수치해석 최적화 알고리즘 프로젝트

확률적 경사하강법 SGD(Stochastic Gradient Descent)

국민대 소프트웨어학부 수치해석 강의를 통하여 배운 내용을 바탕으로 최적화 알고리즘 중 확률적 경사하강법(SGD)를 구현해 보았습니다.

> 소프트웨어학부 20191556 김민정

목차

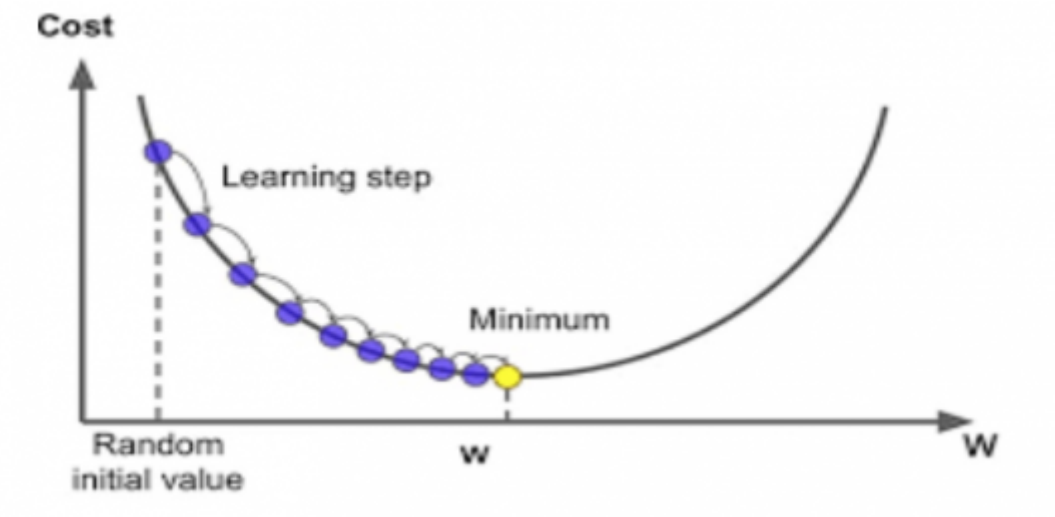
---

1. 최적화 알고리즘 개요 및 동작원리 - 확률적 경사하강법

확률적 경사하강법을 알기 위해서 잠시 경사하강법에 대해 미리 설명하겠습니다.

경사하강법

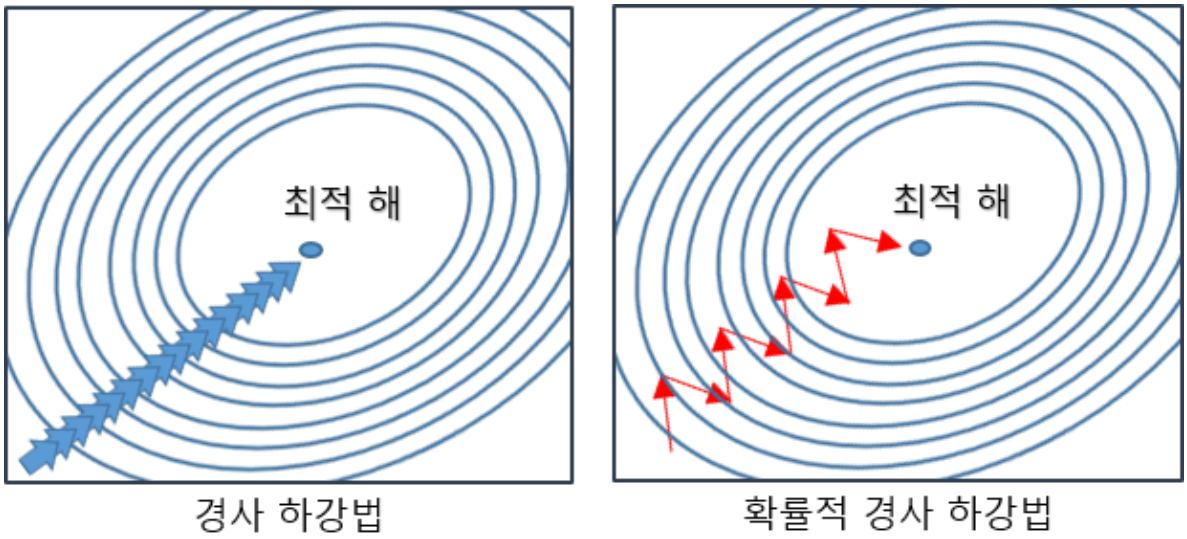
경사하강법이란 함수의 기울기(경사)를 낮은쪽으로 계속 이동시켜 극값에 이를때까지 반복시키는 것을 말합니다. 제시된 함수의 기울기로 최소값을 찾아내는 머신러닝 알고리즘입니다. 여기서 제시된 함수란 비용함수(cost function)을 말합니다. 이 비용함수를 최소화 하기 위해 매개변수를 반복적으로 조정하는 과정이라고도 할 수 있겠습니다. 확률적 경사하강법은 학습을 통해 모델의 최적 파라미터(최소값)을 찾는것이 목표입니다.



