Supply Chain Management System for Optimal Inventory and Logistics

InvOptima AI

Author: Utkarsh Gupta

Date: 22-06-2024

Table of contents

Problem Statement

The complexities in the modern supply chains have intensified over recent years due to globalization, increasing customer demands, and the arrival of e-commerce. The traditional supply chain management systems are often overwhelmed by the vast amount of data generated daily. They lack the capacity for real-time processing and fail to provide predictive insights, leading to several critical inefficiencies:

1.1 Demand Forecasting

Traditional methods rely on historical data and simple statistical methods, which do not account for real-time changes in market conditions, leading to inaccurate demand forecasts. This inaccuracy can result in overstocking or stockouts, both of which are costly for businesses.

1.2 Inventory Management

Without accurate demand forecasting, businesses struggle to maintain optimal inventory levels. Overstocking ties up capital and increases holding costs, while stockouts lead to missed sales opportunities and dissatisfied customers.

1.3 Logistics and Transportation

Inefficiencies in routing and scheduling can significantly increase transportation costs and delivery times. Traditional systems are often unable to adapt quickly to disruptions, such as weather conditions, traffic, or geopolitical issues, leading to delays and increased costs.

1.4 Operational Costs

Manual processes and lack of automation result in high labor costs and increased chances of human error. Additionally, the inability to process and analyze data in real-time means that businesses cannot respond swiftly to changing conditions.

1.5 Customer Satisfaction

Delays, stockouts, and poor product availability directly impact customer satisfaction. In a competitive market, businesses that fail to meet customer expectations risk losing their market share to more agile competitors. The primary problem is the need for an intelligent, real-time product that can leverage AI and machine learning technologies to optimize supply chain operations, addressing these inefficiencies comprehensively.

Business Need Assessment

Supply chain optimization is critical across various industries, including retail, manufacturing and logistics. Here are the specific needs and pain points in these sectors:

2.1.1 Retail

Retailers face fluctuating demand patterns and seasonal variations, making accurate demand forecasting essential. Overstocking leads to increased holding costs and markdowns, while stockouts result in lost sales and dissatisfied customers.

2.1.2 Manufacturing

Manufacturers need to balance the supply of raw materials with production schedules to avoid downtime and ensure timely delivery of finished products. Inefficient supply chain management can lead to production delays, increased operational costs, and missed delivery deadlines.

2.1.3 Logistics

Logistics providers need to optimize routes and schedules to minimize transportation costs and delivery times. Real-time visibility and adaptive planning are crucial to handling disruptions and ensuring timely deliveries.
By addressing the following needs, an AI-driven supply chain management system can provide significant competitive advantages, reduce costs, and enhance customer satisfaction.

2.2.1 Efficiency

Businesses need to streamline their supply chain operations to reduce waste and improve productivity. AI can help automate repetitive tasks, optimize processes, and provide real-time insights for better decision-making.

2.2.2 Cost Reduction

Reducing operational costs is a primary goal. AI can be used to optimize inventory levels, reduce holding costs, improve demand forecasting accuracy, and minimize transportation expenses.

2.2.3 Scalability

As businesses grow, their supply chain operations become more complex. AI-driven solutions offer scalability, allowing businesses to manage increasing volumes and complexity efficiently.

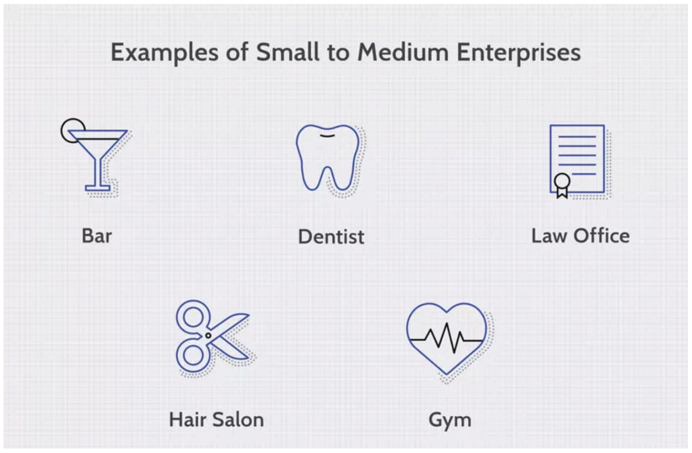
Target Specifications and Characterization

3.1 Target Customers

3.1.1 Mid-sized Enterprises

These businesses have complex supply chain operations involving multiple stakeholders, international logistics, and high transaction volumes. They require advanced solutions to manage and optimize their supply chains effectively. They also have not been exposed to AI technologies that can help them expand their business.

3.1.2 Industries



3.1.2.1 Retail

Including e-commerce giants and brick-and-mortar stores with extensive supply chains.

3.1.2.2 Manufacturing

Spanning various sectors such as automotive, electronics, and consumer goods.

3.2 Target Specifications

3.2.1 Scalability

The solution should be scalable to accommodate growing business needs and increasing data volumes without compromising performance.

3.2.2 Real-time Processing

Ability to process and analyze huge amounts of data in real-time, providing immediate insights and recommendations.

3.2.3 User-friendly Interface

An intuitive and easy-to-use interface that enables users to access insights, generate reports, and make decisions quickly.

External Search

To develop this report, extensive research was done using various online sources, research papers, github repositories including:

4.1 Case Studies and Reports

McKinsey & Company:
https://www.mckinsey.com/industries/metals-and-mining/our-insights/succeeding-in-the-ai-supply-chain-revolution
Hitachi: https://www.hitachi.com/rev/archive/2018/r2018_02/12b03/index.html
https://www.netsuite.com/portal/resource/articles/inventory-management/abc-inventory-analysis.shtml

4.2 Papers and Articles

https://towardsdatascience.com/machine-learning-for-store-demand-forecasting-and-inventory-optimization-part-1-xgboost-vs-9952d8303b48
https://throughput.world/blog/supply-chain-inventory-optimization/
Stochastic Artificial Intelligence benefits andSupply Chain Management: inventory prediction
Applications of deep learning into supply chain management: a systematic literature review and a framework for future research

4.3 Github Repositories

https://github.com/samirsaci/procurement-management

4.4 Miscellaneous

https://www.oracle.com/scm/
https://www.ibm.com/products/supply-chain-suite
https://acropolium.com/blog/employing-supply-chain-analytics-software-for-efficient-workflows-key-features-use-cases/
https://www.investopedia.com/terms/s/smallandmidsizeenterprises.asp
https://builtin.com/artificial-intelligence/ai-in-supply-chain

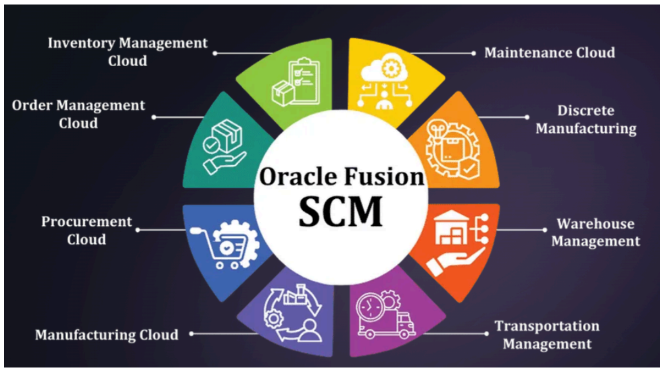
4.5 Kaggle

https://www.kaggle.com/competitions/demand-forecasting-kernels-only/data
https://www.kaggle.com/competitions/nyc-taxi-trip-duration/data

Benchmarking Alternate Products

To position our product effectively in the market, it is crucial to benchmark it against a few existing products and services. Here are detailed comparisons with the other alternatives:

5.1 Oracle SCM Cloud



5.1.1 Strengths

1. Robust functionalities.
2. Strong analytics and AI capabilities.
3. Comprehensive cloud-based solution.

5.1.2 Weaknesses

1. High cost of implementation and ongoing subscription fees.
2. Lengthy and complex integration process.
3. Requires significant customization.

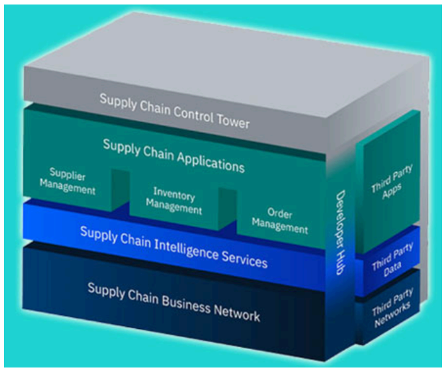
5.2 IBM Sterling Supply Chain Suite

5.2.1 Strengths

1. Strong AI and blockchain capabilities.
2. Real-time visibility and analytics.
3. Modular approach allows for tailored solutions.

5.2.2 Weaknesses

1. Complex integration and deployment process.
2. High cost of implementation and maintenance.
3. Requires significant customization.



Many other companies like Mckinsey, Gartner have launched various similar solutions as well. Most of them can't be used by small community shop chains, local food chains because they don't have people to possess the knowledge and expertise to operate and handle these solutions. Also they don't have enough capital to buy subscriptions, in other words their ROI would be very less.

Our Product Differentiation

The AI-Driven Supply Chain Management System differentiates itself from these competitors by offering:

User-Friendly Interface:
Designed for ease of use, with an intuitive dashboard and graphs that provides real-time insights and recommendations.

Cost-Effective Solution:
More affordable implementation and subscription costs compared to competitors.

Scalability:
Scalable solution that grows with the business, handling increasing data volumes and complexity.

Integration of various methodologies in one system:
From start till the end of the process there are algorithms and processes to help and make cost effective decisions.

advanced inventory management
smart demand forecasting
intelligent route optimization
Supply Chain Analytics
streamlined efficiency
better procurement & logistics

Applicable Regulations

Our product must comply with various regulations to ensure legality, safety, and sustainability. Key regulations include data privacy, trade regulations, environmental restrictions.

Applicable Constraints

Developing and deploying the product involves several constraints:

8.1 Space

Cloud-based Deployment: Utilizing cloud infrastructure minimizes physical space requirements. Cloud providers such as AWS, Azure, and Google Cloud offer scalable solutions that can handle large data volumes without the need for extensive on-premises hardware.

8.2 Budget

8.2.1 Initial Development Costs

Costs include salaries for the development team, cloud infrastructure, software licenses etc

8.2.2 Ongoing Costs

Data collection, maintenance, updates, cloud services, and customer support.

8.2.3 Cost Management

Implementing cost-effective solutions and optimizing resource utilization to manage budget constraints effectively.

8.3 Expertise

8.3.1 Required Expertise

The project requires a team of data scientists, AI specialists, software engineers, and supply chain experts. Finding and retaining skilled professionals can be challenging and costly.

8.3.2 Training and Development

Continuous training and development of the team to keep up with the latest technologies and industry trends.

Business Model

Monetization strategy should include several revenue streams to ensure sustainable growth and profitability:

9.1 Subscription-based Model

9.1.1 Monthly/Annual Subscription Fees

Businesses pay a subscription fee based on their size and the features they use. Different tiers offer varying levels of access, from basic to advanced features.

9.1.2 Tiered Pricing

Offering multiple pricing tiers to cater to different business sizes and needs. For example, small businesses may opt for a basic plan, while large enterprises may choose a premium plan with advanced features.

9.2 Freemium Model

9.2.1 Basic Features for Free

Providing essential features like basic analysis which would give a very minor analysis of the everyday sales and its trend. at no cost to attract users and allow them to experience the product's value.

9.2.2 Premium Features for a Fee

Charging for advanced features such as predictive analytics, real-time optimization, and custom reports. This encourages users to upgrade to paid plans for enhanced capabilities.

9.3 Consulting Services

9.3.1 Implementation Consulting

Offering consulting services to assist businesses with the implementation and integration of the AI-driven supply chain management system.

9.3.2 Optimization Services

Providing ongoing optimization services to help businesses fine-tune their supply chain operations and maximize efficiency.

Concept Development

10.1 Demand Forecasting

Utilizing machine learning algorithms to predict demand accurately based on real-time data and market trends.

10.2 Procurement

design an optimal inventory replenishment strategy for a mid-size retail store considering

10.3 Product Segmentation

Performing ABC analysis on sales data and demand variability. Combining the two gives insights into which SKUs and products are performing better than others.

10.4 Inventory Optimization

Calculating safety stock, reorder point, EOQ and performing pareto analysis.

10.5 Route Planning

Optimizing transportation routes and schedules to reduce costs and improve delivery times.

All methodologies converge to one aim to reduce costs, improve efficiency and increase customer satisfaction.

Final Product Prototype with Schematic Diagram

11.1 Abstract

The final product prototype is an AI-Driven Supply Chain Management System designed to optimize supply chain operations through real-time data processing and predictive analytics. The intuitive user interface offers a comprehensive dashboard for real-time monitoring and decision-making. The aim is to help the user from start to end in their supply chain.

