ABA Finance

Data Science Challenge

January 16, 2020

Purpose

This notebook (written in Python 3) is intended to address the Data Science challenge presented by ABA Finance

Candidate

Andrea Chiappo

Problem

Is it possible to fill 10 backpacks with 20 different size boxes?

Variables:
- two different box types: type A and type B
- boxes have different sizes
- backpacks capacity differs
- box timing: boxes carry time label

Solution

I propose two possible solutions to this Data Science challenge: a methodic one and a pragmatic one.

Methodic solution
The problem at hand resembles the so-called *Knapsack Problem* with three additional levels of complexity:

- the presence of 10 backpacks with different capacity
- the presence of two classes of boxes (typeA and typeB)
- the presence of a time order label on the boxes

From a strict interpretation of the problem statement above, we can temporarily neglect the last two points listed above. Considering only the task of allocating 20 boxes into 10 different backpacks, a solution can be obtained by implementing a “recursive" version of the **0-1 Knapsack Problem**:
Assigning to each box a size and a value, and ordering the 10 backpacks' capacity in a given sequence, we start by applying the *0-1 Knapsack problem* solution recipe assuming the first capacity of the sequence and using all 20 boxes. Once the first backpack is filled - the maximum available space has been occupied - we can repeat the procedure, adopting only the remaining boxes and the second backpack. The process is repeated until all backpacks have been used. The upshot will be a series of backpacks filled to a certain degree, as quantified by the maximum value obtained via the *0-1 Knapsack* routine, and (possibly) a set of unallocated boxes. Summing the maximum values gives the total performance of the allocation process.
This procedure does not necessarily produce the most efficient outcome. To search for other, more performing possibilities, we can shuffle the order of the backpacks - thus their capacity - and repeat the recursive procedure above from the beginning, each time recording the global (cumulative) maximum value obtained via the *0-1 Knapsack* algorithm. In the end, after performing various reshuffles of the backpacks' capacity, the sequence of backpacks yielding the highest cumulative value represents the best performing choice.

The time ordering on the boxes and the two class labels can be reintroduced using these features of the data. Exploiting this information at the end of the process above, one can group the boxes following their time affinity and class membership; inserting this criteria during the boxes allocation procedure above might affect the efficiency of the process. At this point, one can identify, for example, which sequence of backpacks yields the best trade-off between the size-value allocation performance and the time or class grouping of the boxes.

Pragmatic solution
A *brute force* solution could entail exploring all possible allocations by simulating the scenarios. In this case, one would compute all possible combinations of boxes $\binom{n}{k}$ and detect the ones which, for a given capacity, occupy the largest space available, achieving the highest boxes value. On one hand, this approach might more easily accommodate the time ordering and class membership conditions and should perform well when small quantities (number of boxes and backpacks) are involved. However, its performance might quickly worsen as the number of elements involved grows. On the other hand, the execution time of the *Methodic solution* presented above scales roughly as $O(nCur)$, with n the number of boxes, C the capacity of each backpack, u the number of backpacks (assuming, for simplicity, that all have $\sim C$) and r the number of reshuffles. Therefore, the *Methodic solution* might be preferable when large sets are involved.