경사하강법은 정확하게 가중치를 찾아가지만 가중치를 변경할때마다 전체 데이터에 대해 미분해야 하므로 계산량이 매우 많습니다. 이러한 경사하강법의 단점을 보완한 알고리즘 중 하나가 바로 확률적 경사하강법입니다.

확률적 경사하강법

확률적 경사하강법(Stodchastic Gradient Descent)은 파라미터를 업데이트 할 때, 무작위로 샘플링 된 학습 데이터를 이용하여 비용함수(cost function)의 기울기를 계산합니다. 이 방법은 모델을 훨씬 더 자주 업데이트하며, 성능 개선 정도를 빠르게 확인 할 수 있습니다. 그러나 경사하강법보다 노이즈가 심하고 최적해의 정확도가 떨어집니다.



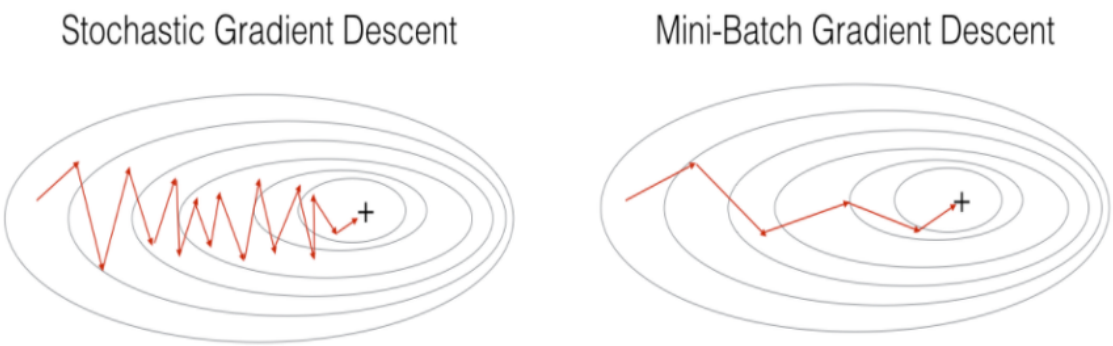
경사 하강법 확률적 경사 하강법

수식은 다음과 같습니다.

$$W(t+1) = W(t) - \eta \frac{\partial}{\partial w} Cost(w)$$

여기서 Cost(w)에 사용되는 입력 데이터는 확률적으로 샘플링 된 학습 데이터가 사용됩니다. 다음 수식에서 η는 학습률(Learning Rate)를 의미합니다.

확률적 경사하강법은 전체 데이터가 아니라 랜덤하게 추출한 일부데이터를 사용하기 때문에 불안정하고 오차율이 큽니다. 이러한 단점을 보완하기 위한 방법이 Mini batch를 이용한 방법입니다.



노이즈가 많이 줄어든 모습을 볼 수 있습니다.

---

2. 확률적 경사하강법(SGD)의 동작코드와 단위테스트

2.1. 확률적 경사하강법(SGD) 동작코드

tensorflow를 이용하였고, 모든 코드는 선형회귀에 이용되는것을 전제로 구현하였습니다.

