UST 과학기술연합대학원대학교
UNIVERSITY OF SCIENCE & TECHNOLOGY

7강: 선형회귀의 이해

인공지능 일반강좌 : 기계학습의 이해(L2-1)

Contents

회귀(Regression) 소개(1)

- 회귀의 역사
  - ✓ 영국의 통계학자 ‘ 갈톤(Galton)’의 유전적 특성중에 부모와 자식의 키 관계
  - ✓ “사람의 키는 평균 키로 회귀(Regression)하려는 경향을 가진다는 자연의 법칙이 있다“
  - ✓ 회귀 분석은 데이터 값이 평균과 같은 일정한 값으로 돌아가려는 경향을 이용한 통계학 기법이다.
- 회귀에 대하여
  - ✓ 대부분 알고리즘은 블랙박스처럼 사용 가능
  - ✓ 하지만, 기본적인 모델이 작동하는 방식을 이해해야 함
  - ✓ 이 장에서 다루는 주제는 대부분 신경망 설계, 훈련, 이해 의 핵심

회귀(Regression) 소개(2)

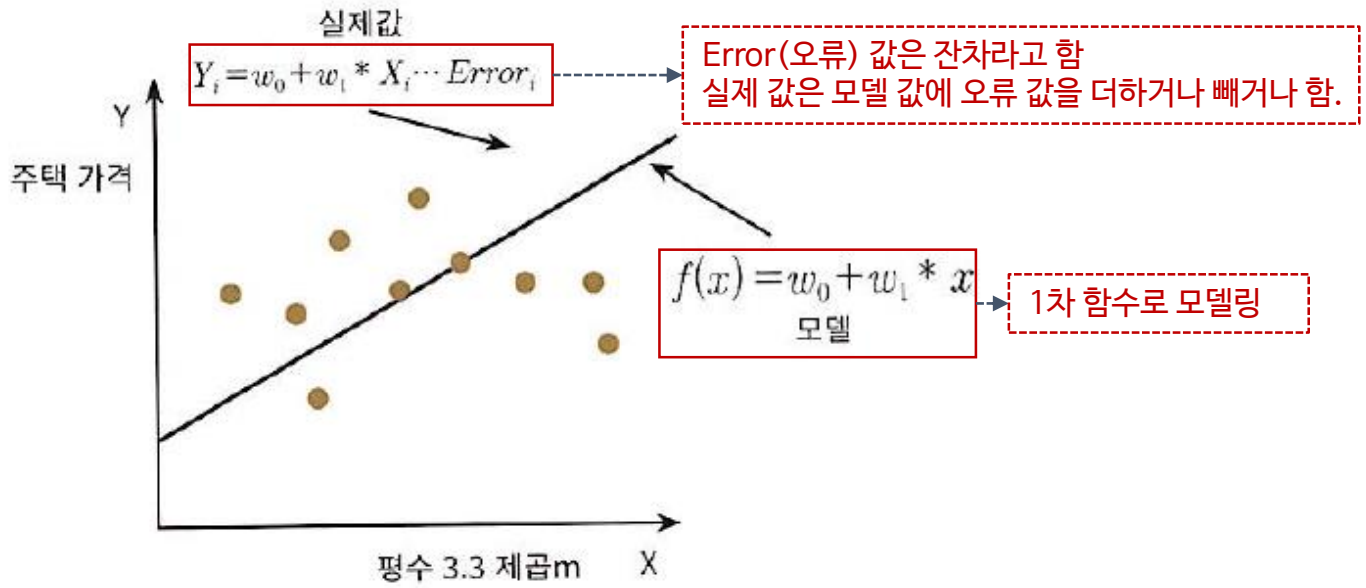
- 통계학 측면에서 회귀는
  - ✓ 독립변수는 기계학습/신경망에서 피처(특성)/입력에 해당하며
  - ✓ 종속변수는 기계학습/신경망에서 결정 값(target)/라벨(label)에 해당
- 지도학습은 2가지 유형으로 나눔
  - ✓ 회귀는 연속적인 숫자 값
  - ✓ 분류는 예측값이 카테고리와 같은 이산형 클래스 값
- 선형회귀
  - ✓ 가장 많이 사용되며, 실제 값과 예측 값의 차이(오류의 제곱)를 최소화하는 직선형 회귀선을 최적화하는 방식
- 선형회귀는 규제(Regularization)에 대하여 분류 가능
  - ✓ 선형 회귀의 과적합을 해결하기 위해서 회귀 계수에 페널티 값을 적용

회귀(Regression) 소개(3)

- **대표적인 선형회귀 모형**
  - ✓ **일반 선형 회귀는 규제를 적용하기 않음**
  - ✓ **릿지(Ridge) 회귀는 선형회귀에 L2 규제를 추가한 모델**
    - L2 규제는 회귀 계수 값의 예측 영향도를 감소시키기 위해 회귀 계수를 작게 만듬
  - ✓ **라쏘(Lasso) 회귀는 선형회귀에 L1 규제를 적용한 모델**
    - L1 규제는 예측 영향력이 작은 피처의 회귀 계수를 0으로 만듬. 피처 선택 가능
  - ✓ **엘라스틱(ElaticNet) 회귀는 L1과 L2가 결합한 모델.**
    - 주로 피처가 많은 모델에 적용되며, L1 규제로 피처를 줄이고, 동시에 L2 규제 적용
  - ✓ **로지스틱 회귀 (Logistic Regression)은 회귀라는 이름이 있지만, 사실은 분류에 사용되는 선형모델.**
    - 강력한 분류 알고리즘.
    - 이진분류 뿐만 아니라 희소영역의 분류에서 뛰어난 예측 성능을 보임

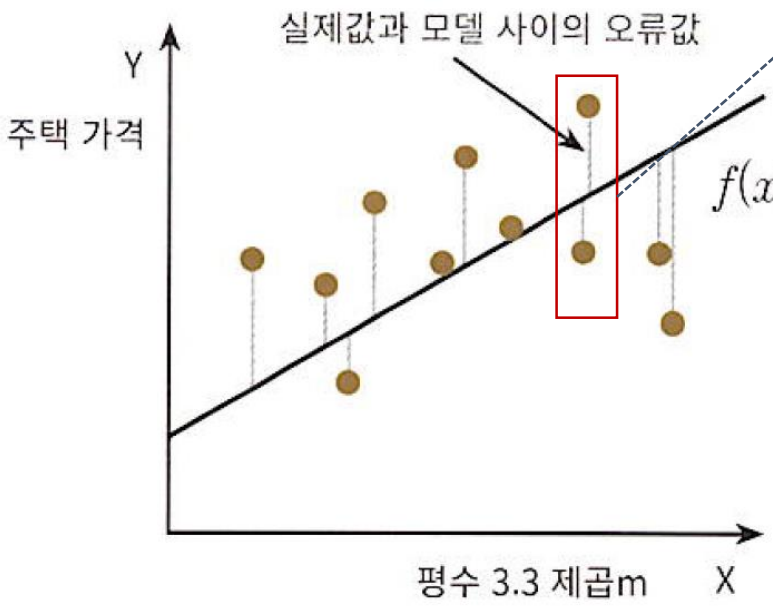
회귀(Regression) 소개(4)

- 단순 선형회귀를 통한 회귀의 이해
  - ✓ 단순 선형회귀는 1개의 독립변수, 1개의 종속변수.
  - ✓ (예) 주택 가격이 주택의 크기로만 결정된다고 해보면,



회귀(Regression) 소개(7)

최적의 회귀 모델은 전체 데이터의 잔차(오차) 합이 최소가 되는 모델을 만드는 것임!



오차는 양수(+) 혹은 음수(-) 가능하므로, 절대값을 취하거나(MAE), 오차 값의 제곱 (MSE)를 사용.
보통 RSS(Residual Sum of Square)~(오차)의 제곱

$$f(x) = w_0 + w_1 * x$$

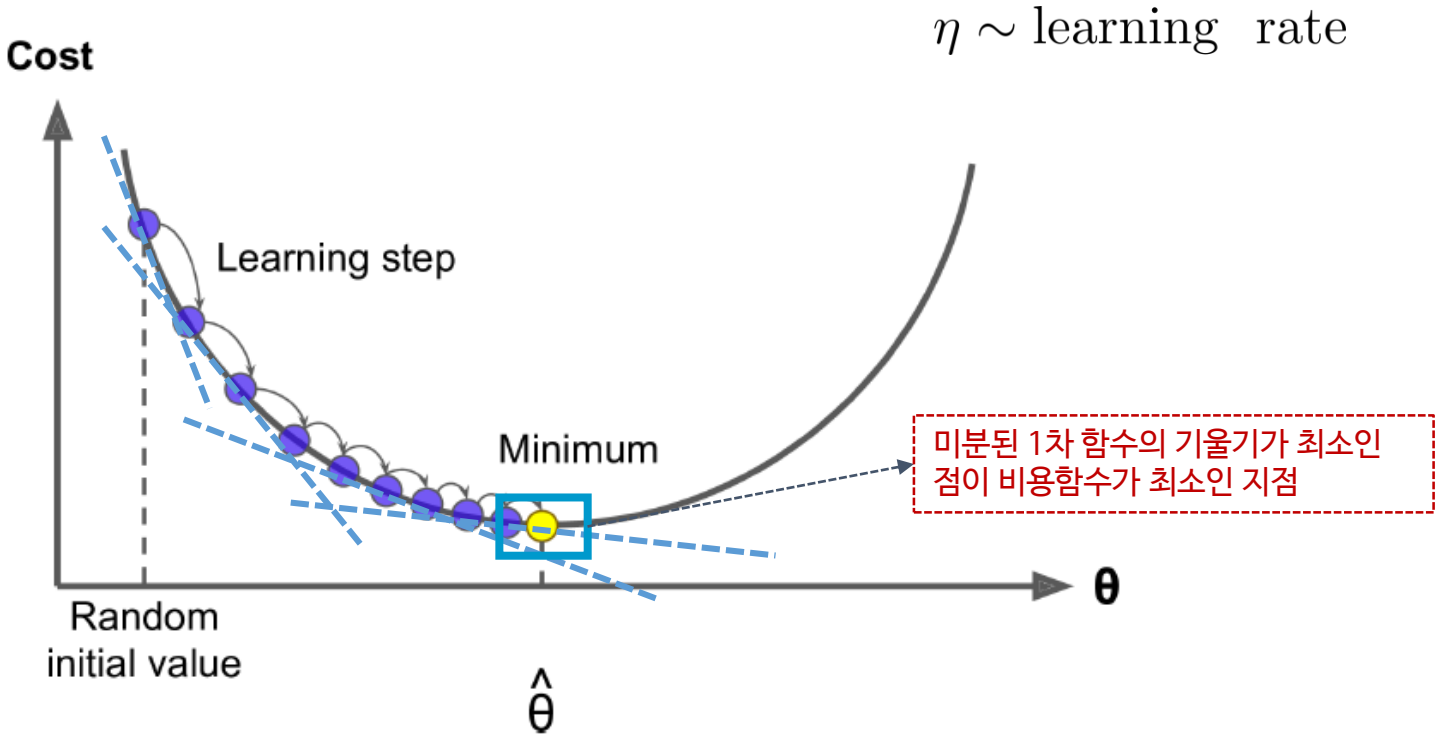
RSS = (#1 주택 가격 – $(w_0+w_1$ #1 주택크기)^2
+ (#2 주택 가격 – $(w_0+w_1$ #2 주택크기)^2
+ (#3 주택 가격 – $(w_0+w_1$ #3 주택크기)^2
+ … (모든 학습 데이터에 대해 RSS 수행)

