Machine Learning

Gradient Descent

HW1

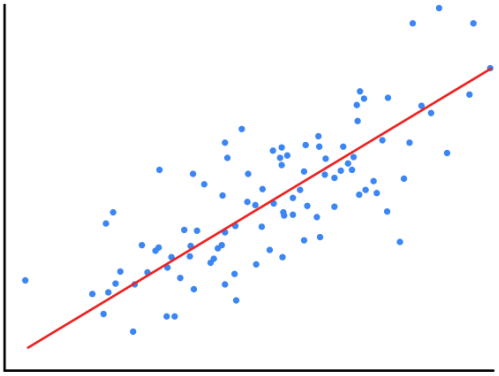
32171550 박다은

2020-10-9

Contents

1. Overview

(1) Introduction



경사하강법(Gradient descent)은 다음과 같은 데이터 분포를 선으로 모델링하는 알고리즘이다. 각 데이터 포인트와 직선과의 최단거리의 제곱의 합을 cost 라 하고, cost function 은 다음과 같다.

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

(2) Goals

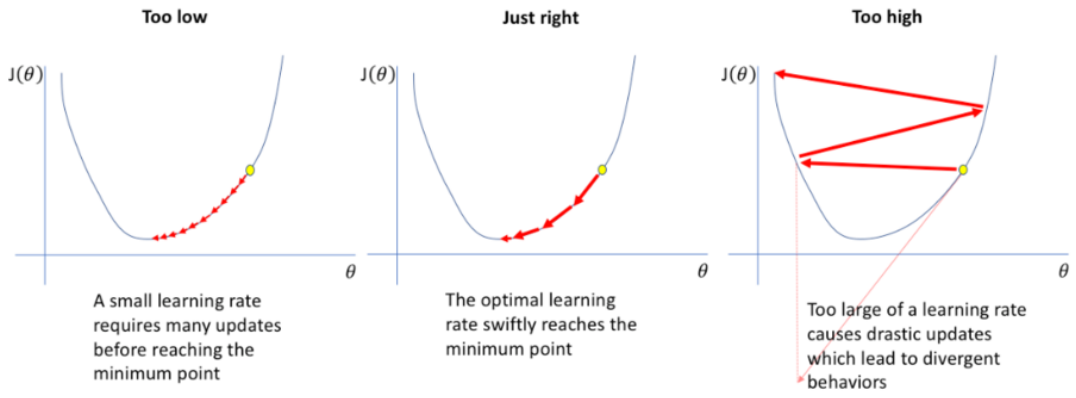
Cost function 이 최소가 되는 다음 직선방정식의 세타 값을 찾는 것이 주된 목표이다.

$$h_\theta(x) = \theta_0 + \theta_1 x$$

세타 값을 아래와 같이 업데이트 시키며 가장 최적화된 방정식을 찾아야 하는데, 이 때 Learning Rate(알파값)이 중요하다.

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

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



알파값이 너무 작으면 오랜시간이 소요되고, 알파값이 너무 크면 분산되므로 적절한 알파값을 찾는 것이 중요하다.

(3) Program Structure

Open_file

+----------------------------

| Import x3 datas and y datas

+----------------------------

Close_file

+----------------------------

| Set initial theta value

| update theta value

| Graphing equation

+----------------------------

(4) Build Environment

Jupyter Notebook, Python3

2.Code & Consideration

Input: hw1_data.csv Output: standard output

(1) Task1 주요 코드

csv 파일에서 필요한 값들 가져오기

```
x = []
y = []

f = open('hw1_data.csv', 'r', encoding='utf-8')
rdr = csv.reader(f)

i = 0
for line in rdr:
    i = i+1
    if i == 1:
        continue
    x.append(float(line[3]))
    y.append(float(line[7]))
f.close()
```

- csv 파일에서 필요한 부분인 x3 의 데이터들과 y 의 데이터들을 가져와 빈 리스트에 담도록 하였다.