전체코드

```
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

class SGD:
    def __init__(self):
        #학습 파라미터 설정
        self.learning_rate = 0.01
        self.training_steps = 1000
        self.display_step = 50
        #학습에 사용될 W와 b 초기값 생성
        self.W = tf.Variable(np.random.randn(), name="weight")
        self.b = tf.Variable(np.random.randn(), name="bias")
        #Training data 생성
        self.X, self.Y = make_regression(n_samples=100, n_features=1, bias=10.0,
                                         noise=15.0, random_state=2)
        self.Y = np.expand_dims(self.Y, axis=1)
        #X의 차원
        self.n_samples = self.X.shape[0]
        #Stochastic Gradient Descent
        self.optimizer = tf.optimizers.SGD(self.learning_rate)

    def linear_regression(self, x):
        return self.W * x + self.b

    def mean_square(self, y_pred, y_true):
        return tf.reduce_sum(tf.pow(y_pred-y_true, 2)) / (2 * self.n_samples)

    def run_optimization(self):
        with tf.GradientTape() as g:
            pred = self.linear_regression(self.X)
```

```
            loss = self.mean_square(pred, self.Y)

        gradients = g.gradient(loss, [self.W, self.b])

        self.optimizer.apply_gradients(zip(gradients, [self.W, self.b]))

    def training(self):
        for step in range(1, self.training_steps + 1):
            self.run_optimization()

            if step % self.display_step == 0:
                self.pred = self.linear_regression(self.X)
                self.loss = self.mean_square(self.pred, self.Y)
                print("step:%i, loss:%f, W:%f, b:%f" % (step, self.loss, self.W,
self.b))

#단위테스트
import unittest
class TestSgd(unittest.TestCase):
    def test_sgd(self):
        testsgd = SGD()
        testsgd.training()
        W, b = testsgd.W, testsgd.b
        loss = testsgd.loss
        self.assertAlmostEqual(float(loss), 98.185654, places=2)
        self.assertAlmostEqual(float(W), 58.797782, places=2)
        self.assertAlmostEqual(float(b), 10.393133, places=2)

if __name__ == "__main__":
    #단위테스트
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
    print("------------------------------------------------------------------------
-")

    #학습
    sgd = SGD()
    sgd.training()

    #결과 확인
    x = np.arange(-3, 3)
    y = sgd.W*x + sgd.b
    plt.scatter(sgd.X, sgd.Y, label="Original data")
    plt.plot(x, y, linewidth=3, color='red', label='Fitted line')
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

코드 부분별로 살펴보겠습니다.

```
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
```

필요한 패키지를 import 합니다. sklearn은 가상 데이터를 생성하기 위해 패키지를 import하였습니다. 주요 알고리즘은 tensorflow를 사용하였습니다.

```
class SGD:
    def __init__(self):
        #학습 파라미터 설정
        self.learning_rate = 0.01
        self.training_steps = 1000
        self.display_step = 50
        #학습에 사용될 W와 b 초기값 생성
        self.W = tf.Variable(np.random.randn(), name="weight")
        self.b = tf.Variable(np.random.randn(), name="bias")
        #Training data 생성
        self.X, self.Y = make_regression(n_samples=100, n_features=1, bias=10.0,
                                         noise=15.0, random_state=2)
        self.Y = np.expand_dims(self.Y, axis=1)
        #X의 차원
        self.n_samples = self.X.shape[0]
        #Stochastic Gradient Descent
        self.optimizer = tf.optimizers.SGD(self.learning_rate)
```

학습에 사용될 파라미터와 초기값, 데이터들을 생성하고 optimizer로 SGD를 지정합니다.

```
def linear_regression(self, x):
    return self.W * x + self.b
```

선형 함수를 정의합니다. 수식으로 표현하면 다음과 같습니다.

$$f(x) = Wx + b$$

```
def mean_square(self, y_pred, y_true):
    return tf.reduce_sum(tf.pow(y_pred-y_true, 2)) / (2 * self.n_samples)
```

손실함수인 평균 제곱 오차 함수(MSE)를 정의합니다. 수식은 다음과 같습니다.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - t_i)^2$$

```
def run_optimization(self):
    with tf.GradientTape() as g:
        pred = self.linear_regression(self.X)
        loss = self.mean_square(pred, self.Y)

    gradients = g.gradient(loss, [self.W, self.b])

    self.optimizer.apply_gradients(zip(gradients, [self.W, self.b]))
```

주어진 입력 변수에 대해 gradient를 계산하기 위해 `tf.GradientTape` 함수를 사용하여 Gradient를 구하였습니다.

구해진 gradient에 따라 W와 b 값을 변화시킵니다.

