Supply Chain Analytics - Case Studies and Demos with Code

Yossiri Adulyasak

Oct 04, 2024

CONTENTS

This book, written by Yossiri Adulyasak, is a compilation of supply chain case studies and examples, each accompanied by Python codes that demonstrate the application of machine learning, data-driven optimization, and analytics to supply chain applications. This is a collection of supply chain examples and case studies with Python codes for the uses of machine learning, data-driven optimization and analtics for supply chain applications.

Part I

Case studies and Demos

CHAPTER

ONE

INTRODUCTION

This online book with code is designed to provide you with a comprehensive exploration of real-world applications and practical insights into the field of supply chain analytics.

Supply chain analytics has emerged as a critical discipline for businesses seeking to optimize their operations, reduce costs, and improve customer satisfaction. By leveraging data-driven insights, organizations can make informed decisions, identify inefficiencies, and gain a competitive edge.

Throughout this book, you will encounter a diverse collection of case studies from various industries, including manufacturing, retail, healthcare, and logistics. Each case study will delve into specific challenges faced by companies, the analytical approaches they employed, and the tangible results achieved.

By examining these real-world examples, you will gain a deeper understanding of how supply chain analytics can be applied to address a wide range of business problems. You will learn about the tools, techniques, and best practices that have proven successful in improving supply chain performance.

Whether you are a seasoned supply chain professional or a newcomer to the field, this book offers valuable insights and practical guidance. It is our hope that these case studies will inspire you to explore the possibilities of supply chain analytics and apply them to your own organization.

1.1 Prerequisite:

Prerequisites:

To fully grasp the concepts and techniques presented in this book, a foundational understanding of the following areas is essential:

- **Basic Python Programming for Data Analytics:** Proficiency in Python programming is crucial for implementing the analytical models and techniques discussed. A solid grasp of fundamental Python concepts, data structures, and libraries will be invaluable. If the reader does not have a supporting meterial for this, we refer the reader to our mini-book "Introduction to Data Analytics with Python for Supply Chains" by Yossiri Adulyasak and Martin Cousineau and the Python tutorial available on Kaggle author: Yossiri Adulyasak and Martin Cousineau
- **Machine Learning Models:** Familiarity with machine learning algorithms is necessary for understanding and applying predictive analytics in supply chain contexts. Knowledge of regression analysis, classification, clustering, and time series forecasting will be beneficial.
- **Optimization Models:** A foundation in optimization models is required to explore techniques for optimizing supply chain decisions. Understanding linear programming, integer programming, and other optimization methodologies will be helpful.

If you have a solid background in these areas, you are well-equipped to follow this online book of supply chain analytics case studies. Throughout the book, we will delve into specific challenges faced by companies, the analytical approaches they employed, and the Python code that demonstrates how the analytics tools can be implemented. By examining these real-world examples, you will gain a deeper understanding of how supply chain analytics can be applied to address a wide range of business problems.

Whether you are a seasoned supply chain professional or a newcomer to the field, this book offers valuable insights and practical guidance. It is our hope that these case studies will inspire you to explore the possibilities of supply chain analytics and apply them to your own organization.

1.1.1 Google Colab for Python Code

This course leverages Google Colab, a cloud-based platform for interactive computing, to facilitate your exploration of supply chain analytics concepts through code examples and demonstrations. Colab offers a user experience very similar to Jupyter Notebook (Anaconda), but with the added benefit of being free and readily accessible from any web browser. Each part of the online book includes a link that brings you direcrly to the code available in Google Colab.

A Google account is necessary to utilize Colab. You can conveniently use your Google email address to register for a free account if you haven't already.

Accessing Colab:

Visit the Colab Website: Navigate to https://colab.research.google.com using a preferred web browser. While most browsers are compatible, utilizing Google Chrome is recommended to ensure optimal performance and avoid potential technical issues. Sign Up with Google Account: To begin using Colab, sign in using your Google account credentials. Learning More about Colab:

Short Introduction: Gain a quick overview of Colab's functionalities through this YouTube video: https://www.youtube.com/watch?v=inN8seMm7UI **Detailed Guide:** For a more comprehensive understanding of Colab, explore the official Colab notebook introduction: https://colab.research.google.com/notebooks/intro.ipynb By utilizing Colab, you will be able to actively participate in coding exercises and demonstrations, solidifying your grasp of supply chain analytics concepts within a practical and interactive environment.

1.1.2 Python Local Installation

Instead of Google Colab, it is also possible to use Python locally on your machine by using Anaconda 3. Please refer to the mini-book "Introduction to Data Analytics with Python for Supply Chains" by Yossiri Adulyasak and Martin Cousineau or the instruction below. Anaconda is a powerful Python distribution that comes with a large collection of packages and tools that are essential for data science, machine learning, and scientific computing. Here's a brief guide on how to install Anaconda 3:

1. Download the Anaconda Installer:

- Visit the official Anaconda download page: https://www.anaconda.com/download
- Select the appropriate installer for your operating system (Windows, macOS, or Linux).

2. Run the Installer:

- Double-click the downloaded installer file.
- Follow the on-screen instructions.
- Make sure to choose the "Add Anaconda to your PATH" option during installation. This will allow you to access Anaconda commands from your terminal or command prompt.

3. Verify Installation:

- Open your terminal or command prompt.
- Type `conda --version` and press Enter.
- If Anaconda is installed correctly, you should see the installed version number.

Additional Notes:

- Anaconda creates a separate environment for Python and its packages, which helps in managing different projects and avoiding conflicts between libraries.
- You can use the `conda` command to create, activate, and deactivate different environments.
- For more information and advanced usage, refer to the Anaconda documentation: https://docs.anaconda.com/

CHAPTER

TWO

CASE: RETAIL ANALYTICS - RUE LA LA

Case Reference: Ferreira, K. J., Lee, B. H. A., & Simchi-Levi, D. (2016). Analytics for an online retailer: Demand forecasting and price optimization. Manufacturing & service operations management, 18(1), 69-88.

2.1 Overview of the case:

Rue La La, an online fashion retail company founded in 2007 in Boston, Massachusetts, specializes in offering limited-time sales on designer apparel and accessories. To cater to their online customers effectively, Rue La La faces several challenges in planning their product and pricing offerings. One such challenge is dealing with new products, as they lack historical data to accurately predict demand. Additionally, event factors like holidays or seasonal trends can significantly impact sales, making it difficult to forecast demand. Furthermore, the demand for one product can be influenced by the availability and pricing of competing styles, creating a complex interdependency that needs to be considered in their planning.

Predictive and Prescriptive Analytics:

- Predicting Demand: The company employs predictive analytics to estimate historical lost sales and forecast future demand for new products. Machine learning techniques are used to analyze factors like product attributes, seasonality, and historical sales data to predict demand.
- Optimizing Pricing: Prescriptive analytics is employed to develop an algorithm that determines optimal pricing strategies for multiple products simultaneously. This algorithm incorporates reference price effects, considering how customers perceive prices relative to their expectations.

Modeling the Effect of Competing Products: The change in the price of one product can impact the demand of other similar products. As a consequence, considering the pricing policy for a product in isolation may not result in the desired outcomes due to this cannibalization effect between similar products (i.e., competing products). This cross-item effect is indirectly taken into account in both the predictive and prescriptive models. More specifically, in the predictive model, the sum (or the average) of prices of all the competing products are also used as an independent variable or feature. In the prescriptive model, a constraint is imposed to enforce that the selected prices across all the competing products will be equal to the predefined value of the sum (or the average) of prices of all the competing products.

The image below (from Ferreira et al. 2018) presents an example of competing products.

The analytics workflow of Rue La La that comprises both the predictive and prescriptive analytics is depicted below.

The overall pipeline of analytics process at Rue La La is depicted below (figure from Ferreira et al. 2018)

2.2 Demo: Simplified retail prediction pipeline for a single item

Link to Google Colab of this Notebook

NOTE: This is a simplified version of the predictive model shown in the Module 1 - Predictive Analytics Model. In this version, we run only the model for one item. Almost all of the parts of the codes folllow the same logic and process as the Module 1A [Colab link for 1A] but it is simplified to one product (UPC). Once you are familiar with this one, it will be easy to understand the demo 1A which consists of multiple items.

In order to continue for S9 for the optimization model, we still need to run the Module 1A. Thus, please also proceed and run the Module 1A and save the fitted models to your Google Drive or local folder.

We begin by loading the required packages.

```
import pandas
import numpy
import sklearn
from sklearn import *

import matplotlib.pyplot as plt
```

2.2.1 Block 1: Data input

In addition to the original data, we add a new variable, which is the squared price (‘PRICE_p2’).

```
url = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/S08_09_Retail_
↪Analytics/salesCereals.csv'

salesCereals = pandas.read_csv(url)
salesCereals['PRICE_p2'] = salesCereals['PRICE']**2
salesCereals.head()
```

```
   Unnamed: 0 WEEK_END_DATE  STORE_NUM         UPC  UNITS  VISITS   HHS  \
0           6    2009-01-14      367.0  1111085319   14.0    13.0  13.0
1           8    2009-01-14      367.0  1111085350   35.0    27.0  25.0
2          12    2009-01-14      367.0  1600027527   12.0    10.0  10.0
3          13    2009-01-14      367.0  1600027528   31.0    26.0  19.0
4          14    2009-01-14      367.0  1600027564   56.0    48.0  42.0

    SPEND  PRICE  BASE_PRICE  ...  DISPLAY  TPR_ONLY  \
0   26.32   1.88        1.88  ...      0.0       0.0
1   69.30   1.98        1.98  ...      0.0       0.0
2   38.28   3.19        3.19  ...      0.0       0.0
3  142.29   4.59        4.59  ...      0.0       0.0
4  152.32   2.72        3.07  ...      0.0       0.0

                       Desc      Category        Sub-Category SUMPRICE  \
0  PL HONEY NUT TOASTD OATS  COLD CEREAL  ALL FAMILY CEREAL    19.54
1   PL BT SZ FRSTD SHRD WHT  COLD CEREAL  ALL FAMILY CEREAL    19.54
2     GM HONEY NUT CHEERIOS  COLD CEREAL  ALL FAMILY CEREAL    19.54
3               GM CHEERIOS  COLD CEREAL  ALL FAMILY CEREAL    19.54
4               GM CHEERIOS  COLD CEREAL  ALL FAMILY CEREAL    19.54

   COUNTPRICE  AVGPRICE  RELPRICE  PRICE_p2
```

