5. 회귀
(Regression)

한승훈

목차

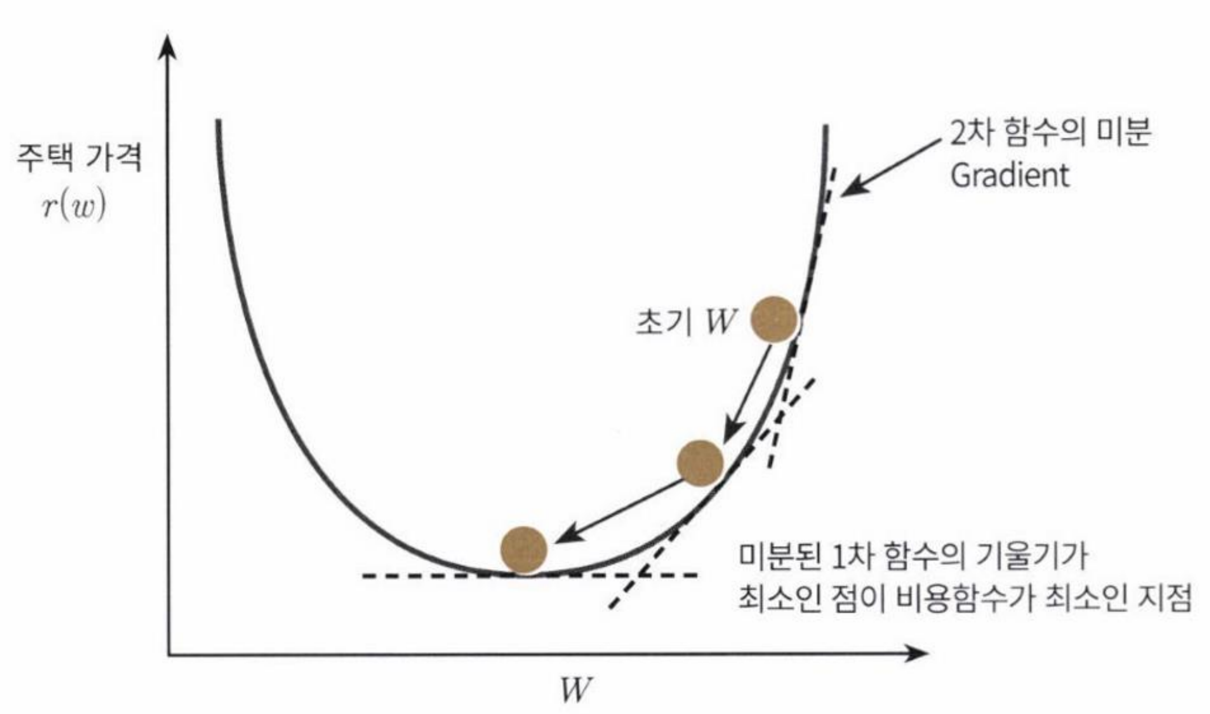
5.3 경사 하강법
(Gradient Descent)

5.3 경사 하강법 (Gradient Descent)

• 경사 하강법이란?

·Cost function

$$R(w) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - (w_0 + w_1 * x_i)\right)^2$$



목적 : Cost Function 최소화
(= 잔차가 최소화)

방법 : –Normal equation
–Gradient Descent

5.3 경사 하강법 (Gradient Descent)

• 경사 하강법의 프로세스

- w_1, w_0 를 임의의 값으로 설정하고 첫 비용함수의 값을 계산

- $w_1 = w_1 - \eta \frac{2}{N} \sum_{i=1}^{N} (y_i - (w_0 + w_1 x_i)) x_i$ * η(eta) = learning rate

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

  로 weight 업데이트(simultaneously)

- 비용함수의 값이 감소했으면 다시 반복, 더 이상 비용함수의 값이 감소하지 않으면 그때의 w_1, w_0를 구하고 stop

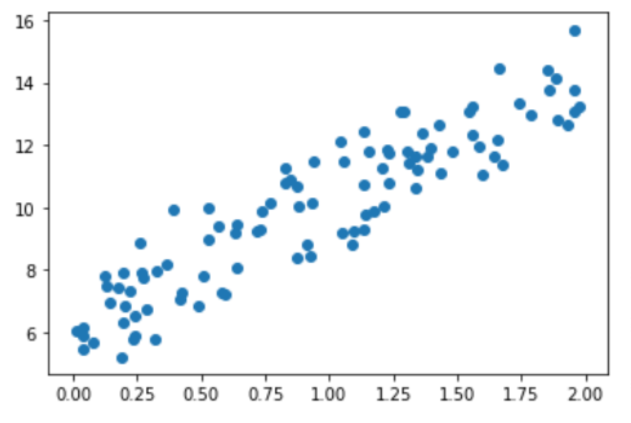
5.3 경사 하강법 (Gradient Descent)

