머신러닝 스터디

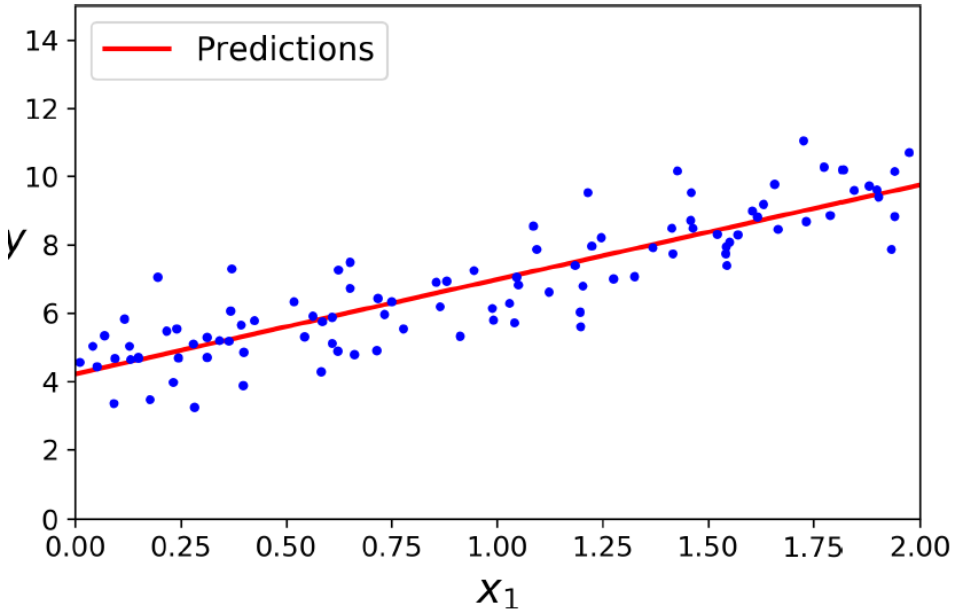
화학소재솔루션센터 학생연구원 한요셉
2022.04.27

Index

4.1. 선형 회귀

1. 선형 회귀

- 회귀 : 여러 개의 독립변수와 한 개의 종속변수간의 상관관계를 모델링 하는 기법
- 선형 회귀 : 데이터의 선형성을 가정하고 선형회귀식의 회귀계수(파라미터)를 추정하는 것



Equation 4-1. Linear Regression model prediction

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

Equation 4-2. Linear Regression model prediction (vectorized form)

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

- 최적의 회귀 계수를 추정하는 것! => " 비용함수가 최소화 될 때 회귀계수 "

4.1. 선형 회귀

2. 비용함수(Cost Function)

- 잔차(Residual) : 모델 값 – 실제값
- RSS(Residual Sum of Squares) : $\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$

- 비용함수 : **RSS / m = MSE(Mean Squared Error)** => MSE(비용함수)가 최소화 될 때의 파라미터를 추정하는 과정

Equation 4-3. MSE cost function for a Linear Regression model

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

최소화 방법은 정규방정식과 경사하강법을 이용함

※ 회귀에서 가장 널리 사용되는 성능은 RMSE를 최소화는 MSE를 최소화 하는 것과 같은 결과 이며 더 쉬운 문제임.

4.1. 선형 회귀

3. 정규방정식

※ 선형대수학 : 최소제곱(자승)법, 특잇값 분해 참고

- 정규방정식 : 비용함수가 최소 일 때의 회귀계수를 찾기 위한 해석적인 방법 => 역행렬 문제를 푼다.

