IE 597 Homework 4

Fall 2020

Haedong Kim

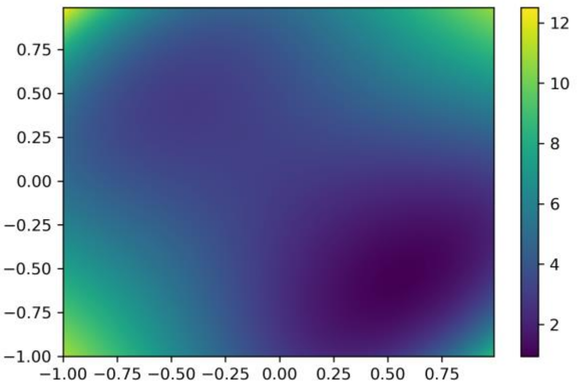
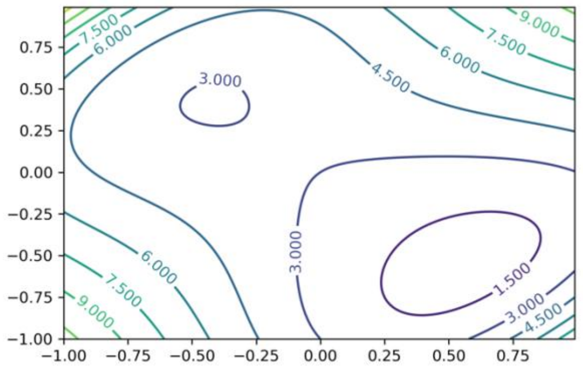
Problem 1.

1-a. Sketch the contour plot of the response surface

The objective function is defined as below. Its color map and contour plot are given in Table 1.

$$J(x_1, x_2) = (x_2 - x_1)^4 + 8x_1x_2 - x_1 + x_2 + 3$$

Table 1. (a) color map of J and (b) contour plot of J

(a)	(b)
	

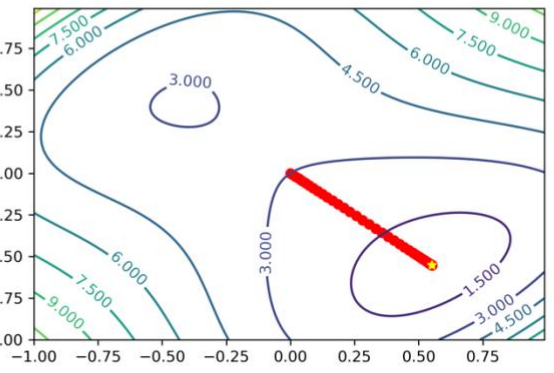
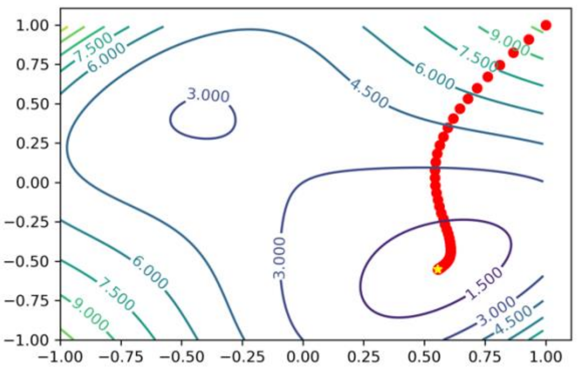
1-b. Gradient learning algorithm

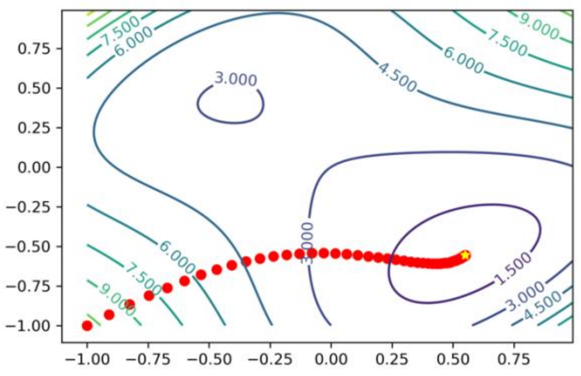
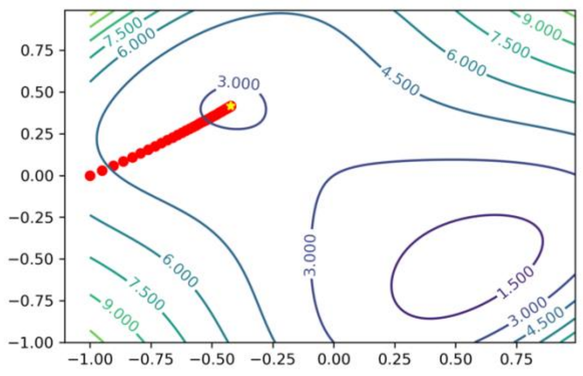
The gradient learning algorithm is implemented in Python and presented in Appendix-[1].

