Chapter4. 모델 훈련

4.1 선형 회귀

일반적으로 선형 모델은 식 4.1 처럼 입력 특성의 가중치 합과 편향이라는 상수를 더해 예측을 생성

- **식 4-1.** 선형 회귀 모델의 예측
  - $\hat{y} = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$
    - $\hat{y}$: *예측값*
    - n : *특성의 수*
    - x_i : *i 번째 특성값*
    - θ_j : *j 번째 모델 파라미터* $(\theta_0$: *편향*, $\theta_1, \theta_2, \ldots, \theta_n$: *특성의 가중치*$)$

- **식 4-2.** 선형 회귀 모델의 예측(벡터 형태)
  - $\hat{y} = h_\theta(x) = \theta \cdot x$
    - θ는 편향과 특성 가중치를 담은 모델의 파라미터 벡터
    - x*은* x_0 *에서* x_n *까지 담은 특성 벡터* $(x_0$ *은 항상* $1)$
    - $\theta \cdot x$ *은 점곱 이는* $\theta_0 x_0 + \cdots + \theta_n x_n$ *의미*

회귀에서 가장 널리 사용되는 성능 측정 지표는 평균 제곱근 오차(RMSE) 임.

그러므로 선형 회귀 모델을 훈련시키려면 RMSE 를 최소화하는 θ를 찾아야함.

실제로는 RMSE 보다 평균제곱 오차인 MSE 를 최소화하는 것이 같은 결과를 내면서 더 간단함.

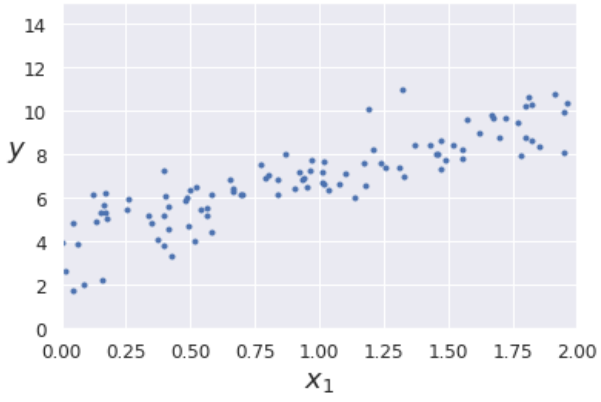
따라서 훈련 세트 X 에 대한 선형 회귀 가설 h_0의 MSE 는 식 4-3 처럼 계산됨.

- **식 4-3.** 선형 회귀 모델의 MSE 비용 함수
  - $MSE(x, h_0) = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^T x^{(i)} - y^{(i)}\right)^2$

4.1.1 정규방정식

- 비용함수를 최소화하는 θ값을 찾기 위한 해석적인 방법은수학 공식의 정규 방정식임.
  - **식 4-4.** 정규 방정식
    - $\hat{\theta} = (x^T x)^{-1} x^T y$
      - $\hat{\theta}$은 비용함수를 최소화하는 θ 값
      - y는 $y^{(1)}$ 부터 $y^{(m)}$ 까지 포함하는 타깃 벡터
- 식 4-4 의 정규방정식 공식을 테스트 하기 위해서 **선형처럼 보이는 데이터를 생성**

```
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])
plt.show()
```



- 정규 방정식을 사용해 $\hat{\theta}$ 를 계산

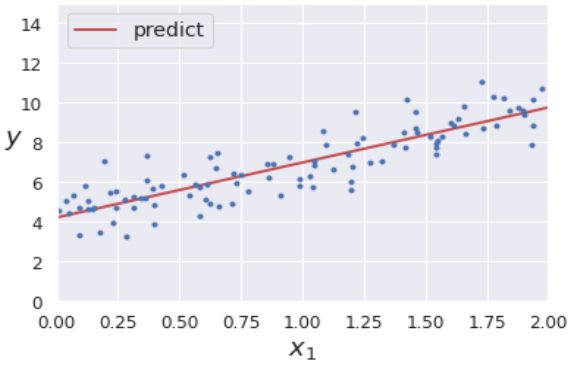
  선형대수 모듈 np.linalg 에 있는 inv 함수를 사용해 역행렬을 계산하고 dot 메서드를 사용해 행렬 곱셈을 진행

