5️⃣

Week5_5장:회귀

1. 회귀 소개

회귀 분석은 데이터 값이 평균과 같은 일정한 값으로 돌아가려는 경향을 이용한 통계학 기법이다.

회귀는 여러 개의 독립변수와 한 개의 종속변수 간의 상관관계를 모델링하는 기법을 통칭함. 예를 들어, 아파트의 방 개수, 방 크기, 주변 학군 등 여러 개의 독립변수에 따라 아파트 가격이라는 종속변수가 어떤 관계를 나타내는지를 모델링하고 예측하는 것.

$Y = W_1 * X_1 + W_2 * X_2 + \cdots + W_n * X_n$ 이라는 선형 회귀식을 예로 들면, Y는 종속변수(아파트의 가격)을 의미하고, $X_1 \cdots X_n$은 방 개수, 학군 등의 독립변수, $W_1 \cdots W_n$은 독립변수의 값에 영향을 미치는 **회귀 계수(Regression coefficients)**이다. 머신러닝 관점에서 보

면 독립변수는 피처에 해당되며 종속변수는 결정값이다. 머신러닝 회귀 예측의 핵심은 주어진 피처와 결정 값 데이터 기반에서 학습을 통해 **최적의 회귀 계수**를 찾아내는 것이다.

독립변수 개수	회귀 계수의 결합
1개: 단일 회귀	선형: 선형 회귀
여러 개: 다중 회귀	비선형: 비선형 회귀

지도학습은 두 가지 유형으로 나뉘는데, 바로 분류와 회귀이다. 둘의 가장 큰 차이는 분류는 예측값이 카테고리와 같은 이산형 클래스 값이고 회귀는 연속형 숫자 값이라는 것이다.

Classification	Category 값(이산값)
Regression	숫자값(연속값)

여러 가지 회귀 중에서 선형 회귀가 가장 많이 사용됨. **선형 회귀**는 실제 값과 예측값의 차이(오류의 제곱 값)을 최소화하는 직선형 회귀선을 최적화하는 방식이다. 선형 회귀 모델은 규제 방법에 따라 다시 별도의 유형으로 나뉠 수 있다. **규제**는 일반적인 선형 회귀의 과적합 문제를 해결하기 위해서 회귀 계수에 패널티 값을 적용하는 것을 말함. 대표적인 선형 회귀 모델은 다음과 같음.

- 일반 선형 회귀: 예측값과 실제값의 RSS(Residual Sum of Squares)를 최소화할 수 있도록 회귀계수를 최적화하며, 규제(Regularization)을 적용하지 않은 모델.
- **릿지(Ridge)**: 릿지 회귀는 선형 회귀에 L2 규제를 추가한 회귀 모델. 릿지 회귀는 L2 규제를 적용하는데, L2는 상대적으로 큰 회귀 계수 값의 예측 영향도를 감소시키기 위해서 회귀계수값을 더 작게 만드는 규제 모델.
- **라쏘(Lasso)**: 라쏘 회귀는 선형회귀에 L1 규제를 적용한 방식. L2 규제가 회귀 계수 값의 크기를 줄이는 데 반해, L1 규제는 예측 영향력이 작은 피처의 회귀 계수를 0으로 만들어 회귀 예측 시 피처가 선택되지 않게 하는 것이다. 이러한 특성 때문에 L1 규제는 피처 선택 기능으로도 불림.
- 엘라스틱넷(ElasticNet): L2, L1 규제를 함께 결합한 모델. 주로 피처가 많은 데이터 세트에 적용되며, L1 규제로 피처의 개수를 줄이고 L2규제로 계수 값의 크기를 조정함.
- 로지스틱 회귀(Logistic Regression): 로지스틱 회귀는 회귀라는 이름이 붙어 있지만, 사실은 분류에 사용되는 선형 모델이다. 로지스틱 회귀는 매우 강력한 분류 알고리즘이다. 일반적으로 이진 분류뿐만 아니라 희소 영역의 분류, 예를 들어 텍스트 분류와 같은 영역에서 뛰어난 예측 성능을 보임.

4.5 Regularization(과적합 규제)

$J(\theta) = \frac{1}{2m}\sum(h_\theta(x^{(i)}) - y^{(i)})^2$ → 이 Cost function에 어떤 항이 추가적으로 붙느냐에 따라 L1, L2 Regularization을 구분

L1 Regularization(Lasso)

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n}|\theta_j|$$

Gradient Descent 식에서 위 Cost function 을 접목

→ W를 업데이트 할 때, **계속해서 특정 상수를 빼주는 꼴**

→ W를 계속해서 업데이트 해나갈 때, W의 원소인 **어떤 wi 는 0 이 되도록 함.**(계속해서 특정상수를 빼나가기 때문)

→ **영향을 크게 미치는 핵심적인 피처 xi 들만 반영하도록 함!**

L2 Regularization(Ridge)

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2$$

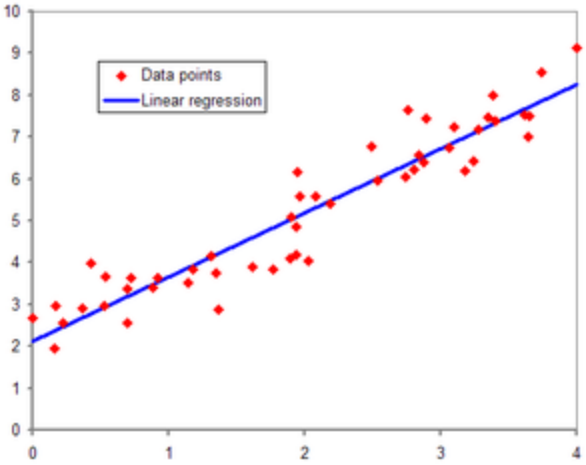
미분하여 Gradient Descent 식 에서 사용

→ **전체적으로 W 값이 작아지도록 함.**

→ Lasso 같이 일부 항의 계수를 0으로 만들어버리지는 않고, **전체적인 wi 값의 절대값을 감소시켜 덜 구불구불하게 하는 것**

2. 단순 선형 회귀를 통한 회귀 이해

단순 선형 회귀는 독립변수도 하나, 종속변수도 하나인 선형 회귀이다. 예를 들어, 주택 가격이 주택의 크기로만 결정된다면 아래와 같은 선형의 관계로 표현할 수 있다.



모델이 $f(x) = w_0 + w_1 * x$이면 예측값 $\hat{Y}$는 $w_0 + w_1 * x$로 계산할 수 있음. 독립변수가 1개인 단순 선형 회귀에서는 이 기울기 w1과 절편 w0을 회귀 계수로 지칭함. (절편은 영어로

intercept). 그리고 회귀 모델을 이러한 $\hat{Y} = w_0 + w_1 * x$와 같은 1차 함수로 모델링했다면 실제 주택 가격은 이러한 1차 함수 값에서 실제 값만큼의 오류 값을 뺀(또는 더한) 값이 됨.

실제 값과 회귀 모델의 차이에 따른 오류 값을 남은 오류, 즉 **잔차**라고 함. 최적의 회귀 모델을 만든다는 것은 전체 데이터의 잔차(오류 값)의 합이 최소가 되는 모델을 만든다는 의미이고, 동시에 오류 값 합이 최소가 될 수 잇는 최적의 회귀 계수를 찾는다는 의미도 됨.

오류 값은 + 나 -가 될 수 있기 때문에, 보통 오류 합을 계산할 때는 절대값을 취해서 더하거나 (Mean Absolute Error), **오류 값의 제곱을 구해서 더하는 방식(RSS, Residual Sum of Sqaure)**을 취함. 일반적으로 미분 등의 계산을 편리하게 하기 위해서는 RSS 방식으로 오류 합을 구함. 즉, $Error^2 = RSS$이다.