(continues on next page)

(continued from previous page)

```
0           7  2.791429  0.673490    3.5344
1           7  2.791429  0.709314    3.9204
2           7  2.791429  1.142784   10.1761
3           7  2.791429  1.644319   21.0681
4           7  2.791429  0.974411    7.3984

[5 rows x 21 columns]
```

'UPC' stands for Universal Product Code, which can be understood as one SKU in this case and in our SCM terms in general. The code below helps us identify the SKUs by which we want to forecast and their corresponding data size (number of data instances). We can see that the number of instances for each UPC is similar and that there is no UPC with only a few data points. This is important because training a model on a small dataset may limit its generalization.

```
print(salesCereals.groupby('UPC').count())
```

```
            Unnamed: 0  WEEK_END_DATE  STORE_NUM  UNITS  VISITS  HHS  SPEND  \
UPC
1111085319         156            156        156    156     156  156    156
1111085350         156            156        156    156     156  156    156
1600027527         156            156        156    156     156  156    156
1600027528         156            156        156    156     156  156    156
1600027564         155            155        155    155     155  155    155
3000006340         133            133        133    133     133  133    133
3800031829         155            155        155    155     155  155    155

            PRICE  BASE_PRICE  FEATURE  DISPLAY  TPR_ONLY  Desc  Category  \
UPC
1111085319    156         156      156      156       156   156       156
1111085350    156         156      156      156       156   156       156
1600027527    156         156      156      156       156   156       156
1600027528    156         156      156      156       156   156       156
1600027564    155         155      155      155       155   155       155
3000006340    133         133      133      133       133   133       133
3800031829    155         155      155      155       155   155       155

            Sub-Category  SUMPRICE  COUNTPRICE  AVGPRICE  RELPRICE  PRICE_p2
UPC
1111085319           156       156         156       156       156       156
1111085350           156       156         156       156       156       156
1600027527           156       156         156       156       156       156
1600027528           156       156         156       156       156       156
1600027564           155       155         155       155       155       155
3000006340           133       133         133       133       133       133
3800031829           155       155         155       155       155       155
```

2.2.2 Block 2: Feature engineering & preparation

We then organize the data by 'UPC.' The model presented here only runs on a predetermined subset of variables in the data. You can add or remove these explanatory variables based on your judgemental call.

Here we select only **one** upc to run the model.

```
feature_list = ['PRICE', 'PRICE_p2', 'FEATURE', 'DISPLAY','TPR_ONLY','RELPRICE']

productList = salesCereals['UPC'].unique()
upc = 1600027528

X = salesCereals.loc[salesCereals['UPC']==upc][feature_list]
y = salesCereals.loc[salesCereals['UPC']==upc]['UNITS']
  # Split into training and testing data
X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(X, y,␣
↪random_state=0)
```

2.2.3 Block 3: Model & algorithm (training & testing)

In the next two cells, we train and test two different types of models, namely Linear Regression and Tree Regression. The first line in each loop is to train the model and the second line is for testing the model's performance on unseen data. The next three lines compute the performance metrics we would like to measure. Then we compute metrics to show the performance of the model.

```
#Linear model

# Fit the model
regr = sklearn.linear_model.LinearRegression().fit(X_train,y_train)

# Measure the RSME on the training set
trainRMSE = numpy.sqrt(sklearn.metrics.mean_squared_error(y_train, regr.predict(X_
↪train)))

# Prediction on the test set
y_pred = regr.predict(X_test)

# Measure the prediction performances on the test set
testMAE = sklearn.metrics.mean_absolute_error(y_test, y_pred)
testMAPE = numpy.mean(numpy.abs((y_test - y_pred) / y_test))
testRMSE = numpy.sqrt(sklearn.metrics.mean_squared_error(y_test, y_pred))

print('Linear regression Summary - UPC:'+str(upc))
print('Training RMSE:' + str(round(trainRMSE,2)))
print('Testing MAE:' + str(round(testMAE,2)))
print('Testing MAPE:' + str(round(testMAPE,2)))
print('Testing RMSE:' + str(round(testRMSE,2)))
```

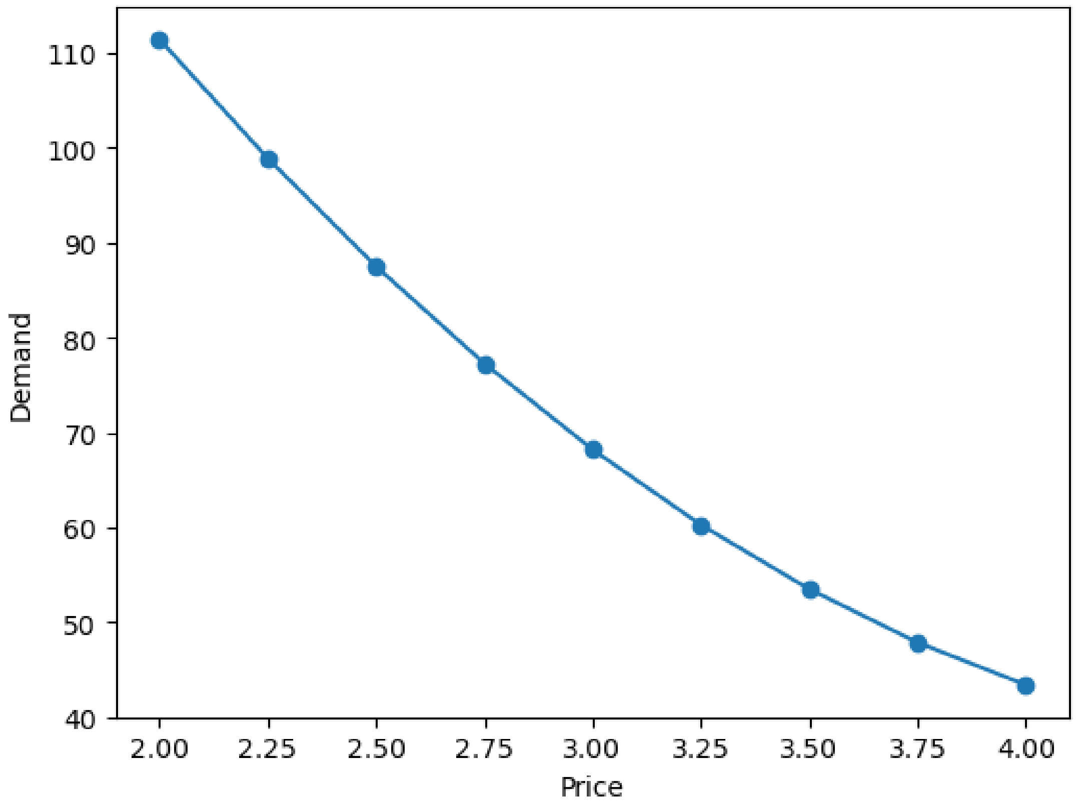
```
Linear regression Summary - UPC:1600027528
Training RMSE:9.07
Testing MAE:8.09
Testing MAPE:0.25
Testing RMSE:14.07
```

In order to see the impact of the price on the demand, we use a simple plot function below from mathplotlib to see how the demand would change when the price changes.

For more details of the plot function, please see: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html

```
prices = [2.0, 2.25, 2.5, 2.75, 3.0, 3.25, 3.5, 3.75, 4.0]
input_x = []

# generate inputs for the plot using simple feature values and varying price points
for p in prices:
  input_x.append([p, p**2, 0,0,0, 1.0])

# obtain the predicted demands
predict_y = regr.predict(input_x)
plt.plot(prices, predict_y, marker='o')
plt.xlabel('Price')
plt.ylabel('Demand')
plt.show()
```

```
D:\ProgramData\Anaconda3\envs\sca_book2\lib\site-packages\sklearn\base.py:464:␣
↪UserWarning: X does not have valid feature names, but LinearRegression was␣
↪fitted with feature names
  warnings.warn(
```



Likewise, we obtain the tree regression results by simply changing the function name. Here you can try the regression tree and random forest (second model) if you outcomment it.

```
#Tree models
# regr = sklearn.tree.DecisionTreeRegressor(random_state = 0).fit(X_train,y_train) #␣
↪standard regression tree
regr = sklearn.ensemble.RandomForestRegressor(random_state = 0).fit(X_train,y_train)
↪# random forest tree

# Measure the RSME on the training set
trainRMSE = numpy.sqrt(sklearn.metrics.mean_squared_error(y_train, regr.predict(X_
↪train)))

# Prediction on the test set
y_pred = regr.predict(X_test)

# Measure the prediction performances on the test set
testMAE = sklearn.metrics.mean_absolute_error(y_test, y_pred)
testMAPE = numpy.mean(numpy.abs((y_test - y_pred) / y_test))
testRMSE = numpy.sqrt(sklearn.metrics.mean_squared_error(y_test, y_pred))

print('Tree regression Summary - UPC:'+str(upc))
print('Training RMSE:' + str(round(trainRMSE,2)))
print('Testing MAE:' + str(round(testMAE,2)))
print('Testing MAPE:' + str(round(testMAPE,2)))
print('Testing RMSE:' + str(round(testRMSE,2)))
```

```
Tree regression Summary - UPC:1600027528
Training RMSE:4.92
Testing MAE:8.7
Testing MAPE:0.28
Testing RMSE:14.53
```

```
# plot to see how the results look like when changing prices
prices = [2.0, 2.25, 2.5, 2.75, 3.0, 3.25, 3.5, 3.75, 4.0]
input_x = []

# generate inputs for the plot using simple feature values and varying price points
for p in prices:
  input_x.append([p, p**2, 0,0,0, 1.0])

input_x
```

```
[[2.0, 4.0, 0, 0, 0, 1.0],
 [2.25, 5.0625, 0, 0, 0, 1.0],
 [2.5, 6.25, 0, 0, 0, 1.0],
 [2.75, 7.5625, 0, 0, 0, 1.0],
 [3.0, 9.0, 0, 0, 0, 1.0],
 [3.25, 10.5625, 0, 0, 0, 1.0],
 [3.5, 12.25, 0, 0, 0, 1.0],
 [3.75, 14.0625, 0, 0, 0, 1.0],
 [4.0, 16.0, 0, 0, 0, 1.0]]
```

