Gradient Descent

&

Normal Equation

Homework #1

32181928 박찬호

Dankook University

Mobile Systems Engineering

2022 Fall

Contents 목차

서론

Linear Regression

Linear 란 무엇일까? Linearity(Linear)는 additivity 와 homogeneity 를 동시에 만족하는 성질이다. 수식으로 보면, 임의의 수 x, y 에 대해 $f(ax+by)=af(x)+bf(y)$ 를 항상 만족하면 함수 f 는 linearity 를 만족한다고 한다.

그렇다면 Linearity 가 왜 중요한가? Linearity 는 단순한 입력들이 존재할 때 그 입력들의 스칼라배와 그 합으로 더 복잡한 입력을 표현할 수 있으면, 단순한 입력에서 얻어지는 단순한 출력으로 복잡한 출력을 구해낼 수 있다는 뜻이다. 따라서, 이러한 방법은 복잡한 관계를 단순한 속성들의 조합으로 표현할 수 있게 하기 때문에 어떠한 데이터를 설명하는 모델을 구성하거나, 또는 시스템을 설명할 때 막강한 도구가 된다. 따라서 linear 하다는 것은 중요한 특성이다.

Linearity 의 막강함을 자세히 들여다보자. 어떠한 데이터가 linearity 를 만족하는 상황에서 이 데이터에 우리가 관심을 갖는 값 y 가 존재한다고 생각하자. 이때, y 가 데이터의 다른 속성과 어떤 관계를 갖는지 궁금하거나, 또는 데이터의 다른 속성을 통해 y 의 값을 예측하고자 할 때 Linearity 를 만족하는 것만으로 문제는 상당히 단순해진다.

Linear Regression 은 이러한 상황에서 아주 효과적인 분석방법이다. 각 종속변수를 Y 로, 독립변수를 X 로, bias 를 포함한 독립변수의 계수를 행렬 θ 로 표현하면 $Y=\theta X$ 의 꼴로 문제를 나타낼 수 있게 되고, 결과적으로 우리가 해결할 문제는 θ 에 한정되게 된다. 즉, 데이터를 잘 설명하는 어떠한 model 을 찾는 과정이 linear regression 에서는 적절한 θ 를 구하는 문제로 바뀌는 것이다. Linear regression 에서 θ 를 찾는 방법은 Gradient Descent 와 Least Square 가 있다.

Cost Function

Gradient Descent 와 Least Square 로 향하기 이전에 optimal θ 와 잘 맞는 model 이라는 것은 과연 무엇일까? 어떠한 데이터가 존재할 때 그 데이터가 갖는 y 의 값과 model 에 x 를 입력해 얻은 출력이 비슷하다면, 우리는 직관적으로 그 model 이 그 데이터를 잘 설명한다고 생각한다. 이를 조금 더 엄밀히 정의하기 위해, 우리는 cost function 에 대해 알아보자.

Cost function 은 model 의 예측과 데이터의 y 가 얼만큼 비슷한지, 다른 말로는 얼만큼 차이나는지 설명하는 함수이다. 이때 이 차이를 Error 또는 Cost 라고 한다. Cost 는 다양한 방식으로 정의될 수 있는데, 주로 MSE, Cross-Entropy 를 사용한다.

다시 Optimal θ로 돌아가면, 결국 optimal θ를 찾는다는 것은 Cost function 의 결과가 가장 작은, model 의 error 가 가장 적은, 즉 model 이 데이터를 가장 비슷하게 설명하는 θ를 찾는다는 것이다. 따라서, Linear Regression 을 수행할 때의 우리의 목적은 cost function J 에 대해 $minimize_{\theta} J(\theta)$로 바뀌게 된다.