11.2 Schematic Diagram

The schematic diagram below illustrates the architecture and key components of the AI-Driven Supply Chain Management System:

```
[Data Collection]
   |
   |----> [Converting to appropriate form]
            |
            |----> [Data Processing and Storage]
                     |
                     |----> [AI and Machine Learning Engine]
                              |
                              |----> [Demand Forecasting]
                              |
                              |----> [Procurement]
                              |
                              |----> [Product Segmentation]
                              |
                              |----> [Inventory Optimization]
                              |
                              |----> [Logistics Planning]
                     |
                     |----> [Analytics and Insights]
                              |
                              |----> [User Interface/Dashboard]
                              |
                              |----> [Graphs and Trends]
```

Code Implementation/Validation on Small Scale

Explanation and graphs have been provided for each methodology. Final product will we integrating all of them into one app/website such that the user can use all the features from one site.

Demand Forecasting

This section is regarding the prediction for sales in the future based on current data. It will be tuned real time as and when real time data is available after the end of every working day. The user just needs to input the csv file of the data recorded.
The dataset should have the following columns in it:

1. The date of the sale
2. The store number of the sale
3. The item number(This can mean the item has been sold from the range of products)
4. The number of products sold

The rest of the work is done by the code itself.

The code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import xgboost as xgb
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from matplotlib.patches import Rectangle
```

```
# Import training and test data

train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
```

Data preprocessing

```
def date_features(df):
    # Date Features
    df['date'] = pd.to_datetime(df['date'])
    df['year'] = df.date.dt.year
    df['month'] = df.date.dt.month
    df['day'] = df.date.dt.day
    df['dayofyear'] = df.date.dt.dayofyear
    df['dayofweek'] = df.date.dt.dayofweek
    df['weekofyear'] = df.date.dt.isocalendar().week

    # Additional Data Features
    df['day^year'] = np.log((np.log(df['dayofyear'] + 1)) ** (df['year'] - 2000))

    # Drop date
    df.drop('date', axis=1, inplace=True)

    return df
# Dates Features for Train, Test
train, test = date_features(train), date_features(test)
```

```
# Daily Average, Monthly Average for train
train['daily_avg']  = train.groupby(['item','store','dayofweek'])['sales'].transform('mean')
train['monthly_avg'] = train.groupby(['item','store','month'])['sales'].transform('mean')
train = train.dropna()

# Average sales for Day_of_week = d per Item,Store
daymonth_avg = train.groupby(['item','store','dayofweek'])['sales'].mean().reset_index()
# Average sales for Month = m per Item,Store
monthly_avg = train.groupby(['item','store','month'])['sales'].mean().reset_index()
```

```
# Merge Test with Daily Avg, Monthly Avg
def merge(df1, df2, col,col_name):

df1 =  pd.merge(df1,df2,how='left',on=None,left_on=col,
right_on=col,left_index=False, right_index=False, sort=True,copy=True, indicator=False)
```

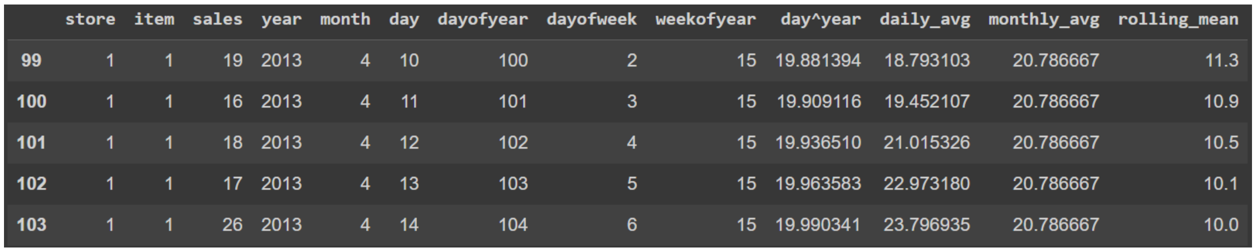
```
  df1 = df1.rename(columns={'sales':col_name})
  return df1

# Add Daily_avg and Monthly_avg features to test
test = merge(test, daymonth_avg,['item','store','dayofweek'],'daily_avg')
test = merge(test, monthly_avg,['item','store','month'],'monthly_avg')
```

```
# Sales Rolling mean sequence per item
rolling_10= train.groupby(['item'])['sales'].rolling(10).mean().reset_index().drop('level_1', axis=1)
train['rolling_mean'] = rolling_10['sales']
train = train.dropna()
a = train['rolling_mean']
a.isna().sum()
```

```
# 90 last days of training rolling mean sequence added to test data
rolling_last90 = train.groupby(['item','store'])['rolling_mean'].tail(90).copy()
test['rolling_mean'] = rolling_last90.reset_index().drop('index', axis=1)
a = test['rolling_mean']
a.isna().sum()
```

```
# Shifting rolling mean 3 months
train['rolling_mean'] = train.groupby(['item'])['rolling_mean'].shift(90) # Create a feature with rolling mean of
day - 90
train.head()
train = train.dropna()
a = train['rolling_mean']
a.isna().sum()
```

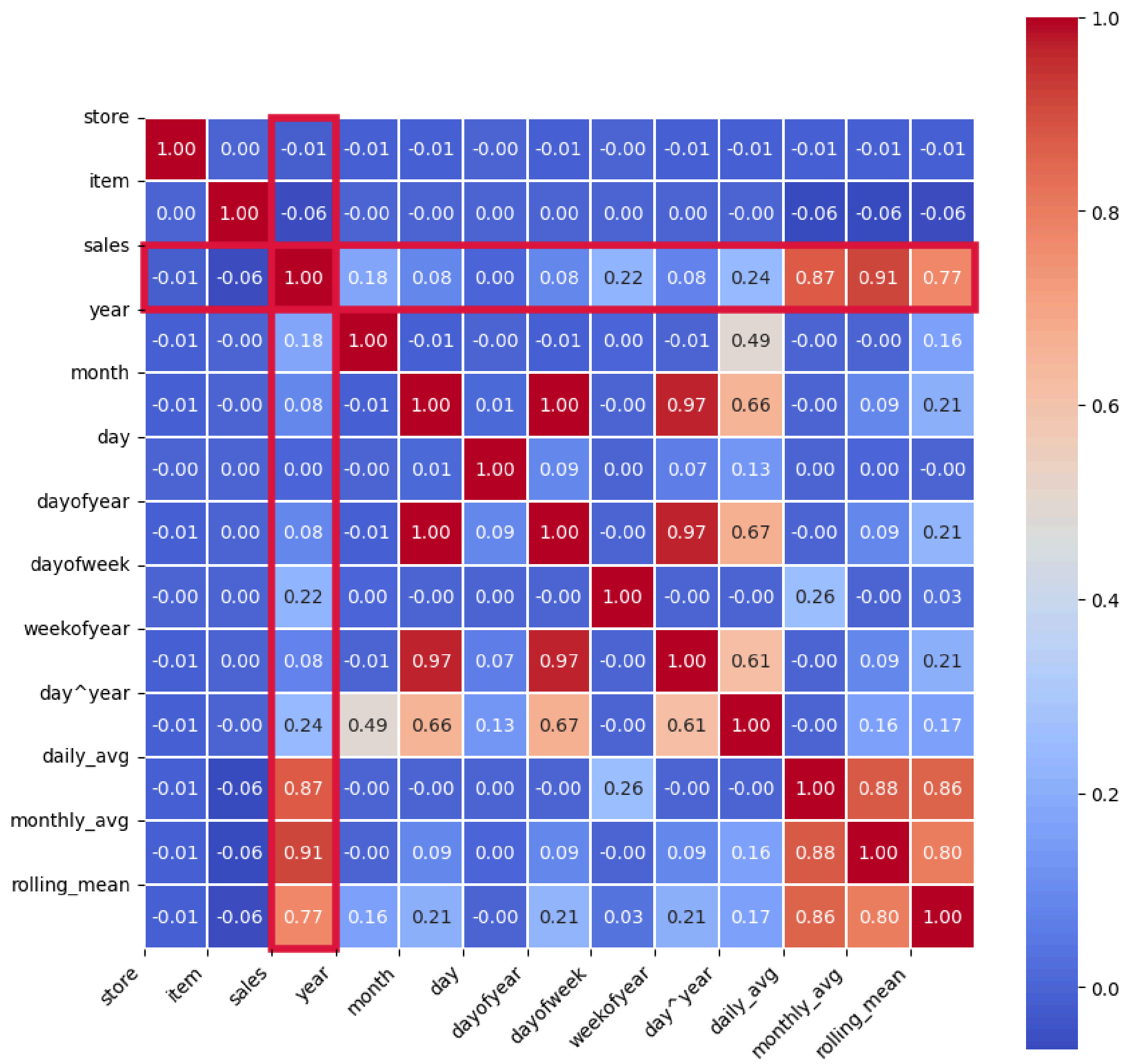
```
train.head()
```



	store	item	sales	year	month	day	dayofyear	dayofweek	weekofyear	day^year	daily_avg	monthly_avg	rolling_mean
99	1	1	19	2013	4	10	100	2	15	19.881394	18.793103	20.786667	11.3
100	1	1	16	2013	4	11	101	3	15	19.909116	19.452107	20.786667	10.9
101	1	1	18	2013	4	12	102	4	15	19.936510	21.015326	20.786667	10.5
102	1	1	17	2013	4	13	103	5	15	19.963583	22.973180	20.786667	10.1
103	1	1	26	2013	4	14	104	6	15	19.990341	23.796935	20.786667	10.0

The Correlation Matrix

```
# Calculate the correlation matrix excluding non-numeric columns
matrix = train.select_dtypes(include=[np.number]).corr()

#extracting variable names
variables = []
for i in matrix.columns:
  variables.append(i)
N = len(variables)

fig, ax = plt.subplots(figsize=(10, 10))
ax = sns.heatmap(matrix, cmap='coolwarm', annot=True, linewidths=.2, xticklabels=variables,
yticklabels=variables,fmt='.2f')

wanted_label = 'sales'
wanted_index = variables.index(wanted_label)
x, y, w, h = 0, wanted_index, N, 1
for _ in range(2):
    ax.add_patch(Rectangle((x, y), w, h, fill=False, edgecolor='crimson', lw=4, clip_on=False))
    x, y = y, x # exchange the roles of x and y
    w, h = h, w # exchange the roles of w and h


# Adding labels to the matrix
plt.xticks(range(len(matrix)), variables, rotation=45, ha='right')
plt.yticks(range(len(matrix)), variables)
ax.set_aspect("equal")
# Display the plot
plt.show()
```



	store	item	sales	year	month	day	dayofyear	dayofweek	weekofyear	day^year	daily_avg	monthly_avg	rolling_mean
store	1.00	0.00	-0.01	-0.01	-0.01	-0.00	-0.01	-0.00	-0.01	-0.01	-0.01	-0.01	-0.01
item	0.00	1.00	-0.06	-0.00	-0.00	0.00	0.00	0.00	0.00	-0.00	-0.06	-0.06	-0.06
sales	-0.01	-0.06	1.00	0.18	0.08	0.00	0.08	0.22	0.08	0.24	0.87	0.91	0.77
year	-0.01	-0.00	0.18	1.00	-0.01	-0.00	-0.01	0.00	-0.01	0.49	-0.00	-0.00	0.16
month	-0.01	-0.00	0.08	-0.01	1.00	0.01	1.00	-0.00	0.97	0.66	-0.00	0.09	0.21
day	-0.00	0.00	0.00	-0.00	0.01	1.00	0.09	0.00	0.07	0.13	0.00	0.00	-0.00
dayofyear	-0.01	0.00	0.08	-0.01	1.00	0.09	1.00	-0.00	0.97	0.67	-0.00	0.09	0.21
dayofweek	-0.00	0.00	0.22	0.00	-0.00	0.00	-0.00	1.00	-0.00	-0.00	0.26	-0.00	0.03
weekofyear	-0.01	0.00	0.08	-0.01	0.97	0.07	0.97	-0.00	1.00	0.61	-0.00	0.09	0.21
day^year	-0.01	-0.00	0.24	0.49	0.66	0.13	0.67	-0.00	0.61	1.00	-0.00	0.16	0.17
daily_avg	-0.01	-0.06	0.87	-0.00	-0.00	0.00	-0.00	0.26	-0.00	-0.00	1.00	0.88	0.86
monthly_avg	-0.01	-0.06	0.91	-0.00	0.09	0.00	0.09	-0.00	0.09	0.16	0.88	1.00	0.80
rolling_mean	-0.01	-0.06	0.77	0.16	0.21	-0.00	0.21	0.03	0.21	0.17	0.86	0.80	1.00

```
# removing low correlation columns
for df in [train, test]:
  df.drop(['dayofyear', 'weekofyear', 'daily_avg', 'day', 'month', 'item', 'store',], axis=1, inplace=True)
```

```
train.std()
train.mean()
test.std()
test.mean()
```

After this we make a model based on the columns which have impact on the sales value.

Modellig and forecasting

```
X_train = train.drop('sales', axis=1).dropna()
y_train = train['sales']
# Test Data
test.sort_values(by=['id'], inplace=True)
X_test = test.drop('id', axis=1)
```

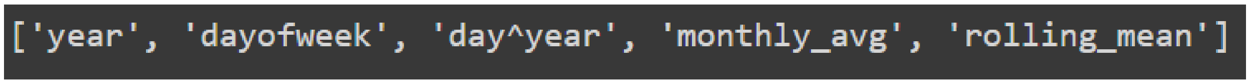
```
# prompt: train a model using for the train dataset to predict sales
df_train = train.copy()

