



Aichemist Session

CHAP 05 회귀

CONTENTS

회귀

1. 회귀
2. 단순 선형 회귀를 통한 회귀 이해
3. 다행 회귀와 과(대)적합 / 과소적합 이해
4. 규제 선형 회귀 모델 - 릿지, 라쏘, 엘라스틱넷
5. 로지스틱 회귀
6. 회귀 트리

01. 회귀



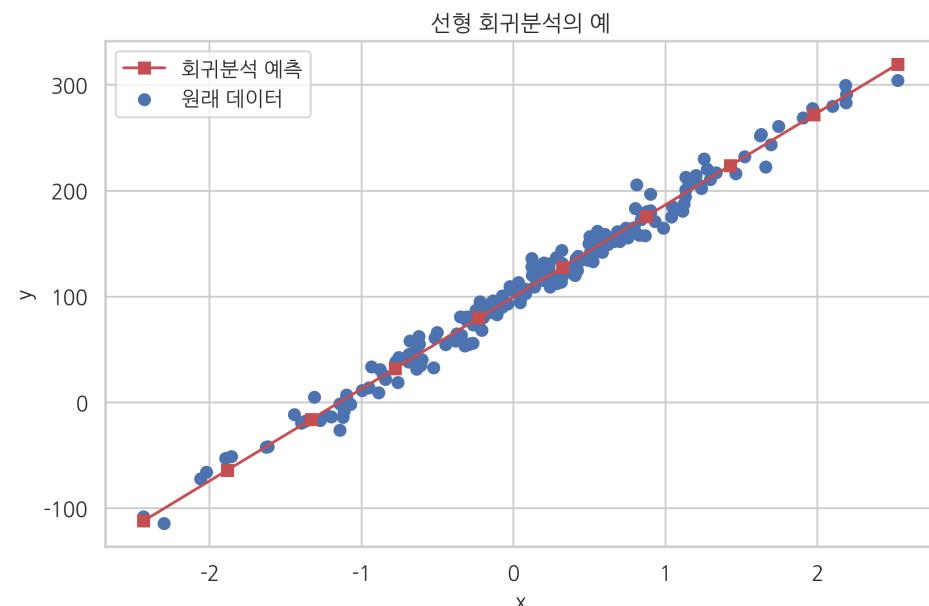
회귀 소개

1. 회귀

- 데이터 값이 **평균**으로 돌아가려는 경향 이용
- 여러 개의 독립 변수와 한 개의 종속 변수간의 상관관계를 모델링하는 기법

독립 변수 -> 예측값에 영향을 미치는 조건들, 피처 / 종속 변수 -> 예측값

회귀 계수 -> 각 독립변수의 종속 변수에 대한 영향력을 나타내는 계수



회귀 소개

2. 회귀 분석

- 모든 데이터는 아래와 같은 수식으로 표현할 수 있다고 가정

$$y = h(x_1, x_2, x_3, \dots, x_k; \beta_1, \beta_2, \beta_3, \dots, \beta_k) + e$$

$h()$: 회귀 모델 / y : 데이터 / x_i : 데이터에 영향을 미치는 조건들 / β_i : 각 조건들의 영향력 / e : 오차항

- 회귀 모델 $h()$ 는 어떠한 조건이 주어지면 각 조건의 영향력을 고려하여 해당 조건에서의 평균을 계산해주는 것
- 이때 e (오차항)은 현실적인 한계로 인해 발생하는 오차항을 나타냄. 이는 잡음이라고도 함

- 회귀 분석이란 결국 조건을 선택하고 조건의 영향력을 조정해가면서 최적의 $h()$ 를 찾아내는 과정
- $h()$ 를 최적화하는 방법 : 오차값의 최소화 / 과(대)적합, 과소적합 고려 -> 규제

회귀 소개

3. 회귀 유형

독립 변수	회귀 계수의 결합
1개 : 단일 회귀	선형 : 선형 회귀
여러개 : 다중 회귀	비선형 : 비선형 회귀

- 회귀 모델의 선형성을 따지는 기준이 독립변수나 종속변수의 관계가 아니라

정규화 그대로

회귀의 결합방식

이라는 것을 기억해야 한다

3-1. 회귀 모델의 선형성

선형성 판별 방법

- $f(c \times u) = c \times f(u)$ (여기서 c 는 상수)
 $f(u + v) = f(u) + f(v)$

어떠한 식이 두 조건을 모두 만족할 때 선형성을 뛴다고 한다

선형 회귀 함수식 예시

$$w_0 = y$$

$$w_0 + w_1 x_1 = y$$

$$w_1 x_1 + w_2 x_2 + \dots + w_n x_n = y$$

$$w_1 x_1 + w_2 x_1^2 + \dots + w_n x_1^n = y$$

$$w_1 x_1 + w_2 x_1 x_2 + \dots + w_n x_1^n = y$$

회귀 소개

4. 회귀 vs. 분류

- 회귀와 분류의 공통점

* **기계학습** : 정답이 있는 데이터를 활용해 모델을 학습시키는 방법

- 회귀와 분류의 차이점

회귀 (Regression)

예측하고자 하는 값이 **실수(숫자)**인 경우

예측 결과 : **연속값**

ex) 손해액, 매출량, 거래량, 파산 확률 등 예측

분류 (Classification)

예측하고자 하는 값이 **범주형 변수**인 경우

예측 결과 : **이산값**

ex) 이진 분류 / 다중 분류

02. 단순 선형 회귀를 통한 회귀 이해

단순 선형 회귀

단순 선형 회귀

단순 선형 회귀

- 독립 변수도 1개, 종속변수도 1개인 선형 회귀

- 즉 **하나의 피처**만 고려해서 결괏값 예측

- 실제값 : $Y_i = w_0 + w_1 * X_i + \epsilon_i$

- 예측값 : $f(x) = w_0 + w_1 * X$ → 단순 선형 회귀 모델

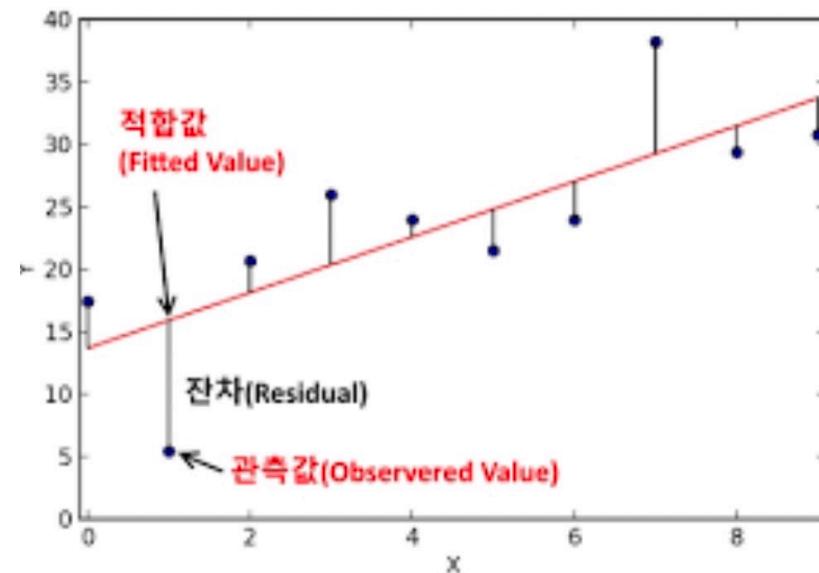
* 회귀 계수 : 기울기(w_1) / 절편(w_0)

* 잔차(e_i) : 실제값과 회귀 모델의 예측값간 차이, **오류값**

=> 최적의 회귀 모델 : **잔차** 가 최소가 되는 모델

~> **RSS 방식** 통해 오류 합 계산

$$e_i = y_i - f(x)$$



02. 단순 선형 회귀를 통한 회귀 이해

RSS와 경사 하강법 / OLS

RSS (Residual Sum of Square)

앞서 말했듯이 회귀 함수를 최적화하는 방법은 오류값을 최소화하고 과(대)적합, 과소적합에 대비하는 것이다
그중에서 우선 [오류값을 최소화하는 방법](#)에 대해 알아보자



RSS식으로 오류값의 합 구하기

$$RSS = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

단순 선형 회귀 함수식 대입

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

y_i : 실제값, \hat{y}_i : 예측값 / $y_i - \hat{y}_i$: 오류값(잔차)

w_0, w_1 : 회귀 계수 / y_i : 실제 값 / $w_0 + w_1 * x_i$: 단순 선형 회귀 함수식

이때 RSS에서 중심 변수는 회귀 계수(w_i)다!

