5강 회귀(308p)

01 회귀 소개

❤️ scikit-learn의 통일된 기준: 모든 `scoring` 함수는 "값이 클수록 좋은 점수"

- **오차(MSE, MAE)** 등은 → 작을수록 좋음
- 오차 관련 점수(MSE, MAE 등)는 **음수로 반환 →** 그래야 "큰 게 좋은 점수" 체계가 유지

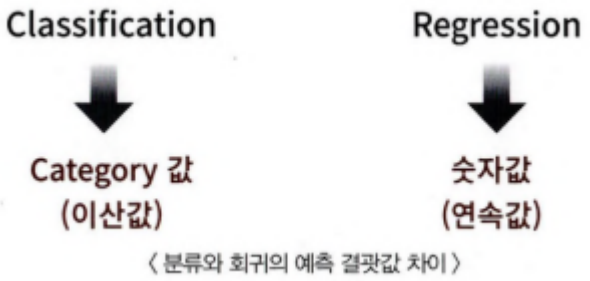
데이터 값이 평균과 같은 일정한 값으로 돌아가려는 경향

- 독립변수 = 피처
- 종속변수 = 결정 값
- 최적의 회귀 계수(Regression coefficients) → 주요 과제

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

〈 회귀 유형 구분 〉

- 회귀 4가지 종류: 독립 변수의 개수 & 회귀 계수의 결합



〈 분류와 회귀의 예측 결괏값 차이 〉

- 지도학습 → 분류와 회귀
  - 이산 vs 연속

- 선형 회귀가 가장 많음:
  - **실제 값과 예측값의 차이(오류의 제곱 값)를 최소화하는 직선형 회귀선을 최적화하는 방식**
  - 규제: 선형회귀에서 과적합 문제를 해결하기 위해서 회귀 계수에 페널티 값을 적용
- 일반 선형 회귀: 예측값과 실제 값의 RSS(Residual Sum of Squares)를 최소화할 수 있도록 회귀 계수를 최적화하며, 규제(Regularization)를 적용하지 않은 모델
- 릿지(Ridge): 선형 회귀 + L2 규제
  - L2 규제는 상대적으로 큰 회귀 계수 값의 예측 영향도를 감소시키기 위해서 회귀 계수값을 더 작게 만드는 규제 모델
- 라쏘(Lasso): 선형 회귀 + L1 규제
  - L1 규제는 예측 영향력이 작은 피처의 회귀 계수를 0으로 만들어 회귀 예측 시 피처가 선택되지 않게 하는 것입니다. → 피처 선택 기능
- 엘라스틱넷(ElasticNet)： L2, L1 규제를 함께 결합
  - 피처가 많은 데이터 세트에서 적용
  - L1 규제로 피처의 개수를 줄임 + L2 규제로 계수 값의 크기를 조정
- 로지스틱 회귀(Logistic Regression)： 사실은 분류에 사용되는 선형 모델
  - 매우 강력한 분류 알고리즘
  - 일반적으로 이진 분류뿐만 아니라 희소 영역의 분류, 예를 들어 텍스트 분류와 같은 영역에서 뛰어난 예측 성능을 보임

02 단순 선형 회귀(310-312p)

❤️ RSS를 최소로 하는 회귀계수**(w 변수)** 찾기

- 독립변수도 하나, 종속변수도 하나
- $Y = w0 + w1 * X +$ 오류값
  - intercept 절편 = 회귀 계수
- 잔차: 실제 값과 회귀 모델의 차이에 따른 오류 값
  - 전체 데이터의 잔차 합이 최소가 되는 모델 = 오류 값 합이 최소가 될 수 있는 최적의 회귀 계수 찾기
  - 평균적인 오차크기: 절댓값을 취해서 더하거나(Mean Absolute Error) or 오류 값의 제곱을 구해서 더하는 방식(RSS, Residual Sum of Square) / N
  - RSS → 미분 등 계산이 더 편리
  - $Error^2 = RSS$

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

(i는 1부터 학습 데이터의 총 건수 N까지)

- **RSS는 비용이며 w변수(회귀계수)로 구성되는 RSS를 비용 함수 = 손실함수라고 함**
- **머신러닝 회귀 알고리즘은 데이터를 계속 학습하며 비용 함수가 반환하는 값(오류 값)을 지속해서 감소시키고 최종적으로는 더 이상 감소하지 않는 최소의 오류 값을 구하는 것이다.**

xw1 + w0 - y→ x^2w^2 + 2xw0w1 - 2xyw1 → 2x^2w + 2xw0 - 2xy

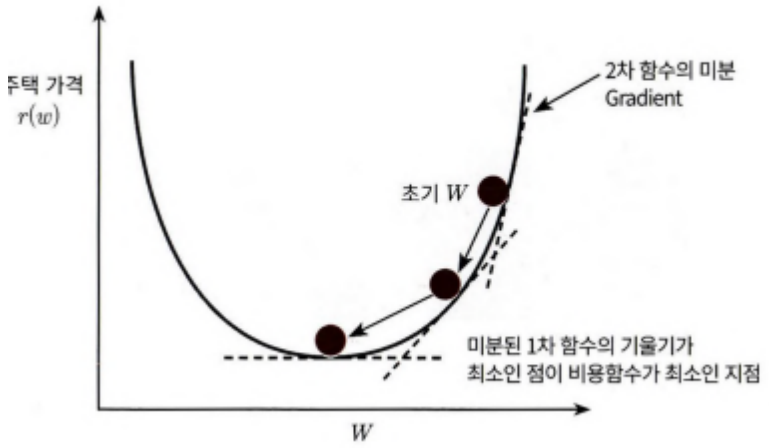
330(pdf)

03 비용 최소화하기 - 경사 하강법(Gradient Descent) 소개 (312-321p, 10p)

❤️ 비용 함수가 최소가 되는 w 파라미터 구하기

- **비용 함수의 반환 값, 즉 예측값과 실제 값의 차이가 작아지는 방향성을 가지고 W 파라미터를 지속해서 보정**

- 데이터를 기반으로 알고리즘이 스스로 학습한다'는 머신러닝의 개념을 가능하게 만들어 준 핵심 기법 중 하나
- '점진적으로' 반복적인 계산을 통해 w 파라미터 값을 업데이트하면서 오류 값이 최소가 되는 W 파라미터를 구하는 방식
- 만약 손실 함수(비용 함수)가 2차 방정식이면 → 1차 함수 기울기의 값이 0일 때 비용 함수가 최소인 지점으로 간주하고 그때의 w를 반환



- R(w) → 변수가 w 파라미터로 이루어진 함수
  - R(w)를 미분해서 미분함수의 최솟값
  - **<u>두 개의 w 파라미터 →각 변수에 편미분</u>**
  - R(w)를 최소화하는 파라미터 → 각각 r(w)를 w0, w1으로 순차적으로 편미분을 수행

$$\frac{\partial R(w)}{\partial w_1} = \frac{2}{N}\sum_{i=1}^{N} -x_t * (y_i - (w_0 + w_1 x_i)) = -\frac{2}{N}\sum_{i=1}^{N} x_i * (\text{실제값}_i - \text{예측값}_i)$$

$$\frac{\partial R(w)}{\partial w_0} = \frac{2}{N}\sum_{i=1}^{N} -(y_i - (w_0 + w_1 x_i)) = -\frac{2}{N}\sum_{i=1}^{N} (\text{실제값}_i - \text{예측값}_i)$$

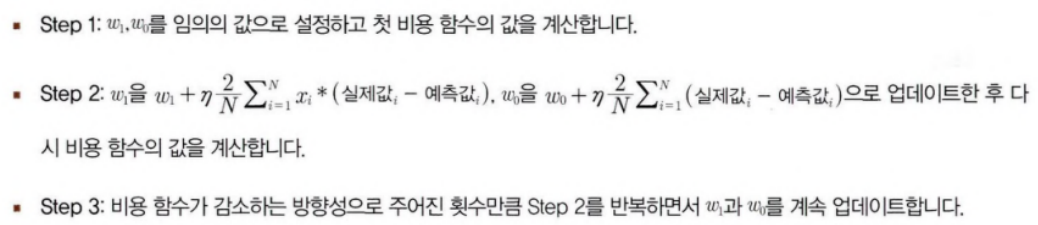
→ $2x^2w + 2xw0 - 2xy$ → $2x(xw + w0 - y) * 1/N$

→ $-2/N \sum_{i=1}^{N} x(y - (w0 + xw))$

→ x, y에는 차례대로 데이터 세트가 들어가면서 학습될 것(세트마다 다른 상수값을 가지는 이차 함수라고 보면됨)

→ 각각의 편미분 값을 반복적으로 보정 → w0, w1 업데이트 → R(w)이 최소가 되는 w0, w1 찾기

- 업데이트
  - 학습률: 편미분 값이 너무 클 수 있기에 보정계수 곱함
- 프로세스




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

np.random.seed(0)
x  = 2 * np.random.rand(100, 1)
y = 6 + 4 * x+ np.random.randn(100, 1) #정규분포 노이즈를 추가 → 실제 데이터처럼 완벽한 직선이 아니라 약간 퍼지게

plt.scatter(x, y)
```

```
def gradient_weight_updates(w1, w0, x, y, learning_rate = 0.01):
  N = len(y)
  #기울기 업데이트 벡터
  w1_update = np.zeros_like(w1)
  w0_update = np.zeros_like(w0)