RSS는 변수가 w_0, w_1인 식으로 표현할 수 있으며, 이 R**SS를 최소로 하는 w_0, w_1, 즉 회귀 계수를 학습을 통해서 찾는 것이 머신러닝 기반 회귀의 핵심 사항이다.** RSS는 회귀식의 독립변수 X, 종속변수 Y가 중심 변수가 아니라 **w 변수(회귀 계수)가 중심 변수임을 인지**하는 것이 매우 중요함.(학습 데이터로 입력되는 독립변수와 종속변수는 RSS에서 모두 상수로 간주함). 일반적으로 RSS는 학습데이터의 건수로 나누어서 다음과 같이 정규화된 식으로 표현함.

$$RSS(w_0, w_1) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (w_o + w_1 * x_i))^2$$

-i는 1부터 학습 데이터의 총 건수 N까지

회귀에서 이 **RSS는 비용(Cost)**이며 w변수(회귀 계수)로 구성되는 RSS를 비용 함수라고 함. 머신러닝 회귀 알고리즘은 데이터를 계속 학습하면서 이 비용 함수가 반환되는 값(즉, 오류 값)을 지속해서 감소시키고 최종적으로는 더 이상 감소하지 않는 최소의 오류 값을 구하는 것이다. **비용 함수를 손실함수(loss function)**이라고도 함.

3. 비용 최소화하기 - 경사하강법(Gradient Descent) 소개

비용 함수가 최소화되는 W 파라미터를 어떻게 구할까? 경사 하강법은 고차원 방정식에 대한 문제를 해결해 주면서 비용 함수 RSS를 최소화하는 방법을 직관적으로 제공하는 뛰어난 방식이다.

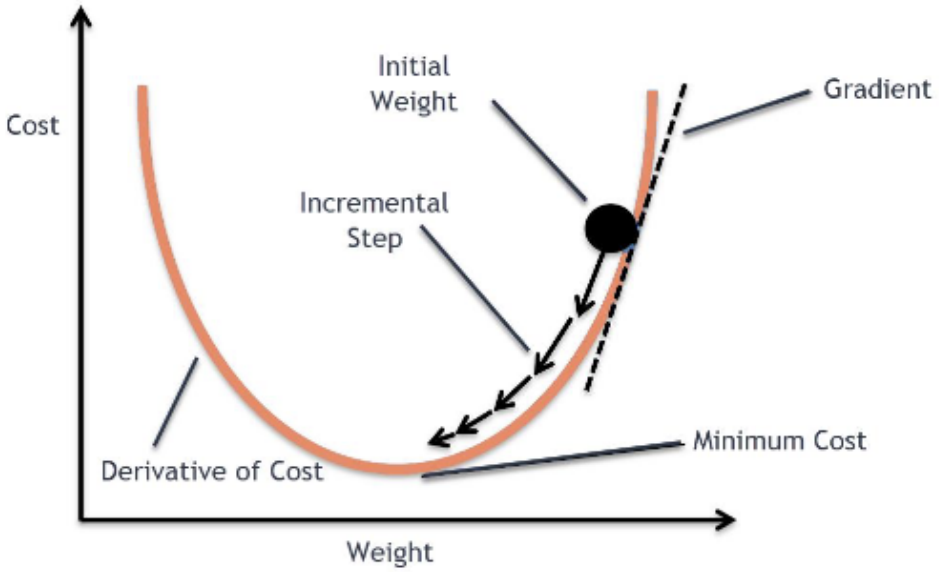
경사 하강법은 '데이터를 기반으로 알고리즘이 스스로 학습한다'는 머신러닝 개념을 기능하게 만들어준 핵심 기법의 하나이다. 경사 하강법의 사전적 의미인 '점진적인 하강'이라는 뜻에서도

알 수 있듯이, '점진적으로' **반복적인 계산을 통해 W 파라미터 값을 업데이트하면서 오류 값이 최소가 되는 W 파라미터를 구하는 방식**이다.

경사 하강법은 반복적으로 비용 함수의 반환 값, 즉 예측값과 실제 값의 차이가 작아지는 방향성을 가지고 W 파라미터를 지속해서 보정해 나감. 그리고 오류 값이 더이상 작아지지 않으면 그 오류 값을 최소 비용으로 판단하고 그때의 W 값을 최적 파라미터로 반환함.

(2차원 함수의 최저점은 해당 2차 함수의 미분 값인 1차 함수의 기울기가 가장 최소일 때이다)

예를 들어 비용함수가 2차 함수라면 경사 하강법은 최초 w에서부터 미분을 적용한 뒤 이 미분 값이 계속 감소하는 방향으로 순차적으로 w를 업데이트한다. 마침내 더 이상 미분된 1차 함수의 기울기가 감소하지 않는 지점을 비용 함수가 최소인 지점으로 간주하고 그때의 w를 반환함.



출처: https://towardsdatascience.com/using-machine-learning-to-predict-fitbit-sleep-scores-496a7d9ec48

https://velog.io/@ghj616/머신러닝-경사-하강법-Gradient-descent-method

비용 함수 $RSS(w_0, w_1)$를 각각 w_0, w_1에 대해 편미분 하는데, 편미분 값이 너무 클 수 있기 때문에 보정 계수 n을 곱하는데, 이를 학습률이라고 함.

<경사하강법의 일반적인 프로세스>

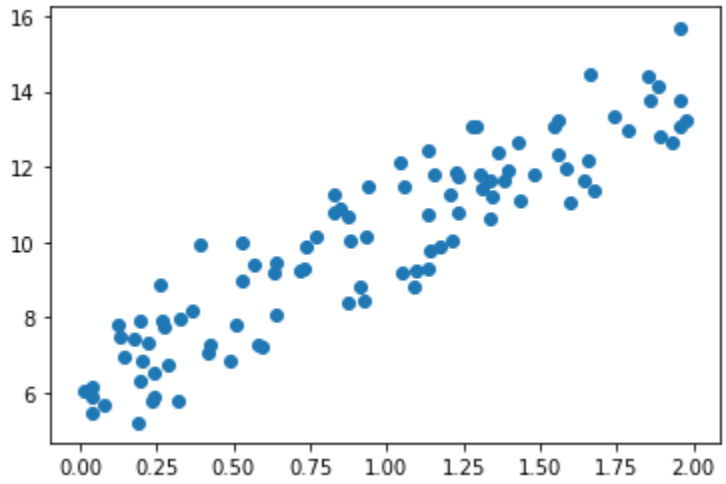
1. w_0, w_1를 임의의 값으로 설정하고 첫 비용 함수의 값을 계산함

2. w_1을 $w_1 + n\frac{2}{N}\sum_{i=1}^{N} x_i * (\text{실제값}_i - \text{예측값}_i)$, w_0을 $w_0 + n\frac{2}{N}\sum_{i=1}^{N} x_i * (\text{실제값}_i - \text{예측값}_i)$으로 예측을 업데이트 한 후 다시 비용함수의 값을 계산함

3. 비용 함수의 값이 감소했으면 다시 step2를 반복함. 더이상 비용함수의 값이 감소하지 않으면 그때의 w_0, w_1를 구하고 반복을 중지함.

