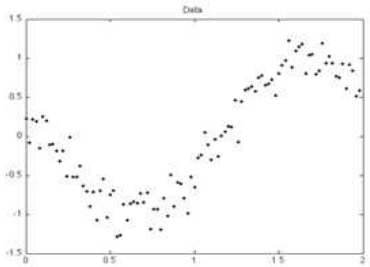
REPORT

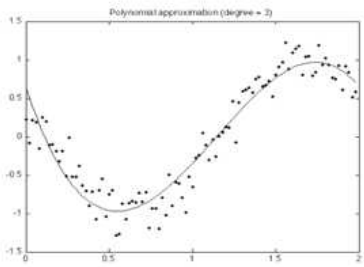
“ 경사하강법을 이용한 근사함수 ”

과 목 명	지능 시스템
담당교수	조영완 교수님
학 과	컴퓨터공학과
학 번	2016305078
이 름	최영환
제 출 일	2021.04.16

- 다음 그림과 같이 주어진 데이터에 대해 오차를 최소화하는 근사함수(3차 다항함수, 파라미터 (a, b, c, d))를 경사하강법으로 구하는 예제를 프로그램을 작성하여 실습하고 결과를 정리하여(학습의 중간 과정 포함, 예를 들어, 10 epoch학습 결과, 100 epoch학습 결과 등등) 보고서 형태로 제출한다.
- 데이터 쌍 (x_i, y_i)은 300개를 다음과 같이 생성하여 사용한다.
- $[0, 3]$ 구간의 x_i를 랜덤하게 발생시켜 $y = x^3 - 4.5x^2 + 6x + 2$의 함수에 대입하여 y를 구하고 여기에 $[-0.5, 0.5]$ 사이의 값을 랜덤하게 생성하여 더한 값을 y_i로 한다. (4월 16일까지 Google Classroom 제출)



(x_i, y_i) for $i = 1, 2, \cdots, N$



$y = ax^3 + bx^2 + cx + d$

- 경사 하강법 함수의 구현은 아래 코드를 참고하여 구현하였음.

매개변수 학습의 개념 – 예제 구현 코드

```
def linear_regression (X, y, m_current=0, b_current=0, epochs=1000,
learning_rate=0.0001):
        N = float(len(y))
        for i in range(epochs):
                y_current = (m_current * X) + b_current
                cost = sum([data**2 for data in (y-y_current)]) / (2*N)
                m_gradient = -(1/N) * sum(X * (y - y_current))
                b_gradient = -(1/N) * sum(y - y_current)
                m_current = m_current - (learning_rate * m_gradient)
                b_current = b_current - (learning_rate * b_gradient)
        return m_current, b_current, cost
```

< 문제의 조건에 맞는 데이터 쌍 x, y 생성 >

```python
# 문제의 조건에 맞는 랜덤한 데이터 생성
def initializing(num_of_data = 300):
    X = []
    Y = []

    for i in range(num_of_data):
        # x 는 [0, 3] 구간의 랜덤한 실수
        x = random.uniform(0, 3)
        # y = x^3 - 4.5x^2 + 6x + 2 의 함수에 랜덤한 실수 x 값 대입 후
        # [-0.5, 0.5] 사이의 값을 랜덤하게 생성하여 더함
        y = x**3 - (4.5 * x**2) + (6 * x) + 2 + random.uniform(-0.5, 0.5)
        X.append(x)
        Y.append(y)

    return X, Y
```

▶ 문제의 조건에 맞게 총 300개의 x와 y 값을 생성하는 함수.

▶ random.uniform() 함수를 사용하여 [0, 3] 구간의 임의의 실수를 생성하고, 이를 x의 값으로 설정하였음.

▶ $y = x^3 - 4.5x^2 + 6x + 2$ 에 x의 값을 대입하여 구한 값에 [-0.5 ,0.5] 구간의 임의의 실수를 더하여 y의 값으로 설정하였음.

▶ x와 y의 값은 총 300개로, 이들 값을 리스트 X와 Y에 넣어서 사용하였음.

