UNIVERSITY OF GRONINGEN

DATA ANALYSIS AND PROGRAMMING FOR OM

FINAL ASSIGNMENT

Warehouse planning for Gomibo

Authors:	*Student Numbers:*
Lucas Krol	S4072103

October 26, 2022

Contents

Introduction

In this report we consider a warehouse planning problem for Gomibo (Belsimpel). Specifically, Gomibo's first warehouse is not large enough anymore to stock all products and therefore Gomibo has to store some products in a second warehouse, which however can not promise next-day delivery. Consequently, products that are stored in the second warehouse will be sold less often. More specifically, products are expected to lose either 20%, 30% or 50% of their sales, for products that belong, respectively, to the class of the 20% most profitable products, the next 30% or the bottom 50%.

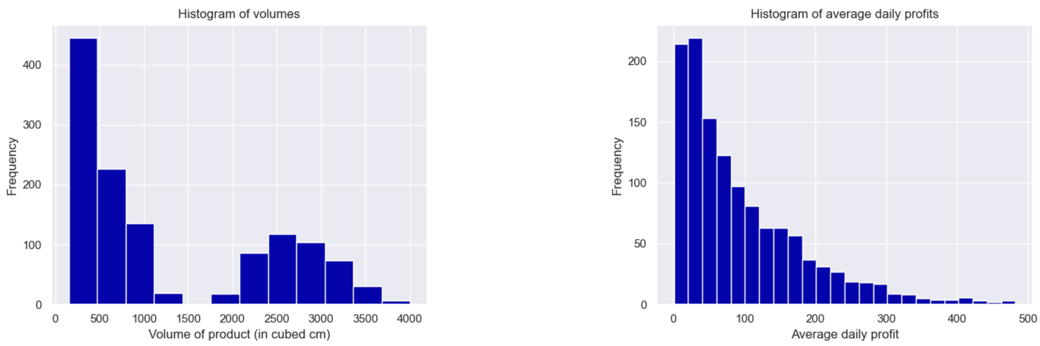
Additionally, we have to take into account that Gomibo has products that are correlated with each other. For instance, the Samsung S22 might often be sold together with a protection case for the Samsung S22. If we would then place the protection case in the second warehouse and the Samsung S22 in the first warehouse, combined delivery would take longer and this might lead to customers not ordering the Samsung S22 at all. To prevent this, we add the decision rule that if products are from different classes (based on profits as defined earlier) and are highly correlated (correlation > 0.6) we have to store them in the same warehouse.

Now, for every product we will stock a base-stock level of products. A base-stock level is defined as the average demand plus a mark-up based on the standard deviation. Importantly, we will never store the same product in two different warehouses: this would complicate operations significantly and consequently more likely result in higher costs. In total, the first warehouse has 960 storage boxes, of size 40x40x20 cms. Of this volume, we will assume that 90% can be used efficiently.

We will solve the problem of which product to assign to which warehouse such that the profit loss is minimized in three different ways. Firstly, we use two heuristics that give an upperbound to the total costs for the linear program we will use later. The first heuristic states that you assign products based on the average daily profit that they generate until you run out of storage space. The second heuristic uses the same algorithm, but then based on the ratio of the average daily profit with the number of boxes needed to store the product. In the linear program, we will solve a variation of the Knapsack problem.

Data

We have received data on the sales during a two-year period and the profit margins and dimensions of every product. In total, we have 1263 different products. To get a feeling for this data, we made histograms of the volumes and the average daily profits of the products in figure 1. This shows us that there are quite a lot of products with a low volume, which means you can store a lot of those products in one storage box. These might for example be protection cases, which usually have quite a low volume since it is not packaged in a box like phones are. The average daily profits plot shows us that there are a few products with very high daily profits, but that the epicentre is still at the lower segment. In figure 2 we show the average daily profit per product and showed in red the division where the top 20% ends and the middle 30% begins and where the middle 30% stops and the bottom 50% ends.



(a) Volumes (b) Average daily profits

Figure 1: Histograms

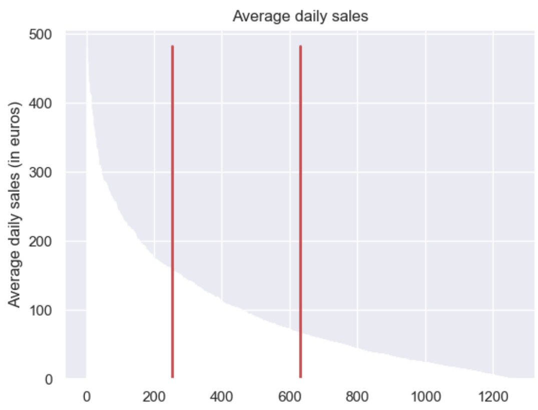
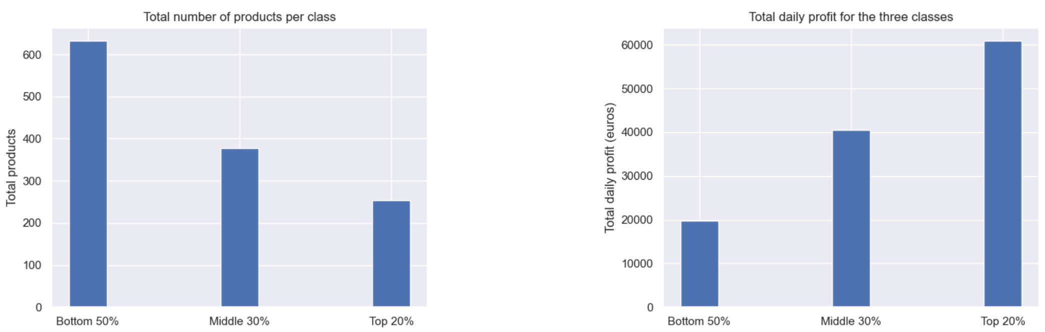