파이썬으로 구현해보겠음. 간단한 회귀식인 y=4X+6을 근사하기 위한 100개의 데이터 셋을 만들고, 여기에 경사 하강법을 이용해 회귀 계수 w_0, w_1를 도출하는 것. 단순 선형 회귀로 예측할 만한 데이터 세트를 먼저 만들어보겠음.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

np.random.seed(0)
# y = 4X + 6 식을 근사(w1=4, w0=6). random 값은 Noise를 위해 만듬(임의의 값은 노이즈를 위해 만듦)
X = 2 * np.random.rand(100,1)
y = 6 +4 * X+ np.random.randn(100,1)

# X, y 데이터 셋 scatter plot으로 시각화
plt.scatter(X, y)
```



데이터는 y = 4X + 6을 중심으로 무작위로 퍼져있음.

~~다음으로 비용 함수 정의. 비용 함수 get_cost()는 실제 y값과 예측된 y 값을 인자로 받아~~ $\frac{1}{N}\sum_{i=1}^{N}(\text{실제값}_i - \text{예측값}_i)^2$ ~~을 계산해 반환함.~~

```
def get_cost(y, y_pred):
  N = len(y)
  cost = np.sum(np.square(y - y_pred))/N
  return cost
```

이제 경사하강법을 gradient_descent()라는 함수를 생성해 구현하겠음.
gradient_descent()는 w_0과 w_1를 모두 0으로 초기화한 뒤 iters 개수만큼 반복하면서 w_0과 w_1을 업데이트 함. 즉 새로운 w_0과 w_1를 반복적으로 적용하면서 업데이트 하는 것.
gradient_descent()는 위에서 무작위로 생성한 X와 y를 입력받는데, X와 y모두 넘파이 ndarray이다.

w0과 w1의 값을 최소화 할 수 있도록 업데이트 수행하는 함수 생성.

- 예측 배열 y_pred는 np.dot(X, w1.T) + w0 임 100개의 데이터 X(1,2,...,100)이 있다면 예측값은 w0 + X(1)*w1 + X(2)*w1 +..+ X(100)*w1이며, 이는 입력 배열 X와 w1 배열의 내적임.
- 새로운 w1과 w0를 update함

```
# w1 과 w0 를 업데이트 할 w1_update, w0_update를 반환.
def get_weight_updates(w1, w0, X, y, learning_rate=0.01):
    N = len(y)
    # 먼저 w1_update, w0_update를 각각 w1, w0의 shape와 동일한 크기를 가진 0 값으로 초기화
```

```
    w1_update = np.zeros_like(w1)
    w0_update = np.zeros_like(w0)
    # 예측 배열 계산하고 예측과 실제 값의 차이 계산
    y_pred = np.dot(X, w1.T) + w0
    diff = y-y_pred

    # w0_update를 dot 행렬 연산으로 구하기 위해 모두 1값을 가진 행렬 생성
    w0_factors = np.ones((N,1))

    # w1과 w0을 업데이트할 w1_update와 w0_update 계산
    w1_update = -(2/N)*learning_rate*(np.dot(X.T, diff))
    w0_update = -(2/N)*learning_rate*(np.dot(w0_factors.T, diff))

    return w1_update, w0_update
```

```
w0 = np.zeros((1,1))
w1 = np.zeros((1,1))
y_pred = np.dot(X, w1.T) + w0
diff = y-y_pred
print(diff.shape)  # (100, 1)
w0_factors = np.ones((100,1))
w1_update = -(2/100)*0.01*(np.dot(X.T, diff))
w0_update = -(2/100)*0.01*(np.dot(w0_factors.T, diff))
print(w1_update.shape, w0_update.shape)  # (1, 1) (1, 1)
w1, w0  # (array([[0.]]), array([[0.]]))
```

다음은 get_weight_updates()을 경사 하강 방식으로 반복적으로 수행하여 w_0, w_1를 업데이트 하는 함수인 gradient_descent_steps() 함수 생성

```
# 입력 인자 iters로 주어진 횟수만큼 반복적으로 w1과 w0를 업데이트 적용함.
def gradient_descent_steps(X, y, iters=10000):
    # w0와 w1을 모두 0으로 초기화.
    w0 = np.zeros((1,1))
    w1 = np.zeros((1,1))

    # 인자로 주어진 iters 만큼 반복적으로 get_weight_updates() 호출하여 w1, w0 업데이트 수행.
    for ind in range(iters):
        w1_update, w0_update = get_weight_updates(w1, w0, X, y, learning_rate=0.01)
        w1 = w1 - w1_update
        w0 = w0 - w0_update

    return w1, w0
```

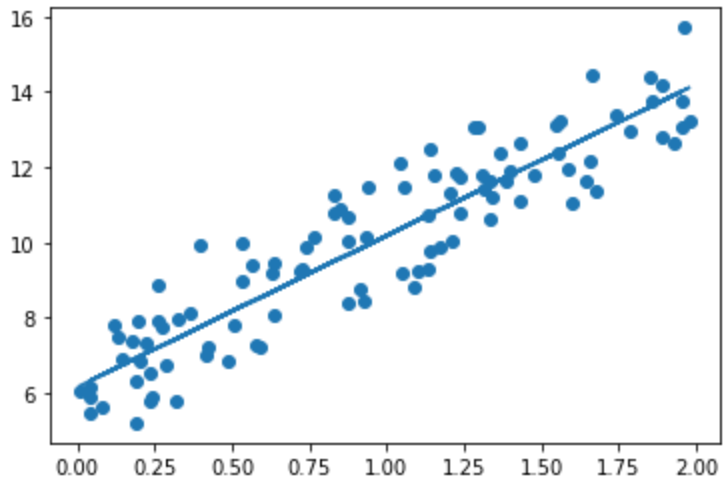
이제 gradient_descent_steps()를 호출해 w1과 w0을 구해 보겠음. 그리고 최종적으로 예측값과 실제값의 RSS 차이를 계산하는 get_cost() 함수를 생성하고 이를 이용해 경사 하강법의 예

측 오류도 계산해 보겠음.

```
def get_cost(y, y_pred):
    N = len(y)
    cost = np.sum(np.square(y - y_pred))/N
    return cost

w1, w0 = gradient_descent_steps(X, y, iters=1000)
print("w1:{0:.3f} w0:{1:.3f}".format(w1[0,0], w0[0,0]))  # w1:4.022 w0:6.162(실제 선형식과 유사하게 도출됨)
y_pred = w1[0,0] * X + w0
print('Gradient Descent Total Cost:{0:.4f}'.format(get_cost(y, y_pred)))  # 0.9935(예측 오류 비용)
```

앞에서 구한 y_pred에 기반해 회귀선을 그려보겠음.

```
plt.scatter(X, y)
plt.plot(X,y_pred)
```



경사하강법을 이용해 회귀선이 잘 만들어짐. 일반적으로 경사 하강법은 모든 학습 데이터에 대해 반복적으로 비용함수 최소화를 위한 값을 업데이트하기 때문에 수행 시간이 매우 오래 걸린다는 단점이 있음.

그때문에 실전에서는 대부분 **확률적 경사 하강법을 이용**함. 확률적 경사 하강법은 전체 입력 데이터로 w가 업데이트되는 값을 계산하는 것이 아닌, 일부 데이터만 이용해 w가 업데이트되는 값을 계산하므로 경사 하강법에 비해 빠른 속도를 보장함. 따라서 대용량의 데이터의 경우 대부분 **확률적 경사 하강법이나 미니 배치 확률적 경사 하강법**을 이용해 최적 비용함수를 도출함.

미니 배치 확률적 경사 하강법을 이용한 최적 비용함수 도출. 앞의 경사 하강법과 크게 다르지 않음, 다만 전체 X, y 데이터에서 랜덤하게 batch_size 만큼 데이터를 추출해 이를 기반으로 w1_
update, w0_update를 계산하는 부분만 차이가 있음.