- 오류값은 + 나 - 가 될 수 있기 때문에 오류합은 각각의 오류값에 [제곱](#) 을 구해서 더하는 방식을 사용한다

- 오류합을 최소화하는 회귀 계수를 구하는 방식으로는 [경사하강법](#)과 [OLS\(Ordinary Least Square\)](#)가 사용된다

- RSS를 응용하여 [회귀함수의 성능을 평가](#)할 수 있다

경사하강법

경사 하강법(Gradient Descent)

- RSS의 값이 작아지는 방향성으로 W (회귀계수)를 계속 업데이트하는 방식
- 반환값이 최솟값을 가지는 지점에서 W 반환

RSS의 값이 작아지는 방향성이란?

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

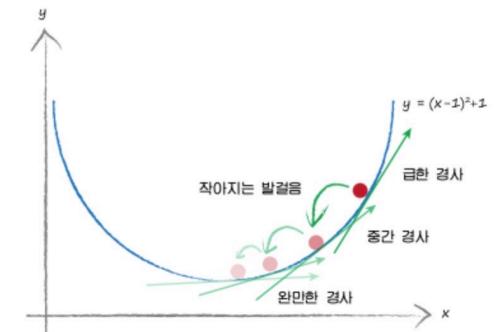
편미분 \rightarrow

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

위의 RSS식(2차 함수식)을 회귀 계수에 대해 편미분하면

RSS 함수의 기울기 함수가 나오고, 기울기의 경사가 완만해지는 방향성이 RSS의 값이 작아지는 방향성이다

경사가 완만해지는 방향을 따라간다면 RSS 함수의 극솟값에 이를 것이다



경사하강법

경사 하강법(Gradient Descent)

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

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

RSS식을 회귀 계수들에 대해 편미분하면 위와 같은 결과를 얻게 되는데

기울기의 경사가 완만해지는 방향으로 회귀 계수들을 업데이트하면 된다

W(회귀 계수)의 업데이트 방식?

$$\theta_i := \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\theta_i)$$

학습률
↓
기울기(편미분값)

W의 업데이트는 왼쪽의 식을 사용하는데

기울기(편미분값)가 음수인 경우 W값은 커지고

기울기(편미분값)가 양수인 경우 W값은 작아지면서

W값은 극값에 더 가까워질 수 있게 되는 것이다

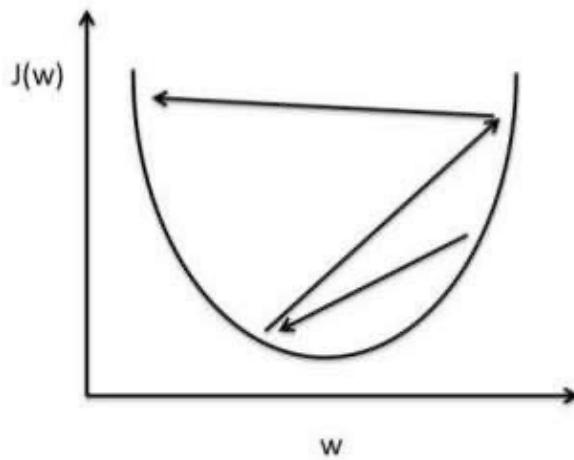
θ는 W값을 의미한다

* 편미분값 앞에 붙어있는 α 는 학습률로 편미분값이 너무 크거나 작은 경우 값을 보정해주기 위해 존재한다

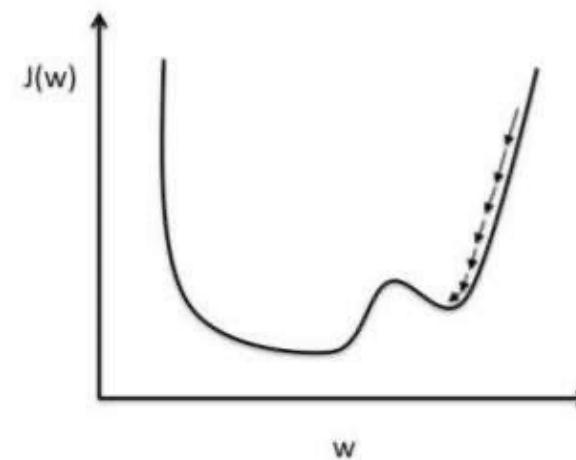
학습률

학습률

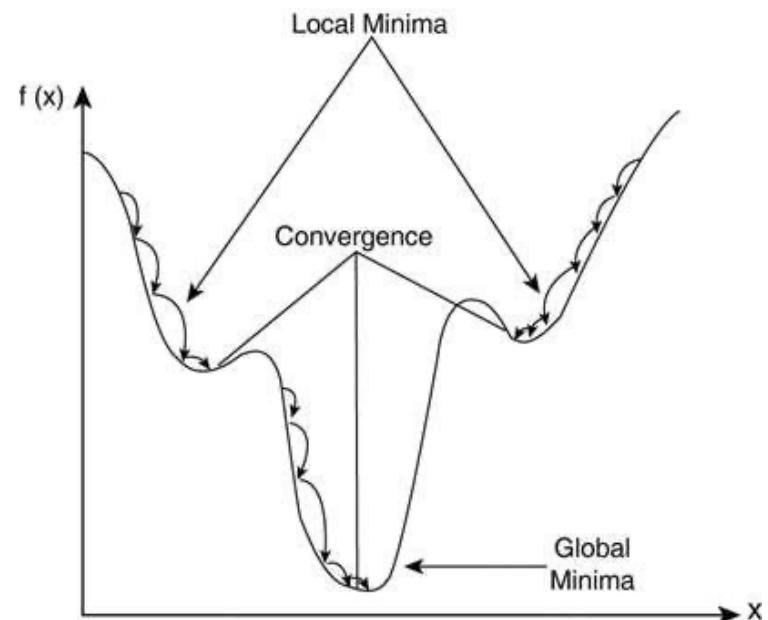
- 학습률이 너무 크면 한 지점으로 수렴하는 것이 아니라  할 가능성 존재 -> overshooting
- 학습률이 너무 작으면 수렴이  -> learn too slow
- 시작점을 어디로 잡느냐에 따라서도 수렴 지점이 달라짐



Overshooting



Learn too slow



경사하강법 실습

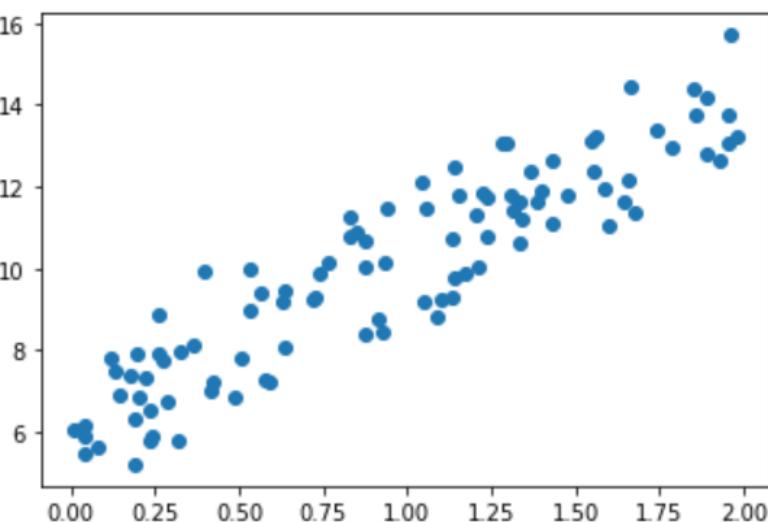
가상 데이터세트 생성 후 시각화

```
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)

plt.scatter(X,y)
```

<matplotlib.collections.PathCollection at 0x7f49a9bbc690>



RSS 함수(비용 함수) 정의

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

w1과 w0을 업데이트할 값을 반환

```
def get_weight_updates(w1,w0,X,y,learning_rate=0.01):
    N = len(y)
    w1_update = np.zeros_like(w1)
    w0_update = np.zeros_like(w0)
    y_pred = np.dot(X,w1.T)+w0
    diff = y - y_pred

    w0_factors = np.ones((N,1))

    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
```

경사하강법 실습

W값 업데이트 함수

```
# 경사하강법 함수
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
```

경사 하강법 예측 오류 계산

```
# 예측값과 실제값의 RSS 차이를 계산하는 함수 생성
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('경사하강 total cost : {0:.4f}'.format(get_cost(y,y_pred)))
```

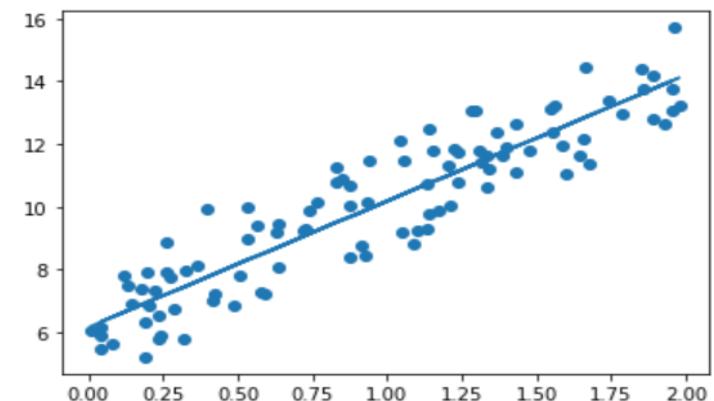
w1:4.022 w0:6.162

경사하강 total cost : 0.9935

추정 회귀선 시각화

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

[<matplotlib.lines.Line2D at 0x7f49a9621990>]



(미니배치) 확률적 경사하강법

확률적 경사하강법 (Stochastic Gradient Descent)

- 랜덤 샘플링을 통해 추출한 일부 데이터만을 이용해 w 업데이트 값을 계산하는 경사 하강법
- 경사 하강법보다 연산시간 단축

미니배치 확률적 경사하강법 (Mini-Batch Stochastic Gradient Descent)

- 전체 데이터를 batch_size개씩 나누어 배치로 학습

확률적 경사하강법의 구현은 앞선 경사하강법에서 batch_size 만큼 랜덤하게 데이터를 추출하는 로직만 추가됩니다.