X_train , X_test ,y_train, y_test = train_test_split(df_train.drop('sales',axis=1),df_train.pop('sales'), random_state=123, test_size=0.2)

# XGB Model
matrix_train = xgb.DMatrix(X_train, label = y_train)
matrix_test = xgb.DMatrix(X_test, label = y_test)
# Run XGB
params = {'objective': 'reg:linear', 'eval_metric': 'mae', 'eta': 0.3, 'gamma': 0.1, 'max_depth': 6, 'min_child_weight': 1, 'subsample': 0.8}

model = xgb.train(params, dtrain=matrix_train, num_boost_round=500, early_stopping_rounds=100, evals=[(matrix_test, 'test')])
# Convert X_test to DMatrix before prediction
test_matrix = xgb.DMatrix(X_test)  # Create DMatrix for X_test
y_pred = model.predict(test_matrix)  # Use DMatrix for prediction

print("Mean Squared Error (MSE):", mean_squared_error(y_test, y_pred))
print("Mean Absolute Error (MAE):", mean_absolute_error(y_test, y_pred))
print(Root Mean Squared Error (RMSE):", np.sqrt(mean_squared_error(y_test, y_pred)))
print("R-squared:", r2_score(y_test, y_pred))
```

Result of the Final few iterations:

```
[138]	test-mae:5.59839
[139]	test-mae:5.59837
Mean Squared Error (MSE): 52.93777843285481
Mean Absolute Error (MAE): 5.598398102446344
Root Mean Squared Error (RMSE): 7.27583523953469
R-squared: 0.9358558807152947
```

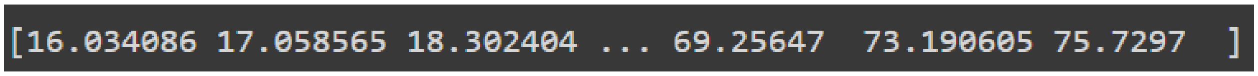
R-squared of **0.936** is a very good indication for first round of modelling. The model can be fine tuned further if required by increasing epochs and adding early stoping.

```
#Features of the above model
X = train.drop('sales', axis=1)
features = X.columns.tolist()
print(features)
```



```
['year', 'dayofweek', 'day^year', 'monthly_avg', 'rolling_mean']
```

```
import xgboost as xgb

# Use the original DataFrame X, not the DMatrix
test_matrix = xgb.DMatrix(train.drop('sales', axis=1))  # Create DMatrix for X_test
y_pred = model.predict(test_matrix)  # Use DMatrix for prediction
print(y_pred)
```



```
[16.034086 17.058565 18.302404 ... 69.25647  73.190605 75.7297   ]
```

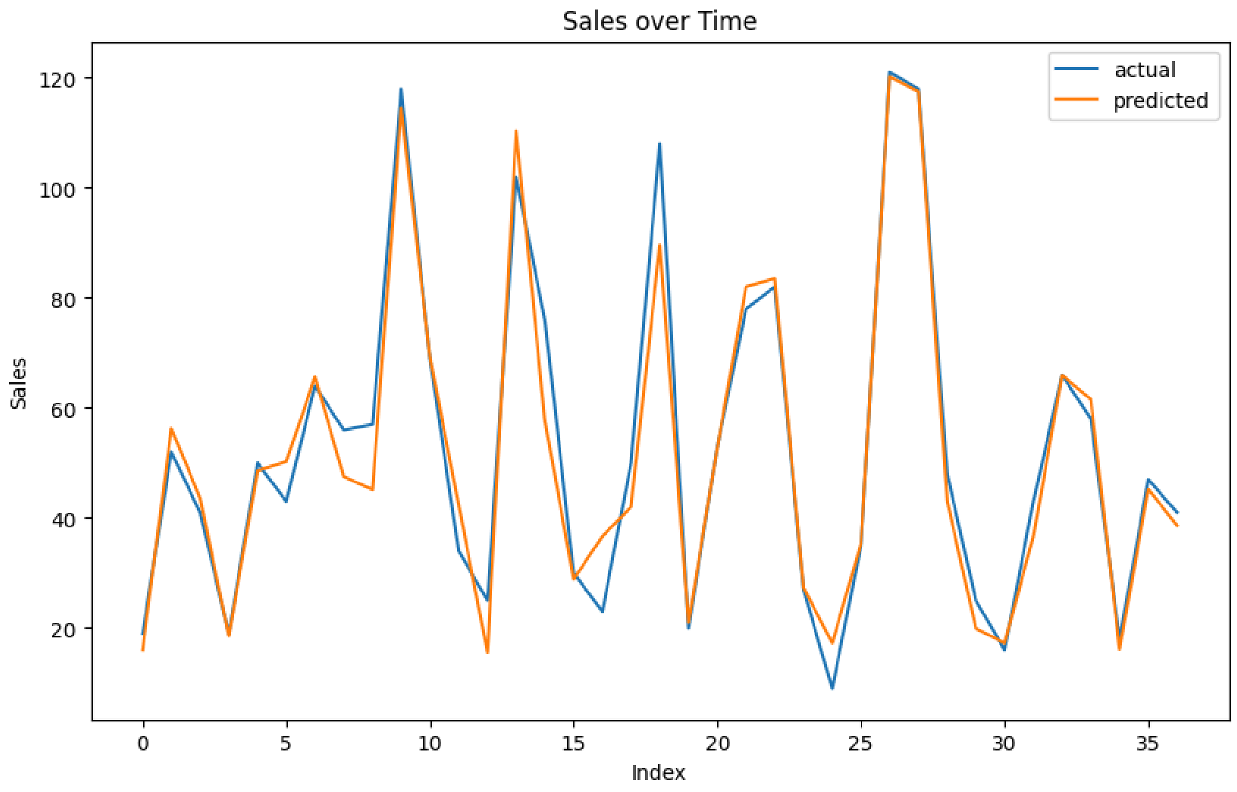
```
arr = train['sales'].to_numpy()
np.shape(train['sales'])
```

```
(        908050,                                                    )
```

We have **90850** data points to look from.

Now we plot and check whether how well our model is following the trend shown by the actual salses data and if we can trust it or not

```
# prompt: plot a graph of sales

plt.figure(figsize=(10, 6))
plt.plot(arr[0::25000], label = 'actual')
plt.plot(y_pred[0::25000], label = 'predicted')
plt.xlabel('Index     ')
plt.ylabel('Sales')
plt.title('Sales over Time')
plt.legend()
plt.show()
```



A very good fit for a few random points taken out of the whole collection. This visually proves that this model can be used for forecasting purposes.

Procurement

(This code has been taken from a source, link has been added in reference) (I tried it on my own but the scipy library is not able to solve the equations hence had to use an external source)

Now that we know what our sales are, we can calculate how much product we require to buy. We have **2** files. One is the demand of each SKU and the other is the cost per carton for delivery. Using this we find the optimized cost for replenishment of all the SKUs.

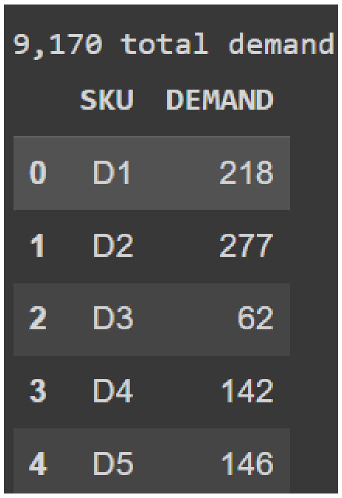
We will present a simple methodology using Non-Linear Programming to design an optimal procurement strategy for a mid-size retail store considering:

1. Transportation Costs from the Supplier Warehouse to the Store Reserve ($/Carton)
2. Costs to finance your inventory (% of inventory value in $)
3. Reserve (Store's Warehouse) Rental Costs for storage ($/Carton)

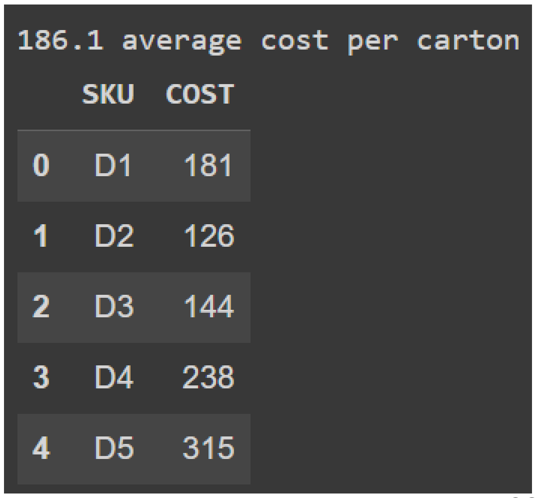
The code

```
import pandas as pd
from pulp import *
import numpy as np
from scipy.optimize import minimize
import math
import time
from datetime import datetime
```

```
# Demand
df_demand = pd.read_csv('df_demandsku.csv', index_col=0)
print("{:,} total demand".format(df_demand.DEMAND.sum()))
df_demand.head()
```



```
9,170 total demand
   SKU  DEMAND
0  D1      218
1  D2      277
2  D3       62
3  D4      142
4  D5      146
```



```
186.1 average cost per carton
   SKU  COST
0  D1   181
1  D2   126
2  D3   144
3  D4   238
4  D5   315
```

```
df_costsku = pd.read_csv('df_costsku.csv', index_col=0)
print("{:,} average cost per carton".format(df_costsku.COST.mean()))
df_costsku.head()
```

```
A = -0.3975
b = 42.250
```

These were the transportation cost variables which can be changed as per requirements.

```
def objective(R):
    result = 0
    for i in range(60):
        # TR Costs
        result += (A*(df_demand.loc[i,'DEMAND']/R[i]) + b) * R[i]
        # Capital Costs
        result += (df_demand.loc[i,'DEMAND']/(2*R[i])) * df_costsku.loc[i,'COST']*0.125
        # Storage Costs
        result += (df_demand.loc[i,'DEMAND']/(2*R[i])) * 12 * 480/2000
    return result
```

```
# Initialize constraints list
cons = []
# Maximum Inventory
def constraint1(R):
    loop = 0
    for i in range(60):
        loop += R[i]
    result = 480 - loop
    return result
cons.append({'type':'ineq','fun':constraint1})
```

```
# Add Order Size Constraints
for i in range(60):
    # Minimum Order Quantity
    c2 = lambda R : (df_demand.loc[i,'DEMAND']/R[i]) - 1
    cons.append({'type':'ineq','fun':c2})
    # Maximum Order Quantity
    c3 = lambda R : 400 - (df_demand.loc[i,'DEMAND']/R[i])
    cons.append({'type':'ineq','fun':c3})
```

```
# All SKU replenished 1 time
```

```
R0 = [2 for i in range(60)]
print("${:,} total cost for initial guessing".format(objective(R0).round(1)))
```

```
$63,206.7 total cost for initial guessing
```

```
# Bound vector
b_vector = (1, 365)
bnds = tuple([b_vector for i in range(60)])
```

```
start = time.time()
sol = minimize(objective, R0, method = 'SLSQP', bounds=bnds, constraints = cons, options={'maxiter': 100})
exec_time = (time.time()-start)
print("Execution time is {}s for 100 iterations".format(exec_time))
```

```
# Initial solution
sol_init = sol.x
# Take the floor of the solution to have an integer as number of replenishment and never exceed stock limit
sol_final = [math.floor(i) for i in sol_init]
```

```
print(("'For {} Iterations
-> Initial Solution: ${:,}
-> Integer Solution: ${:,}
'").format(100, sol.fun.round(1), objective(sol_final).round(1)))
```

```
For 100 Iterations
-> Initial Solution: $28,991.9
-> Integer Solution: $29,221.3
```

```
print("Maximum inventory level with continuous number of replenishment: {}".format(sum(sol_init)))
print("Maximum inventory level with continuous number of replenishment: {}".format(sum(sol_final)))
```

```
Maximum inventory level with continuous number of replenishment: 386.2361790436466
Maximum inventory level with continuous number of replenishment: 356
```

```
start = time.time()
sol = minimize(objective, R0, method = 'SLSQP', bounds=bnds, constraints = cons, options={'maxiter': 500})
exec_time = (time.time()-start)
print("Execution time is {}s for 500 iterations".format(exec_time))
```

```
# Initial solution
sol_init2 = sol.x
# Take the ceiling of the solution to have an integer as number of replenishment
sol_final2 = [math.ceil(i) for i in sol_init2]
```

```
print(("'For {} Iterations
-> Initial Solution: ${:,}
-> Integer Solution: ${:,}
'").format(100, sol.fun.round(1), objective(sol_final2).round(1)))
```

