1 [Q1]

Last week, we talked about accelerated first-order method which contains Heavy Ball Method, Nesterov's method. Besides, we also do an analysis on the convergence. After that, we talked about Newton's Method and quasi-Newton Method. In particular, we talked about using BFGS to implement quasi-Newton Method. And then, we talked about non-smooth condition, for example, Lasso or l_1 norm, and talked about using subgradients to solve non-smooth optimization problems.

2 [Q2]

done

3 [Q3]

3.1 (a)

In homework 3, we have already shown that, if a function is M-smooth, $g(x) = \frac{M}{2}\|x\|_2^2 - f(x)$ is convex.

Simiarly, if a function is strongly convex, $g(x) = f(x) - \frac{m}{2}\|x\|_2^2$ is convex.

- M-smooth proof
  the corresponding function $g(x)$ is,

$$g(x) = \begin{cases} 0 & x < 1 \\ 24x^2 - 48x + 24 & 1 \leq x \leq 2 \\ 48x - 72 & x \geq 2 \end{cases}$$

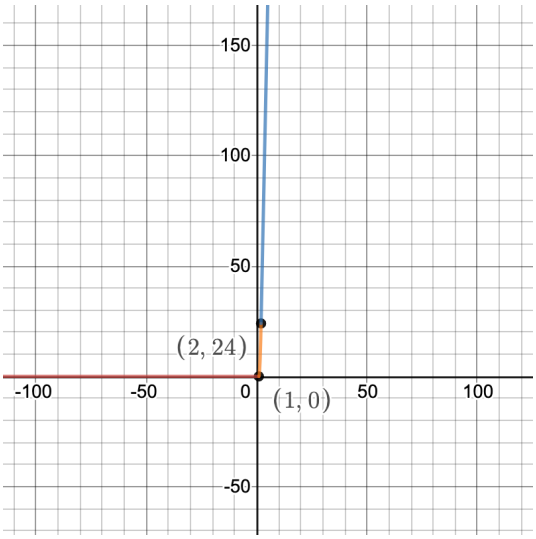

Figure 1: Illustration of M-smooth

As is shown in Figure 1, it's easy to find that it's convex.

- strong convexity proof
  the corresponding function $g(x)$ is,

$$g(x) = \begin{cases} 24x^2 & x < 1 \\ 48x - 24 & 1 \leq x \leq 2 \\ 24x^2 - 48x + 72 & x \geq 2 \end{cases}$$

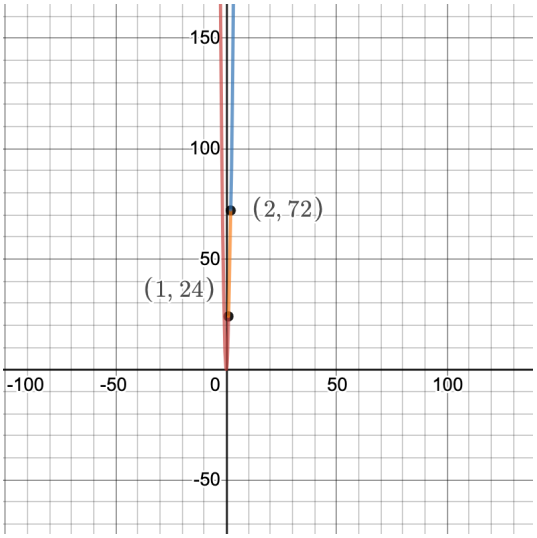

Figure 2: Illustration of strong convexity

As is shown in Figure 2, it' easy to show it's convex.

3.2 (b)

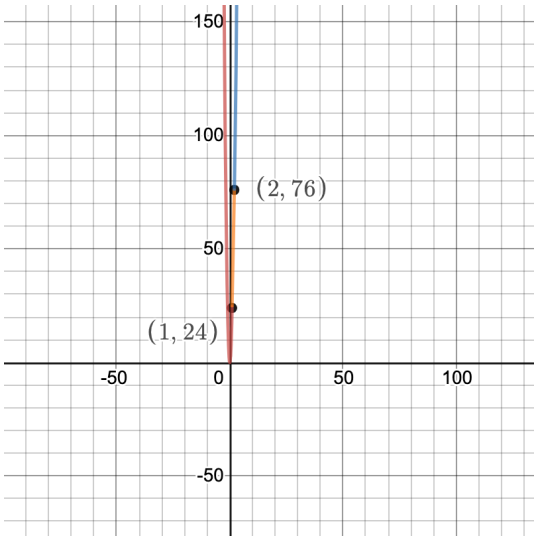

Figure 3: Illustration of global optimizer

As is shown in Figure 3, the global optimizer of f is $x^\star = 0$

3.3 (c)