1-c. Gradient learning algorithm visualization with different initial points

Table 2 show learning procedures with four different points. (a), (b), and (c) successfully found the global minimum, but (d) fell in a local minimum.

Table 2. Learning procedures with four different initial points

(a) Initial point: (0, 0)	(b) Initial point: (1, 1)
	

(a) Initial point: (-1, -1)	(a) Initial point: (-1, 0)
	

1-d. Gradient learning algorithm visualization with different learning rates

Learning rate was 0.01 in 1-c. When the same algorithm ran with the larger learning rate, i.e., 0.1, starting at the initial point (0, 0), the optimization method did not converge. Figure 1 shows that value of the objective function is oscillating with the large learning rate.

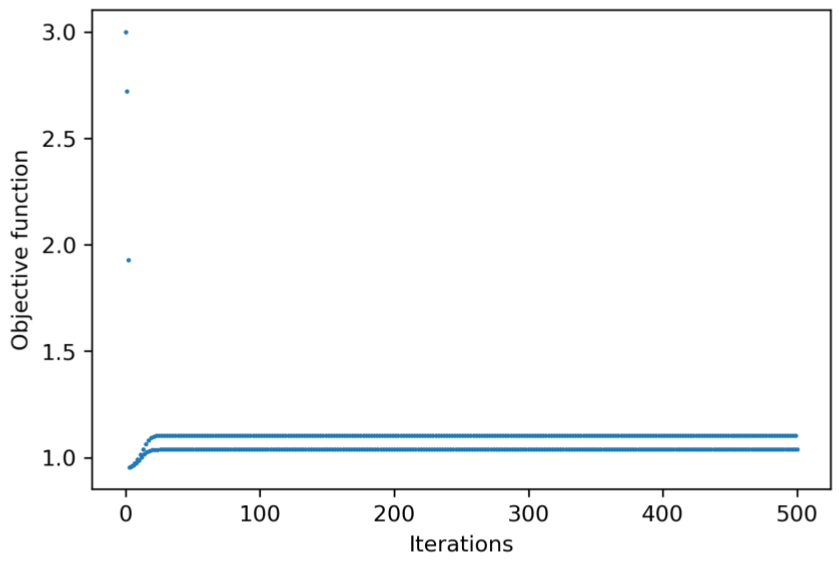


Figure 1. Objective function of J with the learning rate 0.1

Problem 2.

2-a. Data simulation

The error term was generated from the standard normal distribution. In other words, $y = 3.2 + 4.5x + e$ where $e \sim N(0, 1)$. x was generated by randomly selecting 20 integers between 0 and 40. Scatter plot of the generated data is given in Figure 2.

