**ENMG 616 – Advanced Optimization Techniques & Algorithms**



Assignment No. 2

By: Ralph Mouawad

ID - 202204667

To: Dr. Maher Nouiehed

Fall 2023-2024

October 15th, 2023

*Some derivations are hand-written, if they don’t appear please tell me to send them by mail (if a problem occurs while opening the word file from Moodle).*

Exercise 1 –

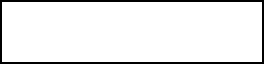
1&2)

data = readmatrix('income.data.csv');

X = data(2:499,2);

Y = data(2:499,3);

3)



MATLAB code:

n = length(X);

X2 = [X,ones(n,1)];

X2;

theta = X2'\*X2 \ X2'\*Y;

theta;

We got an approximated value for theta\*.

theta\* = (0.7138, 0.2043)

4)

L = max(eig(X\*X'));

L;

We got an approximated value for the Lipschitz constant:

L = 1.1437e+04.

5)

Now, we want to implement Gradient Descent to compute the optimal solution. This is the MATLAB code used:

thetaG = [1,1]';

gradient = (2/n)\*(X2'\*X2\*thetaG - X2'\*Y);

a = 1/L;

stop = 10^-3;

while norm(gradient)>=stop

thetaG = thetaG - a\*gradient;

gradient = (2/n)\*(X2'\*X2\*thetaG - X2'\*Y);

end

thetaG;

After running this code, we got the following results:

thetaG = [0.7131, 0.2082]^t.

This response is like the one obtained before.

6)

scatter(X, Y);

hold on;

plot([0:0.5:10],theta(2) + theta(1)\*[0:0.5:10])



7)

We plotted the graph displaying the different points from our data set, as well as the line that approximates the relation between our variables X and Y. We can see that here, income and happiness have a linear relationship that we approximated with our fitting line, while minimizing the Sum of Squared error (SSE) to get the best accuracy.

Part 1.1 –

1&2)

maxValue = max(X);

maxValue;

Xnew = ones(n,1);

Ynew = ones(n,1);

for i = 1:n

Xnew(i) = X(i)/maxValue;

Ynew(i) = Y(i)/maxValue;

end

Xnew;

Ynew;

We got maxValue = 7.4815.

3)

Lnew = max(eig(Xnew\*Xnew'));

Lnew;

We got Lnew = 204.3327.

4)

X2new = [Xnew, ones(n,1)];

thetaGnew = [1,1]';

gradientNew = (2/n)\*(X2new'\*X2new\*thetaGnew - X2new'\*Ynew);

aNew = 1/Lnew;

stop = 10^-3;

while norm(gradientNew)>=stop

thetaGnew = thetaGnew - aNew\*gradientNew;

gradientNew = (2/n)\*(X2new'\*X2new\*thetaGnew - X2new'\*Ynew);

end

thetaGnew;

5)

After running our code, we got that thetaGnew = [0.7030, 0.0340].

We can see that theta0 is much less than our previous part. This is mainly due to the change of our Lipschitz constant which was relatively smaller after normalizing the dataset we had.

When reducing the value of L, our step-size got bigger because a = 1/L (inversely proportional), and chances of divergence of our algorithm increase.

Exercise 2 –

1)

n = 50;

B = randn(n,n);

Q = B\*B';

MinEig = min(eig(Q));

Q = Q + (MinEig + rand(1))\*eye(n);

q = 10\*randn(n,1);

2)



X\_FOSS = -Q\q;

X\_FOSS

See MATLAB code for the output.

3) We will implement the Gradient Descent algorithm to compute the solution.

1. Step-size: Exact Line Search:

In Exact Line Search, we select our step-size to be:



We must find the value of ‘a’ that minimizes the function f( x + a\*d).

g = @(x) (Q\*x + q); % gradient function

f = @(x) (1/2)\*x'\*Q\*x + q'\*x; % obj function

a = 0.1;

x0 = zeros(n,1);

x = x0;

for i =1:1000 % stopping criteria

h = @(a) (f(x - a\*g(x)));

a = fminsearch(h,0);

x = x - a\*g(x);

end

x\_opt1 = x;

x\_opt1;

See MATLAB code for the output.



