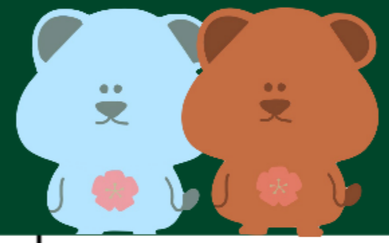


5주차 발표

소예림, 황선경, 김경민

EWHA EURON

목차

#01 회귀 소개

#02 단순 선형 회귀

#03 경사 하강법

#04 다항회귀

#05 편향-분산 트레이드 오프

#06 규제 선형 모델

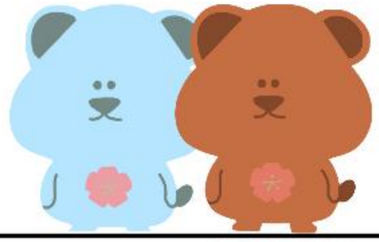
#07 로지스틱 회귀

#08 회귀 트리

#09 스태킹 앙상블 모델

#10 파이프라인

#11 학습곡선 해석



EWHA EURON

1. 회귀 소개

1.1 회귀 vs 분류

출처 : https://yhyun225.tistory.com/8

https://heung-bae-lee.github.io/2020/01/04/machine_learning_01/

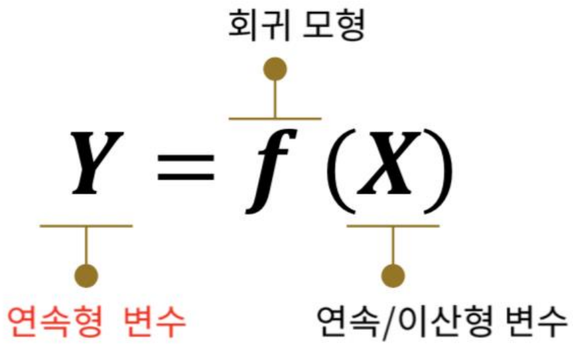
회귀와 분류의 공통점

- 지도 학습 : 정답이 있는 데이터를 활용해 모델을 학습시키는 방법

  => 별도의 테스트 데이터 세트에서 미지의 레이블을 예측

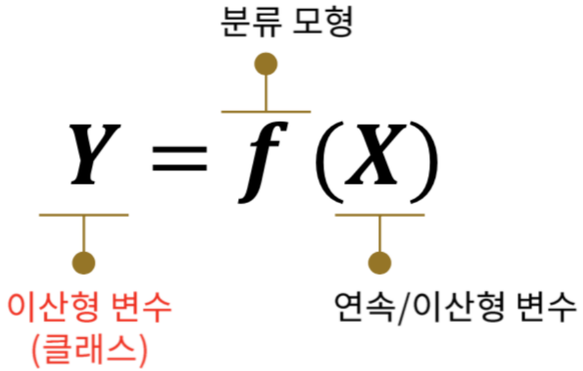
회귀(Regression)

- 예측하고자 하는 타겟값이 실수(숫자)인 경우
- 연속적인 예측 결과
- 손해액, 매출량, 거래량, 파산 확률 등 예측



분류(Classification)

- 예측하고자 하는 타겟값이 범주형 변수인 경우
- 이산적인 예측 결과
- 이진분류 / 다중분류



1.2 회귀

회귀(Regression)

- 여러 개의 독립변수와 한 개의 종속변수 간의 상관관계를 모델링하는 통계적 기법의 통칭
- 독립변수 -> 피처 / 종속변수 -> 결정값
- 선형 회귀식 : $Y = W_1 * X_1 + W_2 * X_2 + W_3 * X_3 + \cdots + W_n * X_n$
- W : 회귀 계수 => 독립변수에 영향을 미치는 최적의 회귀 계수를 탐색 !

회귀 유형

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

1.3 규제 선형 모델

https://steadiness-193.tistory.com/262

선형 회귀

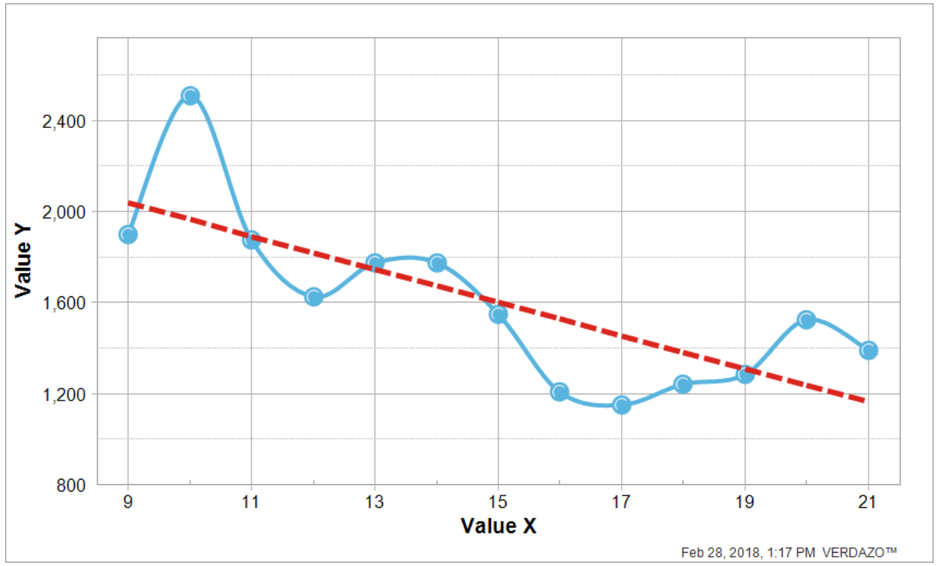
- 실제값과 예측값의 차이(오류의 제곱값)를 최소화하는 직선형 회귀선을 최적화하는 방식

규제

- 일반적인 선형 회귀의 과적합 문제를 해결하기 위해서 회귀 계수에 페널티 값을 적용하는 것
- 규제 방법에 따라 선형 회귀 모델을 별도의 유형으로 분류

오류의 계산

- Mean Absolute Error : 오류에 절댓값을 취해 더하는 방법
- RSS(Residual Sum of Square) : 오류의 제곱을 구해 더하는 방법
- $RSS(w_0, w_1) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - (w_0 + w_1 * x_i)\right)^2$
- RSS를 최소로 하는 회귀 계수를 학습을 통해 찾는 것이 핵심 !



1.3 규제 선형 모델

https://steadiness-193.tistory.com/262

 # 일반 선형 회귀

- 예측값과 실제 값의 RSS(Residual Sum of Squares)를 최소화할 수 있도록 회귀 계수 최적화
- 규제를 적용하지 않은 모델

 # 릿지(Ridge)

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

 # 라쏘(Lasso)

- 선형 회귀에 L1 규제를 적용한 모델
- L1 규제 : 예측 영향력이 작은 피처의 회귀 계수를 0으로 만들어 피처가 선택되지 않게 만드는 규제 모델 => 피처 선택 기능

 # 엘라스틱넷(ElasticNet) - L1 규제(피처의 개수 줄임) + L2 규제(계수 값의 크기 조정)

 # 로지스틱 회귀(Logistic Regression) - 분류에 사용되는 선형 모델

2. 단순 선형 회귀

EWHA EURON

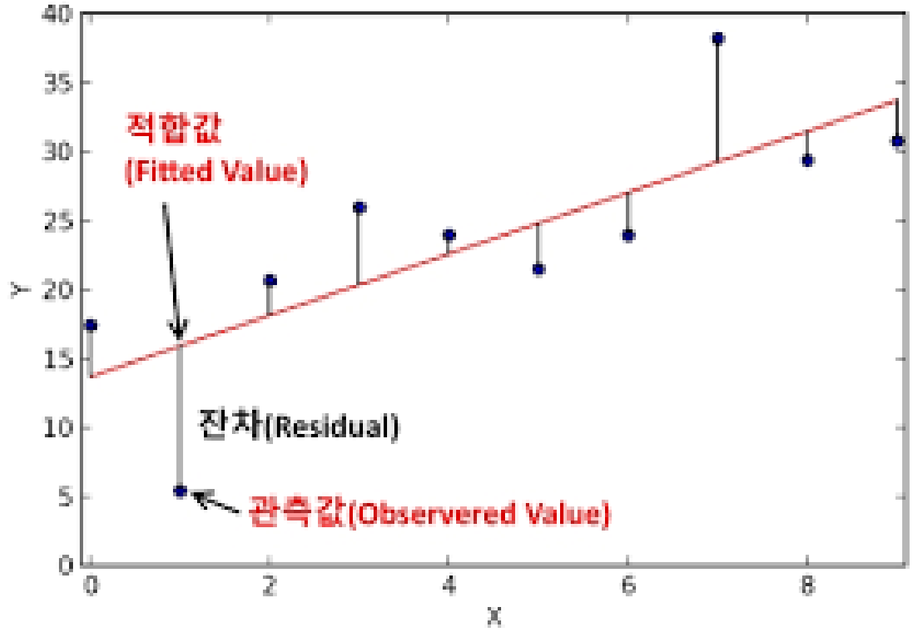
2.1 단순 선형 회귀

 # 단순 선형 회귀

- 독립변수도 하나, 종속변수도 하나인 선형 회귀
- 실제값 : $Y_i = w_0 + w_1 * X_i \cdots \epsilon_i$
- 예측값 : $f(x) = w_0 + w_1 * X$ -> 단순 선형 회귀 모델
- 회귀 계수 : 기울기(w1) / 절편(w0)
- 잔차(e_i) : 실제값과 회귀 모델의 차이에 따른 오류값