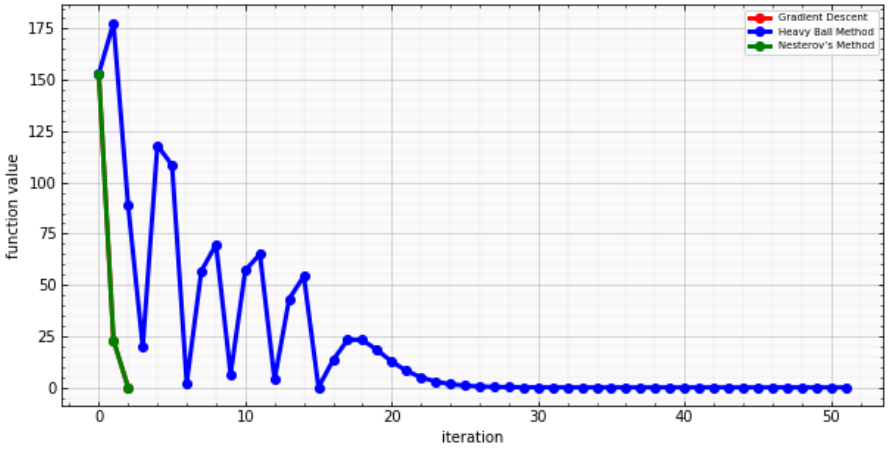

Figure 4: Illustration of global optimizer

Note that gradient descent and Nesterov's method has the same plot.

3.4 (d)

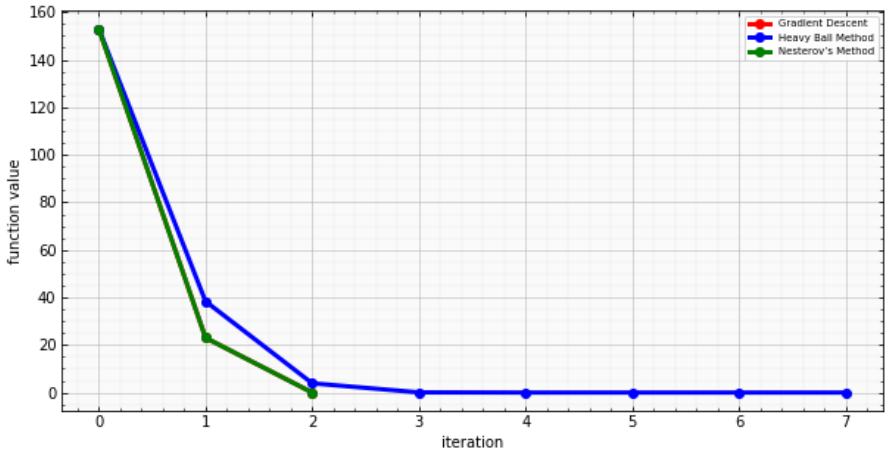

Figure 5: Illustration of global optimizer

Note that gradient descent and Nesterov's method has the same plot.

For gradient descent, the best α is $\mathbf{1/50}$, the total iteration is $\mathbf{2}$. For Heavy Ball Method, the best α is $\mathbf{0.017}$, the total iteration is $\mathbf{7}$. For Nesterov's Method, the best α is $\mathbf{1/50}$, the total iteration is $\mathbf{2}$.

4 [Q4]

4.1 (a)

Since f is M-smooth, we can have,

$$f(x_k + \alpha d_k) \leq f(x_k) + \langle \alpha d_k, \nabla f(x_k) \rangle + \frac{M\alpha^2}{2} \|d_k\|_2^2$$

From given condition,

$$\frac{M\alpha}{2}\|d_k\|_2^2 \leq (c_1 - 1)\langle d_k, \nabla f(x_k)\rangle$$

we can have,

$$f(x_k + \alpha d_k) \leq f(x_k) + \alpha\langle d_k, \nabla f(x_k)\rangle + \alpha(c_1 - 1)\langle d_k, \nabla f(x_k)\rangle$$

where we can simplify the right part as follows,

$$f(x_k) + \alpha\langle d_k, \nabla f(x_k)\rangle + \alpha(c_1 - 1)\langle d_k, \nabla f(x_k)\rangle = f(x_k) + \alpha c_1\langle d_k, \nabla f(x_k)\rangle$$