1. Armijo Rule / Back-Tracking:

g = @(x) (Q\*x + q); % gradient function

f = @(x) (1/2)\*x'\*Q\*x + q'\*x; % obj function

a2 = 0.1;

x0 = zeros(n,1);

x = x0;

sigma = 0.2;

beta = 0.5;

while f(x) - f(x - a2\*beta\*g(x)) <=a2\*sigma\*beta\*g(x)'\*g(x)

a2 = beta\*a2;

x = x - a2\*g(x);

end

x\_opt2 = x; x\_opt2;

1. Diminishing step-size:

We will take a = 0.1/r,

We will increment ‘r’ after each iteration, until the norm of the gradient becomes less than the stopping condition (0.001)

MATLAB code:

x0 = zeros(n,1);

x = x0;

g = @(x) (Q\*x + q);

f = @(x) (0.5\*x'\*Q\*x + q'\*x);

r = 1;

for i =1:1000 % stopping criteria

a3 = 0.1/r;

x = x - a3\*g(x);

r = r+10;

end

x\_opt3 = x;

x\_opt3;

1. Constant step-size:

g = @(x) (Q\*x + q);

x0 = zeros(n,1);

x = x0;

L = max(eig(Q));

for i = 1:1000

x = x -(1/L)\*g(x);

end

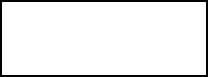
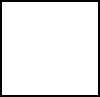
x\_opt4 = x;

x\_opt4;

4) Newton’s Method:



Here we need to compute the Hessian.



MATLAB code:

g = @(x) (Q\*x + q); %gradient. Hessian is inv(Q)

x0 = zeros(n,1);

x = x0;

L = max(eig(Q));

for i = 1:1000

d = -inv(Q)\*g(x);

a5 = 1/L;

x = x + a5\*d;



end

x\_opt5 = x;

x\_opt5;

5) Nesterov Accelerated Gradient Descent Algorithm:

g=@(x)(Q\*x+q);

a=[0 0];

X=zeros(n,2);

L=max(eig(Q));

gradx=1;

while norm(gradx)>0.0001

%updating the point

a(2)=0.5\*(1+sqrt(4\*(a(1))^2)+1);

y=X(1:50,2)+((a(1)-1)/a(2))\*(X(1:50,2)-X(1:50,1));

X(1:50,2)=y-(1/L)\*g(y);

%updating the list of X and a

a(1)=a(2);

X(1:50,1)=X(1:50,2);

gradx=g(X(1:n,1));

end

x\_opt6 = X(1:50,1);

x\_opt6

6) We will find the optimal solution and then find x at 1000 iterations and compute the error. You can find the updated codes in the MATLAB files.

E1 = -10.5402

E2 = -0.3955

E3 = -11.8888

E4 = -10.1267

E5 = -11.2689

E6 = -13.1270

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Realization1 | Realization2 | Realization3 | Realization4 | Realization5 |
| Exact min | -9.3 | -9.48 | -10.24 | -9.73 | -9.9 |
| Armijo | -0.6467 | -0.342 | -1.1164 | -0.25 | -0.43 |
| Dim | -10.83 | -10.44 | -11.10 | -11.11 | -10.62 |
| Constant | -9.0084 | -9.18 | -9.875 | -9.48 | -9.57 |
| Newton | -11.0359 | -11.0828 | -11.2055 | -11.23 | -11.06 |
| Nesterov | -3.23 | -5.8009 | -2.16 | -4.23 | -1.714 |
| Kappa | 335.33 | 263.32 | 265.44 | 332.04 | 225.22 |

|  |  |
| --- | --- |
|  | Average log-relative accuracy |
| Exact minimization | -9.73 |
| Armijo  Dim | -0.56  -10.82 |
| Constant | -9.423 |
| Newton | 11.123 |
| Nesterov | -3.43 |

Part 2.1 –

See the updated MATLAB code.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Realization1 | Realization2 | Realization3 | Realization4 | Realization5 |
| Exact min | -10.94 | -10.63 | -10.7 | -10.94 | -10.77 |
| Armijo | -0.38 | -0.116 | -0.31 | -0.55 | -0.61 |
| Dim | -10.56 | -10.22 | -10.33 | -10.46 | -10.47 |
| Constant | -10.6 | -10.275 | -10.37 | -10.48 | -10.5 |
| Newton | -11.06 | -11.067 | -11.065 | -11.066 | -11,067 |
| Nesterov | -13.23 | -13.103 | -12.94 | -13.078 | -13.17 |
| Kappa | 17.54 | 19.71 | 19.88 | 18.99 | 19.33 |

|  |  |
| --- | --- |
|  | Average log-relative accuracy |
| Exact minimization | -10.796 |
| Armijo  Dim | -0.3932  -10.408 |
| Constant | -10.445 |
| Newton | -11.065 |
| Nesterov | -13.1042 |