```
def stochastic_gradient_descent_steps(X, y, batch_size=10, iters=1000):
    w0 = np.zeros((1,1))
    w1 = np.zeros((1,1))
    prev_cost = 100000
    iter_index =0

    for ind in range(iters):
        np.random.seed(ind)
        # 전체 X, y 데이터에서 랜덤하게 batch_size만큼 데이터 추출하여 sample_X, sample_y로 저장
        stochastic_random_index = np.random.permutation(X.shape[0])
        sample_X = X[stochastic_random_index[0:batch_size]]
        sample_y = y[stochastic_random_index[0:batch_size]]
        # 랜덤하게 batch_size만큼 추출된 데이터 기반으로 w1_update, w0_update 계산 후 업데이트
        w1_update, w0_update = get_weight_updates(w1, w0, sample_X, sample_y, learning_rat
e=0.01)
        w1 = w1 - w1_update
        w0 = w0 - w0_update

    return w1, w0
```

```
w1, w0 = stochastic_gradient_descent_steps(X, y, iters=1000)
print("w1:",round(w1[0,0],3),"w0:",round(w0[0,0],3))
y_pred = w1[0,0] * X + w0
print('Stochastic Gradient Descent Total Cost:{0:.4f}'.format(get_cost(y, y_pred)))  # 0.9
937 예측 오류 비용
```

w0과 w1에도 큰 차이 없고, 예측 오류 비용도 비슷해서 예측 성능의 차이가 없음을 알 수 있다

⇒ 큰 데이터를 처리할 경우에는 경사 하강법은 매우 시간이 오래 걸리니 일반적으로 확률적 경사 하강법을 이용함.

지금까지는 피처(독립변수) 1개인 단순 선형 회귀에서의 경사 하강법을 적용했음. 만약 피처가 여러 개인 경우 어떻게 회귀 계수 도출? 선형대수를 이용해 예측값 도출할 수 있음.

4. 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측

선형 모델 중 규제가 적용되지 않은 선형 회귀인 LinearRegression을 이용해 보스턴 주택 가격 예측 회귀를 구현해 보겠음.

LinearRegression 클래스 - Ordinary Least Sqaures

LinearRegression: 예측값과 실제 값의 RSS를 최소화해 OLS 추정 방식으로 구현한 클래스

LinearRegression 클래스는 fit() 메서드로 X, y 배열을 입력 받으면 회귀 계수인 W를 coef_ 속성에 저장함.

입력 파라미터	fit_intercept: 불린 값으로, 디폴트는 True. intercept(절편) 값을 계산할 것인지 말지를 지정함. 만일 False로 지정하면 intercept가 사용되지 않고 0으로 지정됨. normalize: 불린 값으로 디폴트는 False. fit_intercept가 False인 경우에는 이 파라미터가 무시됨. 만약 True이면 회귀를 수행하기 전에 입력 데이터 세트를 정규화함.
속성	coef_: fit() 메서드를 수행했을 때 회귀 계수가 배열 형태로 저장하는 속성. Shape는 (Target 값 개수, 피처 개수) intercept_: intercept 값

OLS 기반의 회귀 계수 계산은 입력 피처의 독립성에 많은 영향을 받음. **피처 간의 상관관계가 매우 높은 경우 분산이 매우 커져서 오류에 매우 민감해짐. 이러한 현상을 다중공선성 문제**라고 함. 일반적으로 상관관계가 높지 않은 피처가 많은 경우 독립적인 중요한 피처만 남기고 제거하거나 규제를 적용함. 또한 매우 많은 피처가 다중 공선성 문제를 가지고 있다면 PCA를 통해 차원 축소를 수행하는 것도 고려해 볼 수 잇음.

회귀 평가 지표

회귀의 평가를 위한 지표는 실제 값과 회귀 예측값의 차이 값을 기반으로 한 지표가 중심이다. 오류의 절댓값의 평균이나 제곱, 또는 제곱한 뒤 다시 루트를 씌운 평균값을 구한다

https://roytravel.tistory.com/57

회귀 성능 평가 지표

Aa 평가 지표	설명	수식
MAE	Mean Absolute Error이며 실제값과 예측값의 차이를 절대값으로 변환해 평균한 것	$MAE = \frac{1}{n}\sum_{i=1}^{n}\lvert Y_i - \hat{Y}_i\rvert$
MSE	Mean Squared Error이며 실제값과 예측값의 차이를 제곱해 평균한 것	$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$

Aa 평가 지표	설명	수식
RMSE	MSE 값은 오류의 제곱을 구하므로 실제 오류 평균보다 더 커지는 특성이 있으므로 MSE에 루트를 씌운 것이 RMSE.	$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}$
R^2	분산 기반으로 예측 성능을 평가한다. 실제값의 분산 대비 예측값의 분산 비율을 지표로 하며, 1에 가까울수록 예측 정확도가 높음	$R^2 = \frac{예측값Variance}{실제값Variance}$

이 밖에 MSE나 RMSE에 로그를 적용한 MSLE(Mean Squared Log Error)와 RMSLE(Root Mean Squared Log Error)도 사용한다.
사이킷런은 RMSE를 제공하지 않아서, MSE에 제곱근을 씌워서 계산하는 함수를 직접 만들어야 RMSE 구할 수 있다.

다음은 각 평가 방법에 대한 사이킷런의 API 및 cross_val_score나 GridSearchCV에서 평가 시 사용되는 scoring 파라미터의 적용값이다.

scoring 파라미터 적용 값

Aa 평가 방법	사이킷런 평가 지표 API	Scoring 함수 적용 값
MAE	metrics.mean_absoulte_error	'neg_mean_absolute_error'
MSE	metrics.mean_squared_error	'neg_mean_squared_error'
R2	metrics.r2_score	'r2'

cross_val_score, GridSearchCV와 같은 scoring 함수에 회귀 평가 지표를 적용할 때 한 가지 유의할 점이 있음. MAE와 MSE 파라미터 값에 ‘neg_’라는 접두사가 붙어 있음, 이는 음수 값을 가진다는 의미인데. MAE는 절댓값의 합이기 때문에 음수가 될 수 없음. 음수값을 반환하는 이유는, 사이킷런의 Scoring 함수가 score값이 클수록 좋은 평가 결과로 자동 평가하기 때문이다. 그런데 실제값과 예측값의 오류 차이를 기반으로 하는 회귀 평가 지표의 경우 값이 커지면 오히려 나쁜 모델이라는 의미이므로 이를 사이킷런의 Scoring 함수에 일반적으로 반영하려면 보정이 필요함.
⇒ 따라서 -1을 원해의 평가 지표 값에 곱해서 음수를 만들어 작은 오류 값이 더 큰 숫자로 인식하도록 하는 것. (사이킷런 평가 지표 API는 정상적으로 양수 값 반환, 하지만 Scoring 함수의 scoring 파라미터 값 neg_는 음수를 의미함)

LinearRegression을 이용해 보스턴 주택 가격 회귀 구현

선형 회귀 모델을 만들어보겠음. 사이킷런에 내장된 보스턴 주택 가격 데이터 이용.

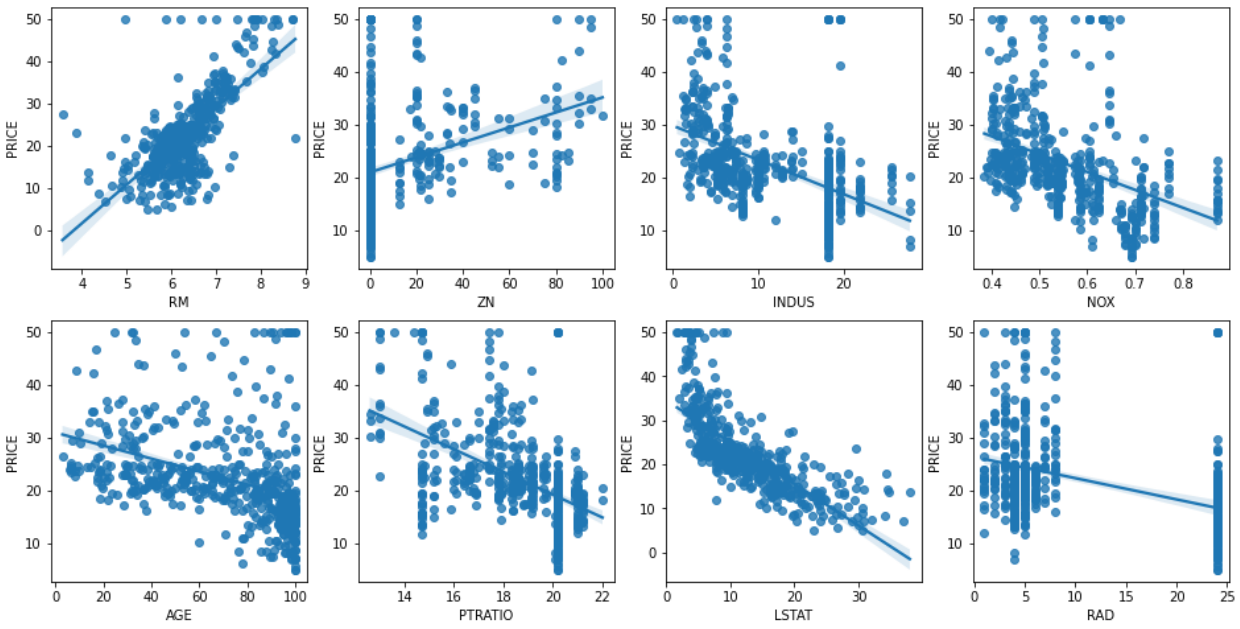
- CRIM: 지역별 범죄 발생률
- ZN: 25,000평방피트를 초과하는 거주 지역의 비율
- NDUS: 비상업 지역 넓이 비율
- CHAS: 찰스강에 대한 더미 변수(강의 경계에 위치한 경우는 1, 아니면 0)
- NOX: 일산화질소 농도
- RM: 거주할 수 있는 방 개수
- AGE: 1940년 이전에 건축된 소유 주택의 비율
- DIS: 5개 주요 고용센터까지의 가중 거리
- RAD: 고속도로 접근 용이도
- TAX: 10,000달러당 재산세율
- PTRATIO: 지역의 교사와 학생 수 비율
- B: 지역의 흑인 거주 비율
- LSTAT: 하위 계층의 비율
- MEDV: 본인 소유의 주택 가격(중앙값)

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from scipy import stats
from sklearn.datasets import load_boston
%matplotlib inline

# boston 데이타셋 로드
boston = load_boston()

# boston 데이타셋 DataFrame 변환
bostonDF = pd.DataFrame(boston.data , columns = boston.feature_names)

# boston dataset의 target array는 주택 가격임. 이를 PRICE 컬럼으로 DataFrame에 추가함.
bostonDF['PRICE'] = boston.target
print('Boston 데이타셋 크기 :',bostonDF.shape)
bostonDF.head()
```

```
bostonDF.info()  ## null 값 없고, 모두 float형
```