Figure 2: Average daily sales per product

In figure 3 we show the number of products in each class and the total daily profit every class generates. This shows us that even though the top 20% has a lot less products, it does generate a lot more profit than the other two classes.



(a) Number of products per class (b) Daily profit per class

Figure 3: Comparison of classes

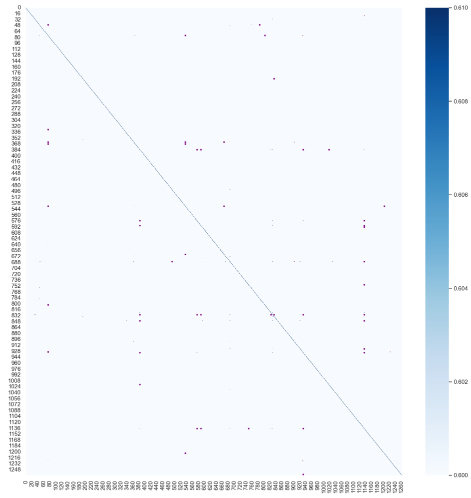


Figure 4: Heat map of the correlation matrix

In figure 4 we can see the correlations higher than 0.6 marked for the products. In here, we marked the "normal" correlations in blue and the correlations between products that should be kept in one warehouse in purple. Obviously, on the diagonal we have one big blue line: a product is 100% correlated with itself. On aggregate, we see that we have no big clumps of correlations, so the correlation requirement will not be too restrictive.

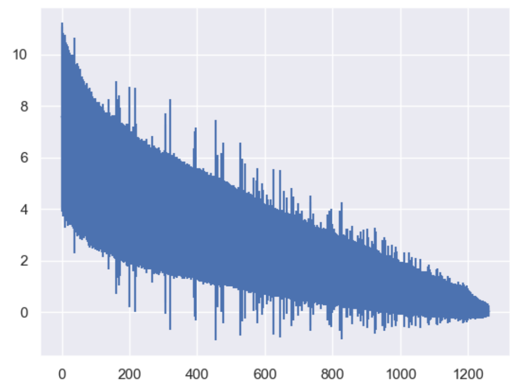


Figure 5: Error bars per product

Lastly, in figure 5 we see the error bars per product, with as mean the average number of orders per day and the length of the error bars is based on the standard deviation. This plot gives a small insight on how many products you will have to stock as base-stock level for a certain product, although the standard deviation is multiplied with a scalar bigger than 1 in the base-stock formula. Hence, these error bars give smaller base-stock levels than the actual ones.

Mathematical model

Heuristics

In the heuristics we prioritize products based on a certain measure. We will then place the products with the highest measure in the boxes, until all boxes are full. Importantly, if we place a product in the first warehouse, we also need to place its correlated products in the same warehouse. Obviously, if a product has a higher measure but there are not enough boxes available for the product (and its correlated products) we will still try to put a product with a lower measure in the first warehouse, as long as there are boxes available.

Specifically, in heuristic 1 the measure will be based on the average daily profit that a product generates, whereas in heuristic 2 the measure will be based on the ratio of the average daily profit to the boxes needed to store the product.

Knapsack problem

Let x_j denote if the product with product id j is stored in the **second** warehouse, $x_j \in \{0,1\}$.
Let p_j denote the number of boxes needed to store a base-stock level of product id j.
Let c_j denote the average daily costs of storing product j in the second warehouse.

Then, we want to solve the following (integer) linear problem:

$$
\begin{aligned}
\min \quad & c^T x \\
\text{s.t.} \quad & \\
& p^T(\iota_n - x) \qquad\qquad \leq 960 \\
& x_i(1-x_j) \qquad\qquad\; = 0 \quad \{i,j\} \in S \quad \forall S \\
& x_j \in \{0,1\} \quad j = 1,\ldots,n
\end{aligned}
$$

where the sets S contain two product ids, one of a higher class and one of a lower class, which are correlated in demand with a correlation of at least 0.6. Furthermore, ι_n is a (column) vector of ones of length n, where n is the number of products we have. Then, since x is binary and denotes the products stored in the second warehouse, $\iota_n - x$ yields the products stored in the first warehouse.

