university of groningen / faculty of economics and business

Data Analysis and Programming for OM

2022 – 2023

FINAL ASSIGNMENT

Tutorial group 3

Name:
Pelle Meuzelaar

Student number:
S3765695

Date: 28-10-2022

NOTE TO READER:

The number of words is 1600. When excluding graphs, title page and appendix the text fitted on 3 pages. Furthermore, I put the steps of the assignment as steps in the report, but the steps in which I comment on the findings are given at the relative steps and not as a separate step. Finally, the code in the appendix has a small font because I aimed to put all the code lines into one line in the appendix as well. Note that the functions are commented so each can be ran separately by uncommenting the function execution command at the bottom of each function

1. Introduction

I have been asked to consult the Belsimpel company in their journey to allocate products to a new warehouse. Throughout this report I will present the methods I applied that led me to the summary my findings. These will be presented in the conclusion.

2. Problem statement

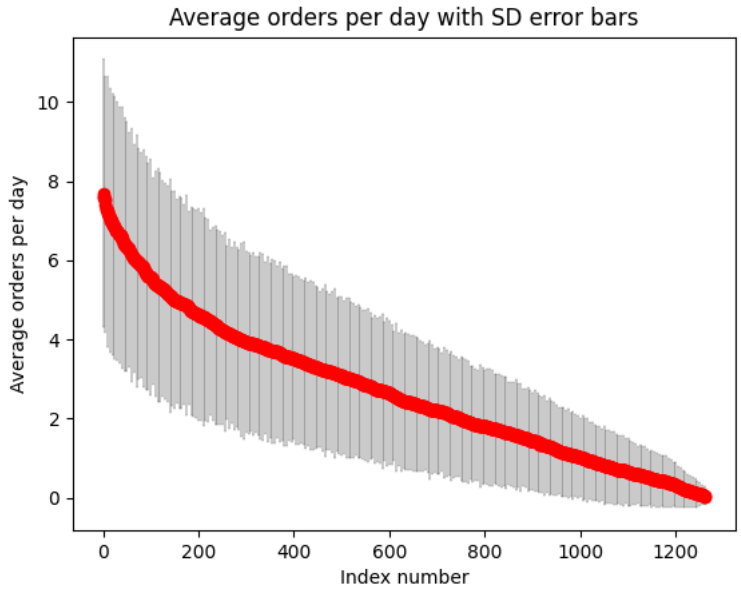
Belsimpel's old warehouse is optimized for fast delivery and moving products to the new warehouse will result in losses. The company wants to expand their operations to this new warehouse, but they have to allocate their products in way that minimizes the profit losses. This report aims to give clarity on how to divide the products between warehouses.

3. Methods and results – *Step 1: data gathering and processing*

I have analyzed the data that was provided as follows.

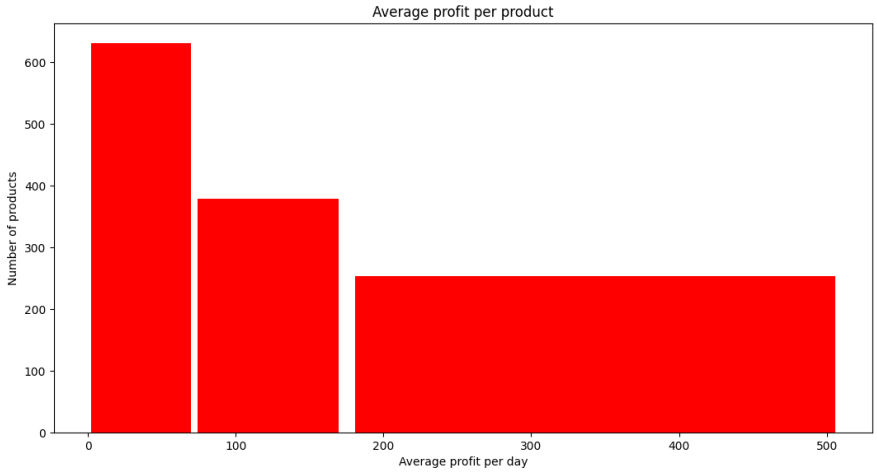
1.1 I gathered the sales per day for each product from the sales data by using Elasticsearch. I loaded the data into Elastic by using the Bulk module. This was done by performing multiple bucket aggregations. I loaded the margins and product dimensions data in Python using Pandas so I could use it later on. These were already formatted to display dimensions and margins per product, so there were no aggregations needed.

1.2 The average and standard deviation per product per day were obtained by adding another aggregation within the bucket of product orders per day using the metrics per bucket method, whereby I added a statistics bucket. I made a list out of the results and added them to a data frame which displays the average orders per day for each product. Finally, I plotted this data on the error bar below. The red line displays the average orders per day and the grey surrounding bars the corresponding standard deviations.