```
def training(self):
    for step in range(1, self.training_steps + 1):

        self.run_optimization()

        if step % self.display_step == 0:
            self.pred = self.linear_regression(self.X)
            self.loss = self.mean_square(self.pred, self.Y)
            print("step:%i, loss:%f, W:%f, b:%f" % (step, self.loss, self.W,
self.b))
```

앞서 구현한 `run_optimizer` 함수를 이용하여 학습을 시작합니다. W와 b를 업데이트 하기 위해 sgd 과정을 반복하고, 반복될 때 일정 step이 지나면 loss, W, b 값을 출력하도록 하였습니다.

2.2. 단위테스트

`make_regression` 함수를 통해 생성된 랜덤 데이터의 W(weight)값은 58.797782 에 근사하고, b(bias)값은 10.391333에 근사합니다. loss값은 98.185654에 근사합니다.

이를 토대로 단위테스트용 코드를 작성해 보았습니다.

단위테스트 클래스

```
import unittest
class TestSgd(unittest.TestCase):
    def test_sgd(self):
        testsgd = SGD()
        testsgd.training()
        W, b = testsgd.W, testsgd.b
        loss = testsgd.loss
        self.assertAlmostEqual(float(loss), 98.185654, places=2)
        self.assertAlmostEqual(float(W), 58.797782, places=2)
        self.assertAlmostEqual(float(b), 10.393133, places=2)
```

실행

```
if __name__ == "__main__":
    #단위테스트
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
    print("------------------------------------------------------------------------
-")
```

단위테스트 결과

```
----------------------------------------------------------------------
Ran 1 test in 4.523s

OK
----------------------------------------------------------------------
```

단위테스트가 성공한 것을 확인 할 수 있습니다. 실제로 코드를 실행하면 step에 따른 loss, W, b 값을 확인할 수 있으나 여기에선 생략하도록 하겠습니다.

2.3. 그래프로 보는 실행결과

작성한 코드를 토대로 그래프를 그려보겠습니다.

실행 코드

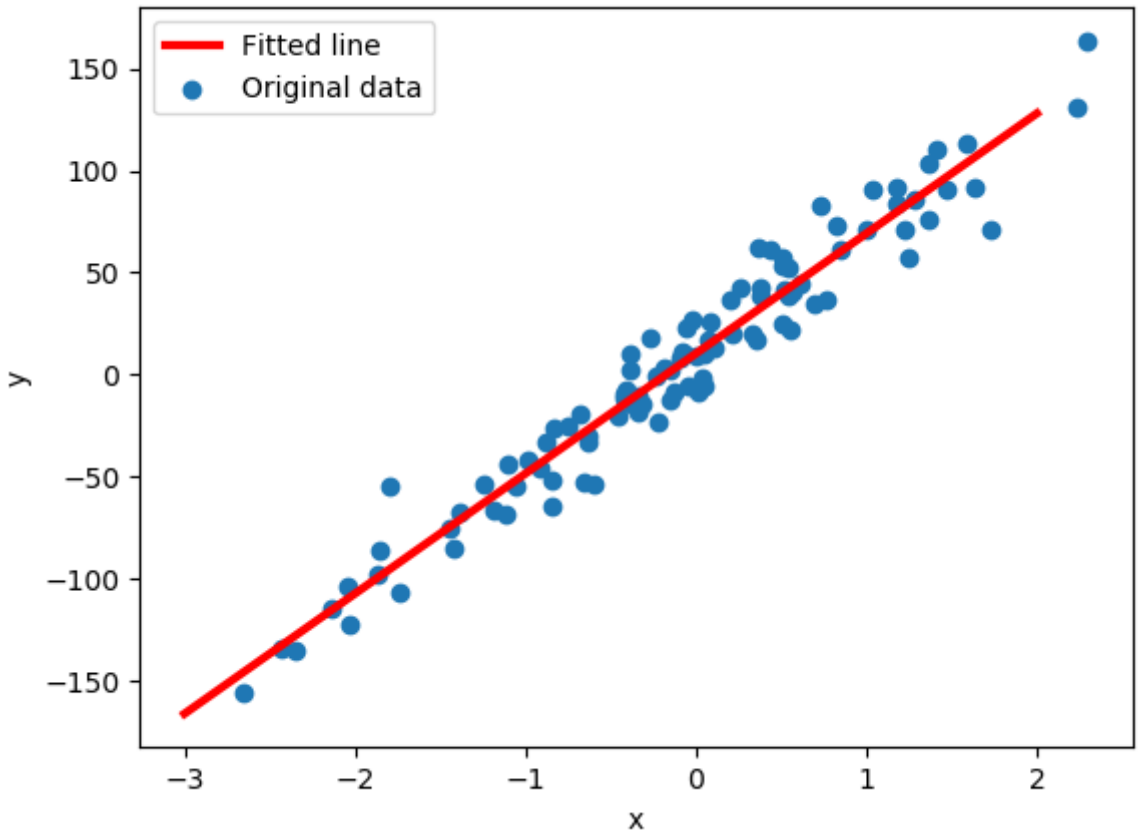
```
if __name__ == "__main__":
    #학습
    sgd = SGD()
    sgd.training()

