**Screen**: excel

Hi, today we’re going to explore a supply chain logistics problem dataset. We’re going to prepare the data and transform it, demonstrate statistical tools in python that give us further insights into the data, and show some visualizations.

First, let’s understand the data. I have opened up an excel file containing 7 tables separated by each sheet.

The OrderList table contains all orders that needs to be assigned a route. Order ID is the ID of the order made by the customer, and product ID is the specific product ID the customer ordered. The table contains historical records of how the orders were routed and demand satisfied.

The FreightRates table describes all available carriers, the weight gaps for each individual lane and rates associated. “tpt\_day\_cnt” means transportation day count which is the estimated shipping time.

The WhCosts table specifies the cost associated in storing the products in a given warehouse measured in dollars per unit.

The WhCapacities lists warehouse capacities measured in number of orders per day. For example, let’s say Customer 1 requests 10 units of X, Customer 2 requests 20 units of Y. The total number of orders is 2, thus total capacity is 2.

The ProductsPerPlant table lists all supported warehouse-product combinations.

The VmiCustomers lists all special cases, where a warehouse is only allowed to support a specific customer.

The PlantPorts table describes the allowed links between the warehouses and shipping ports. In this dataset, the terms warehouse and plant are used interchangeably. Essentially a warehouse is a plant.

Now, that we’ve seen the data, let’s turn our attention to jupyter notebooks.

**Screen**: jupyter notebooks

First let’s import some packages that we’ll use. (type:

Import numpy as np

Import pandas as pd

Import matplotlib.pyplot as plt

Import seaborn as sns) and run it

(create a new cell) now let’s load our excel file. (type: main\_file = pd.ExcelFile("/Users/timothypark/Documents/portfolios/timpark99.github.io/supply chain analysis/supply chain original.xlsx")) this uses a pandas function to read in the excel document that’s located in whatever directory that you have it saved in. Now underneath that, let’s write, (type: df\_dict = {sheet\_name: main\_file.parse(sheet\_name) for sheet\_name in main\_file.sheet\_names}). This dictionary comprehension reads all the sheets into a dictionary of dataframes. And just to show that dictionary, let’s show the keys which will be the names of the tables. (type: df\_dict.keys()) and run the cell. As you can see, here are the keys from the dictionary which match the names of the tables we looked at previously.

(create a new cell). Let’s now show the shape of each dataframe and check if they have any duplicate or missing values. First let’s loop through the dictionary. (type:

for df\_name, df in df\_dict.items():

print(f"{df\_name} - Shape: {df.shape}")

duplicate\_count = df.duplicated().sum() # this line calculates the total number of duplicate rows.

missing\_values\_count = df.isnull().sum().sum() # finds the total number of missing values across all columns. There are two calls for .sum() because the first one looks at each column, but the second call adds up all the missing value counts from all columns.

if duplicate\_count > 0 or missing\_values\_count > 0:

print(f" >>> {df\_name} - Duplicates: {duplicate\_count}; Missing Values: {missing\_values\_count}")

else:

print(f" >>> {df\_name} has no duplicates or missing values.")

(run the cell). As you can see only the FreighRates table has 3 duplicates.

(create new cell). Let’s try and drop the duplicates for the FreightRates table. To that let’s use the drop duplicates function. (type: FreightRates = df\_dict['FreightRates'].drop\_duplicates()). Now let’s ensure that the dictionary is updated by assigning it back to it’s original dataframe. (type: df\_dict['FreightRates'] = FreightRates). Let’s also change the formatting of the columns in every table to where every letter is capitalized, the spaces are replaced with underscores and the slashes are replaced with underscores. So let’s write, (type):

for df\_name, df in df\_dict.items():

df.columns = [col.strip().replace(' ', '\_').replace('/', '\_').upper() for col in df.columns]

to see if changes were made let’s print out the columns of the OrderList dataframe, (type: print(df\_dict['OrderList'].columns). As you can see, the columns have all changed.