- 각 컬럼별로 주택가격에 미치는 영향도를 조사(각 칼럼이 회귀 결과에 미치는 영향 시각화). 총 8개의 칼럼에 대해 값이 증가할 수록 price가 어떻게 변하는지 확인.
  - 시본의 regplot() API는 X, Y축 값의 산점도와 함께 선형 회귀 직선을 그려줌.
  - matplotlib의 subplot()은 여러 개의 그래프를 한번에 표현해줌 (ncols: 열 방향으로 위치할 그래프 개수, nrows: 행 방향으로 위치할 그래프의 개수)

```
# 2개의 행과 4개의 열을 가진 subplots를 이용. axs는 4x2개의 ax를 가짐.
# 2개의 행과 4개의 열 총 8개의 그래프를 행, 열 방향으로 그림.
fig, axs = plt.subplots(figsize=(16,8) , ncols=4 , nrows=2)
lm_features = ['RM','ZN','INDUS','NOX','AGE','PTRATIO','LSTAT','RAD']
for i , feature in enumerate(lm_features):
    row = int(i/4)
    col = i%4
    # 시본의 regplot을 이용해 산점도와 선형 회귀 직선을 함께 표현
    sns.regplot(x=feature , y='PRICE',data=bostonDF , ax=axs[row][col])
```



RM, LSTAT의 PRICE 영향도가 가장 두드러지게 나타남.

- RM(방개수) 양방향 선형 → 방 클수록 가격 증가
- LSTAT(하위 계층의 비율) 음방향 선형 → LSTAT 적을수록 가격 증가

학습과 테스트 데이터 세트로 분리하고 학습/예측/평가 수행

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error , r2_score

y_target = bostonDF['PRICE']
X_data = bostonDF.drop(['PRICE'],axis=1,inplace=False)

X_train , X_test , y_train , y_test = train_test_split(X_data , y_target ,test_size=0.3, r
andom_state=156)

# Linear Regression OLS로 학습/예측/평가 수행.
lr = LinearRegression()
lr.fit(X_train ,y_train )
y_preds = lr.predict(X_test)
mse = mean_squared_error(y_test, y_preds)
rmse = np.sqrt(mse)

print('MSE : {0:.3f} , RMSE : {1:.3F}'.format(mse , rmse))  # MSE : 17.297 , RMSE : 4.159
print('Variance score : {0:.3f}'.format(r2_score(y_test, y_preds)))  # Variance score : 0.
757
```

LinearRegression으로 생성한 주택가격 모델의 intercept(절편)과 coefficients(회귀 계수)값을 보겠음.

```
print('절편 값:',lr.intercept_)  # 40.995595172164755
print('회귀 계수값:', np.round(lr.coef_, 1))  # [ -0.1   0.1   0.    3.  -19.8   3.4   0.
 -1.7   0.4  -0.   -0.9   0.
  -0.6]
```

coef_ 속성은 회귀 계수 값만 가지고 있으므로 이를 피처별 회귀 계수 값으로 다시 매핑하고, 높은 값순으로 출력하겠음.

```
# 회귀 계수를 큰 값 순으로 정렬하기 위해 Series로 생성. index 컬럼명에 유의
coeff = pd.Series(data=np.round(lr.coef_, 1), index=X_data.columns )
coeff.sort_values(ascending=False)
'''
RM          3.4
CHAS        3.0
RAD         0.4
ZN          0.1
INDUS       0.0
AGE         0.0
TAX        -0.0
```

```
B           0.0
CRIM       -0.1
LSTAT      -0.6
PTRATIO    -0.9
DIS        -1.7
NOX       -19.8
dtype: float64
'''
```

RM이 양의 값으로 회귀 계수 가장 크며, NOX 피처의 회귀 계수 -값이 너무 커보임. 최적화 수행하면서 피처 coefficient의 변화도 같이 살펴보겠음.

이번에는 5개의 폴드 세트에서 cross_val_score()를 이용해 교차 검증으로 MSE와 RMSE를 측정해 보겠음. (사이킷런은 MSE 수치 결과를 RMSE로 변환해야 함)

```
from sklearn.model_selection import cross_val_score

y_target = bostonDF['PRICE']
X_data = bostonDF.drop(['PRICE'],axis=1,inplace=False)
lr = LinearRegression()

# cross_val_score( )로 5 Fold 셋으로 MSE 를 구한 뒤 이를 기반으로 다시  RMSE 구함.
neg_mse_scores = cross_val_score(lr, X_data, y_target, scoring="neg_mean_squared_error", c
v = 5)
rmse_scores  = np.sqrt(-1 * neg_mse_scores)
avg_rmse = np.mean(rmse_scores)