자세한 구현 방법은 교재 p318-319를 통해 공부해보세요~~

OLS(Ordinary Least Squares)

OLS 방식

- RSS의 값을 **최소화** 하는 회귀계수를 통계학적으로 구하는 방식
- OLS 방식의 경우, **다중공선성 문제**를 고려해야 함
 - * 다중공선성 : 피처 간 상관관계가 매우 높은 경우 분산이 매우 커져서 오류에 민감해지는 것
 - ~> 해결 방법 : 피처 제거, 규제 적용, 차원 축소

OLS(Ordinary Least Squares)

구현 방법 - Linear Regression 클래스

- fit()을 이용해 X, y를 입력받으면 coef_ 속성에 회귀 계수가 배열 형태로 저장

입력 파라미터	<code>fit_intercept</code> : 절편 값 계산 여부 결정 <code>normalize</code> : 입력 데이터세트 정규화 여부
속성	<code>coef_</code> : fit() 메서드 수행했을 때 회귀 계수가 배열 형태로 저장되는 속성 <code>intercept_</code> : 절편 값

02. 단순 선형 회귀를 통한 회귀 이해

회귀 평가 지표

회귀 평가 지표

- RSS식을 응용한 회귀 평가 지표

평가지표	식	사이킷런 평가지표 API	Scoring 함수 적용 값
MAE	$MAE = \frac{1}{n} \sum_{i=1}^n Y_i - \hat{Y}_i $	metrics.mean_absolute_error	neg_mean_absolute_error
MSE	$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$	metrics.mean_squared_error	neg_mean_squared_error
RMSE	$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2}$	metrics.mean_squared_error (squared=False)	neg_root_mean_squared_error
MSLE	$MSLE = \frac{1}{n} \sum_{i=1}^n (\log(y_i + 1) - \log(\hat{y}_i + 1))^2$	metrics.mean_squared_log_error	neg_mean_squared_log_error
R2	$R^2 = \frac{predictVariance}{realVariance}$	metrics_r2_score	r2

$Y_i - \hat{Y}_i$: 실제값 - 예측값 = 잔차 (오류값)

잔차의 ~~절댓값~~의 평균값

잔차의 제곱의 평균값

잔차의 제곱의 평균에 ~~루트~~ 를 쓴 값

실제값과 예측값에 ~~R2~~ 를 취한 후 평균 제곱 오차를 계산한 값

실제값의 ~~분산~~ 대비 예측값의 ~~분산~~ 비율

회귀 평가 지표

- RSS식을 응용한 회귀 평가 지표

평가지표	식	사이킷런 평가지표 API	Scoring 함수 적용 값
MAE	$MAE = \frac{1}{n} \sum_{i=1}^n Y_i - \hat{Y}_i $	metrics.mean_absolute_error	neg_mean_absolute_error
MSE	$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$	metrics.mean_squared_error	neg_mean_squared_error
RMSE	$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2}$	metrics.mean_squared_error (squared=False)	neg_root_mean_squared_error
MSLE	$MSLE = \frac{1}{n} \sum_{i=1}^n (\log(Y_i + 1) - \log(\hat{Y}_i + 1))^2$	metrics.mean_squared_log_error	neg_mean_squared_log_error
R2	$R^2 = \frac{predictVariance}{realVariance}$	metrics_r2_score	r2

neg의 의미

- Scoring 함수는 score값이 클수록 좋은 평가 결과 나타남
- 회귀 평가 지표는 나쁜 모델일수록 값이 커짐
- 회귀 평가 지표에 의한 평가 결과에 **-1**을 곱해줘서 값이 작을수록 좋은 모델, 값이 클수록 나쁜 모델로 평가받을 수 있도록 보정해줌



02. 단순 선형 회귀를 통한 회귀 이해

보스턴 주택 가격 예측 회귀 구현 - Linear Regression 이용

다중 선형 회귀

다중 선형 회귀

- 독립 변수가 여러개 종속변수가 개인 선형 회귀
- 즉, 여러개의 피처를 고려해서 결괏값 예측
- 예측 회귀식 : $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p$

Y : 종속 변수, 예측값 / X_i : 독립 변수, 피처 / β_i : 회귀 계수

피처가 p개인 경우 회귀 계수 (p+1)개 도출

회귀 계수 도출 방법

- 단순 선형 회귀와 동일하게 오류합을 최소화하는 방향으로 회귀 계수를 도출함
- 선형대수 행렬 내적 활용

$$Y = np.dot(X_{\text{mat}}, W^T)$$

Y : 종속변수(예측값) 행렬, X_{mat} : 독립변수(피처) 행렬, W^T : 회귀 계수의 전치 행렬

이때, W^T 는 피처가 p개인 경우 $(p+1)*1$ 의 형태이며, X_{mat} 또한 $(p+1) * n$ 의 형태를 띤다

보스턴 주택 가격 예측 회귀 구현

보스톤 주택 가격 예측 데이터세트는 현재 윤리적인 문제로 인해 사용이 불가합니다. 교재의 코드를 분석해보는 걸로 만족합시다 😢

데이터 로드

```
# 데이터셋 로드, DataFrame 변경
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
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline

boston = load_boston()
bostonDF = pd.DataFrame(boston.data, columns = boston.feature_names)
bostonDF['PRICE'] = boston.target
print('Boston 데이터셋 크기 : ',bostonDF.shape)
bostonDF.head()
```

Boston 데이터셋 크기 : (506, 14)

칼럼별 영향도 확인

```
# 각 칼럼이 회귀 결과에 미치는 영향 시각화
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
    sns.regplot(x=feature, y='PRICE', data=bostonDF, ax=axs[row][col])

# RM : 양의 상관관계
# LSTAT : 음의 상관관계
```

보스턴 주택 가격 예측 회귀 구현

회귀 모델 학습 / 예측 / 평가

```
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)

# 선형 회귀 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: {:.3f}, RMSE: {:.3f}'.format(mse, rmse))
print('Variance score : {:.3f}'.format(r2_score(y_test, y_preds)))

MSE:17.297, RMSE:4.159
Variance score : 0.757

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

절편 값 : 40.99559517216477
회귀 계수 값 : [-0.1  0.1  0.   3. -19.8  3.4  0. -1.7  0.4 -0. -0.9  0.
 -0.6]
```

피처별 회귀 계수 값 매핑

```
# 피처별 회귀계수 값으로 매핑, 높은 값 순으로 출력
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

보스턴 주택 가격 예측 회귀 구현

교차검증을 통한 MSE, RMSE 계산

```
# 5개의 폴드 세트에서 cross_val_score() 교차 검증으로 MSE, 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()

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)

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 : [-12.46 -26.05 -33.07 -80.76 -33.31]

5 folds의 개별 RMSE scores : [3.53 5.1 5.75 8.99 5.77]