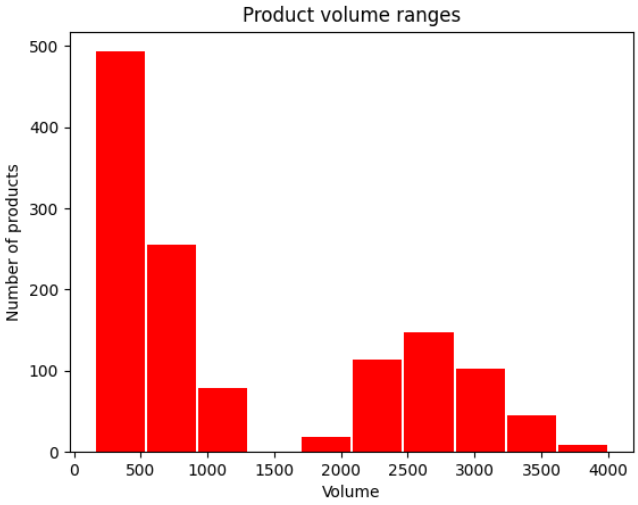
The graph shows that the product demand per day goes up exponentially, but so does its SD. Namely, the higher the average daily demand for a product, the more it will fluctuate. Note that the x-axis does not reflect the actual product number, but it depicts the products relative to each other.

1.3 To calculate average daily profit I multiplied the average orders per product per day column with the corresponding margin. I then plotted a histogram and I used three bins that correspond to the product classes, the bins are however unequal in size because I needed to put the top 20%, 30% and 50% revenue generating products.



The number of product bins (3) cover the ranges of profits as specified by the 3 product classes. The bin (class) containing the most products is the one with the lowest average profits. This implies that Belsimpel's products are largely products have low average daily profits (or: low margins, as we will see soon). Furthermore, the wide right side bin shows that the margins (and profits) of the high profit category differ broadly, whereas the left bin shows little difference between margins (and thus profits).

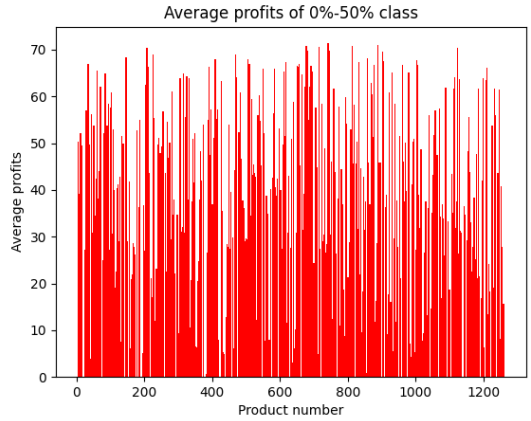
1.4 Next, I used the dimensions data frame that I made and performed computations with the columns to get the volume per product. The Histogram on the left shows that Belsimpel's stock largely consists of small products, but there is also a reasonable spike in medium sized products. This reflects their main sales categories: the main portion is smartphones (small, <1250 cm3), then the main portion becomes tablets (medium sized +/-2750).

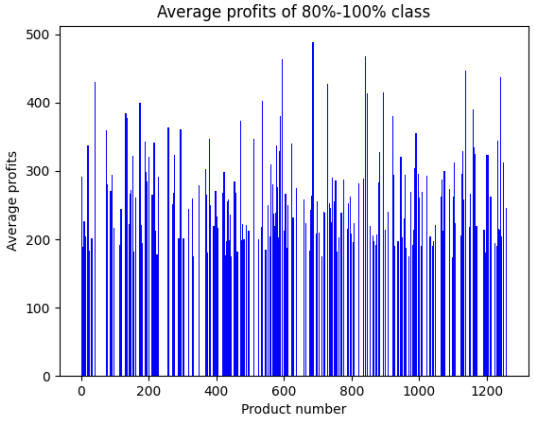


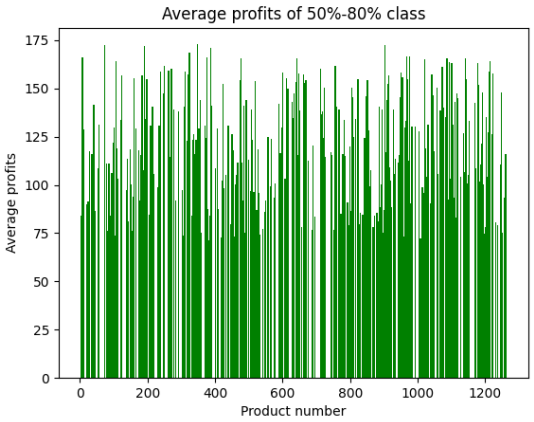
1.6 I proceeded to create a function that shows a list of products in each class. I did so by selecting the product if it matched the previously created bin value per class. This allowed me to create the red, blue and green graphs, which display the average profits of the low, medium and high class respectively.

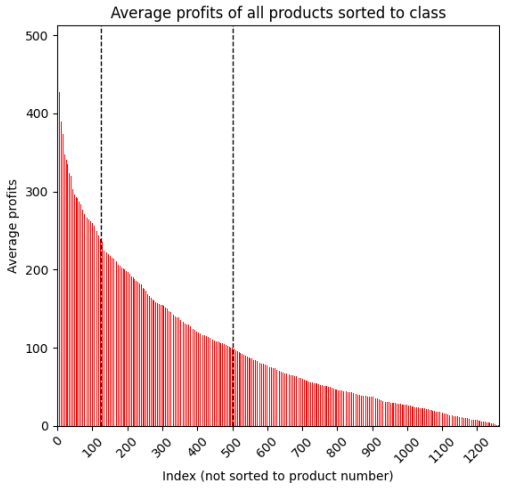
1.7 Note that for the 3 graphs that display the profits per class, the empty space between the bars represents products from other classes that are not in the current plot. If they were to be put into one plot, there would be no empty spaces between bars. From the plots it becomes clear that products within each class are quite evenly distributed in terms of average profit per product. However note the difference in density for the higher and lower plots. This stems from the fact that there are simply much less products in the upper class than the lower class.