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

theta_best
array([[4.0550023 ],
       [2.98482572]])
```

  위의 선형처럼 보이는 데이터의 함수는 $y = 4 + 3x_1 +$ 가우시안 잡음 이지만 정규 방정식으로 계산한 값은 $\theta_0 = 4.05, \theta_1 = 2.98$로 매우 비슷하지만 잡음 때문에 원래 함수의 파라미터를 정확하게 재현하지는 못함

- $\hat{\theta}$를 사용해 예측을 진행

```
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([[ 4.0550023 ],
       [10.02465373]])

plt.plot(X_new, y_predict, "r-", linewidth=2, label="예측")
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([0, 2, 0, 15])
plt.show()
```



- 사이킷런에서 선형 회귀를 수행하는 것은 간단함

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y)

lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))


lin_reg.predict(X_new)

array([[4.21509616],
       [9.75532293]])
```

- LinearRegression 클래스는 scipy.linalg,lstsq 함수를 기반

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

theta_best_svd

array([[4.21509616],
       [2.77011339]])
```

위의 함수는 $\hat{\theta} = x^{+}y$를 계산 여기서 x^{+}는 x 의 유사역행렬임

- np.linalg.pinv 함수를 사용해 유사역행렬을 직접 구할 수 있음

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

array([[4.21509616],
       [2.77011339]])
```

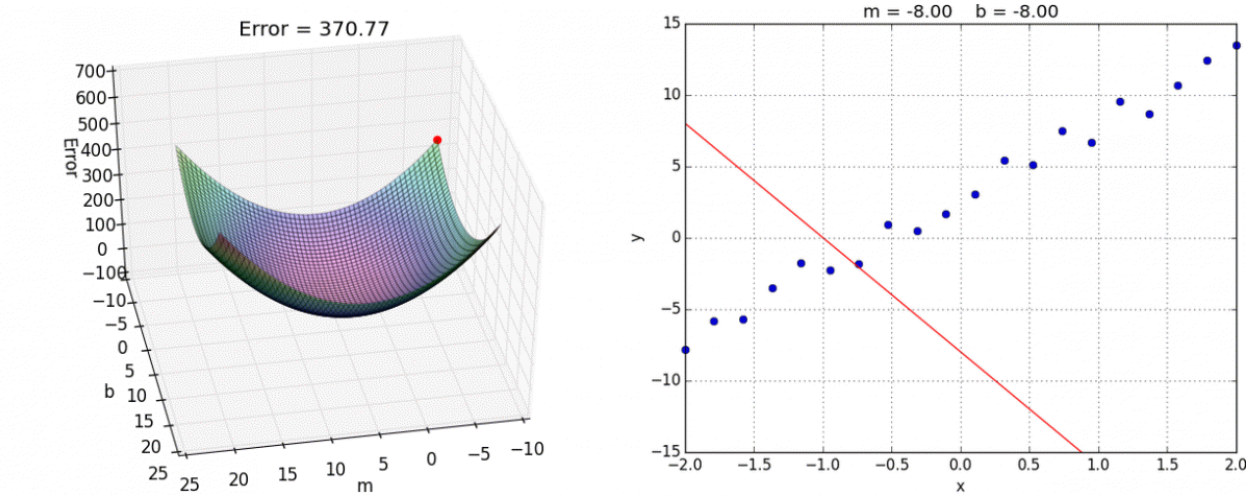
4.1.2 계산 복잡도

- 역행렬을 계산하는 계산 복잡도는 일반적으로$O(n^{2.4})$에서 $O(n^{3})$사이
- 따라서 특성 수가 2 배 늘어나면 계산 시간이 대략 $2^{2.4} = 5.3$에서 $2^{3} = 8$배로 증가