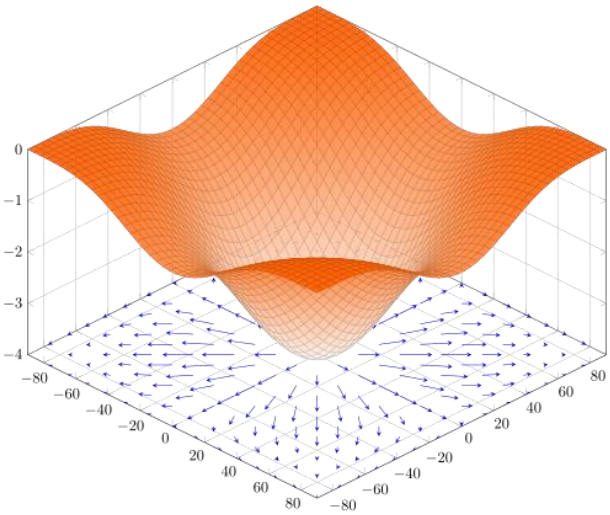
본론

Gradient Descent

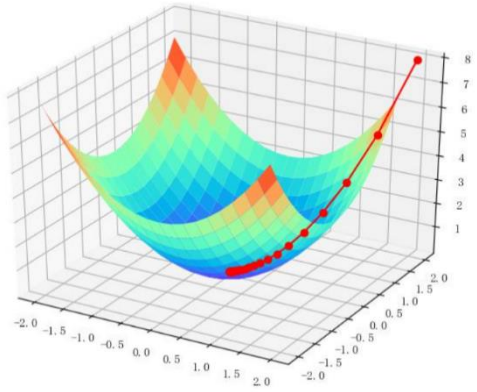
Gradient Descent 는 Gradient 라는 벡터장을 이용하여 cost function 의 optimal θ를 찾는 방법이다. 함수 F 가 존재하고, 이 함수의 출력이 $A \in \mathbb{R}^{m \times n}$인 행렬 A 일 때, gradient($\nabla$)는

$$\nabla_A f(A) \in \mathbb{R}^{m \times n} = \begin{bmatrix} \frac{\partial f(A)}{\partial A_{11}} & \cdots & \frac{\partial f(A)}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(A)}{\partial A_{n1}} & \cdots & \frac{\partial f(A)}{\partial A_{nn}} \end{bmatrix}, \left(\nabla_A f(A)\right)_{ij} = \frac{\partial f(A)}{\partial A_{ij}}$$

위와 같은 꼴을 가진다. 즉, Gradient 는 해당 성분에서의 편미분으로 해당 성분에서의 증가율과 그 방향을 갖는다. 이를 함수의 그래프와 비교해서 확인해보면 아래와 같다.



그래프에서 확인할 수 있듯 어떤 지점에서 함수의 기울기가 가파르면, 그 아래 투영된 gradient 도 크게 나타난다. 따라서 Convex 한 cost function 에서 gradient 에 음수를 곱하면, 크기는 gradient 의 크기에 비례하고 방향은 optimal point 를 향하는 벡터들을 얻을 수 있다(Convex 하지 않은 경우 local minimum 에 빠지는 것과 같은 문제가 발생할 수 있다. 그러나 여전히 gradient descent 를 사용할 수 있다). 이는 optimal point 에 가까워질수록 벡터의 크기가 더 작아지므로, 결과적으로 gradient 를 구하고 음수를 곱한 후, 이를 적용해서 이동한 다음 성분에서 다시 gradient 를 구하는 것을 반복하면 optimal point 로 향하게 될 수 있음을 보인다.



이를 다시금 정리하면 다음과 같다. 이때 alpha 는 step size 로 gradient 를 적용하는 정도이다.

$$repeat\ until\ convergence\ \{$$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\}$$

이러한 과정을 조금 특수한 경우에서 풀어 써보겠다. 먼저 parameter 로 θ_0와 θ_1이 존재하고,

$h_\theta(x) = \theta_0 + \theta_1 x$ 인 h 를 hypothesis 라고 가정한다. 이때 cost function J 로 MSE 를 사용한다고 생각하자. 그렇다면, m 이 전체 샘플의 수일 때 J 는

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

로 정의할 수 있다. 그렇다면, gradient descent 는

$$repeat\ until\ convergence\ \{$$

$$\theta_0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x^{(i)}$$

$$\}$$

가 된다.

과제의 Task1 이 위의 공식을 그대로 이용하는 문제이다.

Task1 에서는 y = ax + b 라는 모델을 gradient descent 를 이용하여 fitting 한다.