# cross_val_score(scoring="neg_mean_squared_error")로 반환된 값은 모두 음수
print(' 5 folds 의 개별 Negative MSE scores: ', np.round(neg_mse_scores, 2))  # [-12.46 -2
6.05 -33.07 -80.76 -33.31]
print(' 5 folds 의 개별 RMSE scores : ', np.round(rmse_scores, 2))  # [3.53 5.1  5.75 8.99
 5.77]
print(' 5 folds 의 평균 RMSE : {0:.3f} '.format(avg_rmse))  # 5.829
```

5개의 폴드 세트에 대해 교차 검증을 수행한 결과, 평균 RMSE는 약 5.836이 나옴.

5. 다항 회귀와 과(대)적합/과소적합 이해

다항 회귀 이해

지금까지는 독립변수와 종속변수의 관계가 일차 방정식으로 표현된 회귀였음. 하지만 모든 관계를직선으로 표현할 수 없음. 회귀 독립변수가 단항식이 아닌 2, 3차 방정식과 같은 다항식으로 표현되는 것을 다항(Polynomial)회귀라고 함.

다항 회귀를 비선형 회귀로 혼동하기 쉽지만, 다항 회귀는 선형 회귀이다. 회귀에서 선형/비선형 회귀를 나누는 기준은 회귀 계수가 선형/비선형인지에 따른 것이지 독립변수의 선형/비선형 여부와는 무관함.

사이킷런은 다항 회귀를 위한 클래스를 명시적으로 제시하지 않음. 다항 회귀 역시 선형 회귀이기 때문에 비선형 함수를 선형 모델에 적용시키는 방법을 사용해 구현함. 이를 위해 사이킷런은 PolynomialFeatures 클래스를 통해 피처를 Polynomial(다항식) 피처로 변환함. 이 클래스는 degree 파라미터를 통해 입력 받음 단항식 피처를 degree에 해당하는 다양한 피처로 변환함. 다른 전처리 변환 클래스와 마찬가지로 Polynomial Features 클래스는 fti(), transform() 메서들을 통해 이 같은 변환 작업 수행함.

아래 예제는 단항값 [x1, x2]를 다항값으로 변환하는 예제이다.

PolynomialFeatures 클래스로 다항식 변환

```
from sklearn.preprocessing import PolynomialFeatures
import numpy as np

# 다항식으로 변환한 단항식 생성, [[0,1],[2,3]]의 2X2 행렬 생성
X = np.arange(4).reshape(2,2)
print('일차 단항식 계수 feature:\n',X )

# degree = 2 인 2차 다항식으로 변환하기 위해 PolynomialFeatures를 이용하여 변환
poly = PolynomialFeatures(degree=2)
poly.fit(X)
poly_ftr = poly.transform(X)
print('변환된 2차 다항식 계수 feature:\n', poly_ftr)
```

이렇게 변환된 Polynomial 피처에 선형 회귀를 적용해 다항 회귀를 구현함. 이번에는 3차 다항 계수를 이용해 3차 다항 회귀 함수식을 유도해보겠음.

이를 위해 3차 다항 회귀 함수를 임의로 설정하고 이의 회귀 계수를 예측할 것. 3차 다항식 결정값을 구하는 함수 polynomial_func(X) 생성. 즉 회귀식은 결정값 $y = 1 + 2x_1 + 3x_1^2 + 4x_2^3$

```
def polynomial_func(X):
    y = 1 + 2*X[:,0] + 3*X[:,0]**2 + 4*X[:,1]**3
    print(X[:, 0])
    print(X[:, 1])
    return y

X = np.arange(0,4).reshape(2,2)

print('일차 단항식 계수 feature: \n' ,X)
```

```
y = polynomial_func(X)
print('삼차 다항식 결정값: \n', y)
```

3차 다항식 계수의 피처값과 3차 다항식 결정값으로 학습

```
# 3 차 다항식 변환
poly_ftr = PolynomialFeatures(degree=3).fit_transform(X)
print('3차 다항식 계수 feature: \n',poly_ftr)

# Linear Regression에 3차 다항식 계수 feature와 3차 다항식 결정값으로 학습 후 회귀 계수 확인
model = LinearRegression()
model.fit(poly_ftr,y)
print('Polynomial 회귀 계수\n' , np.round(model.coef_, 2))
print('Polynomial 회귀 Shape :', model.coef_.shape)
```

일차 다항식 계수 피처는 2개였지만, 3차 다항식 polynomial 변환 이후에는ㄷ ㅏ항식 계수 피처가 10개로 늘어남. 이처럼 사이킷런은 PolynomialFeatures로 피처를 변환한 후에 LinearRegression 클래스로 다항 회귀를 구현함.

이전 예제처럼 피처 변환과 선형 회귀 적용을 각각 별도로 하는 것보다는 사이킷런의 pipeline 객체를 이용해 한번에 다항 회귀를 구현하는 것이 코드를 더 명료하게 작성하는 방법임.

사이킷런 파이프라인(Pipeline)을 이용하여 3차 다항회귀 학습
사이킷런의 Pipeline 객체는 Feature 엔지니어링 변환과 모델 학습/예측을 순차적으로 결합해 줌

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
import numpy as np

def polynomial_func(X):
    y = 1 + 2*X[:,0] + 3*X[:,0]**2 + 4*X[:,1]**3
    return y

# Pipeline 객체로 Streamline 하게 Polynomial Feature변환과 Linear Regression을 연결
model = Pipeline([('poly', PolynomialFeatures(degree=3)),
                  ('linear', LinearRegression())])
X = np.arange(4).reshape(2,2)
y = polynomial_func(X)

model = model.fit(X, y)
print('Polynomial 회귀 계수\n', np.round(model.named_steps['linear'].coef_, 2))
```

다항 회귀를 이용한 과소적합 및 과적합 이해

다항 회귀는 피처의 직선적 관계가 아닌 복잡한 다항 관계를 모델링할 수 있음. 다항식의 차수가 높아질수록 매우 복잡한 피처 간의 관계까지 모델링이 가능함. 하지만 다항 회귀의 차수가 높아질수록 과적합의 문제가 크게 발생함.

좋은 예측 모델은 학습 데이터의 패턴을 잘 반영하면서도 복잡하지 않은 균형 잡힌 모델을 의미함.

편향-분산 트레이드오프(Bias-Variance Trade off)

편향-분산 트레이드오프는 머신러닝이 극복해야 할 가장 중요한 이슈 중 하나.

매우 단순화된 모델 → 지나치게 한 방향성으로 치우친 경향이 있음 ⇒ 고편향성 가졌다
매우 복잡한 모델 → 지나치게 높은 변동성 가짐 ⇒ 고분산성 가졌다

일반적으로 편향과 분산은 한 쪽이 높으면 한 쪽이 낮아지는 경향이 있음. 즉, 편향이 높으면 분산이 낮아지고(과소적합) 반대로 분산이 높으면 편향이 낮아짐(과적합).

⇒ 편향과 분산이 서로 트레이드오프를 이루면서 오류 Cost값이 최대로 낮아지는 모델을 구축하는 것이 가장 효율적인 머신러닝 예측 모델을 만드는 방법이다.

6. 규제 선형 모델 - 릿지, 라쏘, 엘라스틱넷

규제 선형 모델의 개요

회귀 모델은 적절히 데이터에 적합하면서도 회귀 계수가 기하급수적으로 커지는 것을 제어할 수 있어야 함.

이전까지 선형 모델의 비용 함수는 RSS를 최소화하는, 즉 실제 값과 예측값의 차이를 최소화하는 것만 고려했음. 그러다보니 학습 데이터에 지나치게 맞추게 되어, 회귀 계수가 쉽게 켜졌고 → 변동성 심해짐 → 예측 성능 저하. ⇒ 이를 반영해서 비용 함수는 데이터의 잔차 오류 값을 최소로 하는 RSS 최소화 방법과 과적합 방지하기 위해 회귀 계수 값이 커지지 않도록 하는 방법이 서로 균형을 이뤄야 함.

알파를 0에서부터 지속적으로 값을 증가시키면 회귀 계수 값의 크기를 감소시킬 수 있음. 이처럼 비용 함수에 알파 값으로 패널티를 부여해 회귀 계수 값의 크기를 감소시켜 과적합을 개선하는 방식을 규제라고 부름

규제는 L1, L2 방식으로 구분됨. L2 규제를 릿지, L1 규제를 적용한 것이 라쏘 회귀이다.