```
For 100 Iterations
-> Initial Solution: $28,991.9
-> Integer Solution: $29,126.5
```

Finally, we can test several scenarios to see how the model reacts:

1. High rental costs and low transportation costs
2. Non-linear purchasing costs
3. Higher Minimum Order Quantity

Product Segmentation

This is one of the most important analysis because after finding out the demand we cannot just place the order for the items. We need to check how each and every store is performing individually. Marketing measures can be taken to improve the sales in that particular store.

ABC Analysis

Concept

ABC analysis is a technique used in inventory management to categorize items based on their contribution to overall revenue or importance to the business. The categorization helps prioritize resources and efforts, focusing more attention on high-value items while optimizing inventory management for lower-value items.

Graphs and Analysis

1. ABC Categorization:
The first graph in ABC analysis categorizes SKUs into three categories:
A, B, and C. Category A typically includes a small percentage of SKUs (e.g., top 5%) that contribute the most to total turnover. Category B includes moderate-value SKUs, and Category C includes a large percentage of SKUs with the lowest turnover contribution.
This categorization helps identify critical items that require close monitoring and investment versus less critical items where inventory costs can be optimized.

2. Contribution of Top SKUs:
The second graph illustrates the cumulative contribution of the top SKUs (e.g., top 20%) to the total turnover. It shows the Pareto principle in action, where a significant portion of the turnover comes from a relatively small number of SKUs.
This insight guides decision-making in resource allocation, inventory stocking levels, and sales strategies.

3. Distribution of Turnover by ABC Category:
Another graph shows how the total turnover is distributed across different ABC categories (A, B, C). It highlights the percentage of turnover attributed to each category, providing a clear view of where the bulk of revenue comes from and which categories contribute less.

The Code

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
import seaborn as sns
```

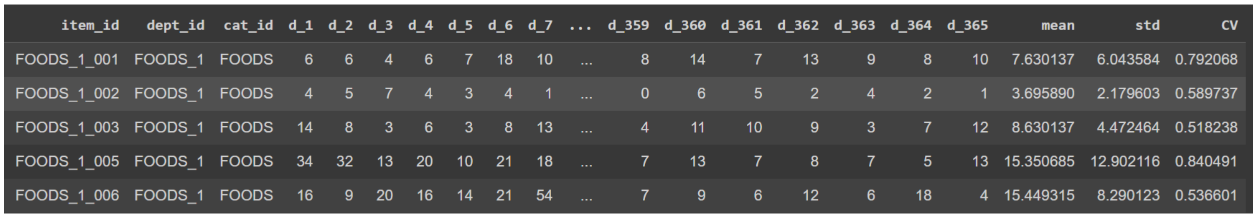
```python
import pandas as pd

# Load datasets
sales = pd.read_csv('sales_train_evaluation.csv')
prices = pd.read_csv('sell_prices.csv')
calendar = pd.read_csv('calendar.csv')

# Assuming COLS_DATE is a list of columns representing daily sales data
COLS_DATE = [f'd_{i}' for i in range(1, 366)]

# Filter sales data for 'XYZ' category
df_filtered = sales[sales['cat_id'] == 'FOODS'].copy()
```

(There were 3 available sets: Households, Hobbies and Foods)

```python
# Group by 'item_id', 'dept_id', and 'cat_id' and sum the daily sales
df_grouped=df_filtered.groupby(['item_id','dept_id','cat_id'])[COLS_DATE].sum().reset_index()
```

```python
# Calculate mean and standard deviation
df_grouped['mean'] = df_grouped[COLS_DATE].mean(axis=1)
df_grouped['std'] = df_grouped[COLS_DATE].std(axis=1)

# Filter out items with zero mean (no sales)
df_grouped = df_grouped[df_grouped['mean'] > 0]
```

```python
df_grouped.head()
```



item_id	dept_id	cat_id	d_1	d_2	d_3	d_4	d_5	d_6	d_7	...	d_359	d_360	d_361	d_362	d_363	d_364	d_365	mean	std	CV
FOODS_1_001	FOODS_1	FOODS	6	6	4	6	7	18	10	...	8	14	7	13	9	8	10	7.630137	6.043584	0.792068
FOODS_1_002	FOODS_1	FOODS	4	5	7	4	3	4	1	...	0	6	5	2	4	2	1	3.695890	2.179603	0.589737
FOODS_1_003	FOODS_1	FOODS	14	8	3	6	3	8	13	...	4	11	10	9	3	7	12	8.630137	4.472464	0.518238
FOODS_1_005	FOODS_1	FOODS	34	32	13	20	10	21	18	...	7	13	7	8	7	5	13	15.350685	12.902116	0.840491
FOODS_1_006	FOODS_1	FOODS	16	9	20	16	14	21	54	...	7	9	6	12	6	18	4	15.449315	8.290123	0.536601

```python
mean_prices = prices.groupby('item_id')['sell_price'].mean()
```

```
df_merged=df_grouped.merge(mean_prices,on=['item_id'],how='left').fillna({'sell_price': 0})

df_merged['turnover'] = df_merged.apply(lambda row: row[COLS_DATE] * row['sell_price'],
axis=1).sum(axis=1)
```

```
# Calculate TO%
total_turnover = df_merged['turnover'].sum()
df_merged['TO%'] = 100 * df_merged['turnover'] / total_turnover

# Sort by TO% in descending order
df_merged=df_merged.sort_values(by='TO%',ascending=False).reset_index(drop=True)
```

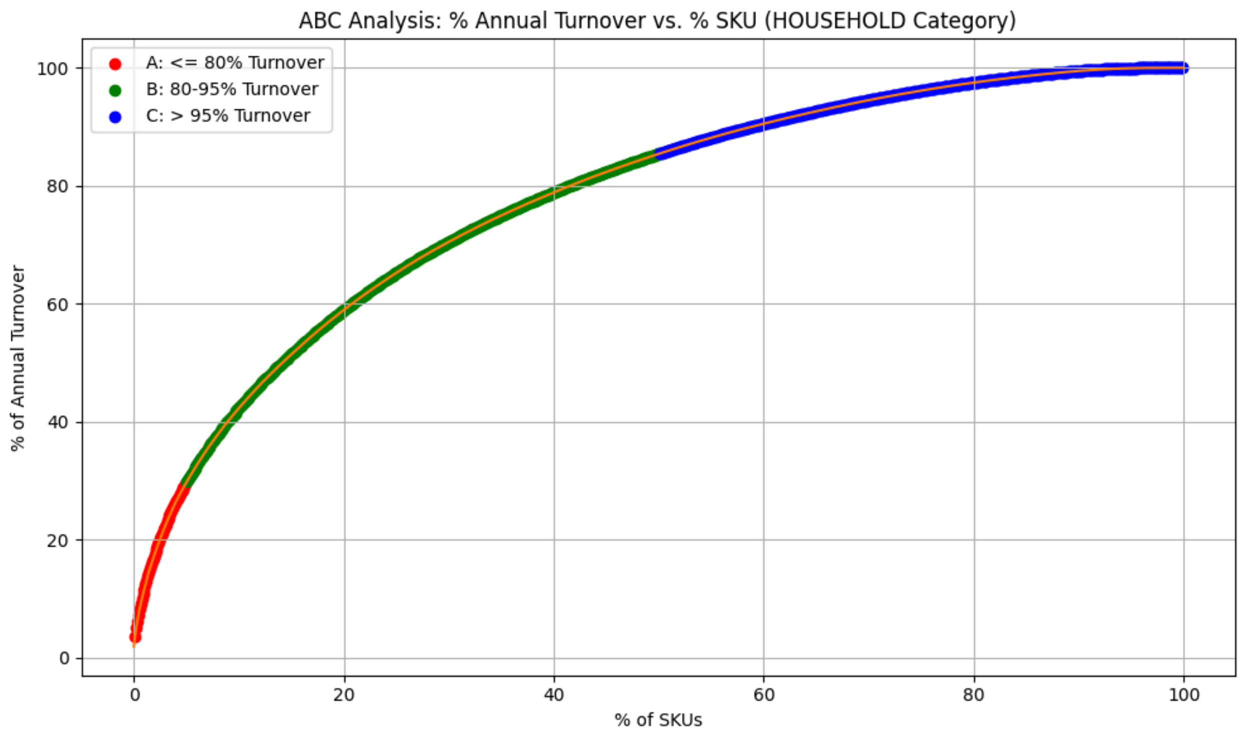
```
# Number of SKUs
len(df_merged)
```

```
912
```

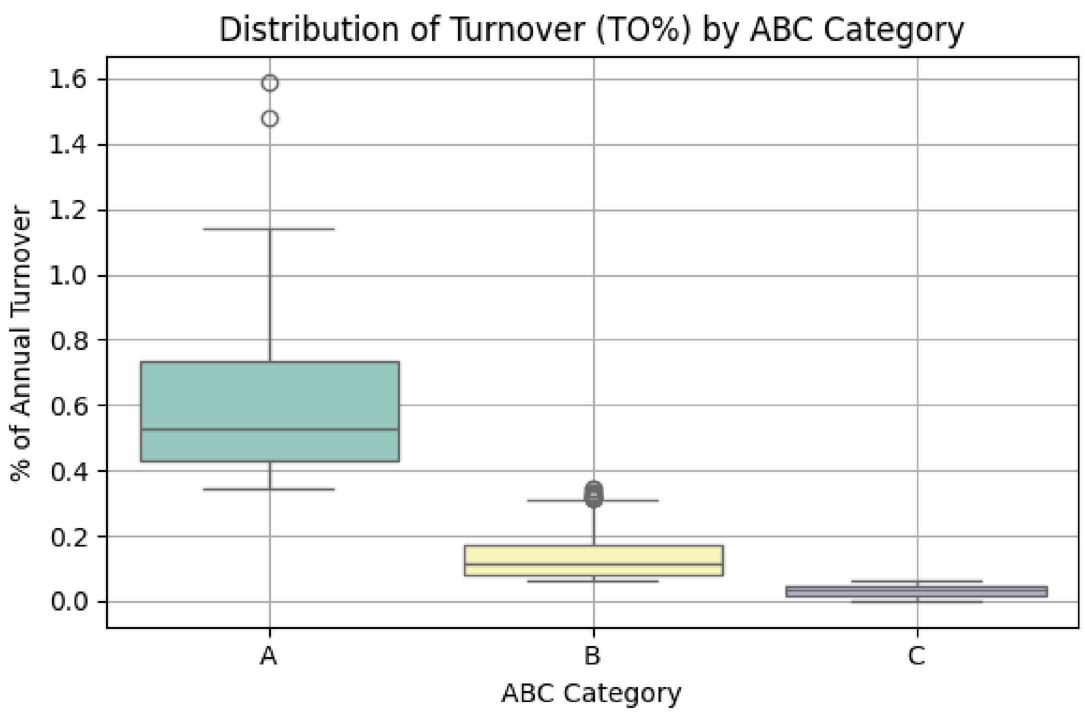
```
# Calculate cumulative percentage of turnover
df_merged['TO%_CS'] = df_merged['TO%'].cumsum()

# Determine boundaries for A, B, and C categories
n_sku = len(df_merged)
n_a, n_b = int(0.05 * n_sku), int(0.5 * n_sku)

# Classify SKUs into A, B, and C categories
df_merged['ABC'] = pd.cut(df_merged.index, bins=[0, n_a, n_b, n_sku], labels=['A', 'B', 'C'])
```

```
import matplotlib.pyplot as plt

# Plotting the Pareto curve
plt.figure(figsize=(10, 6))

# Plot cumulative percentage of turnover vs. cumulative percentage of SKUs
plt.plot(df_merged.index / n_sku * 100, df_merged['TO%_CS'], marker='', linestyle='-', color='C1')

# Markers for A, B, and C categories
plt.scatter(df_merged[df_merged['ABC'] == 'A'].index / n_sku * 100, df_merged[df_merged['ABC'] ==
'A']['TO%_CS'], color='r', label='A: <= 80% Turnover')
```