< 생성 결과 >

```
1: 0.6337919537437542, 3.8830275443063837
2: 0.2251884392372151, 2.677377470188543
3: 0.24114522550189788, 2.7600603539717845
4: 2.5413749742844085, 4.480060358596582
5: 2.822480603058248, 5.4757479436586625
6: 0.7360800061389438, 4.335225647817131
7: 2.7505530713557653, 5.2144740302671
8: 1.9671229627812516, 3.564151733193983
9: 0.9351873008289305, 4.512330015500485
10: 0.12627905133228923, 2.779364542689903
11: 0.9303939714827091, 4.41203584017899
12: 0.5548793420548287, 3.9689937650519704
13: 2.4007419204339024, 4.320903722895641
14: 1.6964346500746925, 4.511165562599788
15: 1.8811501781565438, 3.6051151412134734
286: 2.7249953774832054, 5.403646745124464
287: 2.0423649657708234, 4.27496092862718
288: 2.1425772477778526, 3.907009177422373
289: 0.11758531739945832, 2.8160156356833785
290: 1.0465706402815491, 4.040730023756033
291: 2.5064449088847676, 4.954904305003511
292: 0.19757945522483045, 3.331888859261083
293: 0.9379082460157036, 4.073471800362855
294: 0.27815797492033645, 2.98880702622753
295: 0.6286409626585139, 3.812113908832612
296: 0.6368599908618403, 4.660553407001003
297: 2.784852742073384, 5.7729766141124985
298: 1.891266346713851, 3.710627964971295
299: 1.144872428404593, 4.16650573324087
300: 1.558928542319029, 4.005805718148021
```

▶ 각 번호에서 좌측이 x의 값이며, 우측이 y의 값임.

▶ 문제에서 요구된 범위 내에서 x의 값이 생성되었음을 확인하였음.

▶ 출력문의 길이가 길어, 일부를 제외하고는 생략하였음.

< 경사 하강법 함수 line_regression 함수 구현 >

```python
# y = ax^3 + bx^2 + cx + d
# 데이터 개수 300개, 에포크 1000번, learning rate 는 학습률(스텝사이즈)
def linear_regression(X, Y, num_of_data = 300, epochs = 1000, learning_rate = 0.0001):
    a_current = 0       # 매개변수 a 의 현재값
    b_current = 0       # 매개변수 b 의 현재값
    c_current = 0       # 매개변수 c 의 현재값
    d_current = 0       # 매개변수 d 의 현재값

    # 데이터 개수 300개
    N = num_of_data
    for cnt in range(epochs):
        cost = 0            # 비용함수(cost function) J 값(평균제곱오차), 최소화 하는 것이 목적
        a_gradient = 0      # 매개변수 a 편미분 값
        b_gradient = 0      # 매개변수 b 편미분 값
        c_gradient = 0      # 매개변수 c 편미분 값
        d_gradient = 0      # 매개변수 d 편미분 값
        y_current = []      # i번째 x일 때 예측값 y_current = ax^3 + bx^2 + cx + d
        for i in range(N):
            # i번째 x일 때 예측값 y_current = ax^3 + bx^2 + cx + d
            y_current.append((a_current * X[i] ** 3) + (b_current * X[i] ** 2) + (c_current * X[i]) + d_current)

            a_gradient += (X[i] ** 3) * (Y[i] - y_current[i])   # a 편미분 1단계
            b_gradient += (X[i] ** 2) * (Y[i] - y_current[i])   # b 편미분 1단계
            c_gradient += X[i] * (Y[i] - y_current[i])          # c 편미분 1단계
            d_gradient += Y[i] - y_current[i]                   # d 편미분 1단계
            cost += (Y[i] - y_current[i]) ** 2  # 비용함수(cost function) J 값(평균제곱오차), 최소화 하는 것이 목적

        cost /= (2 * float(N))                          # 비용함수(cost function) J 값(평균제곱오차), 최소화 하는 것이 목적
        a_gradient = -(1 / float(N)) * a_gradient                   # a 편미분 2단계
        b_gradient = -(1 / float(N)) * b_gradient                   # b 편미분 2단계
        c_gradient = -(1 / float(N)) * c_gradient                   # c 편미분 2단계
        d_gradient = -(1 / float(N)) * d_gradient                   # d 편미분 2단계

        a_current = a_current - (learning_rate * a_gradient)    # a 값, 학습률에 따라 경사하강법
        b_current = b_current - (learning_rate * b_gradient)    # b 값, 학습률에 따라 경사하강법
        c_current = c_current - (learning_rate * c_gradient)    # c 값, 학습률에 따라 경사하강법
        d_current = d_current - (learning_rate * d_gradient)    # d 값, 학습률에 따라 경사하강법

        return a_current, b_current, c_current, d_current, cost
```