  => 최적의 회귀 모델 : 잔차의 합이 최소가 되는 모델

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



2.2 회귀 평가 지표

평가지표	식	사이킷런 평가지표 API	Scoring 함수 적용 값
MAE	$\frac{1}{n}\sum_{i=1}^{n}\lvert Y_i - \hat{Y}_i\rvert$	metrics.mean_absolute_error	'neg_mean_absolute_error'
MSE	$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$	metrics.mean_squared_error	'neg_mean_squared_error'
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	MSE에 로그 적용	metrics.mean_squared_log_error	'neg_mean_squared_log_error'
R2	$\frac{\text{예측값 } Variance}{\text{실제값 } Variance}$	metrics_r2_score	'r2'

 # 'neg'

- Scoring 함수는 score 값이 클수록 좋은 평가 결과로 평가
- 오류를 기반으로 한 평가 지표는 값이 커지면 오히려 나쁜 모델
- '-1'을 곱해주어 작은 오류 값을 더 큰 숫자로 인식하도록 보정

2.3 보스턴 주택 가격 예측

사이킷런 LinearRegression 클래스

- 입력 파라미터
  - fit_intercept [default = True] : intercept 값을 계산할 것인지 말지를 지정
  - normalize [default = False] : False일 경우 무시, True일 경우 회귀 수행 전 입력 데이터셋을 정규화

- 속성
  - coef_ : fit( ) 메서드 수행 시 회귀 계수를 배열 형태로 저장
  - intercept_ : 절편 값

- OLS(Ordinary Least Squares) 추정 방식으로 구현
  - 최소제곱법
  - RSS(오차)를 최소화하는 방법으로 회귀 계수를 추정하는 방식
  - 다중공선성 : 피처 간 상관관계가 매우 높은 경우 분산이 매우 커져서 오류에 민감해지는 것
  - 다중공선성과 피처 중요도를 확인할 수 있음

2.3 보스턴 주택 가격 예측

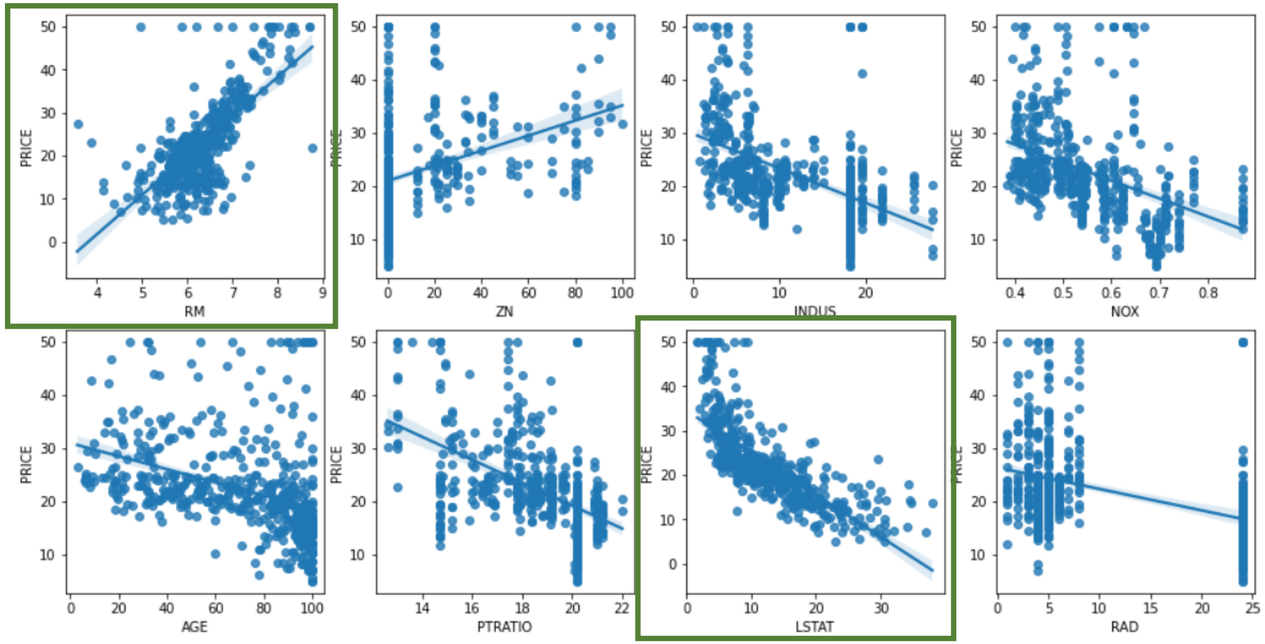
데이터 로드

```
# 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
```



2.3 보스턴 주택 가격 예측

회귀 모델 학습/예측/평가 수행

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

MSE와 R2 측정

```
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:{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
```

```
# 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)
```

회귀 계수를 큰 값 순으로 정렬하기 위해 Series로 생성

```
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
```

EWHA EURON

2.3 보스턴 주택 가격 예측

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))
```

계산된 MSE 값에 -1을 곱한 음수 값 반환

Cross_val_score에서 반환된 값에 다시 -1을 곱하여 원래 모델에서 계산된 MSE(양의 값) 반환

```
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
```

2.4 OLS

OLS Summary

- 사이킷런 이외의 모듈에서 제공하는 Linear regression model
- statsmodels.formula `from statsmodels.formula import api`

```
In [1]: import numpy as np

In [2]: import statsmodels.api as sm

In [3]: import statsmodels.formula.api as smf

# Load data                                          R dataset에도 지원
In [4]: dat = sm.datasets.get_rdataset("Guerry", "HistData").data

