

Training Models [1] – Linear Regression

Samkeun Kim <skim@hknu.ac.kr>

<http://cyber.hankyong.ac.kr>

지금까지 ML 모델과 학습 알고리즘을 블랙박스 취급!!

그러나, 동작원리를 이해하면?

- 적절한 모델, 올바른 학습 알고리즘, 좋은 하이퍼파라미터 집합을 선택하는데
 - 에러 분석을 더 쉽게 하는데
 - 인공신경망을 이해하고, 구축하고, 학습시키는데
- 도움을 얻을 수 있다.

Linear Regression

Polynomial Regression

Logistic Regression

Softmax Regression

Linear Regression

Linear Regression 모델 (가장 단순한 형태의 모델) 학습시키는 방법

- “closed-form” 방정식 사용
 - ✓ Training set에 가장 적합한 모델 파라미터를 직접 계산
- Gradient Descent (GD) 최적화 방법 사용
 - ✓ 모델 파라미터를 점차적으로 조정하여 training set에 대한 비용 함수를 최소화
 - ✓ 결국 첫 번째 방법과 같은 파라미터 집합에 수렴

Linear Regression

Life satisfaction의 단순 회귀 모델:

$$\text{life_satisfaction} = \theta_0 + \theta_1 \times \text{GDP_per_capita}$$

← 입력 특성 GDP_per_capita의 선형 함수
 θ_0, θ_1 : 모델 파라미터

선형 회귀 모델 예측: 입력 특성들의 가중치 합 계산 => 예측 수행

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

- \hat{y} is the predicted value.
- n is the number of features.
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \dots, \theta_n$).

선형 회귀 모델 예측 (벡터 형태):

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

- $\boldsymbol{\theta}$ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n .
- \mathbf{x} is the instance's *feature vector*, containing x_0 to x_n , with x_0 always equal to 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$ is the dot product of the vectors $\boldsymbol{\theta}$ and \mathbf{x} , which is of course equal to $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$.
- $h_{\boldsymbol{\theta}}$ is the hypothesis function, using the model parameters $\boldsymbol{\theta}$.

NOTE

In Machine Learning, vectors are often represented as *column vectors*, which are 2D arrays with a single column. If $\boldsymbol{\theta}$ and \mathbf{x} are column vectors, then the prediction is $\hat{y} = \boldsymbol{\theta}^T \mathbf{x}$, where $\boldsymbol{\theta}^T$ is the *transpose* of $\boldsymbol{\theta}$ (a row vector instead of a column vector) and $\boldsymbol{\theta}^T \mathbf{x}$ is the matrix multiplication of $\boldsymbol{\theta}^T$ and \mathbf{x} . It is of course the same prediction, except that it is now represented as a single-cell matrix rather than a scalar value. In this book I will use this notation to avoid switching between dot products and matrix multiplications.

선형 회귀 모델, 어떻게 학습시킬 것인가?

- 모델을 학습시킨다는 것 => Training set에 가장 적합한 모델 파라미터를 설정하는 것
- 이를 위해 먼저 모델이 훈련 데이터에 얼마나 잘 맞는지 측정해야 한다.
- Root Mean Square Error (RMSE) - 2장에서
- 선형 회귀 모델을 학습시키기 위해 RMSE를 최소화하는 θ 값을 찾아야 한다.
- 실제로는 Mean Square Error (MSE) 를 찾는 것이 더 단순!
- Training set \mathbf{X} 에 대한 선형 회귀 hypothesis h_{θ} (예측)의 MSE:

$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m (\underbrace{\theta^T \mathbf{x}^{(i)}}_{\text{예측 값}} - \underbrace{y^{(i)}}_{\text{실제 값}})^2$$

The Normal Equation

정규 방정식:

- 비용 함수를 최소화하는 θ 값을 찾기 위해 결과 값을 직접 계산하는 "폐쇄형" 수학 방정식

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

- $\hat{\theta}$ is the value of θ that minimizes the cost function.
- y is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

