Fall 2018 Independent Study: Numerical Optimization Final Project

Alyssa Ahn

March 2019

0.1 Introduction

Optimization is associated with the efficacy of a given system or methodology in approaching a solution that should ultimately answer a question that is posed. With respect to Numerical Optimization, the topic of this Independent Study, we have observed a few numerical methods that allow for us to find minimum as well as maximum values iteratively through processes like Newton's Method and Steepest Descent. Additionally, we discussed the Linear Conjugate Gradient Method and the implications to solve Linear Systems of Equations.

In the second half of this course, we focused on Linear Programming, a type of optimization that specifically works with finding solutions based on a system of linear equations. Through Linear Programming, we explored the Simplex Method as a means to find a solution uses solely end points of the feasible region with good conditions in place. The Simplex Method offered a great background to understand Linear Programming Problems such as Min Cut and Max Flow, which are problems that are not NP hard.

Through this project, I expect to review over some general ideas about linear programming as well as complete a mini project that requires an LP formulation. Most notably, this project documentation will (1) Recalling and Providing a brief explanation of Linear Programming, (2) Explaining the k-means problem, (3) Explaining the Linear Program Formulation, and (4) Performing a Numerical Experiment that explores the integrality of solutions as discussed in [1].

Note to Reader: In my write up of parts 1 - 4, I will be calling definitions and notions from the following sources [1] and [2] and [3]. My reference section should serve as a nice list of a sources and textbooks that students with a relatively rigorous Mathematics background should be able to follow. For example, [4] was the textbook that guided class discussions and provided more rigorous proofs and justifications to discuss derivations and ideas of convergence criterion.

1 Brief Explanation of Linear Programming

Definition 1.1. Linear Programming is the study of formulating, solving and interpreting mathematical models based on variables satisfying linear equations and inequalities to maximize or minimize a linear objective function.

The construction of a Linear Programming Problem consists of decision variables, objective function, and constraints. Linear Programming problems always have a goal to either maximize or minimize some linear function composed of decision variables and this equation is called the objective function.

For example, $\zeta = c_1x_1 + c_2x_2 + ... + c_nx_n$. We observe that this is an inner product of a vector composed of the scalar c_i with the vector composed of the decision variables, denoted as x_i.

The last component of linear programming is constraints. They generally take on the form of an equality or inequality associated with a linear combination of the decision variables. In some formulations, we seek to remove inequalities by adding slack variables to ascertain equality. Often times, we will also use the constraint that our decision variables as well as our slack variables need to take on strictly positive numbers.

A general example of a linear programming formulation is as follows:

$$\begin{gathered}
\text{maximize } c_1x_1 + c_2x_2 + ... + c_nx_n \\
\text{subject to } a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \leq b_1 \\
a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \leq b_2 \\
x_1, x_2, ..., x_n \geq 0
\end{gathered}$$

When we discuss the solution that is given to us by a solver, we must also explore the topic of feasible. A solution can be understood as feasible if it satisfies all of the constraints. A solution is optimal if it attains the desired maximum or minimum. On the other hand, infeasible solution is when one constraint happens to contradict another constraint. Another example of infeasbility is an unbounded problem where a problem has feasible solutions with arbitrarily large objective values.

Remark: It is important to understand that Linear Programming graphically tends to look for "corner" solutions and often these corners correspond exactly to basic feasible solutions. We define basic feasible solution of a system $Ax = b$ as a unique solution that satisfies $x_j = 0$ for all $j \notin B$. This can be further by the understanding of the term convexity.

Definition 1.2. Convex Given any two vectors $y, z, \in \mathbb{R}^{\ltimes}$, we define a set $\big(y, z\big) = \big\{\lambda_y + (1 - \lambda)z : 0 < \lambda < 1\big\}$ If $y \neq z$, we call this set the open line segment between y and z. We call a set $F \subset \mathbb{R}^{\ltimes}$ when $y, z \in F$ implies $\big(y, z\big) \subset F$.

Theorem 1. Convexity of a Feasible Region The feasible region of any linear program is convex.

From these ideas of convexity, we can better relate these extreme points with the basic feasible solutions.

