포의 최대 우도 추정값을 계산하는 공식을 간단하게 표현한것임)

$$\text{최적의 람다} = \frac{n}{\sum_{i=1}^{n} x_i}$$

- 여기서 n은 데이터의 개수, Xi는 각 데이터 포인트 이다.

- `optimal_lambda` 부분에서는 `find_optimal_lambda` 함수를 사용해 주어진 데이터에 대한 최적의 람다를 계산하고, 최적의 람다 값은 주어진 데이터를 가장 잘 설명하는 지수분포 모델에서의 람다값으로 해석됨.

1-2) 뉴턴법 이용해 최적의 람다 구하는 알고리즘

- 뉴턴법(**뉴턴-라프슨 방법**) : 최적화 문제에서 함수의 최솟값을 착기 위한 반복적인 방법 중 하나로, 현재 추정값에 대한 함수의 그래디언트(도함수)와 헤시안(이차 도함수) 정보를 사용하여 다음 추정값을 계산하는 방법

```
import numpy as np

# 시간 데이터
data = np.array([3, 5, 7, 18, 43, 85, 91, 98, 100, 130, 230, 487])

# 손실 함수와 그래디언트 정의
def neg_log_likelihood(lambda_value):
    return -np.sum(np.log(lambda_value * np.exp(-lambda_value * data)))

def neg_log_likelihood_gradient(lambda_value):
    return -np.sum(1 - lambda_value * data)

# 초기 추정값
initial_lambda = 0.1

# 뉴턴법을 이용하여 최적의 람다 찾기
lambda_value = initial_lambda
tolerance = 1e-6
max_iterations = 100

for iteration in range(max_iterations):
    gradient = neg_log_likelihood_gradient(lambda_value)
    hessian = np.sum(lambda_value * data ** 2)

    lambda_value -= gradient / hessian

    if np.abs(gradient) < tolerance:
        break

optimal_lambda = lambda_value
print("1.2 최적의 람다:", optimal_lambda)
```

```
1.2 최적의 람다: 0.009252120555149602
```

- `neg_log_likelihood` 음의 로그 우도 함수를 정의한 함수로, 주어진 시간 데이터에 대한 지수분포 모델의 음의 로그 우도를 계산 (음의 로그 우도 함수는 데이터 포인트 각각에 대해 람다 * e^(-람다*데이터)를 계산한 다음 이들의 로그를 취한 후 합산한 값에 음의 부호를 붙이는 것으로, 주로 MLE(지수 분포에 대한 최대 우도 추정)에 사용되는 수식으로, 우도 함수의 로그를 취해서 계산을 간소화 하고, 음의 부호를 붙여서 문제를 최소화 문제로 바꾸는 방식이다)
- `neg_log_likelihood_gradient` 음의 로그 우도 함수의 그래디언트(도함수)를 계산하는 함수
- `initial_lambda` 에서 초기 람다의 추정값을 0.1로 설정하고 뉴턴-라프슨 방법(뉴턴법)을 사용하여 최적의 람다 값을 찾음(함수의 그래디언트 와 헤시안을 사용하여 함수의 최소값을 찾아냄), 그래디언트가 허용된 오차보다 작아질 때 까지 람다 값을 업데이트해줌

2) SVD

```
import numpy as np
from numpy.linalg import svd

#특잇값 분해는 svd함수 사용
A = np.array([[3, 0],[4, 5]])
A
```

```
array([[3, 0],
       [4, 5]])
```

```
U, S, VT = svd(A)
```

```
#U출력
U
```

```
array([[-0.31622777, -0.9486833 ],
       [-0.9486833 ,  0.31622777]])
```

```
#S출력
S
```

```
array([6.70820393, 2.23606798])
```

```
#VT출력
VT
```

```
array([[-0.70710678, -0.70710678],
       [-0.70710678,  0.70710678]])
```