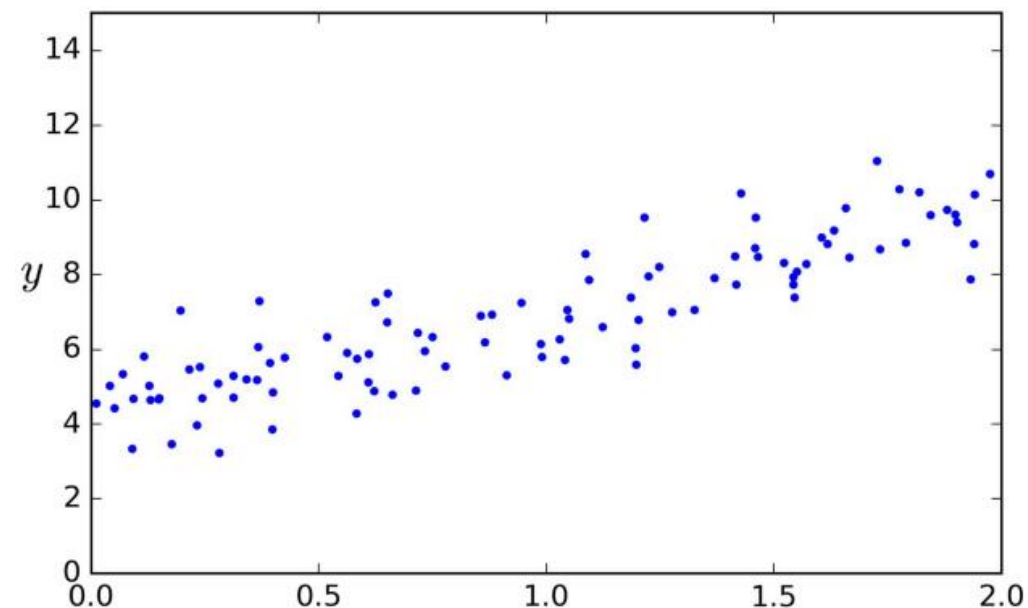
The Normal Equation

정규 방정식 테스트를 위해:

```
import numpy as np

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

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



NumPy's Linear Algebra 모듈 (np.linalg) inv() 함수 사용:

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

데이터 생성에 사용된 실제 함수: $y = 4 + 3x_1$

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

The Normal Equation

희망: $\theta_0 = 4, \theta_1 = 3$

결과: $\theta_0 = 4.215, \theta_1 = 2.770$

이유: 노이즈(noise)로 인해 오리지널 함수의 정확한 파라미터를 얻을 수 없음

$\hat{\theta}$ 을 이용한 예측 수행:

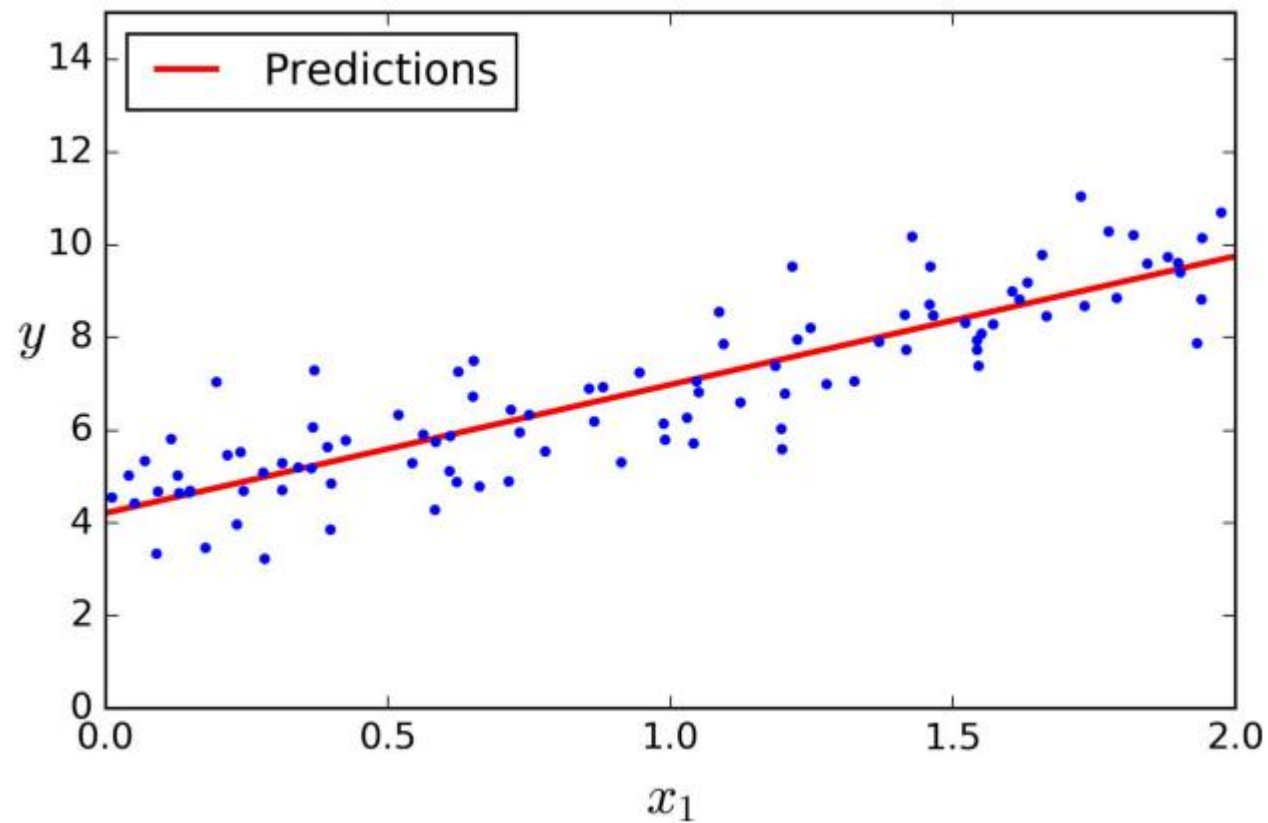
```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # add x0 = 1 to each instance
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[ 4.21509616],
       [ 9.75532293]])
```

$x = 0$: $4.21509616 + 2.77011339 \times 0 = 4.21509616$
$x = 2$: $4.21509616 + 2.77011339 \times 2 = 9.75532293$

The Normal Equation

모델 예측 Plot:

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



The Normal Equation

Scikit-Learn을 이용한 선형 회귀 수행:

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 4.21509616]), array([[ 2.77011339]]))
>>> lin_reg.predict(X_new)
array([[ 4.21509616],
       [ 9.75532293]])
```

⇒ 비슷한 결과!

Computational Complexity

정규 방정식 => $\mathbf{X}^T \mathbf{X}$ 의 역함수 계산

- $(n+1) \times (n+1)$ 행렬 (n : 특성 개수)
- 역행렬 계산 복잡도 => $O(n^{2.4}) \sim O(n^3)$
- 특성 개수 2배 증가 => $2^{2.4} = 5.3 \sim 2^3 = 8$

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

Scikit-Learn의 LinearRegression 클래스에서 사용하는 방식:

- $O(n^2)$
- 입력 특성 개수 2배 증가 => $2^2 = 4$

일단 모델을 학습시켰다면 예측은 매우 빠르게 수행할 수 있다 => 계산 복잡도: linear

이제 특성 개수가 많을 경우에 적합한 선형 회귀 모델을 학습시키는 방법에 대해 알아보자!