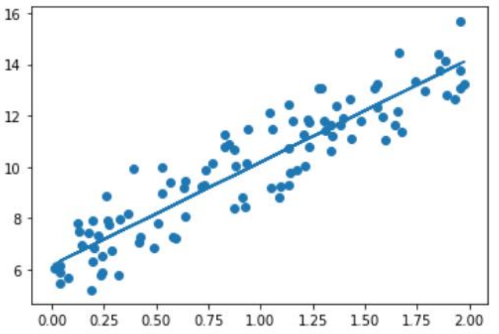
- 경사 하강법 구현

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

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

# X, y 데이터 세트 산점도로 시각화
plt.scatter(X, y)
```






```python
# 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
```

```python
# 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
```

```python
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]))
y_pred = w1[0,0] * X + w0
print('Gradient Descent Total Cost:{0:.4f}'.format(get_cost(y, y_pred)))
```

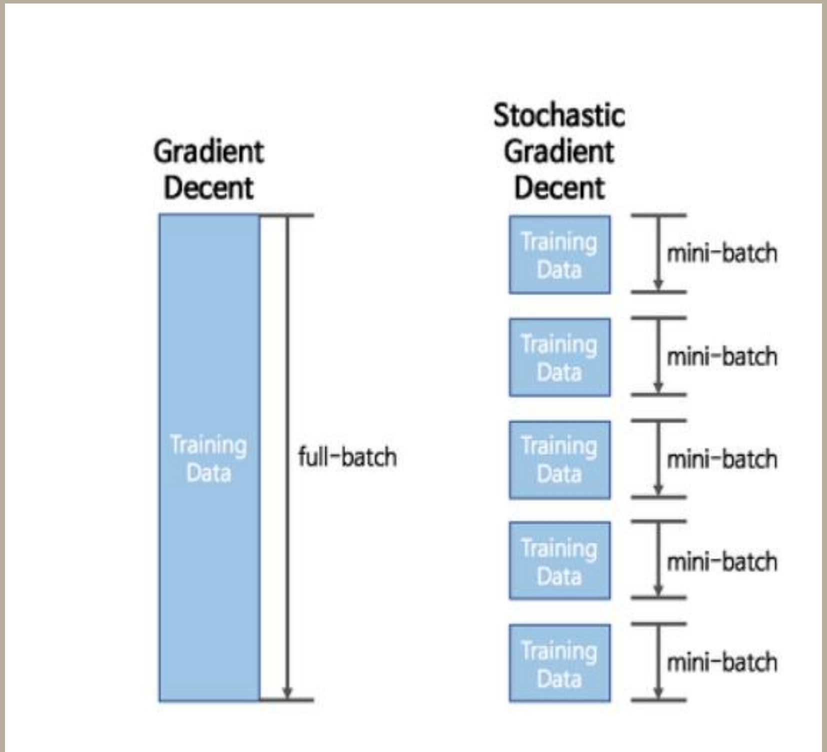
```
w1:4.022 w0:6.162
Gradient Descent Total Cost:0.9935
```

5.3 경사 하강법 (Gradient Descent)

- 경사 하강법의 문제점
  - 모든 학습 데이터에 대해 반복적으로 비용함수 최소화를 위한 값을 업데이트 함 -> 수행 시간이 매우 오래 걸림
  - Normal equation에 비해 정해야 할 parameter 가 많음(learning rate, iteration, etc.)
  - 굴곡이 많은 함수의 경우는? -> 시작점에 따라 다른 최소값을 찾음

5.3 경사 하강법 (Gradient Descent)

- (mini-batch) Stochastic Gradient descent
  - 수행 시간에 대한 단점의 보완을 위한 방법



- 전체 입력 데이터로 w가 업데이트 되는 값을 계산하는 것이 아닌 일부 데이터만 이용해 w가 업데이트 되는 값을 계산하므로 경사하강법에 비해 빠른 속도 보장

5.3 경사 하강법 (Gradient Descent)

- (mini-batch) Stochastic Gradient descent