3) SVD 근사 행렬

```
import numpy as np
from numpy.linalg import svd

def prove_svd_approximation(A, r):
    U, Sigma, VT = svd(A)

    Ur = U[:, :r]
    Sigma_r = np.diag(Sigma[:r])
    Vr_T = VT[:r, :]

    A_approximation = Ur @ Sigma_r @ Vr_T
    return A_approximation
```

```
A = np.array([[2, 4, 0],
              [1, 3, 5]])
r = 1

A_proven = prove_svd_approximation(A, r)

print("원래 행렬 A:")
print(A)

print("₩n근사행렬 A:")
print(A_proven)
```

```
원래 행렬 A:
[[2 4 0]
 [1 3 5]]

근사행렬 A:
[[0.96851857 2.37777713 2.20369993]
 [1.61759068 3.97129206 3.68055353]]
```

4) L2 규제 포함한 로지스틱 모델 구현

p90 로지스틱

In [18]:
```
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

X, y = load_breast_cancer(return_X_y=True, as_frame = False)
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.33,
                                                    random_state=1234)

X_train, X_test = X_train[:, :3], X_test[:, :3]
y_train, y_test = y_train.reshape(-1, 1), y_test.reshape(-1, 1)
```

In [19]:
```
train_mean, train_std = X_train.mean(axis=0), X_train.std(axis=0)

X_train, X_test = (X_train - train_mean) / train_std, (X_test -
                                                       train_mean) / train_std

n, n_test = X_train.shape[0], X_test.shape[0]
X_train, X_test = np.append(np.ones((n, 1)), X_train,
                            axis=1), np.append(np.ones((n_test, 1)),
                                               X_test,
                                               axis=1)
```

In [20]:
```
max_iter = 10000
Tolerance = 0.0001
```

In [21]:
```
beta_old = np.ones((4, 1))
```

In [22]:
```
for cnt in range(1, max_iter):
    W = np.zeros((n, n))
    p = np.zeros((n, 1))

    for i in range(n):
        xb = np.exp((X_train[i].reshape(1, -1) @ beta_old)[0][0])
        pi = xb / (1 + xb)

        W[i][i] = pi * (1 - pi)
        p[i] = pi

    left = np.linalg.inv(X_train.T @ W @ X_train)
    right = X_train.T @ (y_train - p)

    update = 0.0001 * (left @ right)
    beta_new = beta_old + update

    if (np.linalg.norm(update) < Tolerance): break
    if cnt % 1000 == 0:
        print(f'이터레이션: {cnt}, 업데이트 크기:{np.linalg.norm(update)}')
    beta_old = beta_new

print(f'이터레이션: {cnt}, 업데이트 크기: {np.linalg.norm(update)}')
print('\n학습한 파라미터:\n', beta_new)
```

```
이터레이션 : 1000, 업데이트 크기 :0.0014765820896821
이터레이션 : 2000, 업데이트 크기 :0.0008288194941221
이터레이션 : 3000, 업데이트 크기 :0.0005888637557052267
이터레이션 : 4000, 업데이트 크기 :0.0005016460961680756
이터레이션 : 5000, 업데이트 크기 :0.0004405887201861744
이터레이션 : 6000, 업데이트 크기 :0.0004921690754541083
이터레이션 : 7000, 업데이트 크기 :0.0005216582869965073
이터레이션 : 8000, 업데이트 크기 :0.0005614510237058704
이터레이션 : 9000, 업데이트 크기 :0.0006068863712746636
이터레이션 : 9999, 업데이트 크기 : 0.0006545214496912951

학습한 파라미터:
 [[ 4.96767147e-01]
 [ 5.07895825e+00]
 [ 2.22184378e-03]
 [-5.65906436e+00]]
```

In [23]:
```
right = 0
for i in range(X_train.shape[0]):
    xb = np.exp((X_train[i].reshape(1, -1) @ beta_old)[0][0])
    pi = xb / (1 + xb)
    if (pi >= 0.5 and y_train[i] == 1) or (pi < 0.5 and y_train[i] == 0):
        right += 1

print(f'학습 데이터셋 정확도:{right / X_train.shape[0] * 100: .2f}%')

right = 0
for i in range(X_test.shape[0]):
    xb = np.exp((X_test[i].reshape(1, -1) @ beta_old)[0][0])
    pi = xb / (1 + xb)
    if (pi >= 0.5 and y_test[i] == 1) or (pi < 0.5 and y_test[i] == 0): right += 1

print(f'테스트 데이터셋 정확도:{right / X_test.shape[0] * 100: .2f}%')
```

```
학습 데이터셋 정확도: 84.51%
테스트 데이터셋 정확도: 76.06%
```

l2 패널티 로지스틱

In [44]:
```
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

X, y = load_breast_cancer(return_X_y=True, as_frame = False)
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.33,
                                                    random_state=1234)

X_train, X_test = X_train[:, :3], X_test[:, :3]
y_train, y_test = y_train.reshape(-1, 1), y_test.reshape(-1, 1)
```