# Fit regression model (using the natural log of one of the regressors)
In [5]: results = smf.ols('Lottery ~ Literacy + np.log(Pop1831)', data=dat).fit()
                                  fomula = '예측하고자 하는 칼럼 이름 ~ 원인이 되는 칼럼 이름'
# Inspect the results
In [6]: print(results.summary())
```

https://xiang32.tistory.com/13
https://medium.com/swlh/interpreting-linear-regression-through-statsmodels-summary-4796d359035a

2.4 OLS

OLS Summary

OLS Regression Results

Dep. Variable:	VIQ	R-squared:	0.249
Model:	OLS	Adj. R-squared:	0.158
Method:	Least Squares	F-statistic:	2.733
Date:	Mon, 19 Oct 2020	Prob (F-statistic):	0.0455
Time:	14:07:06	Log-Likelihood:	-167.03
No. Observations:	38	AIC:	344.1
Df Residuals:	33	BIC:	352.2
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	169.7719	90.054	1.885	0.068	-13.443	352.987
Gender[T.Male]	10.1579	10.891	0.933	0.358	-12.001	32.317
Weight	-0.1427	0.215	-0.665	0.511	-0.579	0.294
Height	-2.7337	1.410	-1.938	0.061	-5.603	0.136
MRI_Count	0.0002	6.48e-05	2.489	0.018	2.94e-05	0.000

Omnibus:	5.002	Durbin-Watson:	2.272
Prob(Omnibus):	0.082	Jarque-Bera (JB):	1.892
Skew:	-0.066	Prob(JB):	0.388
Kurtosis:	1.915	Cond. No.	2.41e+07

AIC/BIC :

Multiple variabl을 측정하는 과정에서 패널티를 주는 방식으로 선형 회귀 상에서 모델 성능 비교 -> feature selection에 사용

피처 별 coef와 검정통계량, 95% 신뢰구간 제공

- 다중공산성 존재 : coefficient 간 표준오차(std err) 증가
- P-value가 유의수준보다 작을 경우 해당 피처가 모델에 유의미한 영향을 주는지 확인 -> feature importance 확인

Condition Number :

- 다중 공산성을 확인할 수 있는 통계량
- 1~30 : 다중공산성 존재
- 30 이상 : 다중공산성 강함

EWHA EURON

3. 경사 하강법

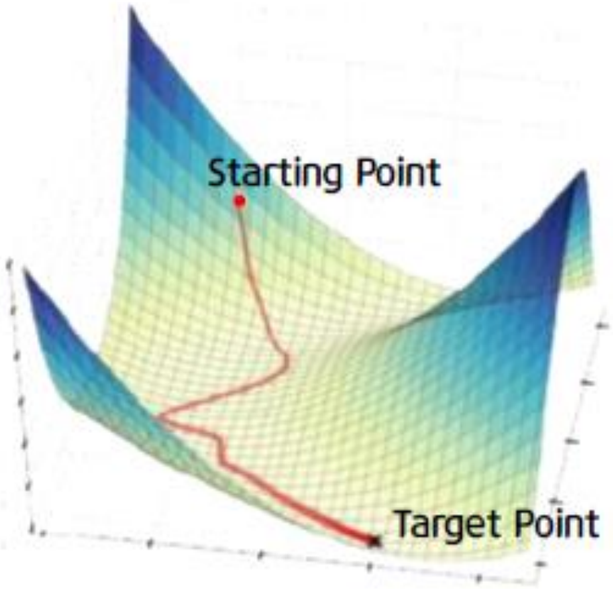
EWHA EURON

3.1 경사 하강법

경사 하강법(Gradient Descent)

- 점진적으로 반복적인 계산을 통해 W 파라미터 값을 업데이트하면서 오류 값이 최소가 되는 W를 구하는 방식
- $\frac{\partial R(w)}{\partial w_1} = \frac{2}{N}\sum -x_t * (y_i - (w_0 + w_1 x_i)) = -\frac{2}{N}\sum_{i=1}^{N} x_i * (\text{실제값}_i - \text{예측값}_i)$

  $\frac{\partial R(w)}{\partial w_0} = \frac{2}{N}\sum -(y_i - (w_0 + w_1 x_i)) = -\frac{2}{N}\sum_{i=1}^{N} x_i * (\text{실제값}_i - \text{예측값}_i)$
- 편미분 값에 보정계수(학습률)를 곱한 값을 반복적으로 적용하며 비용함수 최소 반환값 계산



경사가 가장 가파른 방향으로 최소점을 찾는 방식!

3.1 경사 하강법

경사 하강법 알고리즘

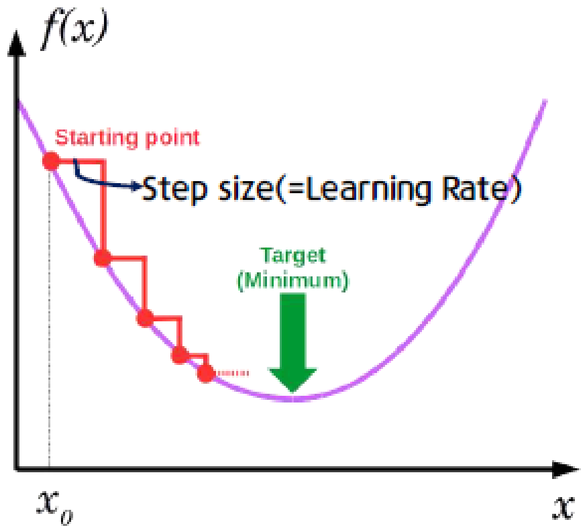
- 임의의 파라미터 값을 시작점으로 지정
- 해당 점에서의 목적함수 Gradient 값을 구함 -> 편미분을 통해 R(w) 값이 최소가 되는 회귀 계수 계산
- 학습률을 곱하여 편미분 값이 너무 커지는 것을 방지

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

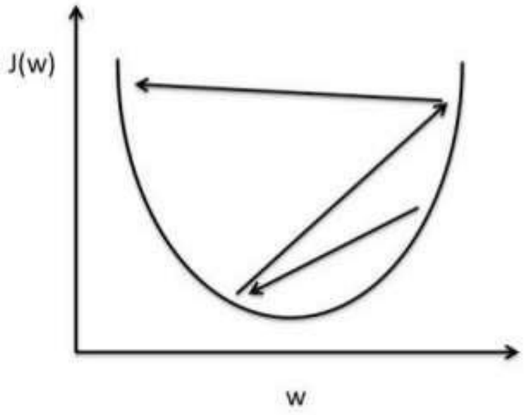
4) 수렴할 때까지 1) ~3) 반복



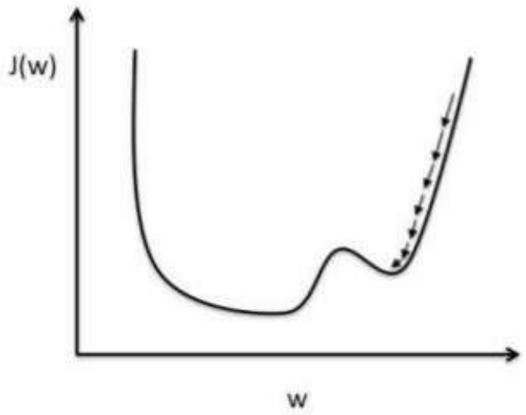
3.1 경사 하강법

학습률

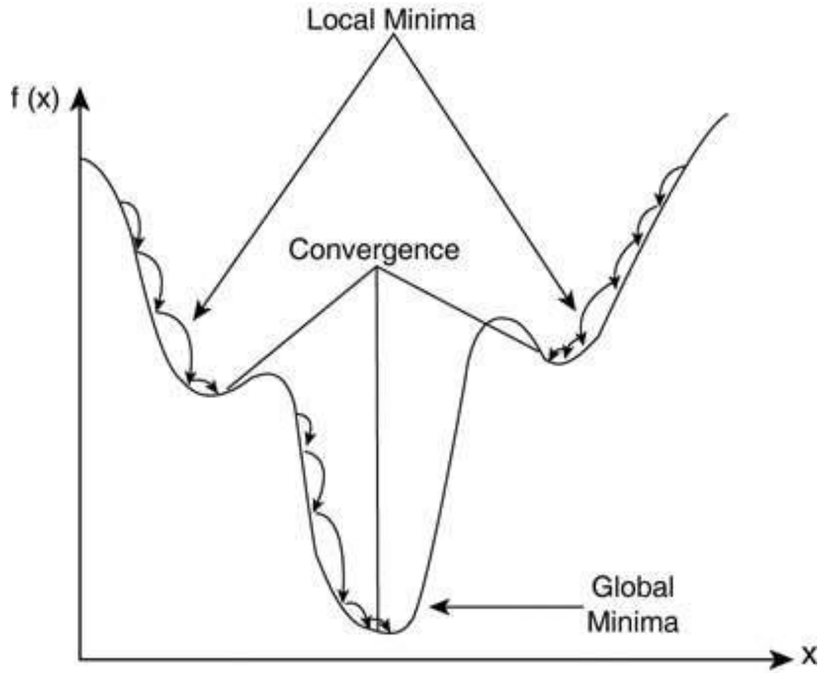
- 학습률이 너무 크면 한 지점으로 수렴하는 것이 아니라 발산할 가능성 존재 -> overshooting
- 학습률이 너무 작으면 수렴이 늦어짐 -> learn too slow
- 시작점을 어디로 잡느냐에 따라서도 수렴 지점이 달라짐



Overshooting



Learn too slow

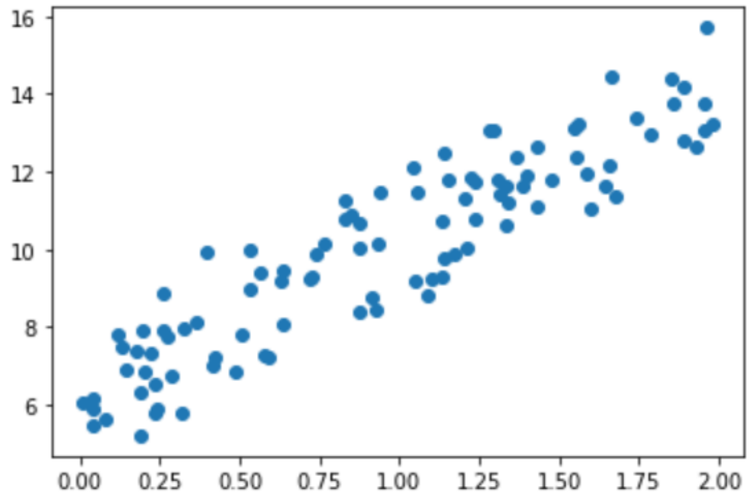


3.1 경사 하강법

예측 데이터셋 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>
```



비용 함수 정의, w1, w0 업데이트 값을 반환하는 함수 정의

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

$$\frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2$$

```
# 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
```

w1_update, w0_update를
W1, w0의 shape과 동일한 크기를 가진 0으로 초기화

3.1 경사 하강법

경사 하강법 함수 생성

```
# 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

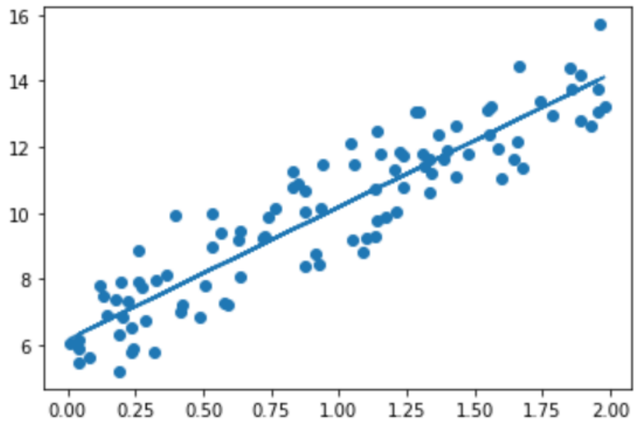
```
# 예측값과 실제값의 RSS 차이를 계산하는 함수 생성
w1, w0 = gradient_descent_steps(X,y,iters=1000)
print('w1:{0:.3f}  w0:{1:.3f}'.format(w1[0,0], w0[0,0]))
y_pred = w1[0,0] * X + w0
print('경사하강 total cost : {0:.4f}'.format(get_cost(y,y_pred)))
```