  #예측 배열 계산, 예측과 실제 값의 차이 계산
```

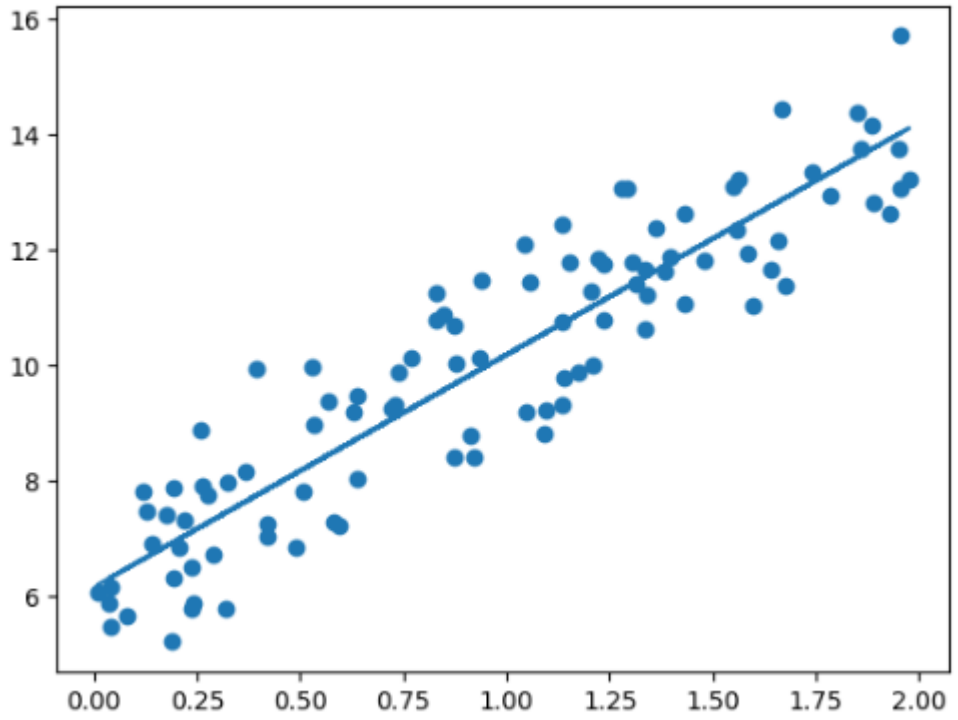
```
  y_pred = np.dot(x, w1.T) + w0
  diff = y - y_pred
  w0_factors = np.ones((N, 1))

  #업데이트
  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
```

```
def gradient_descent_steps(x, y, iters = 10000):
  w1 = np.zeros((1, 1))
  w0 = np.zeros((1,1))
  for i in range(iters):
    w1_update,w0_update = gradient_weight_updates(w1, w0, x, y)
    w1 = w1 - w1_update
    w0 = w0 - w0_update
  return w1, w0
```

```
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: {0:.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: 4.022
Gradient Descent Total Cost:0.9935
```

```
[<matplotlib.lines.Line2D at 0x7d01f8e327e0>]
```



확률적 경사 하강법

- 일부 데이터만 이용해 피가 업데이트되는 값을 계산하므로 경사 하강법에 비해서 빠른 속도를 보장
- 대용량의 데이터의 경우 대부분 확률적 경사 하강법이나 미니 배치 확률적 경사 하강법을 이용해 최적 비용함수를 도출

(미니 배치) 확률적 경사 하강법

```
def stochastic_gradient_descent_steps(x, y, batch_size = 10, iters = 1000):
  w0 = np.zeros((1,1))
  w1 = np.zeros((1,1))

  for ind in range(iters):
    stochastic_random_index = np.random.permutation(x.shape[0])
    sample_x = x[stochastic_random_index[:batch_size]]
    sample_y = y[stochastic_random_index[:batch_size]]

    w1_update, w0_update = gradient_weight_updates(w1, w0, sample_x, sa
mple_y)
    w1 = w1 - w1_update
    w0 = w0 - w0_update
```

```
  return w1, w0

w1, w0 = stochastic_gradient_descent_steps(x, y, iters = 1000)
print("w:", 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)))
```

```
w: 4.029 w0: 6.2
Stochastic Gradient Descent Total Cost:0.9949
```

피처가 여러 개인 경우

- 피처가 M개 → 회귀 계수도 M 十 1(1개는 w0)개
- 선형대수

❤️ $\hat{y} = \text{np.dot}(X_{\text{mat}}, W^T) + w0$

- 데이터 개수가 n개, 피처가 m개

$$\begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} * \begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix}^T + w_0$$

(Feature 1, Feature 2, ..., Feature M / $\hat{Y}$, X_{mat} / * 내적)

$$\begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \cdots & x_{2m} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} * \begin{bmatrix} w_0 & w_1 & w_2 & \cdots & w_m \end{bmatrix}^T$$

(1값을 가진 피처 추가 / Feat 0, Feat 1, Feat 2, ..., Feat M / w_0을 W 배열 내에 포함 / * 내적)

$$\hat{Y} = X_{mat} * W^T$$

04 사이킷런 LinearRegression을 이용한 보스턴 주택 가격 예측(321-328p, 8p)

사이킷런의 linear_models 모듈

- 다양한 종류의 선형 기반 회귀를 클래스로 구현해 제공
- LinearRegression(규제 x)

LinearRegression 클래스 - Ordinary Least Squares

예측값과 실제 값의 RSS(Residual Sum of Squares) 최소화 → OLS(Ordinary Least Squares) 추정 방식으로 구현한 클래스

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

회귀 계수(Coefficients)인 W를 coef_속성에 저장

- fit_intercept： 불린 값
  - 디폴트는 True → 절편 값을 계산할 것인지 말지
  - False → 절편이 0
- normalize: 불린 값으로 디폴트는 False
  - fit_intercept가 False → 무시
  - 만일 True이면 회귀를 수행하기 전에 입력 데이터 세트를 정규화
- 속성
  - coef_ → fit（） 메서드를 수행했을 때 회귀 계수가 배열 형태로 저장하는 속성 → Shape는 (Target 값 개수. 피처 개수).
  - intercept_ → intercept 값
- 다중 공선성 문제
  - 피처 간의 상관관계가 매우 높은 경우 분산이 매우 커져서 오류 민감해짐
  - 독립적 피처만 남기고 나머지 삭제 혹은 규제
  - PCA를 통해 차원 축소
- 회귀 평가 지표
  - 실제 값과 회귀 예측값의 차이 값을 기반으로 한 지표가 중심
  - 더하면 오류가 상쇄(+, -가 섞여서)

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.	$MSE = \frac{1}{n}\sum_{i=1}^{n}(Yi - \hat{Y}i)^2$
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.	$R^2 = \frac{\text{예측값 } Variance}{\text{실제값 } Variance}$

cross_val_score, GridSearchCV와 같은 Scoring 함수

평가 방법	사이킷런 평가 지표 API	Scoring 함수 적용 값
MAE	metrics.mean_absolute_error	'neg_mean_absolute_error'
MSE	metrics.mean_squared_error	'neg_mean_squared_error'
RMSE	metrics.mean_squared_error를 그대로 사용하되 squared 파라미터를 False로 설정.	'neg_root_mean_squared_error'
MSLE	metrics.mean_squared_log_error	'neg_mean_squared_log_error'
R^2	metrics.r2_score	'r2'

RMSE는 mean_squared_error(실제값, 예측값, squared=False)

- 사이킷런의 Scoring 함수가 score값이 클수록 좋은 평가 결과로 자동 평가
- 회귀 평가 지표의 경우 값이 커지면 오히려 나쁜 모델

LinearRegressionS 이용해 보스턴 주택 가격 회귀 구현

```
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 fetch_california_housing
import warnings
warnings.filterwarnings('ignore')
```

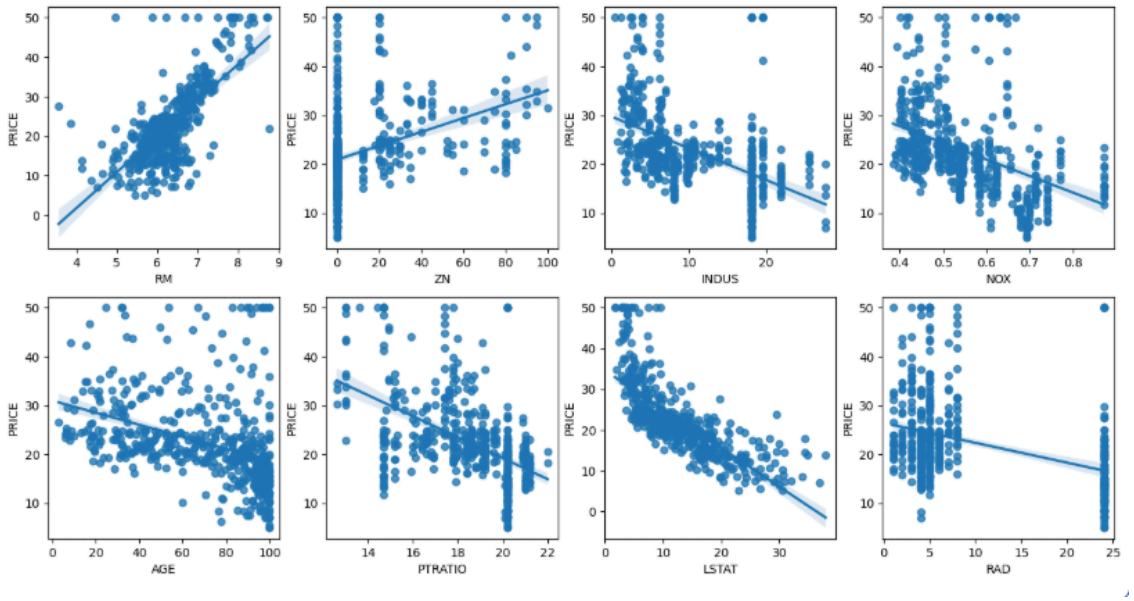
```
%matplotlib inline

boston = pd.read_csv("/content/drive/MyDrive/Euron/BostonHousing.csv")
boston.rename(columns = {'CMEDV':'PRICE'}, inplace = True)
boston.head()
```

시본(Seaborn)의 `regplot( )` 함수: X, Y 축 값의 산점도, 선형 회귀 직선

`matplotlib.subplots( )`를 이용해 각 ax마다 칼럼과 PRICE의 관계를 표현

- 여러 개의 그래프를 한 번에 표현
- `ncols`는 열 방향으로 위치할 그래프의 개수이며, `nrows`는 행 방향으로 위치할 그래프의 개수