Hence, we have

$$f(x_k + \alpha d_k) \leq f(x_k) + \alpha c_1\langle d_k, \nabla f(x_k)\rangle$$

4.2 (b)

We can have,

$$\alpha_k = \overline{\alpha} * \rho^k = \rho^k$$

since armijo condition tells us,

$$\alpha_k \leq 2(c_1 - 1)\frac{\langle d_k, \nabla f(x_k)\rangle}{M\|d_k\|_2^2} \tag{1}$$

$$\leq \frac{2(1 - c_1)}{M} \tag{2}$$

where (1) to (2) follows $d_k = \nabla f(x_k)$.
Hence, we can have

$$\rho^k \leq \frac{2(1 - c_1)}{M}$$

And the number of backtracking iterations is upper bouned by

$$k \leq \log_\rho \frac{2}{M}(1 - c_1)$$

5 [Q5]

5.1 (a)

Yes, it's unique.

5.2 (b)

if x_0 is a minimizer of f, then it satisfies,

$$f(x) \geq f(x_0), \quad \text{for all } x \in \text{dom} f$$

Now consider the second function, apprently, x_0 is the minimizer of $\frac{\delta}{2}\|x - x_0\|_2^2$.
Hence, x_0 is the minimizer of $f_\delta(x)$, namely, $x_\delta^\star = x_0$.

5.3 (c)

By definition, we have,

$$f_\delta(x_\delta^\star) \leq f_\delta(x)$$

for all $x \neq x_\delta^\star$ (including $x = x^\star$).
Hence we can have,

$$f(x_\delta^\star) + \frac{\delta}{2}\|x_\delta^\star - x_0\|_2^2 \leq f(x^\star) + \frac{\delta}{2}\|x^\star - x_0\|_2^2 \tag{3}$$

$$f(x_\delta^\star) \leq f(x^\star) + \frac{\delta}{2}\|x^\star - x_0\|_2^2 \tag{4}$$

where, (9) to (10) equality holds when $x_\delta^\star = x_0$.

5.4 (d)

- M_δ-smooth:
  Let's denote $g(x) = \frac{M_\delta}{2}\|x\|_2^2 - f_\delta(x)$. we can have,

$$\nabla^2 g(x) \geq 0 \tag{5}$$

$$M_\delta - \nabla^2 f(x) - \delta \geq 0 \tag{6}$$

$$\nabla^2 f(x) \leq M_\delta - \delta \tag{7}$$

  Since, $f(x)$ is convex, we have

$$\nabla^2 f(x) \leq M\boldsymbol{I}$$

  Hence, we can conclude from that,

$$M_\delta = M + \delta$$

- Strongly convex (m_δ):
  Let's denote $h(x) = f_\delta(x) - \frac{m}{2}\|x\|_2^2$, we can have,

$$\nabla^2 h(x) \geq 0 \tag{8}$$

$$\nabla^2 f(x) + \delta - m_\delta \geq 0 \tag{9}$$

$$\nabla^2 f(x) \geq -\delta + m_\delta \tag{10}$$

  Since $f(x)$ is convex, we have $\nabla^2 f(x) \geq 0$, hence, we can get,

$$m_\delta = \delta$$

6 [Q6]

6.1 (a)

the update rule is:

$$p_{k+1}^{(n)} = p_k^{(n)} - \alpha_k * 4 \sum_{m \in N_n} (\|p^{(n)} - p^{(m)}\|_2^2 - \Delta^2)(p^{(n)} - p^{(m)})$$

6.2 (b)

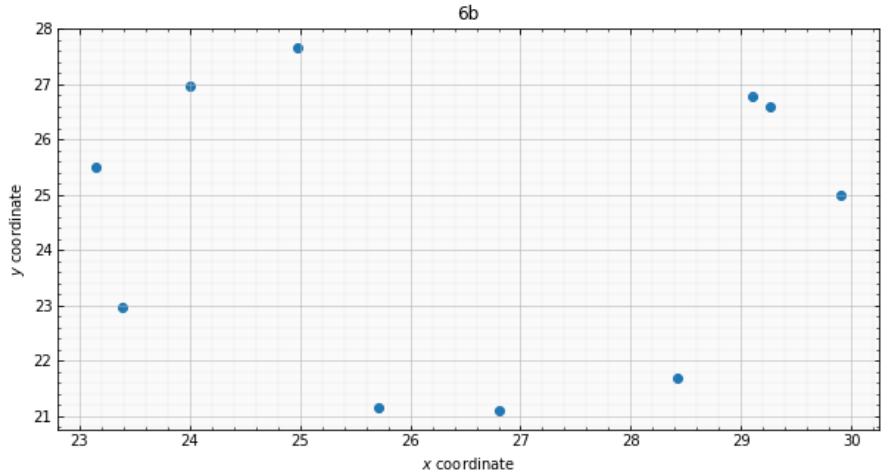

Figure 6: Illustration of global optimizer

I set α as **2e-5** and tolerance as **1e-5**, and the iteration is **6294**.