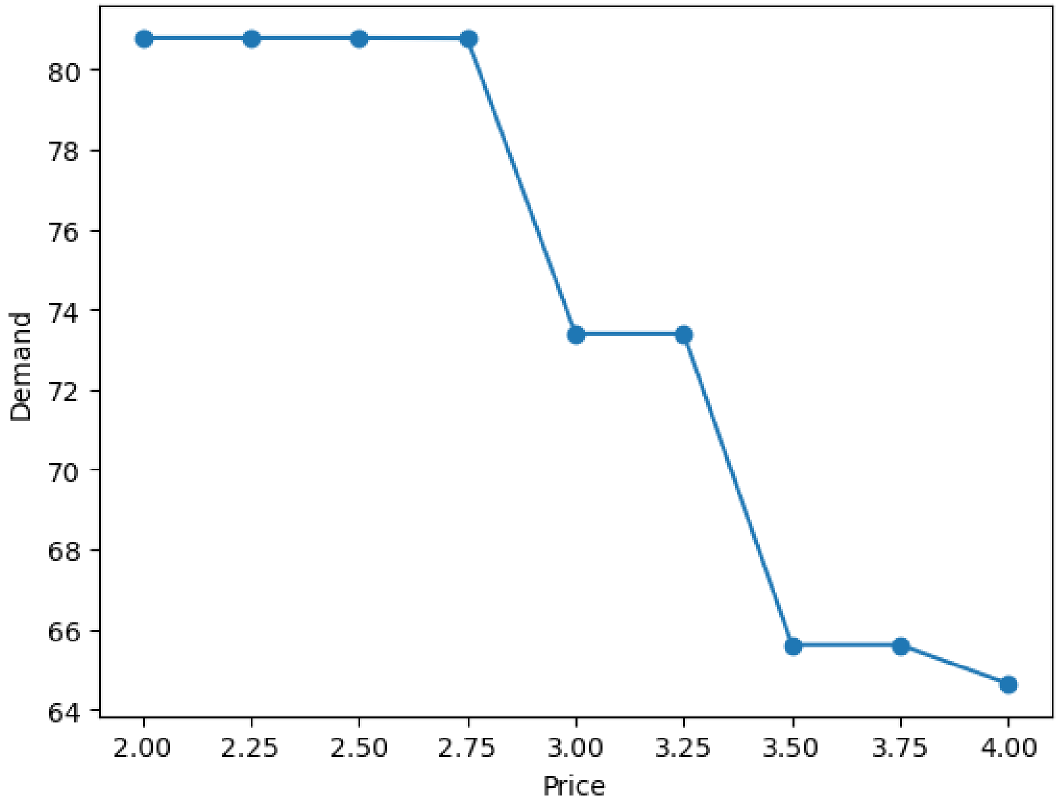
```
# obtain the predicted demands
predict_y = regr.predict(input_x)
plt.plot(prices, predict_y, marker='o')
plt.xlabel('Price')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Demand')
plt.show()
```

```
D:\ProgramData\Anaconda3\envs\sca_book2\lib\site-packages\sklearn\base.py:464:␣
↪UserWarning: X does not have valid feature names, but RandomForestRegressor was␣
↪fitted with feature names
  warnings.warn(
```



2.2.4 Block 4: Model selection

By comparing the average result, we can see that the linear regression model slightly outperformed the decision tree regression and did not overfit the data. In addtion, the predicted function has a better representation since the changes are monotonic (from the plots). Therefore, we proceed with the linear regression model for the whole dataset by replacing 'X_train' with 'X'. Given that the model has 'seen' the whole dataset, its forecast errors normally decrease. Therefore, we will save the trained model and use it for the new data which will be used in the optimization models in the next session.

```
# Selected model

# Fit the model on the entire dataset
regr = sklearn.linear_model.LinearRegression().fit(X,y)

# Prediction on the test set
y_pred = regr.predict(X)
```

(continues on next page)

(continued from previous page)

```
# Measure the prediction performances on the entire dataset
overallMAE = sklearn.metrics.mean_absolute_error(y, y_pred)
overallMAPE = numpy.mean(numpy.abs((y - y_pred) / y))
overallRMSE = numpy.sqrt(sklearn.metrics.mean_squared_error(y, y_pred))

print('Regression Summary - UPC:'+str(upc))
print('Overall MAE:' + str(round(overallMAE,2)))
print('Overall MAPE:' + str(round(overallMAPE,2)))
print('Overall RMSE:' + str(round(overallRMSE,2)))
```

```
Regression Summary - UPC:1600027528
Overall MAE:7.06
Overall MAPE:0.3
Overall RMSE:10.43
```

2.2.5 Save trained models

If you use Jupyter, you can save it to a local folder. The code below will put it in the current folder.

```
cwd = './'
```

Now we can save the files to the folder indicated by using the code below.

```
# save the models to drive (here we save model only for one UPC).
import pickle

filename = cwd+str(upc)+'_single_upc_demand_model.sav'
# save the model to disk
pickle.dump(regr, open(filename, 'wb'))
```

2.3 Module 1A: Retail predictive model pipeline for multiple items

Link to Google Colab of this Notebook

This is the full version of the Module 1 (predictive model) which extends the simplified version of one item described previously to multiple items. Almost all of the parts of the codes folllow the same logic and process as the simplified Module 1 for one product (UPC).

We begin by loading the required packages.

```
import pandas
import numpy
import sklearn
from sklearn import *
```

2.3.1 *Supplement - Plot functions (this is a pre-built plot function)*

They will be used later on for visualizations. There is no need to go through them. You only need to run the codes.

```
import matplotlib.pyplot as plt

#See https://matplotlib.org/devdocs/gallery/subplots_axes_and_figures/subplots_demo.
↪html

def plot_data_scatter(data_x, data_y, X_test, y_pred, feature_list):
    # Plot the results

    n_row_plot = int((len(feature_list)+1)/2) # 2 plots per row
    n_col_plot = 2
    fig, ax = plt.subplots(n_row_plot, n_col_plot, figsize=(12, 12))

    i = 0 # column index of the plot
    j = 0 # row index of the plot

    for count in range(len(feature_list)):
        #print(data_x[:,i])
        ax[j, i].scatter(data_x[:,min(count,len(feature_list))], data_y, s=20,␣
↪edgecolor="black",
                    c="darkorange", label="data")
        ax[j, i].scatter(X_test.values[:,min(count,len(feature_list))], y_pred, s=30,␣
↪marker="X",
                    c="royalblue", label="prediction")
        ax[j, i].set(title=feature_list[count])

        ax[j, i].set(ylabel='UNITS')

        i = min(i+1,len(feature_list)) % n_col_plot
        if i == 0: j += 1

    plt.show()
```

2.3.2 Block 1: Data input

In addition to the original data, we add a new variable, which is the squared price (‘PRICE_p2’).

```
url = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/S08_09_Retail_
↪Analytics/salesCereals.csv'

salesCereals = pandas.read_csv(url)
salesCereals['PRICE_p2'] = salesCereals['PRICE']**2
salesCereals.head()
```

```
   Unnamed: 0 WEEK_END_DATE  STORE_NUM         UPC  UNITS  VISITS   HHS  \
0           6    2009-01-14      367.0  1111085319   14.0    13.0  13.0
1           8    2009-01-14      367.0  1111085350   35.0    27.0  25.0
2          12    2009-01-14      367.0  1600027527   12.0    10.0  10.0
3          13    2009-01-14      367.0  1600027528   31.0    26.0  19.0
4          14    2009-01-14      367.0  1600027564   56.0    48.0  42.0
```

(continues on next page)

(continued from previous page)

```
    SPEND  PRICE  BASE_PRICE  ...  DISPLAY  TPR_ONLY  \
0   26.32   1.88        1.88  ...      0.0       0.0
1   69.30   1.98        1.98  ...      0.0       0.0
2   38.28   3.19        3.19  ...      0.0       0.0
3  142.29   4.59        4.59  ...      0.0       0.0
4  152.32   2.72        3.07  ...      0.0       0.0

                     Desc     Category       Sub-Category SUMPRICE  \
0  PL HONEY NUT TOASTD OATS  COLD CEREAL  ALL FAMILY CEREAL    19.54
1   PL BT SZ FRSTD SHRD WHT  COLD CEREAL  ALL FAMILY CEREAL    19.54
2     GM HONEY NUT CHEERIOS  COLD CEREAL  ALL FAMILY CEREAL    19.54
3               GM CHEERIOS  COLD CEREAL  ALL FAMILY CEREAL    19.54
4               GM CHEERIOS  COLD CEREAL  ALL FAMILY CEREAL    19.54

   COUNTPRICE  AVGPRICE  RELPRICE  PRICE_p2
0           7  2.791429  0.673490    3.5344
1           7  2.791429  0.709314    3.9204
2           7  2.791429  1.142784   10.1761
3           7  2.791429  1.644319   21.0681
4           7  2.791429  0.974411    7.3984

[5 rows x 21 columns]
```

'UPC' stands for Universal Product Code, which can be understood as one SKU in this case and in our SCM terms in general. The code below helps us identify the SKUs by which we want to forecast and their corresponding data size (number of data instances). We can see that the number of instances for each UPC is similar and that there is no UPC with only a few data points. This is important because training a model on a small dataset may limit its generalization.

```
print(salesCereals.groupby('UPC').count())
```

```
            Unnamed: 0  WEEK_END_DATE  STORE_NUM  UNITS  VISITS  HHS  SPEND  \
UPC
1111085319         156            156        156    156     156  156    156
1111085350         156            156        156    156     156  156    156
1600027527         156            156        156    156     156  156    156
1600027528         156            156        156    156     156  156    156
1600027564         155            155        155    155     155  155    155
3000006340         133            133        133    133     133  133    133
3800031829         155            155        155    155     155  155    155

            PRICE  BASE_PRICE  FEATURE  DISPLAY  TPR_ONLY  Desc  Category  \
UPC
1111085319    156         156      156      156       156   156       156
1111085350    156         156      156      156       156   156       156
1600027527    156         156      156      156       156   156       156
1600027528    156         156      156      156       156   156       156
1600027564    155         155      155      155       155   155       155
3000006340    133         133      133      133       133   133       133
3800031829    155         155      155      155       155   155       155

            Sub-Category  SUMPRICE  COUNTPRICE  AVGPRICE  RELPRICE  PRICE_p2
UPC
1111085319           156       156         156       156       156       156
1111085350           156       156         156       156       156       156
1600027527           156       156         156       156       156       156
```

