ML Chap.04

선형 회귀

선형 회귀 모델의 예측

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

벡터 형태

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

모델을 훈련시킨다 => 모델이 훈련세트에 가장 잘 맞도록 모델 파라미터를 설정하는 것

선형 회귀

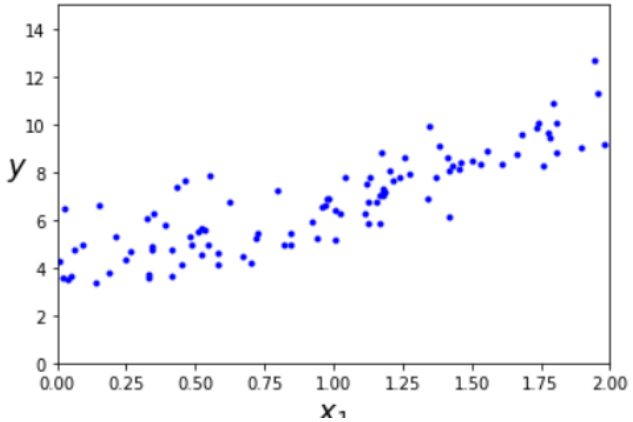
정규 방정식

$$\hat{\theta} = (X^T X)^{-1} X^T y$$

비용 함수를 최소화 하는 θ**값**을 찾는 방법

선형 회귀

정규 방정식을 사용해 $\hat{\theta}$ 계산




```
# 정규방정식을 사용해
X_b = np.c_[np.ones((100, 1)), X] # 모든 샘플에 x0 = 1을 추가
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

```
theta_best
```

```
array([[4.08875801],
       [2.97342257]])
```

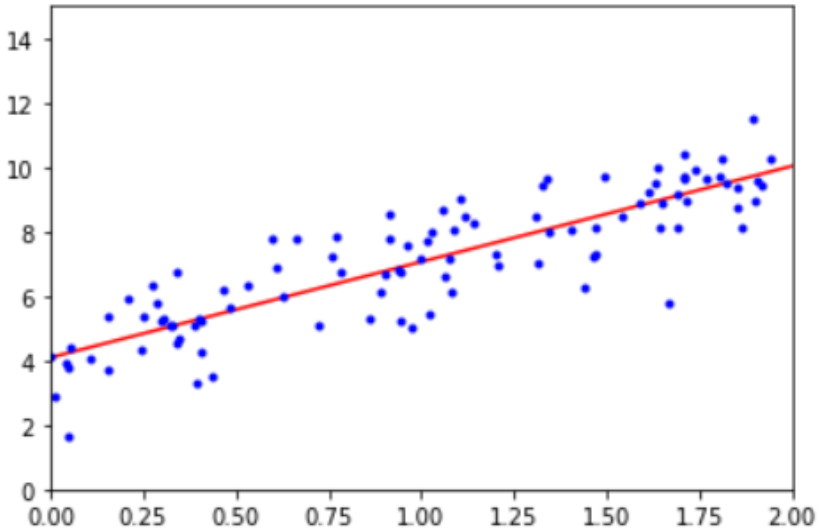
선형 회귀

$\hat{\theta}$ 사용해 예측

```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]   # 모든 샘플에 x0=1 추가
y_predict= X_new_b.dot(theta_best)
y_predict
```

```
array([[ 4.08875801],
       [10.03560314]])
```

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



선형 회귀

사이킷런에서 선형 회귀 수행

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
```

```
(array([4.08875801]), array([[2.97342257]]))
```

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

```
array([[ 4.08875801],
       [10.03560314]])
```

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

```
array([[4.08875801],
       [2.97342257]])
```

```
np.linalg.pinv(X_b).dot(y)  # np.linalg.pinv(): 유사역행렬 직접 구할 수 있음
```

```
array([[4.08875801],
       [2.97342257]])
```