1.8 The plot with all classes (sorted) shows the proportions per class. Notice again how little products there are in the top 20% class and how many there are in the lower 50% class. Also note how high the profits of the high class are compared to the lower: the average profit of the former seems to lie at about 275 whereas the latter only has an average of around 50.





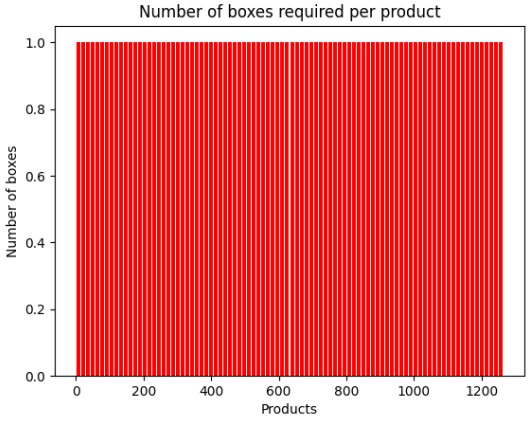




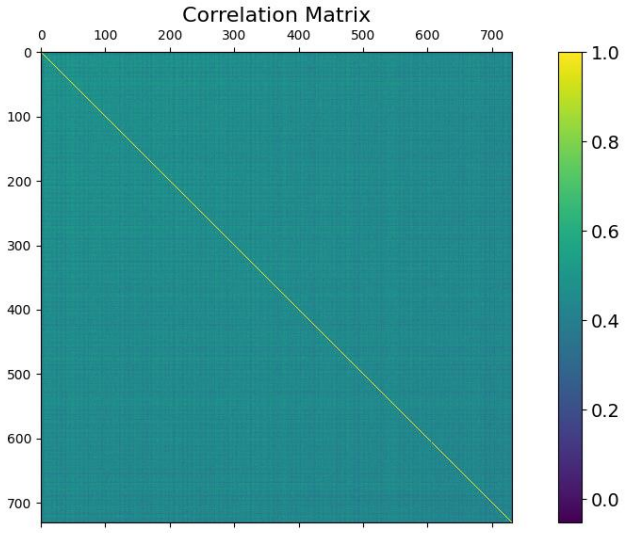
1.10 Next up, I formulated the replenishment interval. Given that this is one week for all products, I simply multiplied the average demand per day per product by 7 to get the interval. The same goes for the standard deviation, except I had to take the square root of the standard deviation and days before multiplying.

1.11 After this, the required base stock was calculated by computing the formula $\mu+z\sigma$ for each product class with different z-values. I set the result of this as a column given that the bin variable is equal to the corresponding class.

1.12 I also computed the number of boxes required by taking the volume data frame and comparing its values with the pick-up box volume of 28800 cm3 (90 % of the pick-up box dimensions). It shows that all products fit within 1 pick-up box. The height of the plot was set to the column with the required number of boxes: notice how it automatically took 1 as upper limit, since there are no products requiring 2 boxes.



1.14 After this, I computed the correlations between the demand per day values by transposing the data frame that was generated in step 1.1 (where I got the orders per day per product). After transposing, I obtained a dataset in which the days are displayed as columns and the products as rows (so in total there are 1263 * 730 datapoints). I then displayed these in the following matrix.



From the matrix, it can be seen that most product orders with a coefficient of about 0.5. This means that the days per product are nor linearly related This means that if orders for a certain product go up, order of the matched product can be expected to go up as well.

1.15 From the results of the matrix I made a list of product couples by transforming it into a list. I then added this to the data frame along with a new column that displays a 1 if the correlation is equal or higher than 0.6. Otherwise it will display a 0.

1.16 I was not able to plot a matrix with the product couples, as I found that there were no products that correlated more than 0.6. Hence, there is no additional correlation matrix.

4. Methods and results – Step 2: optimization

2.1 I computed the average daily profit per product in step 1. Now, according to the given data, it is anticipated that there will be a loss of sales of 20, 30 and 50% on average daily profits in the low, medium and high product classes. To compute the expected losses when transferring to the new warehouse I located the profit column and multiplied the above percentage as a factor to get the average losses per product per day.

2.2 Following this, I was able to make a heuristic that displays the products to be chosen for warehouse 1 based on their expected profit losses. I did this by sorting the profit losses, displaying the highest loss first and outputting these to a list.

2.3 I then wrote a heuristic that displays the sum of losses by means of the ratio of average daily profit to required number of boxes. I computed this by dividing the columns with each other. After obtaining the ratio I could again sort the products and output the number of products to be selected for warehouse 1.

2.4 As a last method, I wrote a program using Gurobi that displays the products to be stored at the old warehouse. It has a 0-1 Knapsack function that maximizes the number of products stored in the current warehouse. This method is useless because it will always maximize product losses.

