**Homework 1 - CLASS-BASED STORAGE ASSIGNMENT**

Alberto Dian\*, Davide Nascivera, Yunju Li  
*\**Group representative

# Problem description

The work focuses on defining a functional and efficient storage system to ensure the presence of products in the warehouse when a request occurs. The developed storage system is based on a class-based policy, where products are classified into 3 classes, from A to C, based on their importance. The class is assigned based on the turnover values for each product, which represent the quantity requested by the customer, thereby determining its strategic position within the warehouse, placing the most requested products closer. The goal of the work is therefore to minimize the total distance to collect the products by associating a set of products with each class.

# Mathematical model

To achieve the goal previously stated, we followed a precise mathematical model. Using CPLEX and Python, we were able to define a model having the following parameters:

C number of classes

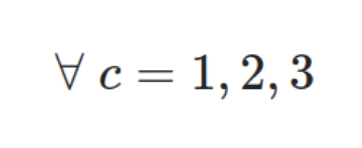
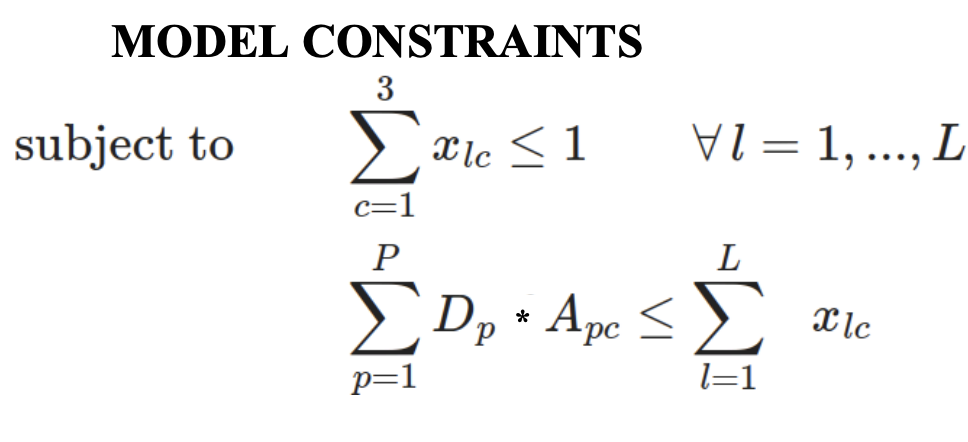
L number of station to store the products

tl travel distance from location l to the single depot

Dp demand for product p out of P products

Apc the binary parameter equal to 1 if prod p belongs to class c.

We also defined a decision variable called xlc that is equal to 1 if the class c products are stored in the location l. The used model was subject to the following constraints:

The first constraint guarantees that there is at most one product class per location,while the second says that the storage capacity must be guaranteed for every class. 

To minimize the expected total picking distance, we used the following objective function:

# Main code components

The first part of the code focuses on importing the data to set up and solve the problem. In this case the input are two .csv tables, one called “distances” containing the distance for every station from the single depot, and the other called “product\_data” containing the demand and the value for each product. Here below is reported the code:

# 1. Compute the turnover and assign each product to its correct class

# Read the CSV file

distances = pd.read\_csv("input/distances.csv", index\_col=0)

product\_data = pd.read\_csv("input/product\_data.csv")

demand = product\_data[['prod\_id','Demand']].transpose().values

value = product\_data[['prod\_id','Value']].transpose().values

dist = distances['distance'].transpose().values

num\_lines = len(product\_data)

#compute the turnover for each product

turnover = demand[1]\*value[1]

# Compute the total turnover

tot\_turnover = np.sum(turnover)

# Compute the total demand

tot\_demand = np.sum(demand[1])

# Compute the turnover in percentage values

percentage\_turnover = np.around((turnover/tot\_turnover)\*100, 2)

# Compute the demand in percentage values

percentage\_demand = np.around((demand[1]/tot\_demand)\*100, 2)

turnover\_data = []

for i in range(num\_lines):

turnover\_data.append((i,np.around(turnover[i],2), percentage\_demand[i], percentage\_turnover[i]))

After reading the .csv files we were able to calculate the turnover value for each product, simply multiplying the demand for each product and its value. Using the numpy library, we obtained the total value of turnover and demand, which will be used later to calculate the percentage value by simply dividing each value by the total value. In the end, we inserted all the data in a list, where each row contains information about the product id, the absolute turnover, the demand in a percentage value and the turnover in a percentage value.

