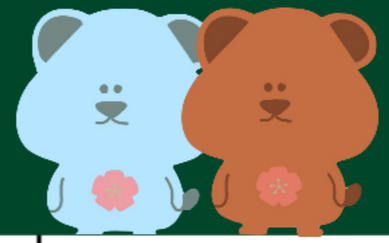
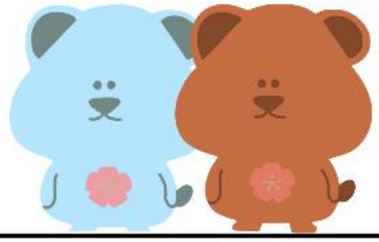


5주차 발표

DA팀 박지운 오연재 이서영

EWHA EURON

목차



EWHA EURON

01. 분류, 회귀

EWHA EURON

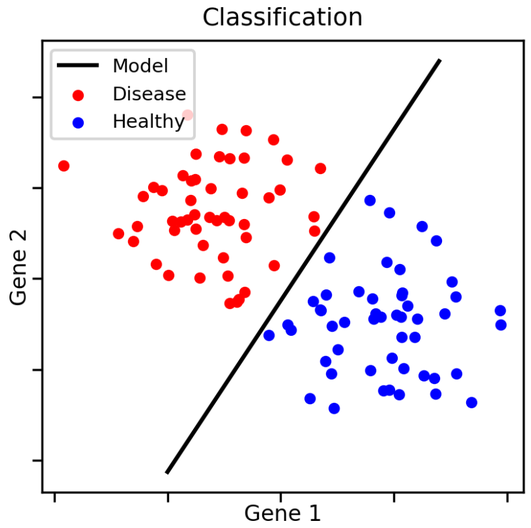
1.1 분류 vs 회귀

지도학습

예측값

분류(classification)

Category(이산값)

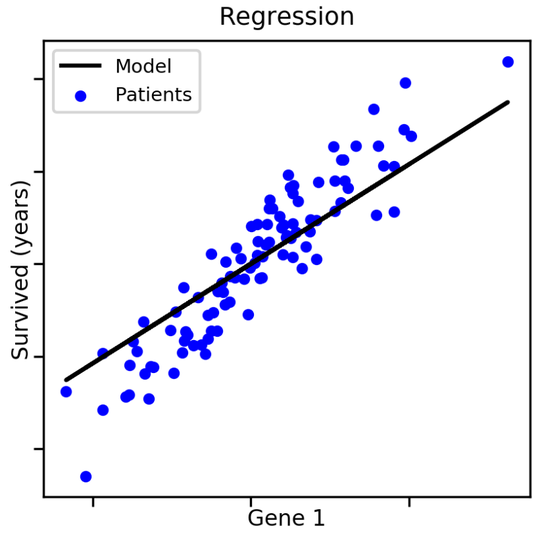


이진 분류 (Binary classification)
예측해야 할 class가 두 가지인 경우이다.

다중 분류 (Multi-class classification)
예측해야 할 class가 여러 가지인 경우이다.
Ex)
입력 text가
영어/한국어 / 일본어 / 중국어 / 등등 어느 언어인지 분류한다.

회귀(regression)

Continuous(연속값)



연속적인 숫자, 즉 예측값이 float 형태인 문제들을 해결하는데 사용된다.

Ex)
지하철 역과의 거리, 일정 거리안의 관공서, 마트, 학군의 수 등등 여러 feature들로 어떤 지역의 땅값을 예측하는 문제와 같은 것들이다.

1.2 회귀 유형 구분

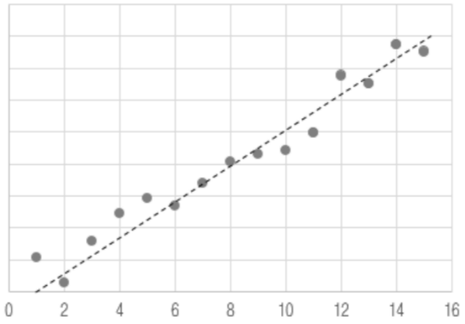
회귀

회귀: 여러 개의 독립변수 와 한 개의 종속변수(y값) 간의 상관관계를 모델링하는 기법을 통칭

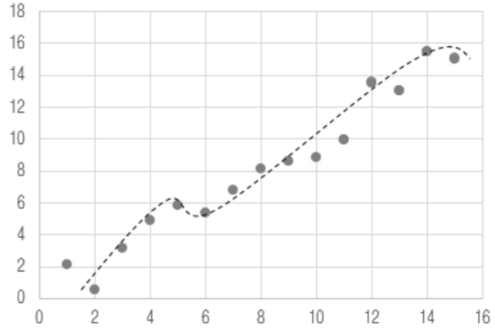
독립변수 / 종속변수 / 회귀계수 → 최적의 회귀계수를 도출하는 것이 목적이다.

$$Y_i = a + bX_{1i} + cX_{2i} + u_i$$

Y_i : 종속 변수
X_{1i} : 첫 번째 독립변수
X_{2i} : 두 번째 독립변수
a, b, c : 회귀 계수



선형회귀



비선형회귀

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

1.3 규제 모형 – 선형 회귀 모형

선형 회귀

실제 값과 예측값의 차이(오류의 제곱 값)를 최소화하는 직선형 회귀선을 최적화하는 방식

규제(Regularization)

모델이 과대 적합을 피하여 일반화 성능을 잃지 않도록 가중치를 제한하는 방법으로, 과대 적합을 완화하기 위한 대표적인 방식

손실함수에 가중치의 노름(norm)을 더한 함수를 목적 함수(Objective function)로 설정하여 가중치를 제한합니다.

L1 규제(L1 Regularization) : 목적 함수 = 손실함수 + L1-norm term

$$L_1 - norm : ||\boldsymbol{w}||_1 = \sum_{i=1}^{n} |w_i|$$

$$J(w) = Loss(w) + \frac{\lambda}{2} \times \frac{1}{n} \sum_{i=1}^{n} |w_i|$$

- **L2 규제(L2 Regularization)** : 목적 함수 = 손실함수 + (L2-norm)

$$L_2 - norm : ||\boldsymbol{w}||_2 = \sqrt{\sum_{i=1}^{n} |w_i|^2}$$

$$J(w) = Loss(w) + \frac{\lambda}{2} \times \frac{1}{n} \sum_{i=1}^{n} |w_i|^2$$

EWHA EURON

1.3 규제 모형

모형	
알반 선형 회귀	RSS를 최소화, 규제를 적용하지 않은 모델
릿지(Ridge)	선형회귀 + L2규제 (상대적으로 큰 회귀 계수 값의 예측 영향도를 감소시키기 위해 회귀 계수값을 더 작게 만든다.)
라쏘(Lasso)	선형회귀 + L1규제 (예측 영향력이 작은 피처의 회귀 계수를 0으로, 일부 피처만을 회귀 예측에 사용→ 피처 선택 기능)
엘라스틱넷(Elastic Net)	L1+L2 피처가 많은 데이터에 적용 L1 규제(피처의 개수를 감소) + L2 규제(계수 값의 크기 조정)
로지스틱 회귀(Logistic regression)	분류에 이용하는 선형 모델 이진 분류, 텍스트 분류 등에 이용

02. 선형 회귀

EWHA EURON

2.1 단순 선형 회귀

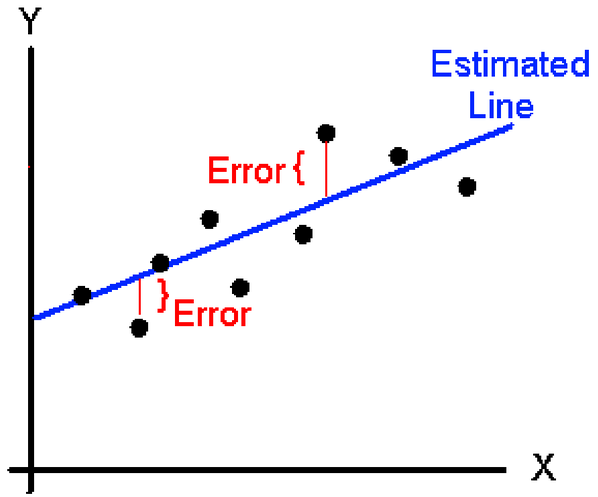
단순 선형 회귀

실제 값과 회귀 모델의 차이에 따른 남은 오류(잔차)→ 잔차(RSS)가 최소가 되는 모델

(mean absolute error): 오류를 절댓값 취해 합함

(RSS, residual sum of square): 오류 값의 제곱을 합함

→ 일반적으로 RSS으로 오류를 측정한다.



RSS

$$\sum_{i=1}^{n}(\varepsilon_i)^2 = \sum_{i=1}^{n}(y_i - f(x_i))^2 = \sum_{i=1}^{n}(y_i - \alpha x_i + \beta)^2$$

α와 β는 RSS를 최소화 하는 값으로 모델 학습을 통해서 얻어지는 값

비용함수

[개념] Gradient Descent: 경사하강법 :: 컴파일도 코드한줄부터 (tistory.com)

RSS(오류 제곱의 합)

2.2 회귀 성능 지표

평가 지표	설명	수식
MAE	실제값과 예측값의 차이인 오차들의 절댓값 평균 MSE보다는 특이치에 덜 민감하다.	MAE $\frac{1}{n}\sum_{i=1}^{n}(\lvert y_i - \hat{y}_i \rvert)$ L1-norm
MSE	회귀 모델의 주요 손실함수 예측값과 실제값의 차이인 오차들의 제곱 평균으로 정의한다. 제곱을 하기 때문에 특이치(아웃라이어)에 민감하다.	MSE $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ L2-norm
RMSE	MSE에 root를 씌운 값 오류 지표를 실제 값과 유사한 단위로 다시 변환하기에 해석이 다소 용이해진다.	RMSE $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$
R^2	분산기반 예측 성능 평가 다른 MAE, MSE 등과 같은 지표들는 데이터의 scale에 따라 값이 다르지만 R2은 상대적인 성능이 어느정도인지 직관적으로 판단할 수 있다.	예측값 variance / 실제값 variance

2.3 선형 회귀(사이킷런) - scoring 파라미터

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
R^2	metrics.r2_score	r2

→MAE는 절대값의 합이므로 음수가 나올 수 없다. 하지만 scoring 함수에서 MAE를 음수로 반환하는 이유는 사이킷 런의 scoring 함수가 score값이 클 수록 좋은 평가 결과로 평가하기 때문이다.

→-1을 원래의 평가 지표 값에 곱해 음수를 만들어 작은 오류 값을 더 큰 숫자로 인식하게 한다.

```
# 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)
```

2.4 선형 회귀 - 보스턴 예시

데이터 세트와 라이브러리 로드

```
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)
```

Target변수는 주택 가격이고 이를 price 칼럼으로 dataframe에 추가

```
bostonDF.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    float64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    float64
 9   TAX      506 non-null    float64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  PRICE    506 non-null    float64
dtypes: float64(14)
memory usage: 55.5 KB
```

Null 값은 없고 모두 float형이다.

2.4 선형 회귀 - 보스턴 예시

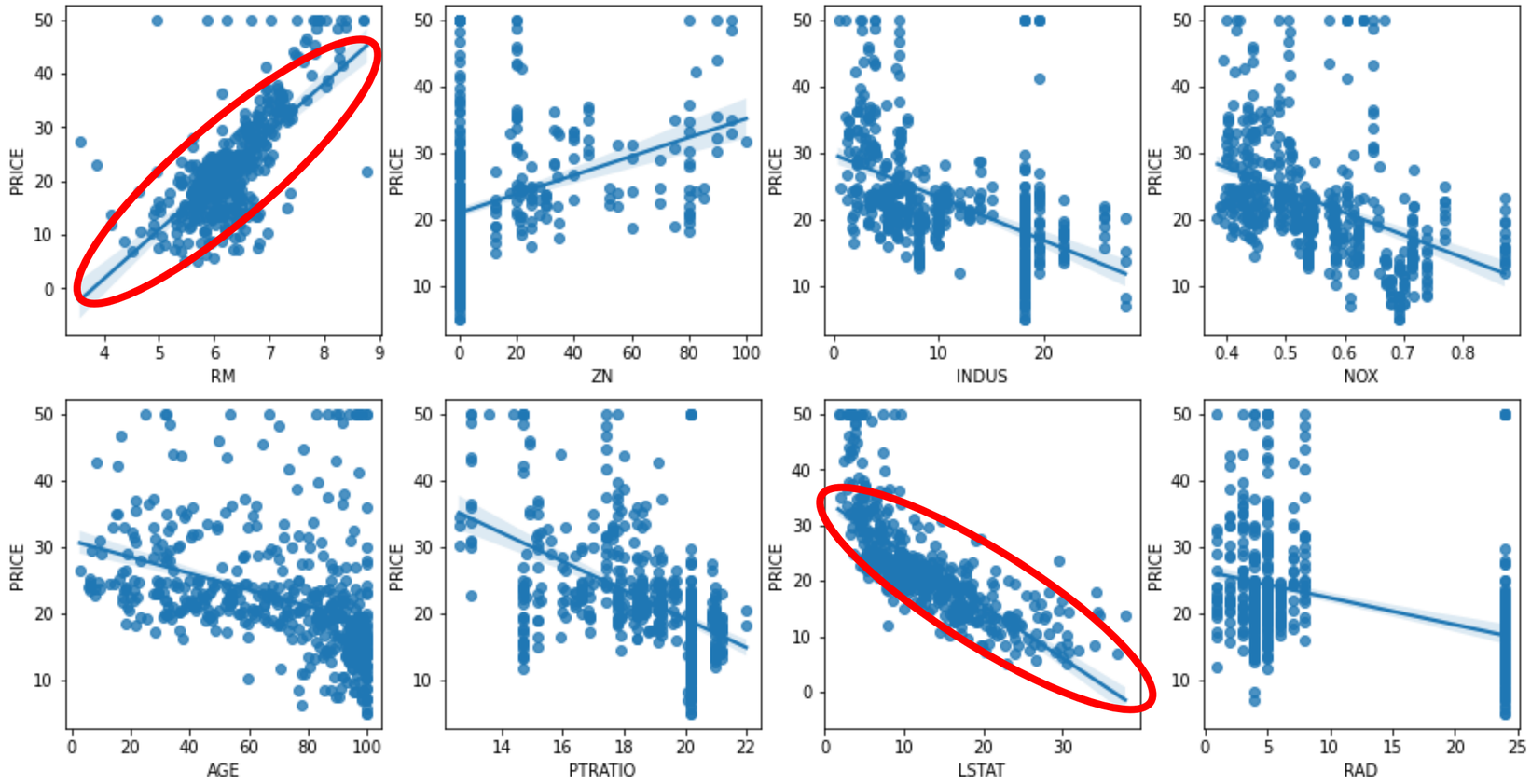
각 칼럼이 회귀 결과에 미치는 영향 파악

```
# 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])
```

fig, axs = plt.subplots(figsize=(가로 inch, 세로 inch), ncols=열 개수, nrows=행 개수)

lm_features=['피처명1 ', '피처명2']