(create new cell) Next I’ll demonstrate the concept of merging dataframes and performing a left join on multiple tables. Similar to a left join in SQL, in python it’s called a merge. So first let’s create a variable to hold the dataframe OrderList. (type: orderList = df\_dict['OrderList']). Then let’s merge the OrderList and FreightRates dataframes, using matching columns as join keys. (type):

orderList = orderList.merge(df\_dict['FreightRates'], left\_on=['CARRIER', 'ORIGIN\_PORT', 'DESTINATION\_PORT'],

right\_on=['CARRIER', 'ORIG\_PORT\_CD', 'DEST\_PORT\_CD'], how='left')

We mergeon 3 key columns which are CARRIER, ORIGIN\_PORT and DESTINATION\_PORT in OrderList, which correspond to CARRIER, ORIG\_PORT\_CD and DEST\_PORT\_CD in FreightRates. How = “left” ensures that all rows in orderList are kept, with matched values from FreightRates, and if the table will be filled with NaN values if no match is found. Now let’s merge orderList with WhCosts. (type):

orderList = orderList.merge(df\_dict['WhCosts'], left\_on='PLANT\_CODE', right\_on='WH', how='left')

again, we see that how = left retains all rows from orderlist, adding whcosts data where available.

Now let’s see if we can create a new column in our orderlist dataframe that shows the cost. We can get cost by multiplying unit quantity with the shipping rate, and adding the storage cost. (type):

orderList['COST'] = (orderList['UNIT\_QUANTITY'] \* orderList['RATE']) + (orderList['UNIT\_QUANTITY'] \* orderList['COST\_UNIT'])

After adding some new columns to our orderlist dataframe, let’s further perform detailed data preparation and transformation for the OrderList dataframe by dropping rows with any NaN values. (type: orderList = orderList.dropna()). Another area that we can clean up is in changing a data type. Right now the order date column is an object, but let’s change it to a datetime format, (type: orderList['ORDER\_DATE'] = pd.to\_datetime(orderList['ORDER\_DATE']))

Let’s perform these edits to another dataframe in ProductsPerPlant so let’s go ahead and write code for that and update the dictionaries (type):

productsPerPlant = df\_dict['ProductsPerPlant']

productsPerPlant = productsPerPlant.dropna()

df\_dict['OrderList'] = orderList

df\_dict['ProductsPerPlant'] = productsPerPlant

now let’s see the first 5 rows of the new orderlist table to see the added columns. (type: df\_dict['OrderList'].head()) (run the cell) here we see the added columns from the merge as well as the newly created COST column.

(create new cell) (type):

df\_dict['OrderList'].describe() (run the cell). The describe function in python is a quick and powerful way to generate summary statistics for numerical and categorical columns in a pandas dataframe. It generates descriptive statistics for all numeric columns by default. Notice how it doesn’t contain any text values. Statistics include:

count: Number of non-missing values.

mean: Average of the column.

std: Standard deviation.

min: Minimum value.

25%, 50%, 75%: Percentiles (Quartiles).

max: Maximum value.

Let’s create a correlation matrix. (new cell) (type):

correlation\_matrix = df\_dict['OrderList'].corr(numeric\_only=True).round(2)

correlation\_matrix

(run the cell) the .corr method calculates pairwise correlation coefficients for numeric columns in the dataframe. By default, the method used is the pearson correlation, which measures the linear relationship between two variables. I’m specifying numeric\_only = True here because it ensures that only numeric columns in the dataframe are included in the correlation calculation. This avoids errors or unexpected results when the dataframe contains non-numeric data types. Diagonal values will always be 1 because a column is perfectly correlated with itself. Positive values close to 1 indicate a strong positive linear relationship and negative values close to -1 indicate a strong negative linear relationship. Values near 0 indicate a weak linear relationship. Let’s look at unit quantity and cost. They are strongly correlated because the higher the unit quantity, the higher the cost.