선형 회귀

계산 복잡도 computational complexity

정규 방정식은 (n+1)x(n+1) 크기의 $X^T X$의 역행렬을 계산

계산 복잡도는 일반적으로 $O(n^{2.4})$ ~ $O(n^3)$

=> 특성수가 많아지면 정규방정식 처리 속도는 매우 느려 진다

Gradient Descent

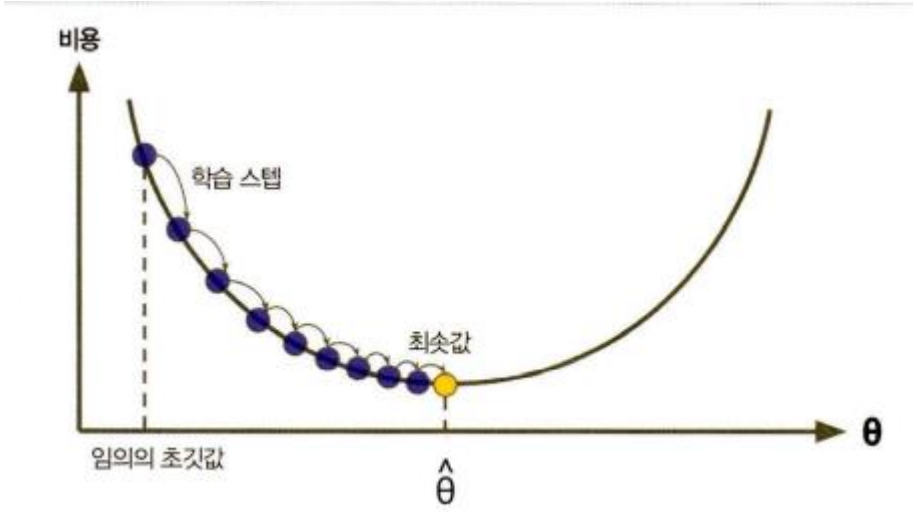
경사 하강법(GD)

- ✓ 여러 종류의 문제에서 최적의 해법을 찾을 수 있는 일반적인 최적화 알고리즘
- ✓ 비용 함수를 최소화 하기 위해 반복해서 파라미터를 조정함

1. 파라미터 벡터 θ에 대해 비용 함수의 현재 gradient 계산
2. gradient가 감소하는 방향으로 진행
3. gradient == 0 ➡ 최솟값에 도달

Gradient Descent

경사 하강법(GD)



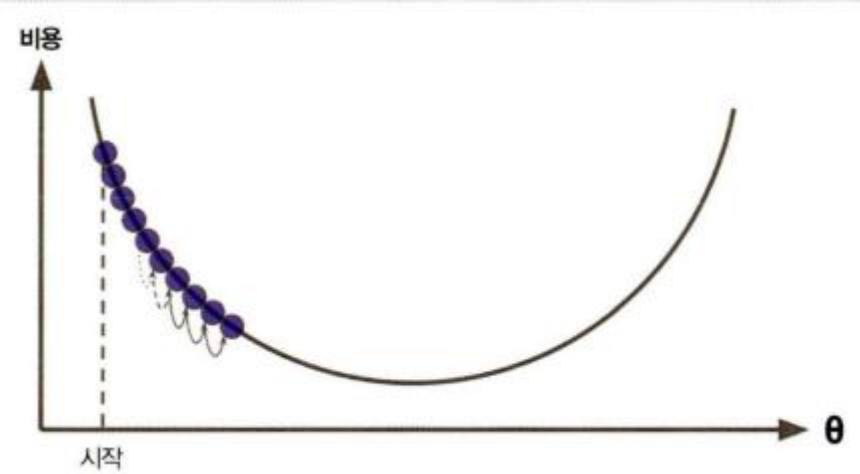
- ✓ 임의의 θ값에서 시작 => **무작위 초기화**
- ✓ 한 번에 조금씩 비용함수가 감소되는 방향으로 진행