5 folds의 평균 RMSE : 5.829

03. 다향 회귀와 과(대)적합 / 과소적합 이해

다향 회귀 이해

다항회귀

다항회귀

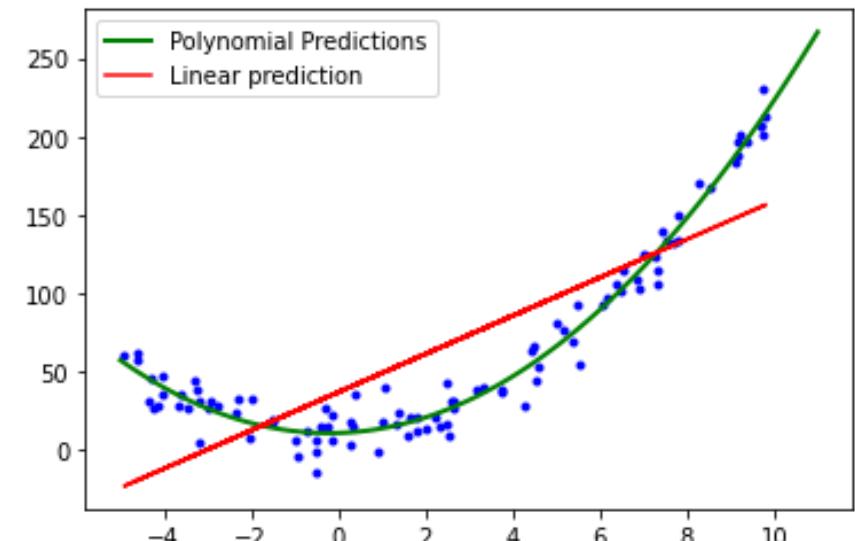
- 회귀가 독립변수의 단항식이 아닌 2차, 3차 방정식과 같은 **다항식**으로 표현되는 것
- 즉, 회귀 함수가 직선형이 아닌 **곡선형**으로 나타남
- 다항 회귀 역시 **선형회귀**

선형/비선형 회귀를 나누는 기준이 **차수** 가 선형/비선형인지에 따른 것이기 때문

(독립변수의 선형 / 비선형 여부와는 무관)

- 다항 회귀 함수식 : $Y = a_0 + a_1X_1 + a_2X_1^2$

Y: 종속 변수, 예측값 / X_i : 독립 변수, 피처 / a_i : 회귀계수



다항회귀 구현 - PolynomialFeature

PolynomialFeatures

- 사이킷런에서 다항식 피처로 변환하는 클래스
- 입력받은 단항식 피처를 `degree`에 해당하는 다항식 피처로 변환
- `fit()`, `transform()` 메서드를 통해 변환 작업 수행

`degree = 2`

$[x_1, x_2] \rightarrow [1, x_1, x_2, x_1^2, x_1x_2, x_2^2]$

```
1 # PolynomialFeatures 사용
2
3 from sklearn.preprocessing import PolynomialFeatures
4 import numpy as np
5
6 # 다항식으로 변환할 단항식 생성. [[0, 1], [2, 3]]의 2x2 행렬 생성
7 X = np.arange(4).reshape(2,2)
8 print('일차 단항식 계수 피처:\n', X)
9
10 # degree=2인 2차 다항식으로 변환하기 위해 PolynomialFeatures를 이용해 변환
11 poly = PolynomialFeatures(degree=2)
12 poly.fit(X)
13 poly_ftr = poly.transform(X)
14 print('변환된 2차 다항식 계수 피처:\n', poly_ftr)
```

일차 단항식 계수 피처:

```
[[0 1]
 [2 3]]
```

변환된 2차 다항식 계수 피처:

```
[[1.  0.  1.  0.  0.  1.]
 [1.  2.  3.  4.  6.  9.]]
```

`degree = 3`

$[x_1, x_2] \rightarrow [1, x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, x_1x_2^2, x_1^2x_2, x_2^3]$

```
1 # 3차 다항식 변환
2 poly_ftr = PolynomialFeatures(degree=3).fit_transform(X)
3 print('3차 다항식 계수 feature:\n', poly_ftr)
4
5 # Linear Regression에 3차 다항식 계수 feature와 3차 다항식 결정값으로 학습 후 회귀 계수 확인
6 model = LinearRegression()
7 model.fit(poly_ftr, y)
8 print('Polynomial 회귀 계수:\n', np.round(model.coef_, 2))
9 print('Polynomial 회귀 shape :', model.coef_.shape)
10
```

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,)

03. 다향 회귀와 과(대)적합 / 과소적합 이해

다향 회귀를 이용한 과소적합 및 과적합 이해, 편향-분산 트레이드오프

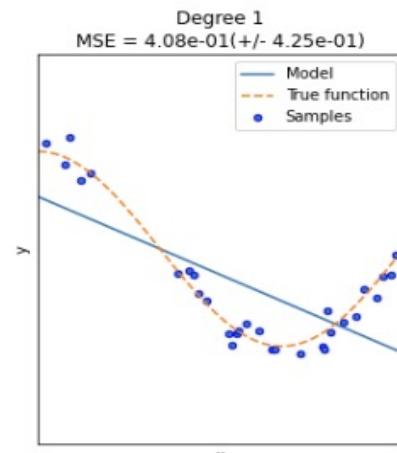
과소적합 / 과대적합

다항 회귀는 단순 선형 회귀나 다중 회귀보다 더 복잡한 피처 간의 관계를 모델링할 수 있다

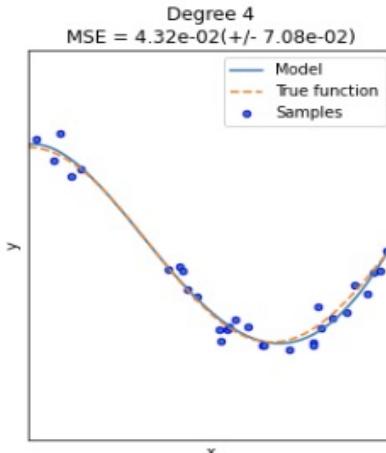
다항 회귀의 차수를 높일수록 매우 복잡한 피처 관계까지 표현할 수 있다

하지만 다항 회귀의 차수가 높아질수록 학습 데이터에만 최적화된 **과대적합**이 발생한다

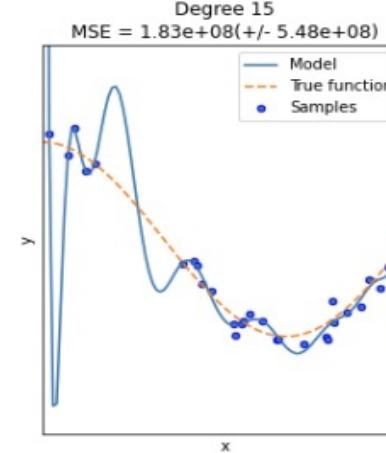
또한, 다항 회귀의 차수가 낮을수록 학습 데이터의 패턴을 반영하지 못하는 **과소적합**이 발생한다



과소적합



Balanced 모델



과대적합

차수 낮음



- 단순 선형 회귀와 거의 동일한 형태
- 학습 데이터의 패턴 반영 거의 안 됨

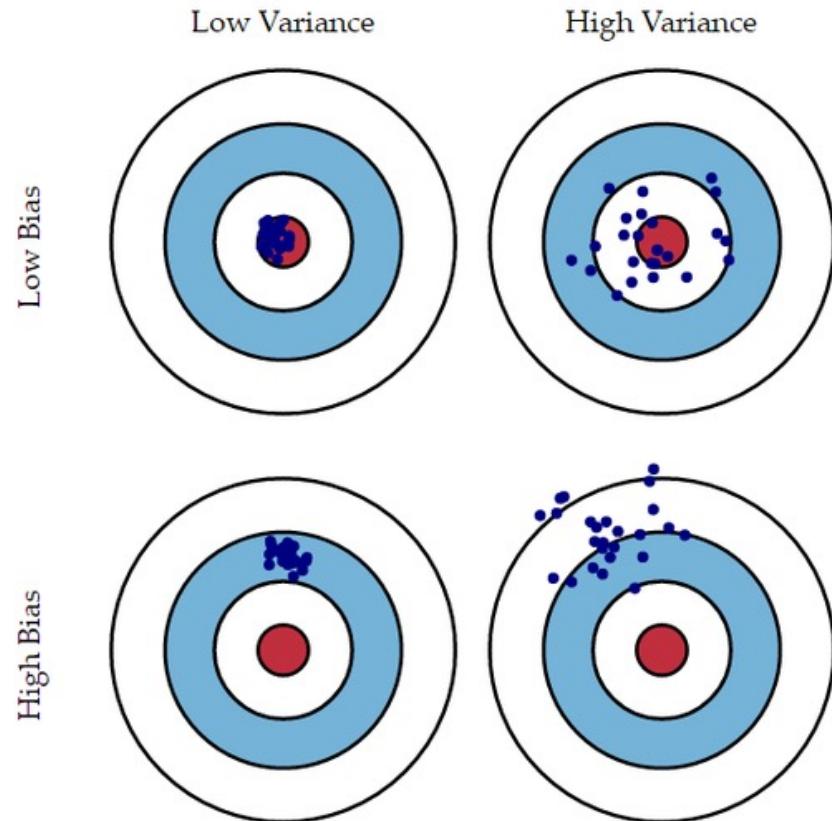
차수 높음



- 실제 데이터 세트와 가장 유사

- 학습 데이터에 너무 최적화
- 높은 MSE 값

편향 / 분산



편향

- 예측값과 실제값 간의 차이
- 지나치게 단순한 모델로 인한 error
- 편향이 높으면 과소평가될 가능성이 큼

분산

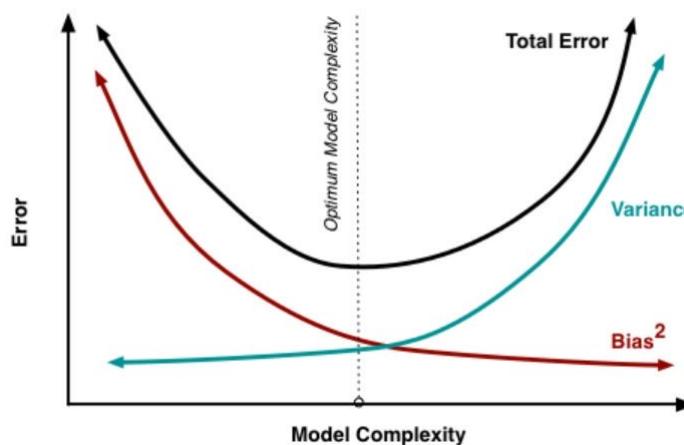
- 예측값 끝의 차이
- 지나치게 복잡한 모델로 인한 error
- 분산이 크면 과소평가될 가능성이 큼

