Open in Colab

Math 210

Aaron Graybill

Problem Set 3

3/4/21

Problem 1.

In [20]:
```
## Dot Product:
import numpy as np

def dot(v1,v2):
  out=0
  for idx, x in np.ndenumerate(v1): #iterator location and value
    out = out + v1[idx]*v2[idx]
  return out

dot(np.array([1,2,3]),np.array([1,2,3]))
```

Out[20]:
```
14
```

In [21]:
```
## Outer Product

def outer(v1,v2):
  out=np.empty((v1.shape[0],v2.shape[0])) #define empty matrix
  for i in range(v1.shape[0]): #one iter for each row
    out[i,:]=v1[i]*v2 # multiply each row vector by the current scalar
  return out

outer(np.array([1,2,3]),np.array([1,2,3]))
```

Out[21]:
```
array([[1., 2., 3.],
       [2., 4., 6.],
       [3., 6., 9.]])
```

In [22]:
```
## Transpose
#This code is quite cute i think

def transp(a) :
  out=np.empty((a.shape[1],a.shape[0])) #make empty matrix of right dims
  for idx, x in np.ndenumerate(a): #iterator location and value
    out[idx[1],idx[0]] = x #give current value to opposite index
  return out
```

In [23]:
```
# You can run this to test your codes
v1 = np.array([1,4,5,6])
v2 = np.array([2,1,3,8])
result1 = dot(v1,v2)
print(result1)
```

```
print()
result2 = outer(v1,v2)
print(result2)
print()
a = np.array([[1,4,6,7],[2,3,9,8]])
print(a)
#print()
m = transp(a)
print(m)
```

```
69

[[ 2.  1.  3.  8.]
 [ 8.  4. 12. 32.]
 [10.  5. 15. 40.]
 [12.  6. 18. 48.]]

[[1 4 6 7]
 [2 3 9 8]]
[[1. 2.]
 [4. 3.]
 [6. 9.]
 [7. 8.]]
```

Problem 2.

a.

In [24]:

```
# Problem 2a -- implements the Target step of SimplexBF
#      Input: tableau "a" (just the numbers, no labels)
#      Output: the column number where the largest c is located
#                 except if all c's are <= 0, then return -1
#               (in giving the column a number, use human numbering, i.e., first
#
def target(a) :
    nrows,ncols = a.shape #define row and column nums
    if np.max(a[nrows-1,:-1])<0: #if highest val <0, return -1 (don't check cons
        return -1
    else:
      bestcol = np.argmax(a[nrows-1,:])+1 #return the maximizing place +1 (for r
      return bestcol
```

b.

In [25]:

```
# Problem 2b -- implements the Candidates/Select step of SimplexBF
#      Two Inputs: tableau "a" (just the numbers, no labels)
#               and column number "pivcolnum" where the Target lives
#      Output: the row number for the pivot that SimplexBF would choose
#                 except if all aij's in this column are <= 0, then return -1
#
#     NOTE: for both the input column number and the output row number,
#               please use human numbering, i.e., first column or row is 1 not 0
#
def select(a,pivcolnum) :
    bestratio=np.inf #initial value that anything will be better than
    nrows = a.shape[0]    # use the shape function to determine number of rows a
    ncols = a.shape[1]
    rownum="Something is wrong if you're seeing this"
    if max(a[:-1,pivcol-1])<=0:#prevents checking the obj function
```

```
        rownum =-1
      else:
        for i in range(nrows-1): #don't check obj function
          ratio=a[i,ncols-1]/a[i,pivcolnum-1]
          if ratio<bestratio:
            bestratio=ratio
            rownum=i+1

# You fill in the rest.  In your code, define "rownum" to equal the row number f
      return rownum;
```

In [26]:
```
# This cell could be useful for you to test the two functions above
import numpy as np
a = np.array([[10,5,150],    # This is the bakery-problem "a"
         [4,3,80],           # target should return column = 1
         [30,20,0]])         # and then select should return row = 1
                             # If you swap the 30 and 20, column and row should c


pivcol=target(a)
print(pivcol)
pivrow=select(a,pivcol)
print("pivot on row ",pivrow," and column ",pivcol)
```

```
1
pivot on row  1  and column  1
```

Problem 3.

a.

