Question 15.2

In the videos, we saw the "diet problem". (The diet problem is one of the first large-scale optimization

problems to be studied in practice. Back in the 1930's and 40's, the Army wanted to meet the nutritional

requirements of its soldiers while minimizing the cost.) In this homework you get to solve a diet problem with real data. The data is given in the file `diet.xls`.

1. Formulate an optimization model (a linear program) to find the cheapest diet that satisfies the maximum and minimum daily nutrition constraints, and solve it using PuLP. Turn in your code and the solution. (The optimal solution should be a diet of air-popped popcorn, poached eggs, oranges, raw iceberg lettuce, raw celery, and frozen broccoli. UGH!)
2. Please add to your model the following constraints (which might require adding more variables) and solve the new model:
   a. If a food is selected, then a minimum of 1/10 serving must be chosen. (Hint: now you will need two variables for each food *i*: whether it is chosen, and how much is part of the diet. You'll also need to write a constraint to link them.)
   b. Many people dislike celery and frozen broccoli. So at most one, but not both, can be selected.
   c. To get day-to-day variety in protein, at least 3 kinds of meat/poultry/fish/eggs must be selected. [If something is ambiguous (e.g., should bean-and-bacon soup be considered meat?), just call it whatever you think is appropriate – I want you to learn how to write this type of constraint, but I don't really care whether we agree on how to classify foods!]

If you want to see what a more full-sized problem would look like, try solving your models for the file `diet_large.xls`, which is a low-cholesterol diet model (rather than minimizing cost, the goal is to minimize cholesterol intake). I don't know anyone who'd want to eat this diet – the optimal solution includes dried chrysanthemum garland, raw beluga whale flipper, freeze-dried parsley, etc. – which shows why it's necessary to add additional constraints beyond the basic ones we saw in the video!

[**Note**: there are many optimal solutions, all with zero cholesterol, so you might get a different one. It probably won't be much more appetizing than mine.]

Approach taken

1) After importing the data set, create a master food dictionary with reference of column#1 Food name vs. each of the nutrient to find "j" nutrients for each "I" food.
2) Similarly generate the cost and min and max values from the imported dataset
3) This problem is a minimization problem (find the *lowest* cost), use "LpMinimize" as the second parameter to setup this model
4) With the model, One variable is the name of each food and set up criteria with Lower limit of each variable is 0, since we can't eat negative amounts of the food.
5) Define objective function to minimize the cost of the food
6) Included additional constraints for add variety to the food

- ⇨ # If a food is eaten, must eat at least 0.1 serving
- ⇨ Include at most 1 of celery and frozen broccoli
- ⇨ # At least 3 kinds of meat/poultry/fish/eggs

7) Solve the problem and print the results

Homework 11 Question 2 using PuLP

---------- Import modules -----------

```
# import PuLP and pandas modules


from pulp import *

import pandas as pd
```

---------- Read data ----------

```
data = pd.read_excel("diet.xls", header=0)  # read all data


dataTable = data[0:64]  # These are datarows, last two are min and max values

dataTable = dataTable.values.tolist()  # Convert dataframe to list


nutrientNames = list(data.columns.values)  # column headers

#print(nutrientNames)
```