회귀에서 RSS는 비용(cost)이며, 혹은 비용함수, 혹은 손실(loss) 함수라고도 함

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

비용 최소화 _ 경사 하강법(1)

- 경사 하강법 (Gradient Descent)



비용 최소화 _ 경사 하강법(2)

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

$$\frac{\partial \mathrm{RSS}(w_0, w_1)}{\partial w_1} = \frac{2}{N}\sum_{i=1}^{N} -x_i \times (y_i - (w_o + w_1 \times x_i)) = -\frac{2}{N}\sum_{i=1}^{N} x_i * (\mathrm{real_i} - \mathrm{pred_i})$$

$$\frac{\partial \mathrm{RSS}(w_0, w_1)}{\partial w_0} = \frac{2}{N}\sum_{i=1}^{N} -(y_i - (w_o + w_1 \times x_i)) = -\frac{2}{N}\sum_{i=1}^{N} (\mathrm{real_i} - \mathrm{pred_i})$$

$$w = w - \eta \frac{2}{N}\sum_{i=1}^{N} (\mathrm{real_i} - \mathrm{pred_i})$$

학습률 도입 $\eta \sim$ learning rate

비용 최소화 _ 경사 하강법(3)

- 데이터를 학습하는 방법으로 배치(Batch)
  - ✓ 매 경사 하강법 스탭에서 전체 훈련 세트 (X)에 대해 계산
  - ✓ 전체 입력 데이터를 훈련에 사용해서 큰 메모리 필요, 계산 시간 오래 걸림

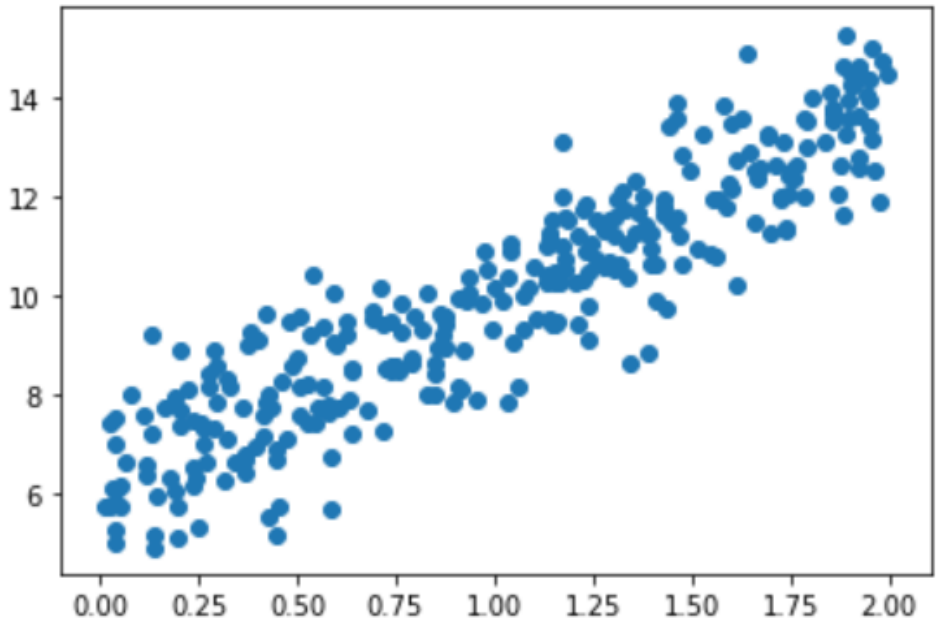
$$\frac{\partial}{\partial \theta_j}\mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^T\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\mathrm{MSE}\left(\boldsymbol{\theta}\right)$$

비용 최소화 _ 경사 하강법(4)

경사 하강법을 이용한 회귀의 간단한 예제
200개의 데이터을 이용하여 y=4x+6에 근사하시오.

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

np.random.seed(0)
X = 2 * np.random.rand(300,1)
y = 6 +4 * X+np.random.randn(300,1)

plt.scatter(X, y)
```



비용 최소화 _ 경사 하강법(5)

```
# 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
```

비용 최소화 _ 경사 하강법(6)

```
# 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
```

비용 최소화 _ 경사 하강법(7)

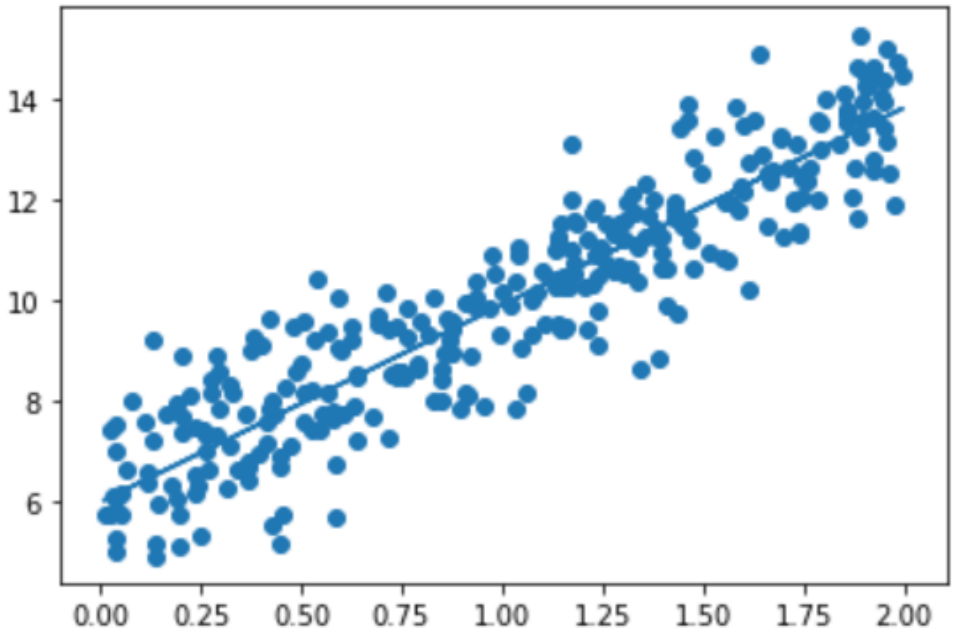
```
def get_cost(y, y_pred):
    N = len(y)
    cost = np.sum(np.square(y - y_pred))/N
    return cost

w1, w0 = gradient_descent_steps(X, y, iters=1000)
print("w1:{0:.3f} w0:{1:.3f}".format(w1[0,0], w0[0,0]))

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

```
w1:3.919 w0:5.964
Gradient Descent Total Cost:0.9837
```

비용 최소화 _ 경사 하강법(8)

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

```
[<matplotlib.lines.Line2D at 0x1ce5ef1d208>]
```



전체 데이터를 사용하는 것은 학습에 시간이 많이 소요된다.

비용 최소화 _ 경사 하강법(9)

실전에서는 확률론적 경사 하강법을 사용한다.

```
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
```

비용 최소화 _ 경사 하강법(10)

```
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: 3.973 w0: 5.92
Stochastic Gradient Descent Total Cost:0.9869
```

확률론적 경사 하강법에 따른 비용함수 계산.
배치 방법의 비용함수는 98.37%이지만,
데이터 개수가 300일 경우 확률론적 경사 하강법이 약간 우수한 성능을 보임.

실전에서는 정확도 보다는 빅데이터를 처리할 경우 전체 데이터를
처리하는데 필요한 계산시간이 중요해진다.

신경망에서는 GPU 고속처리가 필요하며,
Mini-Batch Stocastic Gradient Descent 방법 등이 사용된다.