Results

In table 1 we report the costs of the different policies. We see that heuristic one does not perform very well, which makes sense: after all if a product generates a lot of profit but does need a lot of storage boxes, heuristic 1 will assign it all the boxes whereas in heuristic 2 or in the knapsack problem the product likely won't be assigned, leaving more room for the products that generate more profit per storage box. Heuristic 2 however performs very good, having less than a 2% difference in costs with the (for this set of assumptions) optimal solution given by the knapsack problem. However, even a 2% difference might mean a lot when we start talking about revenues of millions of dollars, as we are in the context of this report. The main reason why the knapsack problem performs slightly better is likely since it takes into account the fact that you do not merely need to account for the storage boxes needed for your own product, but also for the correlated products. E.g., if we have a product that only needs one storage box, it will likely have a high ratio and therefore be preferred by heuristic 2, but if it has a correlated product that needs 6 storage boxes to be stored, the actual ratio would be a lot smaller. This, in turn, will result in higher costs for heuristic 2.

Assignment policy	**Heuristic 1**	**Heuristic 2**	**Knapsack problem**
Daily costs	€ 16626.62	€ 10033.7	€ 9884.26

Table 1: Average daily costs of allocation

Sensitivity analysis

In addition to our regular results, we also present a sensitivity analysis. Our first sensitivity analysis is on the number of boxes that the first warehouse has. We are primarily interested in what the changes of first warehouse sizes would have for impact on the average daily profit loss. Of course, having a bigger first warehouse will always result in a lower profit loss, but from figure 6 we can see that the rate at which the costs decrease gets smaller for larger percentages. From figure 6 we can conclude that adding for instance 5% warehouse space would be beneficial as long as that comes with a corresponding operations costs smaller than approximately 600 euros per day. From figure 7 we can conclude that the results of heuristic 2 stay very good for all different percentages and are therefore quite robust, whereas heuristic 1 is better for very few boxes in warehouse 1 or when almost all products can be stored in the first warehouse. Hence, heuristic 1 performs badly overall.

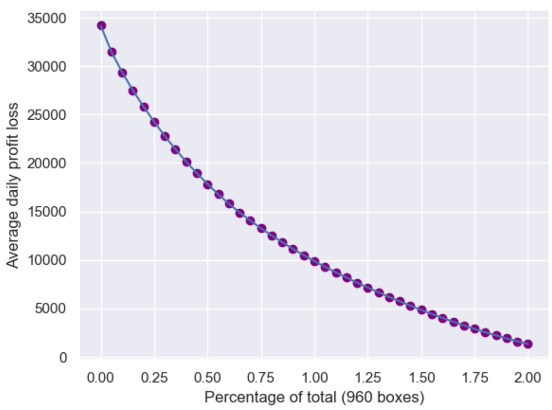


Figure 6: Daily costs for different warehouse sizes (0 - 1920 boxes)

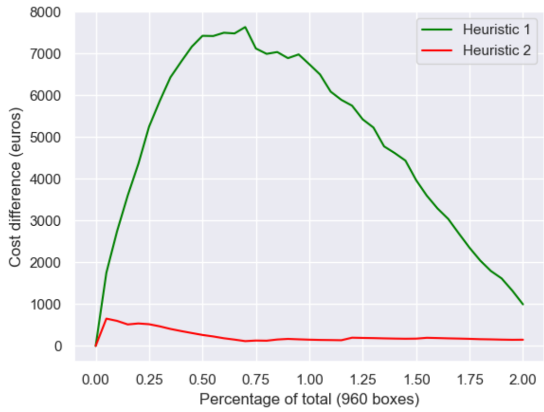


Figure 7: Difference in daily costs between Knapsack solution and heuristics

Our second sensitivity analysis is on the daily profit losses: we investigate what the effect of having different average daily profit losses per product is on the total profit loss. In figure 8 we plot this for different percentages of the usual amount and it might not come as a surprise that the costs simply linearly increase with the percentage. The allocation of which products to store in the first warehouse will after all stay the same and it will only result in different absolute costs, but the costs will stay the same relatively.