-> 알고리즘이 최솟값에 수렴할 때까지 점진적으로 향상시킴

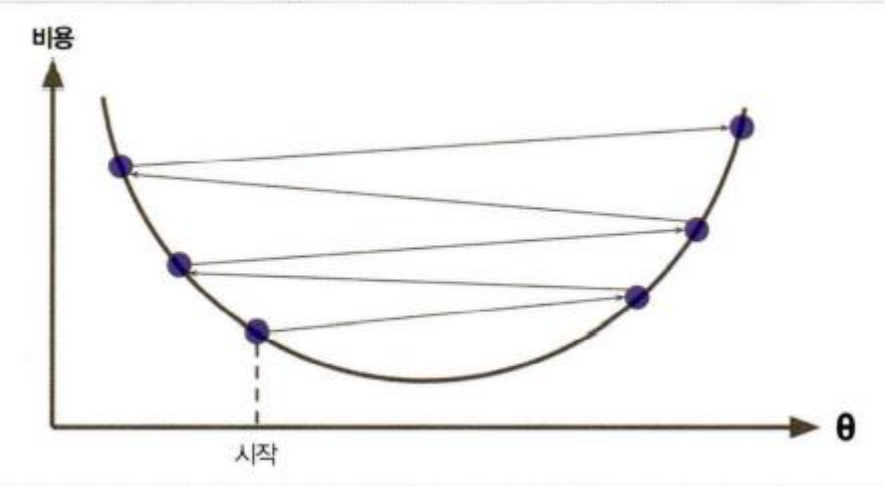
- ✓ 학습 스텝의 크기는 비용 함수의 기울기에 비례

Gradient Descent

경사 하강법(GD) -> 하이퍼파라미터: 학습률



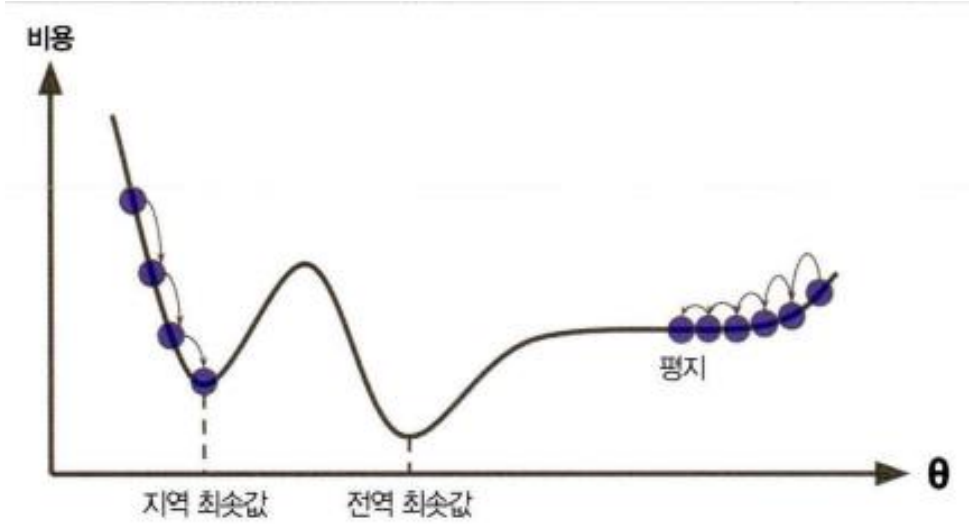
학습률이 너무 작을 때



학습률이 너무 클 때

Gradient Descent

경사 하강법(GD)의 문제점



Gradient Descent

배치 경사 하강법

✓ 매 경사 하강법 스텝에서 전체 훈련세트 X에 대해 계산하는 방법

✓ 비용 함수의 편도함수

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

✓ 비용 함수의 그레이디언트 벡터

$$\nabla_\theta \mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}\mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1}\mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n}\mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

Gradient Descent

배치 경사 하강법

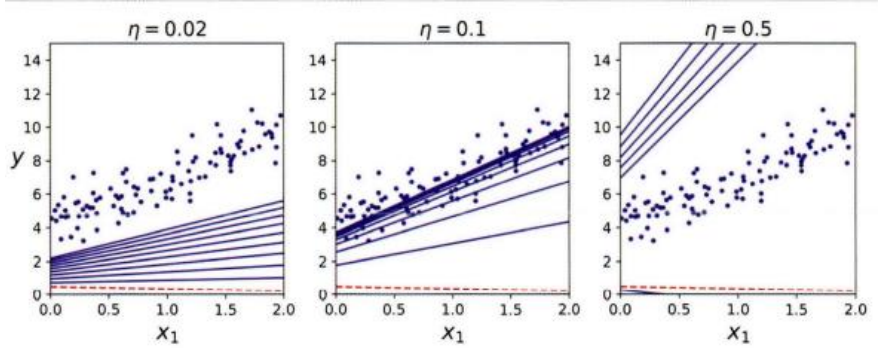
```
# 경사 하강법 알고리즘

eta = 0.1 # 학습률
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # 무작위 초기화

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

```
theta
```

```
array([[4.08875801],
       [2.97342257]])
```

✓ 경사 하강법의 스텝

$$\theta^{(next\ step)} = \theta - \eta \nabla_\theta MSE(\theta)$$



Gradient Descent

확률적 경사 하강법

✓ 매 스텝에서 한 개의 샘플을 무작위로 선택, 그 하나의 샘플에 대한 그레이언트 계산

장점

✓ 매 반복에서 다뤄야 할 데이터가 매우 적음

-> 한 번에 하나의 샘플 처리 시 빠름

✓ 반복마다 하나의 샘플만 메모리에 있으면 됨

-> 매우 큰 훈련 세트 훈련 가능

단점

✓ 확률적이므로 배치 경사 하강법보다 불안정

: 비용 함수가 최솟값에 다다를 때까지 위아래로 요동치며 평균적으로 감소

Gradient Descent

확률적 경사 하강법

- ✓ 확률적 경사 하강법의 무작위 성은 비용 함수가 불규칙 할 때 배치 경사 하강법 보다 전역 최솟값을 찾을 가능성이 더 큼
- ✓ But, 알고리즘을 전역 최솟값에 다다르지 못하게 한다는 점에서는 Bad

↓

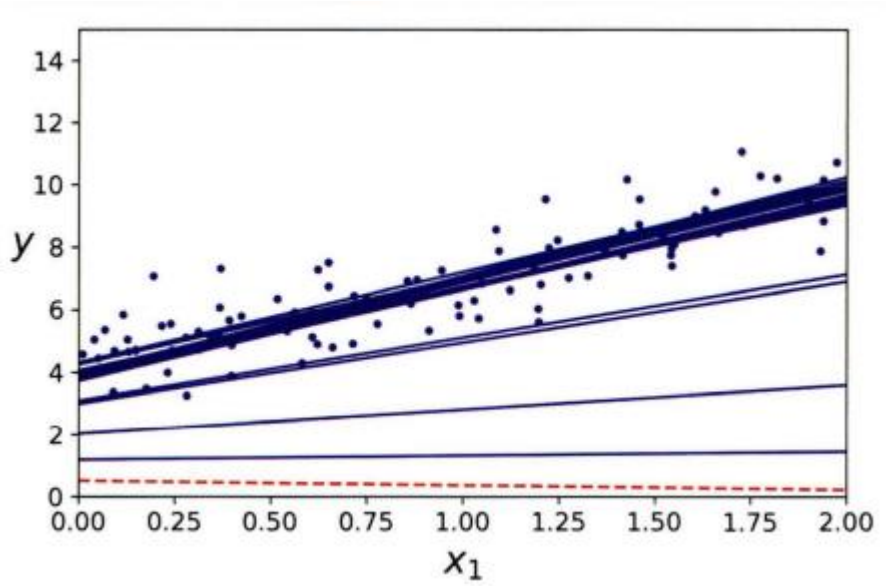
해결 방법 -> 학습률을 점진적으로 감소시킴

Gradient Descent

확률적 경사 하강법