선형회귀 (Linear Regression)

- 선형 모델은 입력 피처의 웨이트 된 합과 바이어스 항으로 구성

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

$$\hat{y} = h_{\theta}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

벡터 표현

hypothesis 함수

모델 파라미터

- 모델 학습 데이터를 가장 잘 피팅(fit)하는 파라미터를 찾자.
  ✓ 오차를 이용. MSE, RMSE(Root Mean Square Error)가 최소가 되도록

$$\mathrm{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

벡터 표현, m은 입력 데이터 개수

$$\hat{\boldsymbol{\theta}} = \left(\mathbf{X}^T \mathbf{X}\right)^{-1} \mathbf{X}^T \mathbf{y}$$

타겟 값

경사 하강법 – 확률론적 (1)

```
import sys
assert sys.version_info >= (3, 5)

import sklearn
assert sklearn.__version__ >= "0.20"

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(99)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

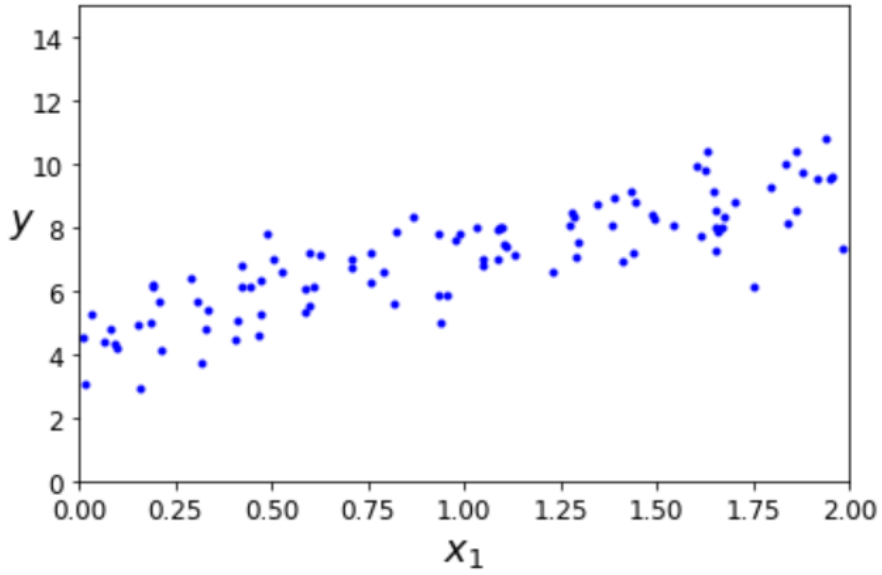
Setup

- 공통 모듈을 임포트
- 파이썬 3.5 이상
- 사이킷런 ≥0.20.

경사 하강법 – 확률론적 (2)

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
save_fig("generated_data_plot")
plt.show()
```

```
Saving figure generated_data_plot
```



경사 하강법 – 확률론적 (3)

```
X_b = np.c_[np.ones((100, 1)), X]
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
theta_best
```

역행렬을 구하고,
dot()로 행렬 곱셈 수행

```
array([[4.6263185 ],
       [2.46782729]])
```

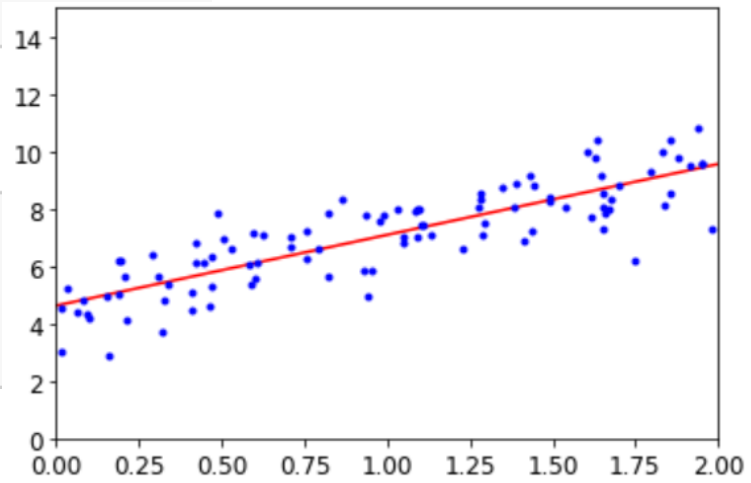
thea_0

thea_1

```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta_best)
y_predict
```

```
array([[4.6263185 ],
       [9.56197309]])
```

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```



경사 하강법 – 확률론적 (4)

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
```

```
(array([4.6263185]), array([[2.46782729]]))
```

```
lin_reg.predict(X_new)
```

```
array([[4.6263185 ],
       [9.56197309]])
```

최소제곱함수
(least square)

```
theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
theta_best_svd
```

```
array([[4.6263185 ],
       [2.46782729]])
```

SVM 분류기

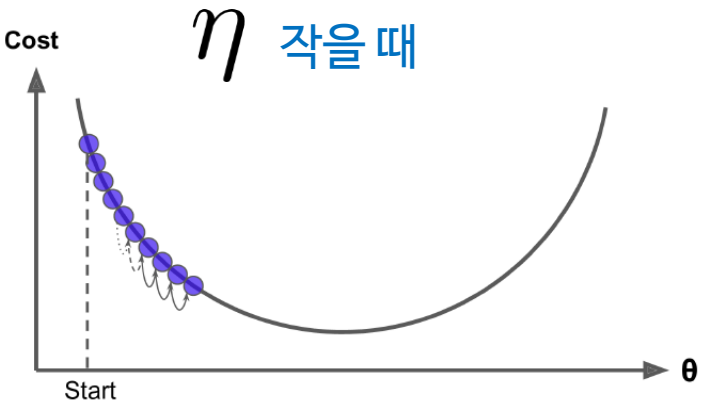
```
np.linalg.pinv(X_b).dot(y)
```

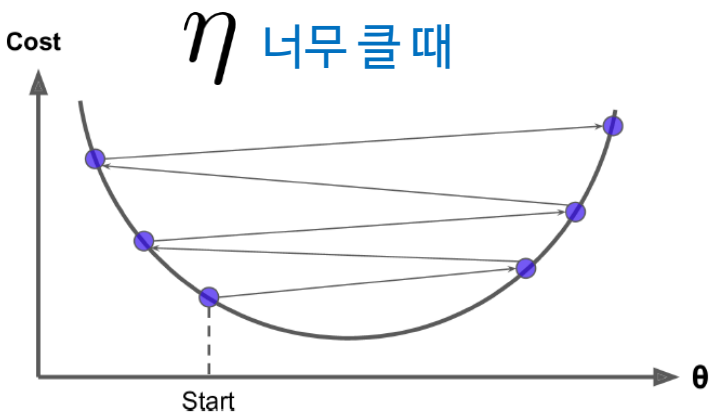
```
array([[4.6263185 ],
       [2.46782729]])
```

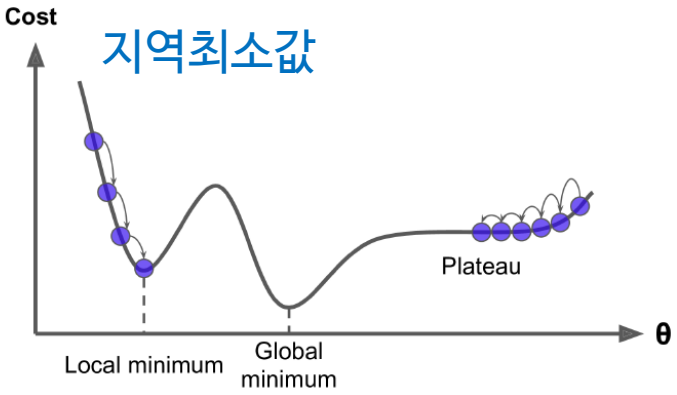
참고, 역행렬 계산은 행렬 크기의 O(3) 소요

역 벡터를 구하는 함수
Pseudoinverse (Moore-Ponrose inverse)

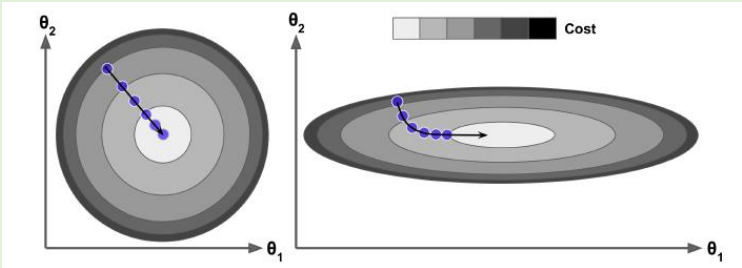
경사하강 학습률(eta)







전역 최소값



- 특성이 다르면 수렴하는 데 시간이 오래 걸림
- 반드시 특성들이 같은 스케일을 갖도록
- StandardScaler 사용 권장

경사 하강법 – 확률론적 (5)

배치(Batch) 경사 하강법

```
eta = 0.1
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients

theta
```

```
array([[4.6263185 ],
       [2.46782729]])
```

```
X_new_b.dot(theta)
```

```
array([[4.6263185 ],
       [9.56197309]])
```

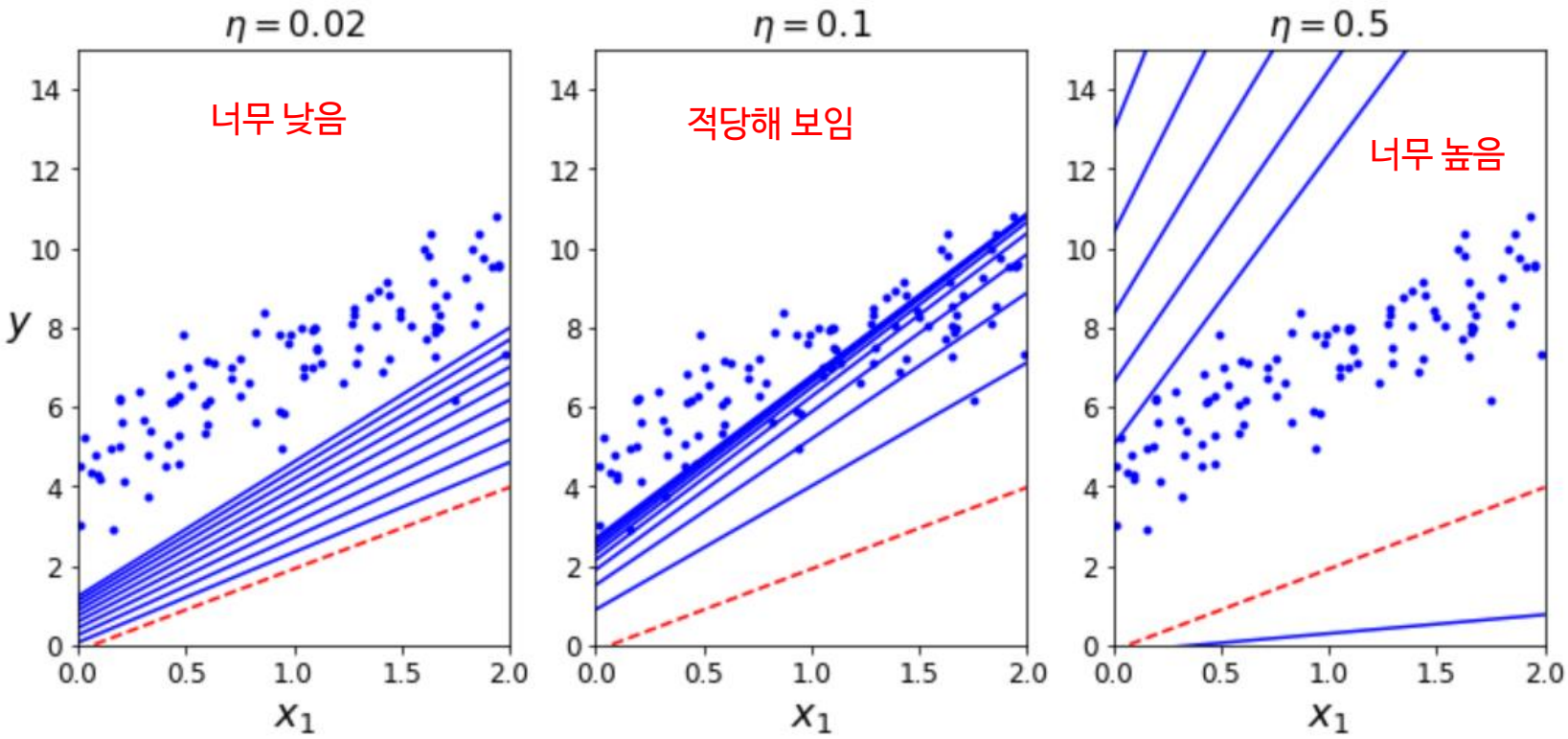
경사 하강법 – 확률론적 (6)

배치(Batch) 경사 하강법