Reading off of the tableau we have that $x = t_1 = 0$ and that $y = t_2 = 1$ with the objective function also equal to one.

b.

Setting $x = M$ and $t_1 = 0$, the first line of the pivoted tableau requires that:

$$-\frac{2}{3}M + 0 - 1 = -y$$
$$0 \cdot M + \frac{1}{2} \cdot 0 - 1 = -t_2$$

which gives: $y = \frac{2}{3}M + 1$ and $t_2 = 1$. This is in the feasible set because this would never require y to be negative and $t_2 = 1 \geq 0$.

The objective function would then be:

$$-\frac{1}{3}M + \frac{2}{3}M + 1 + 0 = \frac{1}{3}M + 1$$

Therefore we can increase the objective function by taking larger and larger Ms and we already showed that any $M \geq 0$ has a corresponding point in the constraint set. The problem has no solution and the objective function is unbounded.

c.

Take $x = M, t_1 = 0$ and solve for the corresponding y reading off of the tableau we have:

$$AM + 1 \cdot 0 - C = -y$$
$$BM + \frac{1}{2} \cdot 0 - D = -t_2$$

Solving the equations gives: $y = -AM + C$ which cannot be negative because $A \leq 0 \leq M, C$. The second equation gives: $t_2 = -BM + D$ which must be greater than zero because $B \leq 0 \leq M, D$. Therefore, the choice remains feasible.

Plugging into the objective function gives:

$$EM + 1$$

Which is increasinig in M because $E > 0$ and any M is feasible as previously shown, so the objective function is unbounded in the positive direction, so there is no satisfactory solution.

Problem 4.

a.

Value of b_i. First note that if $i = k$ then $b_i^{new} = \frac{b_k}{a_{kj}}$ because it is in the same row. Otherwise, $b_i^{new} = \frac{a_{kj}b_i - a_{ij}b_k}{a_{kj}} = b_i - \frac{a_{ij}b_k}{a_{kj}}$.

b.

Note: `SimplexBF` requires that the `select` ed $a_{kj} > 0$. So b_k^{new} is greater than zero because $b_k > 0$. That takes care of $i = k$ (in both cases)

First take $i = k$. Then $b_i^{new} = \frac{b_k}{a_{kj}}$ and since $a_{kj} > 0$ and $b_k \geq 0$ $b_i^{new} \geq 0$ Next take $i \neq k$ and etting $a_{ij} < 0$, we know that both bs are positive. Therefore the whole expression is positive because we are subtracting a negative number.

c.

`SimplexBF` requires that the `select` ed $a_{kj} > 0$. So b_k^{new} is greater than zero because $b_k > 0$. That takes care of $i = k$

Applying similar logic to $i \neq k$, for $b_i^{new} \geq 0$ we need:

$$b_i - \frac{a_{ij}b_k}{a_{kj}} \geq 0 \tag{1}$$

$$b_i \geq \frac{a_{ij}b_k}{a_{kj}} \tag{2}$$

$$\frac{b_i}{a_{ij}} \geq \frac{b_k}{a_{kj}} \tag{3}$$

And we can do those manipulations without flipping the inequality because a_{ij} is given to be greater than zero.

This is convenient because the `select` step of the process guarantees that the ratio on the RHS of the last inequality is the minimum of all of such ratios. This completes the proof, and it's almost like the algorithm was designed to exactly this, hehe.

Problem 5.