# Sort the list considering the turnover percentage

sorted\_data = sorted(turnover\_data, key=lambda x: x[2], reverse=True)

# Compute the cumulative percentage of turnover

cumulative = 0

cum\_demand = 0

for i in range(num\_lines):

cum\_demand = sorted\_data[i][2] + cum\_demand

cumulative = sorted\_data[i][3] + cumulative

sorted\_data[i] = sorted\_data[i] + (np.around(cumulative,2),) + (np.around(cum\_demand,2),)

# Print the table with all the values

header = ["Product Id", "Absolute turnover", "Percentage Demand [%]","Percentage Turnover [%]", "Cumulative turnover [%]", "Cumulative demand [%]"]

print(tabulate(sorted\_data, header))

print("Total turnover: ", tot\_turnover)

Here we sorted the list in descending order considering the value of the percentage demand in order to calculate the cumulative demand and cumulative turnover in the for loop.

# Assignment of each product to its correct class

classA = 60

classB = 80

classC = 100

cumulative\_demand = 0

for i in range(len(sorted\_data)):

cumulative\_demand += sorted\_data[i][2]

if cumulative\_demand < classA:

sorted\_data[i] = sorted\_data[i][:6] + ("A",)

elif cumulative\_demand < classB:

sorted\_data[i] = sorted\_data[i][:6] + ("B",)

else:

sorted\_data[i] = sorted\_data[i][:6] + ("C",)

header = ["Product Id", "Absolute turnover", "Percentage Demand [%]", "Percentage Turnover [%]", "Cumulative turnover [%]", "Cumulative demand [%]", "Class"]

print(tabulate(sorted\_data, header))

To classify the products into the 3 classes we used two different approaches. In the one presented above, we focused on the value of the demand, having the first threshold for a value of 60, from 60 to 80 for the class B and from 80 to 100 for the third class. Using the cumulative demand as the value for the classification we were able to reduce the total picking distance since in this way we are positioning the most required products closer to the entrance of the storage system.

# Assignment of each product to its correct class

classA = 80

classB = 95

classC = 100

cumulative\_turnover = 0

for i in range(len(sorted\_data)):

cumulative\_turnover += sorted\_data[i][3]

if cumulative\_turnover < classA:

sorted\_data[i] = sorted\_data[i][:6] + ("A",)

elif cumulative\_turnover < classB:

sorted\_data[i] = sorted\_data[i][:6] + ("B",)

else:

sorted\_data[i] = sorted\_data[i][:6] + ("C",)

header = ["Product Id", "Absolute turnover", "Percentage Demand [%]", "Percentage Turnover [%]", "Cumulative turnover [%]", "Cumulative demand [%]", "Class"]

print(tabulate(sorted\_data, header))

For the second case analyzed, we used as a threshold the cumulative turnover for each product, classifying products as class A for values lower than 80%, from 80% to 97% for class B and from 97% to 100% for class C.

After the classification, we computed the value of the parameter Apc in the following manner:

# Compute the Apc parameter

Apc = []

temp\_data = sorted(sorted\_data, key=lambda x: x[0])

for i in range(len(temp\_data)):

if temp\_data[i][6] == 'A':

Apc.append((1,0,0))

elif temp\_data[i][6] == 'B':

Apc.append((0,1,0))

else:

Apc.append((0,0,1))

classes = ["A", "B", "C"]

print(tabulate(Apc, classes))

For the second part of the assignment, we implemented the model using CPLEX to run the optimization.