Gradient Descent

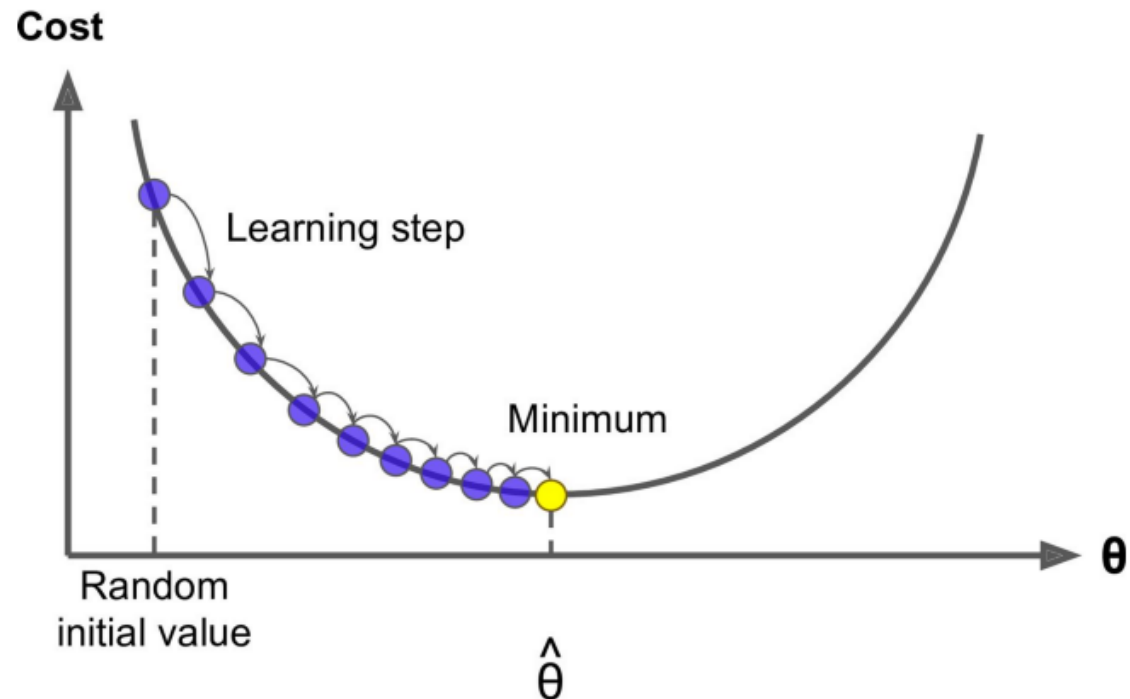
경사 하강법 (Gradient Descent)

- 폭넓은 문제에 대해 최적해를 찾아주는 일반적인 최적해 알고리즘
- 비용 함수를 최소화하기 위해 파라미터를 반복적으로 조정
- 예: 안개가 자욱한 산에서 길을 잃었다; 발 아래 지면의 기울기만 느낄 수 있다.
 - ✓ 계곡의 밑바닥에 빠르게 도착하기 위한 전략 => 가장 경사가 급한 쪽으로 내려 가는 것
- 경사 하강법이 하는 방식과 정확하게 일치:
 - ✓ 파라미터 벡터 θ 에 대해 비용 함수의 로컬 기울기 측정 => 하향 기울기 방향으로 내려 감
 - ✓ 기울기가 0(zero)이 되면 최소 값에 도착한 것으로 간주!!

Gradient Descent

GD 알고리즘 수행 방법:

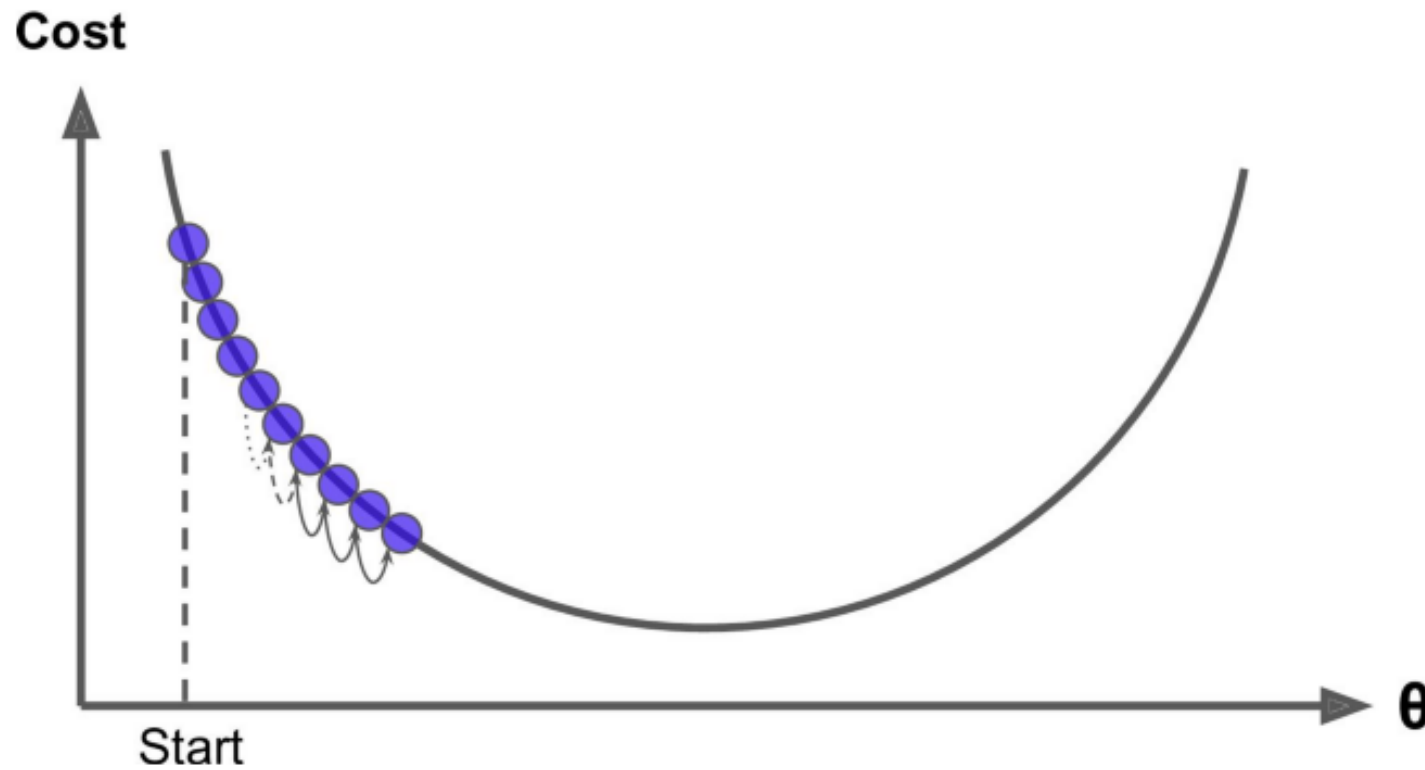
- θ 벡터를 랜덤 값으로 채우는 것부터 시작 (random initialization)
- 점차적으로 개선시켜 나감 => 한 번에 아기 걸음으로 한 걸음 씩
- 비용 함수(예: MSE)를 줄이는 방향으로
- 알고리즘이 최소 값에 수렴할 때까지