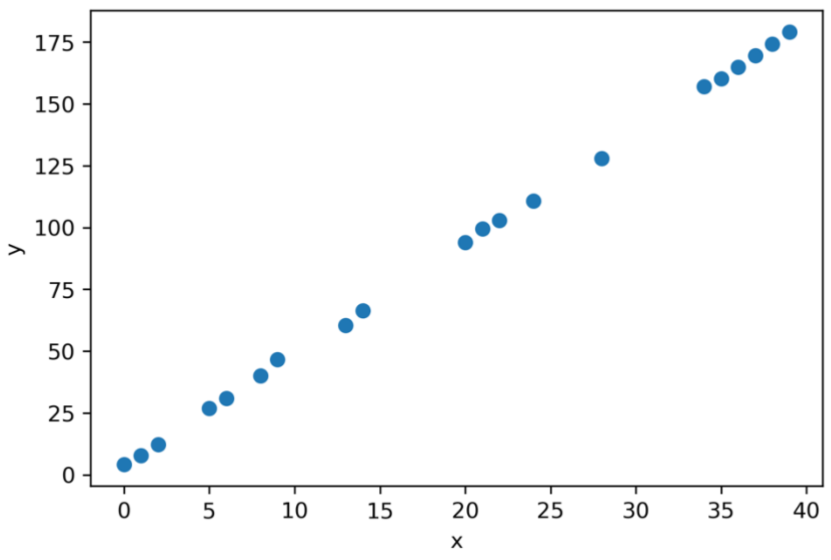


Figure 2. Scatter plot of the generate data

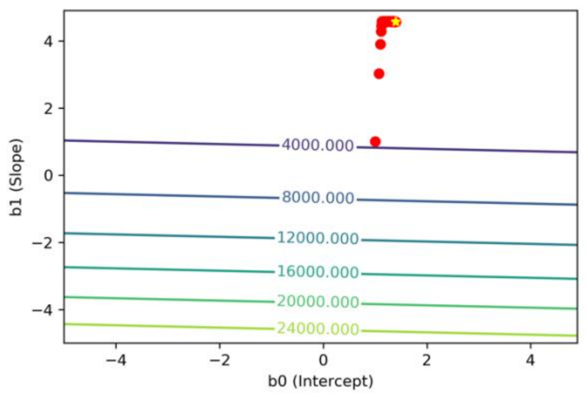
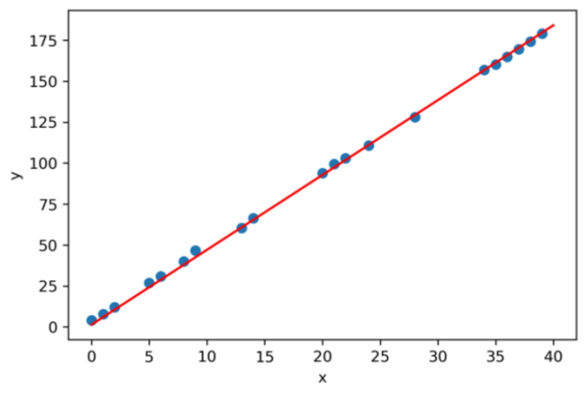
2-b. Gradient learning algorithm for univariate linear regression

The gradient learning algorithm is implemented in Python and presented in Appendix-[2].

2-c. Visualization of the learning process

Initial values were set as 1 for both the intercept and slope. The gradient learning algorithm ran 1,000 times with the learning rate 0.001. (a) in Table 3 shows the learning path and (b) the resulting linear function, presented in red-solid line.

Table 3. Learning results

(a)	(b)
	

Appendix – Python Code

[1] HW4-1_haedong.py

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt


## user-defined functions -----
def obj_fn(x1, x2):
    """cost function (or objective function)

    Args:
        x1 (numpy array)
        x2 (numpy array)

    Returns:
        J (numpy array)
    """
    J = np.power(x2-x1, 4) + 8*np.multiply(x1, x2) - x1 + x2 + 3
    return J

def gradient_fn(x1, x2):
    """calculate gradient

    Args:
        x1 (numeric): 1st dim
        x2 (numeric): 2nd dim

    Returns:
        list: gradient at current point
    """
    wrt_x1 = -4*np.power(x2-x1, 3) + 8*x2 - 1
    wrt_x2 = 4*np.power(x2-x1, 3) + 8*x1 + 1

    grads = [wrt_x1, wrt_x2]
    return grads
```

```
def gradient_learner(x1, x2, alpha):
    """gradient learning algorithm

    Args:
        x1 (numeric): 1st dim
        x2 (numeric): 2nd dim
        alpha (numeric): learning rate

    Returns:
        list: next point
    """
    grads = gradient_fn(x1, x2)

    new_x1 = x1 - alpha*grads[0]
    new_x2 = x2 - alpha*grads[1]

    updated = [new_x1, new_x2]
    return updated


## problem a -----
xx1, xx2 = np.meshgrid(np.arange(-1, 1, 0.01), np.arange(-1, 1, 0.01))
jj = obj_fn(xx1, xx2)

fig1, ax1 = plt.subplots(dpi=300)
cm = ax1.pcolormesh(xx1, xx2, jj)
fig1.colorbar(cm, ax=ax1)

fig2, ax2 = plt.subplots(dpi=300)
cp = ax2.contour(xx1, xx2, jj)
ax2.clabel(cp, inline=True, fontsize=10)


## problem b to d -----
# learning parameters
alpha = 0.01
```

```
init_pt = [0, 0]
tol = 0.00001
max_iter = 500

cnt = 1
pts = list()
fvals = list()

# initialization
pts.append(init_pt)
fvals.append(obj_fn(init_pt[0], init_pt[1]))
pts.append(gradient_learner(init_pt[0], init_pt[1], alpha))
while True:
    pt = pts[cnt]
    fvals.append(obj_fn(pt[0], pt[1]))