2.5 Finally, I computed a function that outputs a table with all three results, which is the following:

method_number	total_daily_loss_per_method
1.0	14755.79803957004
2.0	14755.79803957004
3.0	47558.54164672836

The table shows that method 1 (sorted to loss) and 2 (sorted to ratio) both output a daily loss of 14.755. This is because the number of required boxes is 1 for all products, therefore the ratio between average loss and required boxes will always be divided by 1 and thus the ratio will always be equal to the total daily loss. Then both methods will output the same result. The final method shows the maximum profit loss and should not be considered.

5. Conclusion

Based on the above output I formulated the following advice. Belsimpel achieves highest profits on the small 20% class from phones: these generate 80% of profits but make up around 250 out of 1263 products. These products are high in demand and have high margins. Most of Belsimpel's products are phones with a low box volume and other items such as tablets closely follow (also having low volumes). This is the reason why all products currently fit in 1 pick-up box. As this is the case, we can simply allocate products to warehouses based on their capacity of pick-up boxes. Furthermore, the correlation between products must also be considered. However, there are no products with a correlation higher than 0.6, which means that all products can be stored separately. This means that correlation does not have to be accounted for and we can just look at box capacity.

The old warehouse can currently hold 960 boxes and we Method 1 above shows the losses if the products with the highest potential losses were to be put in the old warehouse. Method 2 shows the same, since there are no products requiring 2 boxes. Method 3 should be disregarded as it will allocate all products to warehouse 1 (this is impossible).