```
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 = i // 4
  col = i % 4
  sns.regplot(x=feature, y = 'PRICE', data = boston, ax = axs[row][col])
```



→ RM과 LSTAT의 PRICE 영향도

- RM(방 개수) : Positive Linearity
- LSTAT(하위 계층의 비율) : Negative Linearity

LinearRegression 클래스를 이용해 보스턴 주택 가격의 회귀 모델

- `mean_squared_error()`와 `r2_score()` API를 이용해 MSE와 R2 Score를 측정

```
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 = boston['PRICE']
X_data = boston.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)

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 : 16.956, RMSE : 4.118
Variance score : 0.760
```

주택가격 모델의 intercept(절편)과 coefficients(회귀 계수) 값

- 절편은 LinearRegression 객체의 `intercept_`속성에, 회귀 계수는 `coef_` 속성에 값이 저장

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

```
절편 값: -779.9344571908823
회귀 계수 값: [  0.   -6.    9.3  -0.1   0.1   0.    3.  -17.3   3.3   0.   -1.6   0.4
  -0.   -0.9   0.   -0.6]
```

피처별 회귀 계수 값으로 다시 매핑

- 판다스 Series의 `sort_values()` 함수를 이용

```
coeff = pd.Series(data = np.round(lr.coef_, 1), index = X_data.columns)
coeff.sort_values(ascending = False)
```

	0
LAT	9.3
RM	3.3
CHAS	3.0
RAD	0.4
ZN	0.1
AGE	0.0
INDUS	0.0
Unnamed: 0	0.0
B	0.0
TAX	-0.0
CRIM	-0.1
LSTAT	-0.6
PTRATIO	-0.9
DIS	-1.6
LON	-6.0
NOX	-17.3

NOX 피처의 회귀 계수 - 값이 너무 커 보임 → 최적화 필요

5개의 폴드 세트에서 `cross_val_score( )`를 이용해 교차 검증으로 MSE & RMSE를 측정

- 반환된 값에 다시 -1을 곱해야 양의 값인 원래 모델에서 계산된 MSE

- MSE값에 넘파이의 `sqrt() 함수`를 적용해 RMSE

```
from sklearn.model_selection import cross_val_score

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

lr = LinearRegression()

neg_mse_scores = cross_val_score(lr, X_data, y_target, scoring = "neg_mea
n_squared_error", cv = 5)
rmse_scores = np.sqrt(-1 * neg_mse_scores)
avg_rmse = np.mean(rmse_scores)

print(' 5 교차 검증의 개별 Negative MSE scores: ', np.round(neg_mse_scores,
2))
print(' 5 교차 검증의 개별 RMSE scores : ', np.round(rmse_scores, 2))
print(' 5 교차 검증의 평균 RMSE : {0:.3f} '.format(avg_rmse))
```

```
 5 교차 검증의 개별 Negative MSE scores:  [-22.37 -26.86 -33.14 -82.68 -32.
4 ]
 5 교차 검증의 개별 RMSE scores :  [4.73 5.18 5.76 9.09 5.69]
 5 교차 검증의 평균 RMSE : 6.091
```

교차검증은

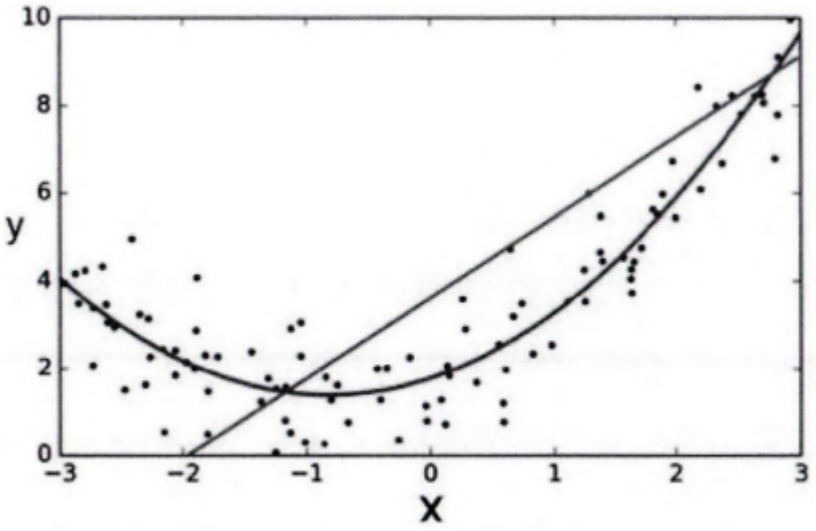
- "쉬운 테스트셋" 뿐 아니라
- "어려운 테스트셋"까지 모두 포함한 평균을 내기 때문에

  → 더 **보수적(=큰 오차)** 결과가 나오는 게 자연스럽다

05 다항 회귀와 과적합. 과소적합 이해 (329-337p, 9p)

- 2차, 3차 방정식과 같은 다항식
- **다항 회귀는 선형 회귀**
- 회귀에서 선형 회귀/비선형 회귀를 나누는 기준

    - 회귀 계수가 선형/비선형인지
    - 독립변수의 선형/비선형 여부 ❌
- 단순 선형 회귀 직선형 < 다항 회귀 곡선형



〈 주어진 데이터 세트에서 다항 회귀가 더 효과적임 〉

비선형 함수를 선형 모델에 적용시키는 방법을 사용해 구현

- `PolynomialFeatures 클래스`
- 피처를 Polynomial(다항식) 피처로 변환
- `degree` 파라미터 : 단항식 피처를 degree에 해당하는 다항식 피처로 변환
- `fit()` , `transform()`

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

def polynomial_func(X): ##이것에 근사하도록 계수를 유추
  y = 1+ 2*X[:,0] + 3*X[:,0]**2 + 4*X[:,1]**3
  return y

X = np.arange(4).reshape(2,2)
print('일차 단항식 계수 feature:\n', X)
y = polynomial_func(X)
print('삼차 다항식 결정값: \n', y)
```

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

#학습, 회귀계수 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)
```

```
3차 다항식 계수 feature:
 [[ 1.  0.  1.  0.  0.  1.  0.  0.  0.  1.]
 [ 1.  2.  3.  4.  6.  9.  8. 12. 18. 27.]]
Polynomial 회귀 계수
 [0.   0.18 0.18 0.36 0.54 0.72 0.72 1.08 1.62 2.34]
Polynomial 회귀 Shape :  (10,)
```

3차 다항식 Polynomial 변환 → 다항식 계수 피처가 10개, 회귀 계수가 10개

→ 원래 다항식 $1 + 2x_1 + 3x_1^2 + 4x_2^3$의 계수 값과는 차이가 있음

사이킷런의 `Pipeline` 객체 → 한 번에 다항 회귀를 구현

- X → 첫번째 모델에 적용하여 X값 변환 → 반환 X 와 y → 두번째 선형 회귀 모델에 적용

```
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

#파이프라인 객체로 변환과 선형회귀 모델 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))
```