```
n_epochs = 50
t0, t1 = 5, 50  # 학습 스케줄 하이퍼파라미터

def learning_schedule(t):
  return t0 / (t + t1)

theta = np.random.randn(2, 1) # 무작위 초기화

for epoch in range(n_epochs):
  for i in range(m):
    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
```

```
array([[4.13688072],
       [2.9198041 ]])
```



Gradient Descent

미니배치 경사 하강법

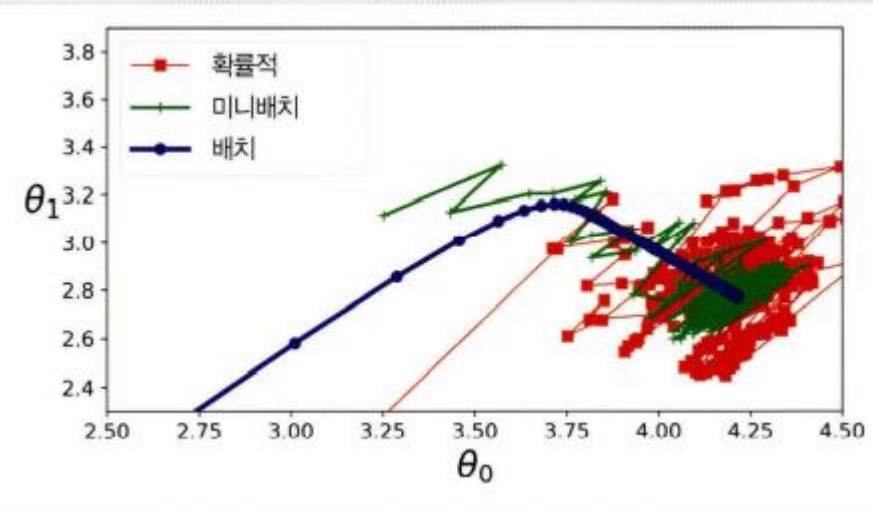
- ✓ 임의의 작은 샘플 세트(미니배치)에 대해 그레이디언트를 계산
- ✓ 미니배치를 어느 정도 크게 하면 알고리즘이 파라미터 공간에서 SGD보다 덜 불규칙하게 움직임

  -> SGD보다 최솟값에 더 가까이 도달

  -> But, 지역 최솟값에서 빠져나오기 힘듦

Gradient Descent

세가지 경사 하강법 비교



배치 경사 하강법: 경로가 최솟값에서 멈춤, but 매 스텝 많은 시간 소요

확률적 경사 하강법, 미니배치 경사 하강법: 근처에서 맴돎

-> 적절한 학습 스케줄 사용시 최솟값에 도달함

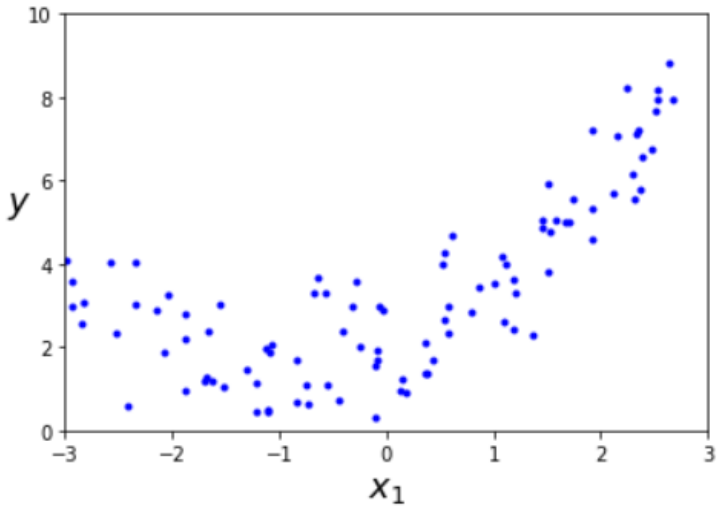
선형 회귀를 사용한 알고리즘 비교

알고리즘	m이 클 때	외부 메모리 학습 지원	n이 클 때	하이퍼파라미터 수	스케일 조정 필요	사이킷런
정규방정식	빠름	NO	느림	0	NO	N/A
SVD	빠름	NO	느림	0	NO	LinearRegression
배치 경사 하강법	느림	NO	빠름	2	YES	SGDRegresso
확률적 경사 하강법	빠름	YES	빠름	>=2	YES	SGDRegresso
미니배치 경사 하강법	빠름	YES	빠름	>=2	YES	SGDRegressor

Polynomial Regression

다항 회귀

비선형 데이터를 학습하는데 특성의 거듭제곱을 새로운 특성으로 추가하고 추가한 데이터셋에 선형 모델을 훈련시키는 기법