(continues on next page)

(continued from previous page)

```
1600027528          156       156        156       156       156       156
1600027564          155       155        155       155       155       155
3000006340          133       133        133       133       133       133
3800031829          155       155        155       155       155       155
```

2.3.3 Block 2: Feature engineering & preparation

We then organize the data by 'UPC.' The model presented here only runs on a predetermined subset of variables in the data. You can add or remove these explanatory variables based on your judgemental call.

```
feature_list = ['PRICE', 'PRICE_p2', 'FEATURE', 'DISPLAY','TPR_ONLY','RELPRICE']

productList = salesCereals['UPC'].unique()
print(productList)

X, X_train, X_test = {}, {}, {}
y, y_train, y_test, y_pred = {}, {}, {}, {}

for upc in productList:

  X[upc] = salesCereals.loc[salesCereals['UPC']==upc][feature_list]
  y[upc] = salesCereals.loc[salesCereals['UPC']==upc]['UNITS']
  # Split into training and testing data
  X_train[upc], X_test[upc], y_train[upc], y_test[upc] = sklearn.model_selection.
↪train_test_split(X[upc], y[upc], test_size=0.25, random_state=0)
```

```
[1111085319 1111085350 1600027527 1600027528 1600027564 3000006340
 3800031829]
```

2.3.4 Block 3: Model & algorithm (training & testing)

In the next two cells, we train and test two different types of models, namely Linear Regression and Tree Regression. In each cell, we create a loop **for** each UPC on the product list. The first line in each loop is to train the model and the second line is for testing the model's performance on unseen data. The next three lines compute the performance metrics we would like to measure.

We organize the linear regression result by 'UPC' (row) and performance metrics (columns). Then we create a dataframe and put the computed metric in the corresponding column (the last line in each loop).

```
#Linear model
regr = {}
regrSummary = pandas.DataFrame(columns=['trainRMSE', 'testRMSE','testMAE','testMAPE'],
 ↪ index = productList)

for upc in productList:
    regr[upc] = sklearn.linear_model.LinearRegression().fit(X_train[upc],y_train[upc])
    trainRMSE = numpy.sqrt(sklearn.metrics.mean_squared_error(y_train[upc], regr[upc].
↪predict(X_train[upc])))
    y_pred[upc] = regr[upc].predict(X_test[upc])

    testMAE = sklearn.metrics.mean_absolute_error(y_test[upc], y_pred[upc])
```

(continues on next page)

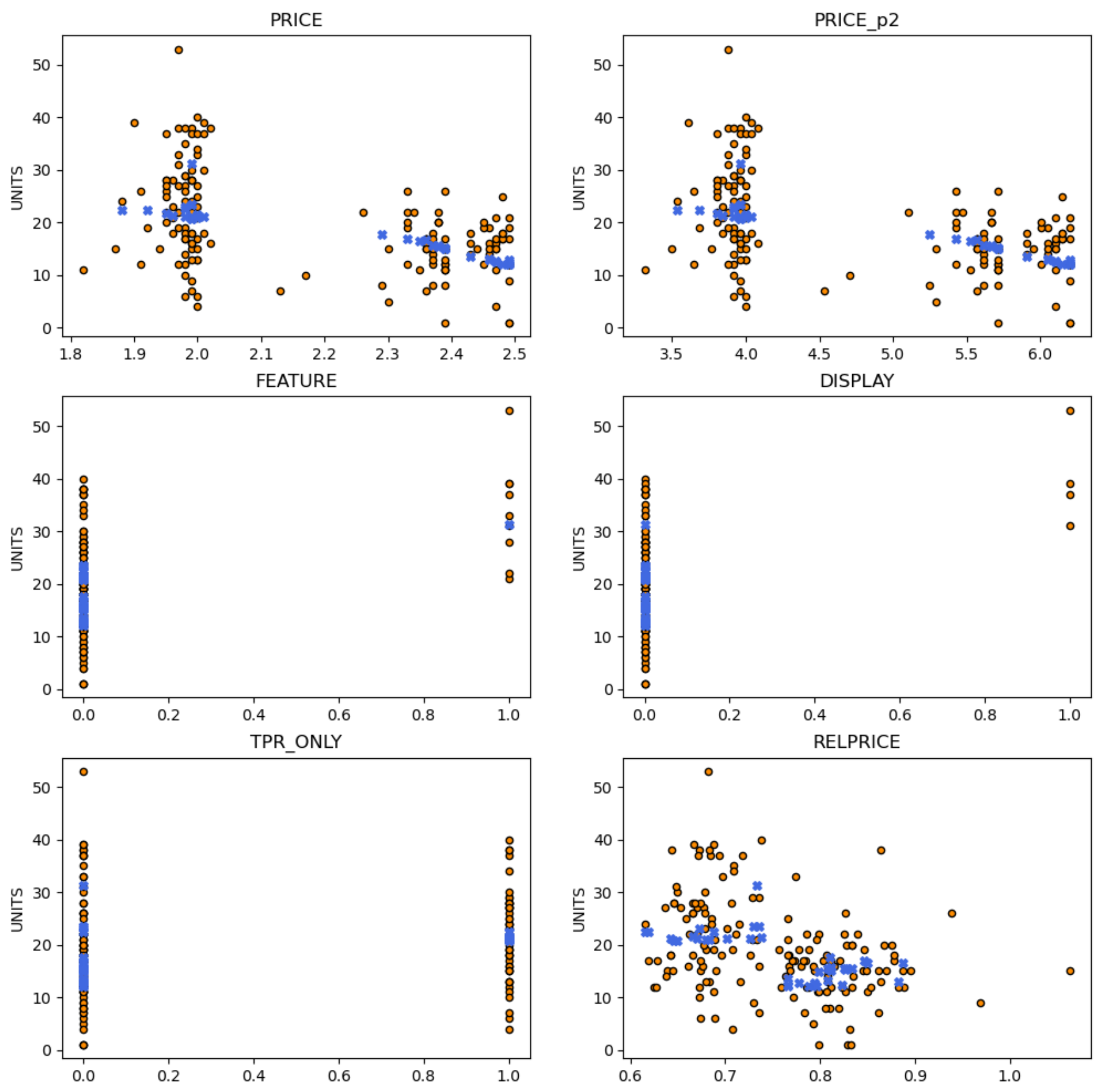
(continued from previous page)

```
    testMAPE = numpy.mean(numpy.abs((y_test[upc] - y_pred[upc]) / y_test[upc]))
    testRMSE = numpy.sqrt(sklearn.metrics.mean_squared_error(y_test[upc], y_
↪pred[upc]))
    regrSummary.loc[upc] =  [trainRMSE, testRMSE, testMAE, testMAPE]

print('Linear regression Summary')
print(regrSummary)
print('average training RMSE:' + str(round(regrSummary['trainRMSE'].mean(),2)))
print('average testing RMSE:' + str(round(regrSummary['testRMSE'].mean(),2)))
print('average testing MAE:' + str(round(regrSummary['testMAE'].mean(),2)))
print('average testing MAPE:' + str(round(regrSummary['testMAPE'].mean(),2)))
```

```
Linear regression Summary
            trainRMSE   testRMSE    testMAE  testMAPE
1111085319   7.712708   8.232999   6.568929  0.838554
1111085350   7.305901   7.696171   6.135274  0.748932
1600027527  15.073315  23.519902  13.546774  0.547776
1600027528   9.073374   14.06519   8.088092  0.251657
1600027564   8.770006   6.796714   5.239537   0.27655
3000006340   4.255693   3.874297   2.886348  0.720298
3800031829   7.650327   8.518524   6.574746  0.379121
average training RMSE:8.55
average testing RMSE:10.39
average testing MAE:7.01
average testing MAPE:0.54
```

Here we visualize the data points and the predictions using the previously defined plot function.

```
# Plot prediction results for a product (UPC)
upc = productList[1]
data_y = salesCereals.loc[salesCereals['UPC']==upc]['UNITS'].values
data_x = salesCereals.loc[salesCereals['UPC']==upc][feature_list].values
plot_data_scatter(data_x, data_y, X_test[upc], y_pred[upc], feature_list)
```



In order to see the impact of the price on the demand, we use a simple plot function below from mathplotlib to see how the demand would change when the price changes.

For more details of the plot function, please see: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html