Part 2.2 – Restart in accelerated Nesterov

n = 50;

B = randn(n,n);

Q = B\*B';

MinEig = min(eig(Q));

Q = Q + (MinEig + 10)\*eye(n);

q = 10\*randn(n,1);

grad=@(x)(Q\*x+q);

X=zeros(n,2);

x0=zeros(n,1);

L=max(eig(Q));

gradx=1;

kapp=max(eig(Q))/min(eig(Q));

ckap=floor(sqrt(kapp));

xop=-inv(Q)\*q;

x1=[];

k1=[];

T=0;

a1=[0 0];

while norm(gradx)>0.0001

if mod(T,ckap)==0

a1=[0 0];

end

%updating the point

a1(2)=0.5\*(1+sqrt(4\*(a1(1)^2)+1));

y=X(1:n,2)+((a1(1)-1)/a1(2))\*(X(1:n,2)-X(1:n,1));

X(1:n,1)=X(1:n,2);

X(1:n,2)=y-(1/L)\*grad(y);

%updating the list of X and a

a1(1)=a1(2);

gradx=grad(X(1:n,1));

k1=[k1 T];

e=log(norm(X(1:n,1)-xop)/norm(x0-xop));

x1=[x1 e];

T=T+1;

end

X=zeros(n,2);

a2=[0 0];

gradx=1;

x2=[];

k2=[];

T=0;

while norm(gradx)>0.0001

if mod(T,5\*ckap)==0

a2=[0 0];

end

%updating the point

a2(2)=0.5\*(1+sqrt(4\*(a2(1)^2)+1));

y=X(1:n,2)+((a2(1)-1)/a2(2))\*(X(1:n,2)-X(1:n,1));

X(1:n,1)=X(1:n,2);

X(1:n,2)=y-(1/L)\*grad(y);

%updating the list of X and a

a2(1)=a2(2);

gradx=grad(X(1:n,1));

k2=[k2 T];

e=log(norm(X(1:n,1)-xop)/norm(x0-xop));

x2=[x2 e];

T=T+1;

end

X=zeros(n,2);

a3=[0 0];

gradx=1;

x3=[];

k3=[];

T=0;

while norm(gradx)>0.0001

if mod(T,20\*ckap)==0

a3=[0 0];

end

%updating the point

a3(2)=0.5\*(1+sqrt(4\*(a3(1)^2)+1));

y=X(1:n,2)+((a3(1)-1)/a3(2))\*(X(1:n,2)-X(1:n,1));

X(1:n,1)=X(1:n,2);

X(1:n,2)=y-(1/L)\*grad(y);

%updating the list of X and a

a3(1)=a3(2);

gradx=grad(X(1:n,1));

k3=[k3 T];

e=log(norm(X(1:n,1)-xop)/norm(x0-xop));

x3=[x3 e];

T=T+1;

end

X=zeros(n,2);

a4=[0 0];

gradx=1;

x4=[];

k4=[];

T=0;

while norm(gradx)>0.0001

%updating the point

a4(2)=0.5\*(1+sqrt(4\*(a4(1)^2)+1));

y=X(1:n,2)+((a4(1)-1)/a4(2))\*(X(1:n,2)-X(1:n,1));

X(1:n,1)=X(1:n,2);

X(1:n,2)=y-(1/L)\*grad(y);

%updating the list of X and a

a4(1)=a4(2);

gradx=grad(X(1:n,1));

k4=[k4 T];

e=log(norm(X(1:n,1)-xop)/norm(x0-xop));

x4=[x4 e];

T=T+1;

end

figure(1)

plot(k1,x1)

figure(2)

plot(k2,x2)

figure(3)

plot(k3,x3)

figure(4)

plot(k4,x4)

I obtained the following plots: (See next page).











We can see that each time we increase the number of iterations before restart, the function becomes less linear. Without restart, it was a straight line, with sqrt(k), it was more a quadratic function, and it kept going each time we increase the restart.