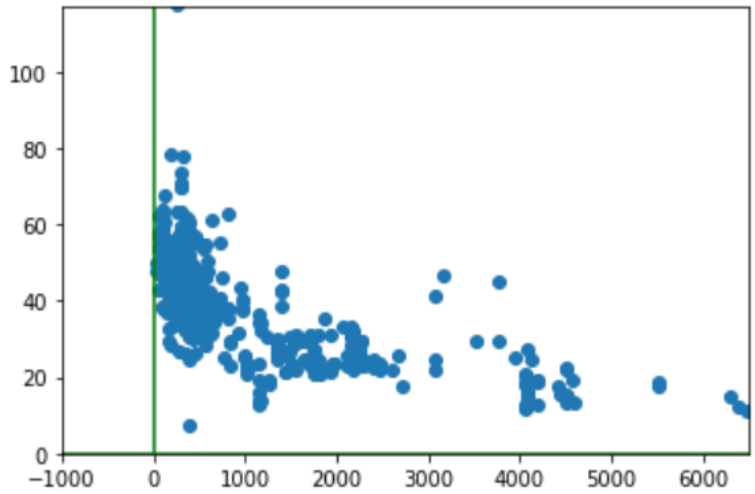
데이터 분포 확인

```
m = len(list(y))

print("max(x):", max(x), "min(x):", min(x))
print("max(y):", max(y), "min(y):", min(y))
```

```
max(x): 6488.021 min(x): 23.38284
max(y): 117.5 min(y): 7.6
```

- 데이터가 어떤 식으로 분포되어 있나 확인하기 위해, 먼저 리스트 x 와 y 의 최댓값과 최소값을 구하도록 하였다.

- 그 후 최댓값과 최소값의 구간에 맞게 그래프를 그려 초기 세타값을 정하는 것을 더 용이하게 하였다.



- 데이터의 분포는 다음과 같이 확인되었다.

세타, 알파 초기값 지정

```
1  theta0 = random.randint(40,50)
2  theta3 = random.random()
3  alpha = 0.0000001
4
5  print("theta0:", theta0)
6  print("theta3:", theta3)
```

```
theta0: 46
theta3: 0.5831716540516232
```

- 초기 세타값은 랜덤으로 받아오게 하였다.
- 위의 데이터 분포를 참고하여 효율적인 초기 세타값을 지정할 수 있도록 구간을 나누어주었다.
- 알파값은 0.1 부터 시작하여 프로그램을 여러 번 돌리면서 최적의 알파를 찾는 과정을 반복하였고, 현재 정해진 알파값은 이미 찾은 적절한 알파값이다.

경사하강법에 사용되는 함수들

```
def h(a,b,x):
    return a*x + b

def sum(a,b,x,y):
    s = 0
    for i,j in zip(x,y):
        s = s + ((h(a,b,i) - j) ** 2)
    return s

def sum_theta0(a,b,x,y):
    s = 0
    for i,j in zip(x,y):
        s = s + (h(a,b,i) - j)
    return s

def sum_theta3(a,b,x,y):
    s = 0
    for i,j in zip(x,y):
        s = s + ((h(a,b,i) - j) * i)
    return s

def j(x,y):
    return (1 / (2*m)) * sum(theta3,theta0,x,y)
```

- 코드를 깔끔하게 완성하기 위해 따로 함수들을 선언해주었다.
- h 와 j 는 각각 앞서 1-(2)와 1-(1)에서 언급한 방정식을 토대로 함수를 만들어주었다.
- sum 함수는 j 함수에서 사용되는 시그마를 따로 함수로 만든 것이다.
- sum_theta0 함수는 theta0 을 업그레이드하며 사용되는 시그마를 따로 함수로 만든 것이다.
- sum_theta3 함수도 마찬가지로 theta3 을 업그레이드하며 사용되는 시그마를 따로 함수로 만든 것이다.