Figure 8: Daily costs for different average daily profit losses

Appendix

Main file

```
import pandas as pd
from elasticsearch import Elasticsearch
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm # Mainly used for the z-statistics
import seaborn as sns; sns.set_theme() # For the heatmap
from gurobipy import Model, GRB
from matplotlib.patches import Rectangle


pd.options.mode.chained_assignment = None # Ignores chained warnings
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
# Ignores unverified https request warning

import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
# Ignores DeprecationWarnings
## Questions:


## Loading the csv file for sales into the elastic database
es = Elasticsearch( # Specifies the elasticsearch "location" to store the data
    hosts="https://elastic:pancakes@localhost:9200/",
    verify_certs=False
)

## The next lines are commented, as we only want to run them once.
## After all, if we have made the database there is no need to make it again


# Creates the elastic index in which we will put the data
settings = {
    'settings': {
        "number_of_shards" : 3
    }
    'mappings': {
        'properties': {
            'pickup_datetime': {'type': 'date',
                                          "format": "yyyy-MM-dd HH:mm:ss" },
            'dropoff_datetime': {'type': 'date',
                                           "format": "yyyy-MM-dd HH:mm:ss" },
```

```
            'pickup_location': {'type': 'geo_point' },
            'dropoff_location': {'type': 'geo_point'}
        }
    }
}
es.indices.create(index="sales", body=settings)
print("index created")

import csv_to_elastic # Importing the python functions that allow us to,
# quickly, put the csv data into our elastic database
csv_to_elastic.ingest_csv_file_into_elastic_index(basepath + "/sales.csv",
es, "sales", 5000)



## We load in the pandas dataframes (made in DataFrameConstruction)
dfProducts = pd.read_pickle("dfProducts.pkl")
dfOrderPerDay = pd.read_pickle("dfOrderPerDay.pkl")
ndays = dfProducts.shape[0]

# Here we construct an errorbar plot, with as mean the average orders and
# error the standard deviation
dfSorted = dfProducts.sort_values(by = ["AvgOrders"], ascending=False)
# Sorts the dataframe descending by Avg Orders
plt.errorbar(x = range(0, len(dfSorted["AvgOrders"])),
             y = dfSorted["AvgOrders"], xerr = 0, yerr = dfSorted["StdOrders"])
plt.savefig('Errorbar orders.png')
plt.show()

# Here we make a histogram of the average daily profit generated by each product
dfProducts["AvgDailyProfit"] = dfProducts["AvgOrders"] * dfProducts["Margins"]
plt.hist(x=dfProducts["AvgDailyProfit"], bins='auto', color='#0504aa')
plt.xlabel('Average daily profit')
plt.ylabel('Frequency')
plt.title('Histogram of average daily profits')
plt.savefig('Histogram average daily profits.png')
plt.show()

# Here we make a histogram of the volume of each product
dfProducts["Volume"] = dfProducts["Length"] * dfProducts["Height"] \
                        * dfProducts["Width"]
plt.hist(x=dfProducts["Volume"], bins='auto', color='#0504aa')
plt.xlabel('Volume of product (in cubed cm)')
plt.ylabel('Frequency')
plt.title('Histogram of volumes')
plt.savefig('Histogram volumes.png')
```

```
plt.show()

# We divide the products up in three different classes,
# the top 20% profit generating products, the following 30% and the bottom 50%
dfProfitSort = dfProducts.sort_values(by=["AvgDailyProfit"])
nrow = dfProfitSort.shape[0]
bottom50 = dfProfitSort[0:(round(0.5*nrow))]
middle30 = dfProfitSort[(round(0.5*nrow)):(round(0.8*nrow))]
top20 = dfProfitSort[(round(0.8*nrow)):nrow]

# We will compute the total profit and the number of products per class and
# display them in a barplot
fig = plt.figure()
totprofits = [sum(bottom50["AvgDailyProfit"]), sum(middle30["AvgDailyProfit"]),
              sum(top20["AvgDailyProfit"])]
plt.bar(["Bottom 50%", "Middle 30%", "Top 20%"], totprofits, color = 'b',
        width = 0.25)
plt.ylabel("Total daily profit (euros)")
plt.title("Total daily profit for the three classes")
plt.savefig('Total daily profit per class.png')
plt.show()

fig = plt.figure()
totsales = [len(bottom50["AvgOrders"]), len(middle30["AvgOrders"]),
            len(top20["AvgOrders"])]
plt.bar(["Bottom 50%", "Middle 30%", "Top 20%"], totsales, color = 'b'
        , width = 0.25)
plt.ylabel("Total products")
plt.title("Total number of products per class")
plt.savefig('Barplot products.png')
plt.show()

# Making a barplot that shows the daily average profits, marking the
# products on the boundary of the product classes
fig = plt.figure()
colours = ["blue"] * nrow
max_profit = dfProfitSort["AvgDailyProfit"].iloc[-1]
plt.bar(range(0, nrow), dfProfitSort["AvgDailyProfit"][::-1],
        color=colours, width = 0.2)
plt.plot([round(0.5*nrow), round(0.5*nrow)], [0,max_profit],
         color='r', linestyle='-', linewidth=2)
plt.plot([round(0.2*nrow), round(0.2*nrow)], [0,max_profit],
         color='r', linestyle='-', linewidth=2)
plt.ylabel("Average daily sales (in euros)")
plt.title("Average daily sales")
plt.savefig('Barplot daily sales.png')
```

```
plt.show()

# Calculating the mean and standard deviation for the demand over the
# replenishment period
avg_list = 7 * dfProfitSort["AvgOrders"]
std_list = np.sqrt(7) * dfProfitSort["StdOrders"]

# Calculating the base-stock levels
quantile = [norm.ppf(0.9), norm.ppf(0.95), norm.ppf(0.99)]
# We do everything times 7 (sqrt(7)) because it is based on weekly data
bottom50["Basestock"] = round(bottom50["AvgOrders"] * 7 +
                              np.sqrt(7) * bottom50["StdOrders"] * quantile[0])
middle30["Basestock"] = round(middle30["AvgOrders"] * 7 +
                              np.sqrt(7) * middle30["StdOrders"] * quantile[1])
top20["Basestock"] = round(top20["AvgOrders"] * 7 +
                              np.sqrt(7) * top20["StdOrders"] * quantile[2])

volume_box = 40 * 40 * 20 # in cm^3
bottom50["NrProdPerBox"] = (0.9 * volume_box) // bottom50["Volume"]
# Notice the floor division and that we only use 90% efficiently
middle30["NrProdPerBox"] = (0.9 * volume_box) // middle30["Volume"]
top20["NrProdPerBox"] = (0.9 * volume_box) // top20["Volume"]

# Calculating the correlation matrix:
corr_basis = dfOrderPerDay.drop(columns="Day")
corr_matrix = np.corrcoef(corr_basis, rowvar = False)
plt.figure(figsize=(16, 16), dpi=100)
g = sns.heatmap(corr_matrix, vmin = 0.6, vmax = 0.61, cmap="Blues", cbar = True)
plt.savefig('heat_map.png')
plt.show()

# Finding the product couples
# Firstly, we set the diagonal elements to 1 (as they should equal,
# but sometimes they are 0.9999 right now)
for i in range(len(corr_matrix[0])):
    corr_matrix[i][i] = 1