Equation 4-4. Normal Equation

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

- 사이킷런 라이브러리를 이용해서 쉽게 구현 가능 : LinearRegression 클래스를 이용하여 구현

※ 특잇값 분해(singular value decomposition, SVD)이용하여 계산하며, 유사역행렬을 구하는 문제로 해결함

※ 데이터수 보다 특성수가 많거나 특성이 중복되어 특이행렬(역행렬이 없는 행렬)에서 유사역행렬을 항상 구할 수 있음

4.1. 선형 회귀

4. 계산 복잡도(computational complexity)

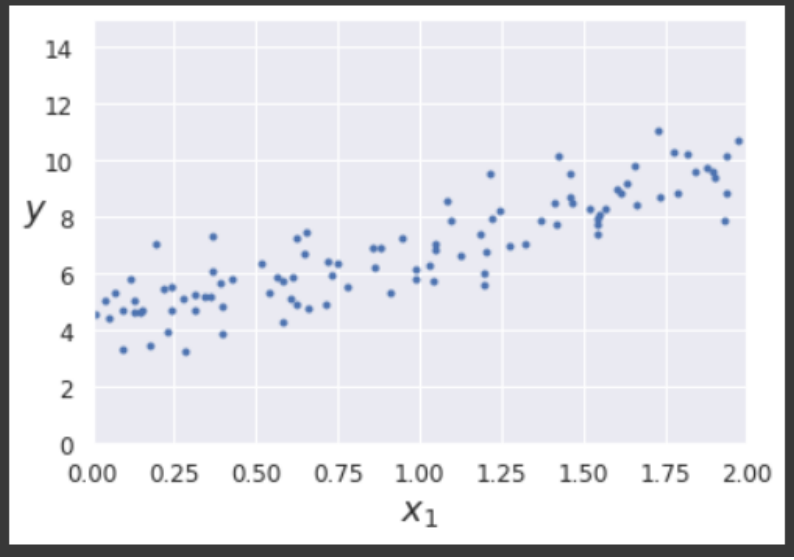
- 정규 방정식은 (n+1) *(n+1) 크기가 되는 X^TX의 역행렬 계산 하며 계산 복잡도는 $O(n^{2.4})$ to $O(n^3)$,
- 즉 특성 수가 2배로 늘어 나면 계산 시간은 $2^{2.4}$ =5.3에서 2^3 =8배로 증가
- 사이킷런의 LinearRegression 클래스가 사용하는 SVD 방법은 $O(n^2)$ 으로, 약 특성 수가 2배면 계산시간은 4배로 증가

4.1. 선형 회귀

1. 선형 회귀 예제

```
[12] # 임의의 데이터 생성

     # 독립변수를 0~1의 균일분포에서 (100,1) 행렬로 난수 생성
     X = 2 * np.random.rand(100, 1)

     # 종속 변수를 생성, 오차항을 정규분포에서 (100,1)행렬 난수 생성
     y = 4 + 3 * X + np.random.randn(100, 1)
```

```
[13] 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();
```



4.1. 선형 회귀

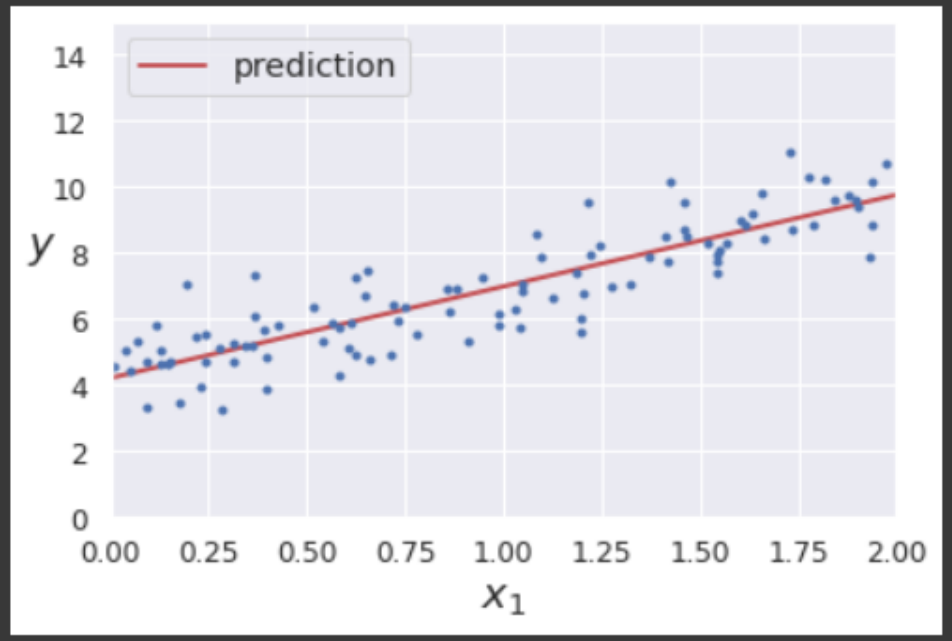
1. 선형 회귀 예제

```
[14] # 정규방정식 계산

     # 모든 샘플에 x0 = 1 추가(편향 추가)
     X_b = np.c_[np.ones((100, 1)), X]

     # 정규방정식 공식적용, 넘파이 선형 대수 모듈에 있는 inv() 함수로 역행렬 계산, dot()메서드로 행렬 곱셈
     theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
     theta_best

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

[15] # 추정한 파라미터로 x가 0과 2일때의 y값 예측
     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([[4.21509616],
            [9.75532293]])
```