```
w1:4.022  w0:6.162
경사하강 total cost : 0.9935
```

추정 회귀선 시각화

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

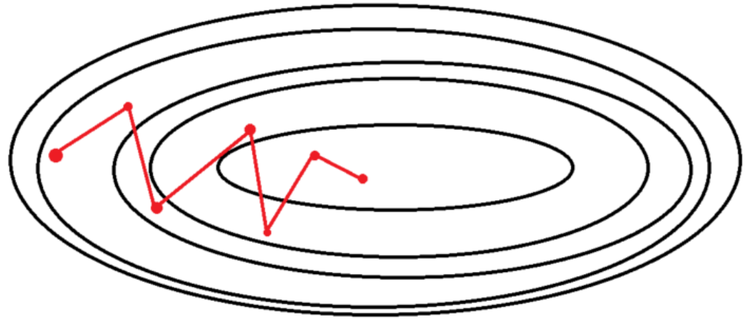
```
[<matplotlib.lines.Line2D at 0x7f49a9621990>]
```



3.2 확률적 경사 하강법

확률적 경사하강법 (Stochastic Gradient Descent)

- 랜덤 샘플링을 통해 추출한 일부 데이터만을 이용해 w 업데이트 값을 계산하는 경사 하강법
- 장점
  - 경사 하강법보다 연산시간 단축
  - Shooting이 일어나기 때문에 지역 최소점에 빠질 risk 적음



- 단점
  - 전역 최소점으로의 수렴을 보장하지 못함
  - 일부 데이터만 포인트로 하여 계산하기 때문에 noise가 큼 -> 정확도 낮음

epoch

- 훈련 세트를 한 번 모두 사용하는 과정
- [딱 하나의 샘플을 훈련 세트에서 랜덤하게 선택 -> 가파른 경사 조금씩 내려옴 -> 또 다른 샘플 선택 -> 반복]

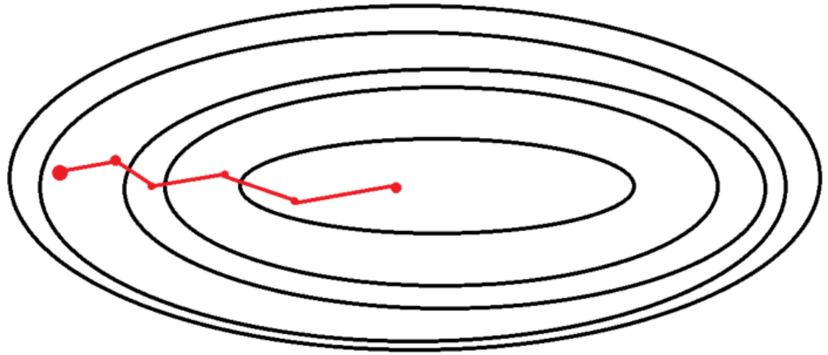
3.3 미니배치 확률적 경사하강법

미니배치 확률적 경사하강법 (Mini-Batch Stochastic Gradient Descent)

- 전체 데이터를 batch_size개씩 나누어 배치로 학습
- 배치 크기를 사용자가 지정하는 것
- ex) 전체 데이터가 1000개인 데이터 학습, batch_size = 100

  전체를 100개씩 총 10묶음의 배치로 나눔

  1 epoch 당 10번 경사하강법 진행



Batch Size

- 보통 2의 n승으로 지정
- 가능하면 학습데이터 개수에 나누어 떨어지도록 지정하는 것이 좋음
- ex) 530개의 데이터를 100개의 배치로 나눔

  각 배치 속 데이터 : 1/100만큼의 영향력

  마지막 배치(30개) 데이터 : 1/30만큼의 영향력 -> 과평가

3.3 미니배치 확률적 경사하강법

```
# 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))

  prev_cost = 100000
  iter_index = 0

  for ind in range(iters):
    np.random.seed(ind)
    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]]
    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
```

batch_size 지정 -> 미니배치

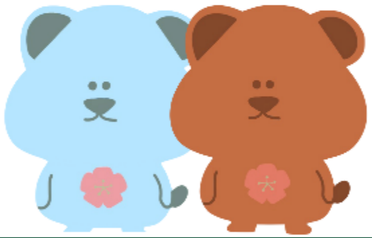
batch_size 만큼 데이터를 추출해 Sample로 저장

```
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 경사 하강 total cost : {0:.4f}'.format(get_cost(y,y_pred)))
```

```
w1: 4.028 w0: 6.156
Stochastic 경사 하강 total cost : 0.9937
```

https://skyil.tistory.com/68

4. 다항회귀



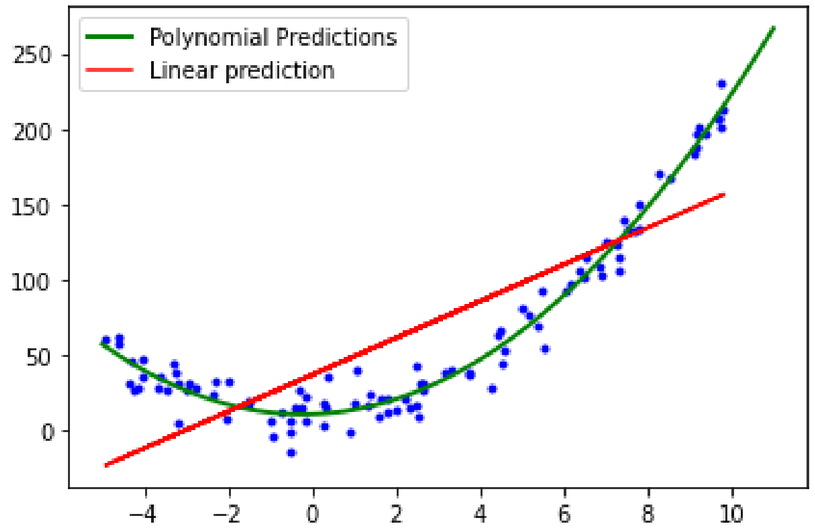
EWHA EURON

4.1 다항회귀

 # 다항회귀
회귀가 독립변수의 단항식이 아닌 2차, 3차 방정식과 같은 다항식으로 표현되는 것

예시)

$$Y = a_0 + a_1 X_1 + a_2 X_1^2$$



오른쪽 그래프의 경우 단항식의 회귀만으로는
독립변수와 종속변수의 관계를 표현하기 힘듦

- 다항 회귀 역시 선형 회귀인데, 이는 선형/비선형 회귀를 나누는 기준은 회귀 계수가 선형/비선형인지에 따른 것이기 때문 (독립변수의 선형/비선형 여부와는 무관)

4.2 사이킷런에서 다항회귀– PolynomialFeatures

사이킷런에는 다항 회귀를 위한 클래스가 명시적으로 제공되지 않음
-> 사이킷런의 PolynomialFeatures 클래스를 통해 피처를 다항식 피처로 변환해 비선형 함수를 선형 모델에 적용시킬 수 있음

 # PolynomialFeatures
- 사이킷런에서 다항식 피처로 변환하는 클래스
- 입력받은 단항식 피처를 degree에 해당하는 다항식 피처로 변환
- fit(), transform() 메서드를 통해 변환 작업 수행

```python
# PolynomialFeatures 이용

from sklearn.preprocessing import PolynomialFeatures
import numpy as np

# 다항식으로 변환할 단항식 생성, [[0,1], [2,3]]의 2x2 행렬 생성
X = np.arange(4).reshape(2,2)
print('일차 단항식 계수 피처:\n', X)

# degree=2인 2차 다항식으로 변환하기 위해 PolynomialFeatures를 이용해 변환
poly = PolynomialFeatures(degree=2)
poly.fit(X)
poly_ftr = poly.transform(X)
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.]]
```

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

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