코드를 통해 살펴보겠다.

먼저 모델을 func 로 정의한다

```
# func is hypothesis h = θ0 + θ1 * X

def func(theta, X):
  return theta[0] + X * theta[1]
```

Gradient Descent 는 X, Y, 그리고 func 를 필수 인자로 받는다.

Epsilon 은 이전 iteration 과의 error 차이가 epsilon 보다 작으면 convergence 라고 판단하여 iteration 을 멈춘다

Alpha 는 step size, Epoch 은 iteration 횟수이다.

```
def gradient_descent(X, Y, func, epsilon=0.00001, batch_size_list=[0], alpha=0.05, epoch=351):
```

핵심 로직 부분이다. 각 코드에 맞는 부분을 수식에서 같은 색으로 표시하였다

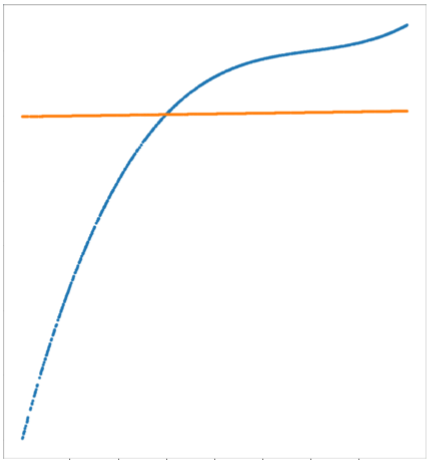
```
Y_pred = func(theta, X_in)
error = Y_in - Y_pred

div = -(1/m)

theta_diff = tuple(div * e for e in (np.sum(error), np.sum(X_in * error)))

theta[1] -= alpha * theta_diff[1]
theta[0] -= alpha * theta_diff[0]
```

$$repeat\ until\ convergence\ \{$$

$$\theta_0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)$$

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x^{(i)}$$

$$\}$$



이를 통해 얻어지는 full batch gradient 의 결과는 아래와 같다

`θ0 = 10.475902, θ1 = 12.278560, epoch = 267, error = 51.713806419871865`

(error 는 mse 이다)

Gradient Descent 는 m 의 크기에 따라 다른 batch size 를 가질 수 있다. 앞서 설명한 gradient descent 는 (full) batch gradient descent 로 전체 데이터를 batch 로 하여 gradient 를 계산하는 방법이다. 이러한 방법은 convergence 로 부드럽게 이어지지만, 속도가 느리고 계산 양이 많으며, local minimum 에 빠질 수 있다는 단점이 있다.

반면 Stochastic Gradient Descent(SGD)는 batch size (m)가 1 일 때의 경우로, 무작위로 하나의 샘플을 선택하여 gradient 를 계산하고 gradient descent 를 수행한다. 이는 빠르고, shooting 특성을 가지기 때문에 local minimum 을 피할 수 있지만, global minimum 에 도달한 후에도 계속 shooting 을 거듭하기 때문에 convergence 에 이르러서도 계속 에러가 들쑥날쑥한 경향이 있다.

이러한 문제를 해결하기 위해, mini-batch (stochastic) gradient descent 가 등장했다. Mini-batch 방법은 batch size 를 1 과 full batch 사이로 정함으로써, full batch 에서 생기는 속도, 계산 양 등의 문제를 해결하면서 동시에 SGD 에서 값이 튀는 문제도 해결하였다.

위의 세 방법은 아래의 코드에서 m 을 설정하면서 구현하였다.

```
m = len(X) # full batch
if batch_size > 0: # if batch_size, this function considers as full batch
  m = batch_size
```

또한, 매 iteration 에서 stochastic 하게 batch 를 구성하는 코드도 구현하였다.