1행 1열에 RM plot ~ 2행 4열에 RAD plot이 오도록 설정



8개의 피처값 중에서 price와
선형성이가장 두드러지는 변수는
RM(양의 상관관계),
LSTAT(음의 상관관계) 임을 알 수 있다.

2.4 선형 회귀 - 보스턴 예시

LinearRegression을 이용하여 회귀 모델 설정

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

MSE와 R^2값을 이용한다.

```
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 계수값:', np.round(lr.coef_, 1))
```

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

```
# 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
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
```

2.4 선형 회귀 - 보스턴 예시

교차 검증 (5개의 폴드 셋)

```
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
```

cross_val_score()에서 반환된 값에 -1 을 곱하여
양의 값인 원래 모델에서 MSE를 계산한다.

scoring="neg_mean_squared_error"
을 통해 나오는 수치는 음수값

2.5 선형 회귀 - OLS

사이킷런 이외의 모듈에서 제공하는 Linear regression model

- statsmodels.formula `from statsmodels.formula import api`

```
#Testing a Linear Regression model with statsmodels

Train_xy = pd.concat([Train_X_std,Train_Y.reset_index(drop=True)],axis=1)
a = Train_xy.columns.values

API = api.ols(formula='{} ~ {}'.format(target,' + '.join(i for i in Train_X.columns)), data=Trai
n_xy).fit()
#print(API.conf_int())
#print(API.pvalues)
API.summary()
```

R과 같은 형식

- statsmodels.regression.linear_model에서도 OLS 제공

```
>>> model = sm.OLS(Y,X)
>>> results = model.fit()
```

- 여기서는 OLS(Y,X)로 fit하면 된다.
- from_formula라는 method 존재해 직접 식을 넣어줘도 된다.

➡ 피처 중요도 & 다중공선성을 확인할 수 있어 자주 사용됨.

statsmodels.regression.linear_model.OLS — statsmodels 0.8.0 documentation

2.5 선형 회귀 - OLS

OLS summary를 통해 알 수 있는 것들

https://medium.com/swlh/interpreting-linear-regression-through-statsmodels-summary-4796d359035a

OLS Regression Results

Dep. Variable:	price	R-squared:	0.679
Model:	OLS	Adj. R-squared:	0.661
Method:	Least Squares	F-statistic:	36.96
Date:	Wed, 09 Feb 2022	Prob (F-statistic):	2.06e-84
Time:	20:12:46	Log-Likelihood:	-6509.2
No. Observations:	426	AIC:	1.307e+04
Df Residuals:	402	BIC:	1.316e+04
Df Model:	23		
Covariance Type:	nonrobust		

- **R-squared** : 모델의 설명력
- **Adj.R-squared** : multiple dependent variables' efficacy
- **F-statistic** : group of variables are significant?
- **AIC/BIC** : feature selection에 사용

Omnibus:	96.025	Durbin-Watson:	2.025
Prob(Omnibus):	0.000	Jarque-Bera (JB):	274.474
Skew:	1.058	Prob(JB):	2.51e-60
Kurtosis:	6.315	Cond. No.	26.3

- **Omnibus** : skew와 kurtosis를 이용한 residual들의 분포를 정규화한 것
  -> 0일수록 좋음
- **Prob(Omnibus)** : residual 분포가 normal인지 -> 1일수록 좋음
- **Skew** : symmetry of data -> 0일수록 좋음
- **Kurtosis** : peakiness of data -> 높을수록 outlier 적음
- **Durbin-Watson** : homoscedasticity -> 1~2 사이가 안정적
- **Prob(JB)** : Prob(Omnibus)와 같다고 취급
- **Condition number** : 데이터의 사이즈에 대한 모델의 sensitivity 측정
  -> 1~30의 경우 다중공선성 존재 / 30 이상의 경우 다중공선성 강함

2.5 선형 회귀 - OLS

OLS summary를 통해 알 수 있는 것들

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	4.717e+06	5.22e+04	90.378	0.000	4.61e+06	4.82e+06
area	4.356e+05	6.41e+04	6.799	0.000	3.1e+05	5.62e+05
mainroad	1.785e+05	5.77e+04	3.092	0.002	6.5e+04	2.92e+05
guestroom	1.197e+05	5.78e+04	2.071	0.039	6082.572	2.33e+05
basement	1.712e+05	6.15e+04	2.784	0.006	5.03e+04	2.92e+05
hotwaterheating	2.006e+05	5.48e+04	3.662	0.000	9.29e+04	3.08e+05
airconditioning	3.635e+05	5.89e+04	6.168	0.000	2.48e+05	4.79e+05
prefarea	2.711e+05	5.75e+04	4.711	0.000	1.58e+05	3.84e+05
furnishingstatus_semi_furnished	1.509e+04	6.71e+04	0.225	0.822	-1.17e+05	1.47e+05
furnishingstatus_unfurnished	-1.688e+05	6.78e+04	-2.489	0.013	-3.02e+05	-3.55e+04
bathrooms_2	3.722e+05	5.98e+04	6.224	0.000	2.55e+05	4.9e+05
bathrooms_3	1.886e+05	5.4e+04	3.492	0.001	8.24e+04	2.95e+05
bathrooms_4	2.801e+05	5.68e+04	4.934	0.000	1.69e+05	3.92e+05
stories_2	1.341e+05	6.97e+04	1.923	0.055	-2986.085	2.71e+05
stories_3	2.289e+05	6.13e+04	3.735	0.000	1.08e+05	3.49e+05
stories_4	3.725e+05	6.46e+04	5.764	0.000	2.45e+05	5e+05
parking_1	1.67e+05	5.78e+04	2.887	0.004	5.33e+04	2.81e+05
parking_2	2.781e+05	5.97e+04	4.662	0.000	1.61e+05	3.95e+05
parking_3	-5.772e+04	5.72e+04	-1.009	0.314	-1.7e+05	5.47e+04
bedrooms_2	-3.385e+04	4.8e+05	-0.070	0.944	-9.78e+05	9.11e+05
bedrooms_3	1.077e+05	5.45e+05	0.197	0.844	-9.64e+05	1.18e+06
bedrooms_4	1.215e+05	4.18e+05	0.291	0.771	-7e+05	9.43e+05
bedrooms_5	3.933e+04	1.66e+05	0.237	0.812	-2.86e+05	3.65e+05
bedrooms_6	8.462e+04	7.49e+04	1.130	0.259	-6.26e+04	2.32e+05

Feature별 coef와 검정통계량, 95% CI 제공

- 다중공선성이 있을 경우
  coefficient 간 standard error이 커진다.
- P-value가 유의수준 a보다 작은 경우
  해당 feature가 모델에 유의미한 영향을 준다.
  (feature importance 확인)

피처 중요도 & 다중공선성을 동시에 확인할 수 있어
대회나 공모전에서 자주 사용됨.

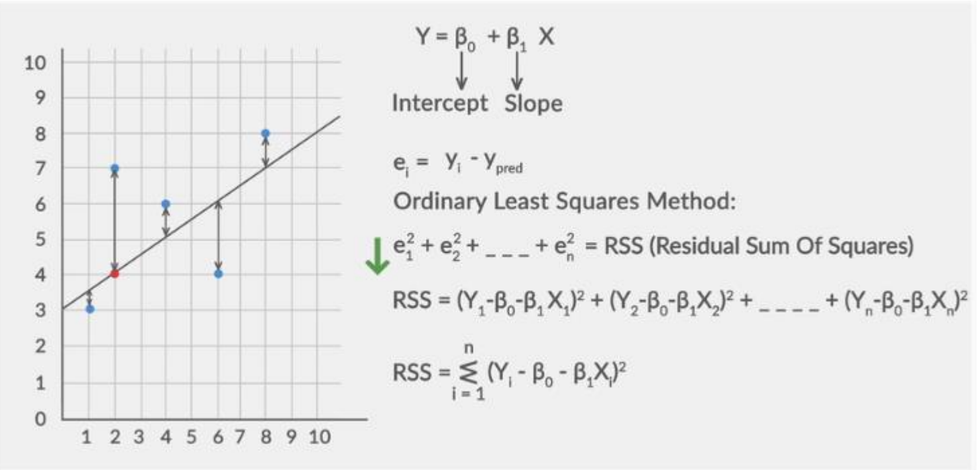
https://www.kaggle.com/code/yasserh/housing-price-prediction-best-ml-algorithms/notebook

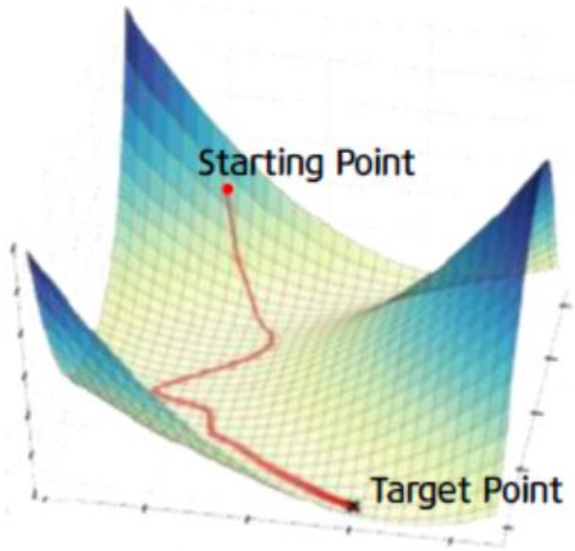
03. 경사 하강법

EWHA EURON

3.1 경사 하강법 – 비용 최소화

경사하강법 - 비용 최소화





경사하강법

비용 함수가 최소가 되는 회귀계수(W) 를 찾아야 한다.
W파라미터가 많은 경우, 고차원 방정식을 이용하기 어렵다.

→**경사하강법**: 고차원 방정식의 문제를 해결, RSS를 최소화하는 방식
→점진적으로 반복적인 계산을 통해 W값을 업데이트하며 RSS를
최소화 한다. (오류 값이 더 이상 작아지지 않을 경우를 최적 파라미터로 반환)

산 밑으로 내려갈 수 있는 간단한 방법은
한 발자국을 내딛었을 때 가장 높이 차이가
많이 나는 방향, 즉 **경사가 가장 가파른 방향으로 이동을 하는 것**이다.
이러한 방식으로 최소점 (최솟값을 갖는 지점) 을 찾는 것을 **Gradient Descent (경사하강법)** 이라고 한다.

참고: [ML/DL] 최적화(Optimization), 경사하강법 (Gradient Descent Algorithms) (tistory.com)

3.1 경사 하강법 – 비용 최소화

경사하강법 - 알고리즘

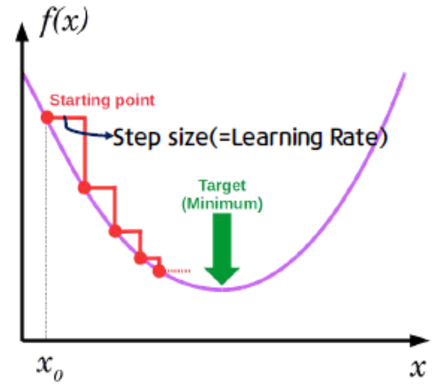
- R(w) 를 미분하여 미분 함수의 최솟값을 구한다.
- R(w)는 두 개의 파라미터가 있으므로 각 변후에 대해 편미분을 한다.
- 편미분을 한 값을 반복적으로 보정하여 R(w) 값이 최소가 되는 회귀계수를 찾는다.
- 이때, 편미분 값이 너무 커지는 것을 방지하고자 학습률을 곱한다.