6. Appendix – code

```
# Author: Pelle Meuzelaar
# Date: 12-10-2022
# Assignment: Belsimpel warehouse case


from elasticsearch import Elasticsearch, helpers
import urllib3
import warnings
import csv
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import json
import numpy as np
from gurobipy import Model, GRB, quicksum
import seaborn as sns
from matplotlib.path import Path
from matplotlib.patches import Rectangle
from matplotlib.patches import PathPatch
matplotlib.use('TkAgg') #errorsolving

""" STEP 1.1: CREATE INDEX AND GATHER TOTAL DEMAND """

def elasticsearch_setup():
  # Disable annoying warnings before the start of the code:
  urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
  warnings.filterwarnings("ignore", category=DeprecationWarning)
  warnings.filterwarnings("ignore", category=UserWarning)

  # Load in elasticsearch client
  es = Elasticsearch("https://localhost:9200", ca_certs=False, verify_certs=False,
                     http_auth=('elastic', 'j3nfTPHpJxz5iYTVMu8V'))
  return es

# elasticsearch_setup()

def create_product_index():
  es = elasticsearch_setup() # Call from es function
  # Define the mapping of the index to be created
  settings = {
    "mappings": {
      "properties": {
        "day": {
          "type": "long"
        },
        "product_id": {
          "type": "long"
        },
        "product_orders_per_day": {
          "type": "long"
        }
      }
    }
  }

  # Create an empty index
  es.indices.create(index="products", body=settings)

  # Load the csv file into the elastic index using bulk command
  with open('sales.csv') as f:
    reader = csv.DictReader(f)
    helpers.bulk(es, reader, index='products')

  # Confirm that the index with product orders per day is created
```

```
  print("index 'products' is created")

  # To delete the index, uncomment the following line
  # es.indices.delete(index='products', ignore=[400, 404])

# create_product_index() #disabled, because you only need to run create index once

def df_total_demand_per_product():
  es = elasticsearch_setup() # Call from es function

  #Perform a terms aggregation to display the buckets for the product orders per day (730),
  search_body = {
    "size": 0,
    "aggs": {
      "total_product_demand": {
        "terms": {
          "field": "product_id",
          "size": 1200
          },
          "aggs": {
            "transactions_per_day": {
              "histogram": {
                "field": "day",
                "interval": 1
            }
          }
        }
      }
    }
  }
  # Note: 1200 is the total number of keys that could possibly be in a bucket because it is the
  # total product number available. However, ES/kibana will not allow that many results you have
  # to it with the set.max.buckets command

  # Print the number of orders per product of the above query
  result = es.search(index="products", body=search_body)

  # Print number of orders per product as we discovered using the query
  print("query results: ",  json.dumps(result, indent=1))

  # Make a list of products per day
  orders_per_day = []

  # Iterate over bucket and subbucket to append their data to the empty list of lists
  for bucket in result["aggregations"]["total_product_demand"]["buckets"]: #iterate the "products"
    id_count = bucket["key"] #create a list to be included in the above list of lists
    for bucket in bucket["transactions_per_day"]["buckets"]: #iterate over the sub-buckets
      orders_per_day.append([id_count, bucket["key"], bucket["doc_count"]]) #append the data

  # Put list into dataframe and print them sorted by product_id
  df = pd.DataFrame(orders_per_day, columns=["product_id", "day", "orders"])
  df.sort_values(by=["product_id"], inplace=True)
  return(df)

# df_total_demand_per_product()

""" STEP 1.2: AVERAGE AND SD OF DEMAND PER DAY """

def df_product_orders_per_day():
  es = elasticsearch_setup() # Call from es function

  # Perform a terms aggregation to display the buckets for the product orders per day (730)/stats
  search_body = {
    "size": 0,
    "aggs": {
      "total_product_demand": {
```

```
        "terms": {
          "field": "product_id",
          "size": 1263
        },
        "aggs": {
          "transactions_per_day": {
            "histogram": {
              "field": "day",
              "interval": 1,
            }
          },
          "data_description_per_product": {
            "extended_stats_bucket": {
              "buckets_path": "transactions_per_day._count",
              "gap_policy": "insert_zeros"
            }
          }
        }
      }
    }
  }
  # I added stats bucket in order to count the average number of orders per day per product & other stats

  # Print the number of orders per product of the above query
  result = es.search(index="products", body=search_body)

  # Make a list of average per product
  statistics_per_product = []

  # Iterate over bucket and subbucket to append their data to the empty list of lists
  for bucket in result["aggregations"]["total_product_demand"]["buckets"]: #iterate the products
    product_id = bucket["key"]
    statistics_per_product.append([product_id,
                                   bucket["data_description_per_product"]["avg"],
                                   bucket["data_description_per_product"]["std_deviation"]])

  # Put list into dataframe and print them sorted by highest average order per day
  df = pd.DataFrame(statistics_per_product, columns=["product_id", "avg_orders_per_day", "standard_deviation"])
  df = df.sort_values("avg_orders_per_day", ascending=False).reset_index(drop=False)
  return df

# df_product_orders_per_day()

def plot_errorbar_avg_demand():
  df = df_product_orders_per_day() # Call df from above function

  # Give the parameters for the errorbar plot
  x = df.index
  y = df["avg_orders_per_day"]
  yerror = df["standard_deviation"]

  # Plot and save the data onto an errorbar
  plt.errorbar(x=x, y=y, yerr=yerror, color="red", fmt="o", ecolor="black", elinewidth=0.3,