```
theta_path_bgd = []

def plot_gradient_descent(theta, eta, theta_path=None):
    m = len(X_b)
    plt.plot(X, y, "b.")
    n_iterations = 1000
    for iteration in range(n_iterations):
        if iteration < 10:
            y_predict = X_new_b.dot(theta)
            style = "b-" if iteration > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
        theta = theta - eta * gradients
        if theta_path is not None:
            theta_path.append(theta)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 2, 0, 15])
    plt.title(r"$\eta = {}$".format(eta), fontsize=16)
```

경사 하강법 – 확률론적 (7)

배치(Batch) 경사 하강법



적절한 학습률은 어떻게 구할까?
1) 답은 그리드서치 하면 된다. 하지만 무엇이 문제일까?
2) 대안은 허용오차를 설정하여 조기 종료

경사 하강법 – 확률론적 (8)

- **확률적 경사하강법(Stochastic Gradient Descent) 장점**
  - ✓ **매 스텝에서 딱 1개의 샘플을 무작위로 선택하여 하나의 샘플의 경사하강 계산**
  - ✓ **계산이 확실히 빠르다**
  - ✓ **큰 훈련 데이터도 학습이 가능하다**
  - ✓ **배치 경사하강법보다 불안정한 계산 결과**
    - 최소점 근처에서 비교적 큰 요동이 발생함
  - ✓ **전역 최소값을 찾을 가능성이 배치 경사하강법보다 높다.**
  - ✓ **무작위성은 지역 최소값을 탈출시켜서 좋지만, 전역에는 다다르지 못하게 한다**
    - 해결방법은 어닐링(Annealing)

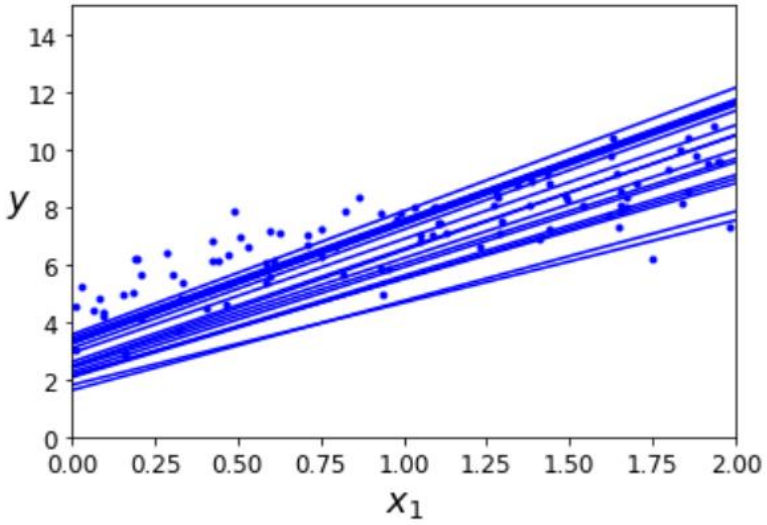
경사 하강법 – 확률론적 (9)

```
n_epochs = 50
t0, t1 = 5, 50
def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)

for epoch in range(n_epochs):
    for i in range(m):
        if epoch == 0 and i < 20:
            y_predict = X_new_b.dot(theta)
            style = "b-" if i > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
        theta_path_sgd.append(theta)
```

무작위 반복횟수

입력 데이터 개수

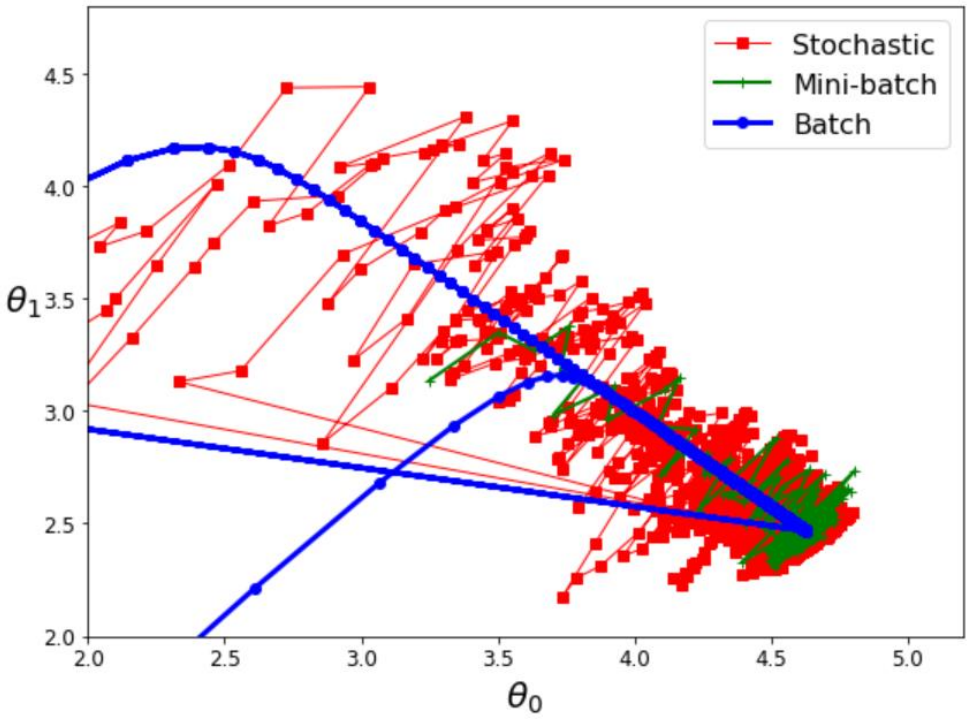
경사 하강법 – 확률론적 (10)



무작위 선택으로 한 샘플은 주어진
한 에포크에서 여러 번 선택될 수 있음

경사 하강법 – 확률론적 (11)

- 미니배치 장점
  - ✓ 각 스텝에서 일부 미니배치라 부르는 작은 데이터 세트로 학습
  - ✓ 미니배치 경사하강법은 확률적 경사하강법 보다 덜 불규칙하다

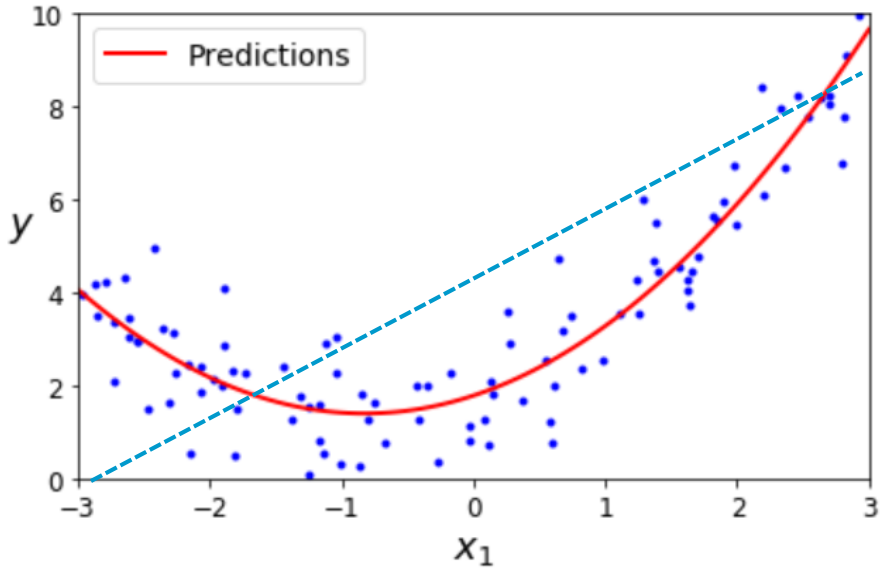


다항회귀와 과적합(1)

- 다항회귀는 2차 3차 방정식과 같은 다항식으로 표현
  ✓ 다항회귀는 선형회귀이지, 비선형 회귀는 아니다.

$$y = w_0 + w_1 * x_1 + w_2 * x_2 + w_3 * x_1 * x_2 + w_4 * x_1^2 + w_5 * x_2^2$$

target y에 대하여 단순 선형 회귀 직선형 보다 다항 회귀 곡선형으로 표현이 더 예측 성능이 높다



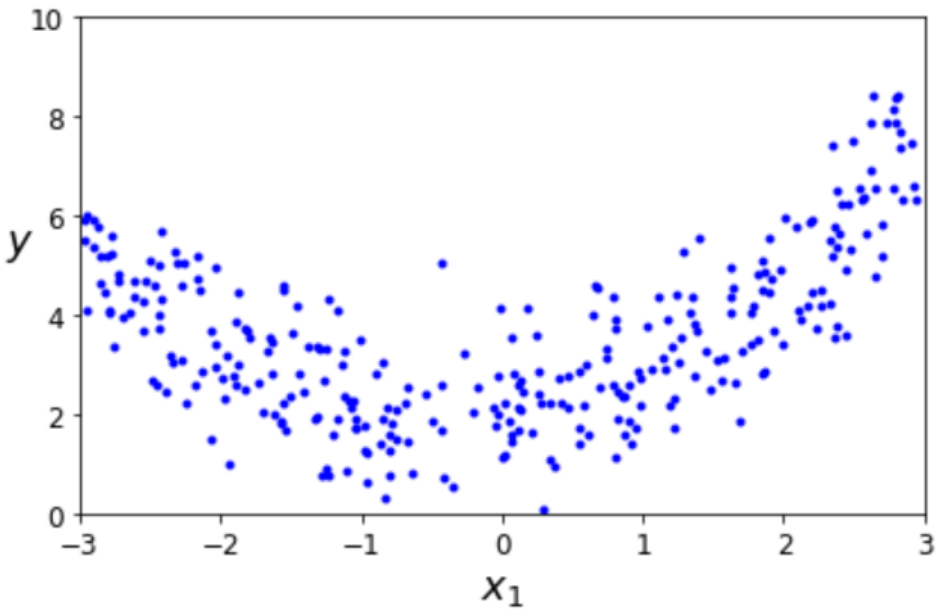
다항회귀와 과적합(2)