# We can then find the values between 0.6 and 1 (exclusive), those are the
# relevant correlations over the treshold value
corr_list = [] # List of indices for which we have a high enough correlation
for i in range(len(corr_matrix[0])):
    temp_list = (corr_matrix[i] > 0.6) & (corr_matrix[i] < 1)
    corr_list.append([i for i, x in enumerate(temp_list) if x])

prod_couples = []
# In this loop we find the combination of top20 products with lower products
```

```
for i in top20["product_id"]:
    if len(corr_list[i]) > 0: # If there is an element in corr_list
        for j in range(0, len(corr_list[i])): # Loop over it
            if (sum(top20["product_id"] == corr_list[i][j]) == 0): # Not in top 20%
                prod_couples.append([i, corr_list[i][j]])
# In this loop we find the combination of middle30 products with lower products
for i in middle30["product_id"]:
    if len(corr_list[i]) > 0: # If there is an element in corr_list
        for j in range(0, len(corr_list[i])): # Loop over it
            if (sum(middle30["product_id"] == corr_list[i][j]) == 0): # Not in middle 30%
                prod_couples.append([i, corr_list[i][j]])

print("Number of products couples between classes is", len(prod_couples))

# Highlighting these cells in the heatmap
plt.figure(figsize=(16, 16), dpi=100)
g = sns.heatmap(corr_matrix, vmin = 0.6, vmax = 0.61, cmap="Blues", cbar = True)
for i in prod_couples:
    g.add_patch(Rectangle((i[0], i[1]), 1, 1, fill=False, edgecolor='purple',
                          lw=2))
plt.savefig('heat_map_marked.png')
plt.show()

##### Optimization ######
# Calculating the average daily profit loss per product, which is the demand
# loss percentage times the average daily profit
bottom50["AvgProfitLoss"] = 0.5 * bottom50["AvgDailyProfit"]
middle30["AvgProfitLoss"] = 0.3 * middle30["AvgDailyProfit"]
top20["AvgProfitLoss"] = 0.2 * top20["AvgDailyProfit"]

# The dataframe that will be used in all optimization procedures
dfMerged = pd.concat([top20, middle30, bottom50], axis = 0) # Merging the dataframes
dfMerged["BoxesNeeded"] = -(dfMerged["Basestock"] // -dfMerged["NrProdPerBox"])
dfMerged = dfMerged.sort_values(by = ["product_id"])

dfMerged["Ratio"] = dfMerged["AvgProfitLoss"] / dfMerged["BoxesNeeded"]
dfMerged = dfMerged.sort_values(by = ["product_id"])

## Heuristic 1:
# In this heuristic, we will place a base-stock level of products in boxes
# for the product with the highest AvgProfitLoss, then fill boxes with the
# second highest AvgProfitLoss etc. until all boxes are full
# Flaws:
# 1. Assumes that the product_ids follow on each other (so 1,2,3 not 1,2,5 etc.)
# 2. Background: Correlations calculated between classes based on Profit, not
# products that on average have the highest Profit Loss
```

```python

# We use ceiling division here, but since this is not defined with a simple
# operator in python, we use the floor division operator in a smart way

def heuristic1(dataframe, boxes):
    boxes_available = boxes
    dfPop = dataframe.copy()  # No reference!
    dfPop = dfPop.sort_values(by=["AvgProfitLoss"], ascending=False)
    # dfPop = dfMerged.copy() # No reference!

    prod_ids_first = [] # This will contain the products that are stored in
                        # the first warehouse
    i = 0
    while i < dfPop.shape[0]: # dfPop will contain the products that are not in
                              # the first warehouse yet
        prod_id = dfPop.iloc[i, 0]
        corr_prods = []

        # In this loop we check if we need to add other products to the first
        # warehouse too
        for j in range(0, len(prod_couples)):
            if prod_id == prod_couples[j][0]: # First element is highest sale value
                corr_prods.append(prod_couples[j][1] - 1)
        if len(corr_prods) > 0:
            corr_prods.append(prod_id - 1)
            row_indices = corr_prods
        else:
            row_indices = [prod_id - 1]

        tot_size_needed = np.sum(dataframe.iloc[row_indices, -2])
        # Total size needed to fit the product and its correlated products

        if (tot_size_needed <= boxes_available): # If we have enough room available
            # We can assign the products to the boxes
            boxes_available -= tot_size_needed

            for k in range(0, len(row_indices)): # Loop over the row_indices
                # Remove the indices from the dataframe such that we do not
                # assign the product to a box twice
                dfPop.drop(dfPop[(dfPop['product_id'] == row_indices[k] + 1)].index,
                           inplace=True)
                prod_ids_first.append(row_indices[k] + 1)

            i -= 1 # Since we removed a row at place i, what was row i + 1 is now
            # row i and therefore we want i to stay the same. Since we add one later
            # we place -1 here. Notice that the rows after row i that were removed
```

```
                # (those from the correlated products), are irrelevant for our choice of i

            # Small if statement to make sure we do not run the loop longer than necessary
            if (boxes_available == 0):
                break

            i += 1
    return prod_ids_first

# Now we also find the products that are stored in the second warehouse:
prod_ids_first = heuristic1(dfMerged, 960)
prod_ids_first.sort()
prod_ids_second = [x for x in range(1, nrow + 1) if x not in prod_ids_first]

# The actual loss made with this policy:
total_loss = 0
for i in range(0, len(prod_ids_second)):
    total_loss += dfMerged.iloc[prod_ids_second[i] - 1, -3]
# Can do this since dfMerged is sorted on product id
print("The total daily cost of implementing Heuristic 1 is: €", round(total_loss, 2),
      sep = "")



## Heuristic 2:
# In this heuristic, we will place a base-stock level of products in boxes
# for the product with the highest ratio AvgProfitLoss / BoxesNeeded, then
# fill boxes with the second highest ratio etc. until all boxes are full
# Flaws:
# 1. Assumes that the product_ids follow on each other (so 1,2,3 not 1,2,5 etc.)
# 2. Does not consider the ratio of the correlated products
# 3. Background: Correlations calculated between classes based on Profit, not
# products that on average have the highest Profit Loss
def heuristic2(dataframe, boxes):
    boxes_available = boxes
    dfPop = dataframe.copy() # No reference!
    dfPop = dfPop.sort_values(by = ["Ratio"], ascending = False) # No reference!

    prod_ids_first = [] # This will contain the products that are stored in
    # the first warehouse
    i = 0
    while i < dfPop.shape[0]: # dfPop will contain the products that are not in
                               # the first warehouse yet
        prod_id = dfPop.iloc[i, 0]
        corr_prods = []