               capsize=1, errorevery=5, capthick=0.1)
  plt.title("Average orders per day with SD error bars")
  plt.xlabel("Index number")
  plt.ylabel("Average orders per day")
  plt.show()

# plot_errorbar_avg_demand()

""" STEP 1.3: PROFIT COMPUTATION """

def profit_computation():
```

```
195.   df = df_product_orders_per_day() # Call df from above function
196.
197.   # Make new dataframe from margins.csv
198.   df2 = pd.read_csv("margins.csv")
199.
200.   # Add the margin column from dataframe 2 to a new dataframe:
201.   margins = df2["margin"]
202.   df = pd.concat([df, margins], axis=1, ignore_index=False)
203.
204.   # Multiply the average number of orders per product with the margin per product
205.   df["avg_daily_profit"] = df["avg_orders_per_day"] * df["margin"]
206.
207.   # Sort new dataframe to display highest daily profit per product first
208.   df = df.sort_values("avg_daily_profit", ascending=False)
209.   return(df)
210.
211. # profit_computation()
212.
213. def bin_finder():
214.   df = profit_computation() # Call df from function
215.
216.   # Determine the array to put in the bin by determining the the i'th quantile (namely 50, 70, 80)
217.   bin_ranges = [df.avg_daily_profit.min(), df.avg_daily_profit.quantile(0.5),
218.                 df.avg_daily_profit.quantile(0.8), df.avg_daily_profit.max()]
219.   return bin_ranges
220.
221. # bin_finder()
222.
223. def plot_histogram_avg_profit():
224.   df = profit_computation()  # Call from functions
225.   bin_ranges = bin_finder()
226.
227.   # Show average profit per product per day in a histogram, input above range for the bin parameter
228.   plt.hist(df["avg_daily_profit"], bins=bin_ranges, rwidth=0.95, color="red")
229.   plt.title("Average profit per product")
230.   plt.ylabel("Number of products")
231.   plt.xlabel("Average profit per day")
232.   plt.show()
233.
234. # plot_histogram_avg_profit()
235.
236. """ STEP 1.4: VOLUME COMPUTATION """
237.
238. def volume_computation():
239.   # Make new dataframe from margins.csv
240.   df2 = pd.read_csv("dimensions.csv")
241.
242.   # Multiply the length, width and height to get the volume per product in a column
243.   df2["volume"] = df2["length"] * df2["width"] * df2["height"]
244.
245.   # Sort new dataframe to display highest daily profit margin products first
246.   df2 = df2.sort_values("product_id", ascending=True)
247.
248.   return df2
249.
250. # volume_computation()
251.
252. def plot_volume_histogram():
253.   df2 = volume_computation() # Call from function
254.
255.   # Show volume per product per day in a histogram
256.   plt.hist(df2["volume"], rwidth=0.95, color="red")
257.   plt.title("Product volume ranges")
258.   plt.ylabel("Number of products")
259.   plt.xlabel("Volume")
260.   plt.show()
```

```
261.
262. # plot_volume_histogram()
263.
264. """ STEP 1.6: PRODUCT CLASSES """
265.
266. def products_in_each_class():
267.   df = profit_computation() # Call from functions
268.   bin_ranges = bin_finder()
269.
270.   # We found the quartiles using the bin_finder() function, so we bin the df accordingly
271.   df["binned"] = pd.cut(df["avg_daily_profit"], bins=bin_ranges, labels=False) # classes are 0, 1, 2
272.   df.at[1262, "binned"] = 0 # There is a NaN value that throws an error
273.
274.   return df
275.
276. # products_in_each_class()
277.
278. def print_list_products_in_each_class():
279.   df = products_in_each_class()
280.
281.   # Make variables
282.   product_class_low = (df.loc[df["binned"] == 0, "product_id"])
283.   product_class_medium = (df.loc[df["binned"] == 1, "product_id"])
284.   product_class_high = (df.loc[df["binned"] == 2, "product_id"])
285.
286.   # Get the list of products
287.   print("the lists of product classes: (left = index and right = product_id):",
288.   "\nProduct class 0-50%:\n", product_class_low,
289.   "\nProduct class 50-80%:\n", product_class_medium,
290.   "\nProduct class 80-100%:\n", product_class_high)
291.
292. # print_list_products_in_each_class()
293.
294. """ STEP 1.7: PRODUCT CLASS BAR CHARTS """
295.
296. def plot_product_low_class_chart():
297.   df = products_in_each_class() # Call from functions
298.
299.   # Input the parameters
300.   x = df.loc[df["binned"] == 0, "product_id"]
301.   height = df.loc[df["binned"] == 0, "avg_daily_profit"]
302.
303.   # Plot the bar chart for the lower class
304.   plt.bar(x, width=3.5, height=height, bottom=0, align="center", color="red")
305.   plt.title("Average profits of 0%-50% class")
306.   plt.xlabel("Product number")
307.   plt.ylabel("Average profits")
308.   plt.savefig("/Users/pellemeuzelaar/PycharmProjects/Practical4/bar1.png")
309.
310. # plot_product_low_class_chart()
311.
312. def plot_product_middle_class_chart():
313.   df = products_in_each_class() # Call from functions
314.
315.   # Input the parameters
316.   x = df.loc[df["binned"] == 1, "product_id"]
317.   height = df.loc[df["binned"] == 1, "avg_daily_profit"]
318.
319.   # Plot the bar chart for the middle class
320.   plt.bar(x, width=3.5, height=height, bottom=0, align="center", color ="green")
321.   plt.title("Average profits of 50%-80% class")
322.   plt.xlabel("Product number")
323.   plt.ylabel("Average profits")
324.   plt.savefig("/Users/pellemeuzelaar/PycharmProjects/Practical4/bar2.png")
325.
326. # plot_product_middle_class_chart()
```

```
327.
328. def plot_product_high_class_chart():
329.   df = products_in_each_class() # Call from functions
330.
331.   # Input the parameters
332.   x = df.loc[df["binned"] == 2, "product_id"]
333.   height = df.loc[df["binned"] == 2, "avg_daily_profit"]
334.
335.   # Plot the bar chart for the middle class
336.   plt.bar(x, width=3.5, height=height, bottom=0, align="center", color="blue")
337.   plt.title("Average profits of 80%-100% class")
338.   plt.xlabel("Product number")
339.   plt.ylabel("Average profits")
340.   plt.savefig("/Users/pellemeuzelaar/PycharmProjects/Practical4/bar3.png")
341.
342. # plot_product_high_class_chart()
343.
344. """ STEP 1.8: PROFIT CHART ALL CLASSES """