▶ ppt의 예제 구현 코드를 참고하였음. 예제 코드는 매개변수가 2개인, 1차 다항함수인 경우의 경사 하강법 적용이므로, 이를 매개변수가 4개인 3차 다항함수에 맞춰 수정하였음.

▶ y_current 는 $y = ax^3 + bx^2 + cx + d$ 식을 적용하여 계산된 각 데이터 별 y의 예측값임.

▶ 각 매개변수의 편미분 식

a_gradient = x^3 * (y 실제값 – y 예측값)　　b_gradient = x^2 * (y 실제값 – y 예측값)

c_gradient = x * (y 실제값 – y 예측값)　　d_gradient = (y 실제값 – y 예측값)

위 계산 과정을 N(데이터의 개수)회 진행한 뒤, 각 값에 1/N을 곱하였음.

여기서 (y 실제값 – y 예측값)은 오차를 뜻함.

자기 자신의 값에서 자기 자신과 학습률(0.0001)을 곱한 값을 빼주었음.

각 매개변수에 대해 a_current = a_current - (learning_rate * a_gradient) 처럼 계산하였음.

▶ 비용함수(cost function)는 변수 cost로 설정하였음.

cost의 값에 오차의 제곱 ((y 실제값 – y 예측값)^2)을 N회 만큼 더한 뒤, 2/N을 곱하였음.

▶ 매개변수 a, b, c, d와 비용함수 cost의 값을 반환함.

```python
# 에포크 10 마다 근사함수 값 출력
if (cnt+1) % 10 == 0:
    print(f'================ Epoch : {cnt+1} ================')
    print(f'y = {a_current:.4f}x^3 + {b_current:.4f}x^2 + {c_current:.4f}x + {d_current:.4f}')
    print(f'cost = {cost}')
```

▶ 추가로, 에포크 10마다의 변화 관측을 위해 출력문을 함수 마지막 부분에 추가하였음.

< 결과 확인 >

```python
num_of_data = 300
X, Y = initializing(num_of_data)

# 생성한 데이터 출력 (실제값)
for i in range(len(X)):
    print(f'{i+1}: {X[i]}, {Y[i]}')
print("============================== End of Initializing ==============================")
print()
print()

# 경사하강법 시행
a, b, c, d, cost = linear_regression(X, Y, num_of_data=num_of_data, epochs=1000, learning_rate=0.0001)

# 생성한 데이터 그래프 출력
plt.subplot(1, 3, 1)
plt.title("Data")
plt.plot(X, Y, 'r.')

# 근사함수 그래프 출력
plt.subplot(1, 3, 2)
plt.title("Graph of Function Approximation")
# x_range = np.linspace(0, 3, 300)
x_range = np.array(X)
y_range = np.array([a*x**3 + b*x**2 + c*x + d for x in x_range])
plt.plot(x_range, y_range, 'b')

# 생성한 데이터 그래프와 근사함수 그래프 동시 출력
plt.subplot(1, 3, 3)
plt.title("Graph of Data & Function Approximation")
plt.plot(X, Y, 'r.', label='Data')
plt.plot(x_range, y_range, 'b', label='Function Approximation')
plt.legend()

plt.show()
```

▶ num_of_data는 데이터의 개수이므로, 300으로 초기화 하였음.