```

```
        # In this loop we check if we need to add other products to the first
        # warehouse too
        for j in range(0, len(prod_couples)):
            if prod_id == prod_couples[j][0]: # First element is highest sale value
                corr_prods.append(prod_couples[j][1] - 1)
        if len(corr_prods) > 0:
            corr_prods.append(prod_id - 1)
            row_indices = corr_prods
        else:
            row_indices = [prod_id - 1]

        tot_size_needed = np.sum(dataframe.iloc[row_indices, -2])
        # Total size needed to fit the product and its correlated products
        # Notice the -2 now since we added a column to dfMerged


        if (tot_size_needed <= boxes_available): # If we have enough room
            # available we can assign the products to the boxes
            boxes_available -= tot_size_needed

            for k in range(0, len(row_indices)): # Loop over the row_indices
                # Remove the indices from the dataframe such that we do not assign
                # the product to a box twice
                dfPop.drop(dfPop[(dfPop['product_id'] == row_indices[k] + 1)].index,
                            inplace=True)
                prod_ids_first.append(row_indices[k] + 1)

            i -= 1  # Since we removed a row at place i, what was row i + 1 is now
            # row i and therefore we want i to stay the same. Since we add one later
            # we place -1 here. Notice that the rows after row i that were removed
            # (those from the correlated products), are irrelevant for our choice of i

        # Small if statement to make sure we do not run the loop longer than necessary
        if (boxes_available == 0):
            break

        i += 1

    return prod_ids_first

prod_ids_first = heuristic2(dfMerged, 960)
prod_ids_first.sort()
# Now we also find the products that are stored in the second warehouse:
prod_ids_second = [x for x in range(1, nrow + 1) if x not in prod_ids_first]

# The actual loss made with this policy:
```

```
total_loss = 0
for i in range(0, len(prod_ids_second)):
    total_loss += dfMerged.iloc[prod_ids_second[i] - 1, -3]
# Can do this since dfMerged is sorted on product id
print("The total daily cost of implementing Heuristic 2 is: €",
      round(total_loss, 2), sep = "")

## Knapsack problem:
def Knapsack(dataframe, boxes):
    m = Model("Knapsack problem")
    x = m.addVars(nrow, vtype=GRB.BINARY)
    c = dataframe["AvgProfitLoss"] # Costs of adding product to second warehouse
    p = dataframe["BoxesNeeded"] # Boxes needed per product

    # The constraint that makes sure no more than 960 boxes are used
    m.addConstr(sum(p[j] * (1 - x[j]) for j in range(len(x))) <= boxes)

    # The constraints that make sure correlated products are in the same warehouse
    for i,j in prod_couples:
        m.addConstr(x[i] * (1 - x[j]) == 0)

    m.setObjective(sum(c[j] * x[j] for j in range(len(x))), GRB.MINIMIZE)
    m.setParam('OutputFlag', 0)
    m.optimize()

    return m.X

prod_ids_second = Knapsack(dfMerged, 960) # Recall our definition specified
# the x-variables as the products stored in the second (!) warehouse
prod_ids_second = [i + 1 for i in range(0, len(prod_ids_second))
                   if prod_ids_second[i] > 0]
prod_ids_first = [x for x in range(1, nrow + 1) if x not in prod_ids_second]

total_loss = 0
for i in range(0, len(prod_ids_second)):
    total_loss += dfMerged.iloc[prod_ids_second[i] - 1, -3]
print("The total daily cost of implementing the Knapsack solution is: €",
      round(total_loss, 2), sep = "")
print("The products stored in the first warehouse are:", prod_ids_first)


#### Senstivity Analysis ####
# On the available boxes
dfSensCost = pd.DataFrame({"Percentage": [], "Heuristic 1": [],
                           "Heuristic 2": [], "Knapsack": []})
for i in range(0, 41):
```

```
    print(i)
    percentage = i * 0.05
    boxes_available = 960 * percentage
    h1_first = heuristic1(dfMerged, boxes_available)
    h2_first = heuristic2(dfMerged, boxes_available)
    k_second = Knapsack(dfMerged, boxes_available)
    k_second = [i + 1 for i in range(0, len(k_second)) if k_second[i] > 0]
    h1_second = [x for x in range(1, nrow + 1) if x not in h1_first]
    h2_second = [x for x in range(1, nrow + 1) if x not in h2_first]
    h1_costs = 0
    h2_costs = 0
    k_costs = 0
    for i in range(0, len(k_second)):
        k_costs += dfMerged.iloc[k_second[i] - 1, -3]
    for i in range(0, len(h1_second)):
        h1_costs += dfMerged.iloc[h1_second[i] - 1, -3]
    for i in range(0, len(h2_second)):
        h2_costs += dfMerged.iloc[h2_second[i] - 1, -3]
    cost_frame = pd.DataFrame({"Percentage": [percentage],
                               "Heuristic 1": [h1_costs],
                               "Heuristic 2": [h2_costs],
                               "Knapsack": [k_costs]})
    dfSensCost = pd.concat([dfSensCost, cost_frame], ignore_index=True, axis = 0)

print(dfSensCost)

plt.plot(dfSensCost.iloc[:, 0], dfSensCost.iloc[:, -1])
plt.xlabel("Percentage of total (960 boxes)")
plt.ylabel("Average daily profit loss")

for i in range(0, 41):
    plt.scatter(dfSensCost.iloc[i, 0], dfSensCost.iloc[i, -1], color = "purple")
plt.savefig('Costs_knapsack.png')
plt.show()

dfSensCost["K-h1"] = dfSensCost["Heuristic 1"] - dfSensCost["Knapsack"]
dfSensCost["K-h2"] = dfSensCost["Heuristic 2"] - dfSensCost["Knapsack"]

plt.plot(dfSensCost.iloc[:, 0], dfSensCost.iloc[:, -2], color = "green",
         label = "Heuristic 1")
plt.plot(dfSensCost.iloc[:, 0], dfSensCost.iloc[:, -1], color = "red",
         label = "Heuristic 2")
plt.xlabel("Percentage of total (960 boxes)")
plt.ylabel("Cost difference (euros)")
plt.legend(loc ="upper right")
plt.savefig('Cost_difference.png')
```

```
plt.show()

# On the average daily profit loss
dfSensCost2 = pd.DataFrame({"Percentage": [], "Knapsack": []})
for i in range(0, 41):
    print(i)
    percentage = i * 0.05
    dfCopy = dfMerged.copy()
    dfCopy["AvgProfitLoss"] = percentage * dfCopy["AvgProfitLoss"]
    k_second = Knapsack(dfCopy, 960)
    k_second = [i + 1 for i in range(0, len(k_second)) if k_second[i] > 0]
    k_costs = 0
    for i in range(0, len(k_second)):
        k_costs += dfCopy.iloc[k_second[i] - 1, -3]
    cost_frame = pd.DataFrame({"Percentage": [percentage], "Knapsack": [k_costs]})
    dfSensCost2 = pd.concat([dfSensCost2, cost_frame], ignore_index=True, axis = 0)

plt.plot(dfSensCost2.iloc[:, 0], dfSensCost2.iloc[:, 1])
plt.xlabel("Percentage of average daily profit loss")
plt.ylabel("Average daily profit loss")

print(dfSensCost2)
for i in range(0, 41):
    plt.scatter(dfSensCost2.iloc[i, 0], dfSensCost2.iloc[i, 1], color = "purple")
plt.savefig('Costs_knapsack_2.png')
plt.show()
```