```
Polynomial 회귀 계수
 [0.   0.18 0.18 0.36 0.54 0.72 0.72 1.08 1.62 2.34]
```

다항 회귀를 이용한 과소적합 및 과적합 이해

다항 회귀의 차수(degree) → 높일수록 학습 데이터에만 너무 맞춘 학습

- plt.subplot(nrows, ncols, index)
  - nrows → 행의 개수 (즉, 몇 줄로 배치할지)
  - ncols → 열의 개수 (한 줄에 몇 개 넣을지)
  - index → 현재 그래프가 몇 번째 위치인지 (1부터 시작)
- `xticks=()` → x축 눈금 제거
- `yticks=()` → y축 눈금 제거
- w_0 → **편향(bias)**, 또는 **절편(intercept)** 이라고 부름
- `xlim`, `ylim`: 축 범위를 0~1, -2~2로 고정
- `legend`: “Samples”, “Model”, “True Function” 라벨을 자동 표시

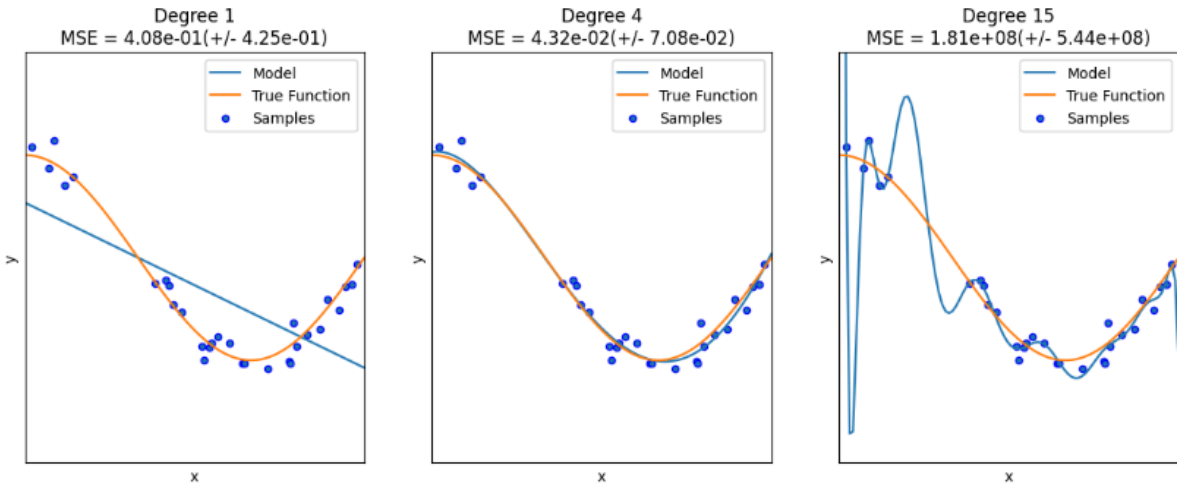
❤️ Scikit-learn의 모든 모델은 **2차원 입력 (n_samples, n_features)** 을 기대

- `X_test[:, np.newaxis]`
- `reshape(-1,1)`

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
%matplotlib inline

#임의의 값으로 구성된 X값에 대해 코사인 변환 값을 반환
def true_fun(X):
  return np.cos(1.5*np.pi*X)

np.random.seed(0)
n_samples = 30
X = np.sort(np.random.rand(n_samples))
y = true_fun(X) + np.random.randn(n_samples) * 0.1

plt.figure(figsize = (14, 5))
degrees = [1, 4, 15]

#다항 회귀 차수 변화하며 비교

for i in range(len(degrees)):
  ax = plt.subplot(1, len(degrees), i+1)
  plt.setp(ax, xticks = ( ), yticks = ( ))

  polynomial_features = PolynomialFeatures(degree = degrees[i], include_bi
as = False)
  linear_regression = LinearRegression()
  pipeline = Pipeline([("polynomial_feature", polynomial_features),
                       ("linear_regression", linear_regression)])
  pipeline.fit(X.reshape(-1,1), y)

  #교차검증으로 다항회귀 평가
  scores = cross_val_score(pipeline, X.reshape(-1,1), y, cv =10, scoring = "n
eg_mean_squared_error")
  #파이프라인 세부 객체에 접근하는 named_steps['객체명'] → 회귀계수 추출
  coeff = pipeline.named_steps['linear_regression'].coef_
```

```
  print('\nDegree {0} 회귀 계수는 \n{1}'.format(degrees[i], np.round(coeff, 2)))
  print('Degree {0} 교차 검증 점수: {1:.3f}'.format(degrees[i], -1*np.mean(scores)))

  #0부터 1까지 테스트 데이터를 100개로 나눠 예측 수행
  #테스트 데이터 세트에 회귀 예측 수행하고 예측 곡선과 실제 곡선을 그려서 비교
  X_test = np.linspace(0, 1, 100)
  plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label = 'Model')
  plt.plot(X_test, true_fun(X_test), label = 'True Function')
  plt.scatter(X, y, edgecolor = 'b', s=20, label = 'Samples')

  plt.xlabel("x"); plt.ylabel("y"); plt.xlim((0,1)); plt.ylim((-2,2)); plt.legend(loc = "best")
  plt.title("Degree {}\nMSE = {:.2e}(+/- {:.2e})".format(degrees[i], -1*np.mean(scores), np.std(scores)))
plt.show()
```

```
Degree 1 회귀 계수는
[-1.61]
Degree 1 교차 검증 점수: 0.408

Degree 4 회귀 계수는
[  0.47 -17.79  23.59  -7.26]
Degree 4 교차 검증 점수: 0.043

Degree 15 회귀 계수는
[-2.98293000e+03  1.03899390e+05 -1.87416123e+06  2.03716219e+07
 -1.44873283e+08  7.09315363e+08 -2.47065792e+09  6.24561050e+09
 -1.15676510e+10  1.56894936e+10 -1.54006023e+10  1.06457264e+10
 -4.91377530e+09  1.35919645e+09 -1.70380786e+08]
Degree 15 교차 검증 점수: 181238256.564
```



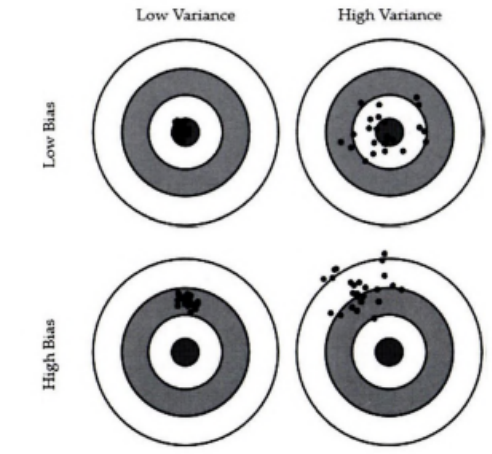
→ 균형 잡힌(Balanced) 모델 중요

편향-분산 트레이드오프(Bias-Variance Trade off)

❤️ **오류 Cost 값이 최대로 낮아지는 모델을 구축하기**

- 분산과 편향의 균형 '골디락스'

구분	의미
편향(bias)	모델이 **얼마나 평균적으로 실제값과 멀리 떨어져 있는지** — 즉, **체계적 오차**
분산(variance)	모델이 **데이터가 바뀔 때마다 예측이 얼마나 요동치는지** — 즉, **불안정성**



〈 편향과 분산의 고/저에 따른 표현,
http://scott.fortmann-roe.com/docs/BiasVariance.html에서 발췌 〉

- 저편향, 저분산 ⇒ 이상적(드물다)
- 저편향, 고분산 → 실제 결과에 근접하지만, 예측 결과가 실제 결과를 중심으로 넓게 분포
- 고편향, 저분산 ⇒ 정확한 결과에서 벗어나면서 특정 부분에 예측이 집중
- 고편향, 고분산 ⇒ 정확한 예측 결과 벗어나면서 넓은 부분에 분포

고편향(High Bias)성

- 매우 단순화된 모델로서 지나치게 한 방향성으로 치우친 경향

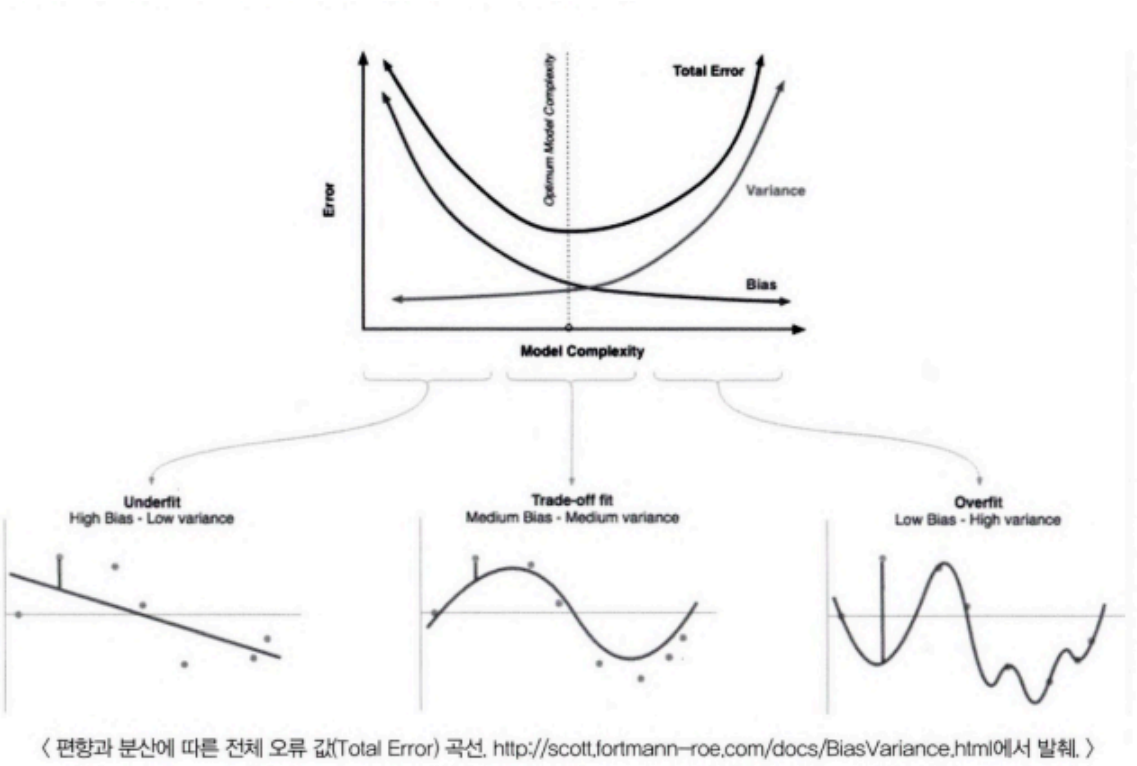
고분산(High Variance)성

- 학습 데이터 하나 하나의 특성을 반영하면서 지나치게 복잡하고 높은 변동성

편향과 분산은 한쪽이 높으면 한쪽이 낮아지는 경향

- 편향 높으면 분산 낮음 → 과소적합

- 편향 낮으면 분산 높음 → 과적합



〈 편향과 분산에 따른 전체 오류 값(Total Error) 곡선. http://scott.fortmann-roe.com/docs/BiasVariance.html에서 발췌. 〉