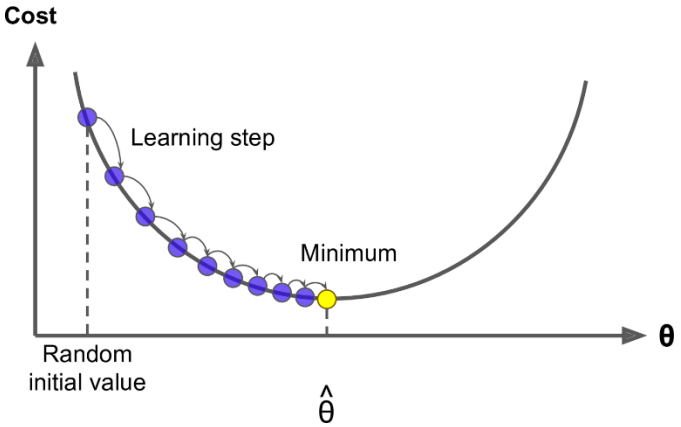
- 정규방정식과 SVD 방법 모두 특성 수, 샘플 수가 많아지면 매우 느려짐
- 예측 계산 복잡도는 특성 수와 샘플 수에 선형적임

4.2 경사 하강법

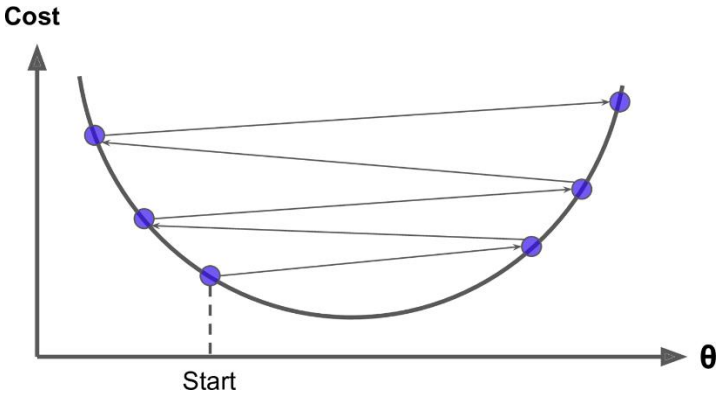
- 경사하강법은 비용 함수를 최소화하기 위해 반복해서 파라미터를 조정해가는 것



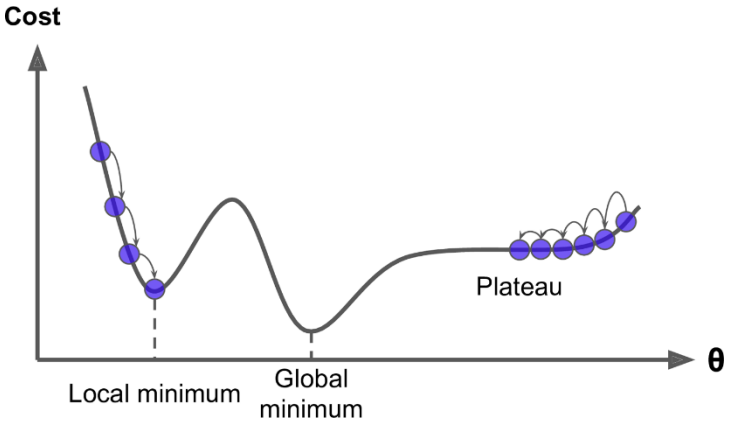
- 파라미터 벡터 θ에 대해서 비용 함수의 현재 gradient 를 계산하고 gradient 가 감소하는 방향으로 진행하다가 gradient 가 0 이 되면 최솟값에 도달한 것
- 즉, θ를 임의의 값으로 시작해서(무작위 초기화) 한번에 조금씩 비용 함수(ex. MSE)를 감소되는 방향으로 진행하여 알고리즘이 최솟값에 수렴할 때까지 점진적으로 향상시키는 것
- 경사 하강법에서 중요한 파라미터는 스텝의 크기로, 학습률 (learning rate)하이퍼파라미터로 결정됨.