4.1. 선형 회귀

1. 선형 회귀 예제

```
plt.plot(X_new, y_predict, "r-", linewidth=2, label="prediction")
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();
```



4.1. 선형 회귀

1. 선형 회귀 예제

```
[18] # 사이킷런에서 선형회귀 모델 불러오기
     from sklearn.linear_model import LinearRegression

     # 선형회귀 모델 적합 시킴
     lin_reg = LinearRegression()
     lin_reg.fit(X, y)
```

```
     LinearRegression()
```

```
[30] # 편향(intercept)과 가중치(coef)

     lin_reg.intercept_, lin_reg.coef_
```

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

```
[26] # X가 0과 2일때 예측
     lin_reg.predict(X_new)
```

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

4.1. 선형 회귀

1. 선형 회귀 예제

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

theta_best_svd
```

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

```
[39] # 유사역행렬(특잇값 분해)로 계산
np.linalg.pinv(X_b).dot(y)
```

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

4.2. 경사 하강법(gradient descent, GD)

1. 경사 하강법

- 비용함수를 최소화 하기 위해 학습률과 손실함수의 순간기울기(gradient)를 이용하여 파라미터를 업데이트 하는 방법
- 함수의 기울기(=gradient)를 이용해서 함수의 최소값일 때의 x값을 찾기 위한 방법
- 파라미터 θ에 대해 gradient 계산하고 gradient 가 감소하는 방향으로 학습률을 함께 곱해서 업데이트.

Equation 4-5. Partial derivatives of the cost function

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

Equation 4-6. Gradient vector of the cost function

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

Equation 4-7. Gradient Descent step

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

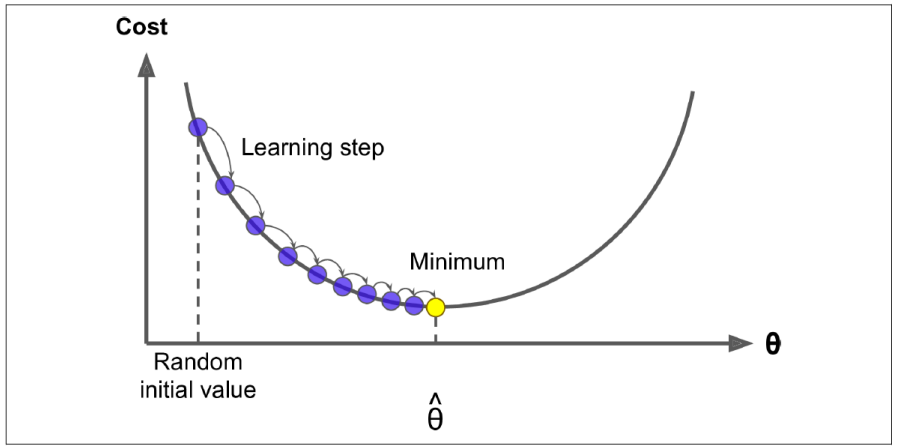


