Blending Linear Programming into Scholarship Allocation Decisions

University of San Francisco Baseball Quantitative Studies Team

Zachariah Zanger M.S.

Shopping at Costco with $100

In Peter Bernstein's great book *Against the God's: The Remarkable Story of Risk,* Bernstein dedicates a chapter to the advancement in Game Theory made by computer scientist John Von Neumann. In the chapter, Bernstein writes in the chapter called, *The Man Who Counted Everything,* "Von Neumann and Morganstern based *The Theory of Games and Economic Behavior* on one essential element of behavior: the winnings that will accrue to an individual who maximizes his utility-makes the best of the available tradeoffs within the constraints set by the game theory-will depend upon how much he can get if he behaves 'rationally'". Although Von Neumann is not accredited with the development of the specific principles associated with this research piece, he advanced the quantification of utility and constraint modeling for decision making. The inventor of Linear Programming was Soviet Mathematician and Economist Leonid Kantorovich. Kantorovich was tasked with developing the optimization of the state-controlled Plywood Industry, and eventually optimizing war resource allocation in World War II.

Simply put, utility is a measurement of satisfaction received from consumption of a good or service. In theory, humans attempt to maximize their utility (satisfaction) in decisions made after a mental examination of all possible decisions. In practice, it is not possible to consume an infinite amount of utility. All decision makers are bound by constraints that limit utility consumption. If a Costco shopper only brings $100 to a Costco warehouse, then that shopper faces some serious decisions to maximize their utility (think of all the ways to spend $100 at Costco). In the mind of the shopper, they assign values of utility to various items. If the shopper were planning on not cooking for their family that evening, then the $4.99 rotisserie chicken brings a large amount of utility without eating into the $100 constraint that much. On the other hand, the $30 Winter jacket at the end of spring brings little to no utility and takes up 30% of the $100 spending constraint.

Linear Programming is a way to help this shopper maximize their utility with their $100 at Costco. The framework is as follows:

Linear Programming is an automated optimization framework that maximizes a utility function, by manipulating decision variables, and abiding to stated constraints.

In an algebraic framework, an example looks something like this:

For Decision Variables X_1, X_2, and X_3

Maximize Utility Function: $(X_1Y_1) + (X_2Y_2) + (X_3Y_3)$

Subject to Constraints: $X_1+X_2+X_3 < 10$,

$X_1, X_2 > 2$,

$X_2+X_3 > 6$

Where: X_1, X_2, X_3 are variables to be optimized, and Y_1, Y_2, Y_3 are pre-defined values of utility for decision variables

Scholarships and an automated Linear program model

In our example of Costco shopping with $100, an automated Linear program would calculate every feasible combination of goods that could be purchased for less than or equal to $100. It would then select the combination that brings the consumer the highest amount of Utility, given their inputs of Utility values for different goods. Fortunately for USF Baseball, shopping at Costco with $100 is not fundamental to the operation. But the process of allocating 11.7 scholarships over 30 roster spots and maximizing Utility from those scholarships is.

Therefore, it may be wise to explore if this model structure of linear optimization/programming could be applicable to the scholarship allocation aspect of the program. This is an attempt to construct a model that would serve as a tool in the toolbox of making decisions. No model is perfect, and every model has its limitations. It is up to the user to understand the limitations. Hence, this would be a tool of many tools in the toolbox, not the only tool.

Let's try to construct a Linear program to help with a hypothetical situation where the program was considering three players to offer scholarship dollars to, and the program only had 1 scholarship to go around to the three players. The program obviously could not offer all three players 1 full scholarship, and that would be unwise, given the program only has 11.7. But again, the goal in a Linear program is to maximize Utility.