6.3 (c)

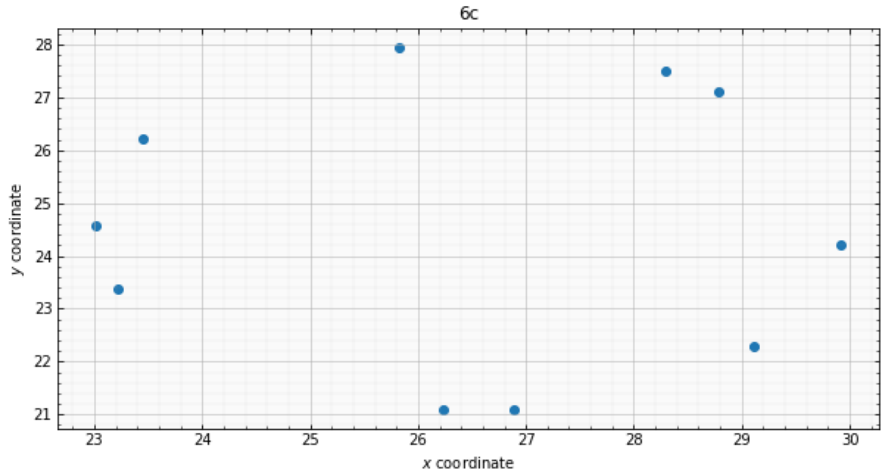

Figure 7: Illustration of global optimizer

I set α as **1e-5** and tolerance as **1e-5**, and the iteration is **4179**. And the update rule is:

$$p_{k+1}^{(n)} = p_k^{(n)} - \alpha_k * 4 \sum_{m \in N_n} (\|p^{(n)} - p^{(m)}\|_2^2 - \Delta^2)(p^{(n)} - p^{(m)})$$

6.4 (d)

$$\nabla^2 f(p^{(n)}) = 4 \sum_{m \in N_n} \left[(\|p_k^{(n)} - p_k^m\|_2^2 - \Delta^2)\boldsymbol{I} + 2(p_k^{(n)} - p_k^{(m)})(p_k^{(n)} - p_k^{(m)})^T \right]$$

Benefits: it will converge in far less iterations.

Drawbacks: it requires too much computation resouces to get hessian matrix.

7 [Q7]

7.1 (a)

Let's denote $f(\boldsymbol{x})$ as the object function, and we can get,

$$\nabla f(\boldsymbol{x}) = (\lambda \boldsymbol{D} + \boldsymbol{I})\boldsymbol{x} - \boldsymbol{y}$$

where, $\boldsymbol{I}$ is a $\mathbb{R}^{N*N}$ identity matrix.

7.2 (b)

$$\nabla^2 f(\boldsymbol{x}) = \lambda \boldsymbol{D} + \boldsymbol{I}$$

7.3 (c)

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - (\nabla^2 f(\boldsymbol{x}_k))^{-1} \nabla f(\boldsymbol{x}_k) \tag{11}$$
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - (\lambda \boldsymbol{D} + \boldsymbol{I})^{-1}((\lambda \boldsymbol{D} + \boldsymbol{I})\boldsymbol{x}_k - \boldsymbol{y}) \tag{12}$$
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{x}_k + (\lambda \boldsymbol{D} + \boldsymbol{I})^{-1}\boldsymbol{y} \tag{13}$$
$$\boldsymbol{x}_{k+1} = (\lambda \boldsymbol{D} + \boldsymbol{I})^{-1}\boldsymbol{y} \tag{14}$$

7.4 (d)

Yes, it's a scalable solution to this problem when N is very large. Because hessian matrix is a sparse matrix. When computing the inverse matrix, it's very fast.

8 [Q8]

Note that the backtracking strategy here I used didn't always take α as 1.0. Instead, it takes previous best α as input and iterates again. The main reason I used this is Because i think it's more engineering.