```
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,)
```

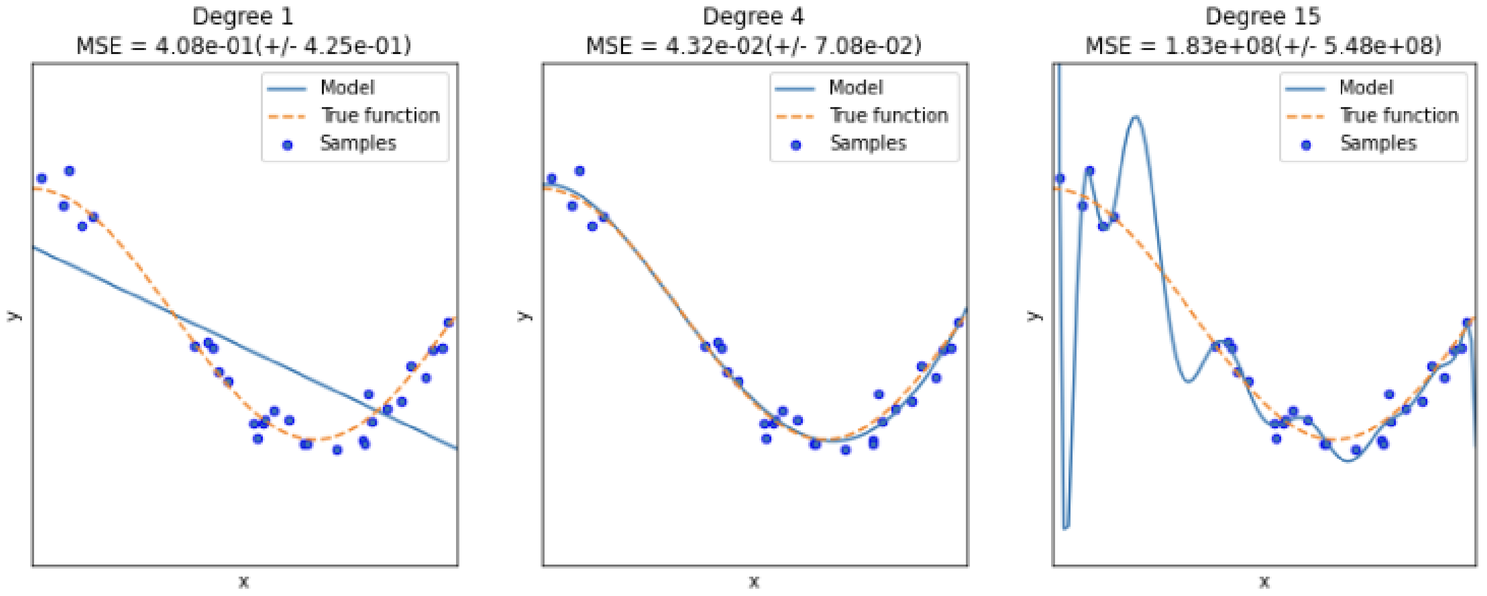
5. 편향-분산 트레이드오프

EWHA EURON

5.1 과(대)적합과 과소적합

 # 과(대)적합
- 모델을 학습할 때 학습 데이터셋에 지나치게 최적화되어 발생하는 문제
- 학습 데이터에만 너무 맞춘 학습이 이뤄져서 학습 데이터셋에서는 모델 성능이 높게 나타나지만 정작 새로운 데이터가 주어졌을 때 오히려 예측 정확도가 떨어지는 현상

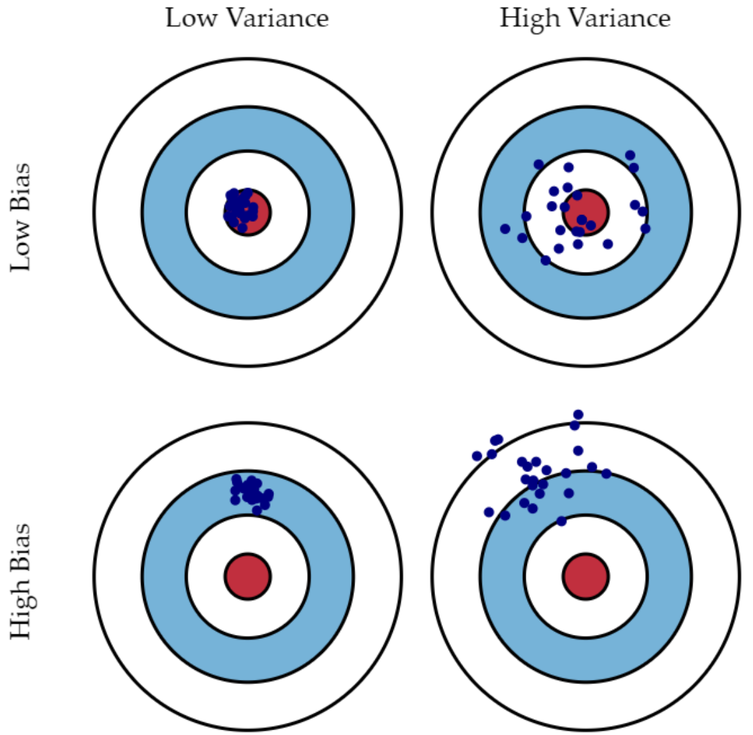
 # 과소적합
- 모델이 충분히 복잡하지 않아(최적화가 제대로 수행되지 않아) 발생하는 문제
- 학습 데이터의 패턴을 정확히 반영하지 못함

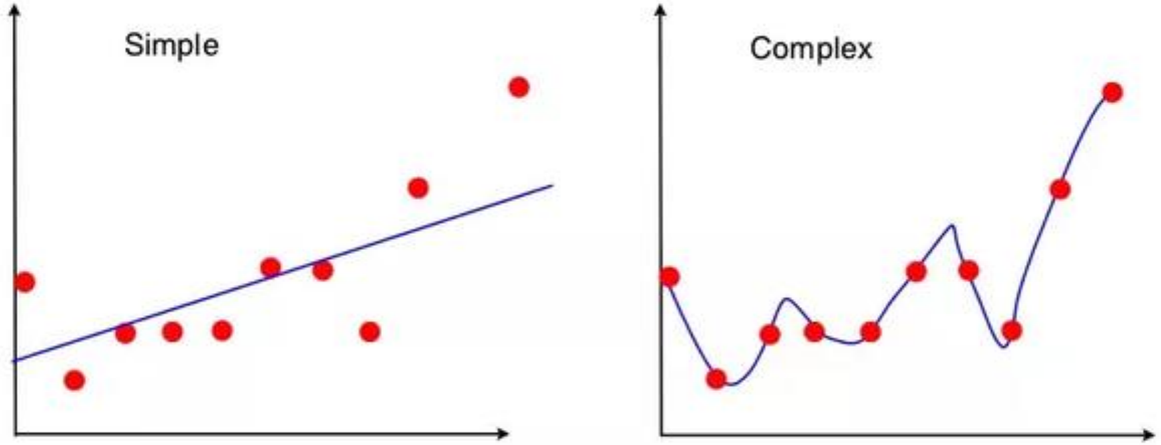


5.2 편향-분산 트레이드오프

 # 편향
- 예측 값과 실제 값 간의 차이
- 지나치게 단순한 모델로 인한 error
- 편향이 크면 과소적합이 되기 쉬우며 모델이 무언가 중요한 걸 놓치고 있음

 # 분산
- 예측 값들끼리의 차이
- 지나치게 복잡한 모델로 인한 error
- 분산이 크면 과대적합이 되기 쉬우며 훈련 데이터의 변동에 모델이 과도하게 반응하여 일반화 되지 않음





왼쪽은 큰 편향, 작은 분산
오른쪽은 작은 편향, 큰 분산

5.2 편향-분산 트레이드오프

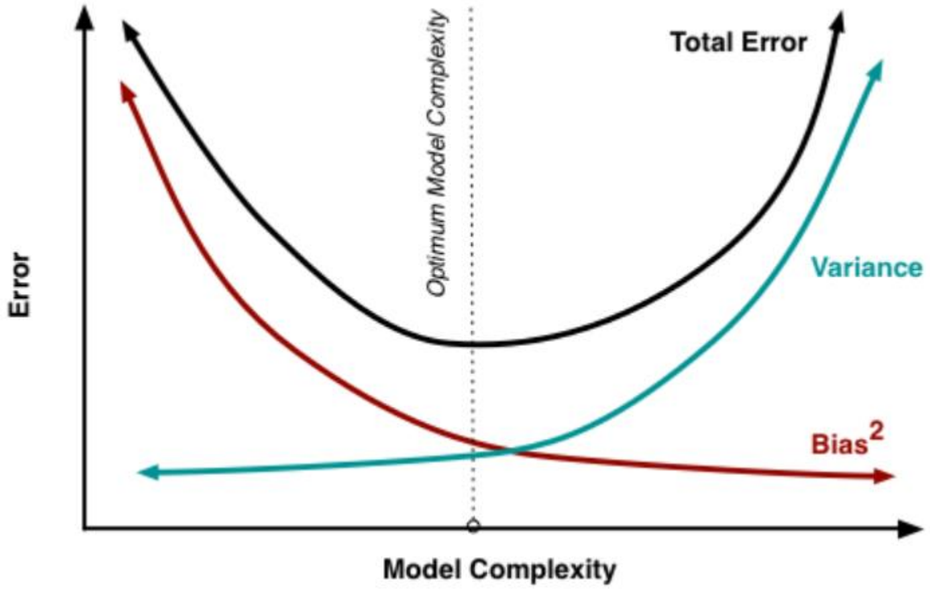
 # Error

- 편향 제곱 + 분산 + 불가피한 오차로 구성

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

 # 편향-분산 트레이드오프

- 모델의 복잡도가 커지면 분산이 늘어나고 편향은 줄어든다
- 모델의 복잡도가 줄어들면 편향이 커지고 분산은 감소한다
- 이렇게 한 쪽이 높으면 한 쪽이 낮아지는 편향과 분산의 관계를 두고 편향-분산 트레이드오프라고 함



* 편향과 분산이 서로 트레이드오프를 이루면서 오류 값이 최대로 낮아지는 모델을 구축하는 것이 가장 효율적인 머신러닝 예측 모델을 만드는 방법이다

6. 규제 선형 모델

EWHA EURON

6.1 규제

비용 함수 RSS(Residual Sum of Squares)는 실제 값과 예측 값의 차이를 최소화하는 것만 고려하다 보니 학습 데이터에 지나치게 과적합되고, 회귀 계수가 쉽게 커졌음
-> 이를 막기 위해 비용함수는 RSS 최소화 방법과 과적합 방지를 위한 회귀 계수 값이 커지지 않도록 하는 방법이 균형을 이뤄야 함

 # 비용 함수 목표

Min(RSS(W) + alpha*W노름 제곱 (L2 기준)

alpha : 학습 데이터 적합 정도와 회귀 계수 값의 크기 제어를 수행하는 튜닝 파라미터
- alpha 값을 크게 하면 비용함수는 회귀 계수 W의 값을 작게 해 과적합을 개선
- alpha 값을 작게 하면 회귀 계수 W의 값이 커져도 어느 정도 상쇄가 가능하므로 학습 데이터 적합을 더 개선할 수 있음

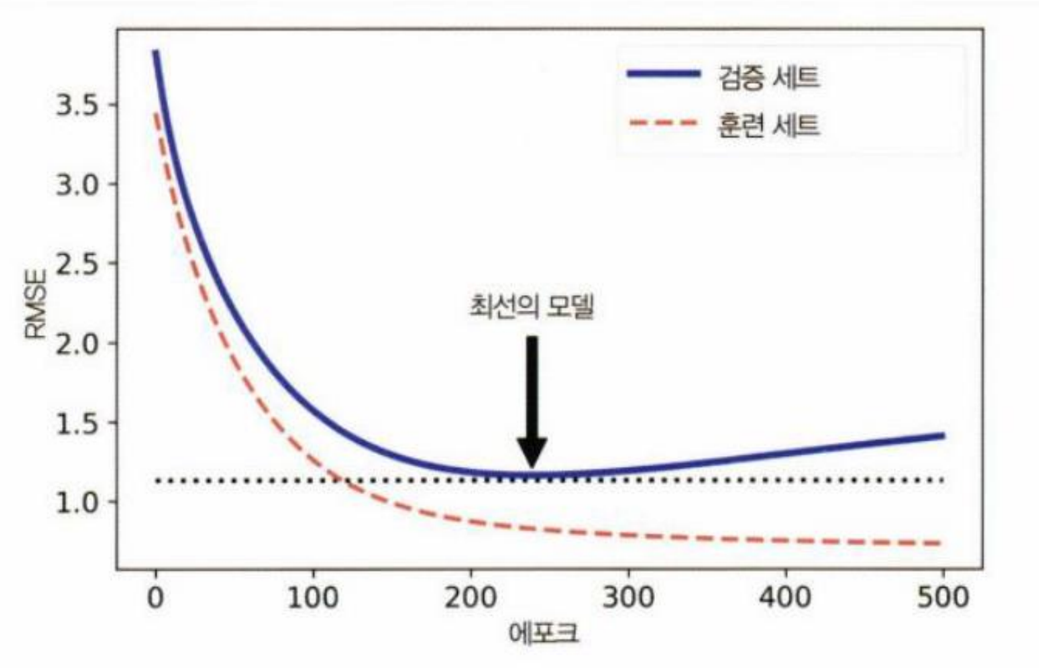
 # 규제
- 비용 함수에 alpha 값으로 페널티를 부여해 회귀 계수 값의 크기를 감소시켜 과적합을 개선하는 방식
- L2 규제 : W의 제곱에 대해 페널티를 부여하는 방식 -> 릿지 회귀
- L1 규제 : W의 절대값에 대해 페널티를 부여하는 방식 -> 라쏘 회귀

6.2 조기 종료

 # 조기 종료
- 과대적합을 방지하는 대표적인 방법

- 특정 Epoch 내 validation loss가 감소하지 않으면 과대적합이 발생했다고 간주하고 학습을 종료함으로써 validation loss가 가장 낮은 지점인 최적 epoch에서의 모델을 저장해주는 기법

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



6.3 릿지

 # 릿지
- 선형 회귀에 L2 규제를 추가한 회귀 모델
- W 제곱에 대해 페널티 부여하여 회귀 계수를 작게 만들어줌

릿지 회귀의 비용 함수)

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

- 입력 특성의 스케일에 민감하여 스케일링 필수

 # L2 규제
- 상대적으로 큰 회귀 계수 값의 예측 영향도를 감소시키기 위해서 회귀 계수 값을 더 작게 만드는 규제 모델

 # Ridge
- 사이킷런에서 릿지 회귀를 구현하는 클래스
- 주요 생성 파라미터
  alpha : 릿지 회귀의 L2 규제 계수에 해당