```
import numpy as np
import numpy.random as rnd
np.random.seed(42)

m = 300
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 +  0.3*X + 2 + np.random.randn(m, 1)
```

```
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
save_fig("quadratic_data_plot")
plt.show()
```

$$y = 2 + 0.3 * x + 0.5 * x^2$$

$$x = -3 + 6 * \text{np.random.rand}(300, 1)$$



다항회귀와 과적합(3)

```
from sklearn.preprocessing import PolynomialFeatures

poly_features = PolynomialFeatures(degree=2, include_bias=False)

X_poly = poly_features.fit_transform(X)

print('x[0]=',X[0])
print('x_ploy[0]=',X_poly[0])
```

이 클래스를 통해서 피처를 다항식 피처로 변환함

```
x[0]= [-0.75275929]
x_ploy[0]= [-0.75275929  0.56664654]
```

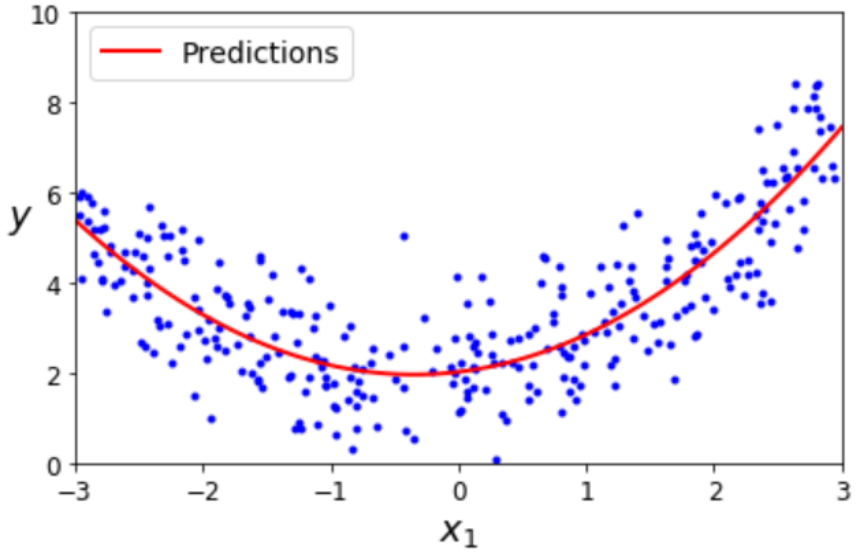
```
lin_reg = LinearRegression()

lin_reg.fit(X_poly, y)

lin_reg.intercept_, lin_reg.coef_
```

```
(array([2.02145529]), array([[0.34309641, 0.48892735]]))
```

다항회귀와 과적합(4)

```
X_new=np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)
plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([-3, 3, 0, 10])
save_fig("quadratic_predictions_plot")
plt.show()
```

```
Saving figure quadratic_predictions_plot
```



다항회귀와 과적합(5)

다항함수의 과적합을 알아보기 위해서, 다항식의 개수를 300, 2, 1개로 수정함

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

for style, width, degree in (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline([
            ("poly_features", polybig_features),
            ("std_scaler", std_scaler),
            ("lin_reg", lin_reg),
        ])
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)
```

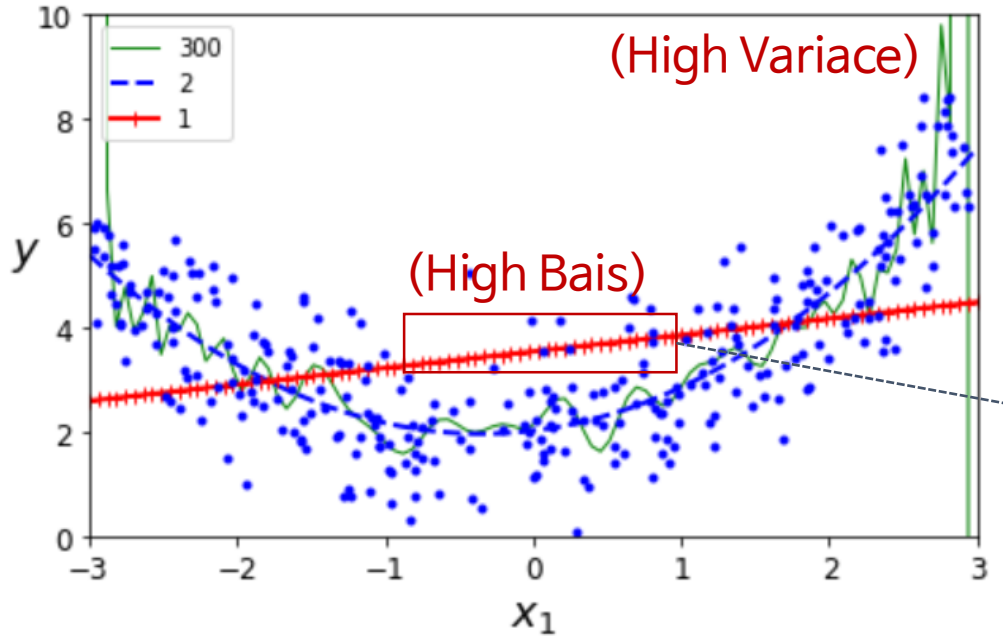
다항회귀와 과적합(6)

```
plt.plot(X, y, "b.", linewidth=3)
plt.legend(loc="upper left")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
save_fig("high_degree_polynomials_plot")
plt.show()
```

```
Saving figure high_degree_polynomials_plot
```

고차 다항회귀 모델의 과적합

고차 다항회귀를 적용하면 선형회귀보다 더 훈련데이터를 잘 맞춤



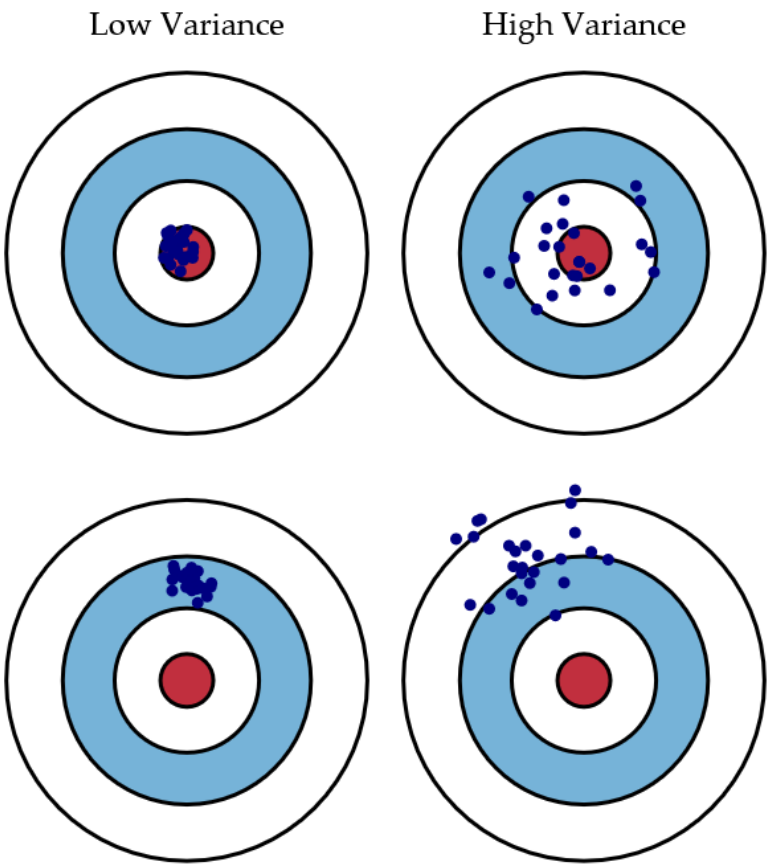
degree가 300일때:
데이터 하나 하나의 특성이 반영된 복잡한 모델.
고분산(High Variace)

degree가 1일때:
지나치게 한방향으로 치우친 경향이 있음.
이 모델은 고편향 (High Bais)

다항회귀와 과적합(7)

저편향/저분산
- 예측 결과가 실제 결과에 매우 잘 접근.
- 아주 뛰어난 성능을 보여줌.

고편향/저분산
- 실제 결과에서 벗어나면서도, 예측이 특정부분에 집중되어 있다.



