Homework 1 - Class-Based Storage

Stefano Tonini*,Lorenzo Sciotto, Jacopo Dallafior

*Group representative

At least 1000 words per homework (without code)

1. Problem description

The problem at hand requires an engineering solution to minimize the expected total picking distance. The warehouse is envisioned to be organized based on different classes, each linked to a specific storage zone. Our approach involves organizing the division according to the appeal of each class.
The ABC analysis, inspired by the concept from the Italian Economist Pareto, suggests that roughly 80% of consequences stem from 20% of causes in many outcomes.
This analysis categorizes goods based on two primary criteria: annual usage and annual turnover.
The type of division decided to use is correlated with the demand for that specific product.
Products entering Class A constitute around 60% of the total demand.
Class B products have a demand ranging between 60% and 80%.
Finally, Class C products encompass the remaining demand.
The goals to be achieved in this homework are differents:
Firstly, find a valid turnover value for each product and print it in descending order, paired with the cumulative percentage turnover.
The second task involves calculating the binary parameter 'Apc' for each product, where 'Apc' indicates whether a product 'p' belongs to class 'c'.
To conclude, implement the mathematical model in Python and solve the optimization problem using the CPLEX solver.

2. Mathematical model

The mathematical model aims to determine the optimal location for each class to minimize the expected total picking distance. The storage assignment policy follows a class-based approach, resulting in the assignment of specific warehouse regions to each class.
To create an efficient mathematical model, it's beneficial to start by identifying the indices. In this scenario, the present indices that prove useful for solving my problem are:

- The type of class ‘c’ for the product where c = 1,...,C, represents the class number
- The location number of each storage unit, where l = 1,....,L
- The product type, which can be represented as: p = 1,....,P

Next, let's describe the parameters of my case study. They include:

- The market demand for product type 'd'.

- The value corresponding to product type 'v'.
- The travel distance between the depots and the storage location of product 'ts'.
- The turnover of product type 'T', calculated as the product of 'd' and 'v'.
- The matrix 'Apc' represents the assignment of classes to products.

The Object function of the mathematical model is:

$$\text{minimize} \qquad \sum_{c=1}^{3}\left[\sum_{l=1}^{L} t_l\, x_{lc} \sum_{p=1}^{P} D_p\, A_{pc}\right]$$

Where:

- t_l is the travel distance from location l to the single depot
- x_{lc} is a binary decision variable that is equal to 1 if class c products are stored in location l, is equal to 0 otherwise
- d_p is the demand for product p out of P products
- A_{pc} is a binary parameter equal to 1 if product p belong in class c, it is equal to 0 in otherwise

That formula searches the minimum travel distance to pick the product , taking in considering classes A, B, C.

The constraint to take in consideration in this scenario are two:

- $$\sum_{c=1}^{3} x_{lc} \leq 1 \qquad \forall\, l = 1, ..., L$$
  Ensure that at most one product class per location

- $$\sum_{p=1}^{P} D_p * A_{pc} \leq \sum_{l=1}^{L} \quad x_{lc} \qquad \forall\, c = 1, 2, 3$$
  Ensure that all storage capacity must be guaranteed for every class

3. Main code components

To make a well-structured code in the beginning every library used in the code with their respective abbreviation must be imported (‘numpy’, ‘seaborn’, ‘matplotlib.pyplot’ and ‘tabulate’) and the solver CPLEX in order to use them later.
Secondly the input csv file must be imported and correctly stored using the library ‘pandas’. The two csv file contained all the information necessary to define the parameter of the mathematical model

```
# Parameters

# Input data from text
C = 3 # Number of classes of product A, B, C

# Input data from csv file
prod_params = pd.read_csv("./product_data.csv", index_col=0)
travel_distance = pd.read_csv("./distances.csv")

d = prod_params['Demand'].values                        # Demand of products
v = prod_params['Value'].transpose().values             # Value of products
ts = travel_distance['distance'].transpose().values     # Travel distance

N = len(d)      # Type of product in the problem
M = len(ts)     # Number total storage allocations

T = d * v       # Turnover
```