create master foods dictionary

```
foods = [x[0] for x in data]
```

```
#print('##FOODS##',foods)
calories = dict([(x[0], float(x[3])) for x in dataTable])
#print('##CALORIES ###',calories)
cholesterol = dict([(x[0], float(x[4])) for x in dataTable])
totalFat = dict([(x[0], float(x[5])) for x in dataTable])
sodium = dict([(x[0], float(x[6])) for x in dataTable])
carbs = dict([(x[0], float(x[7])) for x in dataTable])
fiber = dict([(x[0], float(x[8])) for x in dataTable])
protien = dict([(x[0], float(x[9])) for x in dataTable])
vitaminA = dict([(x[0], float(x[10])) for x in dataTable])
vitaminC = dict([(x[0], float(x[11])) for x in dataTable])
calcium = dict([(x[0], float(x[12])) for x in dataTable])
iron = dict([(x[0], float(x[13])) for x in dataTable])

#minVal = [1500, 30, 20, 800, 130, 125, 60, 1000, 400, 700, 10]
#maxVal = [2500, 240, 70, 2000, 450, 250, 100, 10000, 5000, 1500, 40]

#Dynamically read min and max values
minVal = data[65:66].values.tolist()  # minimum nutrient values
maxVal = data[66:67].values.tolist()  # maximum nutrient values

# ---------- Extract individual vectors of data ----------
#
# python "dict" structure to dynamically create nutrients and cost for each food
# which is more efficient
```

```
foods = [j[0] for j in dataTable]  # list of food names

cost = dict([(j[0], float(j[1])) for j in dataTable])  # cost for each food


nutrients = []

for i in range(0, 11):  # for loop running through each nutrient: 11 times starting with 0

    nutrients.append(dict([(j[0], float(j[i + 3])) for j in dataTable]))  # amount of nutrient i in food j


# ------------ Create a new LP Problem ------------
#
# This problem is a minimization problem (find the *lowest* cost), so "LpMinimize" is the second parameter.


prob = LpProblem('Foodoptimization', LpMinimize)  # 2 parameters: "name" and "sense"


# ------------ Define the variables ---------------
#
# One variable (we chose the name "foodVars") for each food.

# Lower limit of each variable is 0, since we can't eat negative amounts of anything.


foodVars = LpVariable.dicts("Foods", foods, 0)

foodVars_selected = LpVariable.dicts("food_select", foods, 0, 1,

                                     LpBinary)  # Create binary integer variables for whether a food is eaten


# ------------ Create objective function ------------
#
```

```
# Note that the first function we add is taken to be the objective function
prob += lpSum([cost[f] * foodVars[f] for f in foods]), 'TotalCost'

# ---------- Add constraints for each nutrient ----------

for i in range(0, 11):  # for loop running through each nutrient: 11 times starting with 0
    prob += lpSum([nutrients[i][j] * foodVars[j] for j in foods]) >= minVal[0][i + 3], 'minnutrient ' + nutrientNames[
        i]
    prob += lpSum([nutrients[i][j] * foodVars[j] for j in foods]) <= maxVal[0][i + 3], 'maxnutrient ' + nutrientNames[
        i]

# ---------- Adding additional constraints ----------

# CONSTRAINT A

# If a food is eaten, must eat at least 0.1 serving

for food in foods:
    prob += foodVars[food] >= 0.1 * foodVars_selected[food]

# If any of a food is eaten, its binary variable must be 1

for food in foods:
    prob += foodVars_selected[food] >= foodVars[food] * 0.0000001
```

```
# CONSTRAINT B

# Include at most 1 of celery and frozen brocolli

prob += foodVars_selected['Frozen Broccoli'] + foodVars_selected['Celery, Raw'] <= 1

# CONSTRAINT C

# At least 3 kinds of meat/poultry/fish/eggs

prob += foodVars_selected['Roasted Chicken'] + foodVars_selected['Poached Eggs'] \
      + foodVars_selected['Scrambled Eggs'] + foodVars_selected['Bologna,Turkey'] \
      + foodVars_selected['Frankfurter, Beef'] + foodVars_selected['Ham,Sliced,Extralean'] \
      + foodVars_selected['Kielbasa,Prk'] + foodVars_selected['Pizza W/Pepperoni'] \
      + foodVars_selected['Hamburger W/Toppings'] \
      + foodVars_selected['Hotdog, Plain'] + foodVars_selected['Pork'] \
      + foodVars_selected['Sardines in Oil'] + foodVars_selected['White Tuna in Water'] \
      + foodVars_selected['Chicknoodl Soup'] + foodVars_selected['Splt Pea&Hamsoup'] \
      + foodVars_selected['Vegetbeef Soup'] + foodVars_selected['Neweng Clamchwd'] \
      + foodVars_selected['New E Clamchwd,W/Mlk'] + foodVars_selected['Beanbacn Soup,W/Watr'] >= 3

# ---------- Solve the optimization problem ----------
```

```
prob.solve()


# ---------- Print the output in a readable format ----------


print()
print("---------The solution to the diet problem is----------")
for var in prob.variables():
    if var.varValue > 0 and "food_select" not in var.name:  # Print non binary variables
        print(str(var.varValue) + " units of " + str(var).replace('Foods_', ''))
print()
print("Total cost of food = $%.2f" % value(prob.objective))
```