```
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_rate=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)))
```

```
w1: 4.028 w0: 6.156
Stochastic Gradient Descent Total Cost:0.9937
```

(Full-batch) Gradient Descent와 비교해 큰 차이x

>>시간 단축을 위해 일반적으로 확률적 경사 하강법 이용

5.3 경사 하강법 (Gradient Descent)

- 피처(독립 변수)가 여러 개인 경우

〉〉 피처가 1개인 경우를 확장해 도출 가능

피처 1개 예측 회귀식 $\hat{y} = w_0 + w_1 x$

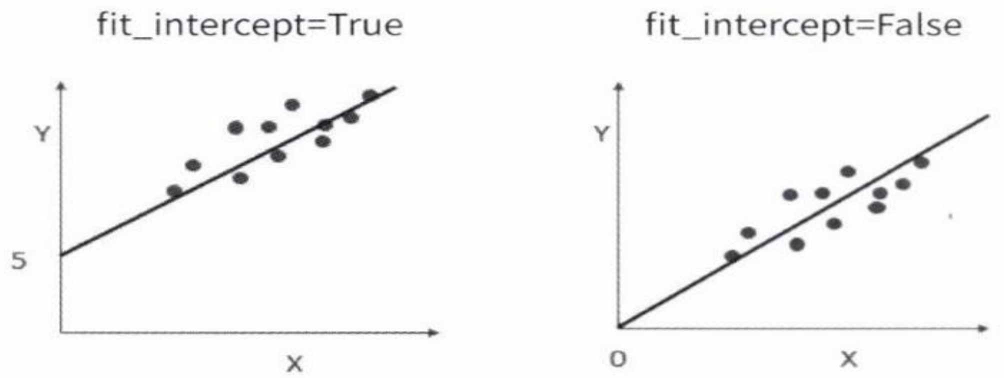
피처 m개 예측 회귀식 $\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$

$$\overset{\hat{Y}}{\begin{bmatrix} y_{11} \\ y_{21} \\ \cdots \\ y_{n1} \end{bmatrix}} = \overset{X_{mat}}{\begin{array}{cccc} \text{Feature 1} & \text{Feature 2} & \cdots & \text{Feature M} \\ \multicolumn{4}{c}{\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}} \end{array}} \underset{\text{내적}}{*} \begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix}^T + w_0$$

5.4 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측

5.4 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측

```
class sklearn.linear_model.LinearRegression(fit_intercept=True, normalize=False, copy_X=True,
n_jobs=1)
```

입력 파라미터	**fit_intercept**: 불린 값으로, 디폴트는 True입니다. Intercept(절편) 값을 계산할 것인지 말지를 지정합니다. 만일 False로 지정하면 intercept가 사용되지 않고 0으로 지정됩니다.  **normalize**: 불린 값으로 디폴트는 False입니다. fit_intercept가 False인 경우에는 이 파라미터가 무시됩니다. 만일 True이면 회귀를 수행하기 전에 입력 데이터 세트를 정규화합니다.
속성	**coef_**: fit() 메서드를 수행했을 때 회귀 계수가 배열 형태로 저장하는 속성. Shape는 (Target 값 개수, 피처 개수). **intercept_**: intercept 값

5.4 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측

• 회귀 평가 지수

· 회귀의 평가를 위한 지수는 실제 값과 예측 값의 차이를 기반으로 한 지표가 중심
· 그냥 실제값과 예측값의 차이를 더하면 상쇄 〉〉 절댓값, 제곱합의 평균 사용

평가 지표	설명	수식
MAE	Mean Absolute Error(MAE)이며 실제 값과 예측값의 차이를 절댓값으로 변환해 평균한 것입니다.	$MAE = \frac{1}{n}\sum_{i=1}^{n}\lvert Yi - \hat{Y}i \rvert$
MSE	Mean Squared Error(MSE)이며 실제 값과 예측값의 차이를 제곱해 평균한 것입니다.	$MSE = \frac{1}{n}\sum_{i=1}^{n}(Yi - \hat{Y}i)$
RMSE	MSE 값은 오류의 제곱을 구하므로 실제 오류 평균보다 더 커지는 특성이 있으므로 MSE에 루트를 씌운 것이 RMSE(Root Mean Squared Error)입니다.	$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Yi - \hat{Y}i)^2}$
R^2	분산 기반으로 예측 성능을 평가합니다. 실제 값의 분산 대비 예측값의 분산 비율을 지표로 하며, 1에 가까울수록 예측 정확도가 높습니다.	$R^2 = \frac{\text{예측값 } Variance}{\text{실제값 } Variance}$

· MSE 등은 값이 낮을수록 좋은 지표
· 사이킷런은 “higher is bette”. 즉, 값이 클수록 좋은 지표로 인식
-〉 음수를 붙여줘 크기를 역전시키는 과정이 필요

5.4 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측

- LinearRegression을 이용해 보스턴 주택 가격 예측

Boston House Price Dataset

-1978년에 발표된 데이터로, 미국 인구통계 조사 결과 미국 보스턴 지역의 주택 가격의 영향 요소들을 정리함

-머신러닝 등 데이터 분석을 처음 배울 때, 가장 대표적으로 사용하는 Example Dataset

- features
  - CRIM: 지역별 범죄 발생률
  - ZN: 25,000평방피트를 초과하는 거주 지역의 비율
  - INDUS: 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)

5.4 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측

- LinearRegression을 이용해 보스턴 주택 가격 예측