```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0]
```

```
array([-0.87487987])
```

```
X_poly[0]
```

```
array([-0.87487987,  0.76541479])
```

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

```
(array([3.25159674]), array([[-0.02907825,  0.02875135]]))
```

학습 곡선

Q) 얼마나 복잡한 모델을 사용할지, 이 모델이 과대적합, 과소적합 되었는지 알려면?

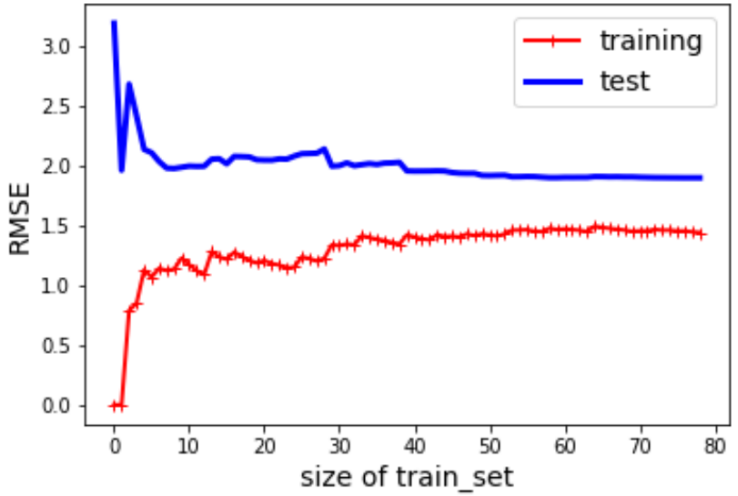
A) 교차 검증을 사용하거나 학습 곡선을 살펴봄

✓ 학습 곡선은 훈련 세트와 검증 세트의 모델 성능을 훈련 세트 크기의 함수로 나타냄

✓ 그래프 생성 방법: 훈련 세트에서 크기가 다른 서브 세트를 만들어 모델을 여러 번 훈련시킴

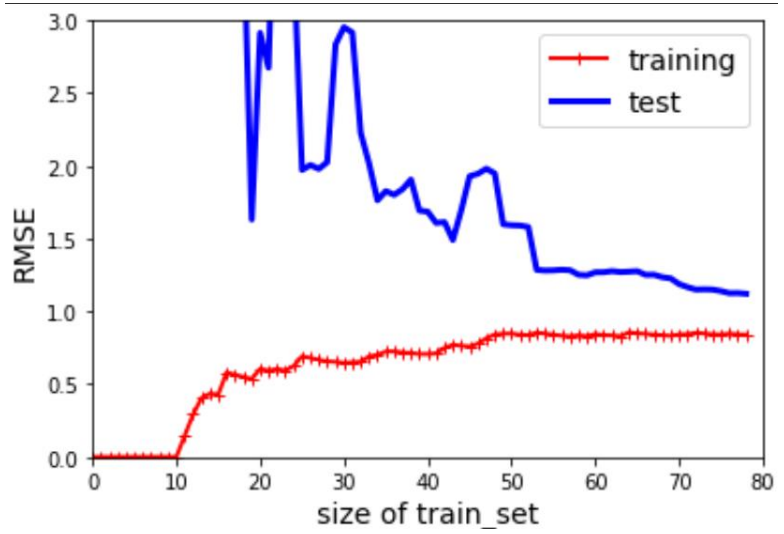
학습 곡선

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
  X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
  train_errors, val_errors = [], []
  for m in range(1, len(X_train)):
    model.fit(X_train[:m], y_train[:m])
    y_train_predict = model.predict(X_train[:m])
    y_val_predict = model.predict(X_val)
    train_errors.append(mean_squared_error(y_train[:m], y_train))
    val_errors.append(mean_squared_error(y_val, y_val_predict))
  plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="훈련 세트")
  plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="검증 세트")
```