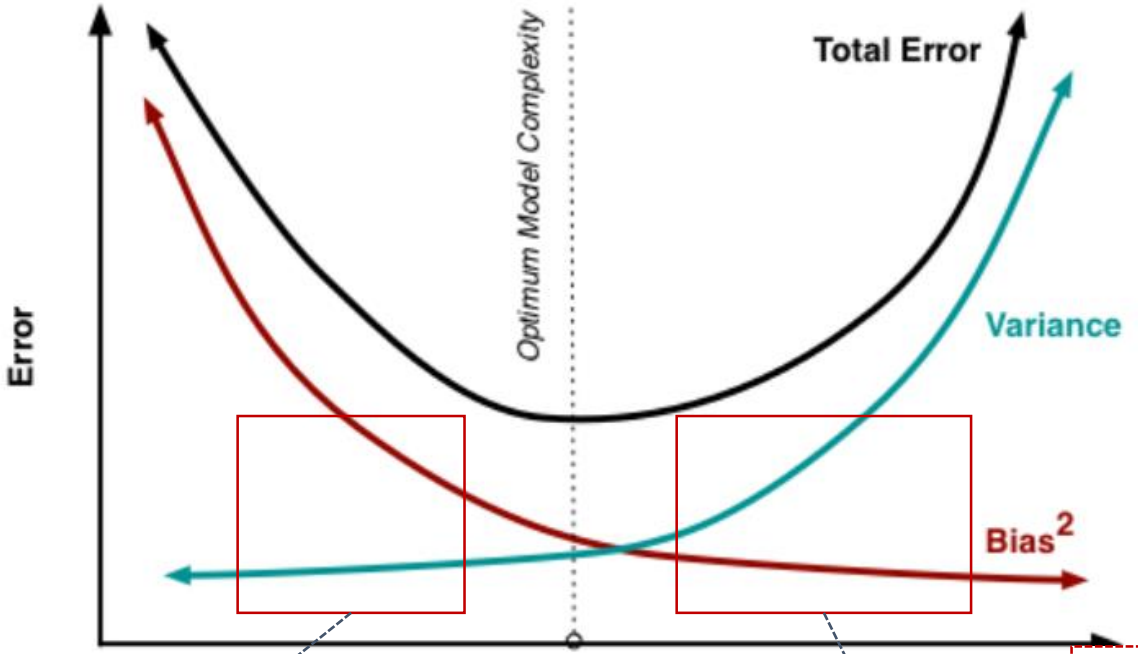
저편향/고분산
- 예측이 실제 결과 중심으로 넓게 분포

고편향/고분산
- 예측 결과를 벗어나면서도 실제 결과 중심으로 넓게 분포

http://scott.fortmann-roe.com/docs/BiasVariance.html

다항회귀와 과적합(8)

편향-분산 트레이드오프(trade off)



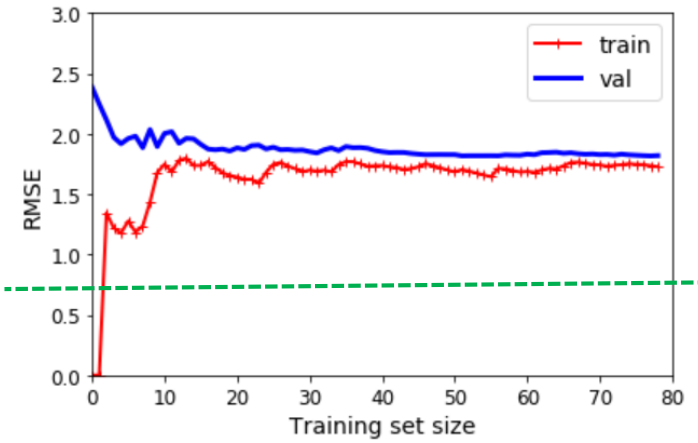
과소적합(Underfit)
고편향-저분산

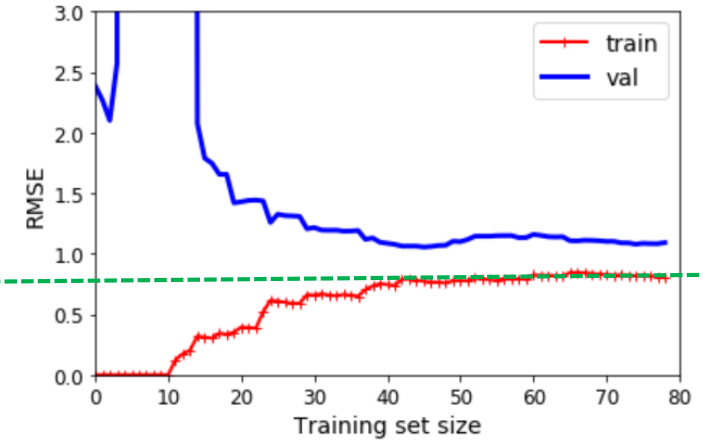
과적합(Overfit)
저편향-고분산

전체 오류 최소점이 가장
효율적인 예측 모델이 된다.

다항회귀와 과적합(8)

과적합을 해소 하기 위해서는 데이터를 추가하면서 오차의 추이를 살펴봄





과소적합 대표사례

- RMSE 값이 높다
- 데이터가 증가할수록 검증과 훈련은 좁아짐

과대적합 대표사례

- RMSE 값이 상대적으로 작다 (이미 훈련됨)
- 데이터가 증가해도 검증과 훈련 간격은 감소 없음

보스턴 주택가격예측(1)

- 회귀 평가 지표
  - ✓ MSE
  - ✓ MAE
  - ✓ RMSE
  - ✓ R2

$$\mathrm{MAE} = \frac{1}{N}\sum_{i=1}^{N}|Y_i - \hat{Y}_i|$$

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

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

$$\mathrm{R}^2 = \frac{\text{pred Variance}}{\text{real Variance}}$$

보스턴 주택가격예측(2)

sklearn.linear_model.LinearRegression

class `sklearn.linear_model.` **`LinearRegression`**(*fit_intercept=True, normalize=False, copy_X=True, n_jobs=None*) [source]

Ordinary least squares Linear Regression.

LinearRegression fits a linear model with coefficients w = (w1, ..., wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

Parameters:

fit_intercept : *bool, optional, default True*

Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered).

보스턴 주택가격예측(3)

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
%matplotlib inline

from scipy import stats

from sklearn.datasets import load_boston

# 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)
```

보스턴 주택가격예측(4)

- CRIM: 지역별 범죄 발생률
- ZN: 25,000평방피트를 초과하는 거주 지역의 비율
- INDUS: 비상업 지역 넓이 비율
- CHAS: 찰스강에 대한 더미 변수(강의 경계에 위치한 경우는 1, 아니면 0)
- NOX: 일산화질소 농도
- RM: 거주할 수 있는 방 개수
- AGE: 1940년 이전에 건축된 소유 주택의 비율
- DIS: 5개 주요 고용센터까지의 가중 거리
- RAD: 고속도로 접근 용이도
- TAX: 10,000달러당 재산세율
- PTRATIO: 지역의 교사와 학생 수 비율
- B: 지역의 흑인 거주 비율
- LSTAT: 하위 계층의 비율
- MEDV: 본인 소유의 주택 가격(중앙값)

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

보스턴 주택가격예측(5)

```
bostonDF.info()
```

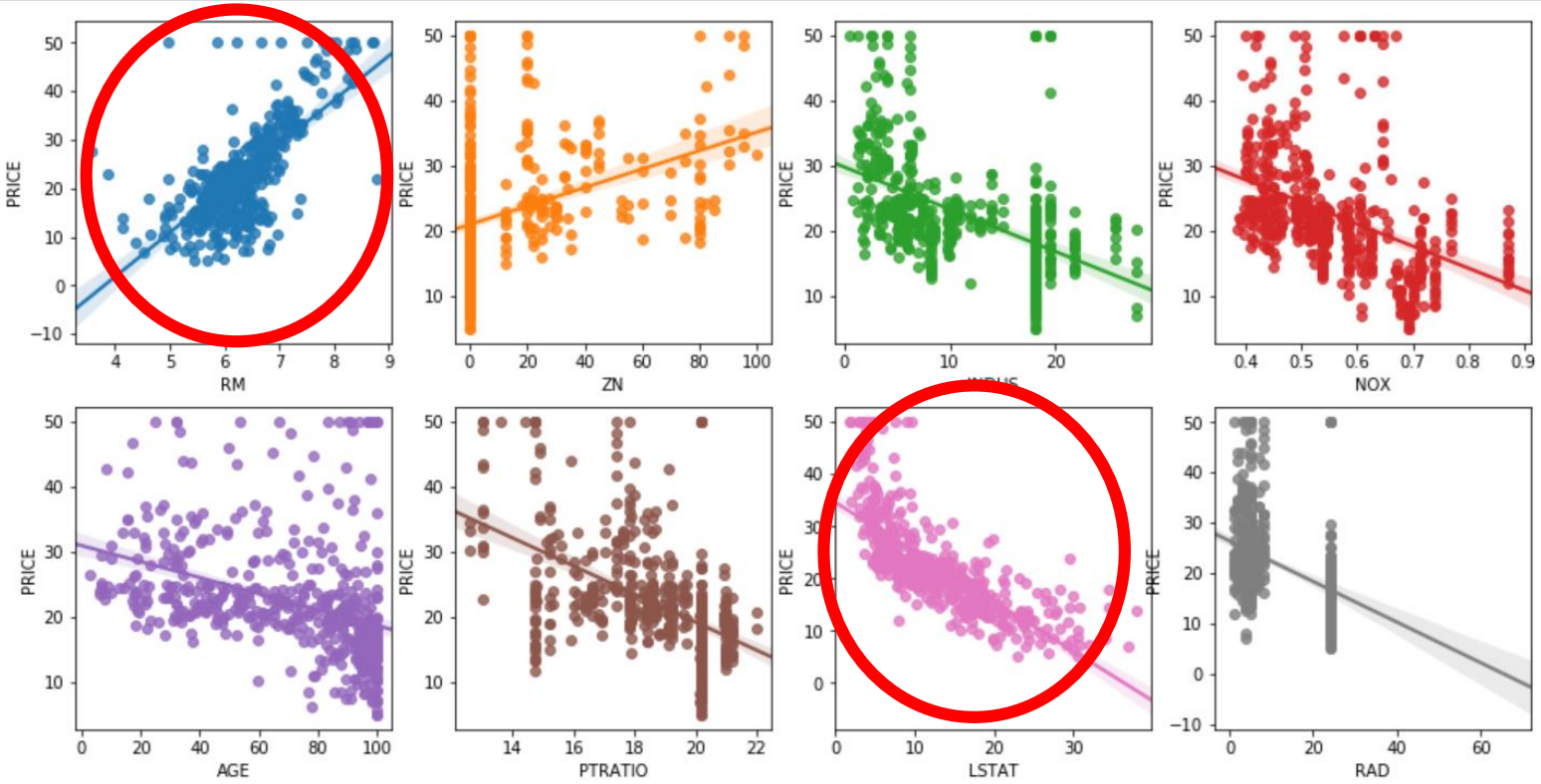
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
CRIM       506 non-null float64
ZN         506 non-null float64
INDUS      506 non-null float64
CHAS       506 non-null float64
NOX        506 non-null float64
RM         506 non-null float64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

보스턴 주택가격예측(6)