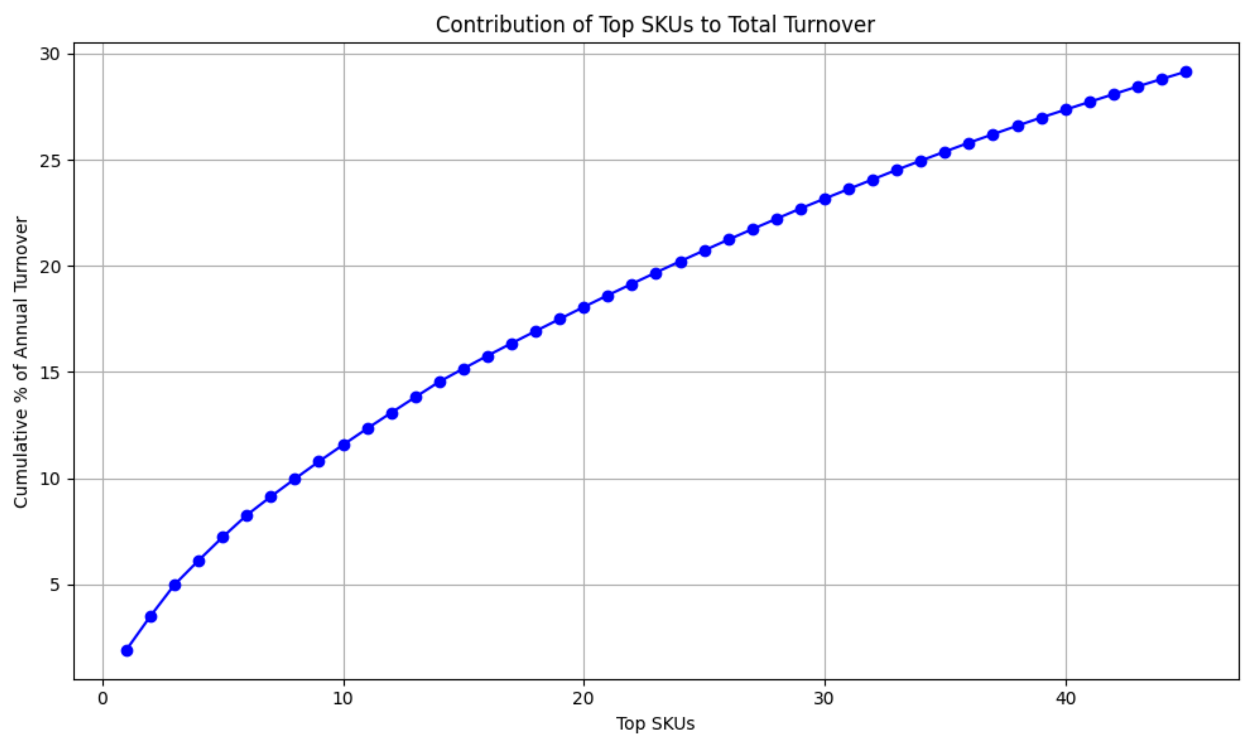
```
plt.scatter(df_merged[df_merged['ABC'] == 'B'].index / n_sku * 100, df_merged[df_merged['ABC'] ==
'B']['TO%_CS'], color='g', label='B: 80-95% Turnover')
plt.scatter(df_merged[df_merged['ABC'] == 'C'].index / n_sku * 100, df_merged[df_merged['ABC'] ==
'C']['TO%_CS'], color='b', label='C: > 95% Turnover')

# Plot settings
plt.title('ABC Analysis: % Annual Turnover vs. % SKU (HOUSEHOLD Category)')
plt.xlabel('% of SKUs')
plt.ylabel('% of Annual Turnover')
plt.legend(loc='best')
plt.grid(True)
plt.tight_layout()
plt.show()
```



```
# Plotting distribution of TO% by ABC category
plt.figure(figsize=(6, 4))
sns.boxplot(x='ABC', y='TO%', data=df_merged, palette='Set3')
plt.title('Distribution of Turnover (TO%) by ABC Category')
plt.xlabel('ABC Category')
plt.ylabel('% of Annual Turnover')
plt.grid(True)
plt.tight_layout()
plt.show()
```

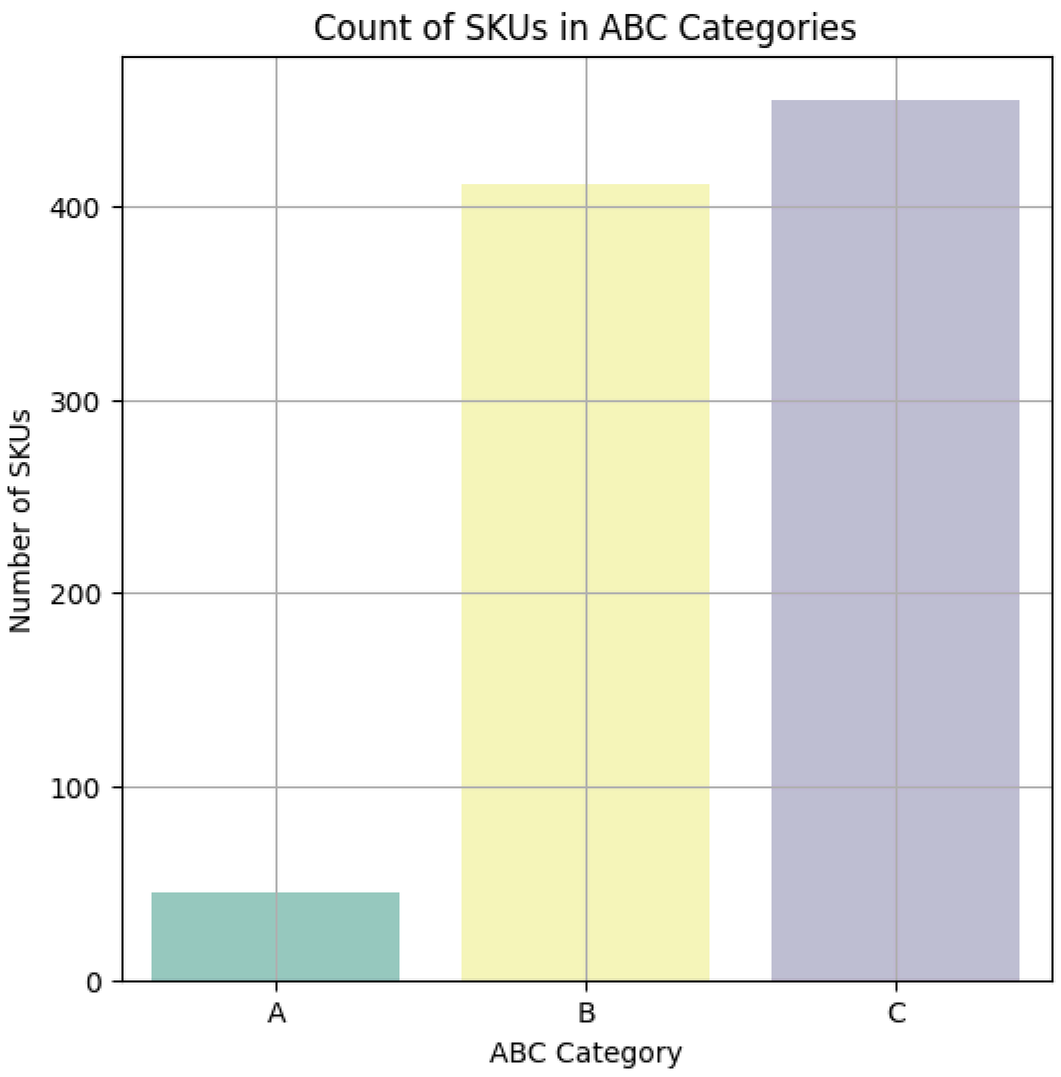


```
# Calculate cumulative TO% for top SKUs
top_sku_count = int(0.05 * len(df_merged))  # Example: Top 5% SKUs
df_top_skus = df_merged.head(top_sku_count)
df_top_skus['Cumulative TO%'] = df_top_skus['TO%'].cumsum()
# Plotting cumulative TO% for top SKUs
plt.figure(figsize=(10, 6))
plt.plot(range(1, top_sku_count + 1), df_top_skus['Cumulative TO%'], marker='o', linestyle='-', color='b')
plt.title('Contribution of Top SKUs to Total Turnover')
plt.xlabel('Top SKUs')
plt.ylabel('Cumulative % of Annual Turnover')
plt.grid(True)
plt.show()
```



This graph shows the turnover because of the top 5% SKUs.

```
# Assuming df_merged contains the DataFrame after ABC classification
# Count the number of SKUs in each ABC category
abc_counts = df_merged['ABC'].value_counts()

# Plotting the count of A, B, C categories
plt.figure(figsize=(6, 6))
sns.barplot(x=abc_counts.index, y=abc_counts.values, palette='Set3')
plt.title('Count of SKUs in ABC Categories')
plt.xlabel('ABC Category')
plt.ylabel('Number of SKUs')

plt.grid(True)
plt.show()
```



Demand Variability

Concept

Demand variability analysis focuses on understanding how sales or demand for products fluctuate over time. It helps in predicting future demand, optimizing inventory levels, and ensuring supply meets customer needs efficiently.

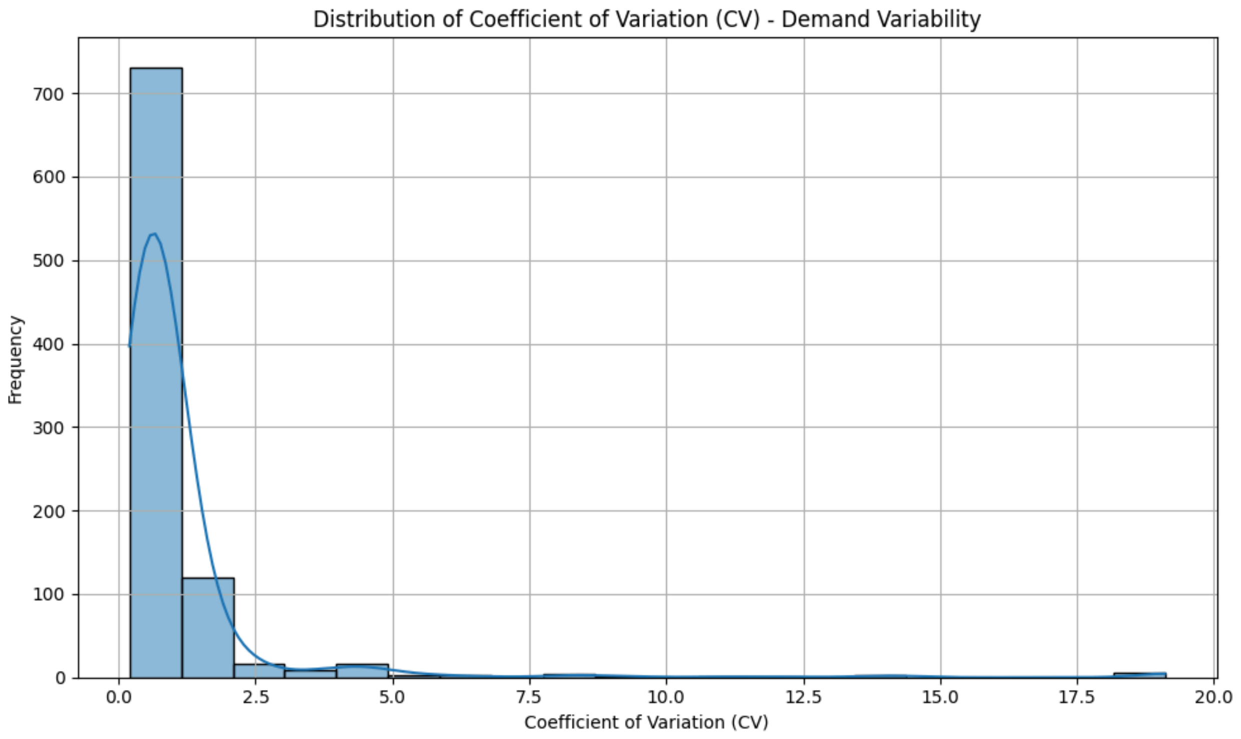
Graphs and Analysis:

1. Seasonal Sales Pattern:
The second graph shows the seasonal sales pattern month-wise. By aggregating daily sales data into monthly sums, it reveals recurring patterns or fluctuations in sales across different months of the year. This insight is crucial for seasonal inventory planning, marketing campaigns, and operational adjustments to meet changing demand.

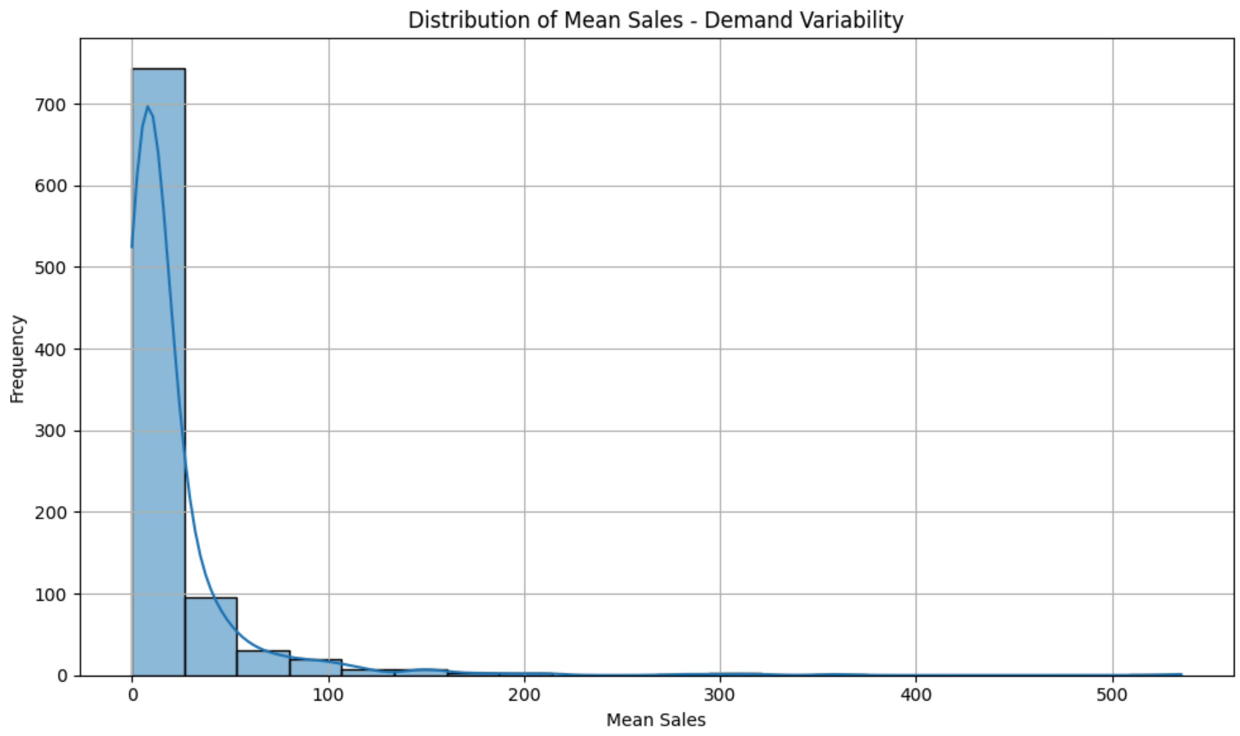
2. Sales Variability over Time:
A third graph examines how sales vary over time for selected SKUs. It helps identify trends, spikes, or periods of low activity in sales, providing insights into SKU-level demand volatility and potential factors influencing sales fluctuations.