Numerical example

Below is displayed a numerical example of the implementation of the *Methodic solution* presented above. The starting point is a definition of the function implementing the *0-1 Knapsack problem* algorithm. For a given backpack capacity, number of boxes and the arrays containing the size, value, class membership and time label of the boxes available, this function returns the cumulative value of the boxes allocatable within this backpack, their corresponding index, class and time label.

```
import numpy as np

def knapSack(C, sz, vl, cl, tl, n):
    # function solving the 0-1 Knapsack problem
    #
    # input:
    #   - C = capacity of the backpack
    #   - sz = array of sizes (backpacks capacity)
    #   - vl = array of boxes value
    #   - cl = array of class label on boxes
    #   - tl = array of time label on boxes
    #   - n = number of boxes to be allocated
    #
    # output:
    #   - total value attained from allocation process
    #   - indices of items placed in the knapsack
    #   - time label of allocated boxes
    #   - class membership of allocated boxes
    #
    # NB: allocated items index, entering the second returned value, refer
    #     to the positional argument of the provided sz, vl, cl, tl arrays
    #     (e.g. '1,2,3,' means first, second, third argument)
    #

    # initialiase an array of empty dictionaries
    K = [[{'v':0, 'i':'', 'c':'', 't':[]} for y in range(C+1)] for z in␣
 ↪range(n+1)]

    # Build table K[][] in bottom up manner
    for i in range(n+1):
        for s in range(C+1):
            if i==0 or s==0:
                K[i][s]['v'] = 0
                K[i][s]['i'] = ''
                K[i][s]['c'] = ''
                K[i][s]['t'] = ''

            elif sz[i-1] <= s:
                T1 = vl[i-1]
                T2 = K[i-1][s-sz[i-1]]['v']
                T3 = K[i-1][s]['v']
                if T1+T2>=T3:
                    K[i][s]['v'] = T1+T2
                    K[i][s]['i'] = K[i][s]['i'] + '%i,'%i
                    K[i][s]['c'] = K[i][s]['c'] + '%s,'%cl[i-1]
                    K[i][s]['t'].append(tl[i-1])
                    if T2!=0:
```

```
                        K[i][s]['i'] = K[i][s]['i'] + K[i-1][s-sz[i-1]]['i']
                        K[i][s]['c'] = K[i][s]['c'] + K[i-1][s-sz[i-1]]['c']
                        K[i][s]['t'].extend(K[i-1][s-sz[i-1]]['t'])
                else:
                    K[i][s]['v'] = T3
                    K[i][s]['i'] = K[i-1][s]['i']
                    K[i][s]['c'] = K[i-1][s]['c']
                    K[i][s]['t'].extend(K[i-1][s]['t'])
            else:
                K[i][s]['v'] = K[i-1][s]['v']
                K[i][s]['i'] = K[i-1][s]['i']
                K[i][s]['c'] = K[i-1][s]['c']
                K[i][s]['t'] = K[i-1][s]['t']
    return K[i][s]
```

Now we can initialise the arrays containing:
- the backpacks' capacity
- the boxes' value
- the boxes' size
- the boxes' class label
- the boxes' time label

```
nBc = 10   # total number of backpacks
minC = 10  # minimum backpack capacity
maxC = 30  # maximum backpack capacity

WW = np.random.randint(minC, maxC, nBc)  # array of backpacks' capacity
OO = list(range(len(WW)))                # initial ordering of backpacks

nBx = 20   # total number of boxes
minV = 50  # minimum boxes value
maxV = 100 # maximum boxes value
minS = 1   # minimum boxes size
maxS = 20  # maximum boxes size

vl = np.random.randint(minV, maxV, nBx)  # array of boxes value
sz = np.random.randint(minS, maxS, nBx)  # array of boxes size
nn = len(vl)                             # initial number of boxes

minT = 100  # minimum time label
maxT = 200  # maximum time label (dimensionless, for simplicity)
tl = np.random.randint(minT, maxT, nBx)          # boxes time label
cl = np.random.choice(['typeA', 'typeB'], nBx, ) # boxes class label
```

Recursive 0-1 Knapsackalgorithm

The code that follows is the heart of the "recursive" *0-1 Knapsack problem* solution strategy presented above. For a given number `Nint` of iterations, the procedure prints the iteration step, the maximum value, obtained from the boxes allocation within the sequence of backpacks, and the number of boxes allocated.

```
from random import shuffle

Nint = 10 # number of iterations (reshuffles)

# initialise lists to contain results of all iterations
Varray= []
Narray= []
Barray= []
Carray= []
Tarray= []

print('{:>10} | {:>10} | {:>10}'.format('iteration', 'local max', 'N boxes␣
 ↪used'))
for r in range(Nint):

    # initialise lists to contain results of individual iteration
    V = 0
    N = 0
    B = []
    C = []
    T = []

    # renaming of arrays (for reiteration purpose)
    newvl = vl
    newsz = sz
    newcl = cl
    newtl = tl
    newn = nn

    # recursive 0-1 Knapsack problem solution algorithm
    for w,W in enumerate(WW[OO]):
        # execution of knapSack function
        res = knapSack(W, newsz, newvl, newcl, newtl, newn)

        # saving maximum value obtained
        V += res['v']

        # extracting the items allocated in last execution
        items = res['i'].split(',')
        items = [i-1 for i in map(int,filter(bool,items))]
```

```
        # saving properties of allocated boxes
        N += len(items)
        B.append(items)
        C.append(cl[items])
        T.append(tl[items])

        # update operational arrays, removing allocated elements
        newvl = np.delete(newvl, items)
        newsz = np.delete(newsz, items)
        newcl = np.delete(newcl, items)
        newtl = np.delete(newtl, items)
        newn = len(newvl)

    # build global lists containing all results
    Varray.append(V)
    Narray.append(N)
    Barray.append(B)
    Carray.append(C)
    Tarray.append(T)
    print('{:10} {:10} {:10}'.format(r+1,V,N))
    shuffle(OO)
```

```
 iteration |  local max | N boxes used
         1       1475         19
         2       1418         18
         3       1418         18
         4       1475         19
         5       1418         18
         6       1475         19
         7       1475         19
         8       1418         18
         9       1475         19
        10       1475         19
```