More precisely ‘C’ is the total number of classes that in the ABC analysis is three, ‘d’ is the demand and ‘v’ the value of every product in the inventory, ‘ts’ is travel distance to every location l. Defining the number of different products can be done with the python function 'len()’ using as input ‘d’ and also in order to define the total number of storage allocation the same python function with ‘ts’ as input can be used.
There is one missing parameter that is still not defined and it is fundamental for the mathematical model and the further calculation, in particular A_{pc} a binary parameter equal to 1 if product ‘p’ belongs in class ‘c’, it is equal to 0 otherwise.
As discussed previously the matrix A_{pc} can be calculated with two criterions that are annual usage and annual turnover.
Afterwards it is proposed the code for both criteria while in the next section it is proposed a detailed discussion about the result of the criterias.
Considering the former criteria, annual usage,the code was

```
# Calculate matrix A[p,c] using demand

# Step 1: Calculate Total Demand
total_demand = sum(d)

# Step 2: Calculate Percentage Contribution and Cumulative Percentage
percentage_contributions_d = [(d[i] / total_demand) * 100 for i in range(len(T))]

# Step 3: Sort products in descending order of turnover
sorted_demand_indices = sorted(range(len(d)), key=lambda k: d[k], reverse=True)
print(sorted_demand_indices)
cumulative_percentage = 0

# Step 4: Assign Classes (A, B, C) based on Thresholds
class_assignments = []
for i in sorted_demand_indices:
   cumulative_percentage += percentage_contributions_d[i]
```

```
    if cumulative_percentage <= 60:
        product_class = 'A'
    elif cumulative_percentage <= 80:
        product_class = 'B'
    else:
        product_class = 'C'

    class_assignments.append(product_class)

# Step 5: Update A matrix based on class assignments (binary values)
classes = ['A', 'B', 'C']
A = np.zeros((len(T), len(classes)))

for i in range(len(T)):
    class_index = classes.index(class_assignments[i])
    A[sorted_demand_indices[i], class_index] = 1  # Set the corresponding class to 1

print(A)
```

Regarding the latter, annual turnover, the code used was

```
# Calculate matrix A[p,c] using turnover

# Step 1: Calculate Total turnover
total_turnover = sum(T)

# Step 2: Calculate Percentage Contribution and Cumulative Percentage
percentage_contributions = [(T[i] / total_turnover) * 100 for i in range(len(T))]

# Step 3: Sort products in descending order of turnover
sorted_turnover_indices = sorted(range(len(T)), key=lambda k: T[k], reverse=True)
print(sorted_turnover_indices)
cumulative_percentage = 0

# Step 4: Assign Classes (A, B, C) based on Thresholds
class_assignments = []
for i in sorted_turnover_indices:
    cumulative_percentage += percentage_contributions[i]

    if cumulative_percentage <= 80:
        product_class = 'A'
    elif cumulative_percentage <= 95:
        product_class = 'B'
    else:
        product_class = 'C'
```

```
    class_assignments.append(product_class)

# Step 5: Update A matrix based on class assignments (binary values)
classes = ['A', 'B', 'C']
A = np.zeros((len(T), len(classes)))

for i in range(len(T)):
    class_index = classes.index(class_assignments[i])
    A[sorted_turnover_indices[i], class_index] = 1  # Set the corresponding class to
1

print(A)
```

All the parameters are now defined and usable for the mathematical model and the further calculation.

After Rearranging the location distance it was possible to compute the distance matrix and display it using “matplotlib.pyplot” library that was imported before
The library is useful to plot heatmap and distance matrices to get a rapid visualization of the data.

```
# Distance Matrix

# Convert the distances to a 12x15 matrix
distance_matrix = np.array(ts).reshape((12, 15))

# Create a DataFrame with the distance matrix
distance_df = pd.DataFrame(distance_matrix, columns=[f'Shelf {i + 1}' for i in range(15)], index=[f'Location {13 - i}' for i in range(12, 0, -1)])

# Plot the heatmap
sns.heatmap(distance_df, cmap='viridis', annot=True, fmt="d", cbar_kws={'label': 'Distance'})
plt.title('Distance Matrix')
plt.show()
```

Afterwards it was possible to compute the absolute turnover values per product in descending order paired with cumulative percentages

```
# Absolute turnover values per product in descending order paired with cumulative

# Step 1: Calculate Total Turnover
total_turnover = sum(T)

# Step 2: Calculate Percentage Contribution and Assign Classes
percentage_contributions = [(T[i] / total_turnover) * 100 for i in range(len(T))]

# Step 3: Print Absolute Turnover in Descending Order and Cumulative Percentage
sorted_turnover_indices = sorted(range(len(T)), key=lambda k: T[k], reverse=True)
cumulative_percentage = 0

print("Absolute Turnover\tCumulative Percentage")
print("-------------------------------------------------")
for i in sorted_turnover_indices:
    cumulative_percentage += percentage_contributions[i]
    print(f"{T[i]}\t\t\t\t{cumulative_percentage:.2f}%")

print("-------------------------------------------------")
print("Total Turnover:",total_turnover)
```

In the mathematical model the only decision variable is a binary matrix and in python must be defined as ‘mdl.binary_var_matrix’

```
# Decision variable
x = mdl.binary_var_matrix(M,C,name='x')
```

To code the objective function is straightforward and requires the mdl.sum function for the summatory in the formula of the mathematical model

```
# Objective function
mdl.minimize(mdl.sum(mdl.sum(x[l,c]*ts[l] for l in range(M)) * mdl.sum(d[p]* A[p,c] for p in range(N)) for c in range(C)))
```