# 2. Solve the optimization problem in CPLEX

mdl = Model(name="prod\_storage")

# Define the parameters

C = 3 #number of classes

L = len(dist) #number of locations

tl = dist #travel distance from location l to single depot

Dp = product\_data['Demand'].values #demand for each product

P = len(Dp) #number of products

print(L)

# Define the decision variable

xlc = mdl.binary\_var\_matrix(L,C, name="x")

# Define the constraints

# At most one product class per location

for l in range(L):

mdl.add\_constraint(mdl.sum(xlc[l,c] for c in range(C)) <= 1)

# Storage capacity must be guaranteed for every class

for c in range(C):

mdl.add\_constraint(mdl.sum(Dp[p]\*Apc[p][c] for p in range(P)) <= mdl.sum(xlc[l,c] for l in range(L)))

# Define the Objective function

mdl.minimize(mdl.sum(mdl.sum(tl[l]\*xlc[l,c] for l in range(L)) \* mdl.sum(Dp[p]\*Apc[p][c] for p in range(P)) for c in range(C)))

As discussed before, to define the model we have to write down all the parameters needed, followed by the decision variable, the constraints and finally the objective function.

print(mdl.export\_to\_string())

sol = mdl.solve()

sol.display()

To solve the problem, we called the built in function solve() and we were able to obtain the results. Using the demand to classify the products, the value for the objective function is 695941 compared to 700175 using the cumulative turnover.

for c in range(len(classes)):

class\_assigned\_locations = [l+1 for l in range(L) if sol.get\_value(xlc[l,c]) == 1]

print(f"Class {classes[c]} Storage Locations: {class\_assigned\_locations}")

To print the class with every product assigned we used this for loop. Finally, to visualize the results, we implemented a heat map that shows the allocation of classes for each location of the storage area. Here below is presented the code:

# Initialize a matrix to store class assignments for each station

class\_matrix = np.zeros((11, 15), dtype=int) # Assuming 11 shelves and 15 locations

tmp = 0

# Mapping dictionary for class indices to letters

class\_letters = {0: 'A', 1: 'B', 2: 'C'} # Add more mappings as needed

# Populate the class assignment matrix

for n in range(0, 11): # assuming 11 shelves

for m in range(0, 15): # iterating through locations

class\_matrix[10 - n][m] = 4

for c in range(len(classes)):

if sol.get\_value(xlc[tmp, c]) == 1:

class\_matrix[10 - n][m] = c # Assign class index to matrix

tmp += 1

# Create the heatmap

plt.figure(figsize=(8, 6))

plt.imshow(class\_matrix, cmap='viridis', interpolation='nearest')

# Add text annotations for class letters inside the cells

for i in range(11):

for j in range(15):

if class\_matrix[i][j] in class\_letters:

plt.text(j, i, class\_letters[class\_matrix[i][j]], ha='center', va='center', color='black')

# Set axes labels

plt.xlabel('Shelves')

plt.ylabel('Location')

# Set colorbar

cbar = plt.colorbar()

cbar.set\_label('Class Assignment')

# Set ticks for shelves and locations

plt.xticks(np.arange(15), np.arange(1, 16))

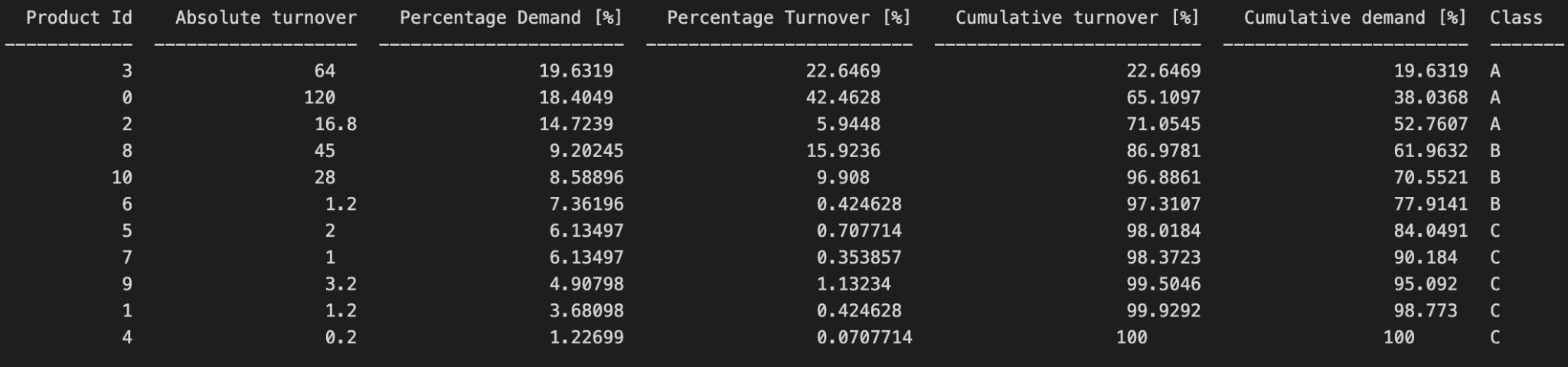
plt.yticks(np.arange(11), np.arange(1, 12))