The code

```
df_grouped['CV'] = df_grouped['std'] / df_grouped['mean']

# Plotting demand variability (CV)
plt.figure(figsize=(10  , 6))
sns.histplot(df_grouped['CV'], bins=20, kde=True)
plt.title('Distribution of Coefficient of Variation (CV) - Demand Variability')
plt.xlabel('Coefficient of Variation (CV)')
plt.ylabel('Frequency')
plt.grid(True)
plt.tight_layout()
plt.show()
```

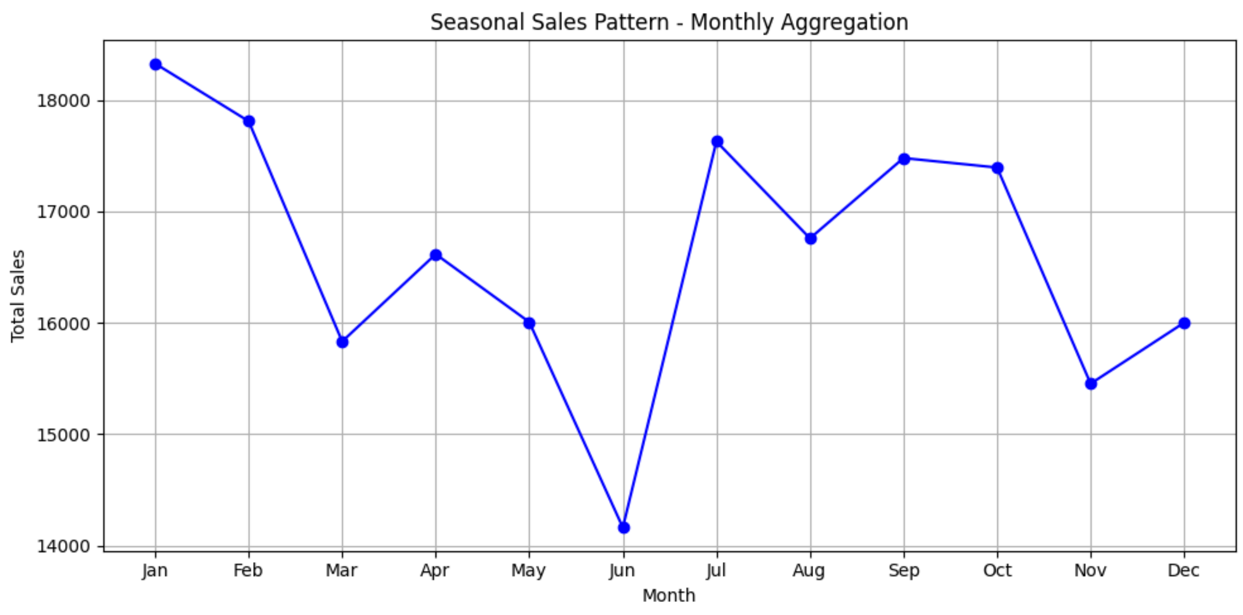


```
# Plotting distribution of mean sales
plt.figure(figsize=(10, 6))
sns.histplot(df_grouped['mean'], bins=20, kde=True)
plt.title('Distribution of Mean Sales - Demand Variability')
plt.xlabel('Mean Sales')
plt.ylabel('Frequency')
plt.grid(True)
plt.tight_layout()
plt.show()
```



```
# Example: Creating a dataframe with day numbers and daily sales data
days_in_year = np.arange(1, 367)  # Assuming you have days 1 to 366
np.random.seed(0)
sales_data = np.random.randint(100, 1000, 366)  # Example of daily sales data
df_filtered = pd.DataFrame({'day': days_in_year, 'sales': sales_data})

# Aggregate sales data by month
df_filtered['month'] = np.ceil(df_filtered['day'] / 30).astype(int)
df_monthly_sales = df_filtered.groupby('month')['sales'].sum()

# Plotting seasonal sales pattern month-wise
plt.figure(figsize=(10, 5))
```

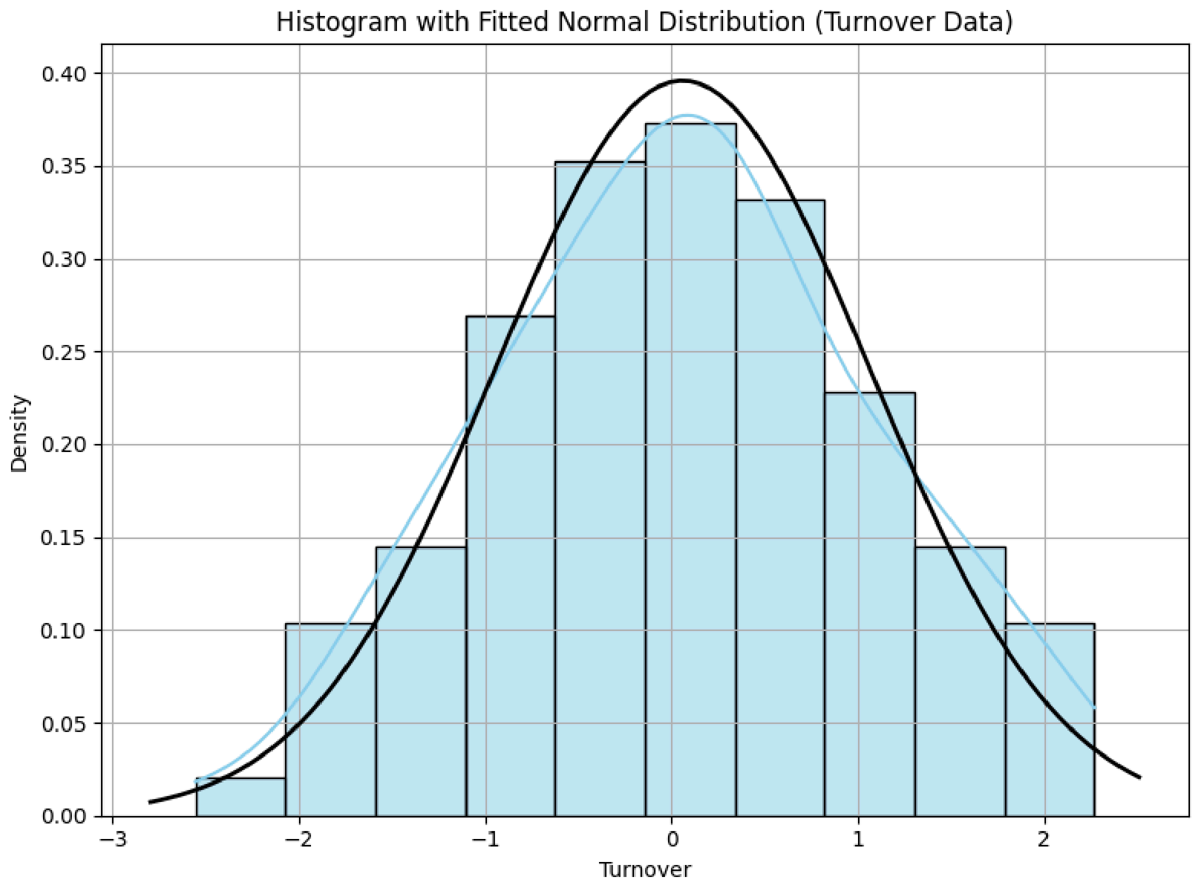
```
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
plt.plot(months[:12], df_monthly_sales.values[:12], marker='o', linestyle='-', color='b')
plt.title('Seasonal Sales Pattern - Monthly Aggregation')
plt.xlabel('Month')
plt.ylabel('Total Sales')
plt.grid(True)
plt.tight_layout()
plt.show()
```



Normality Test

```
from scipy.stats import shapiro
```

```
np.random.seed(0)
turnover_data = df_merged['turnover'].copy()
turnover_data = np.random.normal(loc=0, scale=1, size=100)  # Replace with df_abc['TO'] or actual turnover
data
# Performing Shapiro-Wilk test for normality
statistic, p_value = shapiro(turnover_data)
alpha = 0.05
print(f'Shapiro-Wilk Test - Statistic: {statistic}, p-value: {p_value}')
if p_value > alpha:
    print('Turnover data looks Gaussian (fail to reject H0)')
else:
    print('Turnover data does not look Gaussian (reject H0)')

# Plotting histogram with fitted normal curve
plt.figure(figsize=(8, 6))
sns.histplot(turnover_data, bins=10, kde=True, stat='density', color='skyblue', edgecolor='black')
# Overlaying normal distribution curve
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)
p = np.exp(-0.5 * ((x - np.mean(turnover_data)) / np.std(turnover_data))**2) / (np.sqrt(2 * np.pi) *
np.std(turnover_data))
plt.plot(x, p, 'k', linewidth=2)
plt.title('Histogram with Fitted Normal Distribution (Turnover Data)')
plt.xlabel('Turnover')
plt.ylabel('Density')
plt.grid(True)
plt.tight_layout()
plt.show()
```



These analyses and visualizations enable businesses to optimize inventory management strategies, enhance operational efficiency, and align supply with demand dynamics effectively. Adjustments in stocking levels, marketing initiatives, and strategic planning can leverage these insights to improve overall business performance and customer satisfaction.

Inventory

Aggregation by ABC Categories:
Simplifies the visualization by reducing the number of bars to three, one for each ABC category.

Interactive Plotting:
Provides an interactive way to explore the data, allowing you to zoom in and filter by categories.

Summary Statistics:
Offers a concise numerical summary of the data, useful for reporting and decision-making.

These approaches should help in making the inventory optimization insights clearer and more actionable, even with a large number of SKUs.

The Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
#Calculate Safety Stock and Reorder Point
ordering_cost_per_order = 50  # Example ordering cost per order
holding_cost_per_unit_per_year = 2 #Example holding cost per unit per year
lead_time_days = 5  # Example lead time in days
service_level_target = 0.95  # Example service level target
```

All these values can be done input by the user according to their needs.

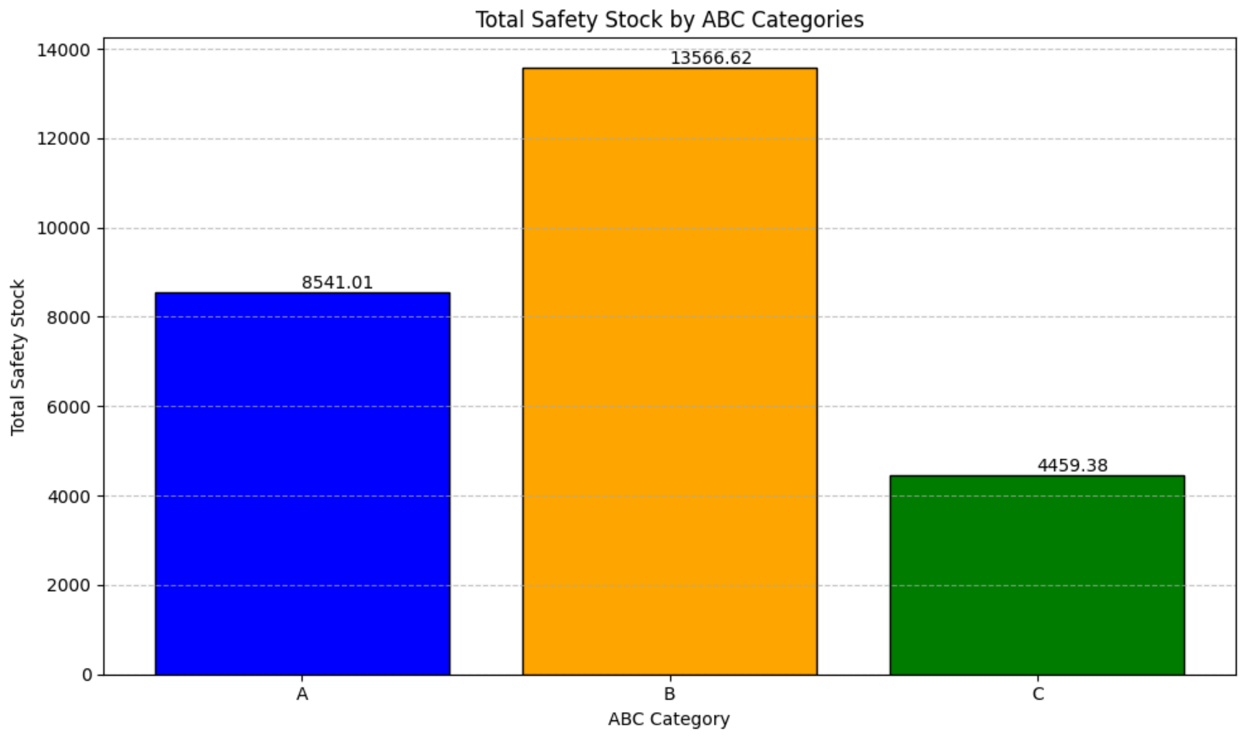
```
# Define demand variability thresholds
high_variability_threshold = 0.2 # Coefficient of Variation (CV) threshold