Gradient Descent

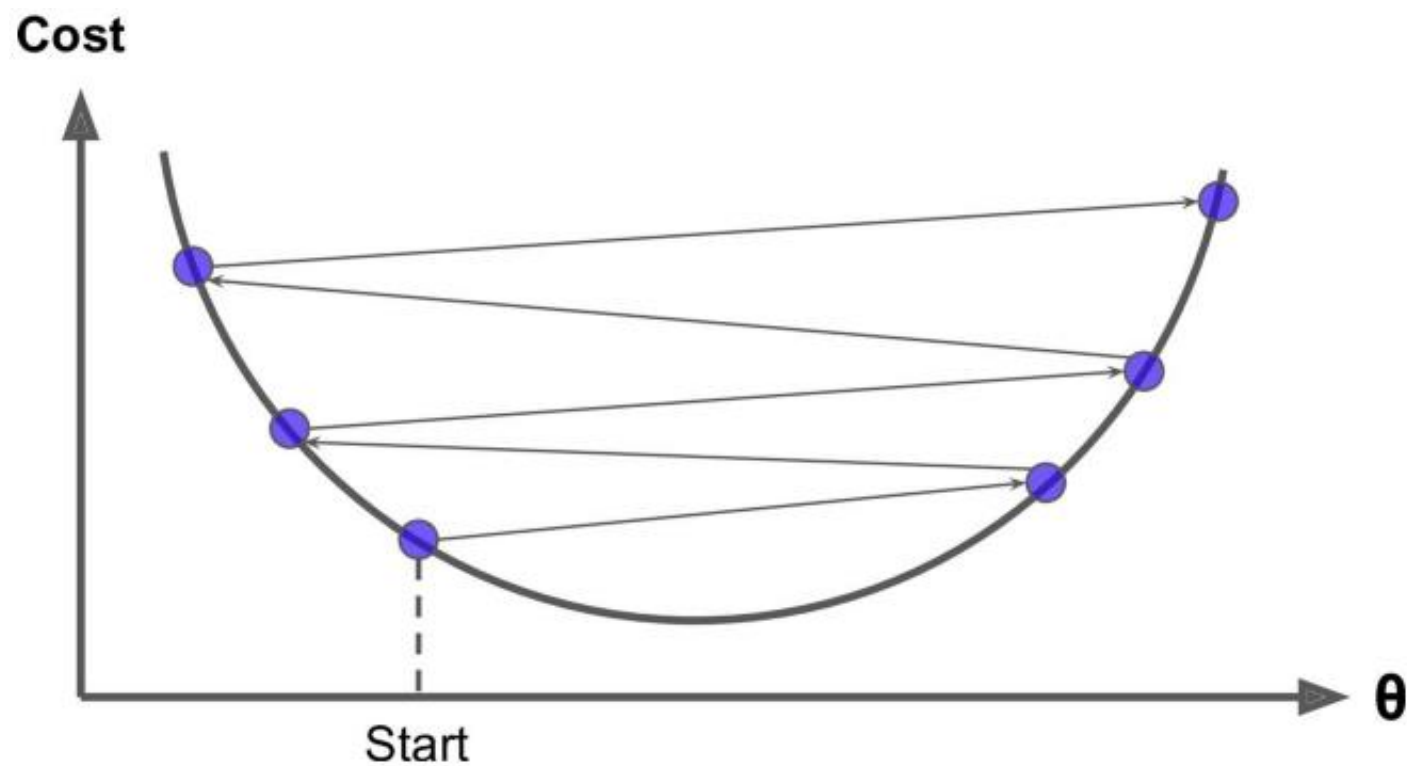
GD에서 중요 파라미터 => 스텝 사이즈 => 이는 학습률(learning rate)에 의해 결정 됨

- 학습률이 너무 낮으면 => 알고리즘이 수렴하는데 너무 많은 반복을 수행 => 긴 시간 소요



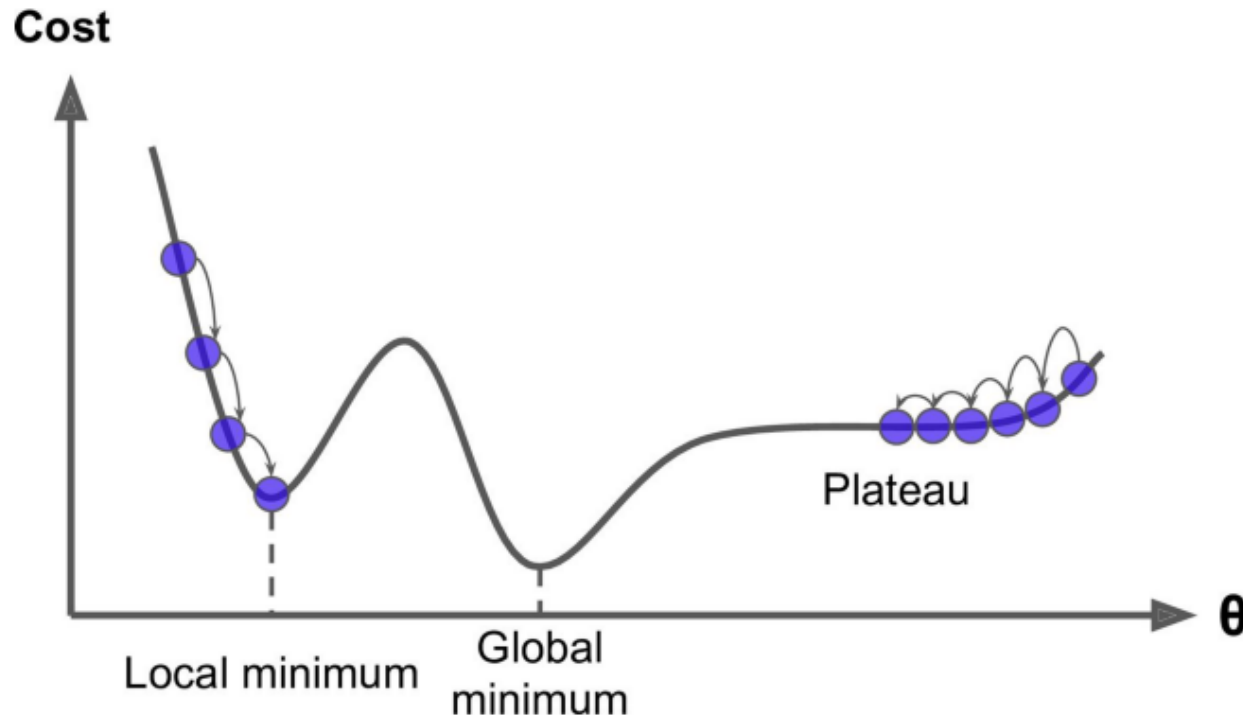
Gradient Descent

- 학습률이 너무 높으면 => 계곡을 가로질러 반대편으로 점프할 수 있음
- 심지어 앞 단계 에러 값보다 높게 될 수 있음



비용 함수가 모두 보기 좋은 오목한 그릇 형태를 갖는 것은 아니다!