릿지 회귀

alpha: 릿지 회귀의 규제 계수에 해당함. 앞의 보스턴 주택 가격을 릿지 클래스를 이용해 다시 예측하겠음.

```
# 앞의 LinearRegression예제에서 분할한 feature 데이터 셋인 X_data과 Target 데이터 셋인 Y_target 데이터셋을 그대로 이용
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

# boston 데이타셋 로드
boston = load_boston()

# boston 데이타셋 DataFrame 변환
bostonDF = pd.DataFrame(boston.data , columns = boston.feature_names)

# boston dataset의 target array는 주택 가격임. 이를 PRICE 컬럼으로 DataFrame에 추가함.
bostonDF['PRICE'] = boston.target
print('Boston 데이타셋 크기 :',bostonDF.shape)

y_target = bostonDF['PRICE']
X_data = bostonDF.drop(['PRICE'],axis=1,inplace=False)


ridge = Ridge(alpha = 10)
neg_mse_scores = cross_val_score(ridge, X_data, y_target, scoring="neg_mean_squared_error", cv = 5)
rmse_scores  = np.sqrt(-1 * neg_mse_scores)
avg_rmse = np.mean(rmse_scores)
print(' 5 folds 의 개별 Negative MSE scores: ', np.round(neg_mse_scores, 3))
print(' 5 folds 의 개별 RMSE scores : ', np.round(rmse_scores,3))
print(' 5 folds 의 평균 RMSE : {0:.3f} '.format(avg_rmse))  # 5.518
```

→ 앞의 규제가 없는 LinearRegression rmse의 평균보다 뛰어난 예측 성능 보여줌.

릿지의 알파값 변화시키면서 rmse와 회귀 계수 값 살펴보겠음. 릿지 회귀는 알파 값이 커질수록, 회귀 계수 값을 작게 만듦.

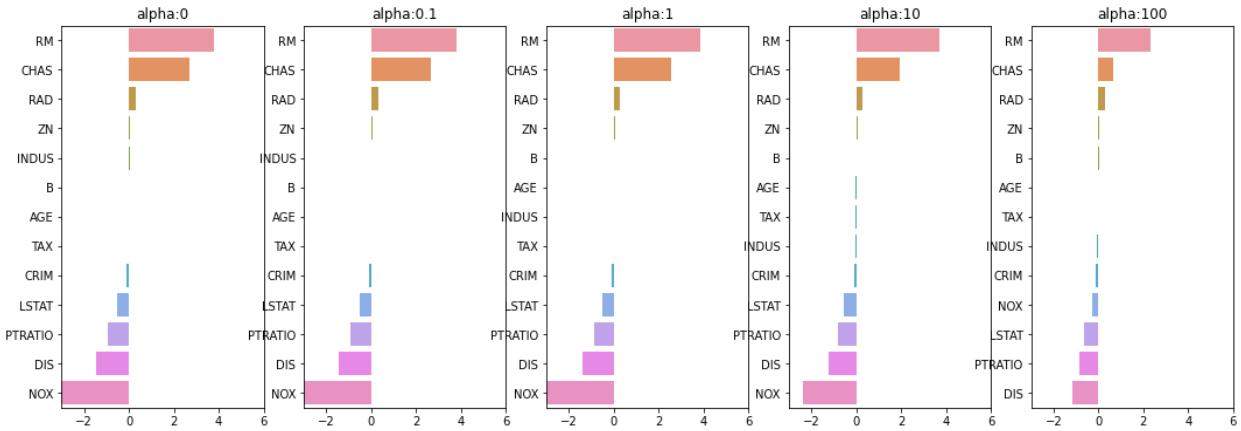
```
# Ridge에 사용될 alpha 파라미터의 값들을 정의
alphas = [0 , 0.1 , 1 , 10 , 100]

# alphas list 값을 iteration하면서 alpha에 따른 평균 rmse 구함.
for alpha in alphas :
    ridge = Ridge(alpha = alpha)

    #cross_val_score를 이용하여 5 fold의 평균 RMSE 계산
    neg_mse_scores = cross_val_score(ridge, X_data, y_target, scoring="neg_mean_squared_error", cv = 5)
```

```
    avg_rmse = np.mean(np.sqrt(-1 * neg_mse_scores))
    print('alpha {0} 일 때 5 folds 의 평균 RMSE : {1:.3f} '.format(alpha,avg_rmse))
```

→ 알파가 100일때 평균 rmse가 가장 좋음.

각 **alpha에 따른 회귀 계수 값을 시각화. 각 alpha값 별로 plt.subplots로 맷플롯립 축 생성**

```
# 각 alpha에 따른 회귀 계수 값을 시각화하기 위해 5개의 열로 된 맷플롯립 축 생성
fig , axs = plt.subplots(figsize=(18,6) , nrows=1 , ncols=5)
# 각 alpha에 따른 회귀 계수 값을 데이터로 저장하기 위한 DataFrame 생성
coeff_df = pd.DataFrame()

# alphas 리스트 값을 차례로 입력해 회귀 계수 값 시각화 및 데이터 저장. pos는 axis의 위치 지정
for pos , alpha in enumerate(alphas) :
    ridge = Ridge(alpha = alpha)
    ridge.fit(X_data , y_target)
    # alpha에 따른 피처별 회귀 계수를 Series로 변환하고 이를 DataFrame의 컬럼으로 추가.
    coeff = pd.Series(data=ridge.coef_ , index=X_data.columns )
    colname='alpha:'+str(alpha)
    coeff_df[colname] = coeff
    # 막대 그래프로 각 alpha 값에서의 회귀 계수를 시각화. 회귀 계수값이 높은 순으로 표현
    coeff = coeff.sort_values(ascending=False)
    axs[pos].set_title(colname)
    axs[pos].set_xlim(-3,6)
    sns.barplot(x=coeff.values , y=coeff.index, ax=axs[pos])

# for 문 바깥에서 맷플롯립의 show 호출 및 alpha에 따른 피처별 회귀 계수를 DataFrame으로 표시
plt.show()
```



→ 각 알파값을 계속 증가시킬수록 회귀 계수 값은 지속적으로 작아짐.

alpha 값에 따른 컬럼별 회귀계수 출력

```
ridge_alphas = [0 , 0.1 , 1 , 10 , 100]
sort_column = 'alpha:'+str(ridge_alphas[0])
coeff_df.sort_values(by=sort_column, ascending=False)
```

⇒ 알파 값이 증가하면서 회귀 계수가 지속적으로 작아지고 있음을 알 수 있다. 하지만 릿지 회귀의 경우에는 회귀 계수를 0으로 만들지는 않음.

라쏘 회귀

L2 규제가 회귀 계수의 크기를 감소시키는데 반해, L1 규제는 불필요한 회귀 계수를 급격하게 감소시켜 0으로 만들고 제거함. 이러한 측면에서 L1 규제는 적절한 피처만 회귀에 포함시키는 피처 선택의 특성을 가지고 있음.

```
from sklearn.linear_model import Lasso, ElasticNet

# alpha값에 따른 회귀 모델의 폴드 평균 RMSE를 출력하고 회귀 계수값들을 DataFrame으로 반환
def get_linear_reg_eval(model_name, params=None, X_data_n=None, y_target_n=None,
                        verbose=True, return_coeff=True):
    coeff_df = pd.DataFrame()
    if verbose : print('####### ', model_name , '#######')
    for param in params:
        if model_name =='Ridge': model = Ridge(alpha=param)
        elif model_name =='Lasso': model = Lasso(alpha=param)
        elif model_name =='ElasticNet': model = ElasticNet(alpha=param, l1_ratio=0.7)
        neg_mse_scores = cross_val_score(model, X_data_n,
                                         y_target_n, scoring="neg_mean_squared_error",
cv = 5)
        avg_rmse = np.mean(np.sqrt(-1 * neg_mse_scores))
        print('alpha {0}일 때 5 폴드 세트의 평균 RMSE: {1:.3f} '.format(param, avg_rmse))
        # cross_val_score는 evaluation metric만 반환하므로 모델을 다시 학습하여 회귀 계수 추출

        model.fit(X_data_n , y_target_n)
        if return_coeff:
            # alpha에 따른 피처별 회귀 계수를 Series로 변환하고 이를 DataFrame의 컬럼으로 추가.
            coeff = pd.Series(data=model.coef_ , index=X_data_n.columns )
            colname='alpha:'+str(param)
            coeff_df[colname] = coeff

    return coeff_df
# end of get_linear_regre_eval
```

```
# 라쏘에 사용될 alpha 파라미터의 값들을 정의하고 get_linear_reg_eval() 함수 호출
lasso_alphas = [ 0.07, 0.1, 0.5, 1, 3]
```

```
coeff_lasso_df =get_linear_reg_eval('Lasso', params=lasso_alphas, X_data_n=X_data, y_targe
t_n=y_target)
```

```
# 반환된 coeff_lasso_df를 첫번째 컬럼순으로 내림차순 정렬하여 회귀계수 DataFrame출력
sort_column = 'alpha:'+str(lasso_alphas[0])
coeff_lasso_df.sort_values(by=sort_column, ascending=False)
```

→ 알파의 크기가 증가함에 따라 일부 피처의 회귀 계수는 아예 0으로 바뀌고 있음. 회귀 계수가 0인 피처는 회귀 식에서 제외되면서 피처 선택의 효과를 얻을 수 있음.