In this example of 3 players, lets assign grades on the 20-80 scale for the five tools, and note their position:

Player	Power	Hit	Speed	Glove	Arm	Position
A	55	40	45	45	50	Corner Inf
B	60	30	65	40	40	Outfielder
C	45	60	40	65	35	Catcher

Like all players, these 3 are different in the skillsets they bring. Instead of attempting to categorize them and place them in buckets, lets focus on Utility. Again, Utility is a measure of satisfaction received from consuming a good or service. For this example, say Utility is our measure of how happy we are with signing a player, and the tools that come with that player. So, we must assign Utility values to a player's tools. Out of 100:

$$Power = 40$$

$$Hit = 20$$

$$Speed = 35$$

$$Fielding = 15$$

$$Arm = 10$$

These are weights for how we value different tools. In this example, Power and Speed are the two most valued tools that bring the program the most satisfaction. I am not going to show the code here (I will include it as a separate attachment), but the PulP package in Python coding language deals with modeling linear programs. As a reminder, the model equation we are attempting to solve goes into the following format:

For Decision Variables X_1, X_2, and X_3

Maximize Utility Function: $(X_1Y_1) + (X_2Y_2) + (X_3Y_3)$

Subject to Constraints: $X_1+X_2+X_3 < 10$,

$$X_1 > 2,$$

$$X_2+X_3 > 6$$

Where: X_1, X_2, X_3 are decision variables to be optimized, and Y_1, Y_2, Y_3 are pre-defined values of utility for decision variables

We need a few things to maximize our Utility by offering these players scholarships, we first need Decision Variables. Decision Variables are what the model decides to assign amounts to. In this case, X_1, X_2, and X_3 are the amounts of scholarship we offer to the three different players.

For Y_1, Y_2, and Y_3, these are the utility scores for each player. They are calculated as:

Player A Utility Score = [(Player A Power Tool Score USF Power Tool Weight) + (Player A Hit Tool Score * USF Hit Tool Weight) + (Player A Speed Tool Score * USF Speed Tool Weight) + (Player A Fielding Tool Score * USF Fielding Tool Weight) + (Player A Arm Tool * USF Arm Tool Weight)] =* ***5775***

Player A Utility Score = [(Player A Power Tool Score USF Power Tool Weight) + (Player A Hit Tool Score * USF Hit Tool Weight) + (Player A Speed Tool Score * USF Speed Tool Weight) + (Player A Fielding Tool Score * USF Fielding Tool Weight) + (Player A Arm Tool * USF Arm Tool Weight)] =* ***5830***

Player A Utility Score = [(Player A Power Tool Score USF Power Tool Weight) + (Player A Hit Tool Score * USF Hit Tool Weight) + (Player A Speed Tool Score * USF Speed Tool Weight) + (Player A Fielding Tool Score * USF Fielding Tool Weight) + (Player A Arm Tool * USF Arm Tool Weight)] =* ***6600***

So far, we have put together an equation that will change according to what scholarship amounts the model decides:

$$Maximize: (X_1*5775) + (X_2*5830) + (X_3*6600)$$

If the model were to end here, then the model would just give all the money to Player C (X_3). Player C, given the weights of different tools, brings the program the most Utility. But the beauty of linear programming is that constraints can be modeled in and worked around. For the Soviets in World War II, guns and war machinery were obviously the most valuable things to manufacture in wartime. But if the government had not allocated any resources to food distribution, the soldiers would have starved on the battlefield. Therefore, food production acted as a constraint in their utility function.

Back to Baseball, the decision to allocate 100% of the scholarship serves as a constraint. For this example, hypothetical constraints already coded in include:

- Player A, B, or C can receive no smaller than 15% scholarship
- Player A, B, or C can receive no larger than 100% scholarship
- For Player B to sign, Player B must receive at least 35% scholarship, given he already has several offers at 20%

These are simple for example purposes. I will follow up with additional commentary on constraints. But our model is now complete. It is set as follows:

For Decision Variables X_1, X_2, and X_3

Maximize Utility Function: $(X_1*5775) +(X_2*5830) +(X_3*6600)$

Subject to Constraints: $X_1+X_2+X_3 < 1$,

$X_1, X_2, X_3 < 1$,

$X_1, X_2, X_3 > .15$,

$X_2 > .35$

Where: X_1, X_2, X_3 are Scholarship amounts to be allocated to Players A, B, and C

Model Output

Once again, the model is coded up in Python with the PulP package. I am not going to incorporate it into this writing because things could get out of control with formatting. The model output is as follows:

Player A: 15% Scholarship

Player B: 35% Scholarship

Player C: 50% Scholarship

It is important to note, that these values are the optimal X values in our utility function [(X_1*5775) +(X_2*5830)+(X_3*6600)] that result in the highest value. The model circled through every possible scholarship allocation value for the three players and arrived at these numbers.

Output Commentary / Conclusion

These are the scholarship amounts that maximize the Utility function FOR THIS EXAMPLE. Player C had the highest Utility score, so it makes intuitive sense that the model gave him 50%. The weights on player tools are customizable. If the roster needed a boost of speed or power tool, we would end up giving speed and power tools a higher weight. I need to know if there is already an existing system in place to quantify recruits and their skillsets. That way, we can begin to model in that framework into utility. We (QST) will also look into developing a Utility framework of our own so that we could target specifically the skills/toolsets we would recommend.

Except for Player B, the model does not have any constraints as to what it would take to sign a player. The model may advise offering 50% to Player C because he brings the program the most utility, but if it only takes 40%, re-run the model and see where the model allocates that other 10%!

All constraints were honored in the example above. With respect to constraints, the main limitation we (QST) need to learn to model in is if the model chooses to allocate an amount to a player, it must be above 15%. Otherwise, the model would give that player 0% if it did not deem that player worthy. In the example above, all players were supposed to receive scholarships. Constraints can get very exotic, and therefore very customizable.