알고리즘

1) 모든 데이터 포인트에서 파라미터(theta)에 대해 Gradient 값을 계산한다. (theta에 대해 목적함수 미분 후, 데이터 값 대입)
2) 1)에서 구한 Gradient에 대한 평균을 낸다
3) theta를 다음과 같이 업데이트 한다.

theta = theta – learning rate*theta의 gradient

4) 수렴할 때까지 1) ~3) 반복



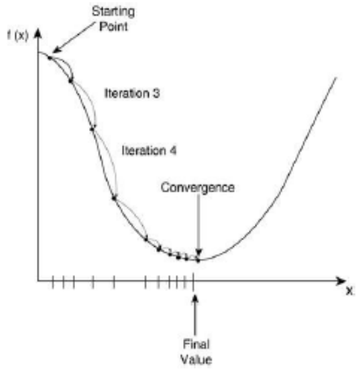
비용함수(E=MSE)

경사하강법에서의 gradient 값 (E를 w에 대해 편미분)

$$\frac{\partial E}{\partial w} = \frac{\partial}{\partial w}\frac{1}{2}\sum_{i=1}^{N}(y_i - t_i)^2$$

가중치 업데이트 $w^{new} = w^{old} - \alpha \Delta w^{old}$

학습률



- 옆의 그림을 보면, x축은 파라미터이고 f(x)는 Cost Function.

1) 임의의 파라미터 값을 시작점으로 지정.
2) 해당 점에서의 목적함수 Gradient값을 구함.
3) 다음과 같이 파라미터를 업데이트 ↓

$$x_{i+1} = x_i - \alpha \nabla f(x_i)$$

목적함수 값이 특정 값으로 수렴할 때까지 위의 과정을 반복.

x_{i+1}: i+1번째 step의 파라미터 x의 값
x_i: i번째 step의 파라미터 x의 값
α: Learning rate or Step size (한 번에 어느 정도 이동할 것인지): 하이퍼파라미터
$\nabla f(x_i)$: 파라미터의 값이 xi일 때의 목적함수 f의 그래디언트 값 (방향 및 기울기)

참고: [ML/DL] 최적화(Optimization), 경사하강법 (Gradient Descent Algorithms) (tistory.com)

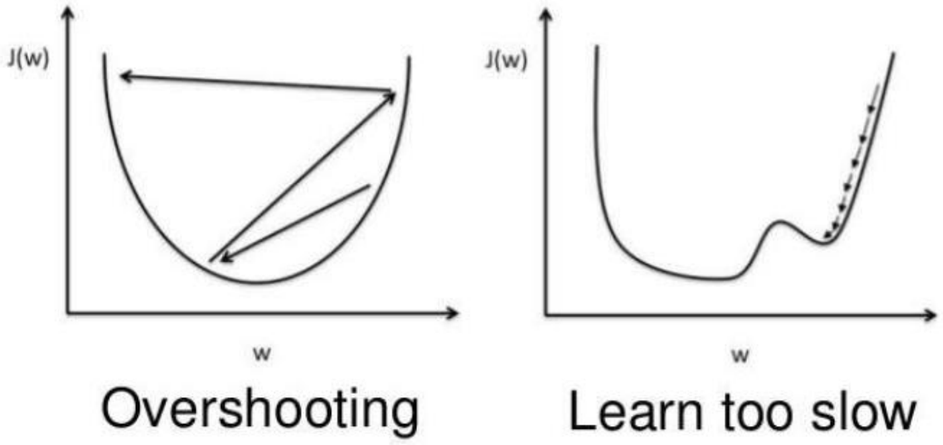
3.1 경사 하강법 – 학습률

경사하강법 – 학습률 설정

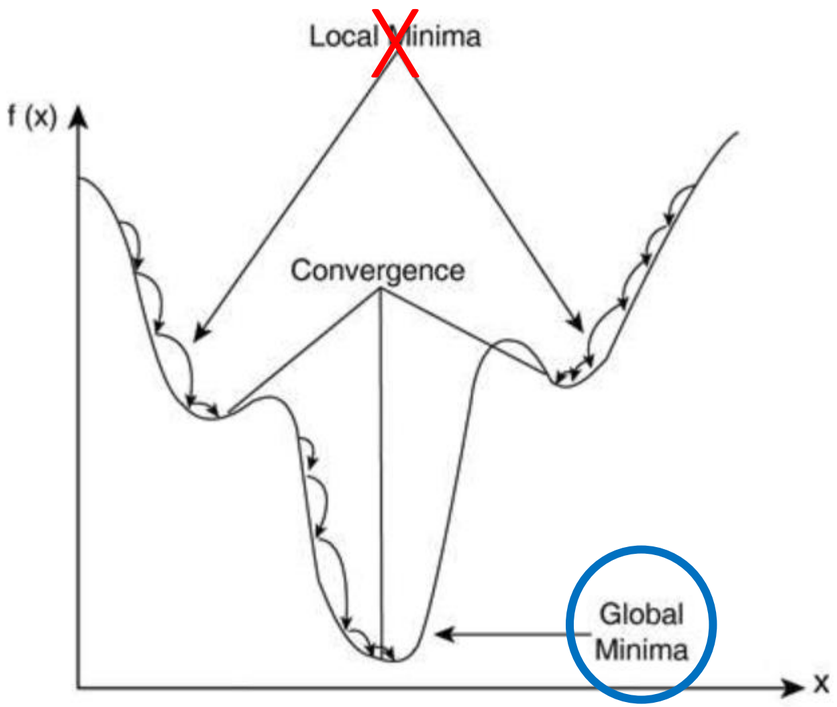
- 학습률이 너무 크면 한 지점으로 수렴하는 것이 아니라 발산할 가능성이 존재한다. (overshooting)

- 너무 작은 학습률을 택할 때에는 수렴이 늦어진다. (learn too slow)

- 시작점을 어디로 잡느냐에 따라 수렴 지점이 달라진다. (의도했던 대로 전역 최소점으로 수렴할 수도 있지만, 지역 최소점으로 수렴할 가능성도 배제할 수 없다.)



학습률 설정 오류에 따른 현상



전역 최소점으로 수렴하지 못하고 지역 최소점으로 수렴하는 경우

참고: [ML/DL] 최적화(Optimization), 경사하강법 (Gradient Descent Algorithms) (tistory.com)

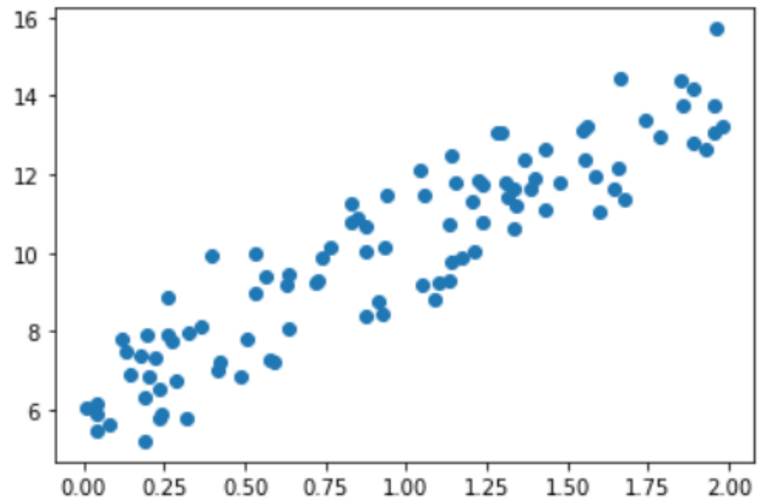
3.1 경사 하강법 – 단순 선형 회귀 예측

임의의 값들을 y=4x+6에 근사시킨다.

노이즈가 섞인 임의의 값 100개(0과 1사이의 값)을
Y=4x+6에 근사시킨다.

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

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

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

```
<matplotlib.collections.PathCollection at 0x216142bdeb0>
```




```
# 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
```

W1_update와 w0_update에는 w1, w0의 크기와 동일한 값으로 원소를 모두 0으로 초기화 시킨다. (np.zeros_like( ))

넘파이의 내적연산인 dt()을 이용하여 계산한다.

```
# 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
```

3.1 경사 하강법 – 단순 선형 회귀 예측

예측 오류 계산

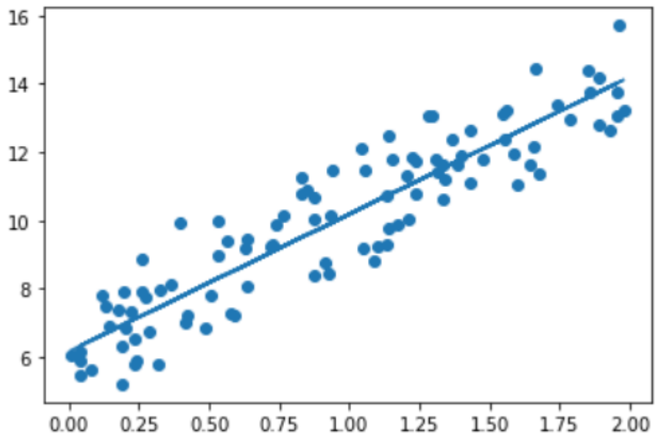
```
def get_cost(y, y_pred):
    N = len(y)
    cost = np.sum(np.square(y - y_pred))/N RSS값을 cost에 지정한다.
    return cost

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
```

→ 처음에 지정한 y=4x+6과 유사하게 회귀계수 값이 나온다.
→ 예측오류 비용: 0.9935

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

```
[<matplotlib.lines.Line2D at 0x2161422d310>]
```



(미니 배치) 확률적 경사 하강법

경사 하강법은 수행 시간이 매우 오래 걸린다. (모든 학습 데이터에 대해 반복적으로 비용함수를 업데이트하기 때문에)

실전에서는 확률적 경사 하강법(Stochastic Gradient Descent)를 이용한다.
→ 전체가 아닌 일부 데이터만을 이용해 W를 업데이터 한다. (빠른 속도)
→ 따라서 대용량의 데이터는 확률적 경사하강법 / 미니 배치 확률적 경사 하강법 을 이용

```
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)))
```

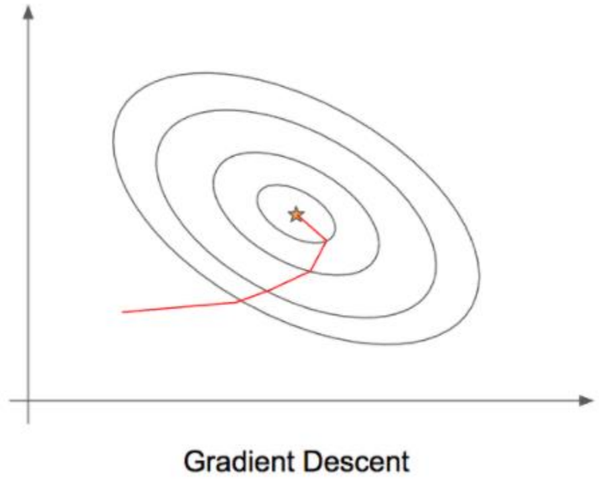
```
w1: 4.028 w0: 6.156
Stochastic Gradient Descent Total Cost:0.9937
```

→W의 값이 경사하강법과 큰 차이가 없고 예측오류 비용도 아주 미세하게 커졌다.
소요시간은 더 적으므로 일반적으로 확률적 경사 하강법을 이용한다.

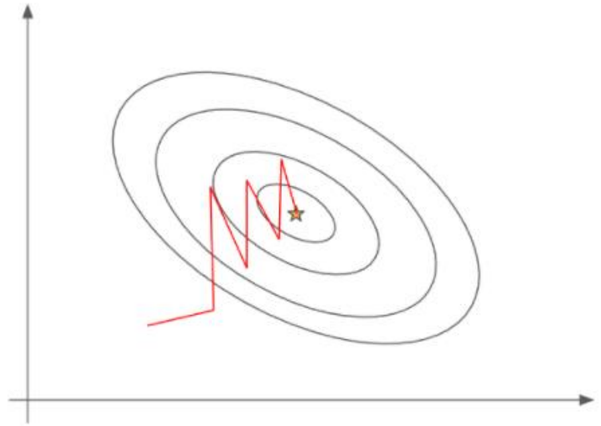
3.2 확률적 경사 하강법(SGD) , epoch

→전체 샘플을 사용하지 않고 딱 하나의 샘플을 훈련 세트에서 랜덤하게 선택
→가파른 경사를 조금씩 내려온다. → 또 다른 샘플을 선택하여 경사를 내려온다.
→전체 샘플을 모두 이용할 때까지 반복한다.

→가장 아래로 내려오지 못할 경우: 처음부터 다시 시작한다.
이처럼 훈련 세트를 한 번 모두 사용하는 과정을 epoch(에포크) 라고 한다.
→다시 시작하는 위험이 있으므로 조금씩 내려가야 한다.
(하지만 확률적 경사 하강법은 잘 작동하는 편이다.)


Gradient Descent

- SGD의 **장점**
1) 실제 모든 데이터에 대해 Gradient 계산하는 것보다 연산 시간이 단축
2) Shooting이 일어나기 때문에 local minima에 빠질 risk가 적다.


Stochastic Gradient Descent

- SGD의 **단점**
1) Global Optimum으로의 수렴을 역시 보장하지 못한다.