# Function to classify demand variability
def classify_demand_variability(row):
    if row['CV'] > high_variability_threshold or row['CV'] == np.NAN:
        return 'High Variability'
    else:
        return 'Low Variability'
```

```
df_abc['Demand_Variability'] = df_abc.apply(classify_demand_variability, axis=1)

# Define safety stock and reorder points
def calculate_safety_stock(row):
    if row['ABC'] == 'A' and row['Demand_Variability'] == 'High Variability':
        return row['mean'] * 2  # High safety stock
    elif row['ABC'] == 'A' or row['Demand_Variability'] == 'High Variability':
        return row['mean'] * 1.5  # Moderate safety stock
    else:
        return row['mean']  # Low safety stock

df_abc['Safety_Stock'] = df_abc.apply(calculate_safety_stock, axis=1)
df_abc['Reorder_Point'] = df_abc['mean'] * lead_time_days + df_abc['Safety_Stock']  # Assuming lead time of
5 days

print(df_abc[['item_id', 'ABC','CV', 'Demand_Variability', 'Safety_Stock', 'Reorder_Point']])
```

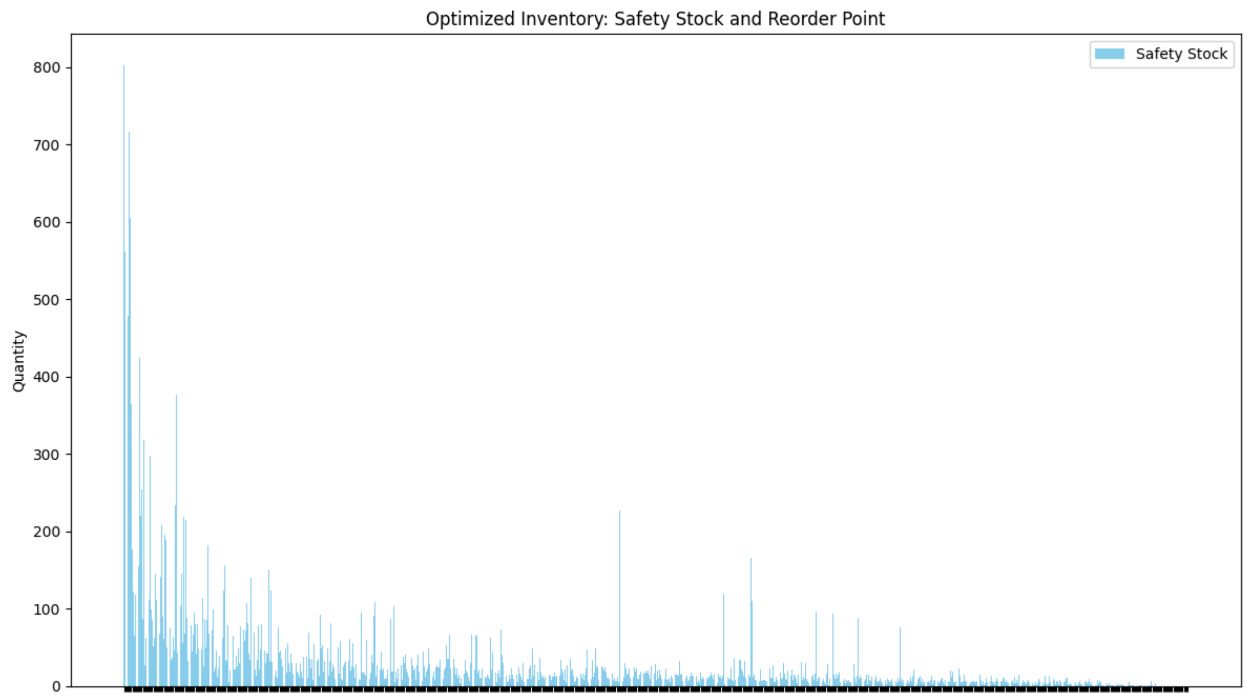
```
        item_id   ABC   CV   Demand_     Variability     Safety_Stocks
0    FOODS_3_586  NaN  0.792068  High Variability    802.882192
1    FOODS_3_587    A    0.589737  High Variability    561.835616
2    FOODS_3_202    A    0.518238  High Variability    309.150685
..          ...  ...      ...               ...           ...
907  FOODS_1_147    C   0.919869   High Variability       0.004110
908  FOODS_3_321    C       NaN       Low Variability       0.008219
909  FOODS_3_181    C       NaN       Low Variability       0.002740

     Reorder_Point
0      3479.156164
1      1966.424658
..             ...
907       0.017808
908       0.049315
909       0.016438
```

```
# Aggregate safety stock by ABC category
df_abc_agg = df_abc.groupby('ABC')['Safety_Stock'].sum().reset_index()
```

```
# Plot aggregated safety stock
plt.figure(figsize=(10, 6))
bars = plt.bar(df_abc_agg['ABC'], df_abc_agg['Safety_Stock'], color=['blue', 'orange', 'green'], edgecolor='black')
plt.title('Total Safety Stock by ABC Categories')
plt.xlabel('ABC Category')
plt.ylabel('Total Safety Stock')
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Adding data labels on top of the bars
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2.0, yval, round(yval, 2), va='bottom')

plt.tight_layout()
plt.show()
```

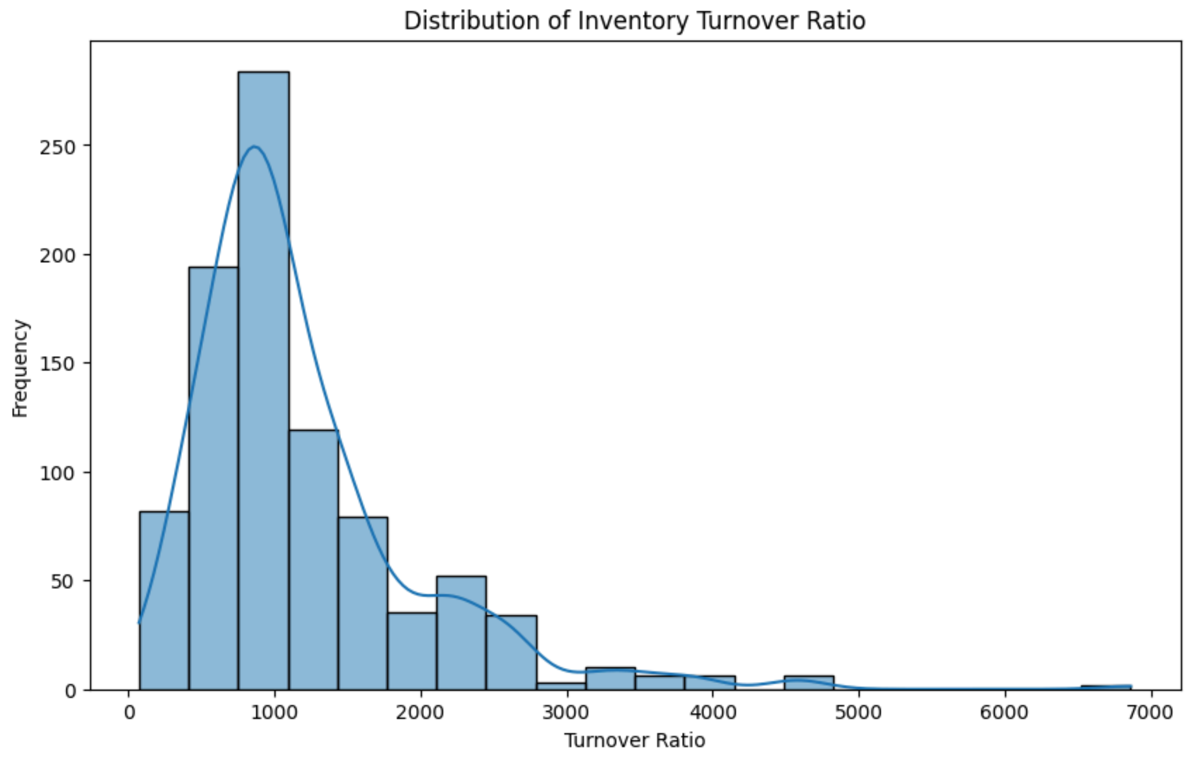


```
#Calculate Economic Order Quantity (EOQ)
df_abc['EOQ'] = np.sqrt((2 * df_abc['mean'] * ordering_cost_per_order) / holding_cost_per_unit_per_year)

#Visualize Results (Optional)
plt.figure(figsize=(12, 8))
plt.bar(df_abc['item_id'], df_abc['Safety_Stock'], color='skyblue', label='Safety Stock')
plt.xlabel('SKU')
plt.ylabel('Quantity')
plt.title('Optimized Inventory: Safety Stock and Reorder Point')
plt.legend()
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()

#Output Optimized Inventory Data
df_optimized_inventory = df_abc[['item_id', 'ABC', 'mean', 'std', 'CV', 'turnover', 'TO%', 'Safety_Stock', 'Reorder_Point', 'EOQ']]
```

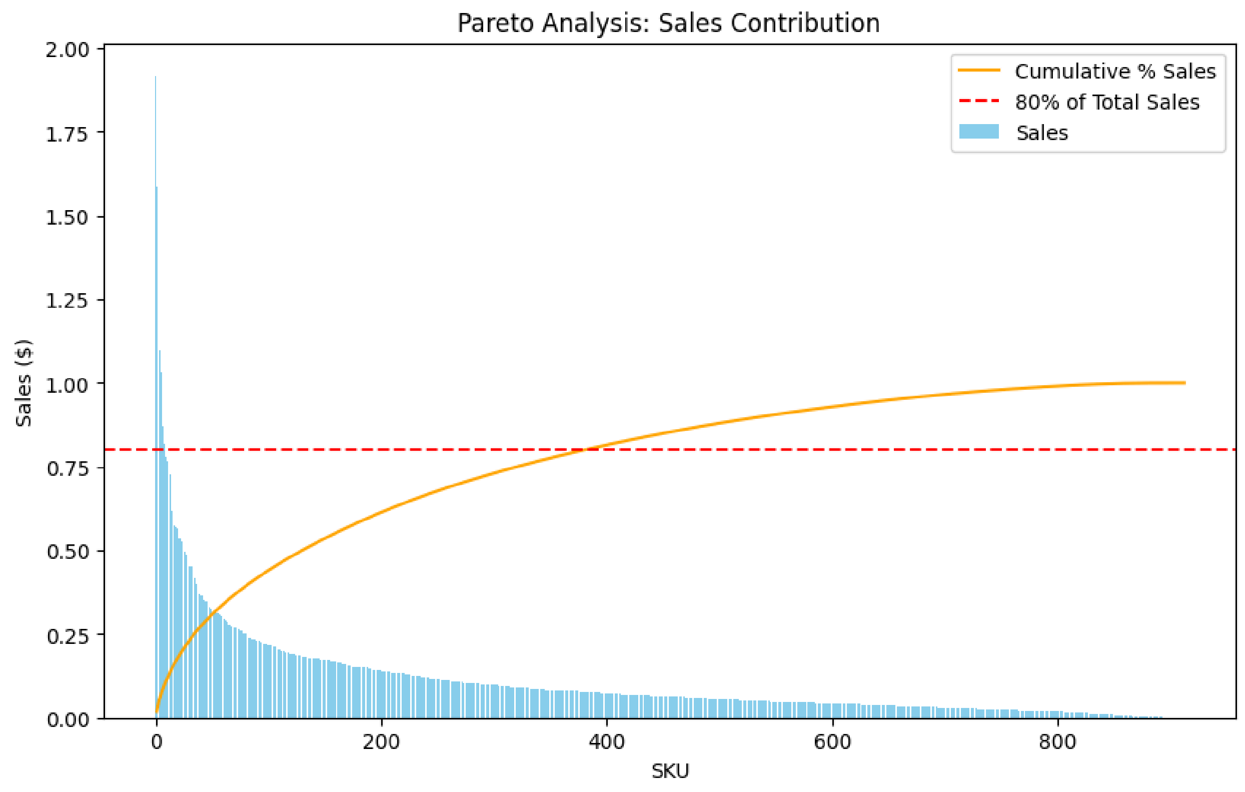