Figure 4-3. Gradient Descent

4.2. 경사 하강법(gradient descent, GD)

① 학습률 고려

- 경사 하강법 중요한 하이퍼파라미터는 스텝의 크기, 학습률(learning rate)이다.
- 학습률이 너무 작으면 알고리즘이 수렴하기 위해 반복을 많이 진행해야 하므로 시간이 오래 걸림
- 학습률이 너무 크면 골짜기를 가로 질러 반대편으로 건너 뛰게 되어 이전보다 더 높은 곳으로 갈 수도 있음
- 선형회귀를 위한 MSE 비용함수는 볼록함수라 최솟값이 하나임 (하나의 전역 최솟값을 가짐)

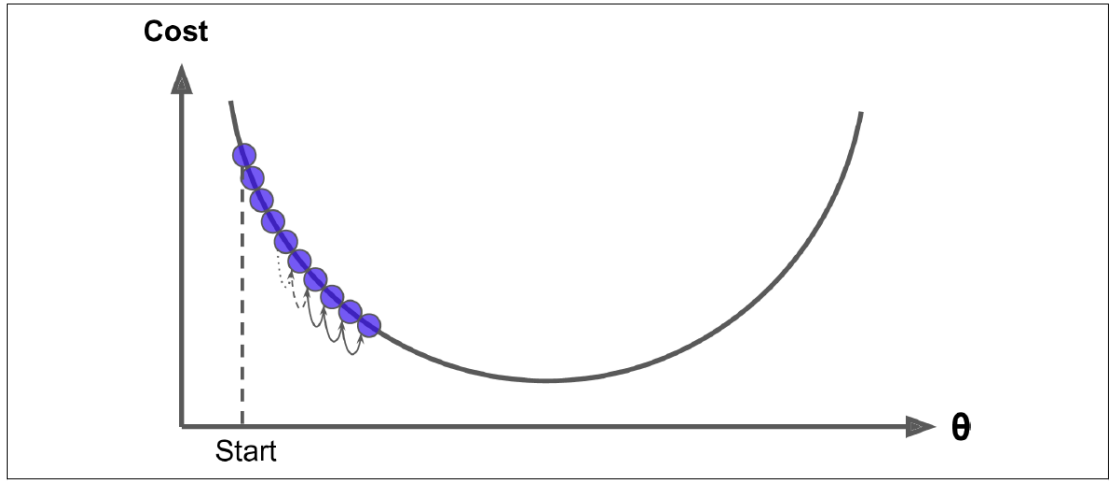


Figure 4-4. Learning rate too small

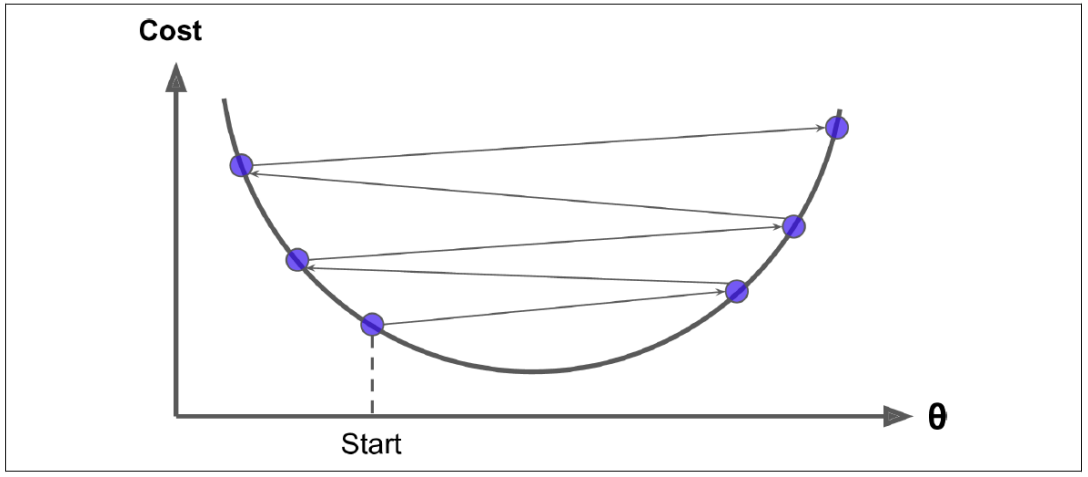


Figure 4-5. Learning rate too large

4.2. 경사 하강법(gradient descent, GD)

② 최솟값 문제 고려

다양한 비용함수의 그래프에서 문제점이 발생

- 왼쪽에서 시작하면 전역 최솟값(global minimum)보다 덜 좋은 지역 최솟값(local minimum)으로 수렴하는 문제
- 오른쪽에서 시작하면 평탄한 지역을 지나기 위해 오랜 시간이 걸리고 밀찍 멈추게 되어 전역 최솟값에 도달 하지 못함

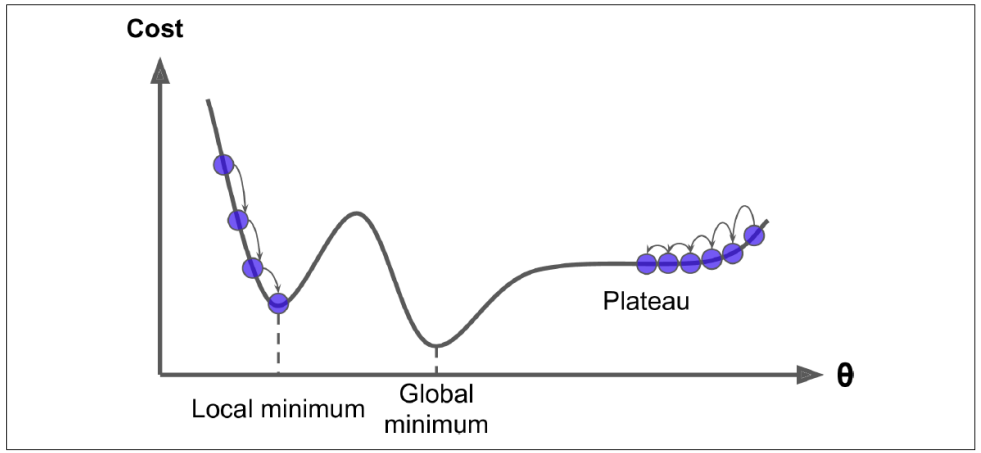


Figure 4-6. Gradient Descent pitfalls