DataFrameConstruction

```python
import statistics
import pandas as pd
from elasticsearch import Elasticsearch
from pandas import json_normalize

import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
# Ignores unverified https request warning

import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
# Ignores DeprecationWarnings

es = Elasticsearch( # Specifies the elasticsearch "location" to store the data
    hosts="https://elastic:pancakes@localhost:9200/",
    verify_certs=False
)

## Loading in margins and dimensions from csv to pandas dataframe
basepath = "C:/Users/lucas/OneDrive/Documenten/RUG/Msc Year 1/Trimester 1" \
           "/Data analysis and programming for OM/Assignment"
margins = pd.read_csv(basepath + "/margins.csv")
dimensions = pd.read_csv(basepath + "/dimensions.csv")

search_body = { # search_body that aggregates on product_id,
    # so giving the demand per product over the whole period
    "size": 0,
    "aggs": {
        "PerProduct": {
            "terms": { # Here we specify that we aggregate on days
                "field": "product_id",
                "size": 10000,
                "order": {
                    "_key": "asc"
                }
            }
        }
    }
}

result = es.search(index="sales", body=search_body)
dfProducts = json_normalize(result['aggregations']['PerProduct']['buckets'])
dfProducts.rename(columns={'key': 'product_id'}, inplace=True)
```

```
search_body = { # search_body that aggregates on days, so giving the
                # demand per day over all products
    "size": 0,
    "aggs": {
        "PerDay": {
            "terms": { # Here we specify that we aggregate on days
                "field": "day",
                "size": 10000,
                "order": {
                    "_key": "asc"
                }
            }
        }
    }
}

result = es.search(index="sales", body=search_body)
days = result['aggregations']['PerDay']['buckets']
ndays = len(days)
df = pd.DataFrame(list(range(1, ndays + 1)), columns=["Day"])
# Dataframe with the days (1 to 730 in this case). We specify this one, as we
# want to merge our dataframes such that if we have no demand on say day 5,
# we get a zero there instead of ignoring that row

for index, row in dfProducts.iterrows():
    search_body = {
        "size": 10000,  # Needs to be larger than number of product orders
        'query': {  # Here we find the orders with product number
                    # row["product_id"]
            'term': {
                'product_id': row["product_id"]
            }
        },
        "aggs": {  # Here we find the orders per day for said product
            "OrdersPerDay": {
                "terms": {
                    "field": "day",
                    "size": 1000,
                    "order": {
                        "_key": "asc"
                    }
                }
            }
        }
    }
    result = es.search(index="sales", body=search_body)
```

```

    secondFrame = pd.DataFrame(result["aggregations"]["OrdersPerDay"]["buckets"])
    # The orders per day
    secondFrame = secondFrame.rename(columns={"key": "Day", "doc_count": "ID" +
                                        str(row["product_id"])})

    df = df.merge(secondFrame, how="left", on=["Day"]).fillna(0)
    # Constructs a dataframe with Orders per day, putting
    # 0 at the places where we had no orders.

Prod_list_mean = []
Prod_list_std = []
for i in range(1, df.shape[1]):
    Prod_list_mean.append(statistics.mean(list(df["ID" + str(i)])))
    Prod_list_std.append(statistics.stdev(list(df["ID" + str(i)])))

dfProducts["AvgOrders"] = Prod_list_mean
dfProducts["StdOrders"] = Prod_list_std
dfProducts["Margins"] = margins["margin"]
dfProducts["Length"] = dimensions["length"]
dfProducts["Width"] = dimensions["width"]
dfProducts["Height"] = dimensions["height"]

dfProducts.to_pickle("dfProducts.pkl")
df.to_pickle("dfOrderPerDay.pkl")
```