Theorem 2. Basic Feasible Solutions and Extreme Points I Consider the feasible region F defined by the system of constraints $Ax = b$ and $x \geq 0$. Suppose the columns of the matrix A span. Then the extreme points of F are exactly the basic feasible solutions of the system.

2 Explanation of the K-Means Problem

As discussed in [1], the K-Means Problem has an input of a collection of points that are in $\mathbb{R}^m$ where $m \in \mathbb{N}$. In this question, we are looking at the squared distances between any two points and this be denoted as $d(x_i, x_j)$. The purpose of this problem is to ultimately partition this collection of points into k clusters with the intention of minimizing the total squared distance. Given k clusters, we seek minimize the squared distances from each point in a cluster to the mean (a point not necessarily in the cluster) of the respective cluster. From this description, we write the K-Means problem as the following:

Definition 2.1. K-Means Problem
Let $A_1, A_2, ..., A_K$ denote a partitioning of the n points into k clusters; if $c_t = \frac{1}{|A_t|}$ minimize $_{A_1 \bigcup A_2 \bigcup ... \bigcup A_K = P} \sum_{t=1}^{k} \frac{1}{|A_t|} \sum_{x_i, x_j \in A_t} d^2(x_i, x_j)$.

In an attempt to understand this dense notated definition, $\forall A_i$ where $i = 1, 2, ..., k$ iteratively calculates the the squared distances between any two points in the A_i partition and then scales by $\frac{1}{|A_t|}$, which makes the total distance a weighted average based on the number of any given element. After it completes, this for the k clusters, it gives you the total squared distance.

With respect to the idea of geometric clustering, we seek to cluster points that are "close enough" with one another so as to minimize the distance from the cluster's mean to all the remaining points. Then for the k clusters, we hope to find k center points that will ultimately minimize the overall sum of the distances of the points in the collection.

3 LP Formulation of the K-Means Problem

As discussed in [1], the authors provide us with some good constraints that were implemented in the LP formulation and the Numerical Experiment portion of this project.

Definition 3.1. K-Means LP Formulation and Constraints
$\min_{z \in \mathbb{R}^{n \times n}} \sum\limits_{p,q,\in P} d^2(p, q)$
subject to

1. $\sum\limits_{q \in P} z_{pq} = 1 \ \forall p \in P$

2. $z_{pq} \leq z_{pp} \ \forall p, q \in P$

3. $\sum\limits_{p \in P} z_{pp} = k$

4. $z_{pq} \in \left\{0, \frac{1}{|A_p|}\right\}$

I will now explain each constraint and what it adds to the formulation. The 4th constraint underscores how Z is as matrix that indicates whether two points are in the same partition. We us the p and q indices of this matrix to indicate whether point p and q are in the same cluster. So, each entry is an indicator that takes on either the value 0 or $\frac{1}{|A_p|}$. One way to think about the second value is that it serves as a "weight" that is scaled by the distance

between those two points and contributes to the overall minimized distance that is produced by the solver.
The 3rd constraint demonstrates how the diagonal entries of Z must add up to the number of clusters, most notably k. This serves as an important constraint because each diagonal entry is in cluster with itself and will at most take on the value of 1, indicating it is in its own cluster. Or, it will take on some rational number that will be the same as the other diagonal elements that are in the same partition.
The second constraint looks at the elements in a given row of the matrix Z. Given any row, the diagonal element should be less than or equal to any other element in the row. We observe this in a similar fashion as for the third constraint. If a point is the only one in a cluster, the diagonal entry will be a 1 and thus larger than every other element in the row, which is most notably 0. There is no need for strict inequality. In the case that a few points are in the same cluster, the z matrix entry that corresponds to two points that are in the same cluster will be equal to both point's diagonal entry, indicating they are in the same set and the possibility of equality amongst elements in a row.
The first constraint emphasizes that each row of z must sum up to 1. Matrix Z serves as an indicator of connectedness between points to construct clusters. If the sum of the rows is greater than 1, then your clusters may not be mutual exclusive. If the sums of the rows is less than 1, then the row is missing parts of your cluster.
One other constraint that was not discussed in the paper was the symmetry about Matrix Z, which is shown in line 72. This detail is important because symmetry preserves the notion of distance in the Euclidean plane and connectedness in a graphical sense. We are saying that the distance between point a to point b is the same as vice versa. Additionally, a point cannot be ”semi” connected as in a is in the same partition to b and b is not in the same partition to a. So, the idea of symmetry is shown by making sure that the columns of z also add up to one.