    #결과 확인
    x = np.arange(-3, 3)
    y = sgd.W*x + sgd.b
    plt.scatter(sgd.X, sgd.Y, label="Original data")
    plt.plot(x, y, linewidth=3, color='red', label='Fitted line')
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

출력 결과

```
step:50, loss:779.864075, W:23.414915, b:2.632110
step:100, loss:336.703735, W:38.035130, b:4.595627
step:150, loss:182.720993, W:46.575958, b:6.239035
step:200, loss:128.524841, W:51.580765, b:7.499780
step:250, loss:109.205284, W:54.522743, b:8.418927
step:300, loss:102.232986, W:56.257652, b:9.067093
```

```
step:350, loss:99.687363, w:57.284035, b:9.513615
step:400, loss:98.747902, w:57.893219, b:9.815985
step:450, loss:98.397835, w:58.255955, b:10.018076
step:500, loss:98.266289, w:58.472614, b:10.151767
step:550, loss:98.216454, w:58.602440, b:10.239487
step:600, loss:98.197487, w:58.680447, b:10.296657
step:650, loss:98.190216, w:58.727455, b:10.333709
step:700, loss:98.187408, w:58.755886, b:10.357618
step:750, loss:98.186325, w:58.773109, b:10.372987
step:800, loss:98.185928, w:58.783581, b:10.382830
step:850, loss:98.185753, w:58.789963, b:10.389120
step:900, loss:98.185692, w:58.793865, b:10.393126
step:950, loss:98.185661, w:58.796249, b:10.395674
step:1000, loss:98.185654, w:58.797707, b:10.397291
```

그래프



안정적으로 피팅된 모습을 확인할 수 있습니다.

---

3. 확률적 경사하강법(SGD) 구체화

3번에서는 numpy level에서 확률적 경사하강법을 구현해보도록 하겠습니다.

선형회귀에 사용된다는 전제로 구현하였습니다.