plt.title('Class Assignment Heatmap - Demand classification')

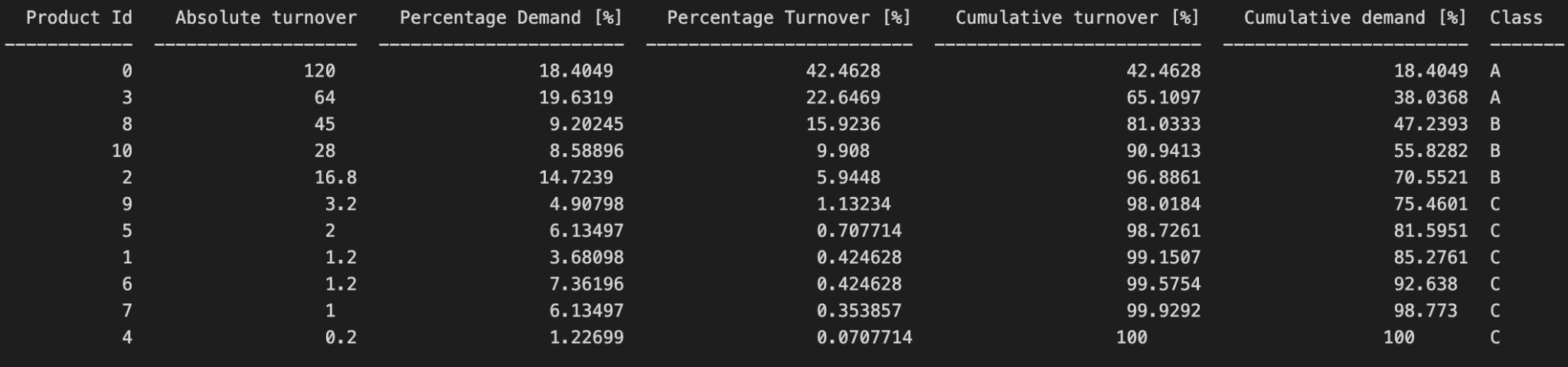
plt.show()

# Results and insights

The key step of the whole process is the first classification we did for each product. Here below are reported the results using the demand-thresholds method:

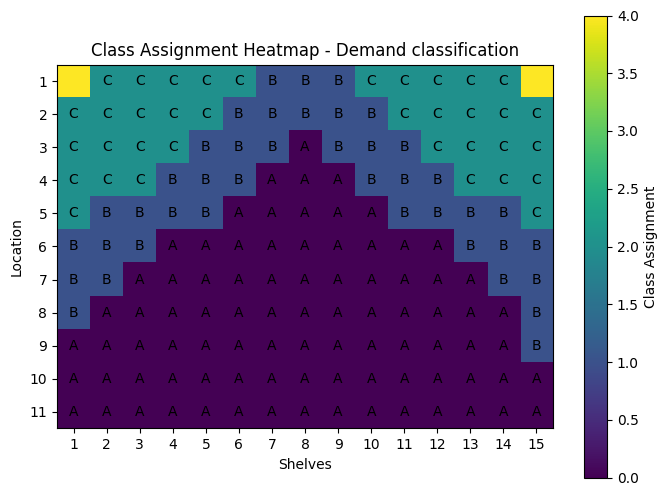
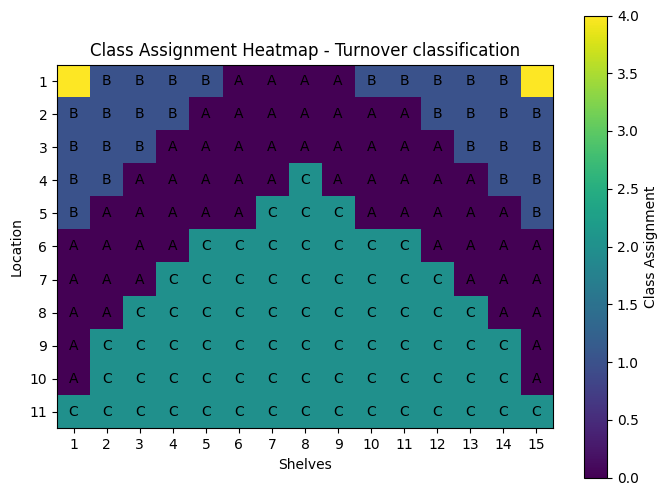