8.1 (a)

Here I set the minimum gradient tolerance as **1e-3**. And I think the best parameters of c and ρ are **0.1** and **0.2**. **962** gradient steps are required when using line search method. By the combined backtracking searches, total number of iterations is **965**. Compared with bisection which brings total iterations to 5120, backtracking method is a more faster way to converge.

8.2 (b)

If we set $\alpha \approx 0.001$ and $\beta_k \approx 0.95$, it converge at iteration **383**. In best case, I set α as **0.001** and β as **0.9**. And the total iterations are **210**. From my persepctive, when using

backtracking to adjust α at each iteration(where β is fixed as **0.9**), total iterations are **210**. Combined backtracking, total iteration is **213**.

8.3 (c)

When setting $\alpha \approx 0.001$, the total iteration is **212**. When using backtracking, here i set the initial α as **0.5**, and the total iteration is **162**. Combined with backtracking, total iteration is **166**.

8.4 (d)

When using backtracking, the total iteration is **6**.

8.5 (e)

When using backtracking, the total iteration is **12**.

9 Appendix

All the code can be found in my personal Github: **github.com/masqueraderx**

9.1 [Q3 code]

```
import numpy as np
from matplotlib import pyplot as plt
# Define function and Gradient
def func(x):
    if x < 1:
        return 25*x**2
    elif 1<= x <= 2:
        return x**2 + 48*x - 24
    else:
        return 25*x**2 - 48*x + 72

def gradient(x):
    if x < 1:
        return 50*x
    elif 1<= x <= 2:
        return 2*x + 48
    else:
        return 50*x - 48

# gradient descent
alpha = 1 / 50
x=3
k=0
iterations_GD = [0]
values_GD = [func(x)]
while abs(gradient(x)) > 1e-3:
    dk = gradient(x)
    x = x - alpha * dk
    k += 1
    values_GD.append(func(x))
    iterations_GD.append(k)
    print('{}th iteration: gradient is {}, x is {}'.format(k, dk, x))
```

```
# heavy ball method
alpha = 0.017 #0.017
beta = 0.007 #0.007
x = [3,3]
k=1
iterations_HBM = [0]
values_HBM = [func(x[k])]
while abs(gradient(x[k])) > 1e-3:
    dk = gradient(x[k])
    x.append(x[k] - alpha * dk + beta * (x[k] - x[k-1]))
    k += 1
    iterations_HBM.append(k-1)
    values_HBM.append(func(x[k]))
    print('{}th iteration: gradient is {}, x is {}'.format(k-1, dk, x[k]))

# Nesterov's method
alpha = 1 / 50
x = [3,3]
k=1
iterations_NM = [0]
values_NM = [func(x[k])]
while abs(gradient(x[k])) > 1e-3:
    beta = (k-1)/(k+2)
    pk = beta*(x[k]-x[k-1])
    dk = gradient(x[k] + pk)
    x.append(x[k] + pk - alpha * dk)
    k += 1
    iterations_NM.append(k-1)
    values_NM.append(func(x[k]))
    print('{}th iteration: gradient is {}, x is {}'.format(k-1, gradient(x[k
                                                ]), x[k]))

# Draw plot for three methods
fid = plt.figure(figsize=(10,5))

Axes = plt.subplot(1,1,1)
Axes.axes.tick_params(which='both',direction='in',top=True, right=True)
plt.minorticks_on()
Axes.set_facecolor((0,0,0,0.02))

plt.plot(iterations_NM, values_NM, 'k-o', linewidth=3, color = 'r', label='
                                        Gradient Descent')
plt.plot(iterations_HBM, values_HBM, 'b-o', linewidth=3, label = 'Heavy Ball
                                         Method')
plt.plot(iterations_NM, values_NM, 'k-o', linewidth=3, color='g', label='
                                        Nesterov's Method')
plt.grid(True,which='major',linewidth=0.5)
plt.grid(True,which='minor',linewidth=0.1)
plt.xlabel("iteration")
plt.ylabel("function value")
plt.legend(loc='upper right',fontsize='x-small')
plt.savefig('/Users/gexueren/Desktop/6270/assignment/hw04/hw04/3d.png')
```