Results

```
Welcome to the CBC MILP Solver
Version: 2.10.3
Build Date: Dec 15 2019

command line - C:\Users\h7853\PycharmProjects\First\venv\lib\site-packages\pulp\apis\..\solverdir\cbc\win\64\cbc.exe
C:\Users\h7853\AppData\Local\Temp\f2a75602fd0646c2bcaa8df9c243997f-pulp.mps timeMode elapsed branch printingOptions all solution
C:\Users\h7853\AppData\Local\Temp\f2a75602fd0646c2bcaa8df9c243997f-pulp.sol (default strategy 1)
At line 2 NAME         MODEL
At line 3 ROWS
At line 157 COLUMNS
At line 1821 RHS
```

At line 1974 BOUNDS

At line 2039 ENDATA

Problem MODEL has 152 rows, 128 columns and 1471 elements

Coin0008I MODEL read with 0 errors

Option for timeMode changed from cpu to elapsed

Continuous objective value is 4.38006 - 0.00 seconds

Cgl0003I 0 fixed, 0 tightened bounds, 14 strengthened rows, 0 substitutions

Cgl0004I processed model has 141 rows, 128 columns (64 integer (64 of which binary)) and 874 elements

Cbc0038I Initial state - 8 integers unsatisfied sum - 1.63213

Cbc0038I Pass 1: suminf. 0.52908 (2) obj. 5.37039 iterations 60

Cbc0038I Solution found of 5.37039

Cbc0038I Relaxing continuous gives 5.37039

Cbc0038I Before mini branch and bound, 55 integers at bound fixed and 53 continuous

Cbc0038I Full problem 141 rows 128 columns, reduced to 31 rows 20 columns

Cbc0038I Mini branch and bound improved solution from 5.37039 to 4.51296 (0.04 seconds)

Cbc0038I Round again with cutoff of 4.50011

Cbc0038I Pass 2: suminf. 0.61600 (4) obj. 4.50011 iterations 9

Cbc0038I Pass 3: suminf. 0.55610 (2) obj. 4.50011 iterations 64

Cbc0038I Solution found of 4.50011

Cbc0038I Infeasible when relaxing continuous!

Cbc0038I Before mini branch and bound, 56 integers at bound fixed and 54 continuous

Cbc0038I Full problem 141 rows 128 columns, reduced to 29 rows 18 columns

Cbc0038I Mini branch and bound did not improve solution (0.05 seconds)

Cbc0038I Round again with cutoff of 4.47699

Cbc0038I Pass 4: suminf. 0.65246 (4) obj. 4.47699 iterations 0

Cbc0038I Pass 5: suminf. 0.48977 (2) obj. 4.47699 iterations 64

Cbc0038I Solution found of 4.47699

Cbc0038I Infeasible when relaxing continuous!

Cbc0038I Before mini branch and bound, 56 integers at bound fixed and 54 continuous

Cbc0038I Full problem 141 rows 128 columns, reduced to 29 rows 18 columns

Cbc0038I Mini branch and bound did not improve solution (0.07 seconds)

Cbc0038I Round again with cutoff of 4.44926

Cbc0038I Pass 6: suminf. 0.69620 (4) obj. 4.44926 iterations 0

Cbc0038I Pass 7: suminf. 0.41018 (2) obj. 4.44926 iterations 64

Cbc0038I Solution found of 4.44926

Cbc0038I Infeasible when relaxing continuous!

Cbc0038I Before mini branch and bound, 56 integers at bound fixed and 54 continuous

Cbc0038I Full problem 141 rows 128 columns, reduced to 29 rows 18 columns

Cbc0038I Mini branch and bound did not improve solution (0.09 seconds)

Cbc0038I Round again with cutoff of 4.42985

Cbc0038I Pass 8: suminf. 0.72682 (4) obj. 4.42985 iterations 0

Cbc0038I Pass 9: suminf. 0.35447 (2) obj. 4.42985 iterations 62

Cbc0038I Solution found of 4.42985

Cbc0038I Infeasible when relaxing continuous!

Cbc0038I Before mini branch and bound, 56 integers at bound fixed and 54 continuous

Cbc0038I Full problem 141 rows 128 columns, reduced to 29 rows 18 columns

Cbc0038I Mini branch and bound did not improve solution (0.11 seconds)

Cbc0038I Round again with cutoff of 4.41173

Cbc0038I Pass 10: suminf. 0.75539 (4) obj. 4.41173 iterations 0

Cbc0038I Pass 11: suminf. 0.30248 (2) obj. 4.41173 iterations 62

Cbc0038I Solution found of 4.41173

Cbc0038I Infeasible when relaxing continuous!

