STA6106 Project 1

Travis Loebs

October 2, 2016

Problem 1

1.1 The problem

can be expressed in the matrix form

where

1.2 This problem can be solved in R using the quadprog library. However, because the quadprog library solves functions of the form

we must turn our maximization problem into a minimization problem by multiplying the objective and constraint functions by -1. First, we load the library and create our matrices at outlined in problem 1.1, except all of the matrices are multiplied by -1 with the exception of , because 'quadprog' by default makes negative. Additionally, the constraint matrix must be modified for use with the 'quadprog' library by putting the equality constraint in the first row, such that

and setting the 'meq' parameter equal to one, so that 'solve.QP' recognizes this constraint as an equality constraint.

library(quadprog)  
K = 2\*diag (c (2, 1, 1));  
C = c(20, 16, 0);  
A = matrix (0, nrow=5, ncol=3);  
A[1,] = c(-1, -1, 1);  
A[2,] = c(-1, -1, 0);  
A[3,] = c(1, 0, 0);  
A[4,] = c(0, 1, 0);  
A[5,] = c(0, 0, 1);  
d = c(0, -5, 0, 0, 0)

Using the 'solve.QP' function in R, we can solve this problem and set the estimator, , equal to the solution.

xHat = solve.QP(K, C, t(A), d, meq=1)$solution  
xHat

[1] 2.333333 2.666667 5.000000

We see that this function, subject to the restraints, is maximized at and .

Problem 2

2.1 Given the objective and constraint functions, the Lagrangian is

2.2 To find the dual problem, we first differentiate the Lagrangian with respect to

Setting to zero and solving for , we get

So, the dual problem becomes

2.3 The Karush-Kuhn-Tucker (KKT) conditions are given by:

2.4 Let and be the solutions to the primal problem, and , , and be the solutions to the dual problem. Then all of the KKT optimality conditions hold, and , where and . Thus, strong duality holds because the duality gap is zero.

2.5 We can use a stochastic hill climb algorithm to solve the dual problem. This is because the Hessian of in the dual problem is negative semi-definite, that is, all of its eigenvalues are negative or zero. This is verified by finding the Hessian matrix, H:

And then finding the eigenvalues of the Hessian matrix

H <- matrix(c(-3/2, -1/2, -1/2, -1/2, -1/2, 0, -1/2, 0, -1/4), nrow=3, ncol=3)  
eigH <- eigen(H)$values  
round(eigH, 4)

[1] 0.0000 -0.4069 -1.8431

Which are all negative or zero. Now, because the function is concave, any local maximimum is a global maximum, and a stochastic hill climb algorithm can be used to find the maximum. This is done by starting with arbitrary ordered points (, , ) under the constraints given in the KKT conditions. Then, a neighboring set of ordered points is randomly selected and is only accepted if it improves (increases) the objective function evaluated at those points. This process is iterated many times until the set of ordered points doesn't change to any of it's neighboring points, i.e. the maximum is found.

This algorithm is implemented in C++.

#include <random>

#include <iostream>

using namespace std;

void randomWalk(const int n, double & lam1,

double & lam2, double & lam3,

default\_random\_engine & generator,

normal\_distribution <double> & rnorm);

double dualProblem(const double lam1,

const double lam2, const double lam3);

int main()

{

// Define the number of runs

const int n = 1000000;

// Set up random number generator

random\_device rd;

default\_random\_engine generator(rd());

// Use standard normal for steps in each direction

normal\_distribution<double> rnorm(0.0, 1.0);

// Set initial values

double lam1 = 100.0;

double lam2 = 100.0;

double lam3 = 100.0;

// Call the random walk function

randomWalk(n, lam1, lam2, lam3, generator, rnorm);

return 0;

}

void randomWalk(const int n, double & lam1,

double & lam2, double & lam3,

default\_random\_engine & generator,

normal\_distribution <double> & rnorm)

{

// create temporary variables to test acceptance

double temp1;

double temp2;

double temp3;

// Take *n* random steps

for (int i = 0; i < n; i++)

{

// Take random step in direction 1

temp1 = lam1 + rnorm(generator);

if (temp1 < 0)

temp1 = 0; // Satisfy the constraint

// Take random step in direction 2

temp2 = lam2 + rnorm(generator);

if (temp2 < 0)

temp2 = 0; // Satisfy the constraint

// Take random step in direction 3

temp3 = lam3 + rnorm(generator);

if (temp3 < 0)

temp3 = 0; // Satisfy the constraint

// If the new point is higher than the old one,

// then accept the new point, otherwise reject

if (dualProblem(temp1, temp2, temp3) > dualProblem(lam1, lam2, lam3))

{

lam1 = temp1;

lam2 = temp2;

lam3 = temp3;

}

}

// Print final results to the screen

cout << "Lambda's: " << lam1 << " " << lam2 << " " << lam3 << endl;

}

double dualProblem(const double lam1,

const double lam2,

const double lam3)

{

// Return the dual problem evaluated at the given points

return (15.0\*lam1 + lam2 + 2.0\*lam3 - 3.0 / 4.0\*lam1\*lam1

- 1.0 / 2.0\*lam1\*lam2 - 1.0 / 2.0\*lam1\*lam3

- 1.0 / 4.0\*lam2\*lam2 - 1.0 / 8.0\*lam3\*lam3 - 9.0);

}

The algorithm converged to , , and .

2.6 With the dual problem solved, we can plug our optimal values obtained from solving the dual problem, , into the primal problem to find the solution. So, we plug , , and into

to get and .

This result is verified by using the 'quadprog' function in R. In 'quadprog's matrix form, the primal problem is

where

Implementing this in R, we get the following result:

library(quadprog)  
K = 2\*diag (c (1, 2));  
C = -1.\*c(2, 8)  
A = matrix (0, nrow=3, ncol=2)  
A[1,] = c(1, 2)  
A[2,] = c(1, 0)  
A[3,] = c(0, 1)  
d = c(10, 0, 0)  
xHat = solve.QP(K, C, t(A), d, meq=0)$solution  
xHat

[1] 4 3

This confirms that and .