4 Numerical Experiment: K-Means Problem

I will use this section discuss the code for the LP Formulation of the K-Means Problem. The actual code is in Python Notebook that can be run in Google Colaboratory. Note to the reader: Many of the commented out lines of code was for debugging and constructing purposes.

4.1 Python Code of the K-Means Problem

```
import numpy as np
import cvxpy as cp
import math as m
import random as rand

def metric(a, b):
  met = 0
  for i in range(len(a)):
    met += (a[i] - b[i])**2
  return met

def distance(x):
  point_count = points.shape[1]
  dimensions = points.shape[0]
  dist_mat = np.zeros((point_count,point_count))

  for i in range(point_count):
    for j in range(point_count):
      a = []
      b = []
      for k in range(dimensions):
        a.append(x[k,i])
        b.append(x[k,j])

      dist_mat[i][j] = metric(a,b)

  return dist_mat

# 2 Different Generators of Randomness

point_count = 5
point_dimensions = 4

points = np.zeros((point_dimensions, point_count))

for i in range(point_dimensions):
  for j in range(point_count):
    points[i,j] = round(rand.gauss(0,5))
    #points[i,j] = rand.randint(-10,10)

```

```python
#print(points)

d = distance(points)
#print(d)

clusters = 2 # partition/cluster number
c = d.reshape((1,point_count**2))
A = np.zeros((point_count*(point_count -1),point_count**2))
blocks = point_count
for i in range(blocks):
  for j in range(blocks):
    for k in range(blocks -1):
      if j == i:
        A[k+(blocks-1)*i,j+(i*blocks)]= -1
      elif i>j:
        if k ==j:
          A[k+(blocks-1)*i,j+(i*blocks)]= 1
      else:
        if k==j-1:
          A[k+(blocks-1)*i,j+(i*blocks)]= 1

b = np.zeros((point_count*(point_count -1),1))
b_hat = np.ones((point_count,1))
b_tilda = np.zeros((point_count,point_count**2))
for i in range(point_count):
  for j in range(point_count**2):
    if(j >= i*point_count and j < i*point_count + point_count):
      b_tilda[i,j] = 1

print(b_tilda)

z = cp.Variable((point_count, point_count), symmetric = True) #, integer = True)
z1 = cp.Variable((point_count**2, 1))
z1 = cp.reshape(z, ( point_count**2, 1))
objective = cp.Minimize(c*z1)
constraints = [A*z1 <= b, z1 >=0, b_tilda*z1 == b_hat, sum(z1[::point_count+1]) == clusters, b_hat.T * z == b_hat.T, z * b_hat == b_hat]
prob = cp.Problem(objective, constraints)

val=prob.solve()

print(val)
print(z.value)
```

```
83
84  #print(sum(z1.value[::point_count+1]))
```

4.2 Explanation of the Code

Lines 1-4: We are importing numpy, cvxpy, and some other Python libraries in order to run this code. Numpy is an important library that allows for us to code up and perform matrix and vector operations. Cvxpy is a library for Convex Optimization problems that serves as the solver. In our code, we provide the solver an objective function as well as a set of constraints and the solver provides us with a solution. Random is imported to help generate these matrices $A \in \mathbb{R}^{m \times n}$. Each column is composed of the coordinates in $\mathbb{R}^m$ and there are ultimately n points that we are trying to cluster.

Lines 6-10: The metric function intends to take in two vectors a and b and returns the squared distance between those two points.

Lines 12 - 27: The distance function is ultimately constructing A which is as matrix of squared distances. $\forall a_{i,j} \in A$ where $i, j \in \mathbb{N}, a_{i,j}$ represents the distance between point i and point j. Ultimately, this matrix that is constructed will be a symmetric matrix of $\mathbb{R}^{n \times n}$.

Lines 29-44: In this code block, we are looking at how to randomly generate the matrix composed of points. So, we have come up with two ways to do so. The first way is to make use of the Random library's Gaussian function that takes in a mean value as well as a standard deviation value and will generate values based on the Normal Distribution that are rounded to the nearest integer and placed into the matrix entry wise. The other way that is currently commented out is a simpler way of just generating random integers between two numbers.