편향 - 분산 트레이드오프

편향-분산 트레이드오프

- 모델의 복잡도가 커지면 분산이 (\uparrow / \downarrow) 편향은 (\uparrow / \downarrow)
- 모델의 복잡도가 줄어들면 편향이 (\uparrow / \downarrow) 분산은 (\uparrow / \downarrow)
- 이렇게 한쪽이 높으면 한쪽이 낮아지는 편향과 분산의 관계를 두고

편향-분산 트레이드오프라고 함



편향과 분산이 서로 트레이드오프를 이루면서 오류 값이 최대로 낮아지는
‘골디락스 지점’의 모델을 구축해야 한다

04. 규제 선형 모델

규제 선형 모델의 개요

규제

앞서 나온 RSS 비용함수와 OLS 방식은 예측값과 실제값의 잔차 합 즉, **오류합을 최소화하는 방법**이었다.

하지만 오류합의 최소화만 따지다 보니 훈련 데이터에 과하게 최적화된 (= **과대적합** = 고분산)

복잡한 회귀 함수식이 만들어졌고 (= 회귀 다항식의 차수가 높아졌고)

이에 따라 **회귀 계수가 과도하게 커지는 문제**가 발생했다.

이번에 다룰 **규제**는 **오류합의 최소화**와 **회귀계수 크기 제어**를 모두 고려하는 방법이다.

이 두 가지가 균형을 잘 이룰수록 좋은 회귀식이 만들어진다.



규제

비용 함수 목표

$$Y = \text{Min} (\text{RSS}(W) + a * \|W\|)$$

- a 가 0 (또는 매우 작은 값)일 때 비용 함수는 기존의 오차합의 최소화만 고려한 $\text{Min}(\text{RSS}(W))$ 가 된다 $Y = \text{Min} (\text{RSS}(W) + a * \|W\|)$
- a 가 무한대 (또는 매우 큰 값)일 때 비용 함수에서 $\text{RSS}(W)$ 의 값을 최대한 작게 만들기 위해
 W 를 0 (또는 매우 작은 값)으로 설정하게 된다 $Y = \text{Min} (\text{RSS}(W) + a * \|W\|)$

규제는 a 값을 이용해 학습 데이터의 합성과 회귀 계수 값의 크기 제어를 모두 고려한다

$\|W\|$ 을 회귀 계수의 절댓값으로 설정할지, 제곱으로 설정할지에 따라서 각각 L1 규제와 L2 규제로 나뉜다

04. 규제 선형 모델

릿지 회귀

릿지 회귀

릿지 회귀

- 선형회귀에 L_2 규제를 추가한 회귀 모델

릿지회귀 비용함수

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

- L2 규제

W 제곱에 대해 패널티를 부여하여 회귀 계수를 작게 만드는 규제

- Ridge

사이킷런에서 릿지 회귀를 구현하는 클래스

주요 생성 파라미터 : alpha (릿지 회귀의 L2 규제 계수)

릿지 회귀

$a = 10$ 일 때 릿지 회귀의 교차 검증 결과

```
# 앞의 LinearRegression 예제에서 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 데이터셋 로드
boston = load_boston()

# boston 데이터셋 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

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

ridge = Ridge(alpha = 10) ➔ 릿지 회귀
neg_mse_scores = cross_val_score(ridge, X_data, y_target, scoring="neg_mean_squared_error", cv = 5) ➔ 릿지 회귀 평가
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 folds 의 개별 Negative MSE scores: [-11.422 -24.294 -28.144 -74.599 -28.517]
5 folds 의 개별 RMSE scores : [3.38 4.929 5.305 8.637 5.34]
5 folds 의 평균 RMSE : 5.518

릿지 회귀

α 값이 각각 0, 0.1, 1, 10, 100 일 때 교차 검증 결과

```
# Ridge에 사용될 alpha 파라미터의 값을 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))|
```