▶ X 와 Y는 각각 x와 y의 실제값으로, initializing 함수를 사용하여 초기화함.

▶ 생성된 데이터의 각 값을 모두 출력해준 뒤, 경사 하강법을 시행하여 매개변수들의 값과 비용함수의 값을 구함. (linear_regression 함수 사용)

▶ 그 다음, 생성된 데이터의 그래프와 근사함수의 그래프를 출력하여 결과를 확인하였음.

▶ 에포크가 1000인 경우, 10000인 경우를 관측해보았음.

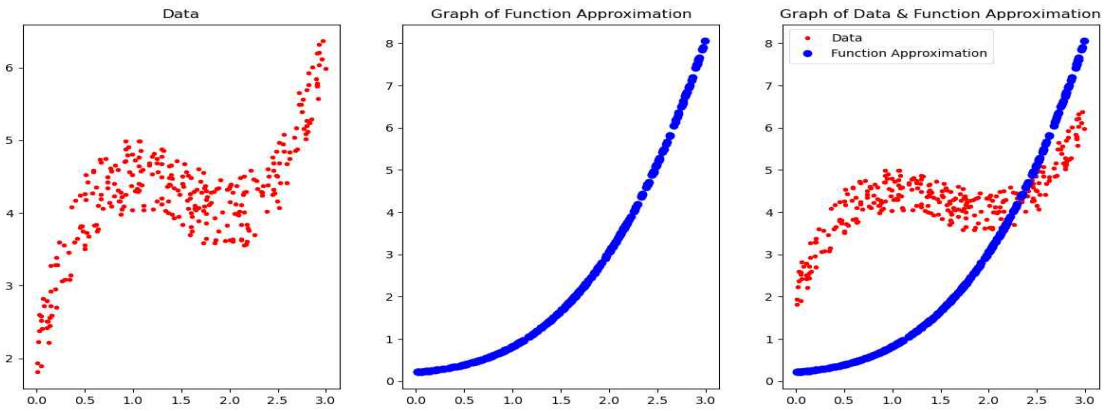
< 에포크 1000인 경우의 결과 관측 >

```
================= Epoch : 10 =================
y = 0.0294x^3 + 0.0131x^2 + 0.0065x + 0.0041
cost = 8.310497801776194
================= Epoch : 20 =================
y = 0.0556x^3 + 0.0248x^2 + 0.0125x + 0.0080
cost = 7.421703198712656
================= Epoch : 30 =================
y = 0.0788x^3 + 0.0354x^2 + 0.0180x + 0.0117
cost = 6.713859656752421
================= Epoch : 40 =================
y = 0.0994x^3 + 0.0450x^2 + 0.0230x + 0.0152
cost = 6.149770212574142
================= Epoch : 50 =================
y = 0.1177x^3 + 0.0536x^2 + 0.0277x + 0.0186
cost = 5.699885145663554
```

```
================= Epoch : 960 =================
y = 0.2025x^3 + 0.2101x^2 + 0.1919x + 0.2046
cost = 3.289877187204555
================= Epoch : 970 =================
y = 0.2018x^3 + 0.2110x^2 + 0.1933x + 0.2064
cost = 3.282982882159778
================= Epoch : 980 =================
y = 0.2011x^3 + 0.2120x^2 + 0.1947x + 0.2082
cost = 3.276105724125696
================= Epoch : 990 =================
y = 0.2003x^3 + 0.2130x^2 + 0.1962x + 0.2101
cost = 3.2692456698430035
================= Epoch : 1000 =================
y = 0.1996x^3 + 0.2139x^2 + 0.1976x + 0.2119
cost = 3.2624026761640863
```