- 학습률이 너무 작으면 알고리즘이 수렴하기 위해 반복을 많이 진행해야 하므로 시간이 오래 걸림



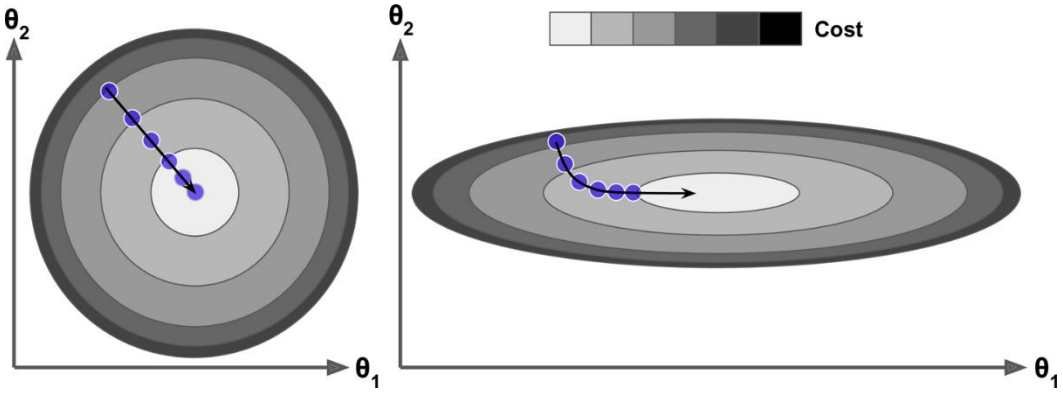
- 학습률이 너무 크면 알고리즘이 더 큰 값으로 발산하게 만들어 적절한 해법을 찾지 못함



- 비용함수에 특이한 지형이 있다면 최솟값으로 수렴하기 매우 어려움
- 위의 그림에서 보면 알고리즘이 왼쪽에서 시작하면 전역 최솟값(Global minimum)에 도달하지 못하고 전역 최솟값보다 덜 좋은 지역 최솟값(local minimum)에 수렴
- 알고리즘이 오른쪽에서 시작하면 시간이 오래 걸리고 일찍 멈추개 되어 전역 최솟값에 도달하지 못함

- 선형 선형회귀를 위한 MSE 비용함수는 볼록함수(convex function)형태여서 하나의 전역최솟값이 존재하고, 연속된 함수이며 기울기가 갑자기 변하지 않기 때문에 경사하강법으로 전역최솟값에 접근이 가능함

- 특성들의 스케일이 다르면 문제가 발생



- 왼쪽 그림은 특성 스케일이 같은 훈련 데이터셋, 오른쪽 그림은 특성 1 이 특성 2 보다 스케일이 작은 훈련데이터셋임
- 이때 오른쪽 그림처럼 특성 1 의 스케일이 훨씬 작기 때문에 비용함수를 변화시키기 위해서는 θ_1이 더 크게 바뀌어야 하므로 좌우로 긴 형태가 되고 이 경우 최솟값에 도달은 하겠지만 시간이 매우 오래걸리는 문제가 발생
- 따라서 경사 하강법을 할때는 모든 변수들이 같은 스케일을 갖도록 스케일링을 해주어야함

4.2.1 배치 경사 하강법

- 경사하강법을 구현하려면 각 모델 파라미터 θ_j에 대해 비용함수의 gradient 를 계산해야함 즉, θ_j가 조금 변경될 때 비용 함수가 얼마나 바뀌는지 계산해야하는데 이를 편도함수라고 함.

  - **식 4-5.** 비용함수의 편도함수

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

- 편도함수를 각각 계산하는 대신 식 4-6 을 사용하여 한꺼번에 계산이 가능함.

  - **식 4-6.** 비용함수의 gradient 벡터

    - $\nabla_\theta MSE(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial}{\partial\theta_0}MSE(\boldsymbol{\theta}) \\ \cdots \\ \frac{\partial}{\partial\theta n}MSE(\boldsymbol{\theta}) \end{bmatrix} = \frac{2}{m}\boldsymbol{x}^T(\boldsymbol{x\theta} - \boldsymbol{y})$

- 이 공식은 매 경사 하강법 스텝에서 전체 훈련 세트 X 에 대해 계산함 따라서 이 알고리즘을 배치 경사 하강법이라고 함.

- 위로 향하는 gradient 벡터가 구해지면 반대 방향인 아래로 가야함
- 따라서 θ에서 $\nabla_\theta MSE(\theta)$를 빼야함 여기서 스텝의 크기를 결정하기 위해서 그레디언트 벡터에 학습률 η를 곱함