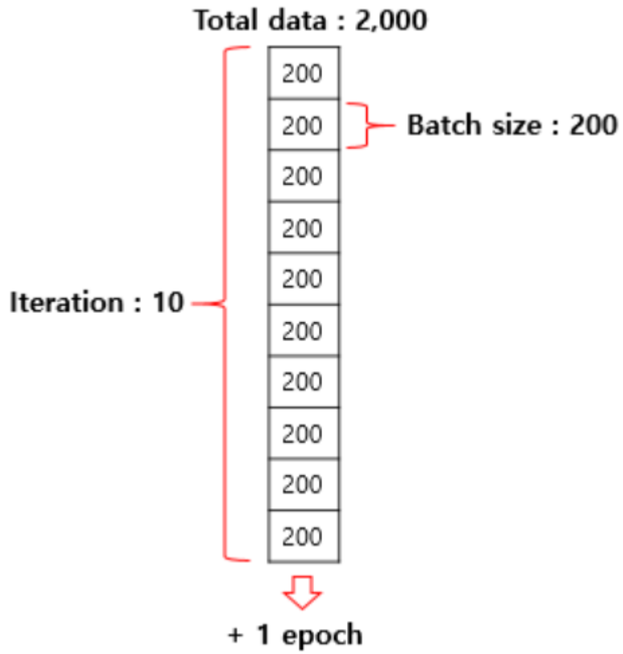
3.3 미니배치 경사하강법 / iteration

SGD는 연산량이 대폭 감소 but, 한 번에 하나의 데이터 포인트만 잡아서 Gradient를 계산하면 noise가 크다 (정확도가 낮아진다).

따라서, (Batch) Gradient Descent(전체 데이터를 모두 이용) 와 Stochastic Gradient Descent 방식 두 가지의 절충안을 사용한 것이 **Mini Batch Stochastic Descent**이다.

이는 한번에 한 개의 데이터가 아닌, 여러 개의 데이터 포인트를 샘플링해서 Gradient를 계산하는 방법으로,

1) 한 번에 고려하는 데이터가 **SGD에 비해** 많아지므로 보다 **정확도가 높다.**
2) 모든 데이터를 한 번에 고려해서 계산하는 것이 아니므로 **(Batch) Gradient Descent보다 더 효율적**이다.



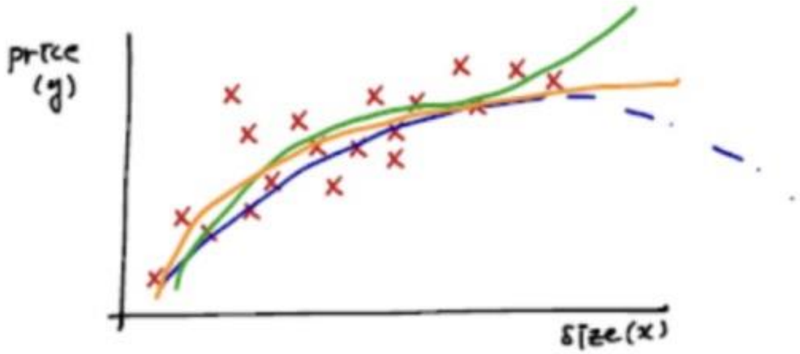
전체 데이터가 2,000일 때 배치 크기를 200으로 한다면 이터레이션의 수는 총 10개 이고 한 번의 에포크 당 매개변수 업데이트가 10번 이루어짐을 의미한다.

참고: 29. Python - 미니배치 경사 하강법 (tistory.com)

04. 다항회귀

EWHA EURON

4.1 다항회귀



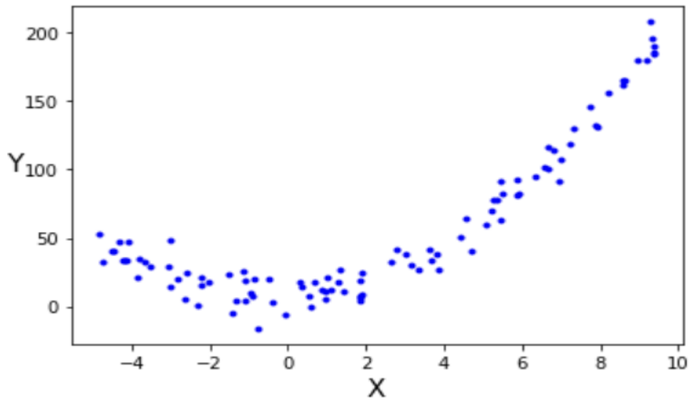
$\theta_0 + \theta_1 x + \theta_2 x^2$

$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$

$\theta_0 + \theta_1 x + \theta_2 \sqrt{x}$

$$h_\theta(x) = \theta_0 + \theta_1 \times (\text{frontage}) + \theta_2 \times (\text{depth})$$

$$x_1 = (\text{area}) = (\text{frontage}) \times (\text{depth})$$



오른쪽 그림처럼 가지고 있는 데이터를 직선으로 예측하기 힘들다면?
⇒ Polynomial regression 사용

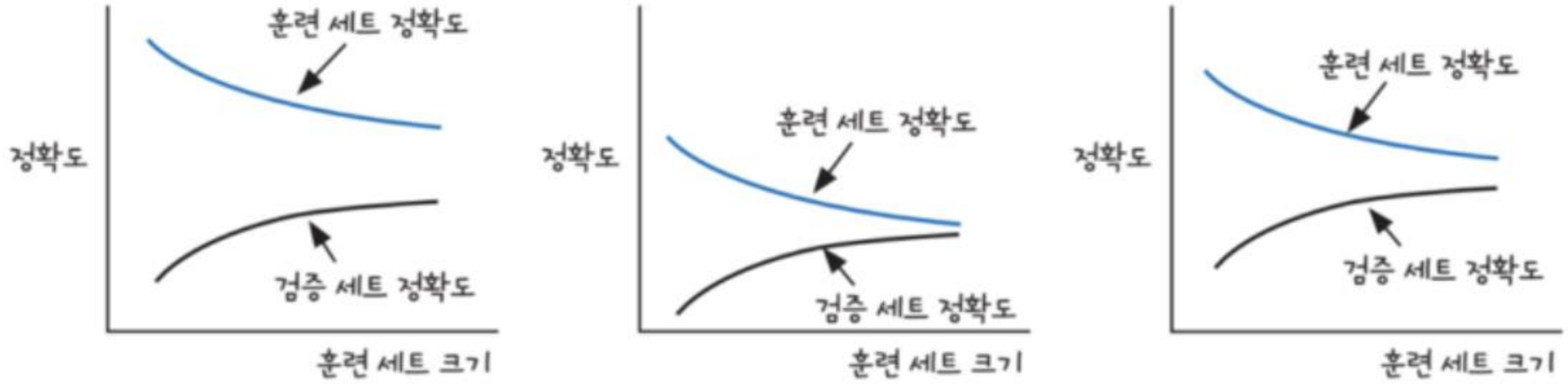
- 비선형 데이터 학습에 선형 모델을 사용할 수 있음
→ 각 특성의 거듭제곱을 새 특성으로 추가한 데이터셋에 선형 모델을 훈련 시키는 기법
예) 집의 넓이 : 집의 가로 길이 * 집의 세로 길이

** 다항 회귀 역시 선형 회귀인데, 선형과 비선형을 나누는 기준은 회귀 계수가 선형인지에 따른 것이기 때문이다.
** 입력변수가 여러 개인 다중 회귀와 다름!!

05. 편향-분산 트레이드오프

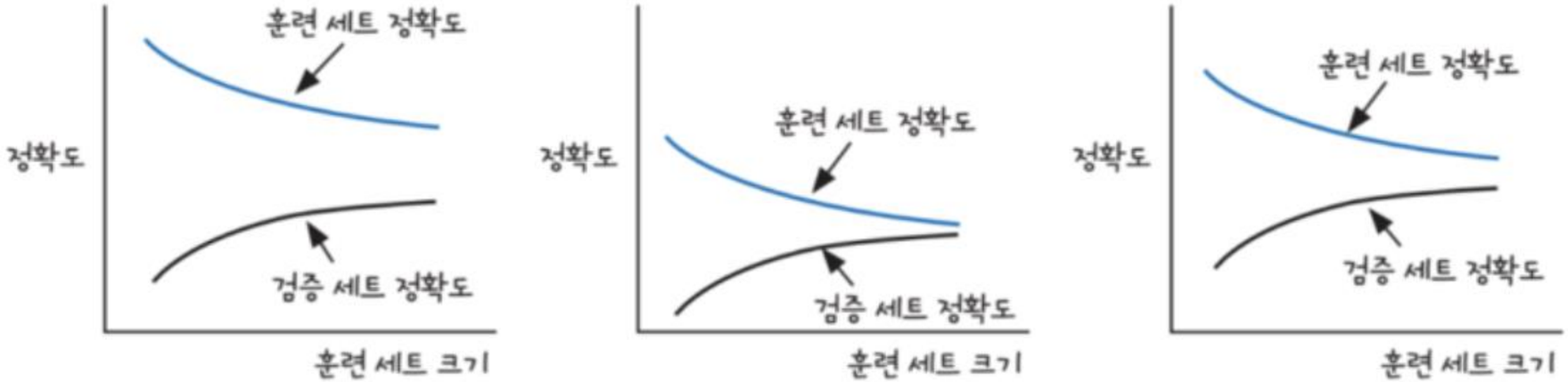
EWHA EURON

5.1 과대적합



- 머신러닝 모델을 학습할 때 학습 데이터셋에 지나치게 최적화하여 발생하는 문제
- 모델을 지나치게 복잡하게 학습하여 학습 데이터셋에서는 모델 성능이 높게 나타나지만 정작 새로운 데이터가 주어졌을 때 정확한 예측을 수행하지 못 함
- 훈련 세트와 검증 세트 간의 간격이 큼
- 훈련 세트에 충분히 다양한 패턴의 샘플이 포함되지 않은 경우 검증 세트에서 제대로 적응하지 못 함
  -> 이런 경우는 훈련세트의 샘플을 더 모으거나 가중치를 제한하며 성능을 향상

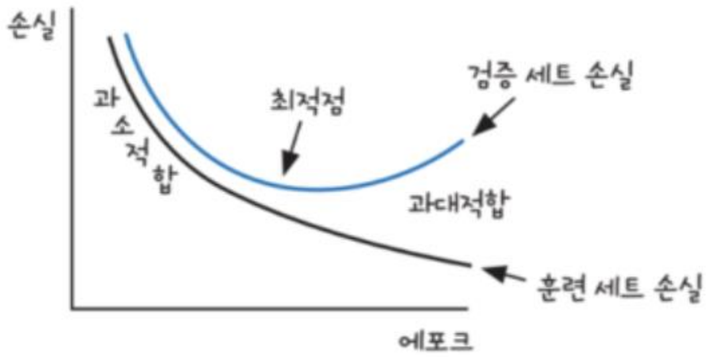
5.2 과소적합

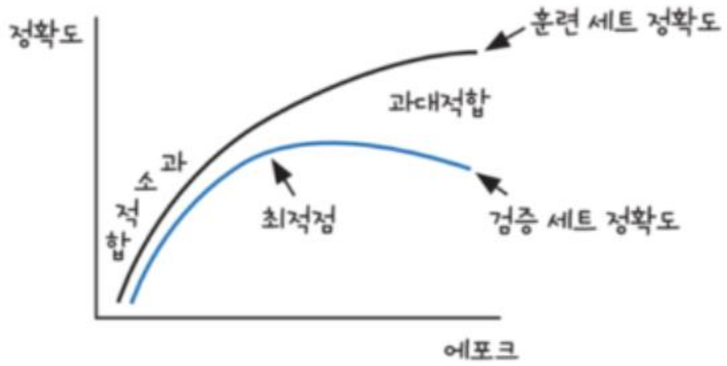


- 머신러닝 모델이 충분히 복잡하지 않아(최적화가 제대로 수행되지 않아) 발생하는 문제
- 학습 데이터의 패턴을 정확히 반영하지 못 함
- 훈련 세트와 검증 세트 간 간격이 가깝지만 정확도 자체가 낮음
- 복잡도가 더 높은 모델을 사용하거나 가중치의 규제를 완화하여 성능을 향상

5.3 에포크 손실 함수

- 에포크에 대한 손실함수를 이용하여 과대적합과 과소적합 분석





- 훈련 세트의 손실은 에포크가 진행될수록 감소하지만, 검증 세트 손실은 최적점을 지나면 증가

- 최적점 이후에도 훈련 세트로 학습시키면 모델이 과대적합되기 시작
- 최적점 이전에는 훈련 세트와 검증 세트 손실이 비슷한 간격으로 감소하는데, 이때 학습을 중지하면 과소적합된 모델이 나옴