Cost Function 최소화

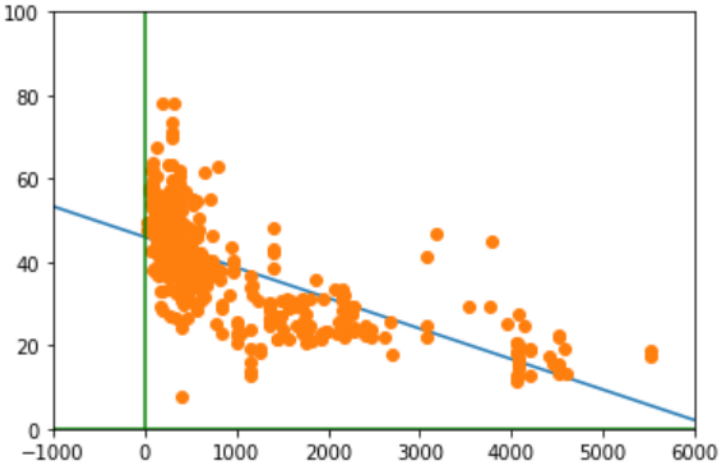
```
iteration = 0

while 1:
    iteration = iteration + 1

    print("#",iteration," theta3: ",theta3, "theta0: ",theta0)
```

```
    origin = j(x,y)

    updated_theta0 = theta0 - alpha * (1/m) * sum_theta0(theta3,theta0,x,y)
    updated_theta3 = theta3 - alpha * (1/m) * sum_theta3(theta3,theta0,x,y)

    theta0 = updated_theta0
    theta3 = updated_theta3

    if origin < j(x,y):
        print("Set the alpha again.")
        break

    if iteration > 10000:
        break
```

- While 문을 통해 세타를 업그레이드하는 과정을 반복하며 비용함수가 최소가 되는 세타값을 찾도록 하였다.
- Theta0 과 theta3 의 값이 동시에 업데이트 돼야 하므로 updated_theta 라는 변수를 만들어 주었다. Theta0 에서 바뀐 값이 바로 theta3 에 적용되는 것을 방지하기 위함이다.
- Iteration 이 10000 을 넘으면 프로그램이 과부하가 돼 10000 까지만 반복하도록 설정하였다.
- 만약 비용함수의 값이 직전값보다 높아지면 분산되고있다는 것이므로 while 문을 멈추고 다시 위로 돌아가 알파의 값을 다시 조정했다. 또한 비용함수의 값이 계속해서 크게 줄어들지 않는다면 알파의 값이 너무 작다는 것이므로 이 경우에도 알파의 값을 다시 조정했다.

```
# 1  theta3:  0.5831716540516232 theta0:  46
       j(x,y):  481906.5543575127

# 2  theta3:  0.41996668954752214 theta0:  45.99993598887832
       j(x,y):  252357.09704962262

# 3  theta3:  0.3018696450192602 theta0:  45.999889667315585
       j(x,y):  132161.76597656196

# 4  theta3:  0.21641322542659436 theta0:  45.99985614612713
       j(x,y):  69225.79661643656

# 5  theta3:  0.15457594847577333 theta0:  45.99983188744105
       j(x,y):  36271.636051822126
```

- 이 때, 출력은 다음과 같이 10000 번째까지 반복된다. 비용함수가 점점 줄어드는 것을 확인할 수 있다.



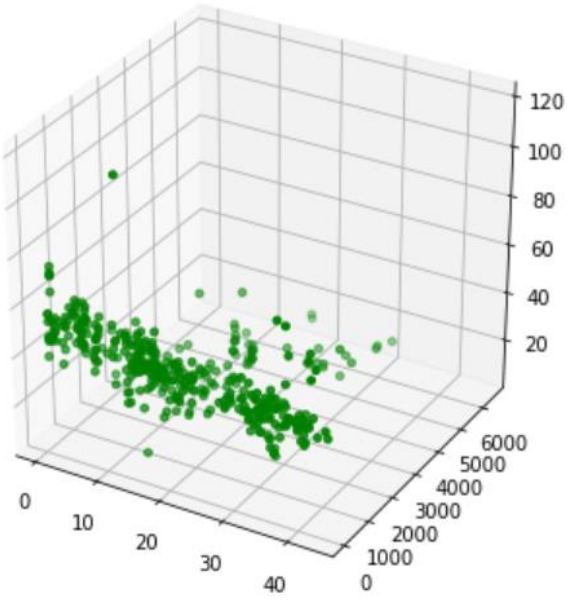

```
1 print("theta0:", theta0, "theta3:",theta3)
```

```
theta0: 45.99968316044594 theta3: -0.0073201918531169636
```

- 최종 그래프와 이 때의 세타값은 다음과 같다.