Cbc0038I Before mini branch and bound, 56 integers at bound fixed and 54 continuous

Cbc0038I Full problem 141 rows 128 columns, reduced to 29 rows 18 columns

Cbc0038I Mini branch and bound did not improve solution (0.13 seconds)

Cbc0038I After 0.13 seconds - Feasibility pump exiting with objective of 4.41173 - took 0.12 seconds

Cbc0012I Integer solution of 4.5216303 found by feasibility pump after 0 iterations and 0 nodes (0.14 seconds)

Cbc0012I Integer solution of 4.5175584 found by DiveCoefficient after 0 iterations and 0 nodes (0.15 seconds)

Cbc0038I Full problem 141 rows 128 columns, reduced to 29 rows 18 columns

Cbc0012I Integer solution of 4.5125434 found by DiveCoefficient after 29 iterations and 0 nodes (0.19 seconds)

Cbc0031I 4 added rows had average density of 18.75

Cbc0013I At root node, 4 cuts changed objective from 4.3845731 to 4.5125434 in 6 passes

Cbc0014I Cut generator 0 (Probing) - 2 row cuts average 2.0 elements, 55 column cuts (55 active) in 0.005 seconds - new frequency is 1

Cbc0014I Cut generator 1 (Gomory) - 8 row cuts average 33.1 elements, 0 column cuts (0 active) in 0.004 seconds - new frequency is 1

Cbc0014I Cut generator 2 (Knapsack) - 2 row cuts average 24.5 elements, 0 column cuts (0 active) in 0.002 seconds - new frequency is 1

Cbc0014I Cut generator 3 (Clique) - 0 row cuts average 0.0 elements, 0 column cuts (0 active) in 0.000 seconds - new frequency is -100

Cbc0014I Cut generator 4 (MixedIntegerRounding2) - 1 row cuts average 35.0 elements, 0 column cuts (0 active) in 0.000 seconds - new frequency is -100

Cbc0014I Cut generator 5 (FlowCover) - 0 row cuts average 0.0 elements, 0 column cuts (0 active) in 0.002 seconds - new frequency is -100

Cbc0014I Cut generator 6 (TwoMirCuts) - 10 row cuts average 34.2 elements, 0 column cuts (0 active) in 0.004 seconds - new frequency is 1

Cbc0001I Search completed - best objective 4.512543398463767, took 29 iterations and 0 nodes (0.19 seconds)

Cbc0035I Maximum depth 0, 1 variables fixed on reduced cost

Cuts at root node changed objective from 4.38457 to 4.51254

Probing was tried 6 times and created 57 cuts of which 0 were active after adding rounds of cuts (0.005 seconds)

Gomory was tried 6 times and created 8 cuts of which 0 were active after adding rounds of cuts (0.004 seconds)

Knapsack was tried 6 times and created 2 cuts of which 0 were active after adding rounds of cuts (0.002 seconds)

Clique was tried 6 times and created 0 cuts of which 0 were active after adding rounds of cuts (0.000 seconds)

MixedIntegerRounding2 was tried 6 times and created 1 cuts of which 0 were active after adding rounds of cuts (0.000 seconds)

FlowCover was tried 6 times and created 0 cuts of which 0 were active after adding rounds of cuts (0.002 seconds)

TwoMirCuts was tried 6 times and created 10 cuts of which 0 were active after adding rounds of cuts (0.004 seconds)

ZeroHalf was tried 1 times and created 0 cuts of which 0 were active after adding rounds of cuts (0.004 seconds)

Result - Optimal solution found

Objective value: 4.51254340

Enumerated nodes: 0

Total iterations: 29

Time (CPU seconds): 0.20

Time (Wallclock seconds): 0.20

Option for printingOptions changed from normal to all

Total time (CPU seconds): 0.21 (Wallclock seconds): 0.21

---------The solution to the diet problem is----------

42.399358 units of Celery,_Raw

0.1 units of Kielbasa,Prk

82.802586 units of Lettuce,Iceberg,Raw

3.0771841 units of Oranges

1.9429716 units of Peanut_Butter

0.1 units of Poached_Eggs

13.223294 units of Popcorn,Air_Popped

0.1 units of Scrambled_Eggs

Total cost of food = $4.51

Process finished with exit code 0