```python
# 릿지 회귀
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

# alpha=10으로 설정해 릿지 수행
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(np.round(neg_mse_scores, 3)))
print('5 folds의 개별 RMSE scores: ', np.round(rmse_scores, 3))
print('5 folds의 평균 RMSE: {0:.3f}'.format(avg_rmse))
```

```
5 folds의 개별 Negative MSE scores:  [-11. -24. -28. -75. -29.]
5 folds의 개별 RMSE scores:  [3.38  4.929 5.305 8.637 5.34 ]
5 folds의 평균 RMSE: 5.518
```

EWHA EURON

6.4 라쏘

 # 라쏘
- 선형 회귀에 L1 규제를 추가한 회귀 모델
- W 절댓값에 대해 페널티 부여하여 예측 영향력이 작은 피처의 회귀 계수를 0으로 만들어줌
- 적절한 피처만 회귀에 포함시킴으로써 피처 선택 효과

라쏘 회귀의 비용 함수)

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

 # L1 규제
- 예측 영향력이 작은 피처의 회귀 계수를 0으로 만들어 회귀 예측 시 피처가 선택되지 않게 하는 규제 모델

 # Lasso
- 사이킷런에서 라쏘 회귀를 구현하는 클래스
- 주요 생성 파라미터
  alpha : 릿지 회귀의 L1 규제 계수에 해당

```python
# 라쏘 회귀
from sklearn.linear_model import Lasso
from sklearn.model_selection import cross_val_score

# alpha=0.07으로 설정해 라쏘 수행
lasso = Lasso(alpha=0.07)
neg_mse_scores = cross_val_score(lasso, 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(np.round(neg_mse_scores, 3)))
print('5 folds의 개별 RMSE scores: ', np.round(rmse_scores, 3))
print('5 folds의 평균 RMSE: {0:.3f}'.format(avg_rmse))
```

```
5 folds의 개별 Negative MSE scores:  [-11. -24. -29. -80. -30.]
5 folds의 개별 RMSE scores:  [3.342 4.938 5.342 8.972 5.468]
5 folds의 평균 RMSE: 5.612
```

6.5 엘라스틱넷

 # 엘라스틱넷
- L2 규제와 L1 규제를 결합한 회귀
- 피처가 많은 데이터셋에서 L1 규제로 피처 개수를 줄이고, L2 규제로 계수 값 크기 조정
- 수행시간이 상대적으로 오래 걸린다는 단점

엘라스틱넷 회귀의 비용 함수)

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

 # ElasticNet
- 사이킷런에서 엘라스틱넷 회귀를 구현하는 클래스
- 엘라스틱넷의 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 L1 규제와 동일

```python
# 엘라스틱넷 회귀
from sklearn.linear_model import ElasticNet
from sklearn.model_selection import cross_val_score

# alpha=0.5, l1_ratio=0.7로 설정해 엘라스틱넷 수행
elastic = ElasticNet(alpha=0.5, l1_ratio=0.7)
neg_mse_scores = cross_val_score(elastic, 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(np.round(neg_mse_scores, 3)))
print('5 folds의 개별 RMSE scores: ', np.round(rmse_scores, 3))
print('5 folds의 평균 RMSE: {0:.3f}'.format(avg_rmse))
```

```
5 folds의 개별 Negative MSE scores:  [-13. -28. -41. -55. -21.]
5 folds의 개별 RMSE scores:  [3.625 5.309 6.378 7.394 4.628]
5 folds의 평균 RMSE: 5.467
```

6.6 선형 회귀 모델을 위한 데이터 변환 방법

1. StandardScaler / MinMaxSclaer을 이용해서 표준 정규 분포 형태의 데이터 세트로 변환하거나 최솟값 0, 최댓값 1인 정규화 수행

-> 예측 향상 기대가 적음

2. 스케일링 / 정규화를 수행한 데이터 세트에 다시 다항 특성을 적용하여 변환

-> 1번 방법이 성능 향상에 효과가 없을 때 적용하는 방법으로 피처가 많으면 과적합과 시간이 오래 걸린다는 문제 발생

3. 로그 변환을 취해 정규 분포에 가깝게 만들기. 원래 값에 log 함수 적용

-> 가장 많이 사용되는 방법. 심하게 왜곡된 데이터에 좋음

07 로지스틱 회귀

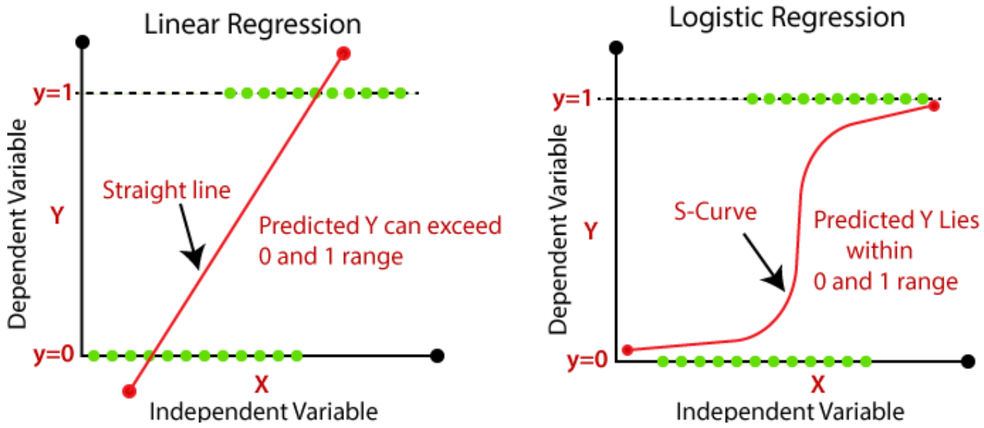
EWHA EURON

#7-1 로지스틱 회귀

로지스틱 회귀란?

선형 회귀 방식을 분류에 적용한 알고리즘

시그모이드 함수 최적선을 찾고 이 시그모이드 함수의 반환 값을 확률로 간주해 확률에 따라 분류를 결정



시그모이드 함수

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

x값이 +, -로 아무리 커지거나 작아져도 y값은 항상 0과 1 사이 값 변환

x값이 커지면 1에 근사하며 x값이 작아지면 0에 근사함

$x = 0$일때는 0.5

#7-2 LogisticRegression 클래스

위스콘신 유방암 데이터 세트

```
import pandas as pd
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline

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

cancer = load_breast_cancer()


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

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

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

#7-2 LogisticRegression 클래스

위스콘신 유방암 데이터 세트

```
from sklearn.metrics import accuracy_score, roc_auc_score

# 로지스틱 회귀를 이용하여 학습 및 예측 수행.
# solver 인자값을 생성자로 입력하지 않으면 solver='lbfgs'
lr_clf = LogisticRegression()
lr_clf.fit(X_train, y_train)
lr_preds = lr_clf.predict(X_test)

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

```
accuracy: 0.977, roc_auc:0.972
```

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

# 여러개의 solver 값별로 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)

    # accuracy와 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)))
```

```
solver:lbfgs, accuracy: 0.977, roc_auc:0.972
solver:liblinear, accuracy: 0.982, roc_auc:0.979
solver:newton-cg, accuracy: 0.977, roc_auc:0.972
solver:sag, accuracy: 0.982, roc_auc:0.979
solver:saga, accuracy: 0.982, roc_auc:0.979
```