Line 46-70: In this section, we are creating the matrices and vectors that will help us put together our constraints for the K-Means Problem. Additionally, in line 46, we are declaring the number of clusters we would like to find. The matrix A that is being formed in lines 48-60 is used to guarantee that z[i][j] when i = j is larger than any other any other point in that row. So, that matrix makes sure this condition is held for each row by comparing the element on the diagonal of z with every other element in the row and making sure that the sum of the negative diagonal element and the other element of that row are less than or equal to 1. The matrix b tilda makes sure that each row of z adds up to 1 and b hat is used to guarantee that the sums of rows and columns add to 1. By having such constraints, we are able to guarantee better and more meaningful answers.

Lines 71-84: Here we are finally putting together all the moving parts of the LP Formulation as well as calling on the solver and displaying the solution to the K-Means Problem based on the randomly generated points in $\mathbb{R}^{m \times n}$ and the number of clusters based on line 46. We define z to be the an $\mathbb{R}^{n \times n}$ symmetric matrix that will indicate to us which points are in the same partitions based on fractional weights, associated with the size of the cluster they are assigned to. We define the objective as it is an argument needed to pass in the .solve() method. Similarly, we have coded up the constraints that were discussed in part 3 and define the problem based of the objective and constraints. By using the .solve() method, CVXPy uses a blackbox that ultimately comes up with a solution based on our objective function and the constraints we presented it with. We display the minimized distance as well as the matrix that will provide us with said minimization.

4.3 Remarks and Potential Future Work

Firstly, thank you for taking the time to read this project write up! My Google Colaboratory document can be reached here and I hope you take the time to play with it.
https://colab.research.google.com/drive/1nY-ftik188XxOvuk0gWTHIxjV2XHyXQk

Secondly, I would like to discuss some of the potential solutions you may run into while randomly generating matrices and running my code for the K-Means Problem. Sometimes the solutions will be approximately close to $\frac{1}{|A_t|}$ and this may occur because of the method that the solver uses. While we did not have the time to discuss interior point methods, it is important to be aware of how they do find solutions but they are not also corner points. Additionally, from the aspect of rounding and approximating, we get arbitrarily close to that $\frac{1}{|A_t|}$ value. Another limitation of the meaningfulness of the solutions is the potential for issues with K = 2. If we think about what the Normal Distribution looks like, we seek some clustering and a peak around the mean of the distribution. With respect to my settings, the mean is 0 and the standard deviation is 5. With this type of prefixed clustering of the Normal Distribution, it is difficult to be able to produced points that will not be arbitrarily too close to one another, leading the solutions that are potentially erroneous. An example of this occurs when reading off z and realizing that the mutual exclusive nature of points in clustering has been broken –that is point 1 is in 2 partitions instead of being in 1, or when point 3 and 4 are in a partition but point 4 and 3 do not appear to be in a cluster together.

Thirdly, while it was been great to see the applications of my Analysis class as well as some ideas from Algebra and other coursework, there is definitely more work that can be done with respect to this project. It would be interesting to adjust the algorithm to try to create clusters of similar size as discussed in [1]. Additionally, it would be interesting to compare our algorithm with the K-Means Method in the scikit-learn library and to explore the limitations of that algorithm with respect to [1]. Also, it would be interesting to see how the different built-in solvers in CVXPY approach the same problem and what type of result they produce. One final point related to Numerical Analysis would be to explore the efficiency of these algorithms with large systems of equations and how we can adjust the algorithm to be more efficient.

References

[1] Pranial Awasthi, Alfonso Bandeira, Moses Charikar, Ravishankar Krishnaswamy, Soledad Villar, and Rachel Ward. Relax, No Need to Round: Integrality of Clustering Formulations. *PRIMUS*, 24(7):594–607, 2014.

[2] A.S. Lewis and D. Shmoys. Linear Programming ORIE 320 - Optimization 1, 2016.

[3] A.S. Lewis and D. Shmoys. ORIE 3310 / 5310 / 5311 - Optimization 2 course notes, 2016.

[4] J. Nocedal and S.J. Wright. *Numerical Optimization.* Springer, 2006.