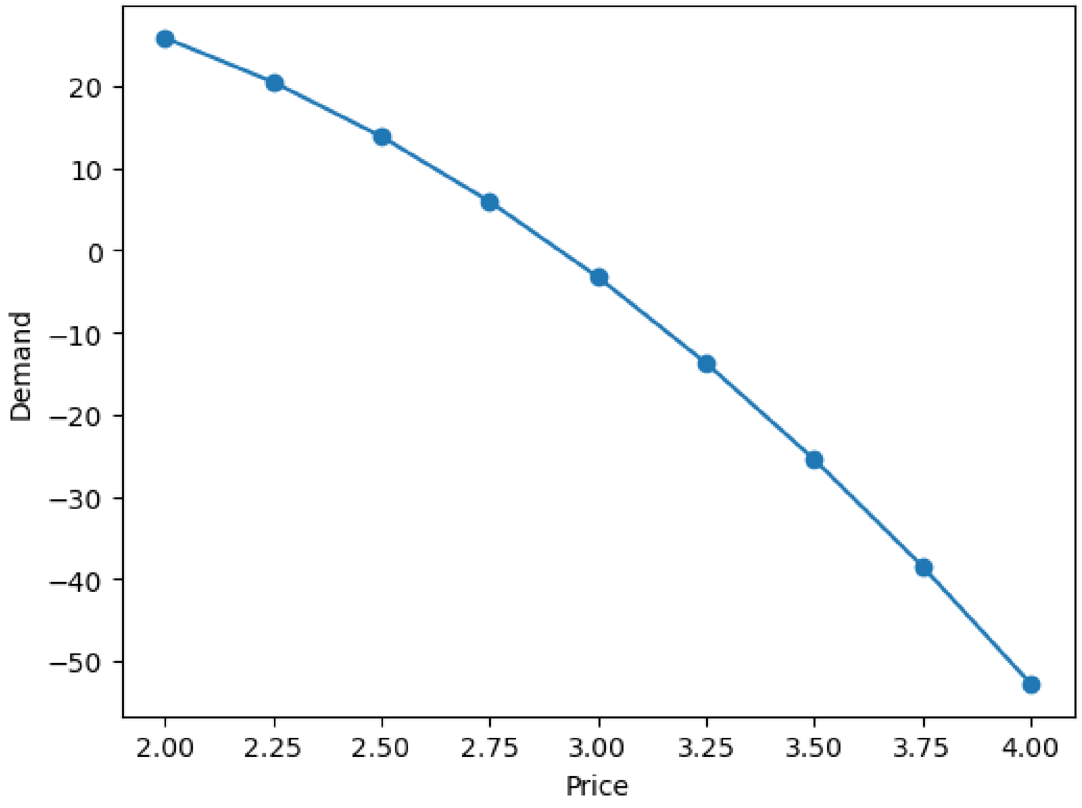
```
upc = productList[1]
input_x = []
prices = [2.0, 2.25, 2.5, 2.75, 3.0, 3.25, 3.5, 3.75, 4.0]

# generate inputs for the plot using simple feature values and varying price points
for p in prices:
  input_x.append([p, p**2, 0,0,0, 1.0])

# obtain the predicted demands
predict_y = regr[upc].predict(input_x)
plt.plot(prices, predict_y, marker='o')
plt.xlabel('Price')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Demand')
plt.show()
```

```
D:\ProgramData\Anaconda3\envs\sca_book2\lib\site-packages\sklearn\base.py:464:␣
↪UserWarning: X does not have valid feature names, but LinearRegression was␣
↪fitted with feature names
  warnings.warn(
```



Likewise, we obtain the tree regression results by simply changing the function name and the result table name.

```
#Tree models
regr = {}
regrSummary = pandas.DataFrame(columns=['trainRMSE', 'testRMSE','testMAE','testMAPE'],
↪ index = productList)

for upc in productList:

    regr[upc] = sklearn.tree.DecisionTreeRegressor(random_state = 0).fit(X_train[upc],
↪y_train[upc]) # standard regression tree
    # regr[upc] = sklearn.ensemble.RandomForestRegressor(random_state = 0).fit(X_
↪train[upc],y_train[upc]) # random forest tree
    trainRMSE = numpy.sqrt(sklearn.metrics.mean_squared_error(y_train[upc], regr[upc].
↪predict(X_train[upc])))
    y_pred[upc] = regr[upc].predict(X_test[upc])

    testMAE = sklearn.metrics.mean_absolute_error(y_test[upc], y_pred[upc])
```

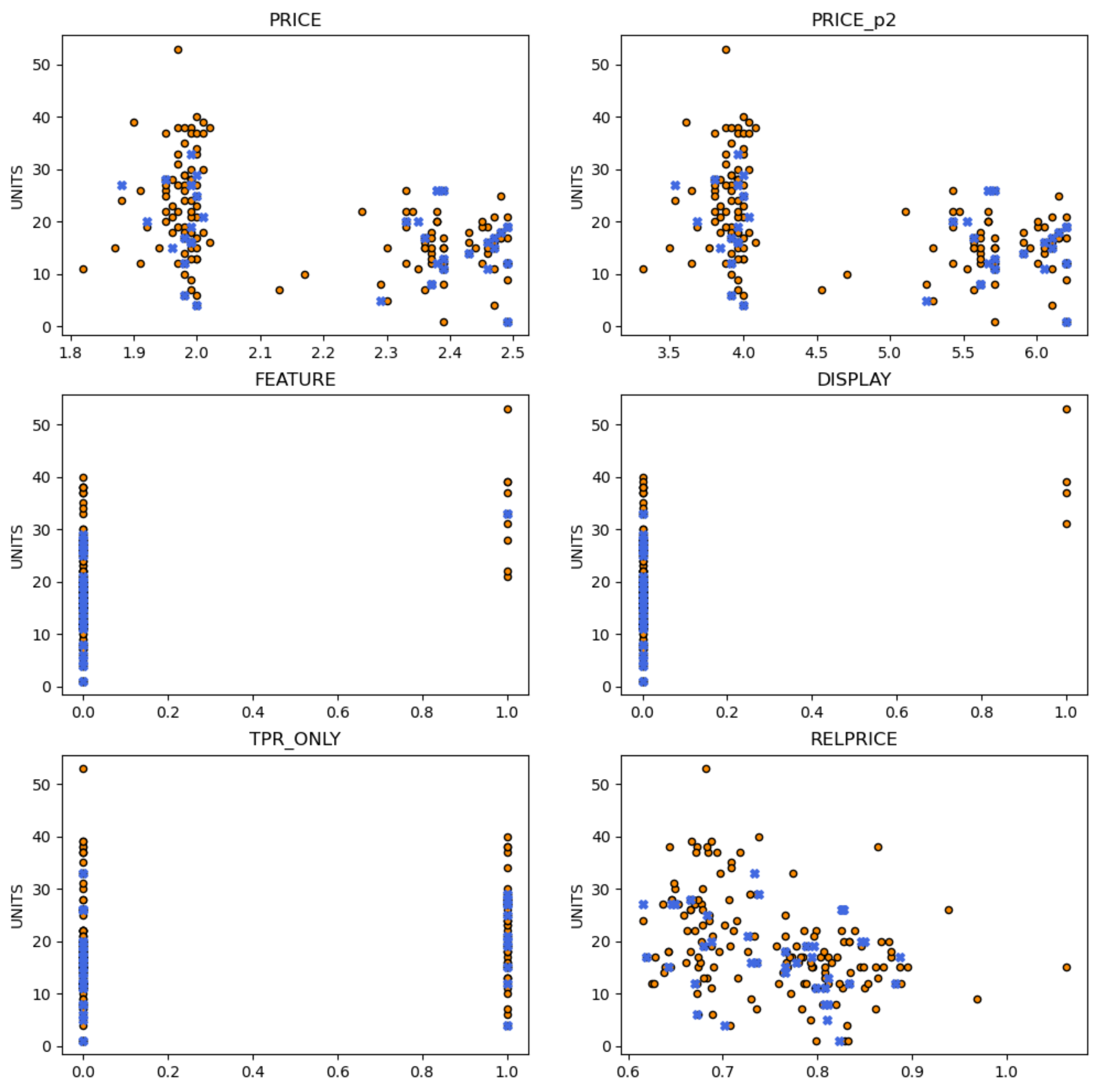
(continues on next page)

(continued from previous page)

```
    testMAPE = numpy.mean(numpy.abs((y_test[upc] - y_pred[upc]) / y_test[upc]))
    testRMSE = numpy.sqrt(sklearn.metrics.mean_squared_error(y_test[upc], y_
↪pred[upc]))
    regrSummary.loc[upc] =  [trainRMSE, testRMSE, testMAE, testMAPE]

print('Regression Tree Summary')
print(regrSummary)
print('average training RMSE:' + str(round(regrSummary['trainRMSE'].mean(),2)))
print('average testing RMSE:' + str(round(regrSummary['testRMSE'].mean(),2)))
print('average testing MAE:' + str(round(regrSummary['testMAE'].mean(),2)))
print('average testing MAPE:' + str(round(regrSummary['testMAPE'].mean(),2)))
```

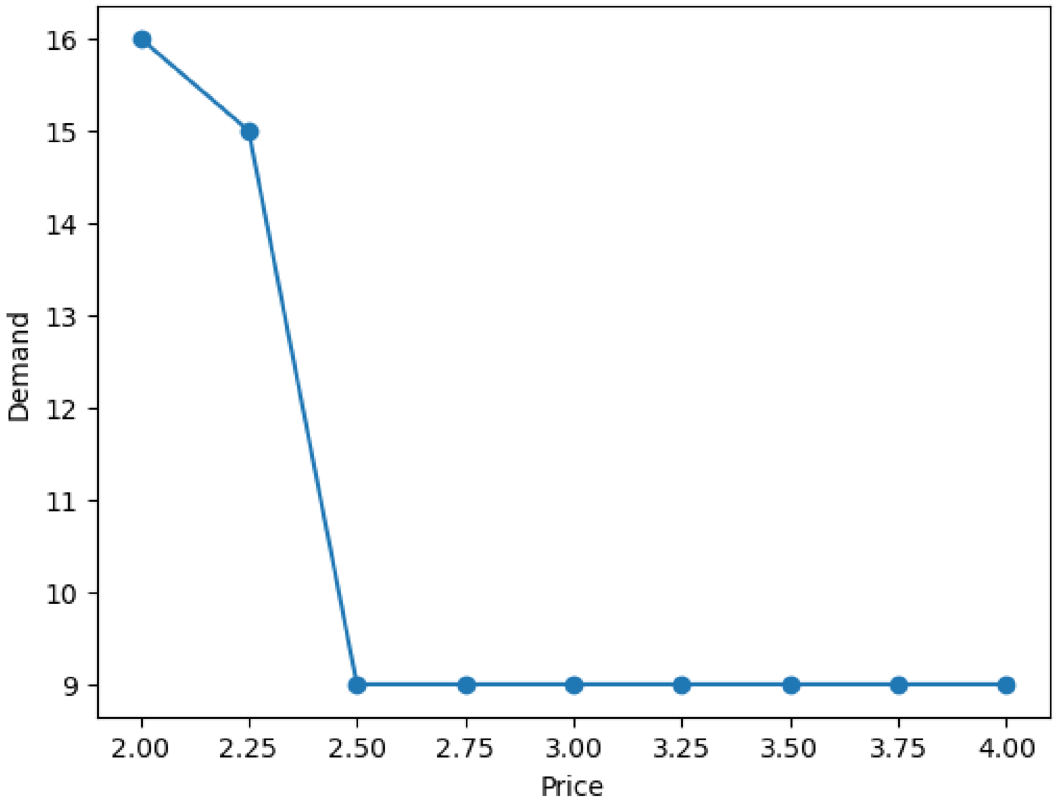
```
Regression Tree Summary
           trainRMSE   testRMSE    testMAE  testMAPE
1111085319  2.038099  10.759611   7.102564  0.841491
1111085350       0.0   7.471313   5.923077  0.625815
1600027527  1.147275  30.330234  15.564103  0.520292
1600027528   2.65623  13.221001   8.641026  0.286716
1600027564       0.0  10.453806   7.897436   0.38854
3000006340  0.514111   7.762087        4.5  0.992923
3800031829  1.543898   9.131518   7.487179  0.421318
average training RMSE:1.13
average testing RMSE:12.73
average testing MAE:8.16
average testing MAPE:0.58
```

```
# Plot prediction results for a product (UPC)
upc = productList[1]
data_y = salesCereals.loc[salesCereals['UPC']==upc]['UNITS'].values
data_x = salesCereals.loc[salesCereals['UPC']==upc][feature_list].values
plot_data_scatter(data_x, data_y, X_test[upc], y_pred[upc], feature_list)
```



```
upc = productList[1]
input_x = []
prices = [2.0, 2.25, 2.5, 2.75, 3.0, 3.25, 3.5, 3.75, 4.0]

# generate inputs for the plot using simple feature values and varying price points
for p in prices:
  input_x.append([p, p**2, 0,0,0, 1.0])

# obtain the predicted demands
predict_y = regr[upc].predict(input_x)
plt.plot(prices, predict_y, marker='o')
plt.xlabel('Price')
plt.ylabel('Demand')
plt.show()
```

```
D:\ProgramData\Anaconda3\envs\sca_book2\lib\site-packages\sklearn\base.py:464:␣
↪UserWarning: X does not have valid feature names, but DecisionTreeRegressor was␣
↪fitted with feature names
  warnings.warn(
```