```
# 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])
```

8개 항목에 대한 집 값 변화 시각화, 2x4 그림 포멧

Seaborn의 regplot() 산점도와 함께 선형 회귀 직선을 그려줌



특징 2가지: RM 방의 크기가 클수록 높은 가격을 받음, LSTAT(하위계층의 비율)은 적을 수록 높은 가격

보스턴 주택가격예측(7)

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

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

X_train , X_test , y_train , y_test = train_test_split(X_data , y_target ,test_size=0.3, random_state=156)

# Linear Regression OLS로 학습/예측/평가 수행.
lr = LinearRegression()
lr.fit(X_train ,y_train )
y_preds = lr.predict(X_test)
mse = mean_squared_error(y_test, y_preds)
rmse = np.sqrt(mse)

print('MSE : {0:.3f} , RMSE : {1:.3F}'.format(mse , rmse))
print('Variance score : {0:.3f}'.format(r2_score(y_test, y_preds)))
```

R2 스코어 값

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

보스턴 주택가격예측(8)

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

coef_ 속성은 회귀 계수 값만 가지고 있으므로, 이를 피처별 회귀 계수 값으로 다시 매핑하고, 높은 값 순서로 출력.

```
절편 값: 40.9955951721646
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
B           0.0
TAX        -0.0
AGE         0.0
INDUS       0.0
CRIM       -0.1
LSTAT      -0.6
PTRATIO    -0.9
DIS        -1.7
NOX       -19.8
dtype: float64
```

RM(거주 할 수 있는 방의 개수)은 양수로 계수가 가장 크다

NOX(일산화질소의 농도) 음수(-) 값이 매우 크다

보스턴 주택가격예측(9)

교차 검증 5개의 폴드 세트를 이용함.

```
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
```

교차 검증 평균 RMSE는 5.83%이고, 처음 계산 한 것은 4.16%이다.
작은 숫자 일수록 예측 성능이 좋은 것이다.
교차 검증 평균 RMSE 값이 더 높다는 것은
무엇을 의미하는 것일까? (답은?)

규제 선형 모델(1)

- 좋은 선형모델
  - ✓ 모델의 비용함수인 MSE(혹은 RSS)를 최소화 하는 방법
  - ✓ 과적합을 방지하기 위해 회귀 계수 값이 커지는 것을 제어 하여 균형인 모델
- 비용함수의 목표

$$\text{Cost}(w) = \min(\text{MSE}(w) + \alpha * ||w||_2^2)$$

- 규제
  - ✓ 릿지 회귀는 L2에 규제를 적용함 즉 규제를 다음 항목에 적용.
  - ✓ 라쏘 회귀는 L1에 규제를 적용.

$$\alpha * ||w||_1)$$

$$\alpha * ||w||_2^2)$$

최적 모델을 위한 비용함수 구성 요소 = 학습데이터 잔차 오차 최소화 + 회귀 계수 크기를 규제(제어)

규제 선형 모델(2)

릿지회귀(L2 규제) $\alpha * ||w||_2^2)$

규제 선형 모델 적용 : 보스턴 주택 가격 데이터

릿지규제 적용 Ridge

```
# 앞의 LinearRegression예제에서 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

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 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))
```

릿지 회귀에서 alpha 값은 10 사용 했는데, 왜 일까요? 다른 값은?

교차 검증 횟수 5회

```
 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
```

릿지회귀 (alpha=10) RMSE=5.51%로 성능이 더 좋다?

규제 선형 모델(3)

릿지 회귀에서 alpha의 증가에 따른 RMSE 변화와 회귀계수의 변화

```
# Ridge에 사용될 alpha 파라미터의 값들을 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))
```

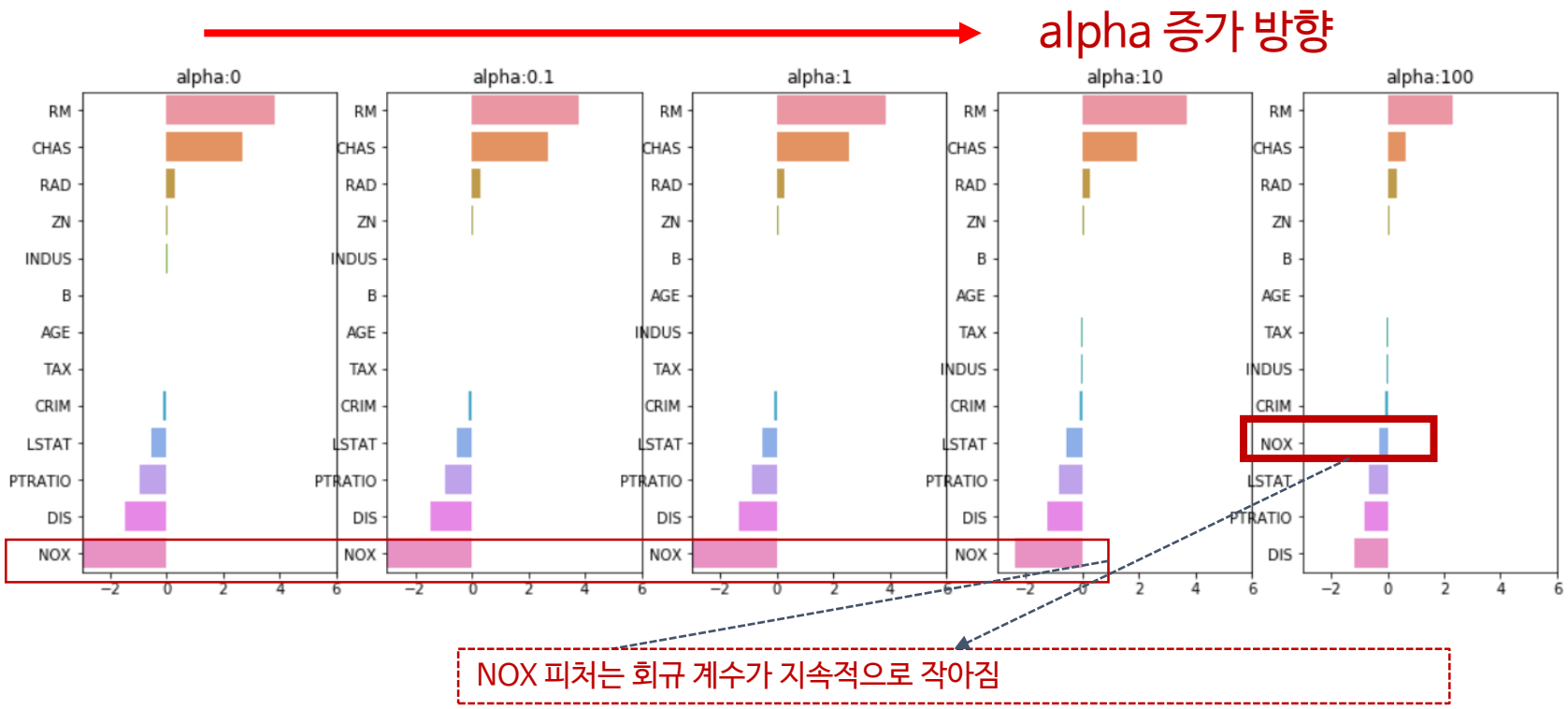
```
alpha 0 일 때 5 folds 의 평균 RMSE : 5.836
alpha 0.1 일 때 5 folds 의 평균 RMSE : 5.796
alpha 1 일 때 5 folds 의 평균 RMSE : 5.659
alpha 10 일 때 5 folds 의 평균 RMSE : 5.524
alpha 100 일 때 5 folds 의 평균 RMSE : 5.332
```

규제 선형 모델(4)

```
# 각 alpha에 따른 회귀 계수 값을 시각화하기 위해 5개의 열로 된 맷플롯립 축 생성
fig , axs = plt.subplots(figsize=(18,6) , nrows=1 , ncols=5)
# 각 alpha에 따른 회귀 계수 값을 데이터로 저장하기 위한 DataFrame 생성
coeff_df = pd.DataFrame()

# alphas 리스트 값을 차례로 입력해 회귀 계수 값 시각화 및 데이터 저장. pos는 axis의 위치 지정
for pos , alpha in enumerate(alphas) :
    ridge = Ridge(alpha = alpha)
    ridge.fit(X_data , y_target)
    # alpha에 따른 피처별 회귀 계수를 Series로 변환하고 이를 DataFrame의 컬럼으로 추가.
    coeff = pd.Series(data=ridge.coef_ , index=X_data.columns )
    colname='alpha:'+str(alpha)
    coeff_df[colname] = coeff
    # 막대 그래프로 각 alpha 값에서의 회귀 계수를 시각화. 회귀 계수값이 높은 순으로 표현
    coeff = coeff.sort_values(ascending=False)
    axs[pos].set_title(colname)
    axs[pos].set_xlim(-3,6)
    sns.barplot(x=coeff.values , y=coeff.index, ax=axs[pos])

# for 문 바깥에서 맷플롯립의 show 호출 및 alpha에 따른 피처별 회귀 계수를 DataFrame으로 표시
plt.show()
```

규제 선형 모델(5)

릿지 회귀에서 alpha의 증가에 따른 회귀 계수의 값은 지속적으로 작아짐.



규제 선형 모델(6)

```
ridge_alphas = [0 , 0.1 , 1 , 10 , 100]
sort_column = 'alpha:'+str(ridge_alphas[0])
coeff_df.sort_values(by=sort_column, ascending=False)
```

	alpha:0	alpha:0.1	alpha:1	alpha:10	alpha:100
RM	3.804752	3.813177	3.849256	3.698132	2.331966
CHAS	2.688561	2.671849	2.554221	1.953452	0.638647
RAD	0.305655	0.303105	0.289650	0.279016	0.314915
ZN	0.046395	0.046546	0.047414	0.049547	0.054470
INDUS	0.020860	0.016293	-0.008547	-0.042745	-0.052626
B	0.009393	0.009449	0.009754	0.010117	0.009471
AGE	0.000751	-0.000212	-0.005368	-0.010674	0.001230
TAX	-0.012329	-0.012415	-0.012907	-0.013989	-0.015852
CRIM	-0.107171	-0.106612	-0.103622	-0.100352	-0.101451
LSTAT	-0.525467	-0.526678	-0.534072	-0.560097	-0.661312
PTRATIO	-0.953464	-0.941449	-0.876633	-0.798335	-0.829503
DIS	-1.475759	-1.459773	-1.372570	-1.248455	-1.153157
NOX	-17.795759	-16.711712	-10.793436	-2.374959	-0.263245

(소결)
릿지(Ridge) 회귀의 경우,
alpha 값의 증가에 따라서 회귀 계수가 지속적으로 작아지고 있음을 알 수 있다.