```
alpha 0 일 때 5 folds 의 평균 RMSE : 5.829
alpha 0.1 일 때 5 folds 의 평균 RMSE : 5.788
alpha 1 일 때 5 folds 의 평균 RMSE : 5.653
alpha 10 일 때 5 folds 의 평균 RMSE : 5.518
alpha 100 일 때 5 folds 의 평균 RMSE : 5.330
```

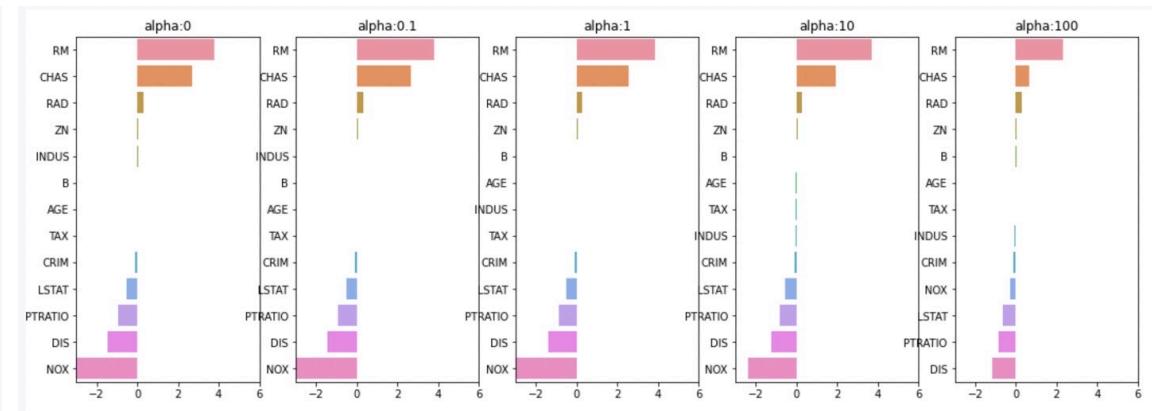
→ α 값이 100일 때 가장 성능이 좋게 나옴

→ 하지만 α 값의 크기와 성능이 항상 비례하진 않음

릿지 회귀

α 값이 각각 0, 0.1, 1, 10, 100 일 때 피처당 회귀 계수값

	alpha:0	alpha:0.1	alpha:1	alpha:10	alpha:100
RM	3.809865	3.818233	3.854000	3.702272	2.334536
CHAS	2.686734	2.670019	2.552393	1.952021	0.638335
RAD	0.306049	0.303515	0.290142	0.279596	0.315358
ZN	0.046420	0.046572	0.047443	0.049579	0.054496
INDUS	0.020559	0.015999	-0.008805	-0.042962	-0.052826
B	0.009312	0.009368	0.009673	0.010037	0.009393
AGE	0.000692	-0.000269	-0.005415	-0.010707	0.001212
TAX	-0.012335	-0.012421	-0.012912	-0.013993	-0.015856
CRIM	-0.108011	-0.107474	-0.104595	-0.101435	-0.102202
LSTAT	-0.524758	-0.525966	-0.533343	-0.559366	-0.660764
PTRATIO	-0.952747	-0.940759	-0.876074	-0.797945	-0.829218
DIS	-1.475567	-1.459626	-1.372654	-1.248808	-1.153390
NOX	-17.766611	-16.684645	-10.777015	-2.371619	-0.262847



→ α 값이 커질수록 회귀 계수 값이 작아짐

→ 하지만 회귀 계수 값이 0이 되지 않음

04. 규제 선형 모델

라쏘 회귀

라쏘 회귀

라쏘 회귀

- 선형 회귀에 $|W|_1$ 를 추가한 회귀 모델

라쏘회귀 비용함수

$$\text{Min}(\text{RSS}(W) + \alpha * ||W||_1)$$

- L1 규제

W 절대값에 대해 패널티를 부여하여 회귀 계수를 작게 만드는 규제

예측 영향력이 작은 피처의 회귀 계수를 0으로 만들어줌 → 피처 선택 가능

- Lasso

사이킷런에서 라쏘 회귀를 구현하는 클래스

주요 생성 파라미터 : alpha (릿지 회귀의 L1 규제 계수)

라쏘 회귀

회귀 종류를 입력받았을 때 해당 규제 선형 회귀의 α 값에 따른 교차검증 결과를 출력하는
get_linear_reg_eval 함수 코드가 교재에 나와있으니 공부해보세요!

α 값이 각각 0.07, 0.1, 0.5, 1, 3 일 때 교차 검증 결과

alpha 0.07일 때 5 폴드 세트의 평균 RMSE: 5.612

alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.615

alpha 0.5일 때 5 폴드 세트의 평균 RMSE: 5.669

alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.776

alpha 3일 때 5 폴드 세트의 평균 RMSE: 6.189

→ α 값이 0.07일 때 가장 성능이 좋게 나옴

라쏘 회귀

α 값이 각각 0.07, 0.1, 0.5, 1, 3 일 때 피처당 회귀 계수값

	alpha:0.07	alpha:0.1	alpha:0.5	alpha:1	alpha:3
RM	3.789725	3.703202	2.498212	0.949811	0.000000
CHAS	1.434343	0.955190	0.000000	0.000000	0.000000
RAD	0.270936	0.274707	0.277451	0.264206	0.061864
ZN	0.049059	0.049211	0.049544	0.049165	0.037231
B	0.010248	0.010249	0.009469	0.008247	0.006510
NOX	-0.000000	-0.000000	-0.000000	-0.000000	0.000000
AGE	-0.011706	-0.010037	0.003604	0.020910	0.042495
TAX	-0.014290	-0.014570	-0.015442	-0.015212	-0.008602
INDUS	-0.042120	-0.036619	-0.005253	-0.000000	-0.000000
CRIM	-0.098193	-0.097894	-0.083289	-0.063437	-0.000000
LSTAT	-0.560431	-0.568769	-0.656290	-0.761115	-0.807679
PTRATIO	-0.765107	-0.770654	-0.758752	-0.722966	-0.265072
DIS	-1.176583	-1.160538	-0.936605	-0.668790	-0.000000

→ α 값이 커질수록 회귀 계수 값이 작아짐
→ 0이 되는 회귀 계수값 존재함

04. 규제 선형 모델

엘라스틱넷 회귀

엘라스틱넷 회귀

엘라스틱넷 회귀

- L2규제 와 L1규제 를 결합한 회귀

엘라스틱넷 회귀 비용함수

$$\text{Min}(RSS(W) + \alpha_2 * \|W\|_2^2 + \alpha_1 * \|W\|_1)$$

- ElasticNet

사이킷런에서 엘라스틱넷 회귀를 구현하는 클래스

엘라스틱 넷의 규제 = $a*L1 + b*L2$ (a는 L1 규제의 alpha값, b는 L2 규제의 alpha값)

주요 생성 파라미터 : alpha = a+b

$$l1_ratio = a/(a+b)$$

*l1_ratio가 0이면 a가 0이므로 L2 규제와 동일, 1이면 b가 0이므로 L1규제와 동일

엘라스틱넷 회귀

α 값이 각각 0.07, 0.1, 0.5, 1, 3 일 때 교차 검증 결과

alpha 0.07일 때 5 폴드 세트의 평균 RMSE: 5.542

alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.526

alpha 0.5일 때 5 폴드 세트의 평균 RMSE: 5.467

alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.597

alpha 3일 때 5 폴드 세트의 평균 RMSE: 6.068

→ α 값이 0.5일 때 가장 성능이 좋게 나옴

엘라스틱넷 회귀

α 값이 각각 0.07, 0.1, 0.5, 1, 3 일 때 피처당 회귀 계수값

	alpha:0.07	alpha:0.1	alpha:0.5	alpha:1	alpha:3
RM	3.574162	3.414154	1.918419	0.938789	0.000000
CHAS	1.330724	0.979706	0.000000	0.000000	0.000000
RAD	0.278880	0.283443	0.300761	0.289299	0.146846
ZN	0.050107	0.050617	0.052878	0.052136	0.038268
B	0.010122	0.010067	0.009114	0.008320	0.007020
AGE	-0.010116	-0.008276	0.007760	0.020348	0.043446
TAX	-0.014522	-0.014814	-0.016046	-0.016218	-0.011417
INDUS	-0.044855	-0.042719	-0.023252	-0.000000	-0.000000
CRIM	-0.099468	-0.099213	-0.089070	-0.073577	-0.019058
NOX	-0.175072	-0.000000	-0.000000	-0.000000	-0.000000
LSTAT	-0.574822	-0.587702	-0.693861	-0.760457	-0.800368
PTRATIO	-0.779498	-0.784725	-0.790969	-0.738672	-0.423065
DIS	-1.189438	-1.173647	-0.975902	-0.725174	-0.031208

→ α 값이 커질수록 회귀 계수 값이 작아짐

→ 0이 되는 회귀 계수값 존재하지만 라쏘 회귀보다는 덜 극단적임

04. 규제 선형 모델

선형 회귀 모델을 위한 데이터 변환 방법

선형 회귀 모델을 위한 데이터 변환 방법

선형 회귀 모델의 경우 피처값과 타겟값 특히 타겟값의 분포가 정규 분포인 형태를 매우 선호합니다.

데이터의 정규화를 진행한다고 성능이 무조건 좋아지는 것은 아니지만

왜곡된 형태의 데이터는 성능에 부정적인 영향을 끼칠 확률이 높기 때문에 정규화 작업을 수행하는 것이 좋습니다



1. **StandardScaler** : 평균이 0, 분산이 1인 표준 정규 분포 형태의 데이터 세트로 변환
2. **MinMaxScaler** : 최솟값 0, 최댓값이 1인 값으로 정규화 수행
3. 스케일링 / 정규화를 수행한 데이터 세트에 다시 **다항 특성을 적용**하여 반환
-> 1, 2번 방법이 성능 향상에 효과가 없을 때 적용하는 방법으로 피처가 많으면 과적합과 시간이 오래 걸린다는 문제 발생
4. **로그 변환**을 취해 정규 분포에 가깝게 만들기, 원래 값에 log 함수 적용
-> 가장 많이 사용되는 방법, 심하게 왜곡된 데이터에 좋음.

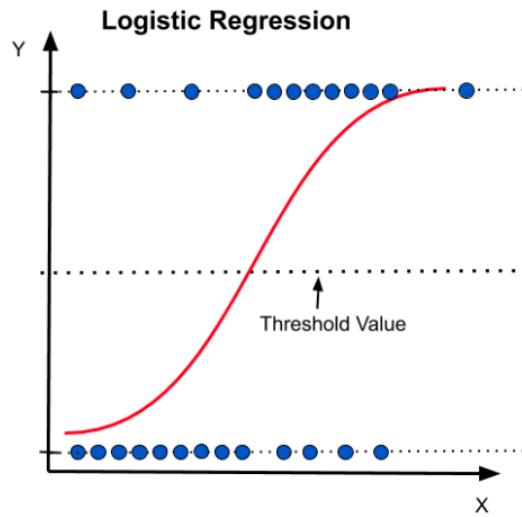
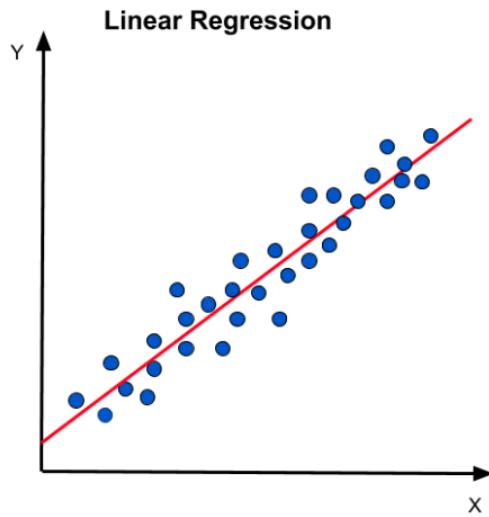
교재 p348-349에 각 정규화 및 스케일링을 진행했을 때 성능 변화를 보여주는 코드가 있으니 공부해보세요!

05. 로지스틱 회귀

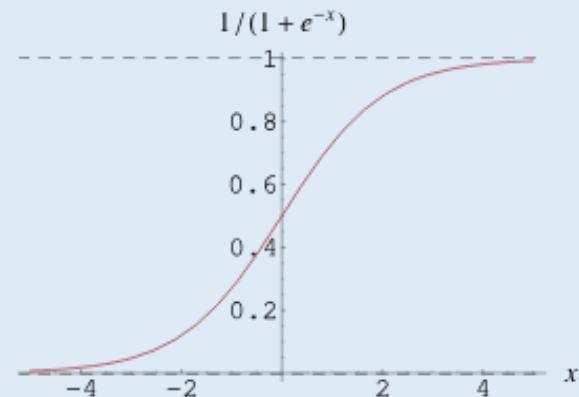
로지스틱 회귀

로지스틱 회귀

- 선형 회귀 방식을 ~~기반~~에 적용한 알고리즘
- 시그모이드 함수 최적선을 찾고 시그모이드 함수 반환 값을 확률로 간주해 확률에 따라 분류를 결정



시그모이드 함수



x값이 +,-로 아무리 커지거나 작아져도 y값은 항상 0과 1사이 값 변환
x값이 커지면 1에 근사하며 x값이 작아지면 0에 근사함
x= 0 일때는 0.5

위스콘신 유방암 - Logistic Regression

데이터세트 및 라이브러리 로드