전체코드

```
import numpy as np
import matplotlib.pyplot as plt
import random

class SGD:
    def predict(self, alpha, beta, x_i):
        '''
        해당 데이터로 linear function의 값 예측
        '''
        return beta * x_i + alpha

    def error(self, alpha, beta, x_i, y_i):
        '''
        오차함수
        '''
        return y_i - self.predict(alpha, beta, x_i)

    def squared_error(self, x_i, y_i, delta):
        '''
        제곱 오차 함수
        '''
        alpha, beta = delta
        return self.error(alpha, beta, x_i, y_i)**2

    def partial_difference_quotient(self, f, v, i, h):
        '''
        각각 다른 변수들은 고정, 하나의 변수만 h만큼 이동했을때의 변화율 구함
        h는 매우 작은 수

        v와 v에서 i만 아주 약간의 차이가 있는 w 두개의 리스트를
        각각 f(x)에 대입했을때의 차이를 h로 나눈 값 반환
        '''
        w = [v_j + (h if j == i else 0)
            for j, v_j in enumerate(v)]
        return (f(w) - f(v)) / h

    def estimate_gradient(self, f, v, h=0.00001):
        '''
        여러 변수의 각각의 편미분 리스트 반환
        '''
        return [self.partial_difference_quotient(f, v, i, h)
                for i, _ in enumerate(v)]

    def get_gradient(self, x_i, y_i, coeff):
        '''
        손실함수인 제곱 오차 함수에 대해 편미분을 구함
        '''
        return self.estimate_gradient(lambda coeff:self.squared_error(x_i,
y_i,coeff),
                                      coeff, h=1e-4)

    def vector_subtract(self, v, w):
        '''
        배열 차
        '''
```

```
        return [v_i - w_i for v_i, w_i in zip(v, w)]

    def scalar_multiply(self, c, v):
        '''
        스칼라배
        '''
        return [c * v_i for v_i in v]


    def stochastic_gradient_descent(self, x, y, delta, learning_rate_0=0.01):
        '''
        확률적 경사하강법 구현
        x, y = 입력 데이터
        delta = α, β 초기값

        100회동안 변화 없이 반복될경우 함수 종료됨
        '''
        min_delta, min_value = None, float("inf")
        no_improvement_cnt = 0
        inf_loop_cnt = 0
        learning_rate = learning_rate_0

        '''사용할 함수 정의'''
        target_fn = self.squared_error
        gradient_fn = self.get_gradient

        while((no_improvement_cnt < 100)):
            inf_loop_cnt += 1

            value = sum(target_fn(x_i, y_i, delta) for x_i, y_i in zip(x, y))

            if value < min_value:
                '''
                새로운 최소값을 찾으면 업데이트하고
                반복 횟수, step size 초기화
                '''
                min_delta, min_value = delta, value
                if inf_loop_cnt%20 == 1:
                    print('min delta update : ', min_delta)
                no_improvement_cnt = 0
                learning_rate = learning_rate_0
            else:
                '''
                최소값이 찾아지지 않으면 반복회수를 늘리고
                step size 축소
                '''
                no_improvement_cnt += 1
                if (no_improvement_cnt%20 == 5):
                    print("no improvement cnt : ", no_improvement_cnt)
                learning_rate *= 0.9

            '''
            delta 업데이트
            '''
            idxs = [i for i in range(len(x))]
            random.shuffle(idxs)

            batch_size = 20
```

```
            for i in idxs[:batch_size]:
                '''랜덤으로 뽑아 업데이트'''
                gradient_i = gradient_fn(x[i], y[i], delta)
                delta = self.vector_subtract(delta,
self.scalar_multiply(learning_rate, gradient_i))

        return min_delta
```

3.1. 확률적 경사하강법(SGD) 모듈 함수

모듈 내에서 확률적 경사하강법 구현을 위해 미리 구현한 함수들을 설명하겠습니다.

```
import numpy as np
import matplotlib.pyplot as plt
import random
```

필요한 패키지를 import 합니다.

```
class SGD:
    def predict(self, alpha, beta, x_i):
        return beta * x_i + alpha
```

입력된 alpha, beta값을 통하여 linear function의 값을 예측하여 나타내는 함수입니다. 수식으로 표현하면 아래와 같습니다. 여기서 beta값이 Weight, alpha값이 bias 입니다.

$$f(x_i) = \beta x_i + \alpha$$

```
def error(self, alpha, beta, x_i, y_i):
    return y_i - self.predict(alpha, beta, x_i)
```

위에서 구현한 `predict` 함수를 이용하여 오차를 구하여 반환합니다. 수식으로 표현하면 아래와 같습니다.

$$Error(x_i) = y_i - (\beta x_i + \alpha)$$

```
def squared_error(self, x_i, y_i, delta):
    alpha, beta = delta
    return self.error(alpha, beta, x_i, y_i)**2
```