The constraints are also trivia to code

```
# Constraints
for l in range(M):
    mdl.add_constraint(mdl.sum(x[l,c] for c in range(C)) <= 1)

for c in range(C):
    mdl.add_constraint(mdl.sum(d[p]*A[p,c] for p in range(N)) <= mdl.sum(x[l,c] for l in range(M)))
```

Using the upcoming code it is possible to display the solution and the final value of the objective function

```
# Solution
print(mdl.export_to_string())

sol = mdl.solve()

sol.display()
```

In conclusion the last part of the code is structured in order to display the solution found in a better and easier way and save it in a csv file that can be sent to the company to implement it

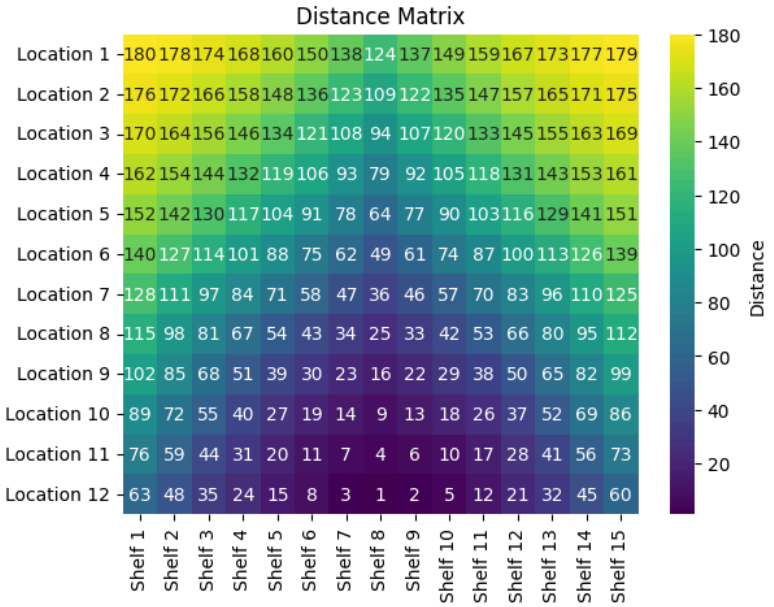
```
# Create a dictionary to store the assigned storage locations for each class
class_storage_data = {}

# Find the maximum number of locations assigned to a class
max_locations = max(sum(sol.get_value(x[l, c]) == 1 for l in range(M)) for c in range(len(classes)))

# Iterate through classes and products to store assigned locations
for c in range(len(classes)):
    class_assigned_locations = [l + 1 if sol.get_value(x[l, c]) == 1 else None for l in range(M)]
    # Pad the list to ensure all lists have the same length
    class_assigned_locations += [None] * (max_locations - len(class_assigned_locations))
    print(f"Class {classes[c]} Storage Locations: {class_assigned_locations}")
    class_storage_data[f'Class {classes[c]} Storage Locations'] = class_assigned_locations

# Ensure all lists have the same length by padding with None if needed
max_length = max(len(lst) for lst in class_storage_data.values())
for key, value in class_storage_data.items():
    class_storage_data[key] = value + [None] * (max_length - len(value))

# Create a DataFrame for class storage locations
class_storage_df = pd.DataFrame(class_storage_data)

# Save the DataFrame to a CSV file without the 'Class' column
class_storage_df.to_csv('class_storage_locations.csv', index=False)
```

and also a table to display all the important parameters calculated

```
# Create a list to store the results
result_data = []

# Populate the list with product information and class assignments
cumulative_percentage_demand = 0
for i in sorted_demand_indices:
    product_id = chr(ord('A') + i)  # Convert index to corresponding letter
    cumulative_percentage_demand += percentage_contributions_d[i]
    result_data.append({
        'Product': product_id,
        'Turnover': T[i],
        'Percentage Turnover Contribution': percentage_contributions[i],
        'Demand': d[i],
        'Cumulative Percentage Demand': cumulative_percentage_demand,
        'Class': class_assignments[sorted_demand_indices.index(i)]
    })

# Create a DataFrame from the list
result_df = pd.DataFrame(result_data)

# Set the 'Product' column as the index
result_df = result_df.set_index('Product')

# Display the final table
print(tabulate(result_df, headers='keys', tablefmt='grid'))

# Save the DataFrame to a CSV file
result_df.to_csv('final_table.csv')
```

4. Results and insights

It was shown in the previous section the code implemented and the library used in order to compute the distance matrix displayed after



Afterwards it was computed the absolute turnover values per product in descending order paired with cumulative percentages and the total turnover

Absolute Turnover	Cumulative Percentage
104.0	56.58%
36.0	76.17%
22.0	88.14%
14.0	95.76%
3.2	97.50%
2.0	98.59%
1.2	99.24%
1.2	99.89%
0.2	100.00%

Total Turnover: 183.8

The section after was regarding the matrix binary parameter for the ABC analysis for product classification that was computed using the annual demand criterion.
It is important to point out that finding the configuration between the two different criteria and the different value of threshold is one of the challenges of this problem and impacts the value of the objective function.
This decision was made after a comparison between the two different criteria.
Firstly, using the annual turnover criterion, with thresholds set as 80% of turnover for class A, 15% of turnover for class B and 5% for class C, the final value of the objective function was 463142.