③ 특성 스케일 고려

- 특성 스케일이 매우 다르면 길쭉한 모양
- 왼쪽의 경사 하강법 알고리즘이 최솟값으로 곧장 진행
- 오른쪽 그래프에서는 처음엔 전역 최솟값의 방향에 거의 직각으로 향하다가 평면 골짜기로 길게 돌아감
- 시간이 오래 걸림 => 사이킷런의 StandardScaler를 사용한다.

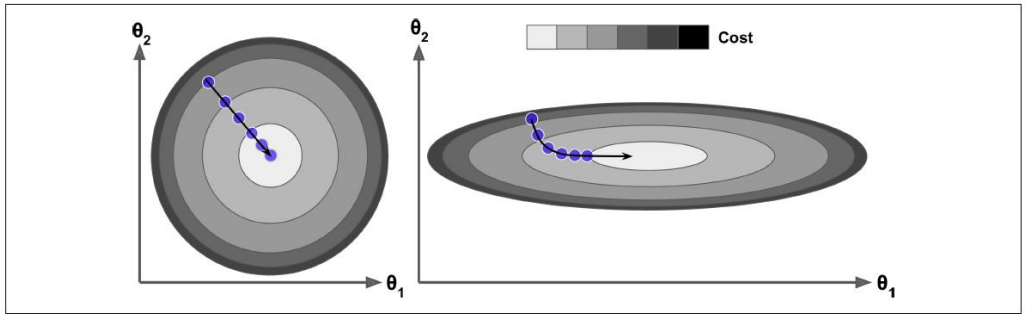


Figure 4-7. Gradient Descent with and without feature scaling

4.2. 경사 하강법(gradient descent, GD)

2. 배치 경사 하강법

- 배치 경사 하강법(BGD)은 비용함수의 gradient 계산에 전체 훈련 데이터 셋을 사용하는 방법.
- 모든 훈련 세트에 대해 계산 하므로 계산 시간도 엄청 길어지고, 소모되는 메모리도 엄청남.
- 적절한 학습률을 설정이 필요함 (너무 작으면 느리며, 너무 크면 발산하게 된다)
- 그리드 서치를 이용해서 적절한 학습률을 찾아야 하며 반복횟수와 허용오차를 지정하여 조절함.

※ 경사하강법은 특성 수에 민감하지 않기 때문에, 수십만 개의 특성에서 정규방정식이나 SVD분해보다 유리함.

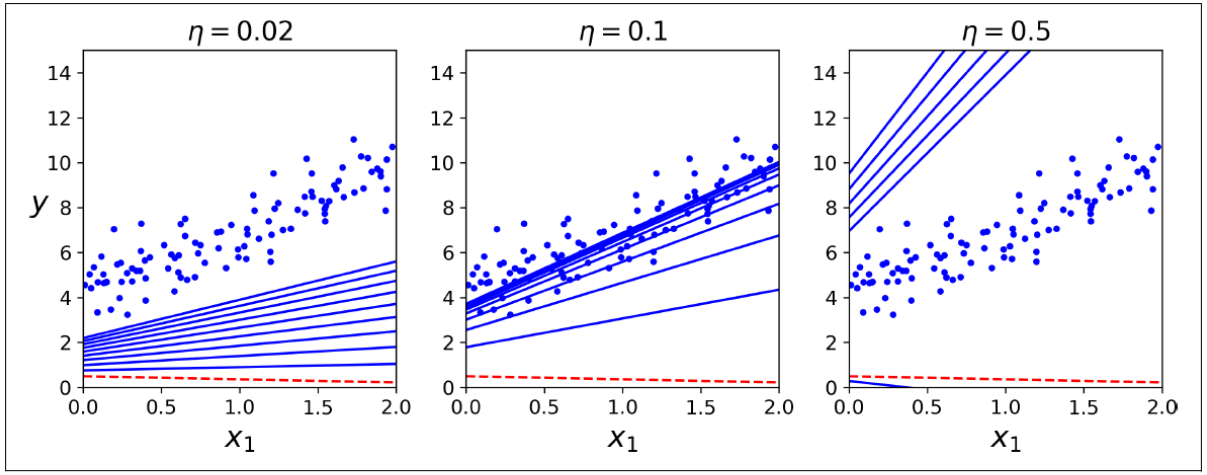


Figure 4-8. Gradient Descent with various learning rates

4.2. 경사 하강법(gradient descent, GD)

배치 경사 하강법 예제