9.2 [Q6 code]

```
import numpy as np
from matplotlib import pyplot as plt
np.random.seed(2021)
p = np.random.uniform(0,50,[2,10])
```

```
#calculate gradient
def gradient(p, n, delta):
    d = np.zeros(2)
    for m in range(p.shape[1]):
        if m == n:
            continue
        else:
            d += np.multiply((np.linalg.norm(p[:,n]-p[:,m]))**2 - delta**2,
                                                    p[:,n]-p[:,m])
    return d

#GD
def gradient_descent(p, alpha, maxiter, threshold):
    d = np.ones((2,10))
    k = 0
    while (k < maxiter) and (np.linalg.norm(d) > threshold):

        for i in range(p.shape[1]):
            d[:, i] = - gradient(p, i, 6)
        p = p + 4 * alpha * d
        k += 1
    print('final gradient is {}'.format(d[-1]))
    return p, k

# show results
points, iterations = gradient_descent(p, 2e-5, 10000, 1e-5)
print('points and total iteration is {} and {}'.format(points, iterations))
fid = plt.figure(figsize=(10,5))
Axes = plt.subplot(1,1,1)
Axes.axes.tick_params(which='both',direction='in',top=True, right=True)
plt.minorticks_on()
Axes.set_facecolor((0,0,0,0.02))
plt.scatter(points[0], points[1], label = 'points')
plt.grid(True,which='major',linewidth=0.5)
plt.grid(True,which='minor',linewidth=0.1)
plt.xlabel("$x$ coordinate")
plt.ylabel("$y$ coordinate")
# plt.legend(loc='upper right',fontsize='x-small')
plt.title('6b')
plt.savefig('/Users/gexueren/Desktop/6270/assignment/hw04/hw04/6b.png')

# Nesterov's method
def nesterov(p, alpha, maxiter, threshold):
    d = np.ones((2,10))
    k = 0
    pre_p = p
    pk = 0
    while (k < maxiter) and (np.linalg.norm(d) > threshold):
        beta = (k - 1) / (k + 2)
        for i in range(p.shape[1]):
            d[:, i] = - gradient(p + pk, i, 6)
        p = p + 4 * alpha * d + pk
        pk = beta * (p - pre_p)
        k += 1
        pre_p = p
    print('final gradient is {}'.format(d[-1]))
    return p, k

points, iterations = nesterov(p, 1e-5, 10000, 1e-5)
print('points and total iteration is {} and {}'.format(points, iterations))
fid = plt.figure(figsize=(10,5))
Axes = plt.subplot(1,1,1)
Axes.axes.tick_params(which='both',direction='in',top=True, right=True)
```

```
plt.minorticks_on()
Axes.set_facecolor((0,0,0,0.02))
plt.scatter(points[0], points[1], label = 'points')
plt.grid(True,which='major',linewidth=0.5)
plt.grid(True,which='minor',linewidth=0.1)
plt.xlabel("$x$ coordinate")
plt.ylabel("$y$ coordinate")
# plt.legend(loc='upper right',fontsize='x-small')
plt.title('6c')
plt.savefig('/Users/gexueren/Desktop/6270/assignment/hw04/hw04/6c.png')
```