(2) Task2 주요 코드

- Task2 는 Task1 과 코드가 거의 유사하므로 확연히 차이나는 부분만 작성하도록 하겠다.



- 먼저 코드를 통해 확인한 데이터 분포는 다음과 같다.

경사하강법에 사용되는 함수들

```
def h(a,b,c,x2,x3):
    return a*x2 + b*x3 + c

def sum(a,b,c,x2,x3,y):
    s = 0
    for i,j,k in zip(x2,x3,y):
        s = s + ((h(a,b,c,i,j) - k) ** 2)
    return s

def sum_theta0(a,b,c,x2,x3,y):
    s = 0
    for i,j,k in zip(x2,x3,y):
        s = s + (h(a,b,c,i,j) - k)
    return s

def sum_theta2(a,b,c,x2,x3,y):
    s = 0
    for i,j,k in zip(x2,x3,y):
        s = s + ((h(a,b,c,i,j) - k) * i)
    return s

def sum_theta3(a,b,c,x2,x3,y):
    s = 0
    for i,j,k in zip(x2,x3,y):
        s = s + ((h(a,b,c,i,j) - k) * j)
    return s
```

- Task1 에서 사용했던 함수들을 Task2 에 맞게 변경해주었다.
- Theta2 의 추가로 인해 sum_theta2 라는 함수도 추가시켰다.

Cost Function 최소화

```
iteration = 0

while 1:
    iteration = iteration + 1

    print("#",iteration," theta2: ",theta2,"theta3: ",theta3, "theta0: ",theta0)
    print("       j(x2,x3,y): ",j(x2,x3,y),"₩n")

    origin = j(x2,x3,y)

    updated_theta0 = theta0 - alpha * (1/m) * sum_theta0(theta2,theta3,theta0,x2,x3,y)
    updated_theta2 = theta2 - alpha * (1/m) * sum_theta2(theta2,theta3,theta0,x2,x3,y)
    updated_theta3 = theta3 - alpha * (1/m) * sum_theta3(theta2,theta3,theta0,x2,x3,y)

```

```
    theta0 = updated_theta0
    theta2 = updated_theta2
    theta3 = updated_theta3

    if origin < j(x2,x3,y):
        print("Set the alpha again.")
        break

    if iteration > 10000:
        break
```

- while 문의 과정에서도 theta2 를 새로 추가시켜 같이 값이 업데이트되도록 했다.

```
# 1  theta2:  0.7130249181904895 theta3:  0.9786767152223484 theta0:  47
     j(x2,x3,y):  1358532.0656098097

# 2  theta2:  0.7110592137105032 theta3:  0.7046553644383186 theta0:  46.999891757701064
     j(x2,x3,y):  711393.6152402868

# 3  theta2:  0.7096297442280699 theta3:  0.5063741169853675 theta0:  46.999813219676774
     j(x2,x3,y):  372555.7723511248

# 4  theta2:  0.7085882935264736 theta3:  0.36289825581126123 theta0:  46.999756175612944
     j(x2,x3,y):  195142.292012525

# 5  theta2:  0.7078276126497002 theta3:  0.25907946180554403 theta0:  46.99971468454276
     j(x2,x3,y):  102249.64683492646
```

- 위 과정의 출력은 다음과 같다. 역시 비용함수가 줄어들고 있는 것을 확인할 수 있다.

```
print("theta0:", theta0, " theta2:",theta2, " theta3:",theta3)
```

```
theta0: 46.99303121666305  theta2: 0.48488431729978865  theta3: -0.01114279433870516
```

- 최종 세타값이다.