- 로지스틱 회귀 파라미터에 위와 같이 max_iter는 10000으로 지정
- penalty 파라미터를 통해 l2 규제 추가

In [45]:
```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter = 10000, penalty = 'l2')
model.fit(X_train, y_train)
```

```
C:₩Users₩01che₩anaconda3₩lib₩site-packages₩sklearn₩utils₩validation.py:993: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

Out[45]:
```
LogisticRegression(max_iter=10000)
```

In [46]:
```
print('학습 데이터셋 정확도: {:.2f}%'.format(model.score(X_train, y_train) * 100))
print('테스트 데이터셋 정확도: {:.2f}%'.format(model.score(X_test, y_test) * 100))
```

```
학습 데이터셋 정확도: 91.60%
테스트 데이터셋 정확도: 86.70%
```

l2 패널티 강화

In [47]:
```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter = 10000, penalty = 'l2', C = 0.01)
model.fit(X_train, y_train)
```

```
C:₩Users₩01che₩anaconda3₩lib₩site-packages₩sklearn₩utils₩validation.py:993: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

Out[47]:
```
LogisticRegression(C=0.01, max_iter=10000)
```

In [48]:
```
print('학습 데이터셋 정확도: {:.2f}%'.format(model.score(X_train, y_train) * 100))
print('테스트 데이터셋 정확도: {:.2f}%'.format(model.score(X_test, y_test) * 100))
```

```
학습 데이터셋 정확도: 90.55%
테스트 데이터셋 정확도: 88.30%
```

결론

- 일반 로지스틱 회귀에 비해 l2규제가 적용된 로지스틱에서 train set과 test set 모두의 설명력이 개선됨을 알 수 있다.
- l2 패널티를 강화하면 과소적합의 문제가 있어 학습 데이터셋의 정확도는 떨어지지만, 테스트셋의 정확도는 올라간다.
  - 하지만 여기의 데이터에서만 적용된 결과일 수 있으니 상황에 맞게 패널티를 완화하거나 강화하는 것을 선택해야 한다.

5) 3장 되새김 문제 2번

- 5. 로지스틱 회귀 모델용 데이터셋 생성하기

로지스틱 함수 (logistic function):

로지스틱 함수의 그래프는 Figure 1과 같고 이는 독립 변수 x가 주어졌을 때 종속 변수가 1의 범주에 속할 확률을 의미.

즉, p(y=1|x).

로지스틱 함수는 로짓 변환을 통해 만들어진다.

$$p(y = 1|x)$$

$$\text{logistic function} = \frac{e^{\beta \cdot X_i}}{1 + e^{\beta \cdot X_i}}$$

In [1]:
```
import numpy as np
import pandas as pd

n = 300
np.random.seed(1234)
X1 = np.random.normal(0, 1, size=n)
X2 = np.random.normal(0, 1, size=n)
X3 = np.random.normal(0, 1, size=n)
X4 = np.random.normal(0, 1, size=n)
X5 = np.random.normal(0, 1, size=n)

X = pd.DataFrame({'X1': X1, 'X2': X2, 'X3': X3, 'X4': X4, 'X5': X5})
```

In [5]:
```
ys = [] # 각 행의 예측값을 저장할 빈 리스트를 초기화
intercept = -2 # 로지스틱 회귀 모델의 절편 설정
beta = np.array([[-2, -2, 3, 4, 5]]).reshape(-1, 1) # 로지스틱 회귀 모델의 계수 설정

np.random.seed(1111) # 난수 생성의 시드 설정 (결과 재현)
for i in range(n):
    xb = np.exp((intercept + (X.iloc[i].values.reshape(1, -1) @ beta)[0][0]) +
                np.random.normal(0, 7.5)) # 로지스틱 함수의 선형 부분 계산 후 노이즈를 추가하여 예측값 생성
    pi = xb / (1 + xb) # 확률 계산
    if pi >= 0.5: y = 1
    else: y = 0 # 확률이 0.5이상이면 1로 그렇지 않으면 0으로 할당
    ys.append(y)
y = pd.Series(ys) # 데이터 저장
```