Linear programming is a useful tool when solving problems of resource allocation. **If the user (USF Baseball / QST) could develop a system of quantifiable Utility values on tools/skillsets, and assign Utility values to recruits, then we could dial up the model to favor (allocate more scholarship to) the players that carry higher values of Utility and shy away from players that do not.**

```
from pulp import *
import numpy as np
import pandas as pd

###DEFINE RECRUIT VALUES###

player_a_attributes = {'Power': '55', 'Hit': '40', 'Glove': '45', 'Speed': '45', 'Arm': '50', 'Position': 'corner_infielder'}
player_b_attributes = {'Power': '60', 'Hit': '30', 'Glove': '40', 'Speed': '50', 'Arm': '40', 'Position': 'outfielder'}
player_c_attributes = {'Power': '45', 'Hit': '60', 'Glove': '60', 'Speed': '65', 'Arm': '35', 'Position': 'catcher'}

###DEFINE POSITION UTILITY WEIGHTS***
position_dict = {'catcher': '75', 'corner_infielder': '25', 'shortstop': '75', 'middle_infielder': '30', 'outfielder': '80'}

###CALCULATE UTILITIES###

def utility(power_weight, hit_weight, speed_weight, glove_weight, arm_weight):
    ###PLAYER A UTILITY CALC###
    player_a_power_utility=int(player_a_attributes.get('Power'))*int(power_weight)
    player_a_hit_utility=int(player_a_attributes.get('Hit'))*int(hit_weight)
    player_a_speed_utility=int(player_a_attributes.get('Speed'))*int(speed_weight)
    player_a_glove_utility=int(player_a_attributes.get('Glove'))*int(glove_weight)
    player_a_arm_utility=int(player_a_attributes.get('Arm'))*int(arm_weight)
    if player_a_attributes.get('Position') == 'catcher':
        player_a_position_utility = int(position_dict.get('catcher'))
    elif player_a_attributes.get('Position') == 'corner_infielder':
        player_a_position_utility = int(position_dict.get('corner_infielder'))
    elif player_a_attributes.get('Position') == 'shortstop':
        player_a_position_utility = int(position_dict.get('middle_infielder'))
    elif player_a_attributes.get('Position') == 'outfielder':
        player_a_position_utility = int(position_dict.get('outfielder'))

    player_a_utility=(player_a_position_utility+player_a_power_utility+player_a_hit_utility+player_a_speed_utility+player_a_glove_utility+player_a_arm_utility)

    ###PLAYER B UTILITY CALC###
    player_b_power_utility=int(player_b_attributes.get('Power'))*int(power_weight)
    player_b_hit_utility=int(player_b_attributes.get('Hit'))*int(hit_weight)
    player_b_speed_utility=int(player_b_attributes.get('Speed'))*int(speed_weight)
    player_b_glove_utility=int(player_b_attributes.get('Glove'))*int(glove_weight)
    player_b_arm_utility=int(player_b_attributes.get('Arm'))*int(arm_weight)
    if player_b_attributes.get('Position') == 'catcher':
        player_b_position_utility = int(position_dict.get('catcher'))
    elif player_b_attributes.get('Position') == 'corner_infielder':
        player_b_position_utility = int(position_dict.get('corner_infielder'))
    elif player_b_attributes.get('Position') == 'shortstop':
        player_b_position_utility = int(position_dict.get('middle_infielder'))
    elif player_b_attributes.get('Position') == 'outfielder':
```

```
57
58  ###PLAYER C UTILITY CALC###
59  player_c_power_utility=int(player_c_attributes.get('Power'))*int(power_weight)
60  player_c_hit_utility=int(player_c_attributes.get('Hit'))*int(hit_weight)
61  player_c_speed_utility=int(player_c_attributes.get('Speed'))*int(speed_weight)
62  player_c_glove_utility=int(player_c_attributes.get('Glove'))*int(glove_weight)
63  player_c_arm_utility=int(player_c_attributes.get('Arm'))*int(arm_weight)
64  if player_c_attributes.get('Position') == 'catcher':
65      player_c_position_utility = int(position_dict.get('catcher'))
66  elif player_c_attributes.get('Position') == 'corner_infielder':
67      player_c_position_utility = int(position_dict.get('corner_infielder'))
68  elif player_c_attributes.get('Position') == 'shortstop':
69      player_c_position_utility = int(position_dict.get('middle_infielder'))
70  elif player_c_attributes.get('Position') == 'outfielder':
71      player_c_position_utility = int(position_dict.get('outfielder'))
72
73  player_c_utility=(player_c_position_utility+player_c_power_utility+player_c_hit_utility+player_c_speed_utility+player_c_glove_utility+player_c_arm_utility)
74
75  ###BEGIN OPTIMIZATION###
76  model = LpProblem(name = "Scholarship Work", sense=LpMaximize)
77
78  ###DEFINE DECISION VARIABLES###
79  player_a=LpVariable('A', lowBound=0, cat = 'Continuous')
80  player_b=LpVariable('B', lowBound=0, cat = 'Continuous')
81  player_c=LpVariable('C', lowBound=0, cat = 'Continuous')
82
83  ###DEFINE OBJECTIVE FUNCTION###
84  model += (player_a_utility * (1+player_a)) + (player_b_utility * (1+player_b)) + (player_c_utility * (1+player_c))
85
86  ###DEFINE CONSTRAINTS###
87  model += player_a + player_b + player_c <=1
88  model += player_a >= .15
89  model += player_b >= .35
90  model += player_c >= .15
91  model += player_a <= .80
92  model += player_b <= .80
93  model += player_c <= .80
94
95  ###SOLVE MODEL###
96  model.solve()
97  print("Allocate {} Scholarship to Player A".format(player_a.varValue))
98  print("Allocate {} Scholarship to Player B".format(player_b.varValue))
99  print("Allocate {} Scholarship to Player C".format(player_c.varValue))
100 print(player_a_utility, player_b_utility, player_c_utility)
101utility(40,20,35,15,10)
102
```