- 구멍, 협곡, 고원 등의 수많은 불규칙한 지역이 존재 => 최소 값에 수렴하는 것을 어렵게 만들
- 랜덤 초기화 => 알고리즘을 왼쪽에서 출발하게 하는 경우 => local minimum에 수렴
- 알고리즘을 오른쪽에서 출발하게 하는 경우 => 고원을 가로질러 오는데 긴 시간 소요
(너무 일찍 멈추면 수렴을 못 할 수도 있음)



다행히도 선형 회귀 모델의 MSE 비용 함수

⇒ Convex 함수 => 굴곡이 없음 => Local minimum이 존재하지 않음

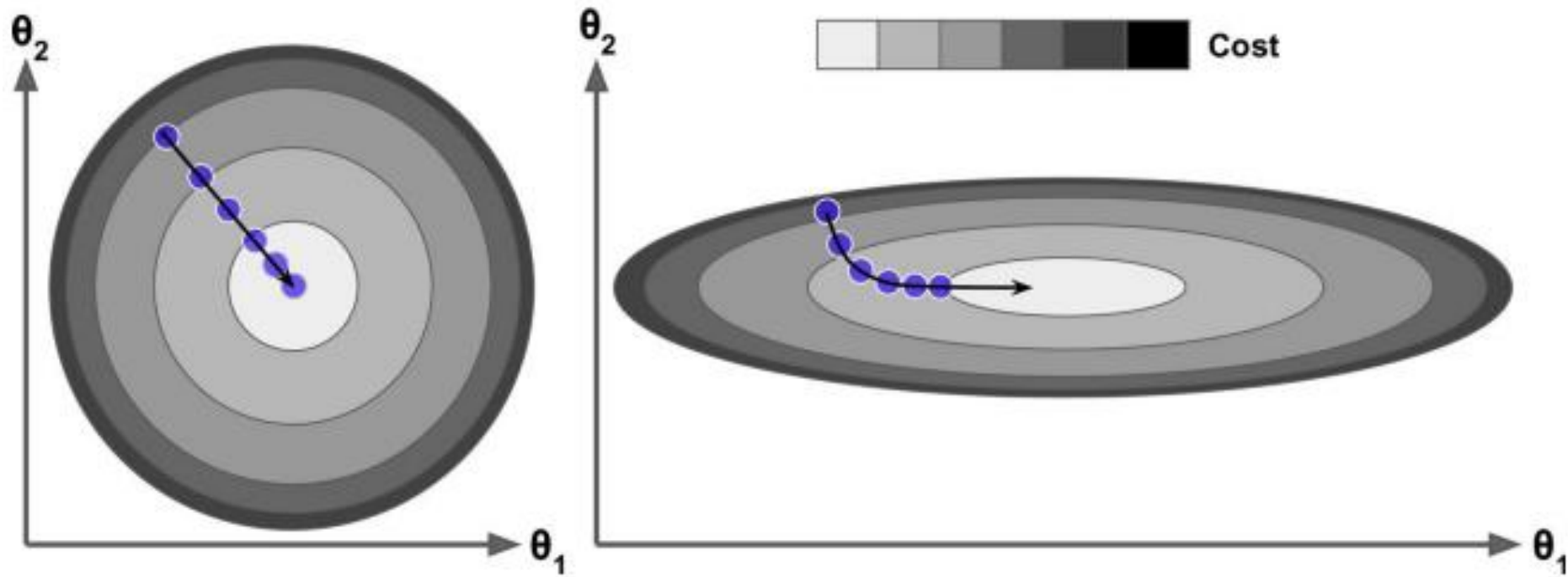
- 단 1개의 Global minimum만 존재
- 어떤 점에서든 기울기를 갖는 연속 함수

GD => 충분히 오랫동안 학습율을 너무 높지 않게 학습시키면 Global minimum에 도달하는 것을 보장 받는다!!

Gradient Descent

특성들이 매우 다른 크기의 단위를 갖는다면 아주 넓은 그릇 모양이 된다:

- 특성 1과 2가 동일한 단위 크기를 갖는 경우 (왼쪽)
- 특성 1이 특성 2보다 매우 작은 값을 가질 경우 (오른쪽)



Gradient Descent with and without feature scaling

WARNING

When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

모델을 학습시키는 것 => 비용 함수를 최소화하는 모델 파라미터 조합을 찾는 것!!

- 모델의 파라미터 공간(parameter space)에서 탐색
- 모델이 파라미터를 많이 가질수록 => 공간 차원이 많아지고 => 탐색이 어려워 짐
- 예: 300 차원의 건초더미에서 바늘을 찾는 것보다 3 차원에서 찾는 것이 훨씬 수월하다!!

Batch Gradient Descent

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

GD 구현

- 모델 파라미터 θ_j 에 대해 비용 함수의 기울기 계산
- θ_j 를 조금 변경했을 때 비용 함수가 얼마나 변하는가를 계산해야 => 편 도함수(partial derivative)
- 예: 내가 동쪽을 바라보고 있을 때 내 발 아래 산의 기울기는 얼마인가?
내가 북쪽을 바라보고 있을 때 내 발 아래 산의 기울기는 얼마인가? ...

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

편 도함수를 개별적으로 계산하는 대신 한꺼번에 계산할 수도 ...

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

위 공식은 각 GD 스텝에서 전체 training set \mathbf{X} 상에서 계산 수행

- Batch Gradient Descent라고 불리우는 이유
- 모든 스텝에서 전체 묶음의 training 데이터를 사용 => 대규모 training set의 경우 엄청나게 느려질 수도 ...
- 그러나, 수십만 개의 특성을 가졌을 때 정규 방정식 보다는 GD가 훨씬 더 빠르다!!