CsvToElastic

```python
from elasticsearch import Elasticsearch, helpers
import uuid
import pandas as pd
import json


def convert_csv_file_to_bufferized_json_lines_list(csv_file_name, buffer_size):
    df = pd.read_csv(csv_file_name)
    json_list = json.loads(df.to_json(orient='records'))

    buffer_list = []
    total_nr_docs = len(json_list)
    for i in range(0, total_nr_docs, buffer_size):
        buffer_list.append(json_list[i: i + buffer_size])
    return buffer_list

def bulk_json(json_buffer, _index):
    for doc in json_buffer:
        # use a `yield` generator so that the data
        # isn't loaded into memory
        if '{"index"' not in doc:
            yield {
                "_index": _index,
                "_id": uuid.uuid4(),
                "_source": doc
            }

def ingest_csv_file_into_elastic_index(csv_file_name,
            elastic_client: Elasticsearch, index_name, buffer_size=5000):
    chunks = convert_csv_file_to_bufferized_json_lines_list(csv_file_name,
                                                            buffer_size=buffer_size)
    for i, buffer in zip(range(len(chunks)), chunks):
        try:
            # print("__TEST: first document from buffer: ", buffer[0])
            response = helpers.bulk(elastic_client, bulk_json(json_buffer=buffer,
                                                              _index=index_name))
            print("bulk_json() RESPONSE for chunk:", i, response)
        except Exception as e:
            print("\nERROR:", e)
```