- 편향을 점점 낮추면 동시에 분산이 높아지고 전체 오류도 낮아짐
- 전체 오류가 가장 낮아지는 '골디락스' 지점을 통과 → 분산을 지속적으로 높이면 → 오류 증가

선형 회귀 모델을 위한 데이터 변환

1. **스케일링/정규화 작업**
    a. 피처값과 타깃값 분포 → 정규 분포(즉 평균을 중심으로 종 모양으로 데이터 값이 분포된 형태) 형태 선호
    b. 특히 타깃값 특정 값으로 왜곡된 형태면 예측 성능에 부정적 영향
    c. 무조건 예측 성능이 향상되는 것 X

- `StandardScaler 클래스` : 평균이 0, 분산이 1 예측 성능 향상 그닥일수도
- `MinMaxScaler` 클래스로 최솟값 0, 최댓값이 1인 정규화 * → , 다항식 변환에 잘 안씀. 예측 성능 향상 그닥일수도

- 스케일링/정규화를 수행한 데이터 세트에 다시 다항 특성을 적용 변환 → 과적합 이슈 가능
- ***원래 값에 log 함수를 → 적용 분포의 모양(shape)을 바꿈

타깃값의 경우는 일반적으로 로그 변환

get_scaled_data() 함수

- method 인자로 변환방법을 결정
- p_degree는 다항식 특성을 추가할 때 다항식 차수가 입력됨(2를 넘기지 않음)
- np.log()가 아니라 np.log1p() 사용 → 언더 플로우 방지 → 인자값에 1을 더하는 방식
  - 0 이상이어야 함

  💡 `np.clip(x, a_min, a_max)` → 모든 값을 **a_min과 a_max 사이로 제한(clamp)** 해주는 함수

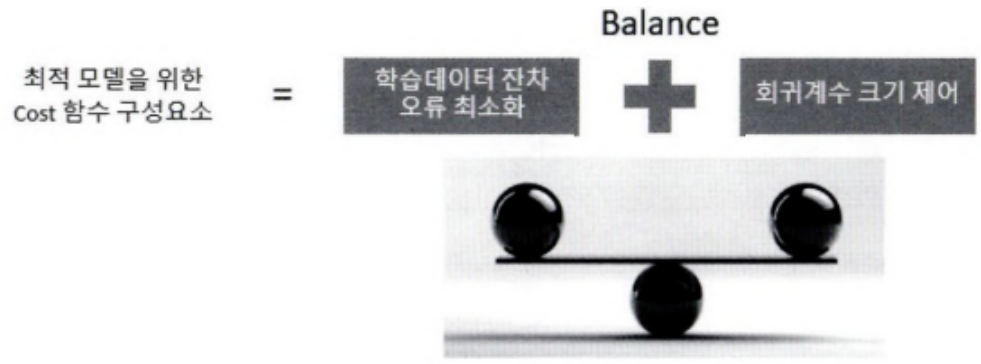
⇒ scaled_methods = [(None, None), ('Standard', None), ('Standard', 2), ('MinMax', None), ('MinMax', 2), ('Log', None)] 5가지 방법

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, Polyno
mialFeatures

def get_scaled_data(method = 'None', p_degree= None, input_data = Non
e):
  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(np.clip(input_data, a_min=0, a_max=None))
  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
```

```
alphas = [0.1, 1, 10, 100]

scaled_methods = [(None, None), ('Standard', None), ('Standard', 2), ('Min
Max', None), ('MinMax', 2), ('Log', None)]

for scaled_method in scaled_methods:
  X_data_scaled = get_scaled_data(method = scaled_method[0], p_degree
= scaled_method[1],
                                  input_data = X_data)
  print('\n## 변환 유형:{0}, Polynomial Degree:{1}'.format(scaled_method[0],
scaled_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)
```

```
## 변환 유형:None, Polynomial Degree:None
alpha 0.1일 때 5 folds의 평균 RMSE: 5.952
alpha 1일 때 5 folds의 평균 RMSE: 5.667
alpha 10일 때 5 folds의 평균 RMSE: 5.553
alpha 100일 때 5 folds의 평균 RMSE: 5.457

## 변환 유형:Standard, Polynomial Degree:None
alpha 0.1일 때 5 folds의 평균 RMSE: 6.087
alpha 1일 때 5 folds의 평균 RMSE: 6.057
alpha 10일 때 5 folds의 평균 RMSE: 5.852
alpha 100일 때 5 folds의 평균 RMSE: 5.553

## 변환 유형:Standard, Polynomial Degree:2
alpha 0.1일 때 5 folds의 평균 RMSE: 12.541
alpha 1일 때 5 folds의 평균 RMSE: 8.246
alpha 10일 때 5 folds의 평균 RMSE: 5.924
alpha 100일 때 5 folds의 평균 RMSE: 5.006

## 변환 유형:MinMax, Polynomial Degree:None
alpha 0.1일 때 5 folds의 평균 RMSE: 6.021
alpha 1일 때 5 folds의 평균 RMSE: 5.700
alpha 10일 때 5 folds의 평균 RMSE: 5.872
alpha 100일 때 5 folds의 평균 RMSE: 7.654

## 변환 유형:MinMax, Polynomial Degree:2
alpha 0.1일 때 5 folds의 평균 RMSE: 5.559
alpha 1일 때 5 folds의 평균 RMSE: 4.475
alpha 10일 때 5 folds의 평균 RMSE: 5.344
alpha 100일 때 5 folds의 평균 RMSE: 6.467

## 변환 유형:Log, Polynomial Degree:None
alpha 0.1일 때 5 folds의 평균 RMSE: 4.729
alpha 1일 때 5 folds의 평균 RMSE: 4.720
alpha 10일 때 5 folds의 평균 RMSE: 5.177
alpha 100일 때 5 folds의 평균 RMSE: 6.556
```

06 규제 선형 모델 - 릿지, 라쏘, 엘라스틱넷 (337-350p, 14p)



$$\text{비용 함수 목표} = \text{Min}(\text{RSS(W)} + \text{alpha} * \|W\|_2^2)$$

alpha는 학습 데이터 적합 정도와 회귀 계수 값의 크기 제어를 수행하는 튜닝 파라미터

- alpha가 0（또는 매우 작은 값) → 비용 함수 식은 기존과 동일 → W가 커져도 상쇄 가능 → 학습 데이터 적합
- alpha가 무한대 (또는 매우 큰 값） → W 값을 작게 → Cost 작게 만들 수 있음 → 과적합 개선

❤️ 규제: 비용 함수에 alpha 값으로 페널티를 부여해 회귀 계수 값의 크기를 감소시켜 과적합을 개선

- L2 - 릿지, W 제곱에 페널티 부여
- L1 - 라쏘, W의 절댓값에 페널티 부여 → 영향력이 적은 회귀 계수 값을 0으로 만듦

릿지 회귀

사이킷런의 `Ridge class`

- 주요 생성 파라미터 `alpha` → L2 규제

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

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 folds의 개별 Negative MSE scores: ', np.round(neg_mse_scores, 2))
print('5 folds의 개별 RMSE scores : ', np.round(rmse_scores, 2))
print('5 folds의 평균 RMSE : {0:.3f} '.format(avg_rmse))
```

```
5 folds의 개별 Negative MSE scores:  [-13.47 -23.56 -28.86 -74.54 -27.42]
5 folds의 개별 RMSE scores :  [3.67 4.85 5.37 8.63 5.24]
5 folds의 평균 RMSE : 5.553
```

릿지값 0, 0.1, 1, 10, 100 변환 RMSE, 회귀계수 값 변화

시각화, DataFrame에 저장

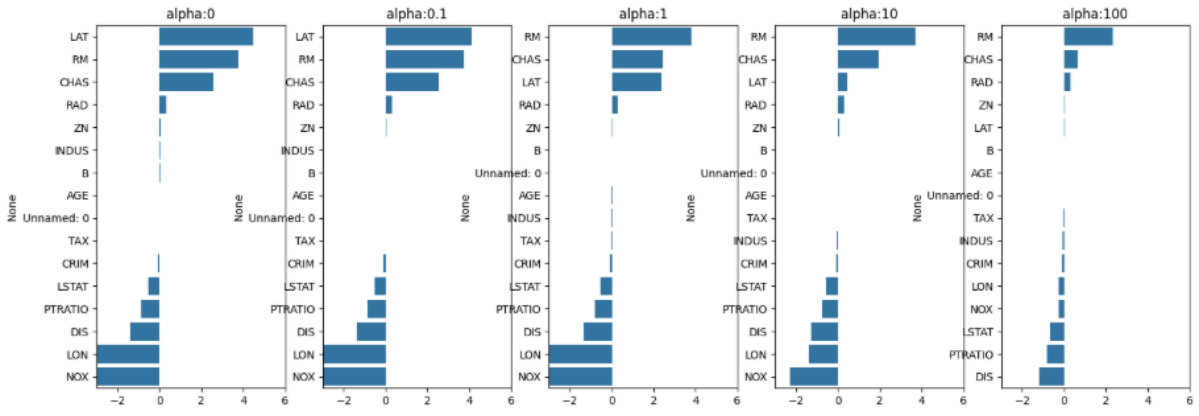
```
alphas = [0, 0.1, 1, 10, 100]

for alpha in alphas:

  ridge = Ridge(alpha = alpha)

  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('alpha: {0} 5 folds의 평균 RMSE: {1:.3f} '.format(alpha, avg_rmse))
```

```
alpha: 0 5 folds의 평균 RMSE: 6.091
alpha: 0.1 5 folds의 평균 RMSE: 5.952
alpha: 1 5 folds의 평균 RMSE: 5.667
alpha: 10 5 folds의 평균 RMSE: 5.553
alpha: 100 5 folds의 평균 RMSE: 5.457
```