Batch Gradient Descent

Gradient Vector => 움직여야 할 양($\nabla_{\theta}MSE(\theta)$)을 계산한 것

- 내려가는 쪽으로 진행해야 => $\theta - \nabla_{\theta}MSE(\theta)$
- 학습률 η 로 스텝 사이즈 조절

Gradient Vector step:

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

Batch Gradient Descent

GD 알고리즘 구현:


```
eta = 0.1 # learning rate
n_iterations = 1000
m = 100

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

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

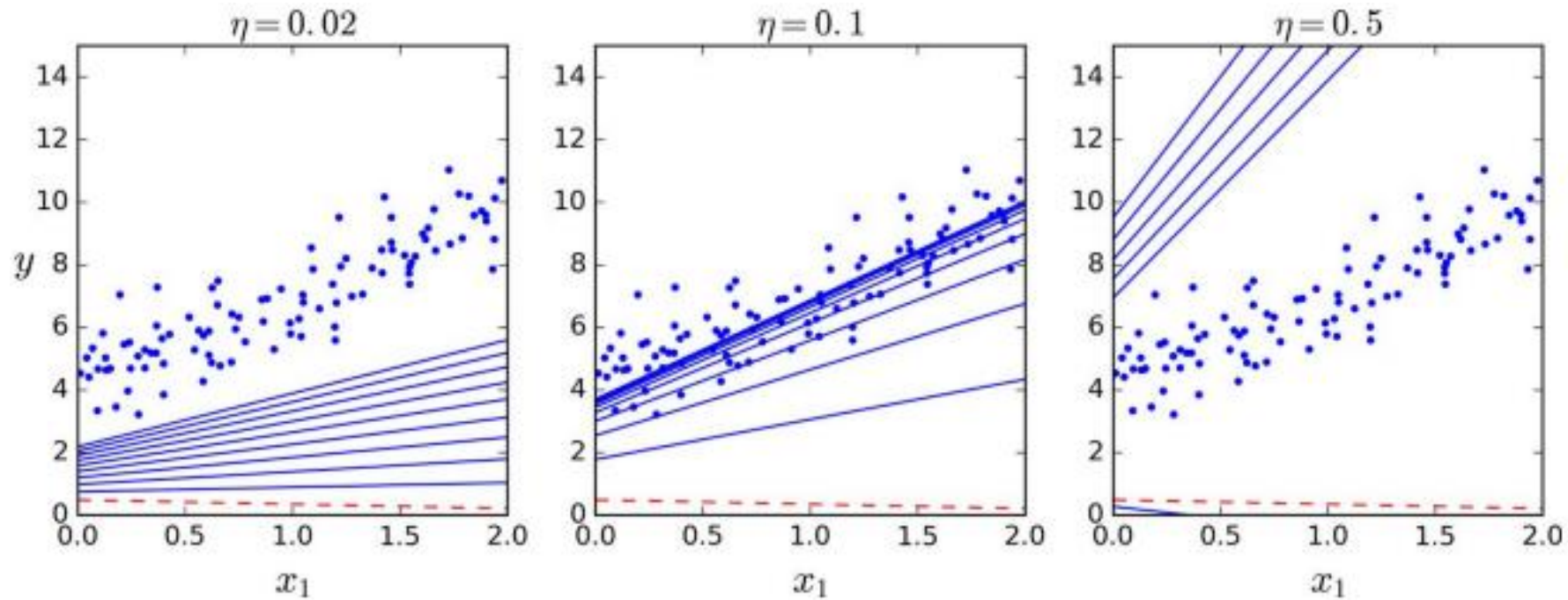
결과: (정규 방정식이 발견한 것과 정확히 일치한다!)

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


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

학습률 η 를 다양하게 바꿔보면:

- 3개의 다른 학습률에 대해 GD의 처음 10 스텝 (빨간색 점선: 시작 포인트)



왼쪽 그림 => 학습률이 너무 낮음 / 중앙 그림 => 학습률 적절 / 오른쪽 그림 => 학습률이 너무 높음

적절한 학습률을 찾기 위해 => Grid search 이용

- Grid search => 너무 오래 걸려서 수렴하지 못하는 모델을 제거할 수 있도록 반복 횟수를 제한할 수도!
- 반복횟수를 너무 낮게 설정하면 => 최적해에 수렴하지 못할 수도
- 반복횟수를 너무 높게 설정하면 => 시간 낭비

해결책: 반복 횟수를 크게 설정하고, gradient vector가 ϵ 보다 작아지면 알고리즘을 멈춘다.

Stochastic Gradient Descent

BGD 단점:

- 기울기를 계산하기 위해 매 스텝마다 전체 training set을 이용한다는 점
- Training set이 클 때 속도를 매우 느리게

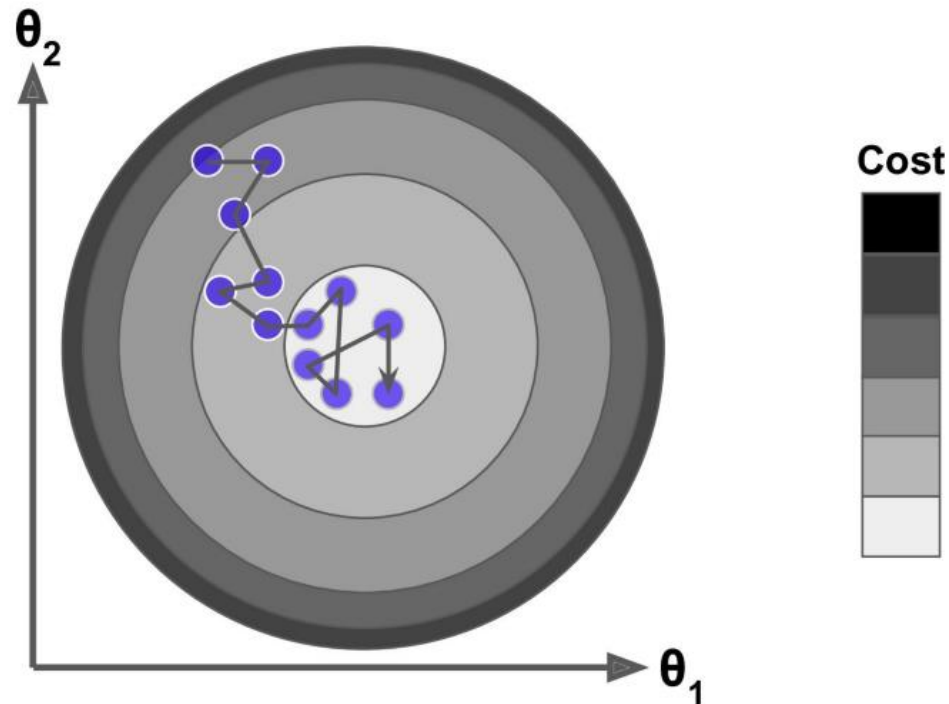
Stochastic Gradient Descent:

- 극한 경우 => 매 스텝마다 training set에서 단 한 개의 인스턴스를 무작위로 뽑아서
- 해당 인스턴스만 가지고 기울기를 계산
- => 알고리즘을 훨씬 더 빠르게 => 거대한 training set을 학습시키는 것도 가능

Stochastic Gradient Descent

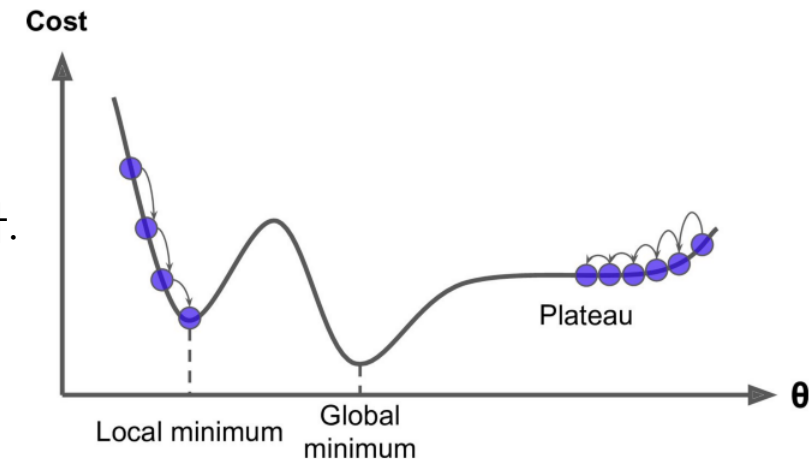
한편, 랜덤 특성으로 인하여 BGD 보다 훨씬 덜 규칙적:

- 최소 값에 도달하기 위해 비용함수가 부드럽게 감소하는 것이 아니라 위아래로 튕긴다 (평균적으로 감소)
- 시간이 지남에 따라 최소 값 근처에 도달하지만 계속 튕기게 되고 결코 안정되지 않음
- 알고리즘이 멈춘다 해도 최종 파라미터 값이 최적이지 아니다.



비용 함수가 매우 불규칙적이므로 local minima에서 벗어나는 데는 도움이 된다.

- 즉 SGD가 BGD 보다 global minima에 도달하기 쉽다.



Randomness:

- Local optima를 벗어나는 데는 좋지만, 알고리즘이 최소값에 도달하지 못할 수도 있다는 점에서는 나쁘다.

해결책: 학습률을 점차적으로 줄인다

- 처음에는 큰 값으로 시작해서, 점차 줄여서 알고리즘이 global minimum에 도달하도록 한다.
- “**simulated annealing**”이라 함 (금속공학 분야에서 담금질 과정과 유사)
- 매 반복에서 학습률을 결정하는 함수 => “**learning schedule**”
- 학습률이 너무 빠르게 줄어들면 local minimum에 빠질 수 있다.
- 학습률이 너무 천천히 줄어들면 긴 시간 동안 minimum 근처를 맴돌다가 학습을 너무 일찍 종료하게 되면 결국 준 최적해에 수렴하게 된다.

간단한 학습 스케줄을 이용한 Stochastic Gradient Descent 구현 코드:

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

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

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

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
```