➢ **식 4-7.** 경사 하강법의 스텝

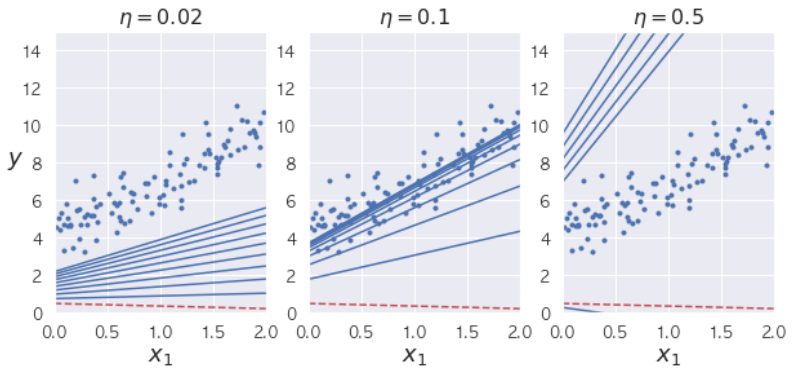
- $\boldsymbol{\theta^{(next\ step)} = \theta - \eta \nabla_\theta MSE(\theta)}$

- 위 수식의 알고리즘을 구현

```
eta = 0.1  # learning rate
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([[4.21509616],
       [2.77011339]])
```

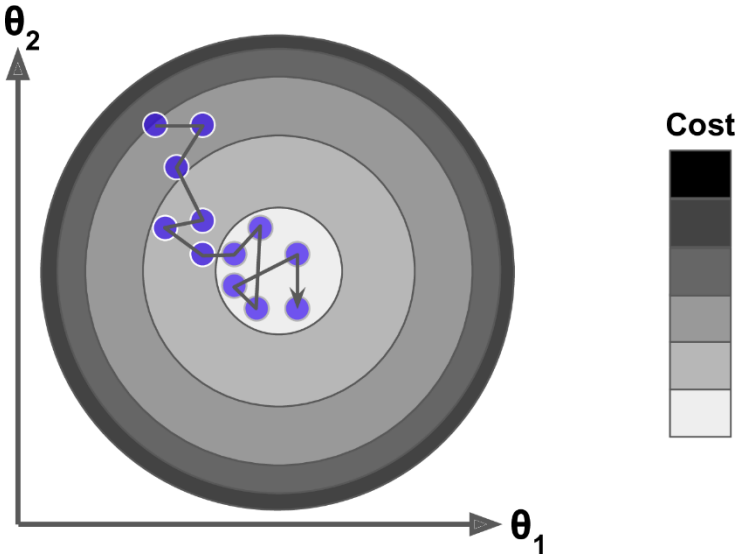
- 학습률을 바꿔보면 ?



- 적절한 학습률을 찾으려면 그리드 탐색을 사용해야함

4.2.2 확률적 경사 하강법

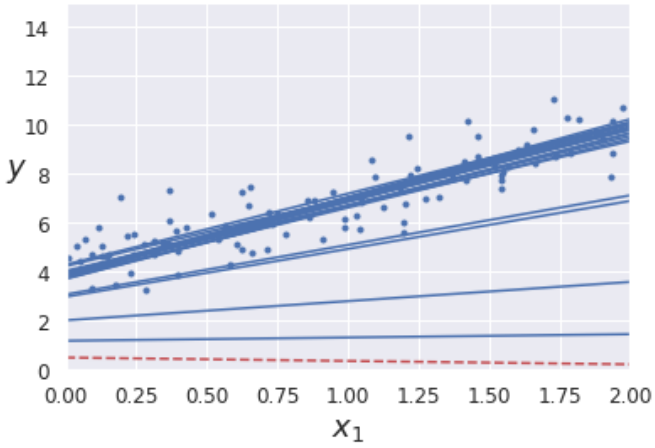
- 배치 경사 하강법의 가장 큰 문제점은 매 스텝에서 전체 훈련 세트를 사용해 그레이디언트를 계산한다는 것 따라서 훈련 세트가 커지면 매우 느려짐
- 이와는 정반대로 **확률적 경사 하강법은 매 스텝에서 한 개의 샘플을 무작위로 선택하고 그 하나의 샘플에 대한 그레디언트를 계산함**
- 매 반복에서 다뤄야 할 데이터가 매우 적기 때문에 한번에 하나의 샘플을 처리하면 알고리즘이 훨씬 빨라짐
- 하지만 배치 경사 하강법 보다는 훨씬 불안정함
- 또한 무작위성은 지역 최솟값에서 탈출시켜줘서 좋지만 알고리즘이 전역 최솟값에 다다르지 못하게 한다는 점에서는 좋지 않음
- 이 딜레마를 해결하는 방법은 학습률을 점진적으로 감소시키는 것
- 즉, 학습률을 크게 설정하고(지역 최솟값을 뛰어넘고 수렴하도록), 점차 작게 줄여서 전역 최솟값에 도달하게 하는 것이 좋음