Now we select only those iterations which yield the maximum value

```
# determine the maximum value(s) within Varray
# and the correspoding items' index
maxV = max(Varray)
indm = np.where(Varray==maxV)[0]

print('global maximum value attainable : ',maxV,'\n')

for ii in indm:
    minN = Narray[ii]
    print('interation {:2} , number of boxes used : {:2}'.format(ii+1,minN))
```

```
global maximum value attainable :  1475

interation  1 , number of boxes used : 19
interation  4 , number of boxes used : 19
interation  6 , number of boxes used : 19
interation  7 , number of boxes used : 19
interation  9 , number of boxes used : 19
interation 10 , number of boxes used : 19
```

At this point, if more than one result is obtained (as in the present situation), we can examine such iterations imposing some criteria based on the class membership of the boxes and their time label. For example, we can search the iterations for backpacks containing boxes with compatible time labels. Expecting a lack of ordering after an application of the *Recursive 0-1 Knapsack algorithm* above, the compatibility could consist of a timing difference ε among boxes within the same backpack. In this case, we can assign a score to each backpack depending on whether the boxes therin contained have time labels, e.g., differing less than $\varepsilon = 10$ from the item with the largest time label. Finally, the sequence of backpacks leading to the largest cumulative score could be the preferred one. An example of this calculation is shown in the snippet below

```python
eps = 10

print('{:>10} | {:>10}'.format('iteration', 'time score'))

for ii in indm:
    Nt = 0
    for TT in Tarray[ii]:
        if len(TT)>1:
            tmax = max(TT)
            for tt in TT:
                if tt!=tmax and tmax-tt<=eps:
                    Nt += 1
    print('{:8} {:8}'.format(ii+1,Nt))
```

```
 iteration | time score
       1        3
       4        1
       6        1
       7        3
       9        1
      10        2
```

Of all six possible backpack orderings leading to same allocation efficiency, two iterations lead to the highest score, given the criteria on the time ordering on the boxes proposed above.

Alternatively, we might look for iterations leading to more backpacks filled with compatible classes of boxes. Since the number of backpacks remains constant across different iterations, we can simply sum the number of times a specific class appears, leading to a “class score". An

example of this calculation is shown in the snippet below

```
print('{:>10} | {:>10}'.format('iteration', 'class score'))

for ii in indm:
    Nc = 0
    for CC in Carray[ii]:
        if len(CC)>0:
            for cc in CC:
                if cc=='typeA':
                    Nc += 1
    print('{:8} {:8}'.format(ii+1,Nc))
```

```
 iteration | class score
       1        6
       4        9
       6        9
       7        8
       9        8
      10       10
```

As mentioned previously, conditions relating to these two features (time label and class membership) can also be included in the `knapSack` function. However, doing so would likely alter the performance of the algorithm, whenever the main criteria is the efficient allocation of the boxes, given the available capacities.

To conclude, whether or not all 20 boxes can be accommodated in all 10 backpacks depends on the backpacks' capacity and the boxes' size. In the solution of the problem just presented, we initially neglect the class membership and time label features, prioritising the most efficient allocation based uniquely on boxes' size. The two properties are used at a later stage, implementing some arbitrary conditions as an illustrative example.