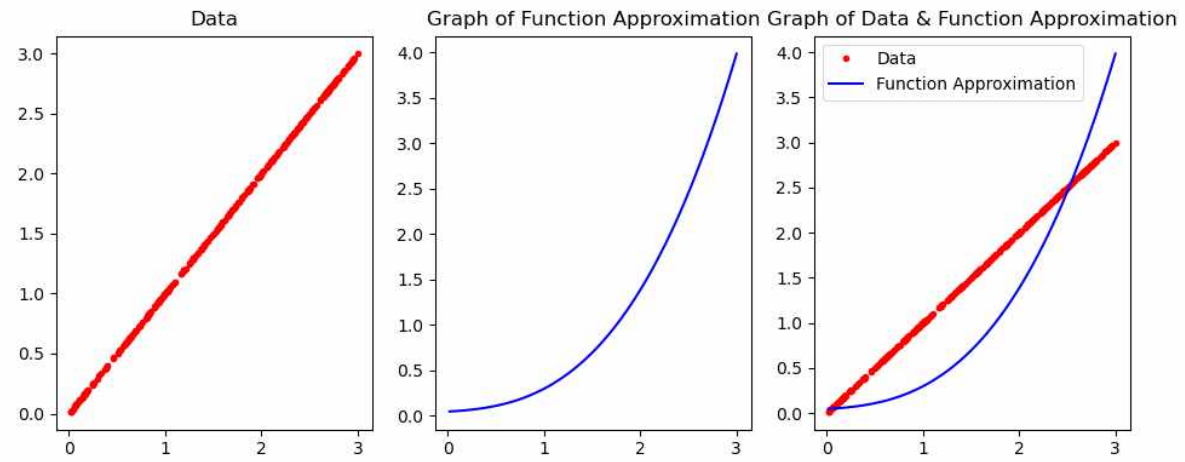
- 에포크가 1000인 경우의 결과.
- 에포크가 거듭될수록, 매개변수들의 값과 비용함수 cost의 값이 변화함을 관측하였음. 출력문이 길어 일부를 제외하고는 생략하였음.
- 에포크 10마다의 출력을 관측하였으며, 초기 cost의 값은 8을 넘는 값이었으나, 에포크가 1000에 가까워지면서 3에 수렴하는 것을 확인하였음.

< 에포크 1000인 경우의 그래프 >



- 실제 데이터의 그래프(좌), 근사함수의 그래프(중), 데이터와 근사함수의 그래프(우) 두 그래프의 차이가 큰 것 같아, 초기화 단계에서 실제 데이터를 오름차순으로 정렬한 뒤의 결과를 관측해보기로 하였음.
- 근사함수의 그래프의 값은 오름차순으로 정렬된 값이기 때문으로 생각하여, x의 값을 오름차순으로 정렬한 뒤 해보기로 하였음.

< 에포크 1000인 경우의 그래프(정렬 후) >



▶ 여전히 두 그래프의 차이가 있으나, 이전과 비교했을 때, 개선되었음을 확인.
(y축의 최댓값이 8 이상이었으나, 정렬 이후에는 5 미만.)

▶ 이후의 결과들은 모두 데이터가 오름차순으로 정렬되어있는 상태의 결과들임.

< 에포크 10000인 경우의 결과 관측(정렬 전) >

```
================== Epoch : 9950 ==================
y = -0.1444x^3 + 0.6089x^2 + 0.8952x + 1.2740
cost = 0.8139209325141684
================== Epoch : 9960 ==================
y = -0.1446x^3 + 0.6091x^2 + 0.8956x + 1.2748
cost = 0.8131061433350694
================== Epoch : 9970 ==================
y = -0.1448x^3 + 0.6093x^2 + 0.8960x + 1.2756
cost = 0.8122930886914441
================== Epoch : 9980 ==================
y = -0.1450x^3 + 0.6095x^2 + 0.8964x + 1.2763
cost = 0.8114817644954758
================== Epoch : 9990 ==================
y = -0.1452x^3 + 0.6096x^2 + 0.8969x + 1.2771
cost = 0.8106721666691026
================== Epoch : 10000 ==================
y = -0.1453x^3 + 0.6098x^2 + 0.8973x + 1.2778
cost = 0.8098642911440002
```

▶ 예상한 것과 같이 에포크가 증가할수록, cost의 값이 매우 작아졌음을 확인하였음.