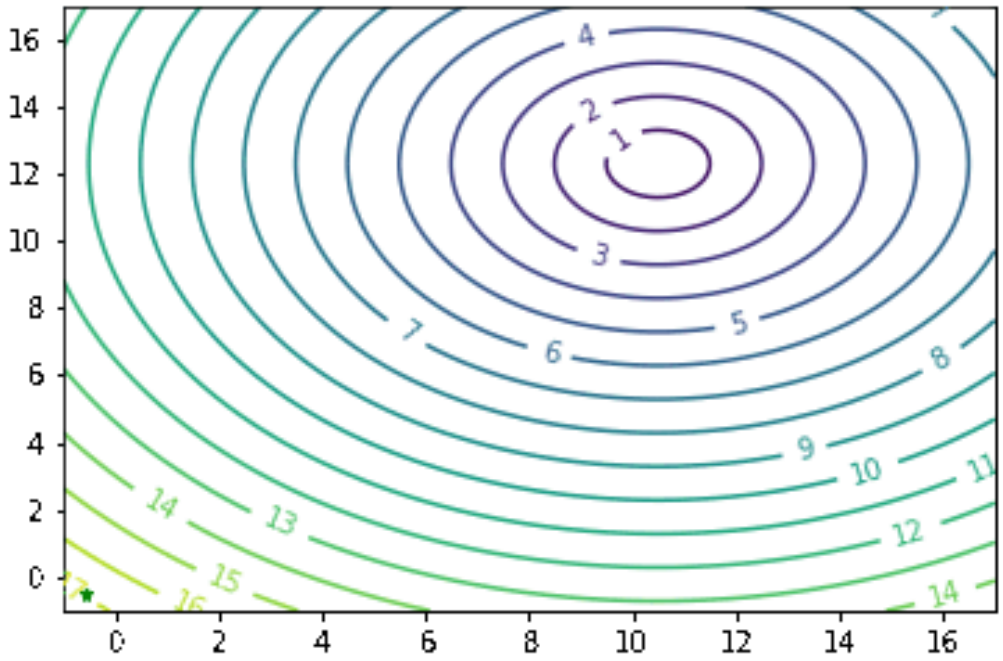
```
# X_in and Y_in are X and Y of batches
X_in = X
Y_in = Y

if m != len(X): # not full batch
  batch = np.random.randint(len(X), size=m) # randomly select batch samples
  X_in = X[batch]
  Y_in = Y[batch]
```

이를 통해 최종적으로 아래와 같은 결과를 얻을 수 있었다.

녹색 점 : full batch gradient descent

빨간 점 : mini batch gradient descent (batch size = 30)

파란 점 : stochastic gradient descent



Least Square

Least Square 는 Mean Squared Error 과 같은 squared error 를 사용한 cost function 에 한정해서 쓸 수 있는 cost optimization 방식이다. Squared Error 는 convex function 이고, squared term 으로 구성되어 있기 때문에 global minimum 을 바로 찾을 수 있다.

Convex function 에서 global minimum 은 derivative 가 0 이 되는 지점에 존재하는데, mean squared 과 같은 cost function J 는 multi variable function 이므로, 각 변수에 대해 partial derivative 를 수행하여 derivative 가 0 이 되는 지점을 찾아야 한다.

$h_\theta(x) = \theta_0 + \theta_1 x$ 와 같은 가장 단순한 경우를 생각해보자. 이때 h 의 각 변수에 대한 partial derivative 는 앞서 gradient 파트에서 다룬 것과 같이, 아래의 수식으로 정리된다.

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$\frac{\partial}{\partial\theta_0}J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

$$\frac{\partial}{\partial\theta_1}J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x^{(i)}$$

이를 통해 각 partial derivative 가 0 이 되는 θ_0와 θ_1을 찾을 수 있다

$$For\ \theta_0, \quad \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(\theta_0 + \theta_1 x^{(i)} - y^{(i)}\right) = 0, \quad so\ \theta_0 = \frac{\sum_{i=1}^{m}\left(y^{(i)} - \theta_1 x^{(i)}\right)}{m}$$

$$For\ \theta_1, \quad \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x^{(i)} = \frac{1}{m}\sum_{i=1}^{m}\left(\theta_0 x^{(i)} + \theta_1 {x^{(i)}}^2 - y^{(i)}x^{(i)}\right) = 0,$$

$$so\ \theta_1 = \frac{\sum_{i=1}^{m}\left(y^{(i)} - \theta_0\right)x^{(i)}}{\sum_{i=1}^{m}{x^{(i)}}^2}$$

위의 식을 다변수에 대해 일반적인 수식으로 행렬을 통해 정리하면 아래와 같다. 증명은 생략한다.

$$Normal\ Equation:\ \boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T y$$

이처럼, least square 를 이용하면 수식적으로 global minimum 을 바로 도출할 수 있다는 장점이 있다. 그러나, 행렬 곱을 수행하는 것이 많은 computation 을 요구하고, 또한 행렬이 symmetric 이고 inverse 가 존재해야 한다는 제약이 있다.