```
df_abc['Inventory_Turnover_Ratio'] = df_abc['turnover'] / df_abc['mean']
# Visualize Inventory Turnover Ratio
```

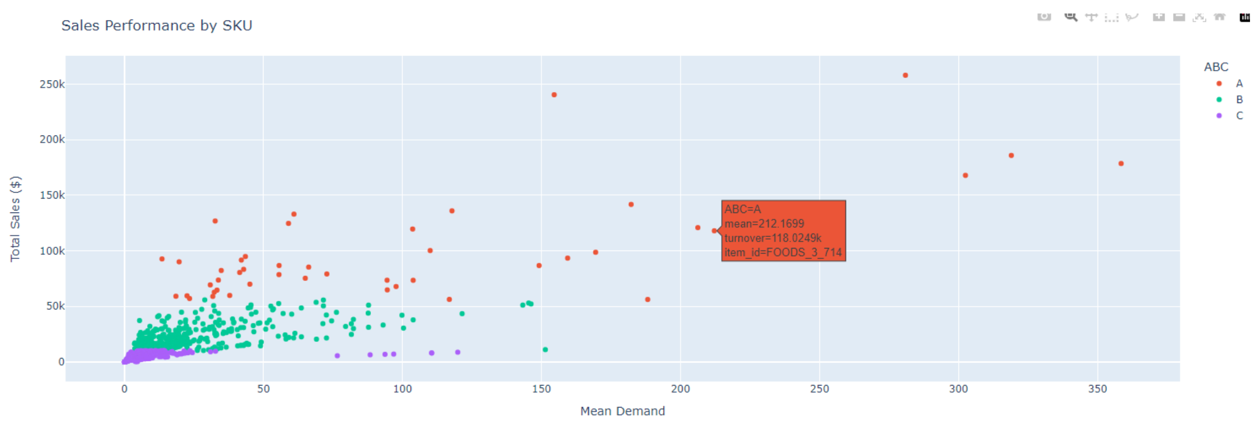
```
plt.figure(figsize=(10, 6))
sns.histplot(df_abc['Inventory_Turnover_Ratio'], bins=20, kde=True)
plt.title('Distribution of Inventory Turnover Ratio')
plt.xlabel('Turnover Ratio')
plt.ylabel('Frequency')
plt.show()
```



```
# Example Pareto Analysis: Top 20% of SKUs contributing to 80% of sales
total_sales = df_abc['turnover'].sum()
df_abc['TO%_CS'] = df_abc['turnover'].cumsum() / total_sales
plt.figure(figsize=(10, 6))
plt.bar(df_abc.index, df_abc['TO%'], color='skyblue', label='Sales')
plt.plot(df_abc.index+1 , df_abc['TO%_CS'], color='orange', label='Cumulative % Sales')
plt.axhline(y=0.8, color='r', linestyle='--', label='80% of Total Sales')
plt.title('Pareto Analysis: Sales Contribution')
plt.xlabel('SKU')
plt.ylabel('Sales ($)')
plt.legend()
plt.show()
```



```
import plotly.express as px

# Example Interactive Dashboard: Sales Performance by SKU
fig = px.scatter(df_abc, x='mean', y='turnover', color='ABC', hover_data=['item_id'])
fig.update_layout(title='Sales Performance by SKU',xaxis_title='Mean Demand',yaxis_title='Total Sales ($)')
fig.show()
```



This is an interactive dashboard also helps the user to see which SKU falls into what category. We can hover over them and check turnover and the item_id(product sold)

Route

The code

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
```

```python
def create_data_model():
    """Stores the data for the problem."""
    data = {}
    # Locations and distances (hypothetical data)
    data['locations'] = [
        (10, 0),     # Warehouse 1 (Depot)
        (1, 111),     # Warehouse 2 (Depot)
        (70, 21),     # Warehouse 3 (Depot)
        (3, 13),     # Location 1
        (4, 4),     # Location 2
        (52, 5),     # Location 3
        (69, 6),     # Location 4
        (7, 72),     # Location 5
        (8, 8),     # Location 6
        (91, 9),     # Location 7
        (10, 10),   # Location 8
        (11, 11),   # Location 9
        (12, 112),   # Location 10
        (132, 13),   # Location 11
        (14, 14),   # Location 12
        (15, 135)    # Location 13
    ]
    data['num_locations'] = len(data['locations'])
    data['num_vehicles'] = 3
    data['starts'] = [0, 1, 2]  # Start depot indices
    data['ends'] = [0, 1, 2]    # End depot indices

    return data
```

```
def compute_euclidean_distance_matrix(locations):
    """Creates callback to return distance between locations."""
    num_locations = len(locations)
    distance_matrix = {}

    for from_node in range(num_locations):
        distance_matrix[from_node] = {}
        for to_node in range(num_locations):
            if from_node == to_node:
                distance_matrix[from_node][to_node] = 0
            else:
                distance_matrix[from_node][to_node] = (
                    abs(locations[to_node][0] - locations[from_node][0]) +
                    abs(locations[to_node][1] - locations[from_node][1]))
    return distance_matrix
```

```
def main():
    # Instantiate the data problem.
    data = create_data_model()

    # Create the routing index manager.
    manager = pywrapcp.RoutingIndexManager(
        data['num_locations'], data['num_vehicles'], data['starts'], data['ends'])

    # Create Routing Model.
    routing = pywrapcp.RoutingModel(manager)

    # Create and register a transit callback.
    distance_matrix = compute_euclidean_distance_matrix(data['locations'])

    def distance_callback(from_index, to_index):
        """Returns the distance between the two nodes."""
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        return distance_matrix[from_node][to_node]

Transit_callback_index= routing.RegisterTransitCallback(distance_callback)
```

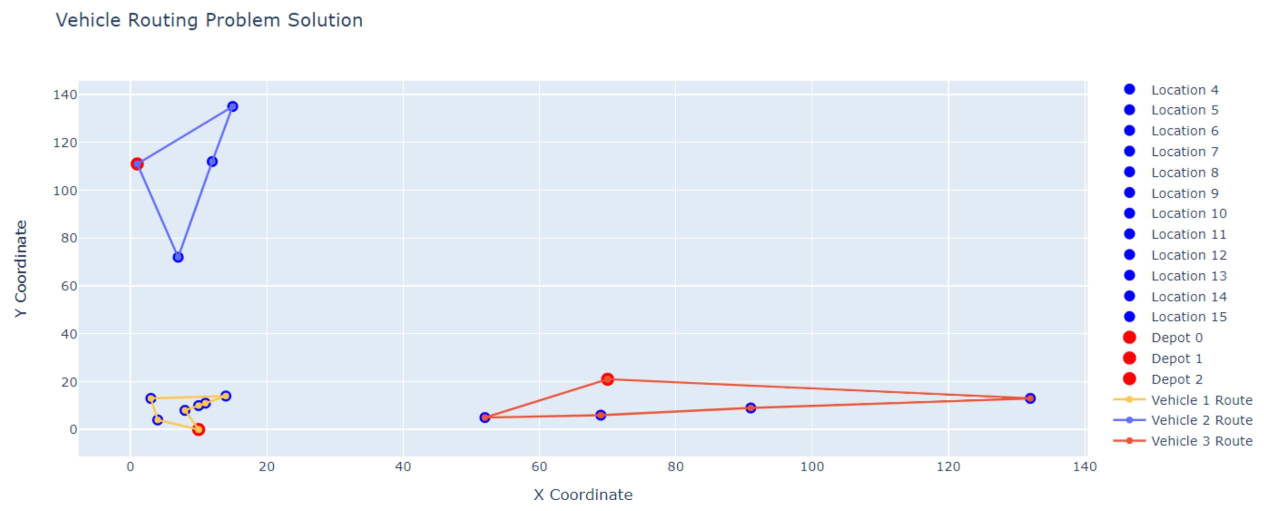
```
    # Define cost of each arc.
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    # Setting first solution heuristic.
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)

    # Solve the problem.
    solution = routing.SolveWithParameters(search_parameters)

    # Print solution on console.
    if solution:
        total_distance = 0
        for vehicle_id in range(data['num_vehicles']):
            index = routing.Start(vehicle_id)
            plan_output = 'Route for vehicle {}:\n'.format(vehicle_id)
            route_distance = 0
            while not routing.IsEnd(index):
                plan_output += ' {} -> '.format(manager.IndexToNode(index))
                previous_index = index
                index = solution.Value(routing.NextVar(index))
                route_distance += routing.GetArcCostForVehicle(previous_index, index, vehicle_id)
            plan_output += '{}\n'.format(manager.IndexToNode(index))
            plan_output += 'Distance of the route: {}m\n'.format(route_distance)
            print(plan_output)
            total_distance += route_distance
        print('Total Distance of all routes: {}m'.format(total_distance))
    else:
        print('No solution found!')


if __name__ == '__main__':
    main()
```

```
Route for vehicle 0:
 0 ->  4 ->  3 ->  14 ->  11 ->  10 ->  8 -> 0
Distance of the route: 54m

Route for vehicle 1:
 1 ->  7 ->  12 ->  15 -> 1
Distance of the route: 154m

Route for vehicle 2:
 2 ->  13 ->  9 ->  6 ->  5 -> 2
Distance of the route: 192m

Total Distance of all routes: 400m
```

```python
import plotly.graph_objects as go
import numpy as np

def main():
    # Instantiate the data problem.
    data = create_data_model()

    # Create the routing index manager.
    manager = pywrapcp.RoutingIndexManager(
        data['num_locations'], data['num_vehicles'], data['starts'], data['ends'])

    # Create Routing Model.
    routing = pywrapcp.RoutingModel(manager)

    # Create and register a transit callback.
    distance_matrix = compute_euclidean_distance_matrix(data['locations'])

    def distance_callback(from_index, to_index):
        """Returns the distance between the two nodes."""
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        return distance_matrix[from_node][to_node]

    transit_callback_index = routing.RegisterTransitCallback(distance_callback)

    # Define cost of each arc.
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    # Setting first solution heuristic.
```

```
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)

# Solve the problem.
solution = routing.SolveWithParameters(search_parameters)

# Extract and plot routes using Plotly.
if solution:
    fig = go.Figure()

    # Plot all locations
    for i, location in enumerate(data['locations']):
        x, y = location
        fig.add_trace(go.Scatter(x=[x], y=[y], mode='markers', name=f'Location {i}',
                                 marker=dict(size=10, color='blue')))

    # Plot depots (warehouses)
    for depot_index in data['starts']:
        x, y = data['locations'][depot_index]
        fig.add_trace(go.Scatter(x=[x], y=[y], mode='markers', name=f'Depot {depot_index}',
                                 marker=dict(size=12, color='red')))

    # Plot routes for each vehicle
    for vehicle_id in range(data['num_vehicles']):
        index = routing.Start(vehicle_id)
        route_x = []
        route_y = []
        while not routing.IsEnd(index):
            node_index = manager.IndexToNode(index)
            x, y = data['locations'][node_index]
            route_x.append(x)
            route_y.append(y)
            index = solution.Value(routing.NextVar(index))
        # Add the route back to the depot
        node_index = manager.IndexToNode(index)
        x, y = data['locations'][node_index]
        route_x.append(x)
        route_y.append(y)
```

```
            route_x.append(data['locations'][manager.IndexToNode(routing.Start(vehicle_id))][0])
            route_y.append(data['locations'][manager.IndexToNode(routing.Start(vehicle_id))][1])
             fig.add_trace(go.Scatter(x=route_x, y=route_y, mode='lines+markers', name=f'Vehicle {vehicle_id+1}
Route',
                                     line=dict(width=2)))

        # Customize layout
        fig.update_layout(title='Vehicle Routing Problem Solution',
                          xaxis_title='X Coordinate', yaxis_title='Y Coordinate',
                          showlegend=True)

        # Show plot
        fig.show()

        # Print total distance
        total_distance = sum([routing.GetArcCostForVehicle(
            manager.IndexToNode(index), solution.Value(routing.NextVar(index)), vehicle_id)
            for vehicle_id in range(data['num_vehicles'])
            for index in range(routing.Size()) if routing.IsVehicleUsed(solution, vehicle_id)])
        print(f'Total Distance of all routes: {total_distance}m')

    else:
        print('No solution found!')

if __name__ == '__main__':
    main()
```



Map which shows the optimized routes.

Data Preparation:
We create a hypothetical dataset representing store locations and use KMeans clustering to simplify the problem by reducing the number of locations.

Distance Matrix:
We compute the Euclidean distance between clustered locations to create the distance matrix.

VRP Model:
We use OR-Tools to define and solve the VRP, specifying the number of vehicles, the depot, and the distance constraints.

Solution Output:
The solution shows the optimized routes for each vehicle, minimizing the total distance traveled.

Practical Application

In a real-world scenario, you would replace the hypothetical data with your actual logistics data, including:

- Actual coordinates of warehouses and stores.
- Vehicle capacity and constraints.
- Delivery time windows.
- Traffic conditions and other real-world constraints.

This approach can significantly enhance the efficiency of your supply chain operations by optimizing delivery routes, reducing costs, and improving service levels.

Conclusion

The AI-Driven Supply Chain Management System presents a comprehensive solution to the inefficiencies plaguing modern supply chains. By leveraging real-time data, advanced machine learning algorithms, and predictive analytics, the system offers significant improvements in demand forecasting, inventory optimization, logistics planning, and product segmentation. This results in reduced operational costs, improved customer satisfaction, and enhanced overall efficiency.

As supply chains continue to evolve and face new challenges, the adoption of AI-driven solutions will be critical for businesses seeking to maintain a competitive edge. The AI-Driven Supply Chain Management System positions itself as a powerful tool to help businesses navigate these complexities and achieve optimal performance.