```
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()
```

StandardScaler를 통한 데이터세트 정규화

```
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_size=0.3, random_state=0)
```

위스콘신 유방암 - Logistic Regression

로지스틱 회귀 학습 / 예측 / 평가

```
from sklearn.metrics import accuracy_score, roc_auc_score

# 로지스틱 회귀를 이용하여 학습 및 예측 수행.
# solver인자값을 생성자로 입력하지 않으면 solver='lbfgs'
lr_clf = LogisticRegression() # solver='lbfgs'
lr_clf.fit(X_train, y_train)
lr_preds = lr_clf.predict(X_test)
lr_preds_proba = lr_clf.predict_proba(X_test)[:, 1]

# accuracy와 roc_auc 측정
print('accuracy: {:.3f}, roc_auc:{:.3f}'.format(accuracy_score(y_test, lr_preds),
                                                roc_auc_score(y_test , lr_preds_proba)))
```

accuracy: 0.977, roc_auc:0.995

로지스틱 회귀의 solvers 파라미터 변화에 따른 평가 결과

```
solvers = ['lbfgs', 'liblinear', 'newton-cg', 'sag', 'saga']
# 여러개의 solver값 별로 LogisticRegression 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]

    # accuracy와 roc_auc 측정
    print('solver:{0}, accuracy: {:.3f}, roc_auc:{:.3f}'.format(solver,
                                                                accuracy_score(y_test, lr_preds),
                                                                roc_auc_score(y_test , lr_preds_proba)))
```

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

위스콘신 유방암 - Logistic Regression

GridSearchCV를 통한 하이퍼 파라미터 튜닝

```
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_params_,  
                                                               grid_clf.best_score_))
```

최적 하이퍼 파라미터:{'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}, 최적 평균 정확도:0.979

로지스틱 회귀의 주요 하이퍼 파라미터 정리

- Penalty : 규제의 유형 결정
- C: 규제 강도를 조절하는 alpha 값의 역수, $C = 1/\alpha$
C가 작을 수록 규제 강도가 커짐.
- solver : 회귀 계수 최적화 방안 -> 큰 차이는 없으나 보통 'liblinear'를 사용함.

07. 회귀 트리

회귀 트리 소개

회귀 트리

지금까지 배운 단순 선형 회귀, 다중 선형 회귀, 다항 회귀, 규제 선형 회귀는 회귀 함수식을 도출해내는 함수 기반 회귀였습니다.

회귀 트리는 별도의 함수식을 도출해내는 것이 아닌 트리를 생성해 회귀를 실행하는 **트리 기반 회귀**입니다



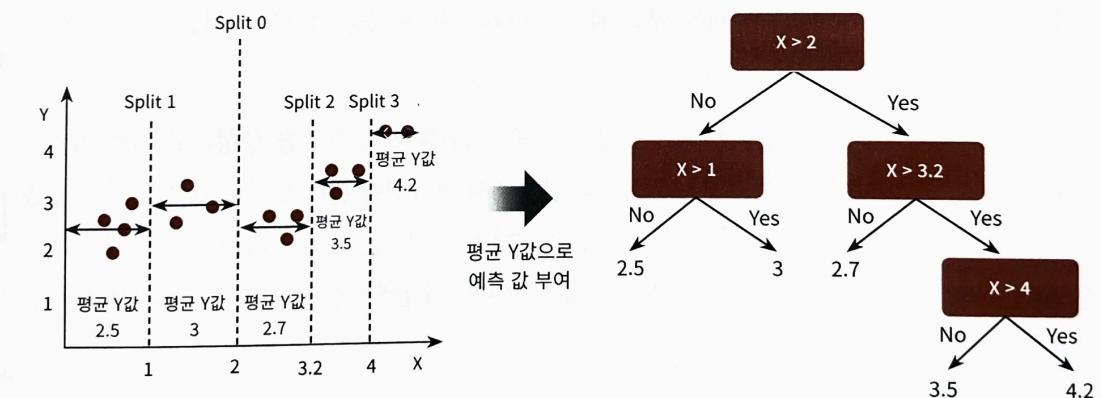
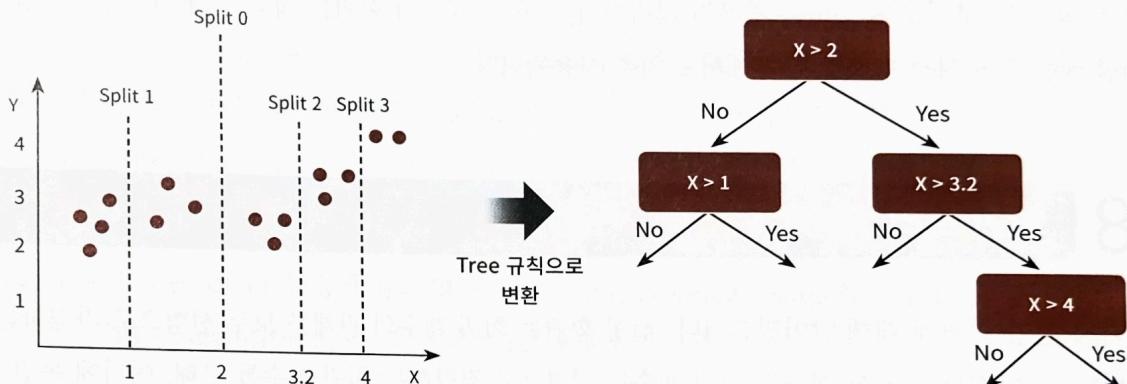
회귀 트리

- 회귀를 위한 **트리**를 생성하고 이를 기반으로 회귀 예측을 하는 알고리즘
- 결정 트리가 특정 클래스 레이블을 결정하는 것과 달리 회귀 트리는 리프 노드에 속한 데이터 값의 **평균**을 구해 회귀 예측 값을 계산함

회귀 트리

회귀 트리 동작 원리

- RSS를 가장 잘 줄일 수 있는 변수를 기준으로 **규칙노드** 생성
- Split을 기준으로 노드 재귀적으로 분할
- 리프 노드에 소속된 데이터 값의 **평균값**을 구해 리프 노드의 결정값으로 할당



CART 알고리즘

CART (classification and Regression Trees)

- 분류 뿐만 아니라 회귀도 가능하게 해주는 트리 생성 알고리즘

-> 결정트리, 랜덤포레스트, GBM, XGBoost, LightGBM 등의 모든 트리 기반 알고리즘은 분류 뿐 아니라 회귀도 가능함

알고리즘	회귀 Estimator 클래스	분류 Estimator 클래스
Decision Tree	DecisionTree Regressor	DecsisionTree Classifier
Gradient Boosting	Gradient Boosting Regressor	Gradient Boosting Classifier
XGBoost	XGB Regressor	XGB Classifier
LightGBM	LGBM Regressor	LGBM Classifier

07. 회귀 트리

보스턴 주택 가격 예측

보스턴 주택 가격 예측 - 회귀 트리

RandomForestRegressor를 이용한 보스턴 주택 가격 예측

```
from sklearn.datasets import load_boston
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings('ignore') # 사이킷런 1.2 부터는 보스턴 주택가격 데이터가 없어진다는 warning 메시지 출력 제거

# 보스턴 데이터 세트 로드
boston = load_boston()
bostonDF = pd.DataFrame(boston.data, columns = boston.feature_names)

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

rf = RandomForestRegressor(random_state=0, n_estimators=1000)    ➔ CART의 회귀 Estimator 중 RandomForestRegressor 응용
neg_mse_scores = cross_val_score(rf, X_data, y_target, scoring="neg_mean_squared_error", cv = 5)    ➔ 회귀 평가
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 : {:.3f}'.format(avg_rmse))
```

보스턴 주택 가격 예측 - 회귀 트리

랜덤포레스트뿐만 아니라 결정트리, GBM, XGBoost, LightGBM의 Regressor를 모두 이용할 수 있음

```
def get_model_cv_prediction(model, X_data, y_target):
    neg_mse_scores = cross_val_score(model, 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('##### ', 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

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=1000)
xgb_reg = XGBRegressor(n_estimators=1000)
lgb_reg = LGBMRegressor(n_estimators=1000)
```

▶ Regressor 생성