```
[34] # 학습률
     eta = 0.1

     # 반복횟수
     n_iterations = 1000

     # 훈련세트
     m = 100

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

     # 경사하강법 적용
     for step 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]])
```

4.2. 경사 하강법(gradient descent, GD)

3. 확률적 경사 하강법 (stochastic gradient descent, SGD)

매 스텝에서 한 개의 샘플을 무작위로 선택하고 그 하나의 샘플에 대해 graident를 계산하는 방식

- 장점 : 매 반복에서 다뤄야 할 데이터가 매우 적어 알고리즘이 훨씬 빠르며, 매우 큰 훈련세트도 훈련에서 유용, 비용함수가 불규칙할수록 지역 최솟값을 건너뛸 가능성이 높음
- 단점 : 배치 경사 하강법 보다 훨씬 불안정 함 => 위아래로 요동치면서 평균적으로 감소 하며 전역최솟값을 다다르지 못할 수 있음.
- 보완 : 학습 스케줄(learning schedule) 이용 : 시작할 때 학습률을 크게 하여 수렴을 빠르게 하고 점차 작게 하여 전역 최솟값에 도달하게 함

※ https://box-world.tistory.com/70 참고

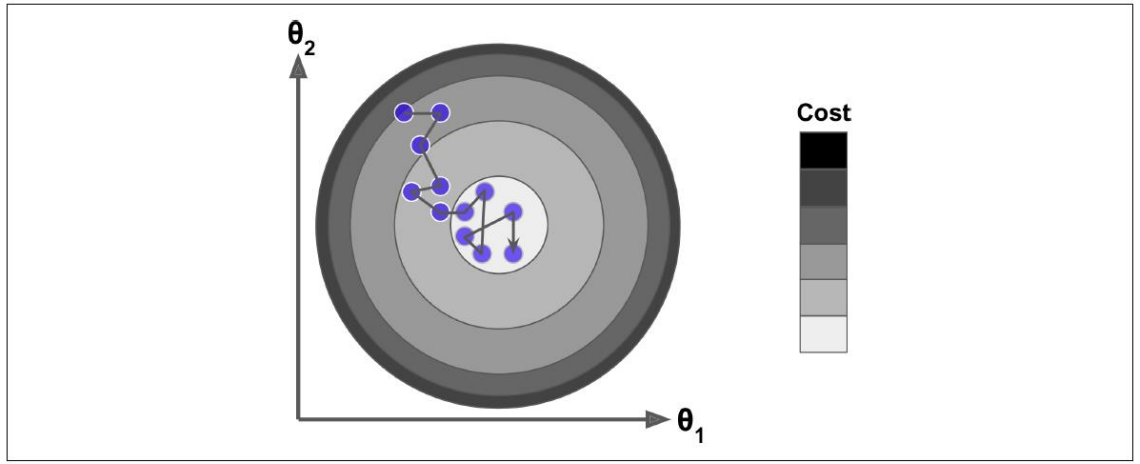


Figure 4-9. Stochastic Gradient Descent

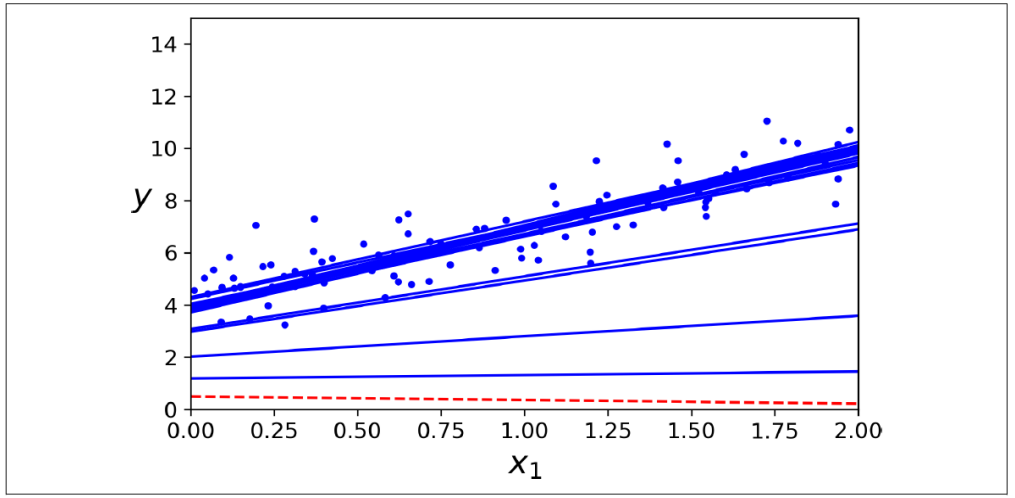


Figure 4-10. Stochastic Gradient Descent first 20 steps

※ 공평하게 무작위로 추출 => IID(independent and identrically distributed)를 만족 해야함,

※ https://pasus.tistory.com/51 참고

4.2. 경사 하강법(gradient descent, GD)

확률적경사 하강법 예제

```
n_epochs = 50
t0, t1 = 5, 50

# 학습 스케줄러 사용
def learning_schedule(t):
    return t0 / (t + t1)

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

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
```

```
[23] theta
```

```
array([[4.00597696],
       [2.27516959]])
```

4.2. 경사 하강법(gradient descent, GD)

확률적 경사 하강법 예제

```
[24] # 사이킷런에서 확률적 경사하강법 이용
     from sklearn.linear_model import SGDRegressor

     sgd_reg = SGDRegressor(max_iter=50, penalty=None, eta0=0.1, random_state=42)
     sgd_reg.fit(X, y.ravel())
```