- 확률적 경사하강법의 구현

```
theta_path_sgd = []
m = len(X_b)
np.random.seed(42)

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 초기값

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)
        learning_rate = learning_schedule(epoch * m + i)
        theta = theta - learning_rate * gradients
        theta_path_sgd.append(theta)


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



확률 경사하강법의 첫 20 개 스텝 : 스텝이 불규칙하게 진행됨

- 일반적으로 한 반복에서 m 번 반복되고, 각 반복을 에포크라고 함

- 앞에서 배치경사하강법에서 1000 번 반복하는 동안 확률적 경사하강법에서는 50 번만 반복하고도 매우 좋은 값에 도달함

- 사이킷런에서 SGD 방식으로 선형 회귀를 사용하려면 기본값으로 제곱 오차 비용 함수를 최적화 하는 SGD Regressor 클래스를 사용함
- 다음 코드는 최대 1,000 번 에포크(**max_iter=1000**) 동안 실행되고, 한 에포크에서 0.001 보다 적게 손실이 줄어들 때까지 실행됨(**tol = 1e-3**)
- 학습률 0.1(**eta0=0.1**)로 기본 학습 스케줄을 사용하고 규제는 사용하지 않음(**penalty=None**)

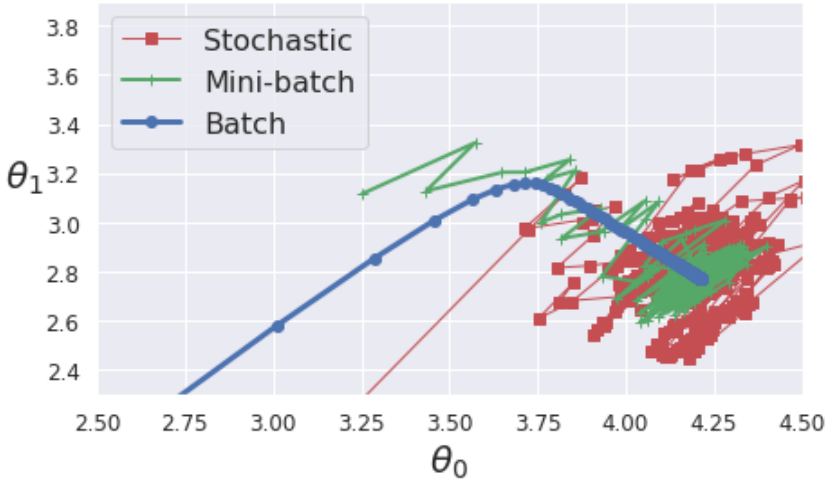
```
from sklearn.linear_model import SGDRegressor
sgd_reg=SGDRegressor(max_iter=1000,tol = 1e-
3,penalty=None,eta0=0.1,random_state=42)

sgd_reg.fit(X, y.ravel())

SGDRegressor(alpha=0.0001, average=False, early_stopping=False,
epsilon=0.1,
             eta0=0.1, fit_intercept=True, l1_ratio=0.15,
             learning_rate='invscaling', loss='squared_loss',
max_iter=1000,
             n_iter_no_change=5, penalty=None, power_t=0.25,
random_state=42,
             shuffle=True, tol=0.001, validation_fraction=0.1, verbose=0,
             warm_start=False

sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

- 정규 방정식에서 구한 것고 매우 비슷한 값을 얻음

4.2.3 미니배치 경사 하강법

- 각 스텝에서 전체 훈련 세트나 하나의 샘플을 기반으로 그레이디언트를 계산하는 것이 아니라 미니배치라 부르는 임의의 작은 샘플 세트에 대해 그레이디언트를 계산함

- 확률적 경사 하강법에 비해 미니배치 경사 하강법의 주요 장점은 행렬 연산에 최적화된 하드웨어, 특히 GPU 를 사용해서 얻은 성능 향상임
- 특히, 미니배치를 어느 정도 크게 하면 이 알고리즘은 파라미터 공간에서 SGD(확률적 경사 하강법)보다 덜 불규칙적으로 움직임.