345.
346. def plot_profits_per_product_sorted():
347.   df = profit_computation()  # Call from functions
348.
349.   # Define the bar chart for all classes, sorted highest profit first
350.   fig, ax = plt.subplots()
351.   df.sort_values("avg_daily_profit", ascending=False)["avg_daily_profit"].plot.bar(
352.     ax=ax, xticks=df.index, rot=45, stacked=False, color="red")
353.
354.   # Set x-axis steps
355.   ax.set_xticks(np.arange(0, len(df.index) + 1, 100))
356.
357.   # Plot vertical lines for classes. These are hardcoded: the values are the product_ids that came
358.   # from the output of the list_products_per_class function
359.
360.   print("put 50% marker at: ", df.query("product_id==742")["index"]) #prod. 742 is border, 123 index
361.   print("put 80% marker at: ", df.query("product_id==347")["index"]) #prod. 347 is border, 500 index
362.   plt.axvline(123, color="k", linestyle="dashed", linewidth=1)
363.   plt.axvline(500, color="k", linestyle="dashed", linewidth=1)
364.
365.   # Set titles and labels and print the boxplot
366.   plt.title("Average profits of all products sorted to class")
367.   plt.xlabel("Index (not sorted to product number)")
368.   plt.ylabel("Average profits")
369.   plt.show()
370.
371. # plot_profits_per_product_sorted()
372.
373. """ STEP 1.10: AVERAGE AND MEAN DEMAND OVER REPLENISHMENT INTERVAL """
374.
375. def avg_and_sd_demand_replenish_interval_low_class():
376.   df = df_product_orders_per_day() # Call df from the first product function
377.
378.   # I need to compute the values m and s per product. I have the m and s per product per day.
379.   # I only need to multiply by the replenish interval (1 week/7 days)
380.   df["avg_demand_replenish_interval"] = df["avg_orders_per_day"] * 7
381.   df["sd_demand_replenish_interval"] = (df["standard_deviation"] * 7) ** (1/2)
382.
383.   df = pd.concat([df,
384.                   df["avg_demand_replenish_interval"],
385.                   df["sd_demand_replenish_interval"]], axis=1, ignore_index=False)
386.
387.   return df
388.
389. # avg_and_sd_demand_replenish_interval_low_class()
390.
391. """ STEP 1.11: COMPUTE BASE STOCK LEVEL """
392.
```

```
def compute_base_stock_level():
  df = avg_and_sd_demand_replenish_interval_low_class() # Call df from the above function
  df2 = products_in_each_class()

  # Add the bin variables as column (because we dit not have it in this df yet)
  df = pd.concat([df, df2["binned"]], axis=1, ignore_index=False)
  df = df.T.drop_duplicates().T # Got an error from importing df, because of duplicate columns

  # Make functions for the base stock formula, z changes to 0.9, 0.95 and 0.99 per class
  base_stock_low = df["avg_demand_replenish_interval"] + 0.90 * df["sd_demand_replenish_interval"]
  base_stock_medium = df["avg_demand_replenish_interval"] + 0.95 * df["sd_demand_replenish_interval"
]
  base_stock_high = df["avg_demand_replenish_interval"] + 0.99 * df["sd_demand_replenish_interval"]

  # Depending on the bin value (i.e. the class), the base stock is calculated and added to the df
  df.loc[df["binned"] == 0, "base_stock"] = base_stock_low
  df.loc[df["binned"] == 1, "base_stock"] = base_stock_medium
  df.loc[df["binned"] == 2, "base_stock"] = base_stock_high

  return df

# compute_base_stock_level()

""" STEP 1.12: COMPUTE PICK UP BOXES """

def pickup_box():
  df2 = volume_computation() # Call df from the above function

  # Compute the volume for the standard box (dimensions given), only 90% can be used!
  volume_standard_box = 0.9 * (40 * 40 * 20)

  # Set up the conditional that determines the amount of boxes required
  df2.loc[df2["volume"] <= volume_standard_box, "required_boxes"] = 1 # If the prod. vol. is </= 1
  df2.loc[df2["volume"] > volume_standard_box, "required_boxes"] = 2 # If vol. is > 1 box, 2 box

  # print(df2.loc[df2["required_boxes"] == 2]) # This returns nothing, no products need 2 boxes
  return df2

# pickup_box()

def plot_box_number():
  df2 = pickup_box() # Call from above function

  #Set parameters
  height = df2["required_boxes"]

  # Show required box number per product in a histogram
  plt.bar(df2["product_id"], height=height, color="red")
  plt.title("Number of boxes required per product")
  plt.ylabel("Number of boxes")
  plt.xlabel("Products")
  plt.show()

# plot_box_number()

""" STEP 1.14: CORRELATION MATRIX """

def transpose_data():
  df = df_total_demand_per_product() # Call the function from the beginning, with total products/day

  # Tranpose the current df so that it displays the days as columns instead of as a group in one row
  df2 = df.pivot(index="product_id", columns="day", values="orders").add_prefix("day ").reset_index(
)
  df.set_index("product_id") #Fix the dataframe
  return df2
```

```
456.
457. # transpose_data()
458.
459. def correlation_matrix():
460.   df2 = transpose_data()  # Call from previous function
461.
462.   # Plot the data in a matrix in python output
463.   corr = df2.corr()
464.   print(corr)
465.   return corr
466.
467. #correlation_matrix():
468.
469. def plot_correlation_matrix():
470.   df = transpose_data() # Call from previous function
471.
472.   # Plot the above correlation matrix as heatmap
473.   fig = plt.figure(figsize=(19, 15))
474.   plt.matshow(df.corr(), fignum=fig.number)
475.   plt.title("Correlation Matrix", fontsize=16);
476.   cb = plt.colorbar()
477.   cb.ax.tick_params(labelsize=14)
478.   plt.show()
479.
480. # plot_correlation_matrix()
481.
482. """ STEP 1.15: PRODUCT COUPLES """
483.
484. def product_couples():
485.   corr = correlation_matrix() # Call from previous function
486.
487.   # Filter the the matrix values to display correlations as a sorted list
488.   s = corr.unstack()
489.   so = s.sort_values(kind="quicksort")
490.   print(so)
491.
492.   # Make a list that displays
493.   df = pd.DataFrame(so, columns=["correlations"])
494.
495.   # Locate the couples that match the threshold 0.6
496.   df.loc[df["correlations"] >= 0.6, "match"] = 1 # If product corr higher than 0.6, input 1 in col