Seaborn or the alias we used as sns, is a package that allows us to plot a heatmap. Let’s create a heatmap. (type):

plt.figure(figsize=(10,6))

sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm')

plt.title('Correlation Matrix')

plt.show()

(run the cell)

First we set the figure to a size of 10 inches wide by 6 inches tall. Annot=True displays the correlation values inside the heatmap cells. Cmap=coolwarm sets the color map for the heatmap. Colors range from cool (blue) for negative correlations to warm (red) for positive correlations.

(create new cell) let’s now look to see if we can reveal the unique pricing strategies involved in the dataset. So the pricing strategy is just the mode of transportation and we can find that in the FreightRates table. (type):

pricing\_strategy = df\_dict['FreightRates']['MODE\_DSC'].unique()

The unique function extracts the unique values in the MODE\_DSC column in order to identify the different transportation modes. Another metric in supply chain that we can solve for is the price elasticity and that’s just the mean of the price, which in this case is the rate found in the FreightRates table: (type):

price\_elasticity = df\_dict['FreightRates']['RATE'].mean()

And then we can find the total historical revenue by multiplying the unit quantity times the cost for each row followed by performing a sum function. (type):

historical\_revenue = df\_dict['OrderList']['UNIT\_QUANTITY'] \* df\_dict['OrderList']['COST']

total\_revenue = historical\_revenue.sum()

Now let’s print out the variables we just defined: (type):

print("Pricing strategies:", pricing\_strategy)

print("Average price elasticity:", price\_elasticity)

print("Total historical revenue:", total\_revenue)

(run the cell)

Here we see there are 2 strategies in air and ground and the average price and historical revenue.

Next, let’s demonstrate the use of matplotlib showing a bar graph and a box plot showing the distribution of service levels and daily capacities respectively. (create new cell) (type):

fig, axes = plt.subplots(1, 2, figsize=(15,6))

here, we’re creating a figure with 1 row and 2 columns of subplots. (type):

sns.countplot(data=df\_dict['OrderList'], x='SERVICE\_LEVEL', ax=axes[0])

axes[0].set\_title('Distribution of Service Levels')

axes[0].set\_xlabel('Service Level')

axes[0].set\_ylabel('Count')

“sns.countplot” plots the count of each unique value in the SERVICE\_LEVEL column of the orderlist dataframe as a bar pot. The “axes[0]” directs the plot to the left subplot.

Now we’ll create a boxplot: (type):

sns.boxplot(data=df\_dict['WhCapacities'], y='DAILY\_CAPACITY', ax=axes[1])

axes[1].set\_title('Distribution of Daily Capacities')

axes[1].set\_ylabel('Daily Capacity')

“sns.boxplot” shows the distribution of the daily capacity column in the whcapacities dataframe. (type):

Plt.tight\_layout()

This is a matplotlib function that automatically adjusts the spacing between subplots to prevent overlapping of plot elements, such as axis labels, titles, or tick labels.

(type):

Plt.show()

(run the cell)

So here we see the bar plot showing the count of unique values in service level and we also see the box plot of daily capacities next to it. A boxplot, also known as a box-and-whisker plot, is a statistical visualization that summarizes the distribution of a dataset and highlights key aspects such as central tendency, spread, and potential outliers. It shows the median which is the central line in the box (hover over line). The box itself spans from the first quartile (hover around bottom line) (Q1, 25th percentile) to the third quartile (hover around top line)(Q3, 75th percentile), covering the middle 50% of the data. The height of the box represents the IQR or interquartile range, which measures the spread of the central data. The whiskers extend from the edges of the box to the smallest and largest data points within 1.5 times the IQR. Data beyond this range are considered potential outliers. The relative position of the median within the box and the length of the whiskers indicate symmetry or skewness of the data. For instance, if the median is closer to the bottom of the box, the data might be positive skewed. So in our example (hover over median) it is slightly positively skewed.