In [6]:
```
from sklearn.linear_model import LogisticRegression # 로지스틱 회귀 모델 훈련

clf = LogisticRegression(random_state=1234) # 로지스틱 회귀 모델 객체 생성
clf = clf.fit(X, y) # 특성 행렬 x와 목표변수 y 사용

print(f'절편: {clf.intercept_[0]}') # 훈련된 모델의 절편 출력
print(f'계수: {clf.coef_[0]}') # 훈련된 모델의 계수 출력

y_pred = clf.predict(X) # 훈련된 모델을 사용해 입력 데이터에 대한 예측 수행
print(f'정확도:{(y == y_pred).mean() * 100: .2f}%') # 예측된 클래스와 실제 클래스를 비교하여 정확도 계산
```

```
절편: -0.578686940895635
계수: [-0.52203159 -0.51494024  0.71392631  1.05416056  1.06007824]
정확도: 78.00%
```

6) Mallow's Cp 근사

- 회귀 모델에서 설명 변수가 많을 경우, 줄여줄 필요 있음 (많으면 성능 하락, 다중 공산성 문제)
- 그럼 설명 변수를 줄여주는 방법은?
  - 변수 선택 방법
    1. Subset Selection
       - AIC, BIC, Mallow's Cp
    2. Shrinkage
    3. Dimension Reduction

Mallow's C_p

$$C_p = \frac{RSS_p}{\sigma^2} + 2p - n \;\; ; \;\; RSS_p\text{: } p\text{변수시의 } Residual\ Sum\ of\ Squares$$

(p= 예측 변수 개수, N= 데이터 개수, S^2=전체 k개의 예측 변수 이용한 RMS, SSE_p=p개의 예측 변수 사용한 잔차제곱 합)

- Cp는 최소자승법을 이용하여 추정된 회귀 모델의 적합성을 평가하는 데 사용하는 지표
- C_p는 bias가 적은 방향으로 변수 선택
- 변수의 수와 비슷할수록 좋다고 판단

  ((변수 개수+상수 개수)의 차이가 작은 모형을 최적으로 산정)
- 변수를 추가하면 RSS 감소하지만, 2p가 패널티로 적용하여 p개로 fitting 했을 때, Cp가 p보다 작으면 좋은 모델로 선택하는 것

⇒ Cp ≈ p 일수록 bias 없다고 판단!

AIC(Akaike Information Criterion)

- 변수가 많은 모델이라면(p가 크다면) RSS 작아짐

⇒ AIC, BIC를 최소화 한다 = likelihood를 가장 크게 하는 동시에, 변수 개수는 가장 적은 최적의 모델

$$AIC = -2\ LogLikelihood + 2p$$

- -2Log(Likelihood) = 모형의 적합도
- p = 모형의 추정된 파라미터의 개수, 해당 모형에 페널티 주기 위해 사용
- 어떤 모형의 적합도를 높이기 위해, 여러 불필요한 파라미터를 사용할 수도 있음
- 실제 모형 비교 시 독립 변수가 많은 모형이 적합도 면에서 유리하게 되는데, 독립 변수에 따라 모형의 적합도에 차이가 난다는 뜻
- 이를 상쇄시키기 위해 독립 변수의 수가 증가할수록 2k를 증가 시켜, 페널티를 부여하여 모델의 품질 평가

BIC(Bayes Information Criterion)

$$BIC = -2\ LogLikelihood + \log(n)p$$

- 마지막 페널티를 수정하여 AIC를 보완한 형태
- 변수가 많을수록 AIC보다 더 페널티를 가하는 형태

⇒ AIC보다 변수 증가에 더 민감하여, 변수 개수가 작은 것이 우선 순위라면 BIC 참고하는 것이 좋음

- 선형 회귀 모델 $Y = X\beta + \varepsilon$ 일 때, 선형 모델 M(M: Y~N)이 있다고 한다.
- 충분히 작은 x의 경우, 로그 변환을 사용하여 $\log(1+x) \approx x$로 근사할 수 있다.
- $\beta_j = 0$ 일 때, 모델 M의 AIC를 Mallow's Cp로 근사해 보이시오.