2.3.5 Block 4: Model selection

By comparing the average result, we can see that the linear regression model generally outperformed the decision tree regression and did not overfit the data. Therefore, we proceed with the linear regression model for the whole dataset by replacing 'X_train' with 'X'. Given that the model has 'seen' the whole dataset, its forecast errors normally decrease. Therefore, we will save the trained model and use it for the new data which will be used in the optimization models in the next session.

```
# Best model
regr = {}
regrSummary = pandas.DataFrame(columns=['totalMAE','totalMAPE', 'totalRMSE'], index =␣
↪productList)

for upc in productList:
    regr[upc] = sklearn.linear_model.LinearRegression().fit(X[upc],y[upc])
    y_pred[upc] = regr[upc].predict(X[upc])
    testMAE = sklearn.metrics.mean_absolute_error(y[upc], y_pred[upc])
    testMAPE = numpy.mean(numpy.abs((y[upc] - y_pred[upc]) / y[upc]))
    testRMSE = numpy.sqrt(sklearn.metrics.mean_squared_error(y[upc], y_pred[upc]))
    regrSummary.loc[upc] =  [testMAE, testMAPE, testRMSE]
```

(continues on next page)

(continued from previous page)

```
print('Best Model Summary')
print(regrSummary)
print('average overall MAE:' + str(round(regrSummary['totalMAE'].mean(),2)))
print('average overall MAPE:' + str(round(regrSummary['totalMAPE'].mean(),2)))
print('average overall RMSE:' + str(round(regrSummary['totalRMSE'].mean(),2)))
```

```
Best Model Summary
            totalMAE totalMAPE  totalRMSE
1111085319  6.081658  0.643456   7.756732
1111085350  5.804481      0.64   7.312385
1600027527  9.773413  0.611309  16.926391
1600027528   7.05949  0.302731  10.426074
1600027564  5.990113  0.296792   8.303061
3000006340  2.865248  0.699564   4.119918
3800031829  6.164282  0.336052   7.810234
average overall MAE:6.25
average overall MAPE:0.5
average overall RMSE:8.95
```

2.3.6 Save trained models

If you use Jupyter, you can save it to a local folder. The code below will put it in the current folder.

```
cwd = './'
```

Now we can save the files to the folder indicated by using the code below.

```
import pickle
# save the models to the drive
for upc in productList:
    filename = cwd+str(upc)+'_demand_model.sav'
    # save the model to disk
    pickle.dump(regr[upc], open(filename, 'wb'))
```

2.4 Module 1B: Predicting demands from trained models for decision models

Link to Google Colab of this Notebook

In Module 1A, we designed our model on a predetermined subset of regressor variables and trained it by UPC. Now in this notebook, we will prepare our inputs for the optimization model by predicting the demands based on different price points.

```
import pandas
import sklearn
from sklearn import *
```

2.4.1 1. Data input

We have prepared the input files which contain the features to be predicted. The first file shows a small dataset whereas the second file consists of a large dataset, i.e.,

1. **'InputFeatures_Prob1.csv'**. This is a small scale problem. The output of this will be used in the optimization model which you will see in Modules 2A (explicit model) and 2B (compact model).
2. **'InputFeatures_Prob2.csv'**. This is a large-scale problem. This one contains a much higher number of variables and constraints to reflect real-life setting. We will use the output of this in the Module 2B.

In order to read the input, we provide two options here. Please run only either option 1 or option 2 (***not both***).

Option 1: download from the URLs. You can you can get it directly from the URLs as usual using the codes below to download 'InputFeatures_Prob1.csv' and save it in DataFrame

```
# small example
url = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/S08_09_Retail_
↪Analytics/InputFeatures_Prob1.csv'
# large example, please outcomment if you want to try
# url = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/S08_09_
↪Retail_Analytics/InputFeatures_Prob2.csv'

predDemand = pandas.read_csv(url)

# Dataset is now stored in a Pandas Dataframe predDemand
predDemand
```

```
    Unnamed: 0  avgPriceChoice         UPC  PRICE  PRICE_p2  FEATURE  DISPLAY  \
0            0             3.0  1600027528    2.5      6.25        0        0
1            1             3.0  1600027528    3.0      9.00        0        0
2            2             3.0  1600027528    3.5     12.25        0        0
3            3             3.0  1600027564    2.5      6.25        0        0
4            4             3.0  1600027564    3.0      9.00        0        0
5            5             3.0  1600027564    3.5     12.25        0        0
6            6             3.0  3000006340    2.5      6.25        0        0
7            7             3.0  3000006340    3.0      9.00        0        0
8            8             3.0  3000006340    3.5     12.25        0        0
9            9             3.0  3800031829    2.5      6.25        0        0
10          10             3.0  3800031829    3.0      9.00        0        0
11          11             3.0  3800031829    3.5     12.25        0        0

    TPR_ONLY  RELPRICE
0          0  0.833333
1          0  1.000000
2          0  1.166667
3          0  0.833333
4          0  1.000000
5          0  1.166667
6          0  0.833333
7          0  1.000000
8          0  1.166667
9          0  0.833333
10         0  1.000000
11         0  1.166667
```

2.4.2 2. Model retrieval

Next, we retrieve the best model that we previously trained and saved from the current working directory (cmd) based on one of the two two options below.

If you model is saved on PC, you need to give the path to the saved models. Here we assume that it is located in the same folder as the notebook.

```
cwd = './'
```

Following the block above, we can now load the model that we previously trained and saved for each UPC.

```
import pickle

productList = predDemand['UPC'].unique()

regr = {}
for upc in productList:
    filename = cwd+str(upc)+'_demand_model.sav'
    print(upc)
    # load the model to disk
    regr[upc] = pickle.load(open(filename, 'rb'))
```

```
1600027528
1600027564
3000006340
3800031829
```

2.4.3 3. Demand forecasting

In this cell, we also create a loop **for** each UPC. Here are the descriptions of each line in the for loop

- The first line in the **for** loop loads the data on the explanatory variables (features) for each UPC.
- The second line retrives the UPC value so that we can call and run the model for that UPC.
- The third line takes the model object for the current UPC (*regr[upc]*) and predicts the demand. We also use the function *clip(0.0)* to make sure that the demand is non-negative (which is possible since the demand is a decreasing function of price and the regression function is unbounded) and function *round(1)* to round the predicted value to one digit.
- The fourth line put the predicted demand into the series which will be added as a new column

Once the for loop terminated, we add a new column *'predictSales'* which shows the predicted demand.

```
feature_list = ['PRICE', 'PRICE_p2', 'FEATURE', 'DISPLAY','TPR_ONLY','RELPRICE']

X = {}
y_pred = {}

# prepare blank series which will be added as a new column to the DataFrame predDemand
predictedValueSeries = pandas.Series()

for upc in productList:
  # Line 1 of for loop: load the data on the explanatory variable
  X[upc] = predDemand.loc[predDemand['UPC']==upc][feature_list]
```

(continues on next page)

(continued from previous page)

```
    # Line 2: retrieve the UPC value
    upcIndex = predDemand.loc[predDemand['UPC']==upc].index

    # Line 3: predice the demands and make sure the demand is non-negative
    y_pred[upc] = regr[upc].predict(X[upc]).clip(0.0).round(1)

    # Line 4: add the predicted demand to the series
    predictedValueSeries = predictedValueSeries._append(pandas.Series(y_pred[upc],␣
↪index = upcIndex))

predDemand['predictSales'] = predictedValueSeries
print(predDemand.to_string())
```

```
    Unnamed: 0  avgPriceChoice         UPC  PRICE  PRICE_p2  FEATURE  DISPLAY  TPR_
↪ONLY  RELPRICE  predictSales
0            0             3.0  1600027528    2.5      6.25        0        0     ␣
↪   0  0.833333          94.9
1            1             3.0  1600027528    3.0      9.00        0        0     ␣
↪   0  1.000000          67.0
2            2             3.0  1600027528    3.5     12.25        0        0     ␣
↪   0  1.166667          46.4
3            3             3.0  1600027564    2.5      6.25        0        0     ␣
↪   0  0.833333          24.1
4            4             3.0  1600027564    3.0      9.00        0        0     ␣
↪   0  1.000000          22.6
5            5             3.0  1600027564    3.5     12.25        0        0     ␣
↪   0  1.166667          19.8
6            6             3.0  3000006340    2.5      6.25        0        0     ␣
↪   0  0.833333           6.2
7            7             3.0  3000006340    3.0      9.00        0        0     ␣
↪   0  1.000000           4.0
8            8             3.0  3000006340    3.5     12.25        0        0     ␣
↪   0  1.166667           3.0
9            9             3.0  3800031829    2.5      6.25        0        0     ␣
↪   0  0.833333          32.9
10          10             3.0  3800031829    3.0      9.00        0        0     ␣
↪   0  1.000000          24.3
11          11             3.0  3800031829    3.5     12.25        0        0     ␣
↪   0  1.166667          20.4
```

Now we save the predicted sales into csv file to be used in the optimization model later on.

```
# Please save it as 'predictedSales_Prob1.csv' if 'InputFeatures_Prob1.csv' is used
# Otherwise, please save it as 'predictedSales_Prob2.csv' if 'InputFeatures_Prob2.csv
↪' is used
predDemand.to_csv(cwd +'predictedSales_Prob1.csv')
```

2.5 Module 2A: Retail price optimization - Simple Explicit Model

Link to Google Colab of this Notebook

In this part, we begin by installing the (i) Pyomo package and (ii) the linear programming solver GLPK (GNU Linear Programming Kit). Installing Pyomo and its utils in Colab is straightforward as shown below. If you wish to install them in Anaconda or in a different distribution package, please consult this page http://www.pyomo.org/installation.