학습 곡선

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
                                  ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
                                  ("lin_reg", LinearRegression()),
])

plot_learning_curves(polynomial_regression, X, y)
```



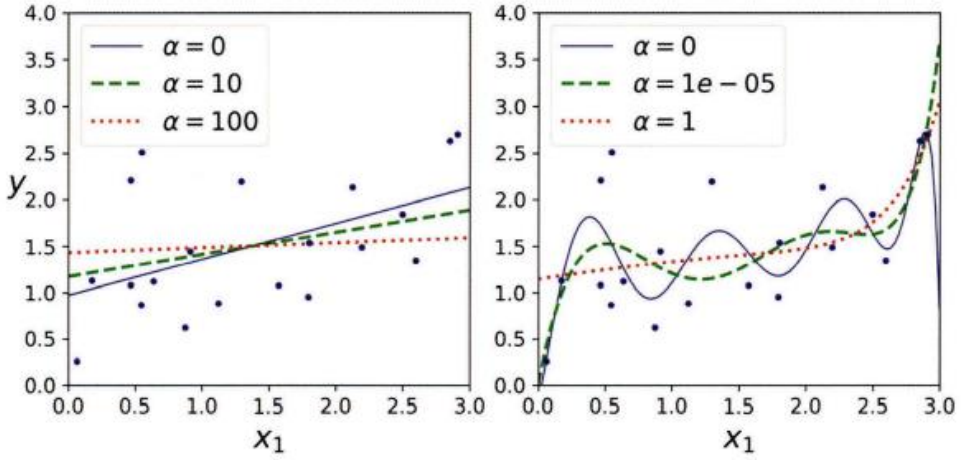
규제가 있는 선형 모델

Lidge

- ✓ 규제가 추가된 선형 회귀 버전
- ✓ 규제항은 훈련 동안에만 비용 함수에 추가됨
- ✓ 모델 훈련 종료시, 규제가 없는 성능 지표로 모델의 성능 평가
- ✓ 하이퍼파라미터 α는 모델을 얼마나 많이 규제할지 조절

- ✓ 릿지 회귀의 비용 함수:

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



규제가 있는 선형 모델

Lidge

✓ 릿지 회귀의 정규 방정식: $\hat{\theta} = (X^T X + a\,A)^{-1} X^T y$

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solver="cholesky")
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

```
array([[5.24597408]])
```

```
# 확률적 경사 하강법 사용시
sgd_reg = SGDRegressor(penalty="l2")
sgd_reg.fit(X, y.ravel())
sgd_reg.predict([[1.5]])
```

```
array([5.23291422])
```

사용할 규제를 지정하는 매개변수

규제가 있는 선형 모델

Lasso

- ✓ 선형 회귀의 또 다른 규제된 버전
- ✓ 리지와 비슷하지만 비용 함수의 규제를 $l_2\ norm$ 대신 $l_1\ norm$ 사용
- ✓ 라쏘 회귀의 비용 함수:

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

$$g(\theta, J) = \nabla_\theta MSE(\theta) + a\begin{pmatrix} sign(\theta_1) \\ sign(\theta_2) \\ \vdots \\ sign(\theta_n) \end{pmatrix}$$

$$sign(\theta_i) = \begin{cases} -1 & \theta_i < 0 \text{ 일 때} \\ 0 & \theta_i = 0 \text{ 일 때} \\ +1 & \theta_i > 0 \text{ 일 때} \end{cases}$$

규제가 있는 선형 모델

Lasso

특징

- ✓ 덜 중요한 특성의 파라미터를 완전히 제거하려고 함

  ⇒ 라쏘 회귀는 0이 아닌 가중치가 적은 희소 모델을 만든다

- ✓ 라쏘의 비용 함수는 θ_i가 0일 때 미분 가능 x

  => 그레이디언트 벡터 대신 서브그레이디언트 벡터 사용

규제가 있는 선형 모델

Lasso

✓ 서브 그레이디언트 벡터를 사용한 라쏘의 비용 함수:

$$sign(\theta_i) = \begin{cases} -1 & \theta_i < 0 \text{ 일 때} \\ 0 & \theta_i = 0 \text{ 일 때} \\ +1 & \theta_i > 0 \text{ 일 때} \end{cases}$$

$$g(\theta, J) = \nabla_\theta MSE(\theta) + a \begin{pmatrix} sign(\theta_1) \\ sign(\theta_2) \\ \vdots \\ sign(\theta_n) \end{pmatrix}$$

```
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)
lasso_reg.predict([[1.5]])
```

```
array([5.2099605])
```

규제가 있는 선형 모델

Elastic net

- ✓ 릿지 회귀와 라쏘 회귀의 절충 모델
- ✓ 규제항은 릿지와 회귀의 규제항을 단순히 더해서 사용
- ✓ 혼합 정도는 혼합 비율 r을 사용해 조절
- ✓ 엘라스틱 넷의 비용 함수:

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

- ✓ 여기서 r=1이면 라쏘 회귀, r=0이면 릿지 회귀

규제가 있는 선형 모델

Elastic net

```
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X, y)
elastic_net.predict([[1.5]])
```