- 에포크 대신 모델 복잡도는 넣어도 같은 그래프 결과를 보임

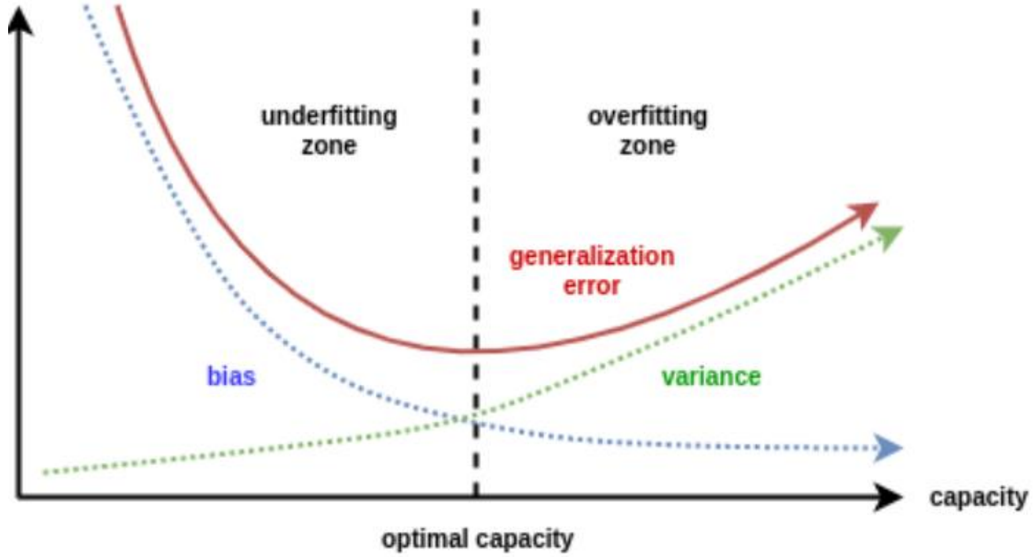
5.4 편향-분산 트레이드 오프

$$\mathrm{E}\left[\left(y-\hat{f}\left(x\right)\right)^{2}\right]=\mathrm{Bias}\left[\hat{f}\left(x\right)\right]^{2}+\mathrm{Var}\left[\hat{f}\left(x\right)\right]+\sigma^{2}$$

Error ⇒ 편향 제곱+ 분산 + 불가피한 오차로 구성

- 편향 : 예측 값과 실제 값 간의 차이
  → 과소적합되기 쉬움
  → 모델이 무언가 놓치고 있음

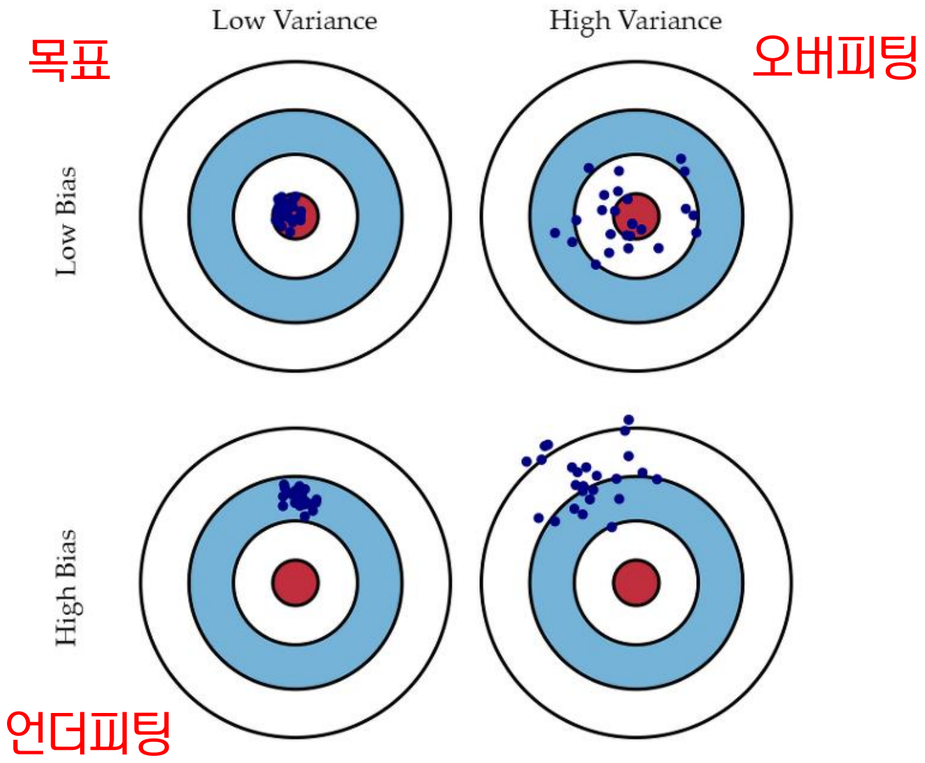
- 분산 : 예측 값끼리 흩어진 정도
  → 훈련 데이터의 변동에 모델이 과도하게 반응
  → 과대적합되기 쉬움
  → 모델이 일반화 되지 않은 상태



- 줄일 수 없는 오차 : 데이터 자체 잡음으로 인해

- 모델의 복잡도가 커지면 분산이 늘어나고 편향은 줄어든다
  반대로 복잡도가 줄어들면 편향이 커지고 분산은 감소하기 때문에 트레이드 오프라고 부름

5.4 편향-분산 트레이드 오프



목표

오버피팅

언더피팅

Fig. 1 Graphical illustration of bias and variance.

구분	Flexibility	Fitting
Low Bias High Variance	Flexible	Overfitting
High Bias Low Variance	Inflexible	Underfitting

06. 규제

EWHA EURON

6.1 규제

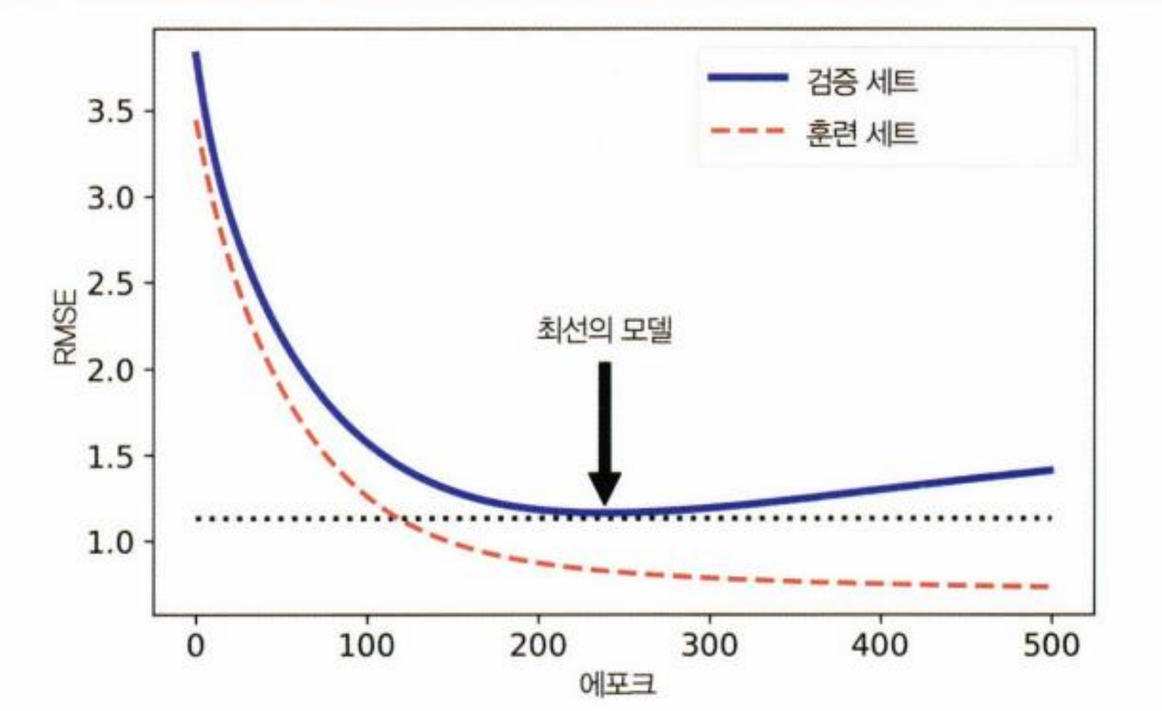
- RSS(Residual Sum of Squares) 를 최소화하는 방향만 고려하다 보면, 학습 데이터에 지나치게 맞추게 되어 회귀 계수가 커지는 문제가 발생
  ⇒ 이로 인해 테스트 셋에서의 예측 성능이 저하됨

- 따라서 RSS 최소 + 회귀계수 커지는 문제 방지 둘 다 만족하는 균형점을 맞춰야 함

- 비용함수: Min(RSS(W) + alpha * W노름 제곱 (L2기준)) 되게끔
  - 알파 값을 크게 하면 비용함수가 회귀계수 W를 작게 하여 과적합 개선
  - 알파 값 작게 하면 회귀 계수가 커져도 비용함수의 증감이 줄어드는 효과가 있어 학습데이터 적합이 개선

⇒ 규제: 비용함수에 알파 값으로 패널티를 부여해 회귀 계수 값 크기를 감소시켜 과적합 개선

6.2 조기 종료



- 과대적합을 방지하는 대표적 방법
- 특정 Epoch 내 validation loss가 감소하지 않으면 과대적합이 발생했다고 간주하고 학습을 종료함으로써 validation loss가 가장 낮은 지점인 최적(optimal) epoch에서의 모델을 저장해 주는 기법

6.2 조기 종료

```
from copy import deepcopy

#데이터 준비
poly_scaler = Pipeline([
    ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
    ("std_scaler", StandardScaler())
])

X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True, penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train) #훈련을 이어서 진행
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = deepcopy(sgd_reg)
```

- warm_start =True로 지정하면, fit() 메서드가 호출될 때, 처음부터 다시 시작하지 않고 이전 모델 파라미터에서 훈련을 이어간다.

6.3 릿지

- 선형 회귀에 L2 규제를 추가한 회귀 모델
- W제곱에 대해 패널티
- L2 규제 : 상대적으로 큰 회귀 계수 값의 예측 영향도를 감소시키기 위해서 회귀 계수 값을 더 작게 만드는 규제 모델

- 규제항을 비용함수에 추가
  → 가중치가 작게 유지되도록 훈련 과정에만 추가

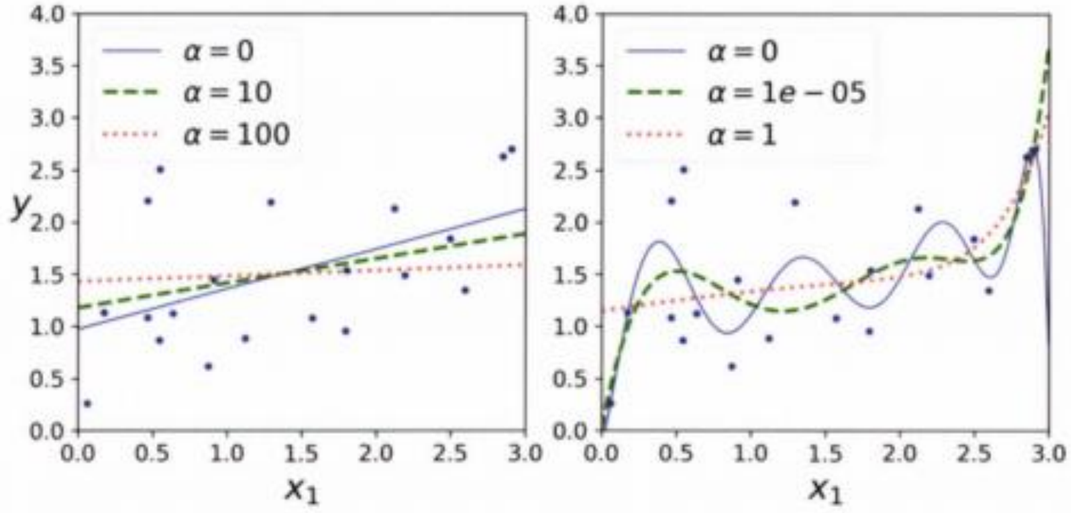
- 입력 특성의 스케일에 민감 → 스케일링 필수
- Feature scaling : 모든 피처 범위를 비슷하게 만들어 gradient descent가 더 빠르게 수렴하도록

- 알파를 증가시킬 수록 직선에 가까워짐 = 분산 감소 &편향 증가
- 제곱합을 하기 때문에 비용함수가 W의 영향을 더 많이 받게 되어 회귀 계수 크기를 비교적 많이 감소시킴

식 4-8 릿지 회귀의 비용 함수

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

$$||W||_2^2$$



6.4 라쏘

- 선형 회귀에 L1 규제를 추가한 회귀 모델
- W 절댓값에 대해 패널티
- L1 규제: 예측 영향력이 작은 피처의 회귀 계수를 0으로 만들어 회귀 예측 시, 피처가 선택되지 않게 함