Pyomo is a modeling language which can be used in conjunction with a number of solvers. For more information on Pyomo, you can also review: http://www.pyomo.org/documentation

The following block will take some time to process. Please wait until all the tools are sucessfully installed. This option on Colab is recommended but if you want to use an office notebook version, you can review the steps here: https://nbviewer.jupyter.org/github/jckantor/ND-Pyomo-Cookbook/blob/master/notebooks/01.01-Installing-Pyomo.ipynb#Step-3.-Install-solvers

```
# Install Pyomo and GLPK in your Python environment first
pip install -q pyomo
conda install conda-forge::glpk
```

2.5.1 Blocks 1 & 2: Data input & input parameters

We have been working on this dataset during the past few weeks. The product list below is just simply a subset of those products. In this session, we would like to work on price optimization for the items on that list. In particular, each product can be priced at 2.5, 3.0 or 3.5 dollar and we wish to decide the prices at which these products are offered so that our total revenue is maximized. As we have learned from last week's analysis that the relative price of competing/substitution products is a factor affecting sales, we must optimize the pricing of all these products together to reach the global optimum.

We need to prepare lists of index for products (*iIndexList*) and price options (*jIndexList*) to be used in the optimization model.

```
from pyomo.environ import *

productList = ['1600027528', '1600027564', '3000006340', '3800031829']
priceList = [2.5, 3.0, 3.5]
avgPriceValue = 3.0

iIndexList = list(range(len(productList)))
jIndexList = list(range(len(priceList)))
```

2.5.2 Block 3: Create an optimization model

Now we will create an optimization model for the prescriptive pricing model of Rue La La. An optimization model consists of (i) decision variables, (ii) objective function, and (iii) constraints. You can review this page [link] for a simple example of an optimization model using Pyomo.

Block 3.1: Variable declarations

We use *model.x* to define the decision variable of our optimization model. We will a binary variable x_{ij} which takes either the value 1 or 0 (yes/no). In other words, $x_{ij} \in \{0, 1\}$ with i being the product index and j being the price index. We can formally define this variable as:

- x_{ij} equals 1 if the price choice j is chosen for product i, 0 otherwise.

Please note that in Python (and many other programming languages), the starting index is 0. By setting $x_{01} = 1$, we sell product '1600027528' (product index 0 in the product list) at price $3.0 (price index 1 on the price list).

In this block of codes, we create an object of the model (using the *ConcreteModel* class) and declare the variable x (using *Var(iIndexList, jIndexList, within = Binary)*). The line *model.pprint()* print out the details of the entire model in the object we created. Since we only created the variables, we only see the variables in here without other components. You can visit this page [Link] for more details on the ConcreteModel class.

```
model = ConcreteModel()
# Variables
model.x = Var(iIndexList, jIndexList, within = Binary)
model.pprint()
```

```
3 Set Declarations
    x_index : Size=1, Index=None, Ordered=True
        Key  : Dimen : Domain              : Size : Members
        None :     2 : x_index_0*x_index_1 :   12 : {(0, 0), (0, 1), (0, 2), (1,␣
↪0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)}
    x_index_0 : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    4 : {0, 1, 2, 3}
    x_index_1 : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    3 : {0, 1, 2}

1 Var Declarations
    x : Size=12, Index=x_index
        Key    : Lower : Value : Upper : Fixed : Stale : Domain
        (0, 0) :     0 :  None :     1 : False :  True : Binary
        (0, 1) :     0 :  None :     1 : False :  True : Binary
        (0, 2) :     0 :  None :     1 : False :  True : Binary
        (1, 0) :     0 :  None :     1 : False :  True : Binary
        (1, 1) :     0 :  None :     1 : False :  True : Binary
        (1, 2) :     0 :  None :     1 : False :  True : Binary
        (2, 0) :     0 :  None :     1 : False :  True : Binary
        (2, 1) :     0 :  None :     1 : False :  True : Binary
        (2, 2) :     0 :  None :     1 : False :  True : Binary
        (3, 0) :     0 :  None :     1 : False :  True : Binary
        (3, 1) :     0 :  None :     1 : False :  True : Binary
        (3, 2) :     0 :  None :     1 : False :  True : Binary

4 Declarations: x_index_0 x_index_1 x_index x
```

Block 3.2: Adding an objective function

The general form of the objective function is $\sum_{i=0}^{3}\sum_{j=0}^{2} p_j \cdot \tilde{D}_{ijk} \cdot x_{ij}$, where p_j is the jth price of each product, e.g. $p_{j=0}$ denotes the price at position 1 on the price list of a product (2.5 dollars in this case). Looking at the price list, we can easily see that $p_{j=1} = 3.0$ dollars. As we defined above, $x_{ij} = 1$ when product i is sold at price j, 0 otherwise. $\tilde{D}_{ijk}$ is the predicted sales of product i when this product is sold at price j while the average price of all competing products, including product i, is equal to k. In our optimization model, k is preset to 3.0 (avgPriceValue). By inputting price p_j and average price k into the predictive model we trained and saved last week, we can attain the corresponding predicted sales $\left(\tilde{D}_{ijk}\right)$. We use Objective($\cdot$) to define the objective function and 'sense = maximize' to indicate that the objective is for maximization.

```
# Objective function

model.OBJ = Objective(sense = maximize, expr = 2.5*95.0*model.x[0,0] + 3.0*67.0*model.
↪x[0,1] + 3.5*46.0*model.x[0,2]
                       + 2.5*24.0*model.x[1,0] + 3.0*23.0*model.x[1,1] + 3.5*20.
↪0*model.x[1,2]
                       + 2.5*6.0*model.x[2,0] + 3.0*4.0*model.x[2,1] + 3.5*3.0*model.
↪x[2,2]
                       + 2.5*33.0*model.x[3,0] + 3.0*24.0*model.x[3,1] + 3.5*20.
↪0*model.x[3,2])
```

2.5.3 Block 3.2: Adding constraints

Constraint 1: One price choice must be selected for each product

As regards the first set of constraints, we wish to make sure that each product is sold at one price only. Therefore, the general form of this constraint set is $\sum_{j=0}^{2} x_{ij} = 1, \forall i \in \{0, 1, 2, 3\}$. We use Constraint($\cdot$) to define the constraints.

```
# Constraints #1
model.PriceChoiceUPC1 = Constraint(expr = model.x[0,0] + model.x[0,1] + model.x[0,2]␣
↪== 1)
model.PriceChoiceUPC2 = Constraint(expr = model.x[1,0] + model.x[1,1] + model.x[1,2]␣
↪== 1)
model.PriceChoiceUPC3 = Constraint(expr = model.x[2,0] + model.x[2,1] + model.x[2,2]␣
↪== 1)
model.PriceChoiceUPC4 = Constraint(expr = model.x[3,0] + model.x[3,1] + model.x[3,2]␣
↪== 1)
```

Constraint 2: The sum of the prices of all products must equal k

The second set of constraints ensures that the average price of all the **4** products considered in our optimization model equals the predefined average price, which is $k = \$3.0 \times 4$ (avgPriceValue x no. of products) in our model. The general form is

$$\frac{\sum_{i=0}^{3}\sum_{j=0}^{2} p_j \cdot x_{ij}}{4} = k \iff \sum_{i=0}^{3}\sum_{j=0}^{2} p_j \cdot x_{ij} = k \cdot 4$$

This can be done using the following block.

```
# Constraints #2

model.sumPrice = Constraint(expr = 2.5*model.x[0,0] + 3.0*model.x[0,1] + 3.5*model.
↪x[0,2]
                       + 2.5*model.x[1,0] + 3.0*model.x[1,1] + 3.5*model.x[1,2]
```

(continues on next page)

(continued from previous page)

```
                         + 2.5*model.x[2,0] + 3.0*model.x[2,1] + 3.5*model.x[2,2]
                         + 2.5*model.x[3,0] + 3.0*model.x[3,1] + 3.5*model.x[3,2] ==␣
↪avgPriceValue*4)
```