```
array([5.20870017])
```

Q) 언제 사용?

A) 특성이 몇 개 뿐이라면 라쏘나 엘라스틱넷이 나음

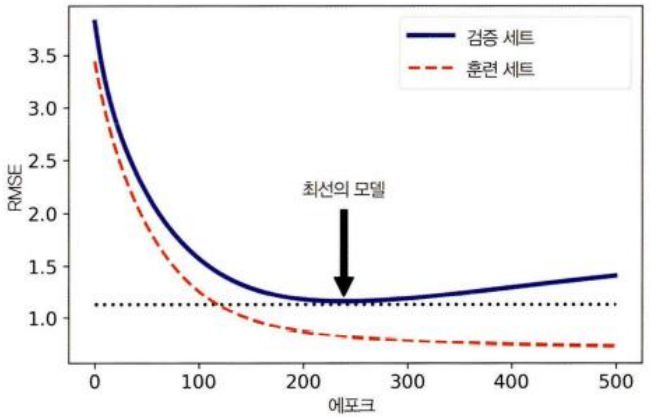
-> 불필요한 가중치를 0으로 만들어줌

A) 특성 몇 개가 강하게 연관되어 있을 땐 라쏘 < 엘라스틱넷

규제가 있는 선형 모델

Early stopping

- ✓ 검증 에러가 최솟값에 도달하면 바로 훈련 중지 시키고 이때의 파라미터를 쓰는 방식
- ✓ 확률적 경사 하강법이나 미니 배치 경사 하강법은 곡선이 들쭉날쭉해 최솟값 찾기 어려울 수 ㅇ



규제가 있는 선형 모델

Early stopping

```
from sklearn.base import clone
from sklearn.preprocessing import StandardScaler

# 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 = clone(sgd_reg)
```

Warm_start=True 로 지정 시:

fit() 메서드가 호출될 때 처음부터 다시 시작 않고, 이전 모델

파라미터에서 훈련을 이어감

Logistic Regression

- ✓ 분류에서도 사용 가능한 회귀 알고리즘
- ✓ 어떤 샘플이 특정 클래스에 속할 확률을 추정하는데 널리 사용

Logistic Regression

확률추정

- ✓ 로지스틱 회귀 모델은 입력 특성의 가중치 합을 계산

  -> But, 바로 결과 출력 x, 결과값의 logistic을 출력

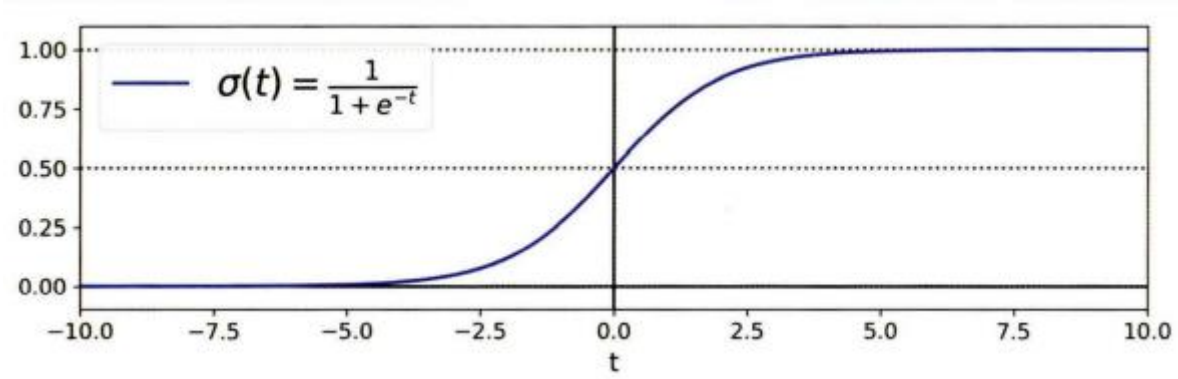
- ✓ 로지스틱은 0과 1사이의 값을 출력하는 시그모이드 함수

- ✓ 로지스틱 회귀 모델의 벡터 표현식:

$$\hat{p} = h_\theta(x) = \sigma(\theta^T x)$$

Logistic Regression

확률추정



✓ 로지스틱 함수:

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

✓ 로지스틱 회귀 모델 예측:

$$\hat{y} = \begin{cases} 0 & \hat{p} < 0.5 \\ 1 & \hat{p} \geq 0.5 \end{cases}$$

Logistic Regression

훈련과 비용함수

훈련 목적: 양성 샘플(y=1y=1)에 대해서는 높은 확률을 추정하고 음성 샘플(y=0y=0)에 대해서는 낮은 확률을 추정하는 모델의 파라미터 벡터 θθ를 찾는 것

✓ 하나의 훈련 샘플에 대한 비용 함수

$$c(\theta) = \begin{cases} -\log(\hat{p}) & if)\ y = 1 \\ -\log(1-\hat{p}) & if)\ y = 0 \end{cases}$$

Logistic Regression

훈련과 비용함수

✓ 로지스틱 회귀의 비용 함수(로그 손실)

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log\left(\hat{p}^{(i)}\right) + (1-y^{(i)})\log\left(1-\hat{p}^{(i)}\right))]$$

✓ 로지스틱 비용 함수의 편도 함수

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\sigma(\theta^T x^{(i)}) - y^{(i)})x_j^{(i)}$$

Logistic Regression

결정 경계

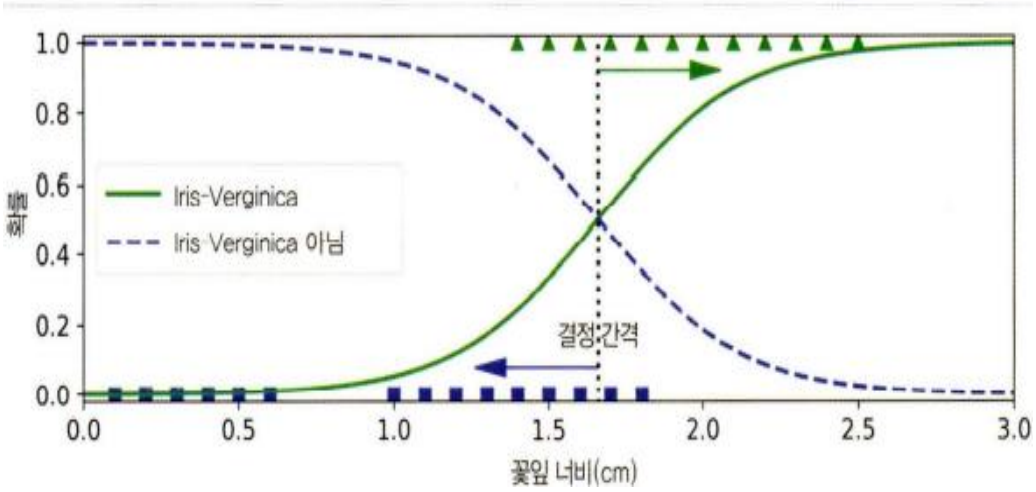
```
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
```