As previously mentioned, for the second case we did instead a turnover-threshold classification:

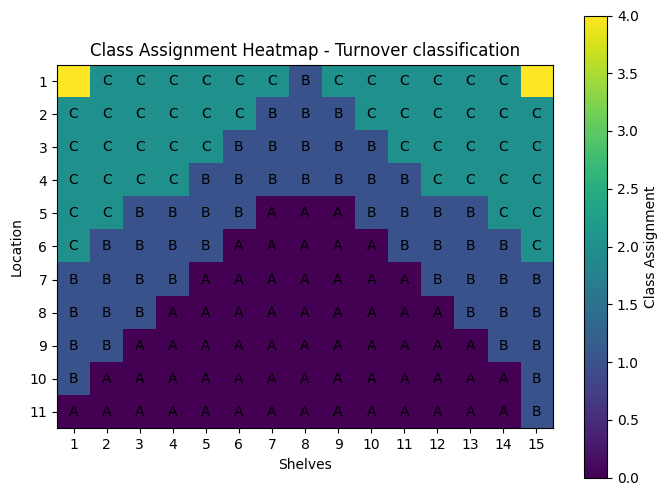


The tables show how both techniques produce similar results, with differences only for product 2 and 6. This is because the turnover considers the demand for each product and also the value of the product itself, giving more importance to those which can have a lower demand. Clearly, the value of the threshold plays an important role in the classification process. In this case, we used the values that were able to minimize the total picking distance, but we saw that the whole process is really sensible to little variations, producing different results.

The optimization process of the model has produced results that are coherent with what we expected to achieve. In both cases analyzed, the one classifying the products given the demand and the other given the turnover, the result is similar. We obtained a value of around 717000 for the objective function using the cumulative turnover and 695000 using the demand threshold, which is the minimum we could achieve considering the positioning of the products in each location. Here below are two maps showing the class assigned to each location, where the first station is positioned in the bottom-left corner while the last is in the top-right corner.

As it is stated in the title of the pictures, the one on the left refers to the case where the classification of the products was done using demand-thresholds while the one on the right using turnover-thresholds. Here we can see how we obtained different results, where in the demand classification case, the products of class A are closer to the depot entry and as the distance increases products of class B and C are positioned respectively further. For the other case instead, the class C has been positioned closer to the entry, followed by class A products and then class B. This result is given by the objective function that works on minimizing the total picking distance for each class. As a consequence, the classes with a higher number of products will be stored closer to the entry of the depot. If we look closer to the results, in the demand classification case, the number of products assigned to class A is 86, 41 for class B and 36 for class C, while in the turnover case we have 72 products for class C, 62 for class A and only 29 for class B. 

By varying the threshold values, we can see how the class assignment changes and consequently also the number of products for each class. In fact, using the cumulative turnover method with the 80% for class A, 17% for class B and 3% for class C the results are the following:



In this case, products from class A have been positioned closer, followed by products from class B and then C. This result follows the reasoning made before: the number of products of class A is 62, 53 for class B and 48 for class C.