- 배치경사 하강법은 최솟값에서 멈춘 반면, SGD 와 미니배치 경사 하강법은 맴돌고 있음을 볼 수 있음
- 하지만, 앞에서 언급한 바와 같이 적절한 학습 스케줄(learning schedule)을 사용하면 최솟값 도달이 가능함
- 특히, 미니배치를 어느 정도 크게 하면 이 알고리즘은 파라미터 공간에서 SGD(확률적 경사 하강법)보다 덜 불규칙적으로 움직임.

- 표 4-1. 선형 회귀를 사용한 알고리즘 비교

알고리즘	m 이 클 때	외부 메모리 학습 지원	n 이 클 때	하이퍼 파라미터 수	스케일 조정 필요	사이킷런
정규방정식	빠름	X	느림	0	X	N/A
SVD	빠름	X	느림	0	X	LinearRegression
배치경사하강법	느림	X	빠름	2	O	SGDRegressor

확률적 경사 하강법	빠름	0	빠름	>=2	0	SGDRegressor
미니배치 경사 하강법	빠름	0	빠름	>=2	0	SGDRegressor

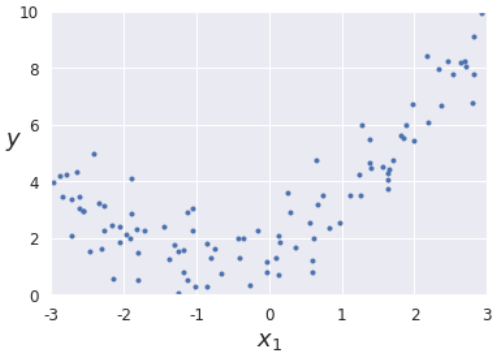
- 위의 알고리즘들은 훈련 결과에 거의 차이가 없음. 모두 매우 비슷한 모델을 만들고 정확히 같은 방식으로 예측을 함.

4.3 다항 회귀

- 비선형 데이터를 학습하는 데 선형 모델을 사용할 수 있음.
- 간단한 방법은 각 특성의 거듭제곱을 새로운 특성으로 추가하고, 이 확장된 특성을 포함한 데이터셋에 선형 모델을 훈련시키는 것
- 이런 기법을 다항 회귀라고 함

- 간단한 2 차방정식으로 비선형 데이터를 생성 (잡음이 포함된 비선형 데이터셋)

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + 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])
plt.show()
```



- 직선은 위의 데이터와 잘 맞지 않을 것 그러므로 사이킷런의 PolynomialFeatures 를 사용해 훈련 데이터를 반환
- 훈련 세트에 있는 각 특성을 제곱하여 새로운 특성으로 추가

```
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_poly[0] :', X_poly[0])
```

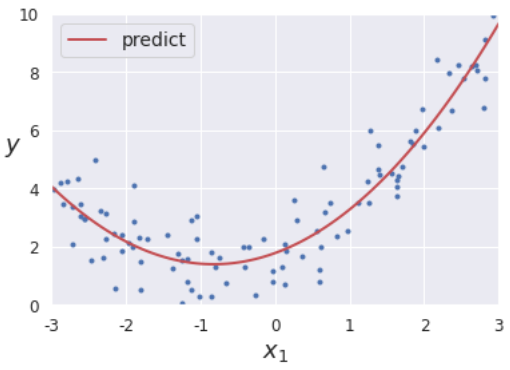
```
X[0] : [-0.75275929]
X_poly[0] : [-0.75275929  0.56664654]
```

- X_ploy 는 이제 원래 특성 X 와 이 특성의 제곱을 포함한 것
- 이 훈련 데이터에 Linear Regression 을 적용

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

```
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

```
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="predict")
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])
plt.show()
```



- 실제 원래 함수 : $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \textit{가우시안 잡음}$
- 예측된 모델 : $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$
- PolynomialFeatures 가 주어진 차수까지 특성 간의 모든 교차항을 추가함
- 예를 들면 두 개의 특성 a,b 가 있을 때 degree = 3 으로 PolynomialFeatures 를 적용하면 a^2, a^3, b^2, b^3뿐만 아니라 ab, a^2b, ab^2도 특성으로 추가함