```
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
```

```
X = iris["data"][:, 3:] # 꽃잎의 너비
y = (iris["target"] == 2).astype(np.int)  # Iris-Virginica면 1, 그렇지 않으면 0
```

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```



Logistic Regression

소프트맥스 회귀

✓ 로지스틱 회귀는 이진 분류만 할 수 있는 모델이 아니라 직접 다중 클래스를 분류할 수 있음

-> 소프트맥스 회귀 또는 다항 로지스틱 회귀라고 함

✓ 개념: 샘플 x가 주어지면 먼저 소프트맥스 회귀 모델이 각 클래스 k에 대한 점수 $s_k(x)$를 계산,

그 점수에 소프트맥스 함수(또는 정규화된 지수 함수)를 적용하여 각 클래스의 확률을 추정

✓ 클래스 k에 대한 소프트맥스 점수:

$$s_k(x) = \theta^{(k)})^T \cdot x$$

Logistic Regression

소프트맥스 회귀

- ✓ 각 클래스는 자신만의 파라미터 벡터 $\theta^{(k)}$ 존재

  -> 이 벡터들은 파라미터 행렬 Θ에 행으로 저장됨

- ✓ 샘플 x에 대해 각 클래스의 점수가 계산되면 소프트맥스 함수를 통과시켜 클래스 k에 속할 확률 $\hat{p}_k$ 추정 가능
- ✓ 소프트맥스 함수:

$$\hat{p}_k = \sigma(s(x))_k = \frac{\exp(s_k(x))}{\sum_{j=1}^{K} \exp(s_j(x))}$$

Logistic Regression

소프트맥스 회귀

- ✓ 로지스틱 회귀 분류기와 마찬가지로 소프트맥스 회귀 분류기는 추정 확률이 가장 높은 클래스 선택
- ✓ 소프트맥스 회귀 분류기의 예측:

$$\hat{y} = argmax_k \sigma(s(x))_k = argmax_k((\theta^{(k)})^T \cdot x$$

Logistic Regression

소프트맥스 회귀

- ✓ 모델이 타깃 클래스에 대해서는 높은 확률, 다른 클래스에 대해서는 낮은 확률을 추정하도록 만드는 것이 훈련의 목적
- ✓ 크로스 엔트로피 비용 함수를 최소 화하는 것 -> 타깃 클래스에 대해 낮은 확률을 예측하는 모델을 억제함
- ✓ 크로스 엔트로피 -> 추정된 클래스의 확률이 타깃 클래스에 얼마나 잘 맞는지 측정하는 용도로 종종 사용
- ✓ 크로스 엔트로피 비용 함수:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y(i)_k \log(\hat{p}_k{}^{(i)})$$

- ✓ 클래스 k에 대한 크로스 엔트로피의 그레이디언트 벡터:

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}(\hat{p}_k{}^{(i)} - \hat{y}_k{}^{(i)})x^{(i)}$$

Thank you