#7-2 LogisticRegression 클래스

위스콘신 유방암 데이터 세트

```
from sklearn.model_selection import GridSearchCV

params={'solver':['liblinear', 'lbfgs'],
        'penalty':['l2', 'l1'],
        'C':[0.01, 0.1, 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
```

주요 하이퍼 파라미터

penalty: 규제의 유형을 설정하며 'l2' 로 설정 시 L2 규제를, 'l1 '으로 설정 시 L1 규제를 뜻함, default = 'l2'

C: 규제 강도를 조절하는 alpha 값의 역수, C = 1/alpha
C가 작을수록 규제 강도가 커짐

로지스틱 회귀는 가볍고 빠르며, 이진 분류 예측 성능도 뛰어남

희소한 데이터 세트 분류에도 뛰어난 성능을 보여 텍스트 분류에서도 자주 사용됨

08 회귀 트리

EWHA EURON

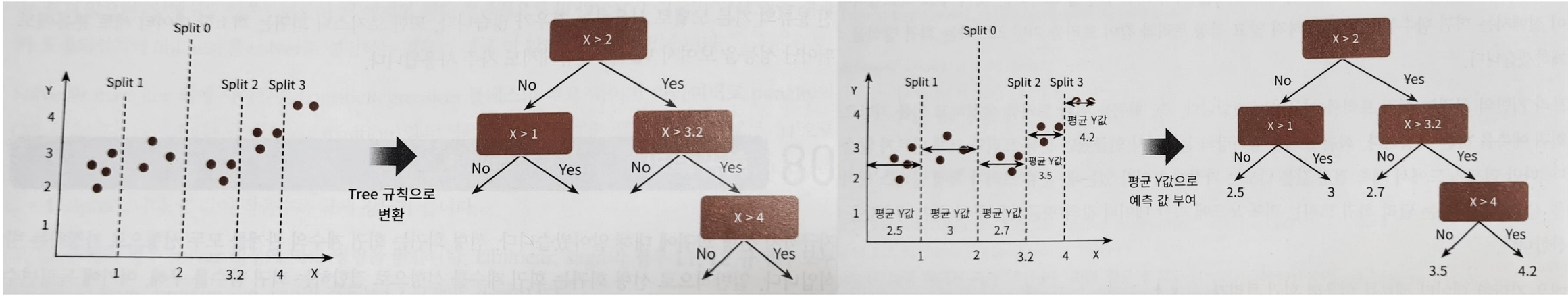
#8-1 회귀 트리

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 예측 값을 계산함

회귀 트리 동작 원리



#8-2 CART 알고리즘

CART (Classification and Regression Trees)

분류 뿐만 아니라 회귀도 가능하게 해주는 트리 생성 알고리즘

➔ 결정 트리, 랜덤 포레스트, GBM, XGBoost, LightGBM 등의 모든 트리 기반 알고리즘은 분류 뿐 아니라 회귀도 가능함

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

#8-3 보스턴 주택 가격 예측

RandomForestRegressor를 이용한 보스턴 주택 가격 예측

```
5 교차 검증의 개별 Negative MSE scores:  [ -7.88 -13.14 -20.57 -46.23 -18.88]
5 교차 검증의 개별 RMSE scores :  [2.81 3.63 4.54 6.8  4.34]
5 교차 검증의 평균 RMSE : 4.423
```

```
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

# 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)
neg_mse_scores = cross_val_score(rf, 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 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))
```

#8-3 보스턴 주택 가격 예측

랜덤포레스트뿐만 아니라 결정 트리, 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)

# 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)
```

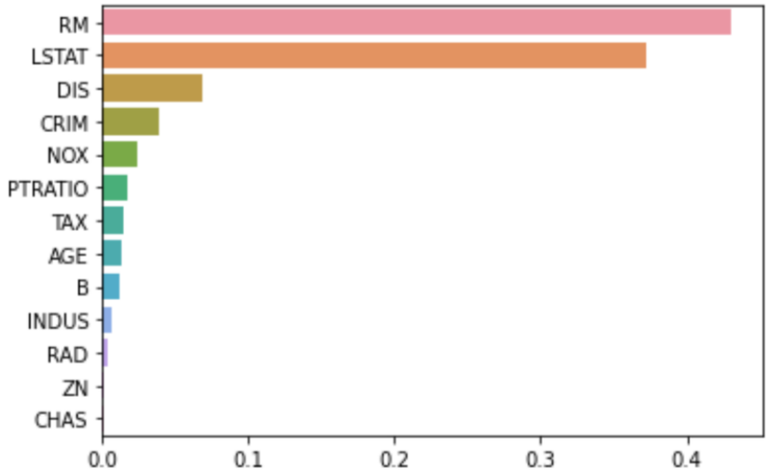
```
#####  DecisionTreeRegressor  #####
 5 교차 검증의 평균 RMSE : 5.978
#####  RandomForestRegressor  #####
 5 교차 검증의 평균 RMSE : 4.423
#####  GradientBoostingRegressor  #####
 5 교차 검증의 평균 RMSE : 4.269
#####  XGBRegressor  #####
 5 교차 검증의 평균 RMSE : 4.251
#####  LGBMRegressor  #####
 5 교차 검증의 평균 RMSE : 4.646
```

#8-3 보스턴 주택 가격 예측

회귀 트리 Regressor 클래스는 선형 회귀와 다른 처리 방식이므로 회귀 계수를 제공하는 coef_ 속성이 없음
→ feature_importances_를 이용해 피처 별 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:>



09 스태킹 앙상블 모델

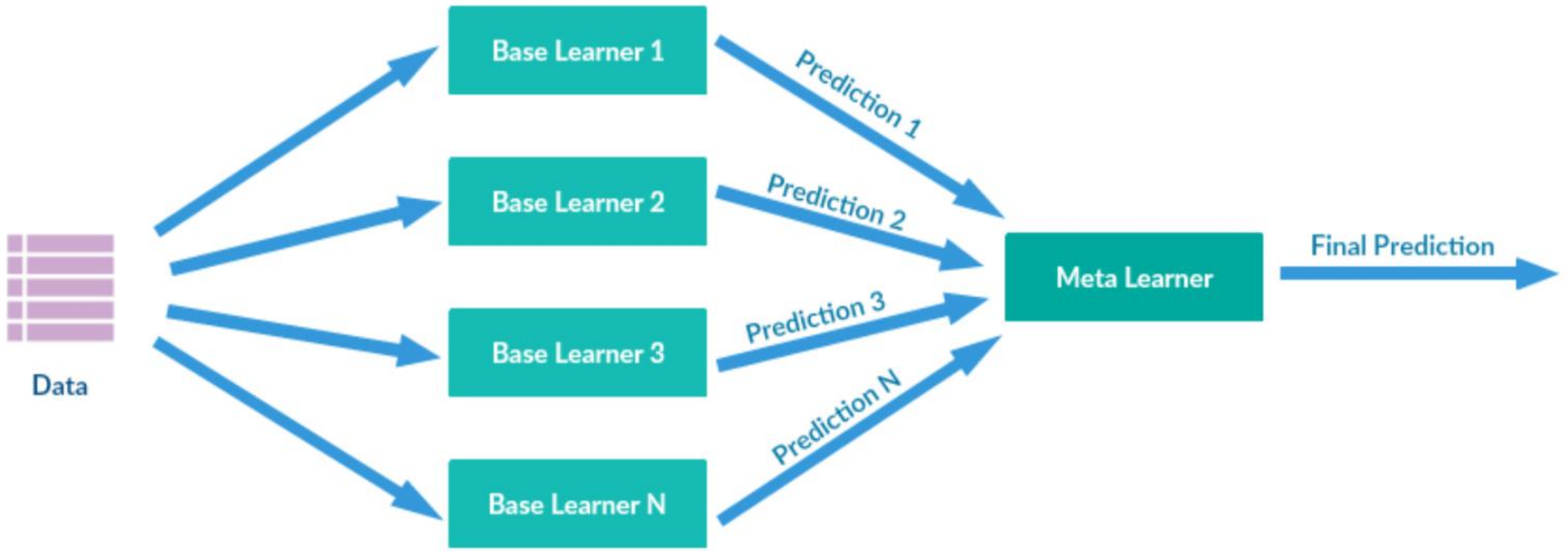
EWHA EURON

#9-1 스태킹 모델

#1 개별적인 기반 모델

#2 개별 기반 모델의 예측 데이터를 학습 데이터로 만들어서 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 것이 핵심!



#9-2 스태킹 앙상블 모델을 통한 회귀 예측

get_stacking_base_datasets()

```
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_folds값으로 KFold 생성.
    kf = KFold(n_splits=n_folds, shuffle=False)
    #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 셋 추출
        print('₩t 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 세트내 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 스태킹 앙상블 모델을 통한 회귀 예측

적용할 개별 모델: 릿지, 라쏘, XGBoost, LightGBM

```
# get_stacking_base_datasets( )은 넘파이 ndarray를 인자로 사용하므로 DataFrame을 넘파이로 변환.
X_train_n = X_train.values
X_test_n = X_test.values
y_train_n = y_train.values

# 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_n, 5)
lasso_train, lasso_test = get_stacking_base_datasets(lasso_reg, X_train_n, y_train_n, X_test_n, 5)
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, 5)

# 개별 모델이 반환한 학습 및 테스트용 데이터 세트를 Stacking 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 모델을 적용.
meta_model_lasso = Lasso(alpha=0.0005)

#기반 모델의 예측값을 기반으로 새롭게 만들어진 학습 및 테스트용 데이터로 예측하고 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: 0.09799152965189684
```