(create new cell)

Another visualization that we can show is a scatter plot covering the relationships between weight and shipping cost. We want to show distinction between carriers by using different colors, as well as showing the size of each point to vary based on the different weights. So let’s go ahead and type: (type):

plt.figure(figsize=(10,6))

sns.scatterplot(data=df\_dict['OrderList'], x='WEIGHT', y='COST', hue='CARRIER', size='WEIGHT', sizes=(20, 200), style='CARRIER')

plt.title('Weight vs. Shipping Cost')

plt.xlabel('Weight')

plt.ylabel('Cost')

plt.show()

(run the cell)

As you can see, the standard x and y will show the two variables we’re looking at, but we can add some things to it by adding a hue (hover over it) which changes the colors, size (hover) which sizes the points based on the weight column, showing relative magnitude and we can change the shape using style (hover over style). From an initial glance, it appears to show that weight is proportional to cost.

(create new cell)

Now let’s say you’re tasked with creating a visualization with two bar plots side by side. The left plot should show the number of products manufactured at each plant and the right plot should show manufacturing costs for each plant. So we know we need to look at the productsperplant and whcosts tables (go to dataframe). Let’s create the 2 subplots first: (type):

fig, axes = plt.subplots(1, 2, figsize=(15, 5))

now we want to know how many products each plant manufactures but the table only gives us a product id per plant. but there’s a method that easily allows us to count them all up and assign them to each plant: (type):

plant\_counts = df\_dict['ProductsPerPlant']['PLANT\_CODE'].value\_counts()

the value\_counts method will allow us to count up all the instances that the plant shows up in that column, and assign them to a plant.

now let’s plot the bar graph: (type):

axes[0].bar(plant\_counts.index, plant\_counts.values, color='skyblue')

just for fun I customized the color to be skyblue. .index gives us each x value and .values gives us each corresponding y value. If we want the bar graph to be more clear, let’s add the exact value above each bar. To do that, let’s write: (type):

for i, value in enumerate(plant\_counts.values):

axes[0].text(i, value+1, str(value), ha='center', fontsize=10)

the enumerate function adds a counter to an iterable which in this case is plant\_counts.values so the values can be properly placed at each index. In the .text method, i is the x position, value + 1 is the y position, str(value) is what will be displayed, ha=’center’ aligns the text centered, and fontsize of course gives you the size. We did value+1 just to ensure it’s readable above the bar. Let’s add some more details to the graph: (type):

axes[0].set\_title(‘Number of Products that Each Plant Manufactures’)

axes[0].set\_xlabel(‘Plant Code’)

axes[0].set\_ylabel(‘Number of Products’)

axes[0].tick\_params(axis='x', rotation=45)

the tick\_params will angle the x axis values to ensure the text fits.

Now let’s create the bar graph for manufacturing costs per plant. We just need to get the wh and cost unit columns (type):

axes[1].bar(df\_dict['WhCosts']['WH'], df\_dict['WhCosts']['COST\_UNIT'], color='salmon')

I did another color customize to make it different from skyblue.

Now let’s do the same things we did as above in annotating each value: (type):

for i, value in enumerate(df\_dict['WhCosts']['COST\_UNIT']):

axes[1].text(i, value, f"{value:.2f}", ha='center', fontsize=10)

(hover over f string) this is an f-string format to round the value to 2 decimal points. Now let’s finish out the details: (type):

axes[1].set\_title('Manufacturing Cost for Each Plant')

axes[1].set\_xlabel('Plant Code')

axes[1].set\_ylabel('Cost per Unit')

axes[1].tick\_params(axis='x', rotation=45)

plt.tight\_layout()

plt.show()

(run the cell)

Here we see the number of products per plant and the manufacturing costs for each plant and we can see the use of different colors and the exact value above each bar.