회귀 계수 값을 가로 막대 그래프로 시각화

```
#회귀 계수 값을 가로 막대 그래프로 시각화
#5개의 열
```

```
fig, axs = plt.subplots(figsize = (18, 6), nrows = 1, ncols = 5)
# 각 알파에 따른 회귀계수 저장하는 DF
coeff_df = pd.DataFrame()
#알파 리스트 값 차례 입력 -> 회귀 계수 값 시각화, 데이터 저장. pos는 axis의 위치 지정

for pos, alpha in enumerate(alphas):
  ridge = Ridge(alpha = alpha)
  ridge.fit(X_data, y_target)

  coeff = pd.Series(data = ridge.coef_, index=X_data.columns)
  col_name = 'alpha:' + str(alpha)
  coeff_df[col_name] = coeff

  coeff = coeff.sort_values(ascending = False)
  axs[pos].set_title(col_name)
  axs[pos].set_xlim(-3, 6)
  sns.barplot(x=coeff.values, y=coeff.index, ax = axs[pos])

plt.show()
```




```
col_sort = 'alpha:'  + str(0)
coeff_df.sort_values(by = col_sort, ascending = False)
```

	alpha:0	alpha:0.1	alpha:1	alpha:10	alpha:100
LAT	4.469808	4.103766	2.364394	0.443810	0.055374
RM	3.753865	3.761137	3.798301	3.681868	2.329743
CHAS	2.577968	2.558139	2.446222	1.930032	0.639979
RAD	0.306802	0.304895	0.295473	0.291323	0.324131
ZN	0.046573	0.046718	0.047819	0.051323	0.056322
INDUS	0.015226	0.010701	-0.012611	-0.041814	-0.049634
B	0.009177	0.009241	0.009576	0.009968	0.009331
AGE	0.002452	0.001535	-0.003771	-0.011287	0.000087
Unnamed: 0	0.000030	0.000107	0.001045	0.001016	0.001445

알파 증가 → 회귀 계수 감소

라쏘 회귀

- 비용 함수 식 최소화하는 W 찾기
- 불필요한 피처의 회귀 계수를 급격하게 감소시킴

`Lasso` 클래스

- 주요 생성 인자 `alpha`

`get_linear_reg_eval()`

- 회귀 모델의 이름, alpha 값들의 리스트, 피처 데이터 세트와 타깃 데이터 세트 입력받아
- 알파값에 따른 폴드 평균 RMSE 출력, 회귀 계수값을 DF로 반환

Python 함수에서 매개변수 뒤에 =None을 붙이면, 그 인자는 **입력 안 해도 에러가 안남**

```
from sklearn.linear_model import Lasso, ElasticNet

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_ra
tio = 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 folds의 평균 RMSE: {1:.3f}'.format(param, avg_rms
e))
    model.fit(X_data_n, y_target_n)
    if return_coeff:
      coeff = pd.Series(data=model.coef_, index=X_data_n.columns)
      colname = 'alpha:' + str(param)
      coeff_df[colname] = coeff
      coeff = coeff.sort_values(ascending=False)
    return coeff_df
```

```
lasso_alphas = [0.07, 0.1, 0.5, 1, 3]
coeff_lasso_df = get_linear_reg_eval('Lasso', params = lasso_alphas, X_dat
a_n = X_data, y_target_n = y_target)
```

```
###### Lasso #####
alpha 0.07일 때 5 folds의 평균 RMSE: 5.677
alpha 0.1일 때 5 folds의 평균 RMSE: 5.694
alpha 0.5일 때 5 folds의 평균 RMSE: 5.831
alpha 1일 때 5 folds의 평균 RMSE: 6.036
alpha 3일 때 5 folds의 평균 RMSE: 6.732
```

```
sort_column = 'alpha:' + str(lasso_alphas[0])
coeff_lasso_df.sort_values(by = sort_column, ascending = False)
```

	alpha:0.07	alpha:0.1	alpha:0.5	alpha:1	alpha:3
RM	3.790649	3.702960	2.486456	0.923414	0.000000
CHAS	1.455639	0.975605	0.000000	0.000000	0.000000
RAD	0.284398	0.287359	0.281888	0.258960	0.035355
ZN	0.051634	0.051719	0.051376	0.050203	0.036227
B	0.010191	0.010190	0.009397	0.008153	0.006386
NOX	-0.000000	-0.000000	-0.000000	-0.000000	0.000000
LON	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
LAT	0.000000	0.000000	0.000000	0.000000	0.000000
Unnamed: 0	-0.002031	-0.001923	-0.000838	0.000301	0.003112
AGE	-0.013748	-0.011976	0.002718	0.021160	0.045137
TAX	-0.014056	-0.014369	-0.015572	-0.015590	-0.009776
INDUS	-0.039042	-0.033552	-0.002228	-0.000000	-0.000000
CRIM	-0.097103	-0.096769	-0.081822	-0.061626	-0.000000
LSTAT	-0.567301	-0.575587	-0.662624	-0.766662	-0.807245
PTRATIO	-0.734773	-0.740418	-0.729665	-0.694510	-0.236730
DIS	-1.220082	-1.203144	-0.970233	-0.695753	-0.000000

불필요한 회귀 계수 0됨

엘라스틱넷 회귀

- L2 + L1

$$\text{RSS}(W) + \text{alpha2} * \|W\|_2^2 + \text{alpha1} * \|W\|_1$$

이 값을 최소화하는 W 찾기

- 라쏘 회귀가 서로 상관관계가 높은 피처들의 경우 → 중요 피처만 셀렉하는 경향이 강함
- alpha값에 따라 회귀 계수의 값이 급격히 변동 가능 → 엘라스틱넷이 완화
- 수행시간이 상대적으로 오래 걸리는 단점

ElasticNet 클래스

- 규제: a * LI + b * L2

- `alpha` : a+b
- `l1_ratio` : a / a + b
  - 0 → L2 규제
  - 1 → L1 규제

```
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)
```

```
###### ElasticNet #####
alpha 0.07일 때 5 folds의 평균 RMSE: 5.611
alpha 0.1일 때 5 folds의 평균 RMSE: 5.606
alpha 0.5일 때 5 folds의 평균 RMSE: 5.645
alpha 1일 때 5 folds의 평균 RMSE: 5.857
alpha 3일 때 5 folds의 평균 RMSE: 6.567
```

	alpha:0.07	alpha:0.1	alpha:0.5	alpha:1	alpha:3
RM	3.574931	3.413639	1.910870	0.927486	0.000000
CHAS	1.346035	0.992987	0.000000	0.000000	0.000000
RAD	0.291906	0.295710	0.304775	0.285479	0.125170
ZN	0.052622	0.053046	0.054488	0.052946	0.037334
B	0.010066	0.010008	0.009047	0.008239	0.006941
LON	-0.000000	-0.000000	-0.000000	-0.000000	-0.000000
LAT	0.000000	0.000000	0.000000	0.000000	0.000000
Unnamed: 0	-0.001977	-0.001866	-0.000779	0.000096	0.002524
AGE	-0.012107	-0.010130	0.006992	0.020477	0.045547
TAX	-0.014306	-0.014630	-0.016185	-0.016499	-0.012445
INDUS	-0.041792	-0.039520	-0.019845	-0.000000	-0.000000

0이 되는 값이 더 적음

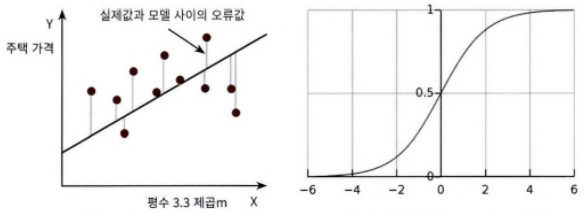
→ 뭐가 제일 좋은지는 상황에 따라 다름

→ 각각의 알고리즘에서 하이퍼파라미터 변경하면서 최적의 예측 성능을 찾아야 함

→ 선형 회귀인 경우 데이터 분포도 정규화화 인코딩 방법이 매우 중요

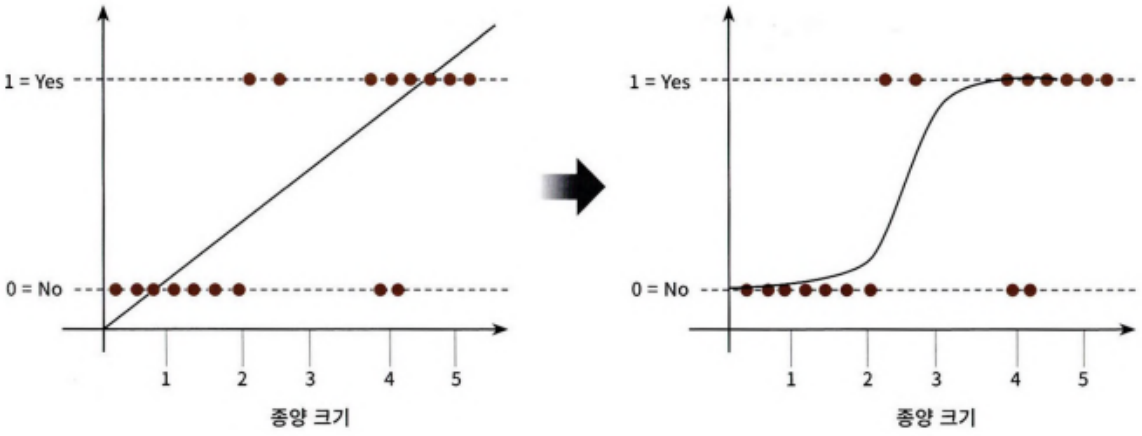
07 로지스틱 회귀(350-354p, 4p)