제곱 오차 함수입니다.

```
def partial_difference_quotient(self, f, v, i, h):
    w = [v_j + (h if j == i else 0)
        for j, v_j in enumerate(v)]
    return (f(w) - f(v)) / h

def estimate_gradient(self, f, v, h=0.00001):
    return [self.partial_difference_quotient(f, v, i, h)
            for i, _ in enumerate(v)]

def get_gradient(self, x_i, y_i, coeff):
    return self.estimate_gradient(lambda coeff:self.squared_error(x_i,
y_i,coeff),
                                  coeff, h=1e-4)
```

다음 세 함수는 함께 설명하겠습니다.

`get_gradient` 함수는 편미분을 구하기 위해 사용되는 함수입니다. 이 코드에서는 제곱오차함수의 편미분 값을 구하도록 설정되었습니다. 편미분값을 직접 설정하여 구현할 수 있었지만, 코드의 범용성을 위하여 각각 함수에 대해 편미분 값을 구할 수 있도록 설정하였습니다.

`estimate_gradient` 함수는 함수의 여러 변수에 대하여 각각에 대해 편미분을 한 리스트를 반환합니다. 예를 들어 위에서 서술한 것 처럼 linear function의 경우 β와 α, 두가지 변수가 있으므로 각각에 대한 편미분이 담긴 리스트를 반환합니다.

`partial_difference_quotient` 함수는 다른 변수들은 고정되고, 하나의 변수만 아주 약간 이동했을 때의 변화율을 구하여 차이가 생긴 두개의 리스트의 차를 구하여 나눈 값을 반환합니다. 도함수를 도출해내는 방식과 유사하다고 할 수 있겠습니다.

```
def vector_subtract(self, v, w):
    return [v_i - w_i for v_i, w_i in zip(v, w)]

def scalar_multiply(self, c, v):
    return [c * v_i for v_i in v]
```

다음 두 함수는 벡터 차와 벡터 스칼라배를 구현한 함수입니다.

3.2. 확률적 경사하강법(SGD) 구현

```
def stochastic_gradient_descent(self, x, y, delta, learning_rate_0=0.01):
    min_delta, min_value = None, float("inf")
    no_improvement_cnt = 0
    inf_loop_cnt = 0
    learning_rate = learning_rate_0

    '''사용할 함수 정의'''
    target_fn = self.squared_error
    gradient_fn = self.get_gradient

    while((no_improvement_cnt < 100)):
        inf_loop_cnt += 1
```

```
        value = sum(target_fn(x_i, y_i, delta) for x_i, y_i in zip(x, y))

        if value < min_value:
            min_delta, min_value = delta, value
            if inf_loop_cnt%20 == 1:
                print('min delta update : ', min_delta)
            no_improvement_cnt = 0
            learning_rate = learning_rate_0
        else:
            no_improvement_cnt += 1
            if (no_improvement_cnt%20 == 5):
                print("no improvement cnt : ", no_improvement_cnt)
            learning_rate *= 0.9

        idxs = [i for i in range(len(x))]
        random.shuffle(idxs)

        batch_size = 20
        for i in idxs[:batch_size]:
            gradient_i = gradient_fn(x[i], y[i], delta)
            delta = self.vector_subtract(delta,
                        self.scalar_multiply(learning_rate, gradient_i))

    return min_delta
```

함수의 입력인자인 x, y 는 입력되는 실제 데이터 값을 의미합니다. delta는 β와 α의 초기값, learning_rate_0은 학습률을 지정합니다.

target_fn과 gradient_fn에 sgd에 사용할 함수를 지정합니다.

학습이 이뤄지는 동안 100회 이상 변화가 없으면 학습이 종료되도록 하였습니다.

while문이 loop 하는동안 새로운 최소값을 찾으면 업데이트하고, step size를 초기화합니다.

update할때 sampling 되는 data의 batch size는 20입니다. 전체 data size 는 100입니다.

학습이 모두 종료되었다면 min_delta값을 반환하는데, 이는 β와 α값을 담고 있는 리스트입니다.

---

4. 구체화 모듈 단위테스트

4.1. 필요 함수 단위테스트