1. $AIC = -2logL(M) + 2(|M| + 1)$
2. $\beta_j = 0 \Rightarrow$ 보수 매개 변수 모두 0
3. 즉, $-2logL(M)$는 단순화, 이를 AIC_M로 대체 가능
4. 로그 변환 근사 적용하여 $\Rightarrow Cp = -2logL(M)$ 근사화 가능
5. 최종 근사치 $AIC_M \fallingdotseq C_p + AIC_{(M_C)} - AIC_M$

7) 4장 되새김 문제 2번

- 7. 불필요한 피처가 많은 상황에서 선형 모델 비교하기

라쏘회귀는 릿지회귀와 비슷하게 생겼지만 패널티 항에 절대값의 합을 준다. 라쏘회귀는 파라미터의 크기에 관계없이 같은 수준의 Regularization을 적용하기 때문에 작은 값의 파라미터를 0으로 만들어 해당 변수를 모델에서 삭제하고 따라서 모델을 단순하게 만들어주고 해석에 용이하게 만든다.

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2+\lambda\sum_{j=1}^{p}|\beta_j| \;=\mathrm{RSS}+\lambda\sum_{j=1}^{p}|\beta_j|.$$

릿지회귀와 라쏘회귀의 차이

Ridge	Lasso
L_2-norm regularization	L_1-norm regularization
변수 선택 불가능	변수 선택 가능
Closed form solution 존재 (미분으로 구함)	Closed form solution이 존재하지 않음 (numerical optimization 이용)
변수 간 상관관계가 높은 상황 (collinearity)에서 좋은 예측 성능	변수 간 상관관계가 높은 상황에서 ridge에 비해 상대적으로 예측 성능이 떨어짐
크기가 큰 변수를 우선적으로 줄이는 경향이 있음	

In [3]:
```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Lasso

X, y = make_regression(n_samples=300,
                       n_features=400,
                       n_informative=50,
                       n_targets=1,
                       bias=0.0,
                       noise=10.0,
                       random_state=1234)
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.33,
                                                    random_state=1234)
```

In [4]:
```
reg1 = LinearRegression() # 선형 회귀 모델 객체 생성
reg1 = reg1.fit(X_train, y_train)
y_train_pred = reg1.predict(X_train) # 훈련 데이터셋과 목표 변수를 사용해 선형 회귀 모델 훈련
print(f'학습 데이터셋 기준 OLS 모델의 MSE:{((y_train - y_train_pred)**2).mean(): .2f}')

y_test_pred = reg1.predict(X_test) # 훈련된 모델을 사용하여 학습 데이터셋에 대한 예측 수행
print(f'테스트 데이터셋 기준 OLS 모델의 MSE:{((y_test - y_test_pred)**2).mean(): .2f}')
```

```
학습 데이터셋 기준 OLS 모델의 MSE: 0.00
테스트 데이터셋 기준 OLS 모델의 MSE: 79374.97
```

In [5]:
```
reg2 = Lasso() # Lasso 회귀 모델 객체 생성
reg2 = reg2.fit(X_train, y_train) # 훈련 데이터셋과 목표 변수를 사용해 학습 데이터셋에 대한 예측 수행
y_train_pred = reg2.predict(X_train) # 훈련된 모델을 사용해 학습 데이터셋에 대한 예측 수행
print(f'학습 데이터셋 기준 LASSO 모델의 MSE:{((y_train - y_train_pred)**2).mean(): .2f}')

y_test_pred = reg2.predict(X_test) # 훈련된 모델을 사용하여 테스트 데이터셋에 대한 예측 수행
print(f'테스트 데이터셋 기준 LASSO 모델의 MSE:{((y_test - y_test_pred)**2).mean(): .2f}')
```

```
학습 데이터셋 기준 LASSO 모델의 MSE: 130.41
테스트 데이터셋 기준 LASSO 모델의 MSE: 457.40
```