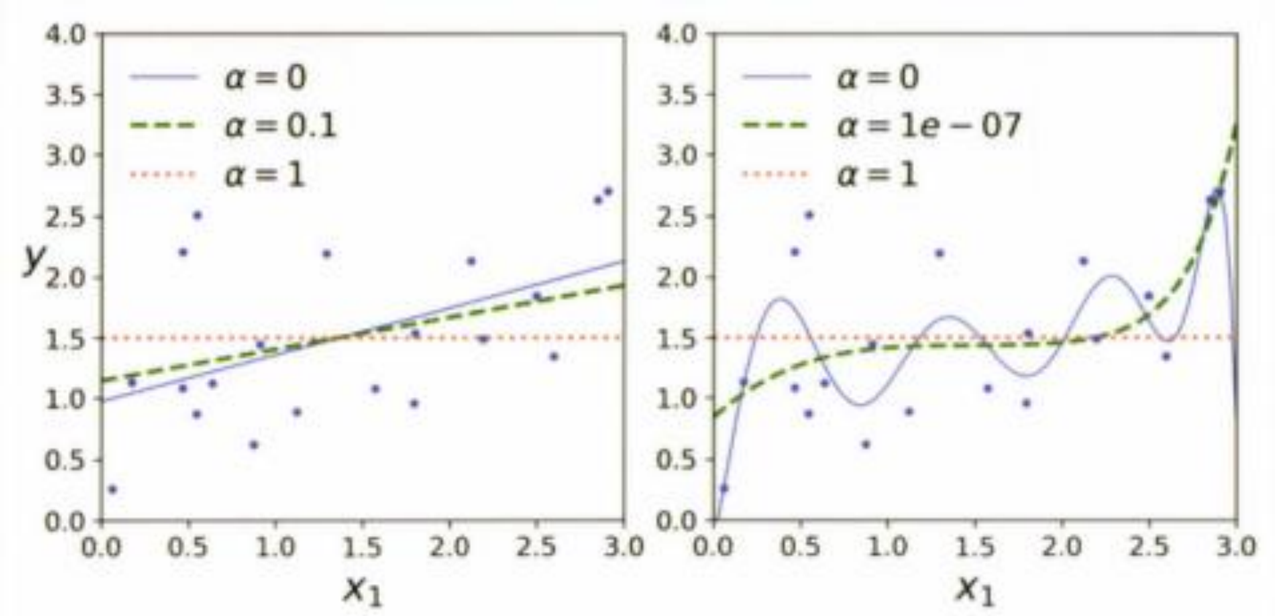
- 적절한 피처만 회귀에 포함 시킴 → 피처 선택 효과
- 덜 중요한 특성의 가중치 제거 → 희소 모델을 만든다!
  (0이 아닌 특성의 가중치가 적음)

- 알파를 증가시킬수록 선형에 가까움

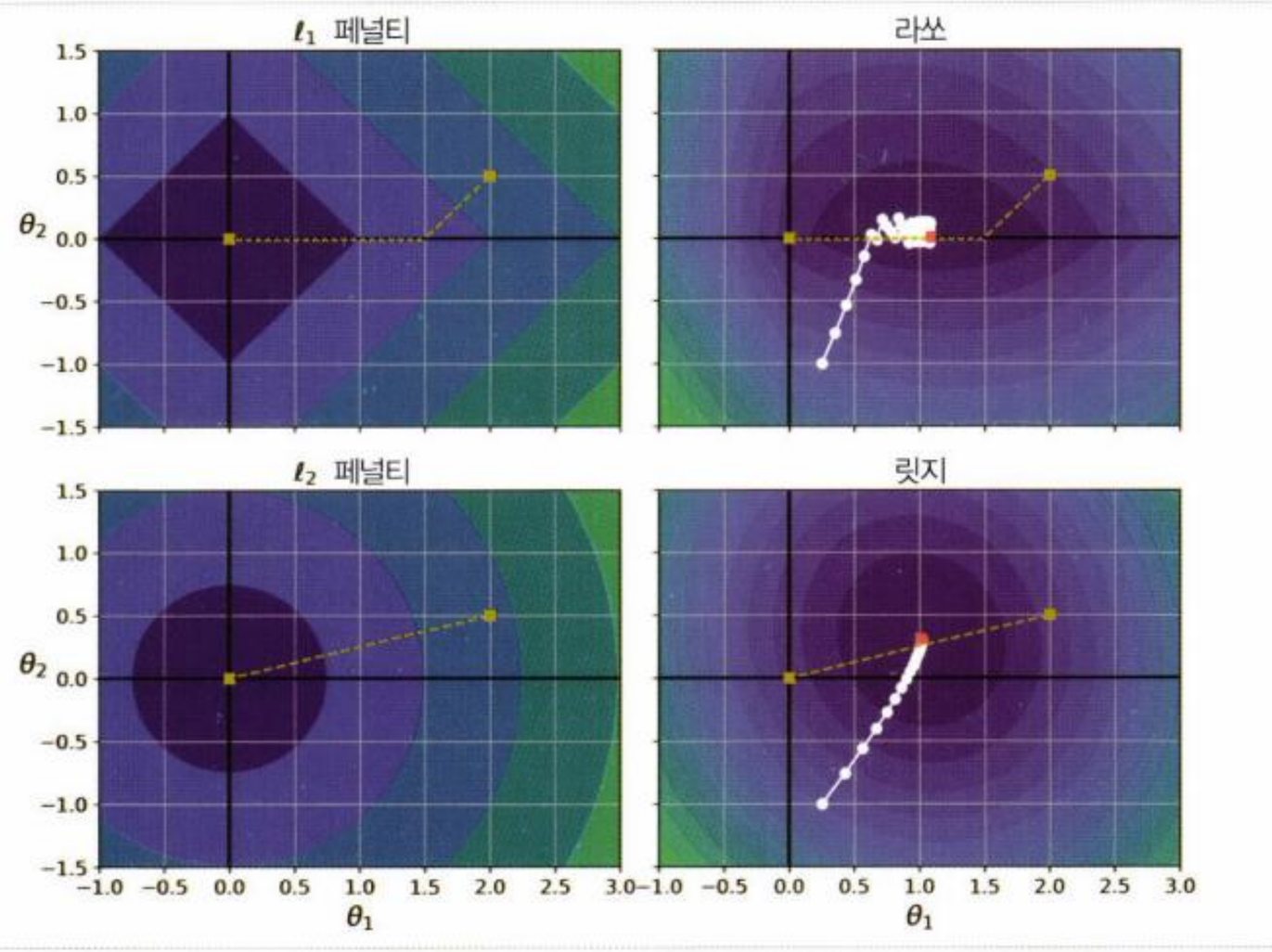
식 4-10 라쏘 회귀의 비용 함수

$$J(\theta) = \mathrm{MSE}(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

$$||W||_1$$



6.5 라쏘와 릿지



- 오른쪽의 두 그래프를 보면 비용함수에 규제항이 포함되어 있어서 등고선이 다르게 나타나며, 빨간색 네모로 표시된 최적 파라미터 값이 다른 것을 확인 가능

- 라쏘의 경우 최적값으로 가는 도중 지그재그로 흰색 선이 표시되는데 이는 θ2가 0에서 갑자기 변했기 때문

6.6 엘라스틱 넷

- L1과 L2를 결합한 모델.
- 피처가 많은 데이터 셋에서 L1 규제로 피처 개수 줄이고,
  L2 규제로 계수 값 크기 조정

- r=0이면 릿지 회귀, r=1이면 라쏘 회귀

[장점]
: 라쏘 회귀에서 피처 선택으로 인한 회귀 계수의 변동 문제를 L2 규제를 통해 완화 가능

[단점]
: 두 규제 방법이 결합되어 수행 시간이 오래 걸림

[파라미터]
엘라스틱 넷 규제항 ⇒ a*L1 + b*L2라고 표현한다면
- alpha : L1과 L2 규제의 두 알파 값을 더한 것 (=a+b)
- l1_ratio : a/(a+b)

식 4-12 엘라스틱넷 비용 함수

$$J(\theta) = \mathrm{MSE}(\theta) + r\alpha\sum_{i=1}^{n}|\theta_i| + \frac{1-r}{2}\alpha\sum_{i=1}^{n}\theta_i^2$$

6.7 보스턴 예시

```
from sklearn.linear_model import Ridge,Lasso, ElasticNet

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

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

    return coeff_df
```

6.7 보스턴 예시

In [40]:
```
ridge_alphas = [0 , 0.1 , 1 , 10 , 100]
coeff_ridge_df =get_linear_reg_eval('Ridge', params=ridge_alphas, X_data_n=X_data, y_target_n=y_target)
```

```
#######  Ridge #######
alpha 0일 때 5 폴드 세트의 평균 RMSE: 5.829
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.788
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.653
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.518
alpha 100일 때 5 폴드 세트의 평균 RMSE: 5.330
```

In [38]:
```
lasso_alphas = [ 0.07, 0.1, 0.5, 1, 3]
coeff_lasso_df =get_linear_reg_eval('Lasso', params=lasso_alphas, X_data_n=X_data, y_target_n=y_target)
```

```
#######  Lasso #######
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
```

In [39]:
```
# 엘라스틱넷에 사용될 alpha 파라미터의 값들을 정의하고 get_linear_reg_eval() 함수 호출
# l1_ratio는 0.7로 고정
elastic_alphas = [ 0.07, 0.1, 0.5, 1, 3]
coeff_elastic_df =get_linear_reg_eval('ElasticNet', params=elastic_alphas,X_data_n=X_data, y_target_n=y_target)
```

```
#######  ElasticNet #######
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
```

6.8 선형 회귀 모델을 위한 데이터 변환 방법

1. StandardScaler/MinMaxScaler 을 이용해서 정규 분포 형태로 만들기
 → 예측 향상 기대 적음

2. 1번 방법이 성능 향상에 효과가 없다면 스케일링과 정규화한 데이터에 다시 다항 특성 적용하기
 → 피처가 많으면 과적합 문제, 시간 오래 걸림

3. 로그 변환을 취해 정규 분포에 가깝게 만들기
 → 주로 사용함! 심하게 왜곡된 데이터에 좋음

07. 로지스틱 회귀 (Logistic Regression)

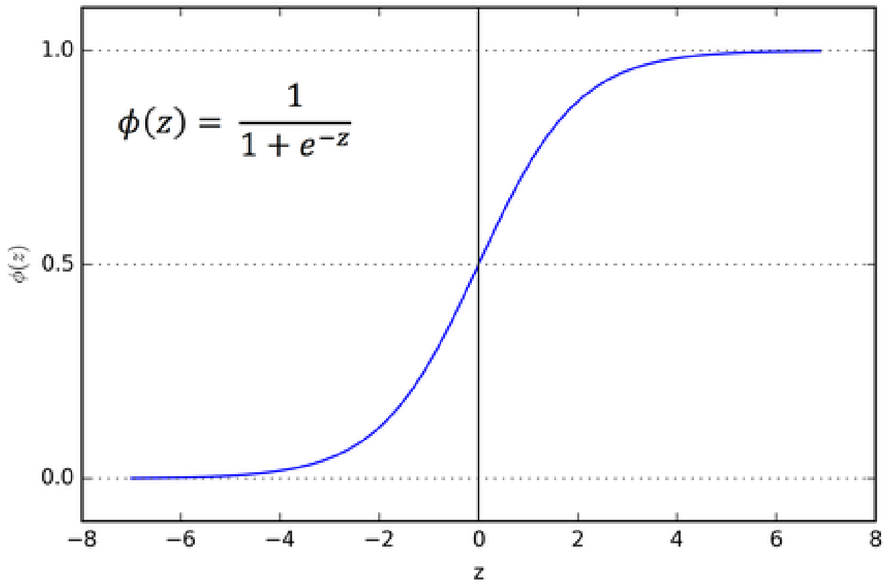
EWHA EURON

7.1 로지스틱 회귀

로지스틱 회귀란?