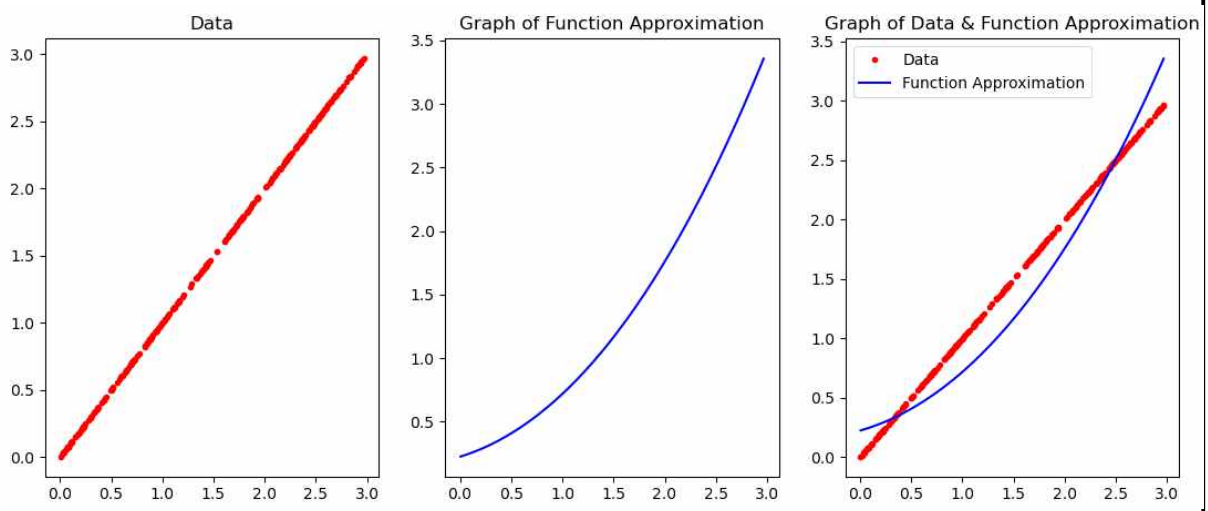
< 에포크 10000인 경우의 결과 관측(정렬 후) >

```
================= Epoch : 9960 =================
y = 0.0133x^3 + 0.2332x^2 + 0.2469x + 0.2232
cost = 0.024690531355583946
================= Epoch : 9970 =================
y = 0.0133x^3 + 0.2333x^2 + 0.2470x + 0.2233
cost = 0.02464675894684951
================= Epoch : 9980 =================
y = 0.0132x^3 + 0.2334x^2 + 0.2471x + 0.2234
cost = 0.024603081889012005
================= Epoch : 9990 =================
y = 0.0131x^3 + 0.2335x^2 + 0.2472x + 0.2235
cost = 0.024559499970072608
================= Epoch : 10000 =================
y = 0.0131x^3 + 0.2336x^2 + 0.2474x + 0.2236
cost = 0.02451601297850503
```

```
================= Epoch : 1000 =================
y = 0.1161x^3 + 0.0806x^2 + 0.0563x + 0.0442
cost = 0.14799951781106532
```

▶ 정렬 이후, cost의 값이 이전과 비교했을 때, 굉장히 많이 줄어들었음을 확인하였음.
▶ 에포크 1000에서의 결과도 확인해보았더니, 이전 값과 굉장히 크게 다름을 확인하였음.
▶ 정렬을 한 것이 cost의 결과를 크게 바꾼 것으로 예상됨.

< 에포크 10000인 경우의 그래프 >



▶ 실제 데이터의 그래프와의 오차가 크게 줄어들었음을 확인하였음.
(y축의 최댓값이 4 미만 3.5 이상임을 확인.)

< 결론 >

에포크가 거듭함에 따라, 그리고 비용함수의 값이 감소함에 따라, 오차 (데이터의 실제 값과 근사함수를 통한 예측값의 차이) 가 줄어들었음을 확인하였음.