하지만, Ridge 회귀의 경우는 회귀 계수를 0으로 만들지 않는다.

규제 선형 모델(7)

- 라쏘(Lasso) 회귀
  - ✓ 라쏘(Lasso) 회귀는 w에 페널티를 부여하는 것으로 L1 규제를 회귀에 적용
  - ✓ 릿지회귀가 L2 규제로 회귀 계수를 크게 감소하는 역할이라면,
  - ✓ L1 라쏘 규제는 불필요한 회귀 계수를 급격하게 감소하여 0으로 만듬
  - ✓ 라쏘는 꼭 필요한 피처(특성)만 선택하여 남김

규제 선형 모델(8)

라쏘에서 alpha를 변화시켜서 출력을 살펴보는 함수를 만들자.

```
from sklearn.linear_model import 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):
    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 , y_target)
        # alpha에 따른 피처별 회귀 계수를 Series로 변환하고 이를 DataFrame의 컬럼으로 추가.
        coeff = pd.Series(data=model.coef_ , index=X_data.columns )
        colname='alpha:'+str(param)
        coeff_df[colname] = coeff
    return coeff_df
# end of get_linear_regre_eval
```

규제 선형 모델(9)

라쏘에 대한 출력을 함

```
# 라쏘에 사용될 alpha 파라미터의 값들을 정의하고 get_linear_reg_eval() 함수 호출
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
```

alpha=0.07 일대 가장 성능이 좋은 RMSE 5.61%를 얻음.

랏소보다 5.3% 높지만,
선형회귀 5.8% 보다 작은 값

규제 선형 모델(10)

라쏘에서 alpha를 증가할 때 회귀 계수의 변화는 줄어듬. 회기계수가 0도 있음.

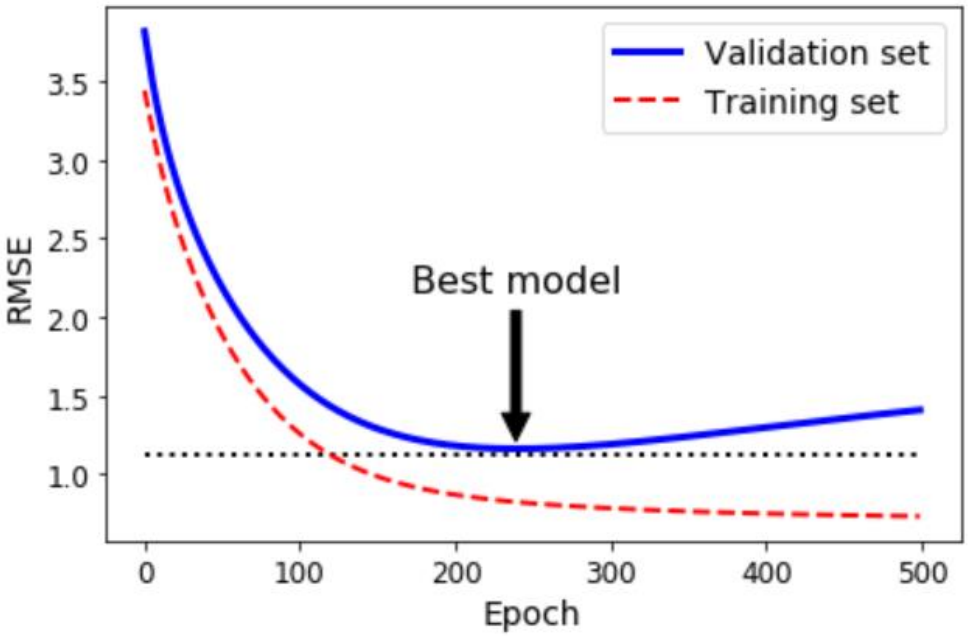
	alpha:0.07	alpha:0.1	alpha:0.5	alpha:1	alpha:3
RM	3.789725	3.703202	2.498212	0.949811	0.000000
CHAS	1.434343	0.955190	0.000000	0.000000	0.000000
RAD	0.270936	0.274707	0.277451	0.264206	0.061864
ZN	0.049059	0.049211	0.049544	0.049165	0.037231
B	0.010248	0.010249	0.009469	0.008247	0.006510
NOX	-0.000000	-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

회귀 계수가 alpha=0.07 부터 0임

INDUS도 0으로 바뀜
CHAS도 0로 바뀜

조기종류 (Early Stopping)

- 반복 학습 알고리즘을 규제하는 방법중에 조기종료
  - ✓ 검증 에러가 최소값에 도달하면 바로 훈련을 중지함
  - ✓ 훌륭한 공짜 점심 (by Geoffrey Hinton)



검증 에러가 일정시간 동안 최소값보다 클 때 학습을 멈춤

(프로젝트) 선형회귀 최적 적용(1)

- **선형 회귀 모형 주의사항**
  - ✓ **피처와 타깃 값이 정규 분포를 매우 선호함**
  - ✓ **특히, 타싯이 정규 분포가 아닐 경우 성능에 매우 부정적인 영향을 끼침**
- **선형 회귀를 위해 데이터 처리**
  - ✓ **스케일링과 정규화를 하는 것이 일반적이다.**
  - ✓ **스케일링과 정규화를 한다고 해서 예측 성능이 좋아지는 것은 아니고,**
    - 사이킷런에서는 StandaradScaler 클래스를 이용하여
    - 타깃 값은 일반적으로 로그(log) 변환을 적용함

(프로젝트) 선형회귀 최적 적용(2)

- 프로젝트 (혹은 숙제)
- 최적 선형 회귀
  - ✓ 보스턴 주택 데이터 셋트를 사용하여, 최적의 선형회귀를 구해라
  - ✓ 정규화/스케일 적용
  - ✓ 최적의 alpha에서 최고 성능의 RMSE를 구하여라.

(프로젝트) 선형회귀 최적 적용(3)

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, PolynomialFeatures

# method는 표준 정규 분포 변환(Standard), 최대값/최소값 정규화(MinMax), 로그변환(Log) 결정
# p_degree는 다항식 특성을 추가할 때 적용. p_degree는 2이상 부여하지 않음.
def get_scaled_data(method='None', p_degree=None, input_data=None):
    if method == 'Standard':
        scaled_data = StandardScaler().fit_transform(input_data)
    elif method == 'MinMax':
        scaled_data = MinMaxScaler().fit_transform(input_data)
    elif method == 'Log':
        scaled_data = np.log1p(input_data)
    else:
        scaled_data = input_data

    if p_degree != None:
        scaled_data = PolynomialFeatures(degree=p_degree,
                                         include_bias=False).fit_transform(scaled_data)

    return scaled_data
```

(프로젝트) 선형회귀 최적 적용(4)

```
# Ridge의 alpha값을 다르게 적용하고 다양한 데이터 변환방법에 따른 RMSE 추출.
alphas = [0.1, 1, 5, 10]

#변환 방법은 모두 6개, 원본 그대로, 표준정규분포, 표준정규분포+다항식 특성
# 최대/최소 정규화, 최대/최소 정규화+다항식 특성, 로그변환

scale_methods=[(None, None), ('Standard', None), ('Standard', 2),
               ('MinMax', None), ('MinMax', 2), ('Log', None)]

for scale_method in scale_methods:
    X_data_scaled = get_scaled_data(method=scale_method[0], p_degree=scale_method[1],
                                    input_data=X_data)
    print('₩n## 변환 유형:{0}, Polynomial Degree:{1}'.format(scale_method[0], scale_method[1]))
    get_linear_reg_eval('Ridge', params=alphas, X_data_n=X_data_scaled,
                        y_target_n=y_target, verbose=False)
```

(프로젝트) 선형회귀 최적 적용(5)

```
## 변환 유형:None, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.788
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.653
alpha 5일 때 5 폴드 세트의 평균 RMSE: 5.562
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.518

## 변환 유형:Standard, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.826
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.803
alpha 5일 때 5 폴드 세트의 평균 RMSE: 5.717
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.637

## 변환 유형:Standard, Polynomial Degree:2
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 8.827
alpha 1일 때 5 폴드 세트의 평균 RMSE: 6.871
alpha 5일 때 5 폴드 세트의 평균 RMSE: 5.889
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.485
```

```
## 변환 유형:MinMax, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.764
alpha 1일 때 5 폴드 세트의 평균 RMSE: 5.465
alpha 5일 때 5 폴드 세트의 평균 RMSE: 5.457
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.754

## 변환 유형:MinMax, Polynomial Degree:2
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 5.298
alpha 1일 때 5 폴드 세트의 평균 RMSE: 4.323
alpha 5일 때 5 폴드 세트의 평균 RMSE: 4.907
alpha 10일 때 5 폴드 세트의 평균 RMSE: 5.185

## 변환 유형:Log, Polynomial Degree:None
alpha 0.1일 때 5 폴드 세트의 평균 RMSE: 4.770
alpha 1일 때 5 폴드 세트의 평균 RMSE: 4.676
alpha 5일 때 5 폴드 세트의 평균 RMSE: 4.729
alpha 10일 때 5 폴드 세트의 평균 RMSE: 4.836
```

(프로젝트) 선형회귀 최적 적용(6)

- Lasso, Ridge 등 최적의 alpha를 찾고
  - ✓ GridSearch, RandomSearch 가능
  - ✓ 최적의 alpha와 RMSE를 제시하시오.
  - ✓ 최적인 것을 증명할 수 있도록 그림(Matplotlib) 그려서 제출하시오.

정리 및 연습문제

- 퀴즈 및 숙제
  - ✓ 선형회귀의 장단점을 논해라 (BGD, SGD, Mini-BGD)
  - ✓ 훈련세트가 특성이 각기 다른 스케일로 구성되었다. 최적의 방법은
  - ✓ 로지스틱 회귀에서 로칼미너엄을 빠져나올 방법은?
  - ✓ 배치 경사 강하에서 에포크 마다 검증오차가 일정하게 상승한다면 문제점은?
  - ✓ 검증오차가 상승하면 미니매치 경사 하강법을 즉시 중단하는 것에 대하여
  - ✓ 다항회귀에서 검증과 훈련 오차가 사이 간격이 크다면 의미는

Thank You!

www.ust.ac.kr