- 선형 회귀 방식을 분류에 적용
- 로지스틱 회귀가 선형 회귀와 다른 점
  - 학습을 통해 회귀 최적선 찾는것 X
  - 시그모이드(Sigmoid) 함수 최적선 → 함수의 반환 값을 확률로 간주해 확률에 따라 분류를 결정



〈 선형 회귀의 선형 함수 〉 〈 로지스틱 회귀의 시그모이드 함수 〉

$$y = \frac{1}{1+e^{-x}}$$



❤️ **로지스틱 회귀: 선형 회귀 방식을 기반으로 하되 시그모이드 함수를 이용해 분류를 수행**

- 정규 분포도에 따라 성능 영향 → 정규 분포 형태의 표준 스케일링을 적용 우선
- 최적화 파라미터

`LogisticRegression`클래스의 **회귀 계수 최적화**

- `solver` 파라미터
  - `lbfgs` : 사이킷런 버전 0.22부터 solver의 기본 설정값, 메모리 공간 절약 & 병렬 수행
  - `liblinear` : 사이킷런 버전 0.21까지에서 solver의 기본 설정값, 다차원 & 작은 데이터 세트에서 효과적, 국소 최적화(Local Minimum)에 이슈, 병렬 X
  - `newton-cg` : 더 정교한 최적화, 대용량 → 속도 저하
  - `sag` : Stochastic Average Gradient로서 경사 하강법 기반의 최적화, 대용량의 데이터 빠르게
  - `saga` : sag와 유사한 최적화 방식이며 L1 정규화를 가능
- 성능 차이는 미비 → `lbfgs` `liblinear`
  - 작은 데이터에서 `liblinear` 가 더 좋다고 항상 말할 수는 없음
  - 다만 더 빠르긴 함, 수행 성능 약간 나음

```
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
```

```
scaler = StandardScaler()
data_scaled = scaler.fit_transform(cancer.data)

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

```
from sklearn.metrics import accuracy_score, roc_auc_score

#로지스틱 회귀를 이용하여 학습 및 예측 수행
#solver 인자값을 생성사로 입력 X 기본은 lbfgs
lr_clf = LogisticRegression()
lr_clf.fit(X_train, y_train)
lr_preds = lr_clf.predict(X_test)
lr_preds_proba = lr_clf.predict_proba(X_test)[:, 1]

#정확도와 roc_auc 측정
print('accuracy: {0:.3f}, roc_auc: {1:.3f}'.format(accuracy_score(y_test, lr_pr
eds),
                                          roc_auc_score(y_test, lr_preds_proba)))
```

```
accuracy: 0.977, roc_auc: 0.995
```

sovler 별로 다르게 해서 성능 측정

```
solvers = ['lbfgs', 'liblinear', 'newton-cg', 'sag', 'saga']

#sovler 값 별로 학습 후 성능 평가
for solver in solvers:
  lr_clf = LogisticRegression(solver = solver, max_iter = 600)
  lr_clf.fit(X_train, y_train)
  lr_preds = lr_clf.predict(X_test)
  lr_preds_proba = lr_clf.predict_proba(X_test)[:, 1]

  #accuracy roc_auc 측정
  print('solver:{0}, accuracy: {1:.3f}, roc_auc: {2:.3f}'.format(solver,
                                              accuracy_score(y_test, lr_preds),
                                              roc_auc_score(y_test, lr_preds_prob
a)))
```

```
solver:lbfgs, accuracy: 0.977, roc_auc: 0.995
solver:liblinear, accuracy: 0.982, roc_auc: 0.995
solver:newton-cg, accuracy: 0.977, roc_auc: 0.995
solver:sag, accuracy: 0.982, roc_auc: 0.995
solver:saga, accuracy: 0.982, roc_auc: 0.995
```

크게 의미 있는 결과 X

주요 하이퍼 파라미터

- `penalty` → 'l1', 'l2'
- `C` = 1/alpha → 작을수록 규제 강도가 큼
- `Liblinear`, `sag` → 규제 모두 가능, 나머지는 L2만 가능

```
from sklearn.model_selection import GridSearchCV

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

lr_clf = LogisticRegression()
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_p
arams_, grid_clf.best_score_))
```

```
최적 하이퍼 파라미터: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}, 최적 평균 정확
도: 0.979
/usr/local/lib/python3.12/dist-packages/sklearn/model_selection/_validatio
n.py:528: FitFailedWarning:
15 fits failed out of a total of 60.
The score on these train-test partitions for these parameters will be set to n
an.
```

```
If these failures are not expected, you can try to debug them by setting erro
r_score='raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
15 fits failed with the following error:
Traceback (most recent call last):
  File "/usr/local/lib/python3.12/dist-packages/sklearn/model_selection/_vali
dation.py", line 866, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.12/dist-packages/sklearn/base.py", line 1389, i
n wrapper
    return fit_method(estimator, *args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logisti
c.py", line 1193, in fit
    solver = _check_solver(self.solver, self.penalty, self.dual)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logisti
c.py", line 63, in _check_solver
    raise ValueError(
ValueError: Solver lbfgs supports only 'l2' or None penalties, got l1 penalty.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
/usr/local/lib/python3.12/dist-packages/sklearn/model_selection/_search.p
y:1108: UserWarning: One or more of the test scores are non-finite: [0.9648
5659 0.94555834 0.92261209        nan 0.97891024 0.97364708
 0.96131997        nan 0.97539218 0.97539218 0.96660169        nan
 0.97011974 0.96836536 0.96662025        nan 0.96661097 0.96661097
 0.96134781        nan]
  warnings.warn(
```

FitFailedWarning → lbfgs일 때 L1 규제 X 넣어서

08 회귀 트리(355-361p, 7p)

- 회귀 함수를 기반으로 하지 않고 결정 트리와 같이 트리를 기반으로 하는 회귀 방식

- 회귀를 위한 트리를 생성 → 회귀 예측
- **리프 노드에서 예측 결정 값을 만드는 과정에 차이**
  - 분류 트리가 특정 클래스 레이블 결정 X
  - 리프 노드에 속한 데이터 값의 평균값을 구해 회귀 예측값 계산
- 트리 CART(Classification And Regression Trees) → 분류, 회귀 전부 가능

알고리즘	회귀 Estimator 클래스	분류 Estimator 클래스
Decision Tree	DecisionTreeRegressor	DecisionTreeClassifier
Gradient Boosting	GradientBoostingRegressor	GradientBoostingClassifier
XGBoost	XGBRegressor	XGBClassifier
LightGBM	LGBMRegressor	LGBMClassifier

예측할 새 데이터가 들어온다면?

예를 들어 새 샘플이 `x = 5.5` 라면:

1️⃣ 트리가 루트에서부터 시작해서

- "x ≤ 3?" → 아니오
- "x ≤ 6?" → 예

→ Leaf 2 로 이동

2️⃣ 그 리프의 평균값 5.1이

그 데이터의 예측 결과 (ŷ) 를 5.1로 함✅

$\hat{y}(x = 5.5) = 5.1$

❤️ 머신 러닝의 회귀

- 회귀 계수를 기반으로 하는 **최적 회귀 함수**를 도출하는 것이 주요 목표

❤️ 선형 회귀

- 회귀 계수를 선형으로 결합하는 회귀 함수 구함
- 여기에 독립변수 입력해 결괏값을 예측

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

rf = RandomForestRegressor(random_state= 0, n_estimators = 1000)
neg_mse_scores = cross_val_score(rf, X_data, y_target, scoring = "neg_me
an_squared_error", cv = 5)
rmse_scores = np.sqrt(-1 * neg_mse_scores)
avg_rmse = np.mean(rmse_scores)

print('5 교차 검증의 개별 negative MSE scores: ', np.round(neg_mse_scores,
2))
print('5 교차 검증의 개별 RMSE scores : ', np.round(rmse_scores, 2))
print('5 교차 검증의 평균 RMSE : {0:.3f} '.format(avg_rmse))
```

`get_model_cv_prediction( )` 입력 모델과 데이터 세트를 입력받아 교차 검증 → 평균 RMSE

```
def get_model_cv_prediction(model, X_data, y_target):
  neg_mse_scores = cross_val_score(model, X_data, y_target, scoring = "ne
g_mean_squared_error", cv = 5)
  rmse_scores = np.sqrt(-1 * neg_mse_scores)
  avg_rmse = np.mean(rmse_scores)
  print('#####', model.__class__.__name__, '#####')
  print('5 교차 검증의 평균 RMSE: {0:.3f}'.format(avg_rmse))
```

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import GradientBoostingRegressor
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
import warnings
warnings.filterwarnings('ignore')
```

```
dt_reg = DecisionTreeRegressor(random_state = 0, max_depth = 4)
rf_reg = RandomForestRegressor(random_state = 0, n_estimators = 1000)
gb_reg = GradientBoostingRegressor(random_state = 0, n_estimators = 100
0)
xgb_reg = XGBRegressor(n_estimators = 1000)
lgb_reg = LGBMRegressor(n_estimators = 1000, verbose=-1)