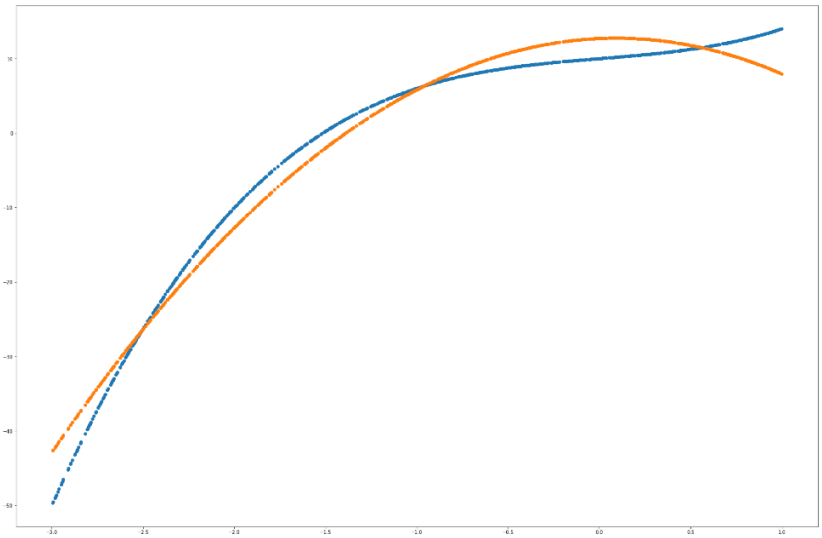
이를 코드로 구현하기 위해서는 위의 행렬 곱을 구현하면 된다.

Task2 에서는 $y = ax^2 + bx + c$ 모델에 대해 Normal Equation 을 통해 fitting 을 수행한다.

```
# Normal Equation

X_0 = np.ones(X.shape[0]) # X0 is ones
X_1 = X # X1 is X
X_2 = X**2 # X2 is X squared

X_mat = np.c_[X_0, np.c_[X_1, X_2]] # X_mat is [X0, X1, X2]

# variable names show intermediate steps to get normal equation
# @ is matrix multiplication
XTX = X_mat.T@X_mat
XTX_1 = np.linalg.inv(XTX) #inverse of XTX
XTX_1XT = XTX_1@X_mat.T
theta = XTX_1XT@Y

Y_pred = theta[0] * X_0 + theta[1] * X_1 + theta[2] * X_2

plt.figure(figsize=(30, 20))
plt.scatter(X,Y)
plt.scatter(X,Y_pred)
plt.savefig('normal equation')

print(theta)
```

결과는 아래와 같이 나온다.

```
[12.70482255  1.05708197 -5.82074893]
```



최종 결과

	a	b	c
Task1	12.278560	10.475902	0
Task2	-5.82074893	1.05708197	12.70482255

결론

느낀 점

Gradient Descent 를 잘 안다고 생각했는데, 막상 구현을 하는 과정에서 깔끔하게 로직을 작성하려다 보니 이해가 조금 얕은 부분을 발견할 수 있었습니다. 그러한 부분을 메꿀 수 있는 좋은 경험이었습니다.

실행 방법

HW1.ipynb 를 jupyter notebook 이나 colab 에서 열고 하나씩 셀을 실행하면 됩니다. Numpy, matplotlib 이 설치되어 있어야합니다.

제공해드리는 HW1.py 는 HW1.ipynb 와 동일한 코드입니다. conda run HW1.py 를 통해서 실행할 수 있습니다. 그러나 셀 환경에서 실행되지 않기 때문에 원활한 동작이 이뤄지지 않습니다. Plot 을 마지막에만 출력하도록 수정 후 실행해야 합니다