```python
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()
```

```
Boston 데이타셋 크기 : (506, 14)
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

```python
bostonDF.info()
```

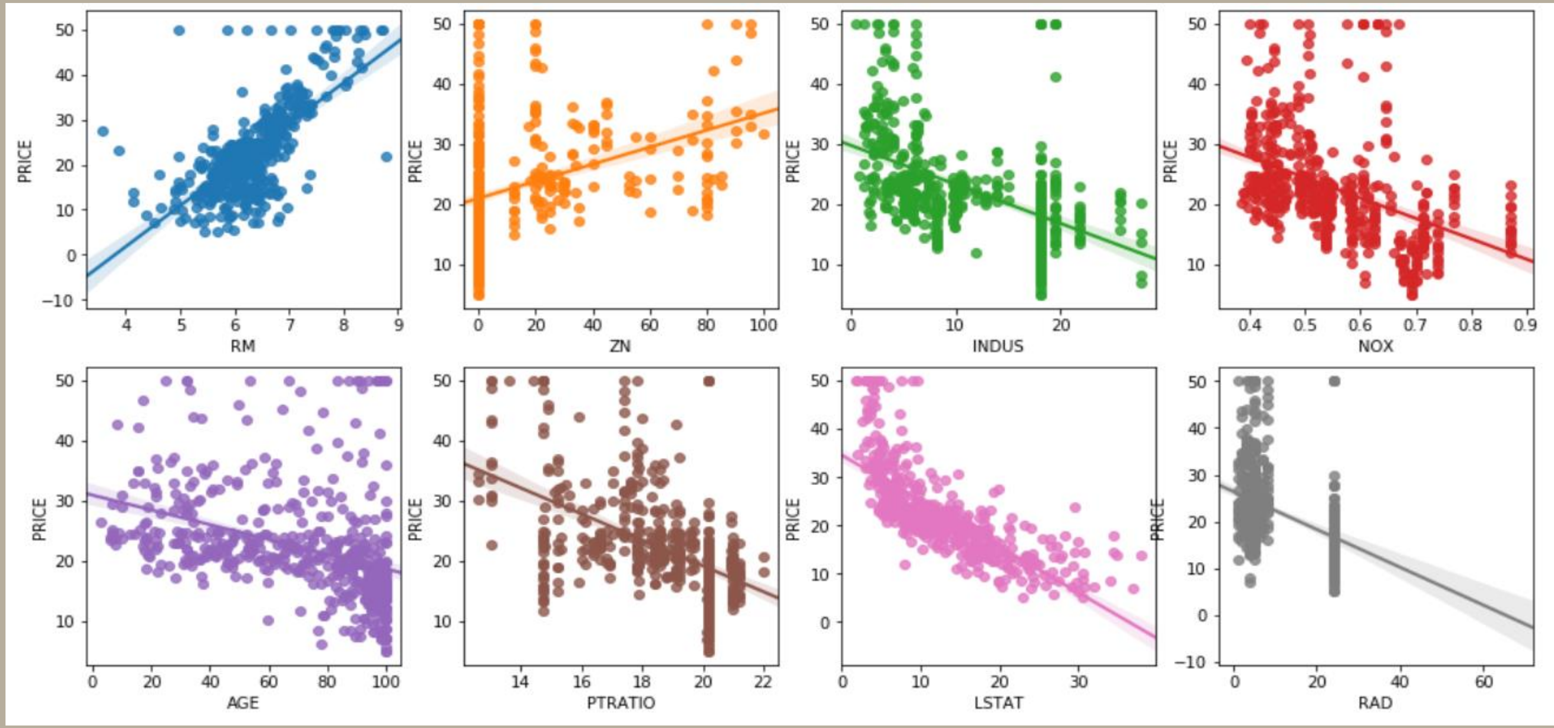
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
CRIM       506 non-null float64
ZN         506 non-null float64
INDUS      506 non-null float64
CHAS       506 non-null float64
NOX        506 non-null float64
RM         506 non-null float64
AGE        506 non-null float64
DIS        506 non-null float64
RAD        506 non-null float64
TAX        506 non-null float64
PTRATIO    506 non-null float64
B          506 non-null float64
LSTAT      506 non-null float64
PRICE      506 non-null float64
dtypes: float64(14)
memory usage: 55.4 KB
```

5.4 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측

- LinearRegression을 이용해 보스턴 주택 가격 예측