(3) Task3 주요 코드

csv 파일에서 필요한 값들 가져오기

```
X_list = [[], [], []]
y_list = []

f = open('hw1_data.csv', 'r', encoding='utf-8')
rdr = csv.reader(f)
```

```
i = 0
for line in rdr:
    i = i+1
    if i == 1:
        continue
    X_list[0].append(1)
    X_list[1].append(float(line[2]))
    X_list[2].append(float(line[3]))
    y_list.append(float(line[7]))

X,y =np.array(X_list).transpose(),np.array(y_list)

f.close()
```

- Task1, 2 와는 다르게 데이터를 행렬로 바꾸어 사용한다.
- Feature matrix 인 X 에 1 행에는 1(x0)을 채우도록 하였고, 2 행과 3 행에는 각각 x2 와 x3 의 값들로 채우도록 하였다.
- 리스트를 행렬로 바꾸는 과정에서, X 가 1 열에 x0, 2 열에 x2, 3 열에 x3 이 되도록 전치하여 바꾸어 주었다.

정규화

```
theta = numpy.dot(numpy.dot(numpy.linalg.inv(numpy.dot(X.transpose(),X)), X.transpose()), y)
```

- 파이썬 내장함수를 사용하여 정규화를 시켜주었다.

세타값 출력

```
print("theta0: ", theta[0])
print("theta2: ", theta[1])
print("theta3: ", theta[2])
```

```
theta0:  49.88558575690665
theta2:  -0.23102658345724777
theta3:  -0.007208620143015241
```

- Normal Equation 을 통해 구한 세타의 값은 다음과 같다.

(4) ‘Gradient Descent’ vs ‘Normal Equation’

A. Task1

```
theta0: 45.99968316044594 theta3: -0.0073201918531169636
```

먼저 Gradient Desecent 로 구한 세타값은 다음과 같다.

```
theta0: 45.85142705777496  theta3: -0.007262051618149452
```

Normal Equation 으로 구한 세타값은 다음과 같다.

값이 99%일치하는 것을 확인할 수 있다.

B. Task2

```
theta0: 46.99303121666305  theta2: 0.48488431729978865  theta3: -0.011142794338705l6
```

Gradient Descent 로 구한 세타의 값은 다음과 같다.

```
theta0: 49.88558575690665  theta2: -0.23102658345724777  theta3: -0.007208620143015241
```

Normal Equation 으로 구한 세타값은 다음과 같다.

Theta2 의 부호가 반대이긴 하지만 두 경우가 큰 차이가 없는 것을 확인할 수 있다.

(5) Personal Feeling

이번 과제는 여러 과제들과 완전히 겹쳐 많은 시간을 쏟을 수가 없었다. 따라서 확실히 개념을 습득한 후에 코드를 작성해야겠다고 생각했고 그렇게 했다. 개념을 정확히 모른 채 코드를 작성하면 훨씬 많은 시간이 걸리고, 오류도 훨씬 많이 생긴다는 것을 알고 있었기 때문이다.

과제를 끝낸 지금은 관련 개념들을 정확히 파악한 상태이고 크게 어렵지 않은 내용이라는 것을 알지만, 과제를 처음 시작할 때는 '머신러닝'이라는 단어의 웅장함과 여러 그래프들, 수학 식들로 인해 많이 막막한 상태였다. 강의를 여러 번 들을 수 있다는 비대면강의의 장점을 이용해 모르는 부분은 계속 돌려보고 연구하며 개념을 습득했다.

따라서 코드를 작성하는 과정은 크게 어렵지는 않았다. 중간중간 자잘한 오류들도 많았고, 파이썬이 익숙하지 않아 이것저것 찾아보는 데 많은 시간을 쏟았지만 생각보다 단기간 내에 끝낼 수 있었다.

또한 이번 과제는 '기계학습의기초및응용' 과목의 실질적인 첫 과제였다. 그 동안은 강의만 듣느라, 이해하고 있는 게 맞는건지, 잘하고 있는건지 헷갈렸었는데 과제를 끝 마치고 나니 더 흥미가 생기고 이해가 잘 되는 것 같다. 이론으로만 배울 때보다 이렇게 직접 구현해보며 원리를 이해하는 게 훨씬 이해도도 높고 재미도 있다.

앞으로도 과제가 몇 개 남아있는 것으로 알고 있는데, 다음 과제들도 충분한 이해 후에 과제를 시작해서 어려움 없이 끝내고 싶다.