➔ 현재까지 가장 좋은 성능 평가를 보임

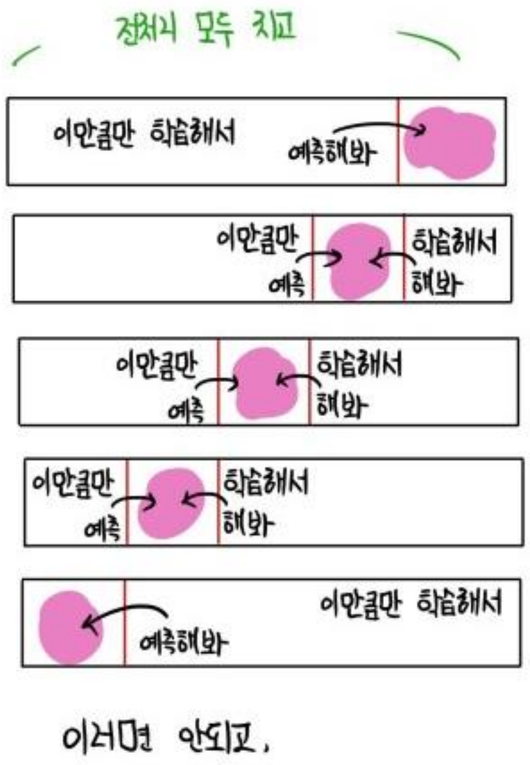
10 파이프라인

EWHA EURON

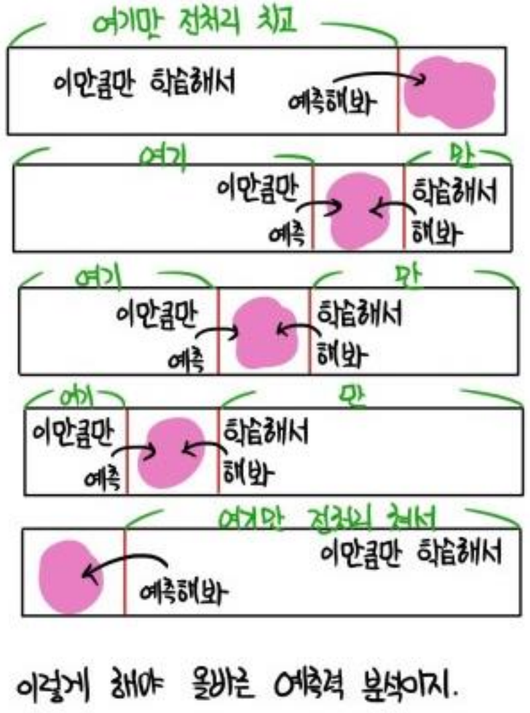
#10-1 파이프라인

파이프라인이란?

머신러닝에서 Pipeline이란 데이터의 가공, 변환 등의 전처리와 알고리즘 적용을 마치 '수도관에서 물이 흐르듯' 한꺼번에 스트림 기반으로 처리한다는 의미



이러면 안되고,



이렇게 해야 올바른 예측력 분석이지.

파이프라인의 필요성

train_test_split()을 통해 train, test set을 나누고 성능 평가를 하는 것처럼 교차 검증시에도 전체 학습
데이터를 전처리하고 이것을 교차 검증하면 안됨
교차 검증 시 학습 데이터만 전처리하고,
검증하는 데이터는 전처리가 되지 않은 데이터여야 함
➔ 파이프라인을 이용

EWHA EURON

#10-1 파이프라인

파이프라인의 장점

Pipeline을 이용하면 데이터의 전처리와 머신러닝 학습 과정을 통일된 API 기반에서 처리할 수 있어 더 직관적인 ML 모델 코드를 생성할 수 있음

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 원하는 대로)

파이프 라인에서 마지막 객체를 제외한 나머지 객체들은 transform, fit_transform 메소드를 제공하는 변환기만 허용

맨 마지막 객체는 predict 메서드가 있는 객체가 들어와야 함

#10-2 파이프라인 예시

파이프라인을 사용하지 않았을 때

```
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.fit_transform(X_train)
X_test = poly.transform(X_test)

lr = LinearRegression()
lr.fit(X_train, y_train)
pred = lr.predict(X_test)
mse = mean_squared_error(y_test, pred)
rmse = np.sqrt(mse)

print('MSE: {0}, RMSE: {1}'.format(mse, rmse))
print('Variance Score: ', r2_score(y_test, pred))
print('절편 값: ', lr.intercept_)
print('회귀 계수 값: ', np.round(lr.coef_,1))
```

```
MSE: 18.871409324914627, RMSE: 4.344123539324662
Variance Score:  0.7665241262729234
```

#10-2 파이프라인 예시

파이프라인을 사용했을 때

```
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_transfrom() 가능 객체, 마지막 객체는 predict 가능 객체

pipe.fit(X_train, y_train)
print('Varience Score: ', pipe.score(X_test, y_test))
```

```
Varience Score:  0.7665241262729234
```

11 학습곡선 해석

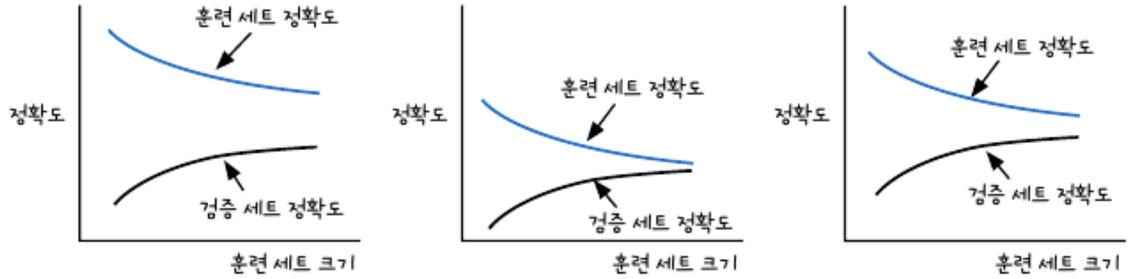
EWHA EURON

#11-1 학습곡선을 통해 과대적합과 과소적합 알아보기

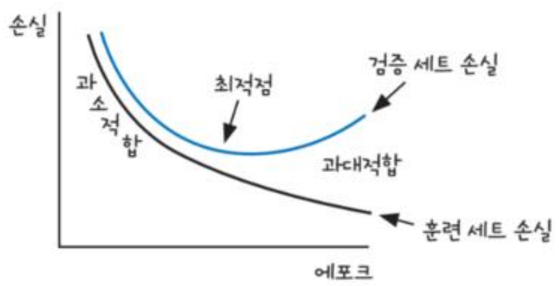
과대적합 (overfitting)
모델이 train dataset에서 좋은 성능을 내지만 test dataset에서는 낮은 성능을 내는 경우
train dataset의 정확도와 test dataset의 정확도의 간격이 큰 것 ➔ 분산이 크다

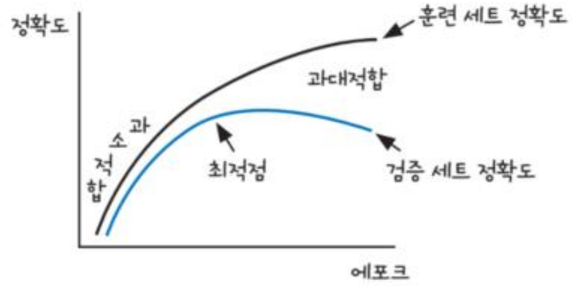
주요 원인으로 train dataset에 충분히 다양한 패턴의 샘플이 포함되지 않은 경우가 있음
→ train data의 샘플을 더 모아 성능을 향상시킬 수 있음
→ train sample을 더 많이 모을 수 없는 경우, 모델이 train dataset에 집착하지 않도록 가중치를 제한
(= 모델의 복잡도를 낮춘다)

과소 적합 (underfitting)
train dataset와 test dataset의 성능에는 차이가 크지 않지만 모두 낮은 성능을 나타내는 경우
주요 원인은 모델이 충분치 않은 것 → 복잡도가 높은 모델을 사용



#11-2 에포크와 손실 함수의 그래프



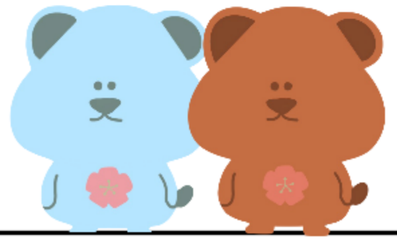


최적점에 맞추어 early stopping 해야 함
= test set의 손실이 올라가는 시점
= test set의 정확도가 떨어지는 시점

왼쪽 그래프
- test dataset의 손실과 train dataset의 손실을 나타낸 것
- train dataset의 손실은 에포크가 진행될수록 감소하지만test dataset의 손실은 에포크의 횟수가 최적점을 지나가면 오히려 상승함
- 최적점 이후에도 계속해서 train dataset으로 모델을 학습시키면 모델이 train dataset에 더 밀착하여 학습하기 때문 = overfitting
- 반대로 최적점 이전에는 train dataset과 test dataset의 손실이 비슷한 간격을 유지하면서 함께 줄어드는데, 이 영역에서 학습을 중지하면 과소 적합된 모델이 만들어짐 = underfitting

오른쪽 그래프
- 정확도 그래프는 손실 그래프를 뒤집은 것과 같음

THANK YOU



EWHA EURON