- 선형 회귀 방식을 **분류에 적용**한 알고리즘

- **시그모이드(Sigmoid) 함수의 최적선**을 찾고 이 시그모이드 함수의 반환 값을 확률로 간주해 **확률에 따라 분류를 결정**

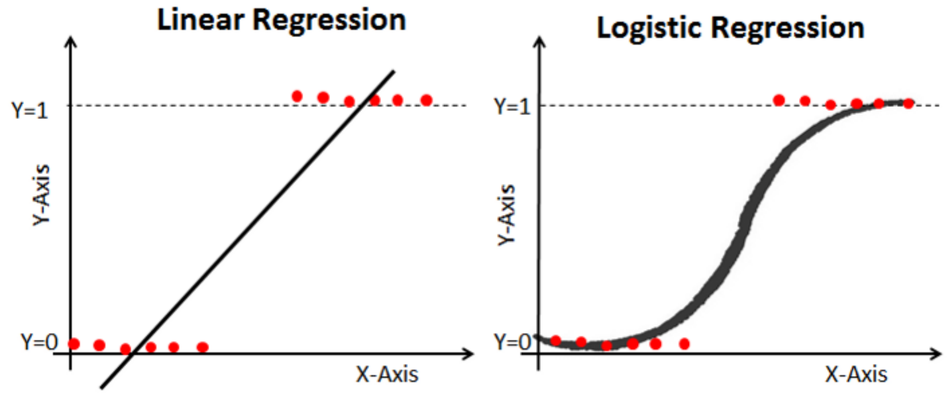


시그모이드 함수의 그래프와 수식

→ 시그모이드 함수는 x 값이 +, -로 아무리 커지거나 작아져도 y값은 항상 0과 1 사이 값 반환

→ x값이 커지면 1에 근사, x값이 작아지면 0에 근사함, x=0이면 0.5

→ 독립변수가 어떻든 결과값이 항상 [0,1]사이에 있도록 하는 것이 시그모이드 함수



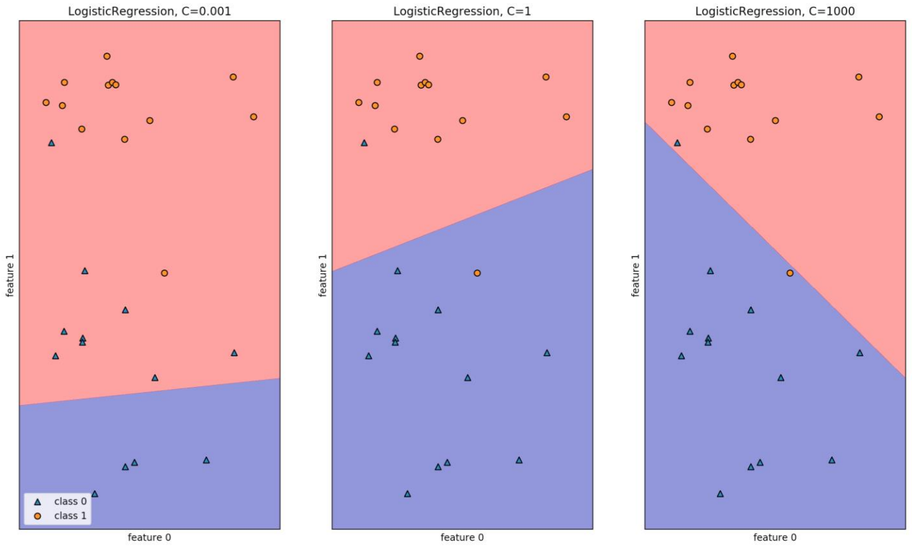
시그모이드와 선형 회귀

7.2 위스콘신 유방암 분류 – 로지스틱 회귀

LogisticRegression 클래스의 주요 하이퍼 파라미터

1. penalty : 규제의 유형을 설정, default는 l2(L2 규제), l1으로 설정 가능

2. C : 규제 강도를 조절하는 alpha값의 역수, C = 1/alpha




```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

cancer = load_breast_cancer()
#StandardScaler()로 평균이 0, 분산이 1인 데이터 분포로 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=123)

from sklearn.metrics import accuracy_score, roc_auc_score

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

#정확도와 roc_auc 측정
print('accuracy:', accuracy_score(y_test, lr_preds))
print('roc_auc:', roc_auc_score(y_test, lr_preds))
```

accuracy: 0.9941520467836257
roc_auc: 0.9926470588235294

```
from sklearn.model_selection import GridSearchCV

params = {'penalty':['l2', 'l1'],
          'C':[0.01, 0.1, 1, 1.5, 5, 10]}
grid_clf = GridSearchCV(lr_clf, param_grid = params, scoring='accuracy', cv=3)
grid_clf.fit(data_scaled, cancer.target)
print('최적 하이퍼 파라미터:', grid_clf.best_params_,
      '최적 평균 정확도:', grid_clf.best_score_)
```

⇒ 로지스틱 회귀는 가볍고, 빠르고, 이진 분류 예측 성능이 뛰어남
(희소한 데이터 세트 분류에도 뛰어난 성능 → 텍스트 분류에서도 자주 사용)

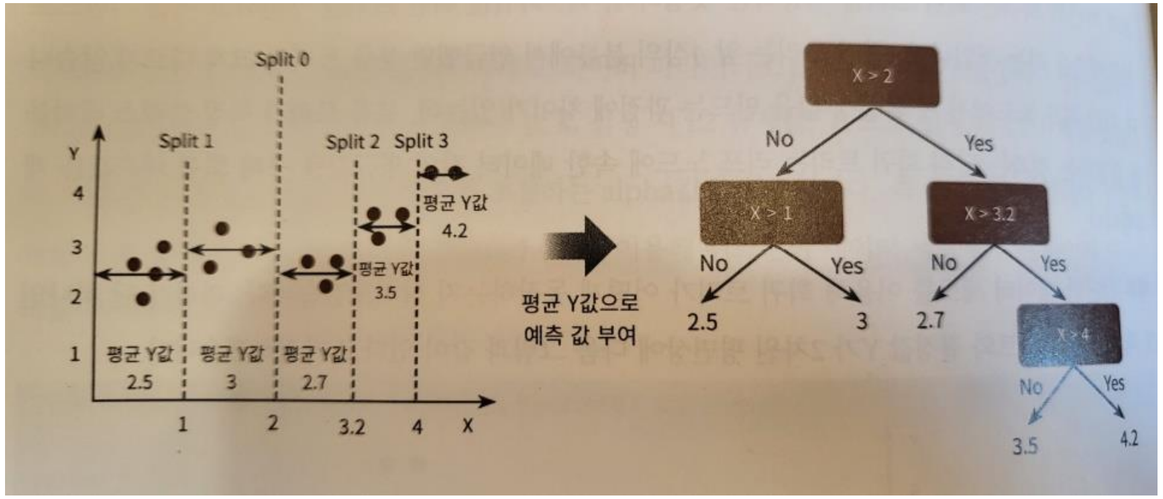
08.회귀 트리

EWHA EURON

8.1 회귀 트리

회귀 트리란?

- 회귀를 위한 트리를 생성하고 이를 기반으로 회귀 예측을 하는 알고리즘
- 분류 트리가 특정 클래스 레이블을 결정하는 것과 달리 **회귀 트리는 리프 노드에 속한 데이터 값의 평균값을 구해 회귀 예측값을 계산함.**



<회귀 트리의 동작 방법>

1. X 피처를 결정 트리 기반으로 분할 → X값의 균일도를 반영한 지니 계수에 따라 Split
2. 루트 노드를 Split 0 기준으로 분할하고 분할된 규칙 노드에서 다시 Split 1, Split 2 규칙노드로 분할 → 재귀적으로 Split 2는 Split 3 규칙 노드로 분할됨
3. 리프 노드 생성 기준에 부합하는 트리 분할이 완료되면 리프 노드에 소속된 데이터 값의 평균값을 구해 최종적으로 리프 노드에 결정 값으로 할당됨.

8.2 CART

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

모든 **트리 기반의 알고리즘**은 분류뿐만 아니라 회귀도 가능하다!
(결정 트리, 랜덤 포레스트, GBM, XGBoost, LightGBM 등)

⇒ 트리 생성이 모두 CART 알고리즘에 기반하고 있기 때문

CART(Classification And Regression Trees)

분류와 회귀에서 모두 사용할 수 있는 의사결정나무 알고리즘

지니계수, RSS 등을 기준으로 불순도를 계산하여 노드를 분할해나감

8.3 보스턴 주택 가격 예측 – RandomForestRegressor

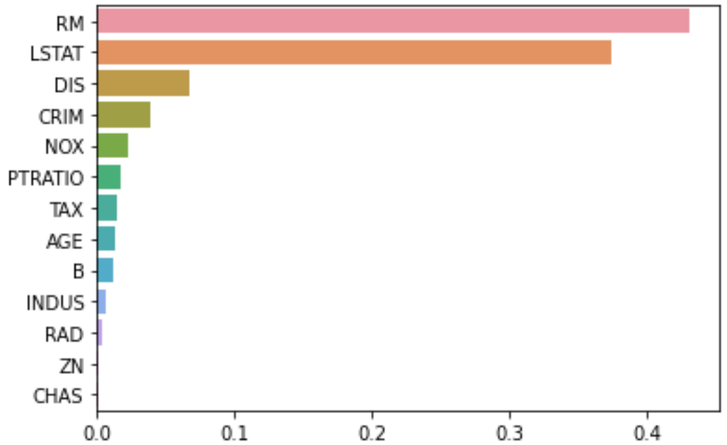
```
from sklearn.datasets import load_boston
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor
import pandas as pd
import numpy as np

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=123, n_estimators=1000)
neg_mse_scores = cross_val_score(rf, X_data, y_target, scoring='neg_mean_squared_error', c
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, 3))
print('5 교차 검증의 개별 RMSE scores:', np.round(rmse_scores, 3))
print('5 교차 검증의 평균 RMSE:', np.round(avg_rmse, 3))
```

```
5 교차 검증의 개별 Negative MSE scores: [ -7.89  -12.854 -20.058 -46.339 -18.594]
5 교차 검증의 개별 RMSE scores: [2.809 3.585 4.479 6.807 4.312]
5 교차 검증의 평균 RMSE: 4.398
```

→ 랜덤포레스트 뿐만 아니라 결정 트리, GBM, XGBoost, LightGBM의 Regressor를 모두 이용할 수 있음

주의! ⚠

회귀 트리 Regressor 클래스는 선형회귀와 다른 처리 방식이므로 회귀 계수를 제공하는 coef_ 속성이 없다!

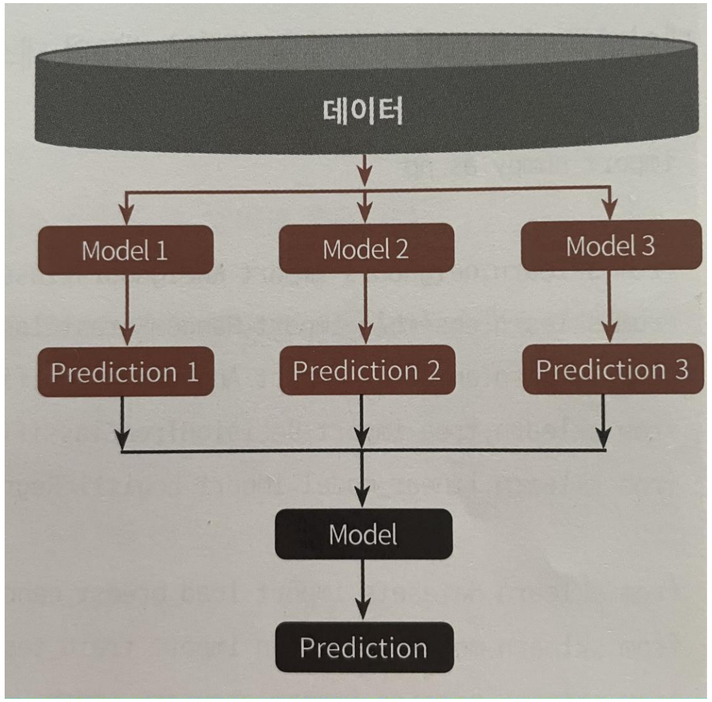


→ feature_importances_ 메서드를 이용해 피처별 중요도를 알 수 있음

09. 스태킹 앙상블 모델을 통한 회귀 예측

EWHA EURON

9.1 스태킹 모델의 구현 방법



1. 개별적인 기반 모델
2. 1번 개별 기반 모델의 예측 데이터를 학습 데이터로 만들어 학습하는 최종 메타 모델

⇒ 여러 개별 모델의 예측 데이터를 각각 스태킹 형태로 결합해 최종 메타 모델의 학습용 피처 데이터 세트와 테스트용 피처 데이터 세트를 만드는 것

1. 개별 기반 모델은 원래 사용되는 학습 데이터와 테스트용 피처 데이터를 인자로 입력 받음
2. 함수 내에서 개별 모델이 K-폴드 세트로 설정된 폴드 세트 내부에서 원본의 학습 데이터를 다시 추출해 학습과 예측을 수행 및 결과 저장
3. 저장된 예측 데이터는 추후에 메타 모델의 학습 피처 데이터 세트로 이용
4. 함수 내에서 폴드 세트 내부 학습 데이터로 학습된 개별 모델이 인자로 입력된 원본 테스트 데이터를 예측한 뒤, 예측 결과를 평균해 테스트 데이터로 생성

```python
#스태킹 앙상블
from sklearn.model_selection import KFold
from sklearn.metrics import mean_absolute_error

#개별 기반 모델에서 최종 메타 모델이 사용할 학습 및 테스트용 데이터를 생성하기 위한 함수
def get_stacking_base_datasets(model, X_train_n, y_train_n, X_test_n, n_folds):
    #지정된 n_folds 값으로 KFold 생성
    kf = KFold(n_splits=n_folds, shuffle=False, random_state=123)
    #추후에 메타 모델이 사용할 학습 데이터 반환을 위한 넘파이 배열 초기화
    train_fold_pred = np.zeros((X_train_n.shape[0],1))
    test_pred = np.zeros((X_test_n.shape[0],n_folds))
    print(model.__class__.__name__, 'model 시작')

    for folder_counter, (train_index, valid_index) in enumerate(kf.split(X_train_n)):
        #입력된 학습 데이터에서 기반 모델이 학습, 예측할 폴드 데이터 세트 추출
        print('폴드 세트:', folder_counter, '시작')
        X_tr = X_train_n[train_index]
        y_tr = y_train_n[train_index]
        X_te = X_train_n[valid_index]
        #폴드 세트 내부에서 다시 만들어진 학습 데이터로 기반 모델의 학습 수행
        model.fit(X_tr, y_tr)
        #폴드 세트 내부에서 다시 만들어진 검증 데이터로 기반 모델 예측 후 데이터 저장
        train_fold_pred[valid_index, :] = model.predict(X_te).reshape(-1,1)
        #입력된 원본 테스트 데이터를 폴드 세트 내 학습된 기반 모델에서 예측 후 데이터 저장
        test_pred[:,folder_counter] = model.predict(X_test_n)

    #폴드 세트 내에서 원본 테스트 데이터를 예측한 데이터를 평균하여 테스트 데이터로 생성
    test_pred_mean = np.mean(test_pred, axis=1).reshape(-1,1)
    #train_fold_pred는 최종 메타 모델이 사용하는 학습 데이터, test_pred_mean은 테스트 데이터
    return train_fold_pred, test_pred_mean
```