    # check stopping criteria
    fdiff = abs(fvals[cnt] - fvals[cnt-1])
    if (fdiff < tol) or cnt==max_iter:
        break
    else:
        pts.append(gradient_learner(pt[0], pt[1], alpha))
        cnt += 1

x1_search = [x[0] for x in pts]
x2_search = [x[1] for x in pts]

fig3, ax3 = plt.subplots(dpi=300)
cp = ax3.contour(xx1, xx2, jj)
ax3.clabel(cp, inline=True, fontsize=10)
ax3.scatter(x1_search, x2_search, color='red')
ax3.plot(x1_search[-1], x2_search[-1], marker='*', color='yellow')

fig4, ax4 = plt.subplots(dpi=300)
p = ax4.scatter(np.arange(len(fvals)), fvals, s=1)
plt.xlabel('Iterations')
plt.ylabel('Objective function')
```

[2] HW4-2_haedong.py

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt


## user-defined functions -----
def simulator(n, sig):
    x = np.random.choice(2*n, n, replace=False)
    x.sort()

    e = np.random.normal(0, sig, n)
    y = 3.2 + 4.5*x + e

    x = np.expand_dims(x, axis=1)
    y = np.expand_dims(y, axis=1)

    sim_data = np.concatenate((x, y), axis=1)
    return sim_data

def sse(x, y, b0, b1):
    hat = b0 + b1*x

    sse = 1/(2*len(x)) * np.sum(np.power(y - hat, 2))
    return sse

def sse_grid(x, y, bb0, bb1):
    grid_size = bb0.shape

    ee = np.empty(shape=grid_size)
    row_size = grid_size[0]
    col_size = grid_size[1]

    for i in range(row_size):
        for j in range(col_size):
            ee[i,j] = sse(x, y, bb0[i,j], bb1[i,j])
    return ee

def gradient_learner(x, y, b0, b1, alpha):
    # estimation
    hat = b0 + b1*x

    # calculate derivatives
    wrt_b0 = 1/len(x) * np.sum(hat - y)
    wrt_b1 = 1/len(x) * np.sum(np.multiply((hat - y), x))

    # update
    new_b0 = b0 - alpha * wrt_b0
    new_b1 = b1 - alpha * wrt_b1

    updated = [new_b0, new_b1]
    return updated


## problem a -----
sim_data = simulator(20, 1)
x = sim_data[:,0]
y = sim_data[:,1]

fig1, ax1 = plt.subplots(dpi=300)
ax1.scatter(x, y)
plt.xlabel('x')
plt.ylabel('y')


## problem b -----
alpha = 0.001
init_pt = [1, 1]
tol = 1.5
max_iter = 1000

cnt = 1
```

```
params = list()
fvals = list()

params.append(init_pt)
e = sse(x, y, init_pt[0], init_pt[1])
fvals.append(e)
# fvals.append(sse(x, y, init_pt[0], init_pt[1]))
params.append(gradient_learner(x, y, init_pt[0], init_pt[1], alpha))
while True:
    param = params[cnt]
    e = sse(x, y, param[0], param[1])
    fvals.append(e)
    # fvals.append(sse(x, y, param[0], param[1]))

    if (fvals[cnt]) < tol or cnt >= max_iter:
        break
    else:
        new_param = gradient_learner(x, y, param[0], param[1], alpha)
        params.append(new_param)
        cnt += 1


## problem c -----
bb0, bb1 = np.meshgrid(np.arange(-5, 5, 0.1), np.arange(-5, 5, 0.1))
ee = sse_grid(x, y, bb0, bb1)

b0_search = [p[0] for p in params]
b1_search = [p[1] for p in params]

fig2, ax2 = plt.subplots(dpi=300)
cp = ax2.contour(bb0, bb1, ee)
ax2.clabel(cp, inline=True, fontsize=10)
ax2.scatter(b0_search, b1_search, color='red')
ax2.plot(b0_search[-1], b1_search[-1], marker='*', color='yellow')
plt.xlabel('b0 (Intercept)')
plt.ylabel('b1 (Slope)')
```

```
best_param = params[-1]
x4fit = np.arange(0, 40, 0.01)
y4fit = best_param[0] + best_param[1]*x4fit

fig3, ax3 = plt.subplots(dpi=300)
ax3.scatter(x, y)
ax3.plot(x4fit, y4fit, color='red')
plt.xlabel('x')
plt.ylabel('y')
```