```
# 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)
```

▶ Regressor들에 대해 평가 수행

보스턴 주택 가격 예측 - 회귀 트리

회귀 트리 Regressor 클래스는 선형회귀와 다른 처리 방식이므로 회귀 계수를 제공하는 coef_ 속성이 없음
feature_importances_를 이용해 피처별 중요도를 시각화

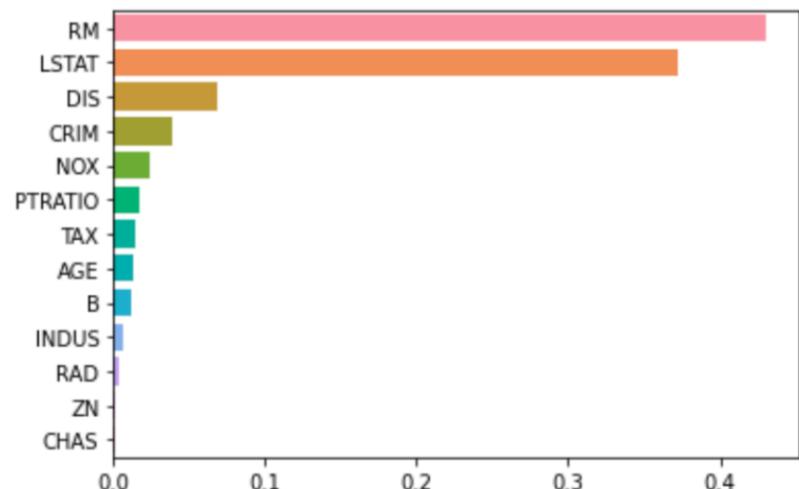
```
import seaborn as sns
%matplotlib inline

rf_reg = RandomForestRegressor(n_estimators=1000)

# 앞 예제에서 만들어진 X_data, y_target 데이터 셋을 적용하여 학습합니다.
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)
```

<AxesSubplot :>



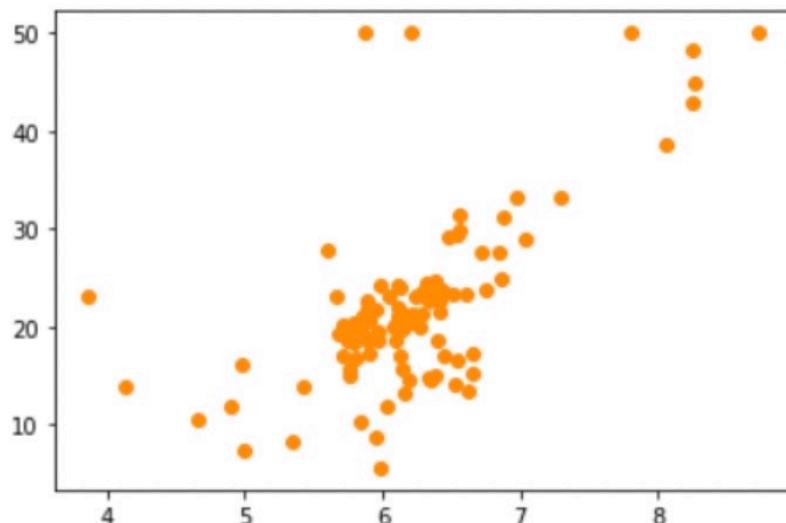
보스턴 주택 가격 예측 - 회귀 트리

‘RM’과 ‘PRICE’ 피처만 사용, 데이터세트 산점도로 시각화

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

bostonDF_sample = bostonDF[['RM', 'PRICE']]
bostonDF_sample = bostonDF_sample.sample(n=100, random_state=0)
print(bostonDF_sample.shape)
plt.figure()
plt.scatter(bostonDF_sample.RM , bostonDF_sample.PRICE,c="darkorange")
```

<matplotlib.collections.PathCollection at 0x2625202cf70>



→ ‘PRICE’와 가장 밀접한 ‘RM’을 이용해 예측 회귀선을 시각화하기 위해
데이터세트를 산점도로 시각화

보스턴 주택 가격 예측 - 회귀 트리

OLS 기반 선형 회귀와 DecisionTreeRegressor 모델 학습 / 예측 수행

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

# 선형 회귀와 결정 트리 기반의 Regressor 생성. DecisionTreeRegressor의 max_depth는 각각 2, 7
lr_reg = LinearRegression()          → 선형 회귀
rf_reg2 = DecisionTreeRegressor(max_depth=2)    → 결정 트리 Regressor (max_depth : 2 / 7)
rf_reg7 = DecisionTreeRegressor(max_depth=7)

# 실제 예측을 적용할 테스트용 데이터 셋을 4.5 ~ 8.5 까지 100개 데이터 셋 생성.
X_test = np.arange(4.5, 8.5, 0.04).reshape(-1, 1)

# 보스턴 주택가격 데이터에서 시각화를 위해 피처는 RM만, 그리고 결정 데이터인 PRICE 추출
X_feature = bostonDF_sample['RM'].values.reshape(-1,1)
y_target = bostonDF_sample['PRICE'].values.reshape(-1,1)

# 학습과 예측 수행.
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)
```

보스턴 주택 가격 예측 - 회귀 트리

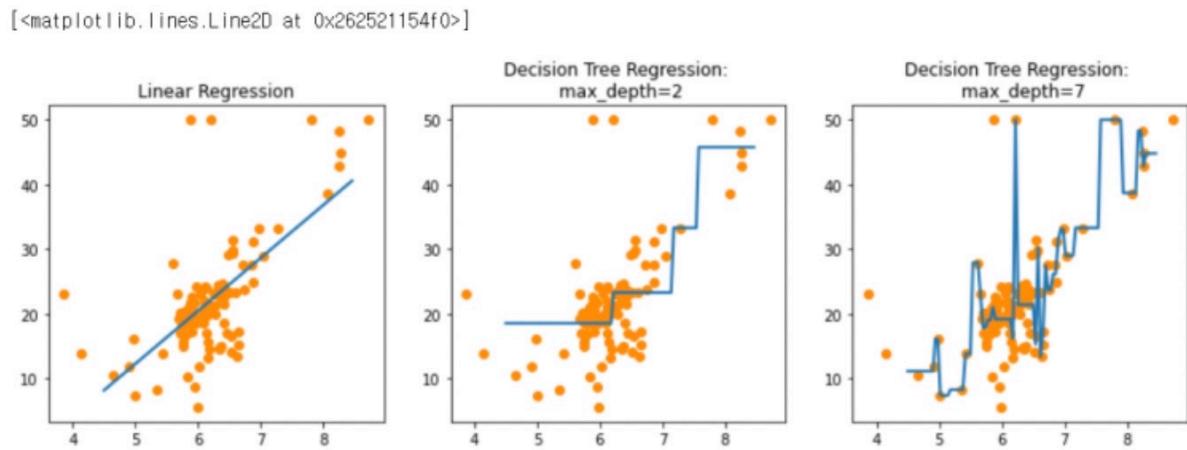
각각의 조건에서 회귀 예측선 시각화

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

# X축값을 4.5 ~ 8.5로 변환하며 입력했을 때, 선형 회귀와 결정 트리 회귀 예측 선 시각화
# 선형 회귀로 학습된 모델 회귀 예측선
ax1.set_title('Linear Regression')
ax1.scatter(bostonDF_sample.RM, bostonDF_sample.PRICE, c="darkorange")
ax1.plot(X_test, pred_lr,label="linear", linewidth=2 )

# DecisionTreeRegressor의 max_depth를 2로 했을 때 회귀 예측선
ax2.set_title('Decision Tree Regression: \n max_depth=2')
ax2.scatter(bostonDF_sample.RM, bostonDF_sample.PRICE, c="darkorange")
ax2.plot(X_test, pred_rf2, label="max_depth:3", linewidth=2 )

# DecisionTreeRegressor의 max_depth를 7로 했을 때 회귀 예측선
ax3.set_title('Decision Tree Regression: \n max_depth=7')
ax3.scatter(bostonDF_sample.RM, bostonDF_sample.PRICE, c="darkorange")
ax3.plot(X_test, pred_rf7, label="max_depth:7", linewidth=2)
```



회귀 총정리

회귀 총정리

	기반		최적화			함수	평가 지표
회귀	함수	회귀 함수	선형 회귀 비선형 회귀 단일 회귀 다항 회귀	오류합 최소화	경사하강법	LinearRegression	MAE
					OLS		MSE
					-		RMSE
	트리	회귀 트리	CART의 회귀 Estimator	파라미터 튜닝	min_samples_split	DecisionTreeRegressor	MSLE
					min_samples_leaf		
					max_features	GradientBoostingRegressor	R ²
					max_depth		
					max_leaf_nodes	XGBRegressor	
					LGBMRegressor		

수고하셨습니다