```
     SGDRegressor(eta0=0.1, max_iter=50, penalty=None, random_state=42)
```

```
[36] sgd_reg.intercept_, sgd_reg.coef_
```

```
     (array([4.24365286]), array([2.8250878]))
```

4.2. 경사 하강법(gradient descent, GD)

4. 미니배치 경사 하강법

미니배치라 부르는 임의의 작은 샘플 세트에 대해 그레이디언트를 계산 진행 => 훈련세트에서 몇 개의 샘플의 가져와서 계산

- 행렬 연산에 최적화된 하드웨어 GPU를 사용해야 성능을 향상 시킴
- 미니배치를 어느정도 크게 하면 파라미터 공간에서 SGD보다 덜 불규칙하게 도달 => 최솟값에 더 가까이 도달 하지만, 지역 최솟값에서 더 빠져 나오기 힘들 수도 있음

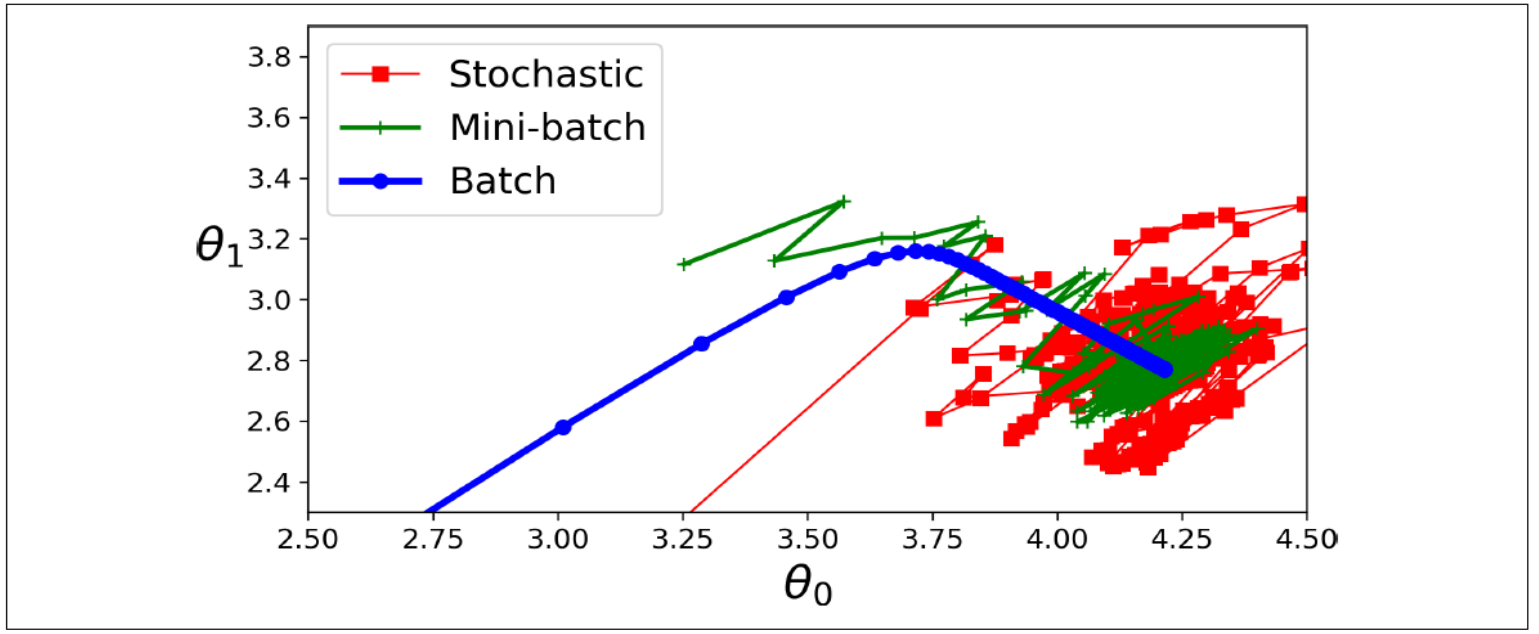


Figure 4-11. Gradient Descent paths in parameter space

4.2. 경사 하강법(gradient descent, GD)

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	n/a
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`