The tableau as it stands is:

In [27]:
```
#@title
def print_tableau(a,indep_names,dep_names):
#
# Given matrix "a" and lists of variables names "indep_names" and "dep_names",
# this function prints the matrix and labels in standard tableau format
# (including adding the -1, the minus signs in the last column, and labeling the
#
# First, check the inputs: indep_names should be one shorter than the number of
#                          dep_names should be one shorter than the number of ro
#
    nrows = a.shape[0]     # use the shape function to determine number of rows a
    ncols = a.shape[1]
    nindep = len(indep_names)
    ndep = len(dep_names)
    if nindep != ncols-1:
        print("WARNING: # of indep vbles should be one fewer than # columns of m
    if ndep != nrows-1:
        print("WARNING: # of dep vbles should be one fewer than # rows of matrix
# Now do the printing (uses a variety of formatting techniques in Python)
    for j in range(ncols-1):                      # Print the independent variable
        print(indep_names[j].rjust(10),end="")    # rjust(10) makes fields 10 wide
                                                  #    the end command prevents ne
    print("        -1")                           # Tack on the -1 at the end of t
    for i in range(nrows-1):
        for j in range(ncols):                    # Print all but the last row of
            print("%10.3f" % a[i][j],end="") # The syntax prints in a field 10 w
        lab = "= -" + dep_names[i]
        print(lab.rjust(10))
    for j in range(ncols):
        print("%10.3f" % a[nrows-1][j],end="")    # Print the last row of the matr
    lab = "= obj"
    print(lab.rjust(10))
    print(" ")    # Put blank line at bottom


def pivot(a,pivrow,pivcol,indep_names,dep_names) :
#
```

```
# Given matrix "a", a row number "pivrow" and column number "pivcol",
#  and lists of variable names "indep_names" and "dep_names", this
#  function does three things:
#    (1) outputs the new version of the matrix after a pivot,
#    (2) updates the lists of variable names post-pivot
#    (3) prints the new matrix, including labels showing the variable names
#
# First, check the inputs: indep_names should be one shorter than the number of
#                          dep_names should be one shorter than the number of ro
#                          you should not be pivoting on the last row or last co
#
    a = a.astype(float)   # make sure entries are treated as floating point numb
    nrows = a.shape[0]    # use the shape function to determine number of rows a
    ncols = a.shape[1]
    nindep = len(indep_names)
    ndep = len(dep_names)
    if nindep != ncols-1:
        print("WARNING: # of indep vbles should be one fewer than # columns of m
    if ndep != nrows-1:
        print("WARNING: # of dep vbles should be one fewer than # rows of matrix
    if pivrow > nrows-1 or pivcol > ncols-1:
        print("WARNING: should not pivot on last row or column")
    newa = a.copy()       # make a copy of A, to be filled in below with result
    p = a[pivrow-1][pivcol-1]   # identify pivot element
    newa[pivrow-1][pivcol-1] = 1/p   # set new value of pivot element
    # Set entries in p's row
    for j in range(ncols):
        if j != pivcol-1:
            newa[pivrow-1][j]=a[pivrow-1][j]/p;
    # Set entries in p's column
    for i in range(nrows):
        if i != pivrow-1:
            newa[i][pivcol-1]=-a[i][pivcol-1]/p;
    # Set all other entries
    for i in range(nrows):
        for j in range(ncols):
            if i != pivrow-1 and j != pivcol-1:
                r = a[i][pivcol-1]
                q = a[pivrow-1][j]
                s = a[i][j]
                newa[i][j]=(p*s-q*r)/p
    # Now swap the variable names
    temp = indep_names[pivcol-1]
    indep_names[pivcol-1]=dep_names[pivrow-1]
    dep_names[pivrow-1]=temp
    print_tableau(newa,indep_names,dep_names) # Print the matrix with updated la
    return newa;
```

In [28]:

```
import numpy as np
a=np.array([[-1,-1,-2],[1,-2,0],[-2,1,1],[-3,1,0]])
indep_names=["x","y"]
dep_names=["t1","t2","t3"]
print_tableau(a,indep_names,dep_names)
```

```
         x         y        -1
    -1.000    -1.000    -2.000     = -t1
     1.000    -2.000     0.000     = -t2
    -2.000     1.000     1.000     = -t3
    -3.000     1.000     0.000     = obj
```