# 트리 기반의 회귀 모델을 반복하면서 평가 수행
models = [dt_reg, rf_reg, gb_reg, xgb_reg, lgb_reg]
for model in models:
  get_model_cv_prediction(model, X_data, y_target)
```

```
##### DecisionTreeRegressor #####
5 교차 검증의 평균 RMSE: 6.303
##### RandomForestRegressor #####
5 교차 검증의 평균 RMSE: 4.336
##### GradientBoostingRegressor #####
5 교차 검증의 평균 RMSE: 4.050
##### XGBRegressor #####
5 교차 검증의 평균 RMSE: 4.682
##### LGBMRegressor #####
5 교차 검증의 평균 RMSE: 4.517
```

💡 **GUI = 사용자가 마우스·화면으로 조작할 수 있는 그래픽 기반 인터페이스**

💡 `%matplotlib inline` = "그래프를 새 창으로 안 띄우고, 노트북 셀 안에 바로 그려줘!"

LightGBM → JSON 문자에 따라서 피처명 지정해야 함. 특정 특수 문자 처리 X

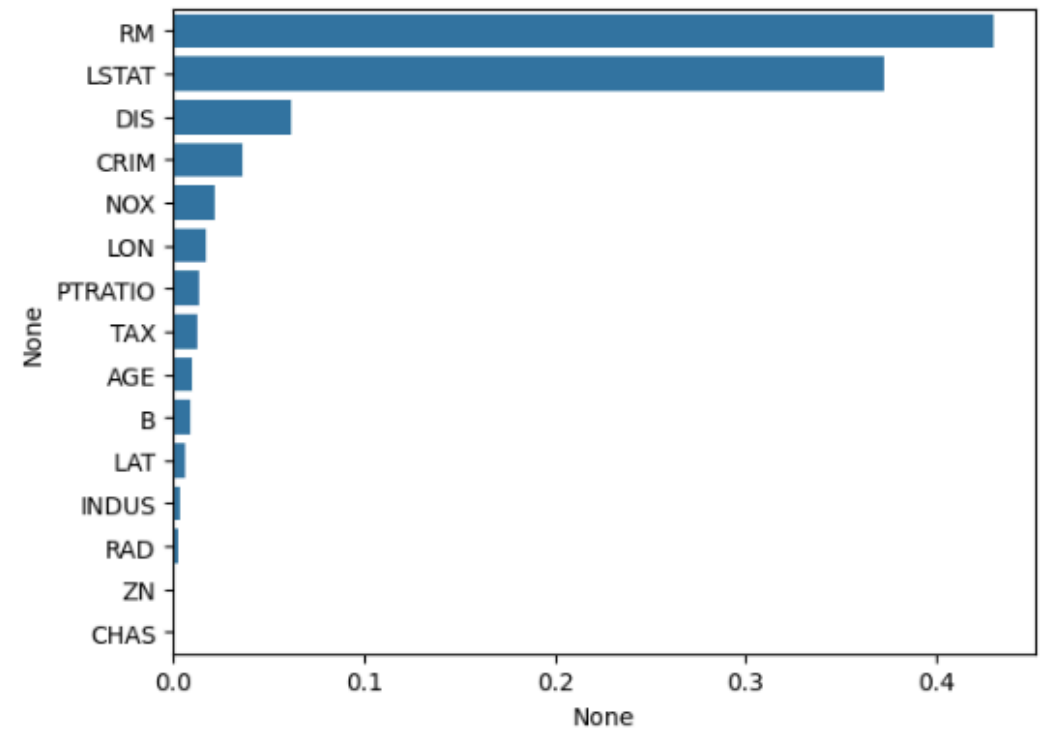
`warnings.filterwarnings('ignore')`

- **파이썬의 "warnings 모듈"에서 나오는 '경고 메시지'만 막는 기능**
- LightGBM 경고 → **파이썬 경고가 아니라 C++ 쪽에서 직접 출력하는 'stdout 로그'**

```
import seaborn as sns
%matplotlib inline

rf_reg = RandomForestRegressor(n_estimators = 1000)
rf_reg.fit(X_data, y_target)
```

```
feature_series = pd.Series(data = rf_reg.feature_importances_, index = X_data.columns)
feature_series = feature_series.sort_values(ascending = False)
sns.barplot(x = feature_series, y = feature_series.index)
```

```
<Axes: xlabel='None', ylabel='None'>
```

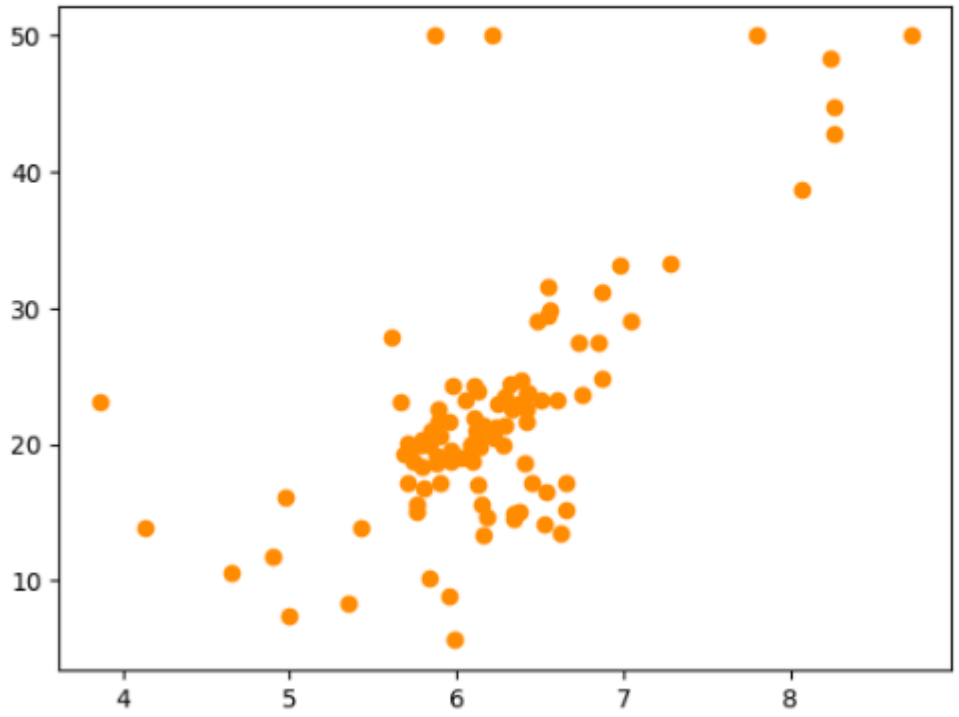


회귀 트리 Regressor가 어떻게 예측값을 판단하는지 선형 회귀와 비교해 시각화

보스턴 데이터 세트의 개수를 100개만 샘플링하고 RM과 PRICE 칼럼만 추출

```
from matplotlib import pyplot as plt
boston_sample = boston[['RM', 'PRICE']]
boston_sample = boston.sample(n=100, random_state = 0)
print(boston_sample.shape)
plt.figure()
plt.scatter(boston_sample['RM'], boston_sample['PRICE'], c='darkorange')
```

```
(100, 16)
<matplotlib.collections.PathCollection at 0x7fc927914a40>
```




```
import numpy as np
from sklearn.linear_model import LinearRegression

#선형 회귀와 결정 트리 기반의 Regressor 생성, 결정트리 최대 깊이는 각각 2, 7
lr_reg = LinearRegression()
rf_reg2= DecisionTreeRegressor(max_depth = 2)
rf_reg7 = DecisionTreeRegressor(max_depth = 7)

# 테스터용 데이터 세트 4.5~8.5 100개 데이터
X_test = np.arange(4.5, 8.5, 0.04).reshape(-1, 1)

X_feature = boston[['RM']].values.reshape(-1, 1)
y_target = boston['PRICE'].values.reshape(-1, 1)

lr_reg.fit(X_feature, y_target)
rf_reg2.fit(X_feature, y_target)
rf_reg7.fit(X_feature, y_target)

pred_lr = lr_reg.predict(X_test)
```

```
pred_rf2 = rf_reg2.predict(X_test)
pred_rf7 = rf_reg7.predict(X_test)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(figsize=(14, 4), ncols =3)

#X축 값을 4.5~8.5 -> 선형회귀, 결정트리 회귀(2, 7) 예측선 시각화

#선형회귀
ax1.set_title('Linear Regression')
ax1.scatter(boston_sample['RM'], boston_sample.PRICE, c='darkorange')
ax1.plot(X_test, pred_lr, label='linear', linewidth=2)

#결정트리 max_depth = 2
ax2.set_title('Decision Tree Regression: \n max_depth=2')
ax2.scatter(boston_sample['RM'], boston_sample.PRICE, c='darkorange')
ax2.plot(X_test, pred_rf2, label='max_depth:2', linewidth=2)

#결정트리 max_depth = 7
ax3.set_title('Decision Tree Regression: \n max_depth=7')
ax3.scatter(boston_sample['RM'], boston_sample.PRICE, c='darkorange')
ax3.plot(X_test, pred_rf7, label='max_depth:7', linewidth=2)
```

```
[<matplotlib.lines.Line2D at 0x7fc925d9c080>]
```

❤️ 선형 회귀는 직선으로 예측 회귀선을 표현
회귀 트리의 경우 분할되는 데이터 지점 → 브랜치를 만들면서 계단 형태로 회귀선