497.   df.loc[df["correlations"] <= -
    0.6, "match"] = 1 # If product has negative 0.6, input 1 in the col
498.   df.loc[df["correlations"] < 0.6, "match"] = 0 # If product has lower than 0.6, input 0 in the col

499.   df.loc[df["correlations"] == 1, "match"] = 0 # If product is 1 (same values), we don't want them
500.
501.   return df
502.
503.   # I WAS NOT ABLE TO TELL WHICH PRODUCTS WERE CORRELATED AND WHICH 1 OF THE 2 WAS IN A HIGH CLASS
504.
505. # product_couples()
506.
507. """ STEP 1.16: PRODUCT COUPLES MATRIX """
508.
509. def plot_product_couples_matrix():
510.   df = product_couples() # Call from above function
511.   corr = correlation_matrix()
512.
513.   # Formulate x/y values for the matched products that are to be marked
514.   coords = df.loc[df["match"] == 0]
515.
516.   # Plot a new correlation matrix and input the match as variable in "patch" plot function to get ma
    rk
517.   fig, ax = plt.subplots()
518.   ax = sns.heatmap(corr, annot=True, linewidths=.5, ax=ax)
```

```
    ax.add_patch(plt.Rectangle((coords), 1, 1, fill=True, edgecolor="red", lw=3))
    plt.show()

# THIS FUNCTION RETURNS NOTHING BECAUSE I FOUND NO MATCHED VARIABLES, IF MATCH = 0 IT WILL WORK

# plot_product_couples_matrix()

""" STEP 2.1: LOSS IN SALES """

def loss_in_profit():
    df = products_in_each_class() # Call df from function

    # It is anticipated that the drop in sales for the three product classes will be 20%, 30%, and 50%
    # avg_daily_profit (that we computed in the dataframe) will drop by the respective percentages
    profit_loss_low = 0.2 * df["avg_daily_profit"]
    profit_loss_medium = 0.3 * df["avg_daily_profit"]
    profit_loss_high = 0.5 * df["avg_daily_profit"]

    # Depending on the bin value (i.e. the class), the base stock is calculated and added to the df
    df.loc[df["binned"] == 0, "avg_daily_profit_loss"] = profit_loss_low
    df.loc[df["binned"] == 1, "avg_daily_profit_loss"] = profit_loss_medium
    df.loc[df["binned"] == 2, "avg_daily_profit_loss"] = profit_loss_high
    return df

# loss_in_profit()

""" STEP 2.2: RANKING BASED ON PROFIT LOSS """

def ranking_on_profit_loss():
    df = loss_in_profit() # Call from above function

    #Sort the dataframe based on profit loss and reset the index so we can count later
    df.sort_values(by=["avg_daily_profit_loss"], inplace=True)
    df.reset_index(drop=True, inplace=True)

    # Make a variable for the sum of total profit loss if these products were used:
    sum_losses = df.loc[df.index <= 960].avg_daily_profit_loss.sum()

    # Return the products that have the lowest profit losses. The max number of products in the cur.
    # warehouse is 960,because only 960 boxes fit and all products require only 1 box
    print("The 960 products with the lowest losses are: \n", df.iloc[0:960, 1]) #1 is product_id colum
n
    print("Using these will result in total average daily profit loss of:", sum_losses)

    return df, sum_losses

# ranking_on_profit_loss()

""" STEP 2.3: RANKING TO RATIO OF PROFIT LOSS AND BOXES """

def ranking_on_ratio_losses_and_boxes():
    df = loss_in_profit() # Call from above functions
    df2 = pickup_box()

    # Calculate the ratio of profit losses to boxes required and put into new column:
    df["ratio"] = df["avg_daily_profit_loss"] / df2["required_boxes"]

    #Sort the dataframe based on profit loss and reset the index so we can count later
    df.sort_values(by=["ratio"], inplace=True)
    df.reset_index(drop=True, inplace=True)

    # Make a variable for the sum of total profit loss if these products were used:
    sum_losses = df.loc[df.index <= 960].avg_daily_profit_loss.sum()

    #Print the boxes to be stored in warehouse 1
```

```
  print("The products with the lowest ratios are: \n", df.iloc[0:960, 1:2]) #1 is product_id column

  print("Using these will result in total average daily profit loss of:", sum_losses)

  return df, sum_losses

# ranking_on_ratio_losses_and_boxes()

""" STEP 2.4: KNAPSACK PROBLEM """

def knapsack():
  df = loss_in_profit() # Call from previous function

  # The upper limit of storage in the storage warehouse 1 may not exceed 960 pick-up boxes (ub)
  # and if 2 products have a correlation higher than 0.6, they have to be stored together (const).
  # However, since none of the products correlate more than 0.6, this is not a constraint.

  # Decision variables (capacity, profit loss, number of products selected):
  p = df["avg_daily_profit_loss"] # P stands for profit loss
  n = len(p) # Number of products that we could potentially lose profits on

  # Call the Gurobi model
  m = Model("Belsimpel")
  x = m.addVars(n, vtype=GRB.BINARY, ub=960, name="Average daily profit loss") # Lower bound is 960

  # Objective (minimize profit loss):
  m.setObjective(quicksum(p[i] * x[i] for i in range(n)), GRB.MAXIMIZE) # We need to display the max
  # loss if the full capacity is used

  # Run the model
  m.optimize()

  # Display the output
  sum_losses_3 = m.ObjVal
  products_selected = []
  for i in range(n):
    if x[i].X > 0.5:
      products_selected.append(i)

  print("If the products", products_selected,
        "are selected, total profit loss will be minimal: ", sum_losses_3)
  return products_selected, sum_losses_3

# knapsack()

""" STEP 2.5: TABLE OF PROFIT """

def print_table_with_solutions():
  sum_losses_1 = ranking_on_profit_loss()
  sum_losses_2 = ranking_on_ratio_losses_and_boxes()
  sum_losses_3 = knapsack()

  # Combine losses from al methods into a list
  losses_each_method = [sum_losses_1, sum_losses_2, sum_losses_3]

  # Put the losses in a pandas dataframe
  df = pd.DataFrame(losses_each_method, columns=["product_id", "total_daily_loss_per_method"])
  df = df.drop(["product_id"], axis=1) # We don't need product_ids in the table
  df.insert(0, "method_number", [1, 2, 3], True) # Insert method number for clarity

  # Plot a table with the solutions from the dataframe
  plt.table(cellText=df.values, colLabels=df.columns, loc="center")
  plt.show()

# print_table_with_solutions()
```