We only have one negative b_i, so we target row 1. I will then choose x as the column with the pivot. That leaves both the a_{11} and a_{21} as candidate solutions, the others in this column are negative so cannot be candidates. We now compute the ratios to find the minimum. $r_{11} = 2$ and $r_{21} = 0$. Zero is smaller than 2, so we will pivot on a_{21}. Doing that gives:

In [29]:
```
a2=pivot(a,2,1,indep_names,dep_names)
```

```
     t2         y        -1
  1.000    -3.000    -2.000      = -t1
  1.000    -2.000     0.000       = -x
  2.000    -3.000     1.000      = -t3
  3.000    -5.000     0.000      = obj
```

We then must select row 1 because it is again the only negative. As such we can only select column y because it is the only one less than 0. This makes a_{12} a candidate. We need not compute any ratios because there are no positive values in the column below a_{12}. Therefore, a_{12} is our pivot:

In [30]:
```
a3=pivot(a2,1,2,indep_names,dep_names)
```

```
     t2        t1        -1
 -0.333    -0.333     0.667       = -y
  0.333    -0.667     1.333       = -x
  1.000    -1.000     3.000      = -t3
  1.333    -1.667     3.333      = obj
```

Now we're in basic feasible. Let's run the `BasicFeasible` algorithm. First let's target t_2, column 1, because it is the only positive c_i. The corresponding rations are:

$$r_{11} = \text{not computed}, a_{11} < 0 \tag{4}$$
$$r_{21} = 4 \tag{5}$$
$$r_{31} = 3 \tag{6}$$

Since r_{31} is the smallest, a_{31} our pivot:

In [31]:
```
a4=pivot(a3,3,1,indep_names,dep_names)
```

```
     t3        t1        -1
  0.333    -0.667     1.667       = -y
 -0.333    -0.333     0.333       = -x
  1.000    -1.000     3.000      = -t2
 -1.333    -0.333    -0.667      = obj
```

The problem is solved, we set $t_1 = t_3 = 0$ and $y = \frac{5}{3}, x = \frac{1}{3}, t_2 = 3$ with the objective function attaining $\frac{2}{3}$.

Problem 6.

The independent variables in this problem are the number of each final product produced. Let p be the quantity of pizza. Let s be quantity soup. And let y be stuffed peppers quantity. (grr no good letter). Tackling the constraints in order of their appearance in the problem, we have:

$$.5p + .3s + .1y \leq 30 \tag{7}$$
$$.3p + .1s + .8y \leq 20 \tag{8}$$
$$.1p + .4s + .05y \leq 40 \tag{9}$$
$$1.5p + .1s + .25y \leq 40 \tag{10}$$
$$-p - y \leq -30 \tag{11}$$

Now the interesting part comes in the objective function which is not as straightforward as in previous cases. So we make profit from selling final products in the following way: $\pi_{fg}(p, s, y) = 5p + 1s + 1.5y$. However we can also make profit from leftovers in the following manner. $\pi_l(p, s, y) = 2\left(30 - (.5p + .3s + .1y)\right) = 60 - p - .6s - .2y$. This function takes the remaning tomatoes and multiplies them by 2 to get the profit. Combining these two profit functions gives:

$$\pi(p, s, y) = 5p + 1s + 1.5y + 60 - p - .6s - .2y = 4p + .4s + 1.3y + 60$$

Let's tableau-ify this because everything is now in the right form:

In [32]:
```
b=np.array([
    [.5,.3,.1,30],
    [.3,.1,.8,20],
    [.1,.4,.05,40],
    [1.5,.1,.25,40],
    [-1,0,-1,-30],
    [4,.4,1.3,-60]])
indep_names=["pizza","soup","stuffed"]
dep_names=["tomato","pepper","kale","time","Hometown"]
print_tableau(b,indep_names,dep_names)
```

```
    pizza      soup   stuffed        -1
    0.500     0.300     0.100    30.000 = -tomato
    0.300     0.100     0.800    20.000 = -pepper
    0.100     0.400     0.050    40.000   = -kale
    1.500     0.100     0.250    40.000   = -time
   -1.000     0.000    -1.000   -30.000= -Hometown
    4.000     0.400     1.300   -60.000     = obj
```

We are not in Basic Feasible, so we should first select row 5 because it is the $b < 0$ in the lowest row. Then I will select to pivot on column 1, (because it has the largest coefficient in the objective function). Since there are no numbers below a_{51}, we must pivot on a_{51}. Okay so pivot:

In [33]:
```
b2=pivot(b,5,1,indep_names,dep_names)
```

```
 Hometown      soup   stuffed        -1
    0.500     0.300    -0.400    15.000 = -tomato
    0.300     0.100     0.500    11.000 = -pepper
    0.100     0.400    -0.050    37.000   = -kale
    1.500     0.100    -1.250    -5.000   = -time
   -1.000    -0.000     1.000    30.000  = -pizza
    4.000     0.400    -2.700  -180.000     = obj
```

Using similar logic our candidates are a_{43} and a_{53}, computing their ratios, we have: $r_{43} = 4, r_{53} = 30$. So we pivot on a_{43} giving:

In [34]:
```
b3=pivot(b2,4,3,indep_names,dep_names)
```

```
Hometown      soup      time        -1
   0.020     0.268    -0.320    16.600 = -tomato
   0.900     0.140     0.400     9.000 = -pepper
   0.040     0.396    -0.040    37.200   = -kale
  -1.200    -0.080    -0.800     4.000= -stuffed
   0.200     0.080     0.800    26.000  = -pizza
   0.760     0.184    -2.160  -169.200     = obj
```

We are now basic feasible. So I will target column 1 because it has the largest coefficients. Computing the ratios gives:

In [35]:
```
print(b3[:-1,3]/b3[:-1,0])
```

```
[830.          10.          930.          -3.33333333 130.        ]
```

I do not consider the negative values, so my lowest ratio comes from row 2. Therefore we pivot on a_{21}:

In [36]:
```
b4=pivot(b3,2,1,indep_names,dep_names)
```

```
  pepper      soup      time        -1
  -0.022     0.265    -0.329    16.400 = -tomato
   1.111     0.156     0.444    10.000= -Hometown
  -0.044     0.390    -0.058    36.800   = -kale
   1.333     0.107    -0.267    16.000= -stuffed
  -0.222     0.049     0.711    24.000  = -pizza
  -0.844     0.066    -2.498  -176.800     = obj
```

We must target soup, column two, so we compute the ratios:

In [37]:
```
print(b4[:-1,3]/b4[:-1,1])
```

```
[ 61.91275168  64.28571429  94.41277081 150.         490.90909091]
```

The minimal ratio is attained in row 1, so let's pivot on a_{12}:

In [38]:
```
b5=pivot(b4,1,2,indep_names,dep_names)
```

```
  pepper    tomato      time        -1
  -0.084     3.775    -1.242    61.913   = -soup
   1.124    -0.587     0.638     0.369= -Hometown
  -0.012    -1.471     0.426    12.668   = -kale
   1.342    -0.403    -0.134     9.396= -stuffed
  -0.218    -0.185     0.772    20.973  = -pizza
  -0.839    -0.248    -2.416  -180.872     = obj
```

And we're done, the maximal profit is attained with 0 peppers remaining, 0 tomatoes, 0 time remaining. Creating 61.913 soups, 9.396 stuffed peppers, and 20.973 pizzas. We will exceed the Hometown market constraint by .369 and we will leave 12.668 kales unsold. No tomatoes are sold as leftovers.

All told, the company attains $180.862 in profit.