9.3 [Q8 code]

```
import numpy as np
from sklearn import datasets
import math
import time
# the logistic function
def logistic_func(theta, x):
    t = x.T @ theta
    g = np.zeros(t.shape)
    # split into positive and negative to improve stability
    g[t>=0.0] = 1.0 / (1.0 + np.exp(-t[t>=0.0]))
    g[t<0.0] = np.exp(t[t<0.0]) / (np.exp(t[t<0.0])+1.0)
    return g

# function to compute output of LR classifier
def lr_predict(theta,x):
    # form Xtilde for prediction
    x = np.vstack((x.T , np.ones(x.shape[0])))
    return logistic_func(theta,x)

# function to evaluate objective function (-f)
def f_eval(theta, x, y):
    t = x.T @ theta
    return -np.vdot(t,y) + np.sum(np.log(1+np.exp(t)))

# function to compute the gradient of -f
def grad(theta, x, y):
    g = logistic_func(theta,x)
    return -(x @ (y-g))

def hessian(theta, x):
    g = logistic_func(theta, x)
    n = g.shape[0]
    return np.dot(np.dot(np.dot(x, np.diag(g.reshape(n))), np.diag(1-g.
                                        reshape(n))), x.T)


# gradient descent
# returns theta and number of iterations
def gradDesc(x, y, alpha, c, rho, delta, maxiter, backTracking=False):
    # Initialization
    theta = np.zeros(x.shape[0])
    d = -grad(theta, x, y) # 3*1
    k = 0
    inner = 0
    while (k < maxiter) and (np.linalg.norm(d) > delta):
        '''
        this is backtracking tragetegy, Or should I say: Strategy :)
        '''
```

```
        if backTracking:
            alpha, m = back_tracking(x, y, theta, d, alpha, c, rho)
        theta = theta + alpha * d
        d = -grad(theta, x, y)
        k = k + 1
        if backTracking:
            inner = inner + m
    total = k + inner
    return theta, k, total


# heavy ball method
# returns theta and number of iterations
def heavyBall(x, y, alpha, beta, c, rho, delta, maxiter, backTracking=False)
                                                :
    # Initialization
    theta = [np.zeros(x.shape[0]), np.zeros(x.shape[0])]
    d = -grad(theta[-1], x, y) # 3*1
    k = 1
    inner = 0
    while (k < maxiter) and (np.linalg.norm(d) > delta):
        '''
        this is backtracking tragetegy, Or should I say: Strategy :)
        '''
        if backTracking:
            alpha, m = back_tracking(x, y, theta[-1], d, alpha, c, rho)
        theta.append(theta[k] + alpha * d + beta * (theta[k] - theta[k-1]))
        d = -grad(theta[-1], x, y)
        k = k + 1
        if backTracking:
            inner = inner + m
    total = k + inner
    return theta[-1], k-1, total-1



# nesterov's method
# returns theta and number of iterations
def nesterov(x, y, alpha, c, rho, delta, maxiter, backTracking=False):
    # Initialization
    theta = [np.zeros(x.shape[0]), np.zeros(x.shape[0])]
    d = -grad(theta[-1], x, y) # 3*1
    k = 1
    p = 0
    inner = 0
    while (k < maxiter) and (np.linalg.norm(d) > delta):
        '''
        this is backtracking tragetegy, Or should I say: Strategy :)
        '''
        if backTracking:
            alpha, m = back_tracking(x, y, theta[-1], d, alpha, c, rho)
        theta.append(theta[k] + alpha * d + p)
        k = k + 1
        beta = (k - 1) / (k + 2)
        p = beta * (theta[-1] - theta[-2])
        d = -grad(theta[-1] + p, x, y)
        if backTracking:
            inner = inner + m
    total = k + inner
    return theta[-1], k-1, total-1
```

```
# newton's method
# returns theta and number of iterations
def newton(x, y, alpha, c, rho, delta, maxiter, backTracking=False):
    # Initialization
    theta = np.zeros(x.shape[0])
    d = np.linalg.inv(hessian(theta, x)) @ (-grad(theta, x, y)) # 3*1
    k = 0
    inner = 0
    while (k < maxiter) and (np.linalg.norm(d) > delta):
        '''
        this is backtracking tragetegy, Or should I say: Strategy :)
        '''
        if backTracking:
            alpha, m = back_tracking(x, y, theta, d, alpha, c, rho)
        theta = theta + alpha * d
        d = np.linalg.inv(hessian(theta, x)) @ (-grad(theta, x, y))
        k = k + 1
        if backTracking:
            inner = inner + m
    total = inner + k
    return theta, k, total

def bfgs(x, y, alpha, c, rho, delta, maxiter, backTracking=False):
    # Initialization
    theta = np.zeros(x.shape[0])

    h=np.linalg.inv(hessian(theta,x))
    g=grad(theta, x, y)
    d = -h @ g
    k = 0
    inner = 0
    while (k < maxiter) and (np.linalg.norm(d) > delta):
        theta0 = theta
        g0 = g
        if backTracking:
            alpha, m = back_tracking(x, y, theta, d, alpha, c, rho)
        theta = theta + alpha*d
        g = grad(theta, x, y)
        s = np.mat(theta-theta0).T
        r = np.mat(g-g0).T
        a = h @ r
        gama=s.T @ r
        h = h + np.array((gama+(r.T @ a))/gama**2)*np.array((s @ s.T)) - np.
                                                array((a @ s.T)/gama) - np.