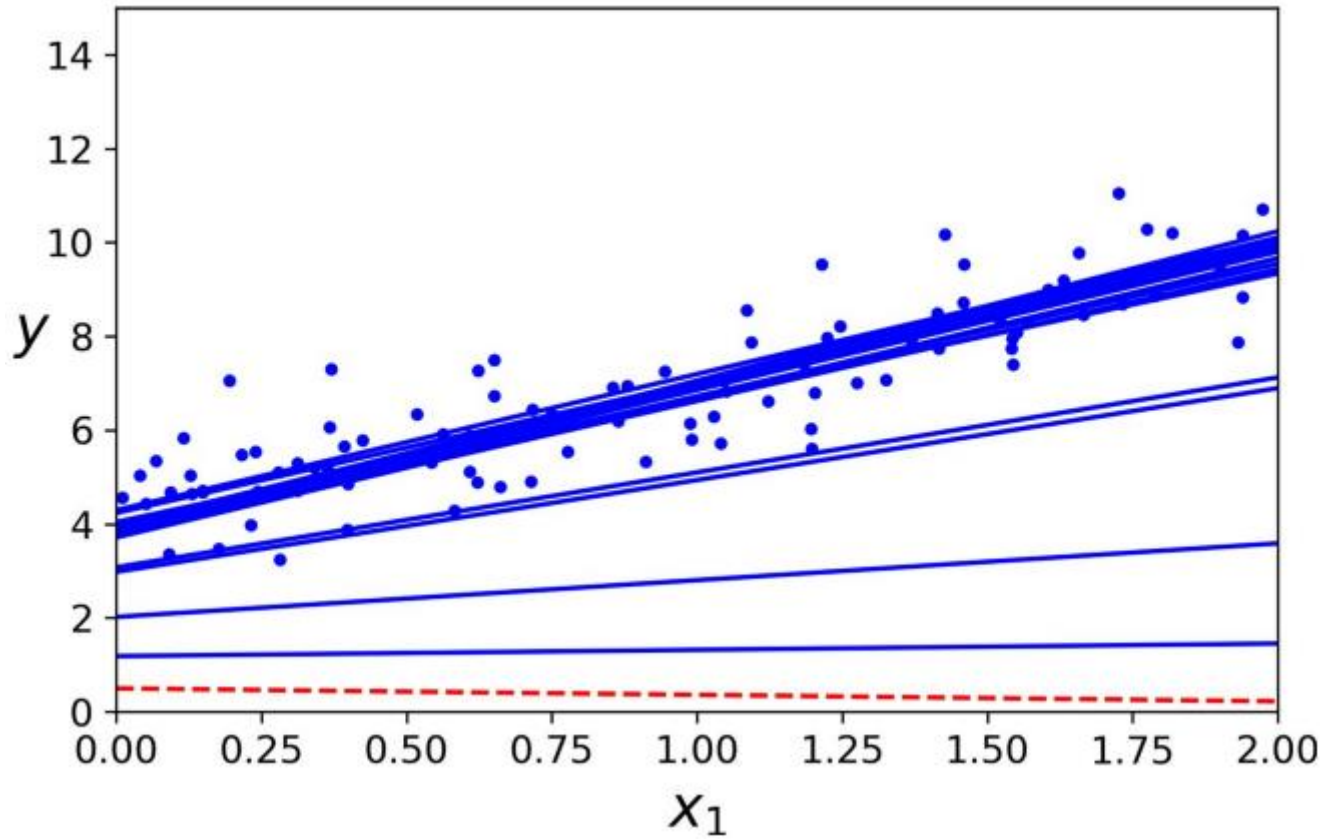
$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