엘라스틱넷 회귀

L1 규제와 L2 규제를 결합한 회귀이다. 라소 회귀가 서로 상관관계가 높은 피처들의 경우에 이들 중에서 중요 피처만을 셀력션하고 다른 피처들은 모두 회귀 계수를 0으로 만드는 성향이 강함. 엘라스틱넷 회귀는 이를 완화하기 위해 L2 규제를 라쏘 회귀에 추가한 것. 단점은 상대적으로 시간이 오래 걸린다는 점이다.

```
# 엘라스틱넷에 사용될 alpha 파라미터의 값들을 정의하고 get_linear_reg_eval() 함수 호출
# l1_ratio는 0.7로 고정
elastic_alphas = [ 0.07, 0.1, 0.5, 1, 3]
coeff_elastic_df =get_linear_reg_eval('ElasticNet', params=elastic_alphas,
                                      X_data_n=X_data, y_target_n=y_target)
```

선형 회귀 모델을 위한 데이터 변환

선형 회귀 모델은 피처값과 타깃값의 분포가 정규 분포인 형태를 매우 선호함. 특히 타깃값의 경우 왜곡된 형태의 분포도일 경우 예측 성능에 부정적인 영향을 미칠 가능성이 높음.

따라서 선형 회귀 모델을 적용하기 위해서는 먼저 데이터에 대한 스케일링/정규화 작업을 수행하는 것이 일반적임. 하지만 이런 작업을 선행한다고 무조건 예측 성능이 향상되는건 아님. 일반적으로 중요 피처들이나 타깃값의 분포도가 심하게 왜곡됐을 경우에 이러한 변환 작업을 수행함.

1. StandardScaler → 평균 0, 분산 1인 표준 정규 분포 가진 데이터세트로 변환 / MinMaxScaler → c최솟값 0 최댓값 1인 값으로 정규화 수행
2. 스케일링/정규화 수행한 데이터 세트에 다시 다항 특성을 적용하여 변환하는 방법.

3. 원래 값에 log 함수를 적용하면 보다 정규 분포에 가까운 형태로 값이 분포됨. 이러한 변환을 로그 변환이라고 부름.

타깃값은 일반적으로 로그 변환을 적용함. 결정 값을 정규 분포나 다른 정규값으로 변환하면 다시 원본 타깃값으로 원복하기 어려울 수 있기 때문.

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, PolynomialFeatures

# method는 표준 정규 분포 변환(Standard), 최대값/최소값 정규화(MinMax), 로그변환(Log) 결정
# p_degree는 다항식 특성을 추가할 때 적용. p_degree는 2이상 부여하지 않음.
def get_scaled_data(method='None', p_degree=None, input_data=None):
    if method == 'Standard':
        scaled_data = StandardScaler().fit_transform(input_data)
    elif method == 'MinMax':
        scaled_data = MinMaxScaler().fit_transform(input_data)
    elif method == 'Log':
        scaled_data = np.log1p(input_data)
    else:
        scaled_data = input_data

    if p_degree != None:
        scaled_data = PolynomialFeatures(degree=p_degree,
                                         include_bias=False).fit_transform(scaled_data)

    return scaled_data
```

```
# Ridge의 alpha값을 다르게 적용하고 다양한 데이터 변환방법에 따른 RMSE 추출.
alphas = [0.1, 1, 10, 100]
#변환 방법은 모두 6개, 원본 그대로, 표준정규분포, 표준정규분포+다항식 특성
# 최대/최소 정규화, 최대/최소 정규화+다항식 특성, 로그변환
scale_methods=[(None, None), ('Standard', None), ('Standard', 2),
               ('MinMax', None), ('MinMax', 2), ('Log', None)]
for scale_method in scale_methods:
    X_data_scaled = get_scaled_data(method=scale_method[0], p_degree=scale_method[1],
                                    input_data=X_data)
    print(X_data_scaled.shape, X_data.shape)
    print('\n## 변환 유형:{0}, Polynomial Degree:{1}'.format(scale_method[0], scale_method
[1]))
    get_linear_reg_eval('Ridge', params=alphas, X_data_n=X_data_scaled,
                        y_target_n=y_target, verbose=False, return_coeff=False)
```

7. 로지스틱 회귀

선형 회귀 방식을 분류에 적용한 알고리즘. 로지스틱 회귀가 선형 회귀와 다른 점은 학습을 통해 선형 함수의 회귀 최적선을 찾는 것이 아니라 시그모이드 함수 최적선을 찾고 이 시그모이드 함수의 반환 값을 확률로 간주해 확률에 따라 분류를 결정함.

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression

cancer = load_breast_cancer()
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# StandardScaler( )로 평균이 0, 분산 1로 데이터 분포도 변환
scaler = StandardScaler()
data_scaled = scaler.fit_transform(cancer.data)

X_train , X_test, y_train , y_test = train_test_split(data_scaled, cancer.target, test_siz
e=0.3, random_state=0)
```

```
from sklearn.metrics import accuracy_score, roc_auc_score

# 로지스틱 회귀를 이용하여 학습 및 예측 수행.
lr_clf = LogisticRegression()
lr_clf.fit(X_train, y_train)
lr_preds = lr_clf.predict(X_test)

# accuracy와 roc_auc 측정
print('accuracy: {:0.3f}'.format(accuracy_score(y_test, lr_preds)))
print('roc_auc: {:0.3f}'.format(roc_auc_score(y_test , lr_preds)))
```

```
from sklearn.model_selection import GridSearchCV

params={'penalty':['l2', 'l1'],
        'C':[0.01, 0.1, 1, 1, 5, 10]}

grid_clf = GridSearchCV(lr_clf, param_grid=params, scoring='accuracy', cv=3 )
grid_clf.fit(data_scaled, cancer.target)
print('최적 하이퍼 파라미터:{0}, 최적 평균 정확도:{1:.3f}'.format(grid_clf.best_params_,
                                                    grid_clf.best_score_))
```

로지스틱 회귀는 가볍고 빠르지만, 이진 분류 예측 성능도 뛰어남. 이 때문에 로지스틱 회귀를 이진 분류의 기본 모델로 사용하는 경우가 많음. 또한 로지스틱 회귀는 희소한 데이터 세트 분류에도 뛰어난 성능을 보여서 텍스트 분류에도 자주 사용됨.

8. 회귀 트리

트리 기반의 회귀는 회귀 트리를 이요하는 것이다. 즉, 회귀를 위한 트리를 생성하고 이를 기반으로 회귀 예측을 하는 것. 분류트리와 크게 다르지 않는데, 다만 리프 노드에서 예측 결정 값을 만드는 과정에서 차이가 있다. 회귀 트리는 리프 노드에 속한 데이터 값의 평균값을 구해 회귀 예측값을 계산함.

선형 회귀는 직선으로 예측 회귀선을 표현하는데 반해, 회귀 트리의 경우 분할되는 데이터 지점에 따라 브랜치를 만들면서 계단 형태로 회귀선을 만듦.