Whereas in the second scenario implementing annual demand criterion, with thresholds set as 60% of demand for class A, 20% of demand for class B and 20% for class C, the final value of the objective function reduced to 383779.

The raw output of printing A_{pc} is shown on the left and it is just a matrix with ones if product 'p' is assigned to class 'c' and zero otherwise. In order to display A_{pc} in a more fashion way the library 'tabulate' can be used and gave the image on the right

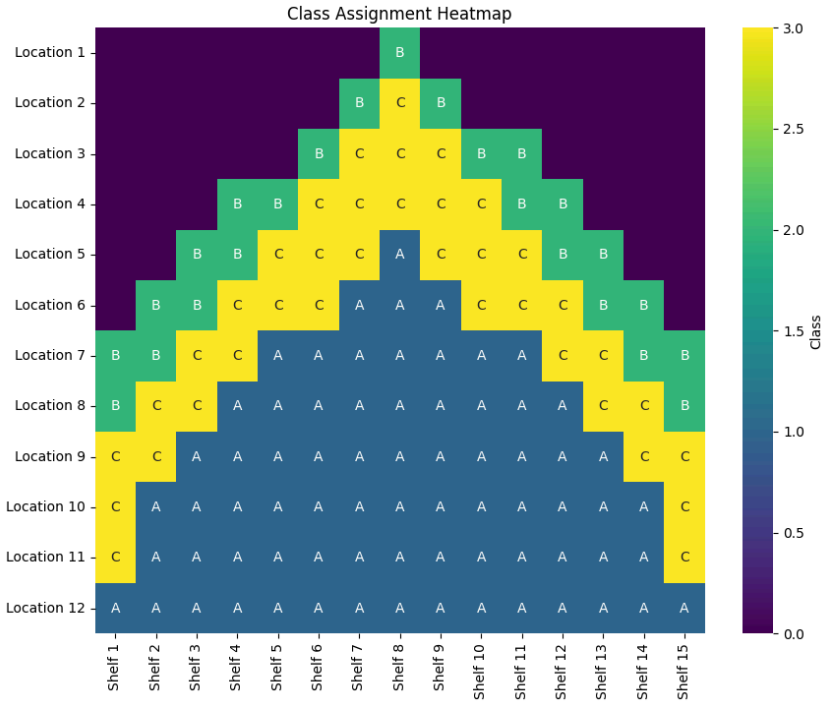
```
[[1. 0. 0.]
 [1. 0. 0.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 0.]]
```

Product	Class
B	A
A	A
H	B
I	B
G	C
D	C
F	C
E	C
C	C

Furthermore after setting all parameters, decision variable, objective function and constraint the solver CPLEX found a solution and the value of the objective function was 383779.
In conclusion the last part of the code was supposed to print and save the solution in trivia format that can be sent to the company.
The first one is basically a summary table with all the parameters calculated and the the assigned class for every product

Product	Turnover	Percentage Turnover Contribution	Demand	Cumulative Percentage Demand	Class
B	104	56.5832	52	39.0977	A
A	14	7.61697	20	54.1353	A
H	36	19.5865	12	63.1579	B
I	1.2	0.652884	12	72.1805	B
G	22	11.9695	11	80.4511	C
D	2	1.08814	10	87.9699	C
F	3.2	1.74102	8	93.985	C
E	1.2	0.652884	6	98.4962	C
C	0.2	0.108814	2	100	C

The last output is a csv file with 180 rows for the locations and three columns for the three classes in which in every class there will be the number of locations to store that specific class of products.
The solution can be displayed with a plot using 'seaborn' library in order to have a better visualization idea.



The visualization of the matrix showed that the solution found stores products in class B further than products in class C. This result can be explained by the structure of the objective function of the mathematical model. It is built to minimize the total picking distance for every class separately and then summed up. This basically means that after computing the matrix A if a class has more product than another class the objective function will suggest to store the class with higher number of products as near as possible and the class with lower number of products farther.
Furthermore, looking at the previous matrix, class C products are closer to the entrance than class B just because they are 37 whereas products in class B are fewer, only 24.