```python
# 2개의 행과 4개의 열을 가진 subplots를 이용. axs는 4x2개의 ax를 가짐.
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])
```

시각화



5.4 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측

- LinearRegression 클래스를 이용해 주택 가격 회귀 모델 만들기

```
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, random_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))
print('Variance score : {0:.3f}'.format(r2_score(y_test, y_preds)))
```

```
MSE : 17.297 , RMSE : 4.159
Variance score : 0.757
```

```
print('절편 값: ', lr.intercept_)
print('회귀 계수 값: ', np.round(lr.coef_, 1))
```

```
절편 값:  40.995595172164336
회귀 계수 값:  [ -0.1   0.1   0.    3.  -19.8   3.4   0.   -1.7   0.4  -0.   -0.9   0.
  -0.6]
```

5.4 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측

```
# 회귀 계수를 큰 값 순으로 정렬하기 위해 Series로 생성. index가 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
B           0.0
TAX        -0.0
AGE         0.0
INDUS       0.0
CRIM       -0.1
LSTAT      -0.6
PTRATIO    -0.9
DIS        -1.7
NOX       -19.8
dtype: float64
```

- RM 피처가 target인 가격에 대해 가장 높은 양 방향의 선형성을 갖고, NOX 피처가 가장 높은 음 방향의 선형성을 가짐

- 계수 값이 너무 커 후에 최적화가 필요해 보임

5.4 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측

• cross_val_score( )을 이용해 교차 검증으로 MSE와 RMSE 측정

```
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", cv = 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))
print(' 5 folds 의 개별 RMSE scores : ', np.round(rmse_scores, 2))
print(' 5 folds 의 평균 RMSE : {0:.3f} '.format(avg_rmse))
```

```
5 folds 의 개별 Negative MSE scores:  [-12.46 -26.05 -33.07 -80.76 -33.31]
5 folds 의 개별 RMSE scores :  [3.53 5.1  5.75 8.99 5.77]
5 folds 의 평균 RMSE : 5.829
```

추가 내용

RMSE 단점

• 예측 대상의 크기에 영향을 바로 받게 됨 >> 크기 의존적 에러

• 비율 에러

MAPE(Mean Absolute Percentage Error)

$$\mathrm{M} = \frac{100}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right|,$$

• A_t=실제 값, F_t=예측 값
• 오차를 절대적 크기가 아닌 비율적 크기로 보고자 하는 방법
• A_t가 0에 가까울수록 비정상적인 값이 나옴

•크기 조정된 에러

MASE(Mean Absolute Scaled Error)

$$\mathrm{MASE} = \frac{1}{T}\sum_{t=1}^{T}\left(\frac{|e_t|}{\frac{1}{T-1}\sum_{t=2}^{T}|Y_t - Y_{t-1}|}\right) = \frac{\sum_{t=1}^{T}|e_t|}{\frac{T}{T-1}\sum_{t=2}^{T}|Y_t - Y_{t-1}|}$$

MAPE과 비교하여 분모의 성질이 다름. 분모가 의미하는 것은 평소의 변동 폭
즉, MASE는 평소의 변동폭에 비해 얼마나 오차가 나는지를 측정하는 기준
시계열 예측에 있어서 변동폭이 큰 지표와 낮은 지표를 같이 비교하는 등의 상황에서 쓰인다고 함