- n_epochs번의 반복 수행 => 각 라운드를 "epoch"이라 함
- BGD => 전체 training set에 대해 1,000번 반복, SGD => 단지 50번만 반복해도 매우 훌륭한 해를 얻음

Stochastic Gradient Descent

SGD의 처음 20 스텝 (매우 불규칙적):



Scikit-Learn의 Stochastic GD를 이용한 Linear Regression을 수행하기 위해

- SGDRegressor 클래스 이용 => MSE 비용 함수 최적화
- 최대 1,000 epochs 까지 반복 수행하거나 기울기 값이 0.001 미만으로 떨어질 때까지 수행

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

- 다시 한번 정규방정식에 의한 값과 유사함을 확인할 수 있다:

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

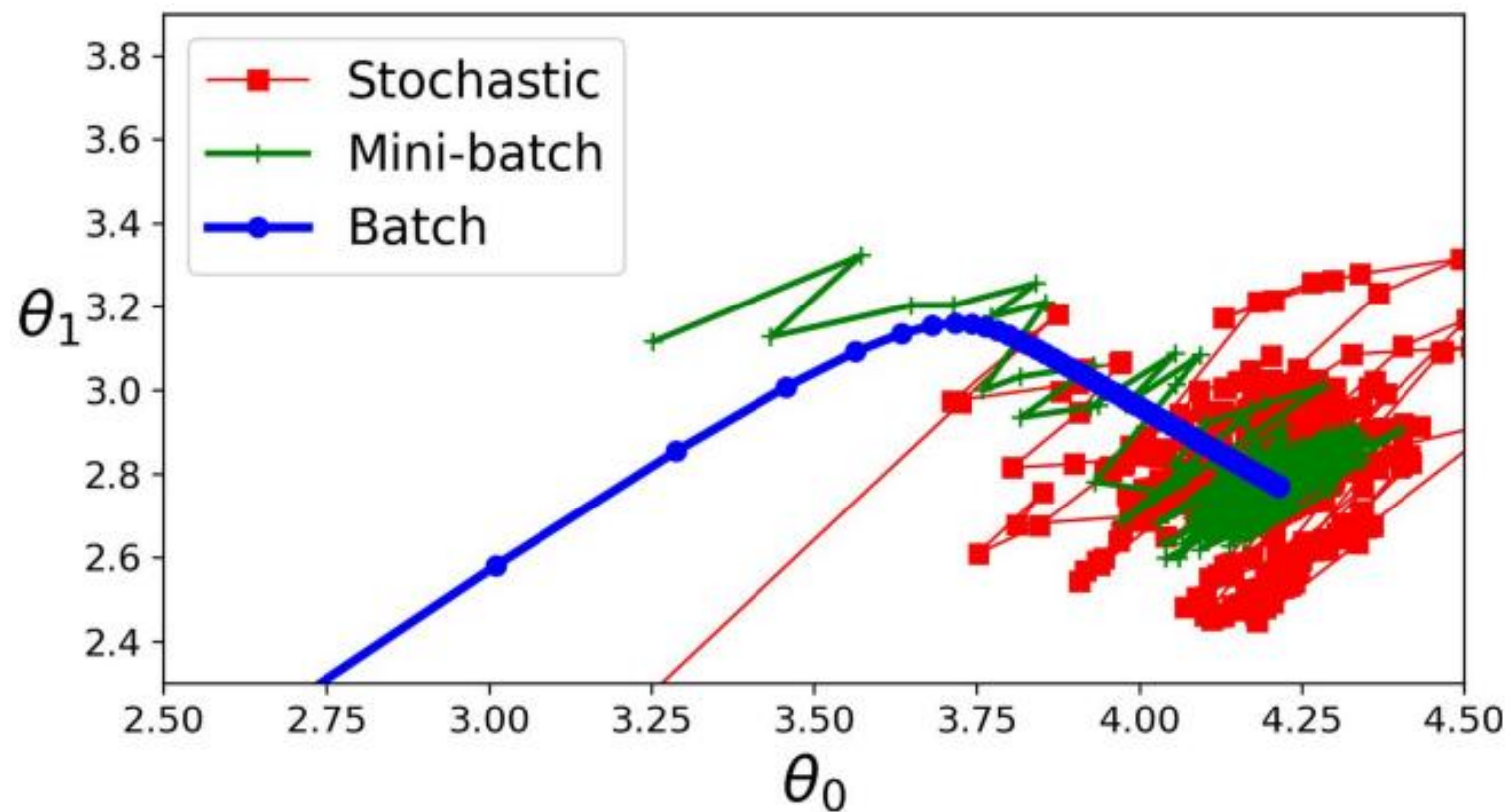
Mini-batch Gradient Descent

Mini-batch Gradient Descent:

- Batch GD => 전체 training set에 기반하여 기울기 계산
- Stochastic GD => 단 한 개의 인스턴스에 기반하여 기울기 계산
- Mini-batch GD => mini-batch라는 작은 랜덤 인스턴스 세트에 대해 기울기 계산
- Mini-batch GD가 Stochastic GD보다 좋은 점 => 덜 불규칙적 => Minimum에 더 가까이에 갈 수 있음
- 단점 => Local minima에서 벗어나기가 더 어려울 수 있다!

Mini-batch Gradient Descent

3개의 Gradient Descent 알고리즘 비교:



실습과제 5-1

본문에 나오는 전체 내용 PyCharm에서 실행하기

참고: [제 05강 실습과제 #5 Training Models \[1\] - Linear Regression.pdf](#)