(create new cell)

Now let’s say you’re tasked with creating an interactive visualization to show connections between plants and ports from the plantports dataframe (go to dataframe). From the dataframe we see that a port has one or more plant connections. Let’s start by importing plotly: (type):

import plotly.graph\_objects as go

plotly.graph\_objects is a module in the plotly library that allows you to create highly customizable and interactive visualizations by explicitly defining every component of a figure. Next let’s create a variable to capture the plantports dataframe: (type):

df\_plant\_ports = df\_dict['PlantPorts']

we need to initialize a figure object when working with plotly so let’s do that: (type):

fig = go.Figure()

now let’s iterate through each row and create a scatter trace: (type):

for \_, row in df\_plant\_ports.iterrows():

fig.add\_trace(

go.Scatter(

x = [row['PLANT\_CODE'], row['PORT']],

y = [1, 0 + np.random.uniform(-0.1, 0.1)], # x and y are lists and so plant code will be at position 1, while port is around position 0 but we add np.random.uniform(-0.1,0.1) to add a small random variation to slightly offset points for better visualization. It’ll become more clear when we run the function.

mode = 'lines+markers', # draws lines connecting the points and places markers on them.

marker = dict(

size = 10, # size of the point

symbol = 'circle', # shape of the point

line = dict(

color = 'blue', # color of the point

width = 2

)

),

hoverinfo = 'text',

text = f"Plant: {row['PLANT\_CODE']}<br>Port: {row['PORT']}", # if you hover over each point, you’ll see this

)

)

Now let’s update the layout: (type):

fig.update\_layout(

title\_text = 'Plant and Port Connections',

showlegend = False,

xaxis = dict(

title = 'PLANT\_CODE - PORT',

tickangle = -45

),

yaxis = dict(

title = '', # two single quotes

showticklabels = False,

range = [-.2, 1.2]

), # right here, end it with fig.show() and do the update menus later. Run it and then modify code

updatemenus=[ # this functionality adds interactive menus to the chart for dynamic filtering. And we’ll add the different filters here soon.

dict(

buttons=[

dict(

label=’Show All’, # this will show the label where all trace are visible

method=’update’,

args=[{"visible": [True] \* len(df\_plant\_ports)}, # the first argument in args modifies trace visibility.

{"title": "All Plant-Port Connections"}]

), # after this, go down and finish it and show prior to putting the other ports in

dict(

label="Filter by Port 01",

method="update",

args=[{"visible": [row['PORT'] == 'PORT01' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 01"}]

),

# copy and paste for all ports to port 11

dict(

label="Filter by Port 02",

method="update",

args=[{"visible": [row['PORT'] == 'PORT02' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 02"}]

),

dict(

label="Filter by Port 03",

method="update",

args=[{"visible": [row['PORT'] == 'PORT03' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 03"}]

),

dict(

label="Filter by Port 04",

method="update",

args=[{"visible": [row['PORT'] == 'PORT04' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 04"}]

),

dict(

label="Filter by Port 05",

method="update",

args=[{"visible": [row['PORT'] == 'PORT05' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 05"}]

),

dict(

label="Filter by Port 06",

method="update",

args=[{"visible": [row['PORT'] == 'PORT06' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 06"}]

),

dict(

label="Filter by Port 07",

method="update",

args=[{"visible": [row['PORT'] == 'PORT07' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 07"}]

),

dict(

label="Filter by Port 08",

method="update",

args=[{"visible": [row['PORT'] == 'PORT08' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 08"}]

),

dict(

label="Filter by Port 09",

method="update",

args=[{"visible": [row['PORT'] == 'PORT09' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 09"}]

),

dict(

label="Filter by Port 10",

method="update",

args=[{"visible": [row['PORT'] == 'PORT10' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 10"}]

),

dict(

label="Filter by Port 11",

method="update",

args=[{"visible": [row['PORT'] == 'PORT11' for \_, row in df\_plant\_ports.iterrows()]},

{"title": "Connections for Port 11"}]

)

],

type = 'dropdown', # this will give us a drop down button to make dynamic filters

showactive = True, # this highlights the currently selected button in an interactive menu.

direction = 'down'

)

]

)

fig.show()

(run the cell)

Here we see all the connections, but if we play around with it, we can see the different ports and which plants they’re connected to.

That’s it for this video, I hope you learned something new and thanks so much for watching.