                                                array((s @ a.T)/gama)

        d = -h @ g
        k = k + 1
        if backTracking:
            inner = inner + m
        total = inner + k
    return theta, k, total


# back_tracking
# returns alpha
def back_tracking(x, y, theta, d, alpha, c, rho):

    '''
    Phi function, see notes
    '''
    def phi(alpha):
        return f_eval(theta + alpha * d, x, y)
    '''
```

```
    h function, see notes
    '''
    def h(alpha):
        return f_eval(theta, x, y) + c * alpha * (d.T @ d)
    '''
    backtracking
    '''
    m = 0
    while phi(alpha) > h(alpha):
        alpha = rho * alpha
        m += 1
    return alpha, m

# Generate dataset
np.random.seed(2020) # Set random seed so results are repeatable
x,y = datasets.make_blobs(n_samples=100,n_features=2,centers=2,cluster_std=6
                                                    .0)

# Form Xtilde
x = np.vstack((x.T , np.ones(x.shape[0])) ) #3*100

# Gradient Descent
'''
x: np.array
y: np.array
alpha: float
c: float 1e-4 - 0.3
rho: float 0.1 - 0.8
delta: float
maxiter: int
backTracking: bool, default: False
'''
# theta_gd, num_iters, total = gradDesc(x, y, 0.5, 0.1, 0.2, 1e-3, 10000,
                                                    False)
theta_gd, num_iters, total = gradDesc(x, y, 0.5, 0.1, 0.2, 1e-3, 10000, True
                                                    )
print('Number of iterations required (Gradient Descent): {0}'.format(
                                                    num_iters))
print('Number of iterations required (Gradient Descent Combined backtracking
                                                    ): {0}'.format(total))
print('Solution: [{0} {1} {2}]^T'.format(theta_gd[0], theta_gd[1], theta_gd[
                                                    2]))

# Heavy Ball Method
'''
x: np.array
y: np.array
alpha: float
beta: float
c: float
rho: float
delta: float
maxiter: int
backTracking: bool, default: False
'''

# theta_hbm, num_iters = heavyBall(x, y, 0.001, 0.95, 0.1, 0.2, 1e-3, 10000,
                                                    False)
# theta_hbm, num_iters = heavyBall(x, y, 0.001, 0.9, 0.1, 0.2, 1e-3, 10000,
                                                    False)
theta_hbm, num_iters, total = heavyBall(x, y, 0.25, 0.9, 0.1, 0.2, 1e-3,
                                                    10000, True)
```

```
print('Number of iterations required (Heavy Ball Method): {0}'.format(
                                        num_iters))
print('Number of iterations required (Heavy Ball Method Combined
                                        backtracking): {0}'.format(total))
print('Solution: [{0} {1} {2}]^T'.format(theta_hbm[0], theta_hbm[1],
                                        theta_hbm[2]))

# Nesterov's method
'''
x: np.array
y: np.array
alpha: float
c: float
rho: float
delta: float
maxiter: int
backTracking: bool, default: False
'''


# theta_nm, num_iters, total = nesterov(x, y, 0.001, 0.1, 0.2, 1e-3, 10000,
                                        False)
theta_nm, num_iters, total = nesterov(x, y, 0.5, 0.1, 0.2, 1e-3, 10000, True
                                        )
print('Number of iterations required (Nesterov\'s Method): {0}'.format(
                                        num_iters))
print('Number of iterations required (Nesterov\'s Method Combined
                                        backtracking): {0}'.format(total))
print('Solution: [{0} {1} {2}]^T'.format(theta_nm[0], theta_nm[1], theta_nm[
                                        2]))

# Newton's method
'''
x: np.array
y: np.array
alpha: float
c: float
rho: float
delta: float
maxiter: int
backTracking: bool, default: False
'''
# theta_Nm, num_iters, total = newton(x, y, 1.0, 0.1, 0.2, 1e-3, 10000,
                                        False)
theta_Nm, num_iters, total = newton(x, y, 1.0, 0.1, 0.2, 1e-3, 10000, True)
print('Number of iterations required (Newton\'s method): {0}'.format(
                                        num_iters))
print('Number of iterations required (Newton\'s Method  Combined
                                        backtracking): {0}'.format(total))
print('Solution: [{0} {1} {2}]^T'.format(theta_Nm[0], theta_Nm[1], theta_Nm[
                                        2]))

# BFGS
'''
x: np.array
y: np.array
alpha: float
c: float
rho: float
delta: float
maxiter: int
backTracking: bool, default: False
'''
```

```
theta_bfgs, num_iters, total = bfgs(x, y, 1.0, 0.1, 0.2, 1e-3, 10000, True)
print('Number of iterations required (BFGS): {0}'.format(num_iters))
print('Number of iterations required (BFGS Combined backtracking): {0}'.
                                        format(total))
print('Solution: [{0} {1} {2}]^T'.format(theta_bfgs[0], theta_bfgs[1],
                                        theta_bfgs[2]))
```