9.2 보스턴 주택 가격 예측 – 스태킹 활용

적용할 개별 모델: 릿지, 라쏘, XGBoost, LightGBM

```
#보스턴 집값
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)
X_train, X_test, y_train, y_test = train_test_split(X_data, y_target, test_size=0.2, random_s

X_train_n = X_train.values
X_test_n = X_test.values
y_train_n = y_train.values

from sklearn.linear_model import Ridge, Lasso
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
ridge_reg = Ridge(alpha=8)
lasso_reg = Lasso(alpha=0.001)
xgb_reg = XGBRegressor(n_estimators=1000, learning_rate=0.05)
lgbm_reg = LGBMRegressor(n_estimators=1000, learning_rate=0.05)
#각 개별 기반(base) 모델이 생성한 학습용, 테스트용 데이터 반환
ridge_train, ridge_test = get_stacking_base_datasets(ridge_reg, X_train_n, y_train_n, X_test_
lasso_train, lasso_test = get_stacking_base_datasets(lasso_reg, X_train_n, y_train_n, X_test_
xgb_train, xgb_test = get_stacking_base_datasets(xgb_reg, X_train_n, y_train_n, X_test_n, 5)
lgbm_train, lgbm_test = get_stacking_base_datasets(lgbm_reg, X_train_n, y_train_n, X_test_n,
```

```
#개별 모델이 반환한 학습 및 테스트용 데이터 세트를 스태킹 형태로 결합
Stack_final_X_train = np.concatenate((ridge_train, lasso_train, xgb_train, lgbm_train), axis=1
Stack_final_X_test = np.concatenate((ridge_test, lasso_test, xgb_test, lgbm_test), axis=1)

#최종 메타 모델은 라쏘 모델 적용
meta_model_lasso = Lasso(alpha=0.0005)

#개별 모델 예측값을 기반으로 새롭게 만들어진 학습, 테스트 데이터로 메타 모델 예측 및 RMSE 측정
meta_model_lasso.fit(Stack_final_X_train, y_train)
final = meta_model_lasso.predict(Stack_final_X_test)
mse = mean_squared_error(y_test, final)
rmse = np.sqrt(mse)
print('스태킹 회귀 모델의 최종 RMSE 값은:', rmse)
```

스태킹 회귀 모델의 최종 RMSE 값은: 3.961432796069961

→ 지금까지 했던 모델 중 가장 예측 성능이 뛰어남

10. 파이프라인(Pipeline)

EWHA EURON

10.1 파이프라인의 필요성

파이프 라인(Pipeline)이란?

- 연속된 변환을 순차적으로 처리할 수 있는 기능을 제공하는 래퍼(Wrapper)도구

🦁파이프라인을 사용하지 않고 전처리 및 모델 fitting하는 단계

```
#Pipeline 사용하지 않고 전처리 및 모델 fitting
#우리가 할 전처리는 PolynomialFeatures
X_data = bostonDF.drop(['PRICE'], axis=1, inplace=False)
y_target = bostonDF['PRICE']

X_train, X_test, y_train, y_test = train_test_split(X_data, y_target,
                                                    test_size=0.3, random_state=123)

#PolynomialFeatures 전처리
poly = PolynomialFeatures(degree=2, include_bias=False)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)
#선형 회귀 OLS로 학습, 예측, 평가 수행
lr = LinearRegression()
lr.fit(X_poly, y_train)
preds = lr.predict(X_test_poly)
mse = mean_squared_error(y_test, preds)
rmse = np.sqrt(mse)

print('MSE:', mse, 'RMSE:', rmse)
print('Variance Score:', r2_score(y_test, preds))
print('절편 값:', lr.intercept_)
print('회귀 계수 값:', np.round(lr.coef_,1))
```

→ 이 예시에선 전처리를 하나밖에 하지 않았지만 원-핫 인코딩, Scaler 등 넣으면 코드가 더 길고 가독성도 안 좋고 복잡 해질것!

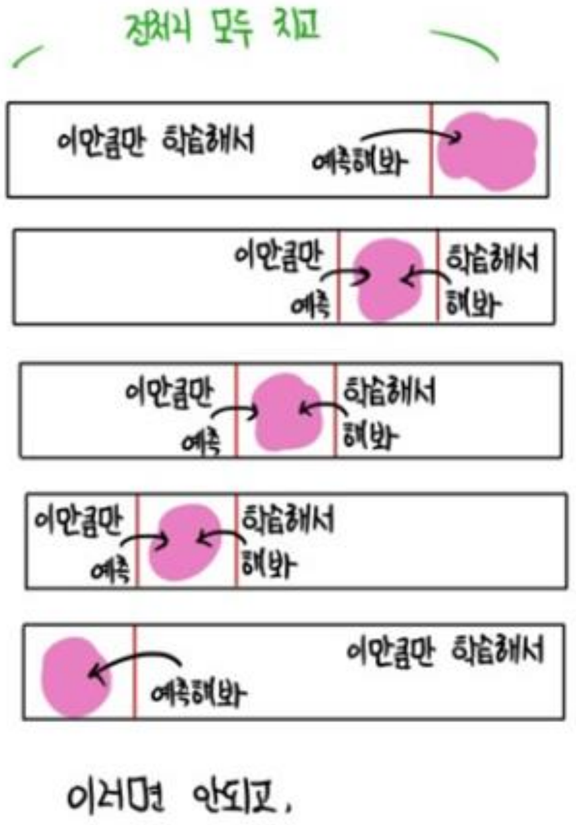
→ 실제 데이터에서 전처리 하고.. GridSearchCV 해서 최적 파라미터도 찾고 예측 성능도 높이고 싶은데 어쩌지?

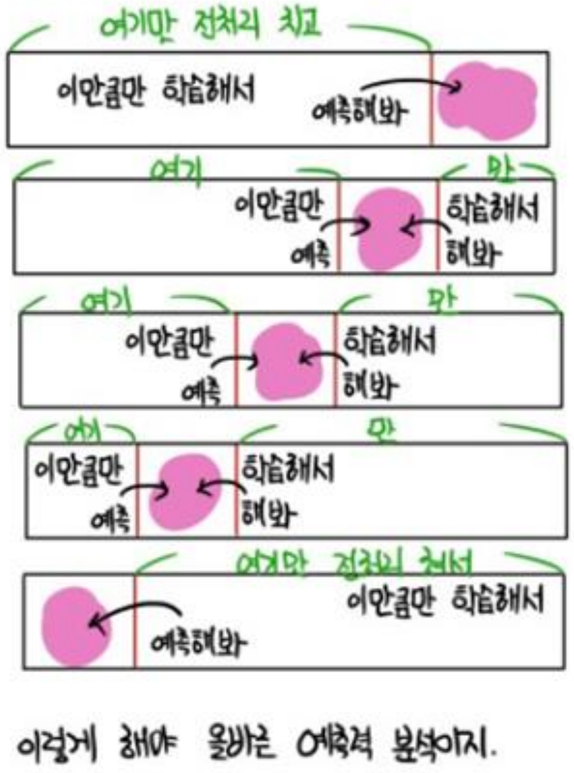
⇒ 이것이 파이프라인(Pipeline)의 필요성! (간단하고 가독성 좋게 코딩하고, 전처리와 동시에 GridSearchCV도 사용하고!)

https://mindsee-ai.tistory.com/61
https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=gdpresent&logNo=221730873049

10.1 파이프라인의 필요성

✚ 실제 데이터에서 X_train 데이터 전부를 전처리한 후 교차검증을 수행하면 안되는 이유





→ 우리가 train, test set를 나누어 스케일링을 진행하는 것처럼, 교차 검증 단계에서도 전체 학습 데이터를 전처리하고 그를 교차 검증하면 안됨!(미래 참조가 되어 버림)

→ 교차 검증 시 학습 데이터만 전처리하고, 검증하는 데이터는 전처리가 되지 않은 데이터여야 함! (어떻게 코딩을…)

⇒ **파이프라인(Pipeline)을 이용하자!!**

10.2 파이프라인의 생성 규칙

pipe = Pipeline(steps=[('첫번째 변환기 클래스 객체 이름', 객체), ('두번째 변환기 클래스 객체 이름', 객체)...])

→ 일의 진행 순서대로 튜플로 넣어주기
(변환기 클래스 객체 이름은 자기가 원하는 이름 넣어줘도 됨)

- 파이프라인에서 마지막 객체를 제외한 나머지 객체들은 transform, fit_transform 메소드를 제공하는 변환기만 허용(scaler, PolynomialFeatures와 같은 것들)
  맨 마지막 객체는 predict 매서드가 있는 객체가 들어와야 함!(LinearRegression과 같은 estimator들)

```
#Pipeline 사용하기
from sklearn.pipeline import Pipeline
X_data = bostonDF.drop(['PRICE'], axis=1, inplace=False)
y_target = bostonDF['PRICE']
X_train, X_test, y_train, y_test = train_test_split(X_data, y_target, test_size=0.3,
                                                    random_state=123)

poly = PolynomialFeatures(degree=2, include_bias=False)
model = LinearRegression()
pipe = Pipeline([('my_poly', poly),
                 ('my_model', model)])
#첫번째 객체는 fit_transform()가능 객체, 마지막 객체는 predict 가능 객체
pipe.fit(X_train, y_train)
print('테스트 평가:', pipe.score(X_test, y_test))
```

→ pipe에 'my_poly' 라는 전처리 클래스와 'my_model' 이라는 선형회귀 모델이 장착됨
(첫번째 객체는 fit_transform()가능 객체, 마지막 객체는 predict 가능 객체)
→ 그 pipe에 X_train과 y_train을 fit 시켜주면 'my_poly' 를 만나 fit&transform이 되고, 변환된 train 데이터가 'my_model' 을 만나 fit이 된 상태
→ pipe에 score 메서드까지 써주면 predict까지 해주고 평가 지표 도출(R2 score)

10.3 파이프라인의 실행 과정

```
#Pipeline 사용하기
from sklearn.pipeline import Pipeline
X_data = bostonDF.drop(['PRICE'], axis=1, inplace=False)
y_target = bostonDF['PRICE']
X_train, X_test, y_train, y_test = train_test_split(X_data, y_target, test_size=0.3,
                                                    random_state=123)

poly = PolynomialFeatures(degree=2, include_bias=False)
model = LinearRegression()
pipe = Pipeline([('my_poly', poly),
                 ('my_model', model)])
#첫번째 객체는 fit_transform()가능 객체, 마지막 객체는 predict 가능 객체
pipe.fit(X_train, y_train)
print('테스트 평가:', pipe.score(X_test, y_test))
```

1.입력된 X 데이터를 첫번째 변환기 클래스의 객체로 전달(fit 메소드 호출)

2.입력된 X 데이터를 transform 메소드를 통해 변환

3.변환된 입력 데이터 X를 다음에 위치한 변환기 or 예측기로 전달하여 fit 메소드 실행

4.다음 객체가 변환기인 경우 transform 메소드의 실행 결과를 반환하여 다음 객체로 전달하고, 예측기 클래스인 경우 실행 종료

10.4 GridSearchCV와 Pipeline

```
#linear regression은 파라미터가 없으니까 릿지 회귀로 모델 변경
from sklearn.pipeline import Pipeline
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import KFold, GridSearchCV

X_data = bostonDF.drop(['PRICE'], axis=1, inplace=False)
y_target = bostonDF['PRICE']
X_train, X_test, y_train, y_test = train_test_split(X_data, y_target, test_size=0.3,
                                                    random_state=123)

poly = PolynomialFeatures(degree=2, include_bias=False)
model = Ridge(random_state=123)
pipe = Pipeline([('my_poly', poly),
                 ('my_model', model)])

###GridSearchCV 코드
#pipe에서 지정한 모델 이름 뒤에 언더바 두개 + 파라미터명
kfold = KFold(n_splits=3, shuffle=True, random_state=123)
param_grids = {'my_model__alpha':[0.1,1,10],
               'my_model__max_iter':[100, 1000, 2000]
               }

grid = GridSearchCV(estimator=pipe,
                    param_grid=param_grids,
                    cv=kfold,
                    n_jobs=-1,
                     iid=True,
                    scoring='neg_mean_squared_error').fit(X_train, y_train)

mse = -1*grid.score(X_test, y_test)
rmse = np.sqrt(mse)

print('최적의 하이퍼 파라미터:', grid.best_params_)
print('RMSE:', rmse)
```

→ 파라미터 설정할 때 pipe에서 지정한 모델 이름 뒤에 _ 두개 + 파라미터명 형식

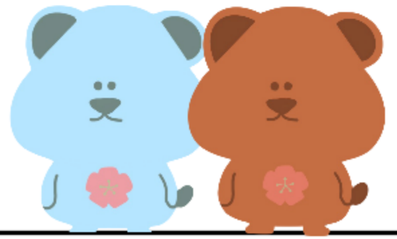
→ GridSearchCV의 estimator는 앞서 만들어두었던 pipe로 지정하기

→ scoring= ‘neg_mean_squared_error’ 로 지정하여 스코어값이 –MSE 값으로 나오도록 설정함 (음수 주의)

최적의 하이퍼 파라미터: {'my_model__alpha': 10, 'my_model__max_iter': 100}
RMSE: 4.134716956578526

EWHA EURON

THANK YOU



EWHA EURON