Now we can print the model again to see all the components that were created.

```
model.pprint()
```

```
3 Set Declarations
    x_index : Size=1, Index=None, Ordered=True
        Key  : Dimen : Domain              : Size : Members
        None :     2 : x_index_0*x_index_1 :   12 : {(0, 0), (0, 1), (0, 2), (1,␣
↪0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)}
    x_index_0 : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    4 : {0, 1, 2, 3}
    x_index_1 : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    3 : {0, 1, 2}

1 Var Declarations
    x : Size=12, Index=x_index
        Key    : Lower : Value : Upper : Fixed : Stale : Domain
        (0, 0) :     0 :  None :     1 : False :  True : Binary
        (0, 1) :     0 :  None :     1 : False :  True : Binary
        (0, 2) :     0 :  None :     1 : False :  True : Binary
        (1, 0) :     0 :  None :     1 : False :  True : Binary
        (1, 1) :     0 :  None :     1 : False :  True : Binary
        (1, 2) :     0 :  None :     1 : False :  True : Binary
        (2, 0) :     0 :  None :     1 : False :  True : Binary
        (2, 1) :     0 :  None :     1 : False :  True : Binary
        (2, 2) :     0 :  None :     1 : False :  True : Binary
        (3, 0) :     0 :  None :     1 : False :  True : Binary
        (3, 1) :     0 :  None :     1 : False :  True : Binary
        (3, 2) :     0 :  None :     1 : False :  True : Binary

1 Objective Declarations
    OBJ : Size=1, Index=None, Active=True
        Key  : Active : Sense    : Expression
        None :   True : maximize : 237.5*x[0,0] + 201.0*x[0,1] + 161.0*x[0,2] + 60.
↪0*x[1,0] + 69.0*x[1,1] + 70.0*x[1,2] + 15.0*x[2,0] + 12.0*x[2,1] + 10.5*x[2,2] +␣
↪82.5*x[3,0] + 72.0*x[3,1] + 70.0*x[3,2]

5 Constraint Declarations
    PriceChoiceUPC1 : Size=1, Index=None, Active=True
        Key  : Lower : Body                     : Upper : Active
        None :   1.0 : x[0,0] + x[0,1] + x[0,2] :   1.0 :   True
    PriceChoiceUPC2 : Size=1, Index=None, Active=True
        Key  : Lower : Body                     : Upper : Active
        None :   1.0 : x[1,0] + x[1,1] + x[1,2] :   1.0 :   True
    PriceChoiceUPC3 : Size=1, Index=None, Active=True
        Key  : Lower : Body                     : Upper : Active
        None :   1.0 : x[2,0] + x[2,1] + x[2,2] :   1.0 :   True
    PriceChoiceUPC4 : Size=1, Index=None, Active=True
        Key  : Lower : Body                     : Upper : Active
```

(continues on next page)

(continued from previous page)

```
        None :   1.0 : x[3,0] + x[3,1] + x[3,2] :   1.0 :   True
    sumPrice : Size=1, Index=None, Active=True
        Key  : Lower : Body                                                                     ␣
↪                                                                                               ␣
↪              : Upper : Active
        None :  12.0 : 2.5*x[0,0] + 3.0*x[0,1] + 3.5*x[0,2] + 2.5*x[1,0] + 3.0*x[1,
↪1] + 3.5*x[1,2] + 2.5*x[2,0] + 3.0*x[2,1] + 3.5*x[2,2] + 2.5*x[3,0] + 3.0*x[3,1]␣
↪+ 3.5*x[3,2] :  12.0 :   True

10 Declarations: x_index_0 x_index_1 x_index x OBJ PriceChoiceUPC1 PriceChoiceUPC2␣
↪PriceChoiceUPC3 PriceChoiceUPC4 sumPrice
```

2.5.4 Block 4: Solution and results

Finally, we call for the solver and obtain the solution. The first line indicates which solver we want to use and the second line solves the model (this is equivalent to *fit()* in sklearn).

```
# Solve the model
opt = SolverFactory('glpk')
opt.solve(model)
```

```
{'Problem': [{'Name': 'unknown', 'Lower bound': 400.5, 'Upper bound': 400.5,
↪'Number of objectives': 1, 'Number of constraints': 6, 'Number of variables': 13,
↪ 'Number of nonzeros': 25, 'Sense': 'maximize'}], 'Solver': [{'Status': 'ok',
↪'Termination condition': 'optimal', 'Statistics': {'Branch and bound': {'Number␣
↪of bounded subproblems': '1', 'Number of created subproblems': '1'}}, 'Error rc
↪': 0, 'Time': 0.0847010612487793}], 'Solution': [OrderedDict([('number of␣
↪solutions', 0), ('number of solutions displayed', 0)])]}
```

Now you can print out the solution to review using the function *model.display()* below. We can see that $x_{00} = 1, x_{12} = 1, x_{22} = 1, x_{30} = 1$ and the optimal objective value is $400.5. In other words, we achieve the optimal revenue of $400.5 when product '1600027528' is sold at price $2.5, products '1600027564' and '3000006340' both at price $3.5 and product '3800031829' at price $2.5. We can easily double-check that all the constraints are satisfied as shown in the results below.

```
model.display()
```

```
Model unknown

  Variables:
    x : Size=12, Index=x_index
        Key    : Lower : Value : Upper : Fixed : Stale : Domain
        (0, 0) :     0 :   1.0 :     1 : False : False : Binary
        (0, 1) :     0 :   0.0 :     1 : False : False : Binary
        (0, 2) :     0 :   0.0 :     1 : False : False : Binary
        (1, 0) :     0 :   0.0 :     1 : False : False : Binary
        (1, 1) :     0 :   0.0 :     1 : False : False : Binary
        (1, 2) :     0 :   1.0 :     1 : False : False : Binary
        (2, 0) :     0 :   0.0 :     1 : False : False : Binary
        (2, 1) :     0 :   0.0 :     1 : False : False : Binary
        (2, 2) :     0 :   1.0 :     1 : False : False : Binary
        (3, 0) :     0 :   1.0 :     1 : False : False : Binary
        (3, 1) :     0 :   0.0 :     1 : False : False : Binary
```

(continues on next page)

(continued from previous page)

```
        (3, 2) :     0 :   0.0 :     1 : False : False : Binary

Objectives:
  OBJ : Size=1, Index=None, Active=True
      Key  : Active : Value
      None :   True : 400.5

Constraints:
  PriceChoiceUPC1 : Size=1
      Key  : Lower : Body : Upper
      None :   1.0 :  1.0 :   1.0
  PriceChoiceUPC2 : Size=1
      Key  : Lower : Body : Upper
      None :   1.0 :  1.0 :   1.0
  PriceChoiceUPC3 : Size=1
      Key  : Lower : Body : Upper
      None :   1.0 :  1.0 :   1.0
  PriceChoiceUPC4 : Size=1
      Key  : Lower : Body : Upper
      None :   1.0 :  1.0 :   1.0
  sumPrice : Size=1
      Key  : Lower : Body : Upper
      None :  12.0 : 12.0 :  12.0
```

2.6 Module 2B: Retail price optimization - Automated Implicit Model

Link to Google Colab of this Notebook

This notebook is the script version of the Module2A (explicit model). Unlike the explicit model in which we need to explicitly add each complete equation one by one, we can automate the model generation process by using a script version of it. For this one, we do not expect that you understand in detail how to generate the script but simply understand what each block does. Creating the script would require some experience. The main purpose of this is to provide an example of real-life mathematical programming workflow which automate the prescriptive analytics process.

More particularly, we want to create an **implicit (or compact) model** of the following prescriptive pricing model of Rue La La.

The following blocks install Pyomo and solver. We also provide an option to use a more powerful solver *CBC* in addition to GLPK we used earlier. You can outcomment it if you want to switch to CBC.

```
# Install Pyomo and GLPK in your Python environment first
pip install -q pyomo
conda install conda-forge::glpk
```

2.6.1 Block 1: Data input

We prepared the data inputs in two files, i.e.,

1. **'predictedSales_Prob1.csv'**. This is a small scale problem. It is identical to the problem you see in the Module 2A (explicit model).
2. **'predictedSales_Prob2.csv'**. This is a large-scale problem. This one contains a much higher number of variables and constraints to reflect real-life setting.

Please mainly focus on the file *'predictedSales_Prob1.csv'* since you will get to see the same model as Module_1A. You can also try *'predictedSales_Prob2.csv'* if you are interested to see the large-scale model.

In order to read the input, we take the file from the URL. This is the same file that you would obtain if you run the module 1B. If you want, you can change this block so that you can upload it from your PC or load it from Google Drive (see also Module 1B how these two options can be done).

```
import pandas

# Prob1 is the same problem as Module 2A
url = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/S08_09_Retail_
↪Analytics/predictedSales_Prob1.csv'

# Prob2 is the large-scale problem
# url = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/S08_09_
↪Retail_Analytics/predictedSales_Prob2.csv'

predDemand = pandas.read_csv(url)

# Dataset is now stored in a Pandas Dataframe predDemand
predDemand
```

```
    NA  avgPriceChoice         UPC  PRICE  PRICE_p2  FEATURE  DISPLAY  \
0    0             3.0  1600027528    2.5      6.25        0        0
1    1             3.0  1600027528    3.0      9.00        0        0
2    2             3.0  1600027528    3.5     12.25        0        0
3    3             3.0  1600027564    2.5      6.25        0        0
4    4             3.0  1600027564    3.0      9.00        0        0
5    5             3.0  1600027564    3.5     12.25        0        0
6    6             3.0  3000006340    2.5      6.25        0        0
7    7             3.0  3000006340    3.0      9.00        0        0
8    8             3.0  3000006340    3.5     12.25        0        0
9    9             3.0  3800031829    2.5      6.25        0        0
10  10             3.0  3800031829    3.0      9.00        0        0
11  11             3.0  3800031829    3.5     12.25        0        0

    TPR_ONLY  RELPRICE  predictSales
0          0  0.833333          95.0
1          0  1.000000          67.0
2          0  1.166667          46.0
3          0  0.833333          24.0
4          0  1.000000          23.0
5          0  1.166667          20.0
6          0  0.833333           6.0
7          0  1.000000           4.0
8          0  1.166667           3.0
9          0  0.833333          33.0
```

(continues on next page)

(continued from previous page)

```
10          0  1.000000          24.0
11          0  1.166667          20.0
```

With the new dataset, we first need to check how many average price values are there because we need to run the optimization model for each value of the average price.

```
avgPriceList = predDemand['avgPriceChoice'].unique()
inputColumns = ['avgPriceChoice', 'UPC', 'PRICE','predictSales']
print("Possible average price choices (k/N.Product):"+str(avgPriceList))
```

```
Possible average price choices (k/N.Product):[3.]
```

2.6.2 Block 2: Prepare input parameters for the model

We can choose which value of k we want to use in the optimization model from the *avgPriceChoice* we have in the dataset. In *'predictedSales_Prob1.csv'*, there is only one average price choice at $3.0 whereas in *'predictedSales_Prob2.csv'* there are 4 different price choices you can choose form.

If you want to try different average price choices, we would need to repeat this procedure for each average price value and record the corresponding optimal solution to decide how each product should be priced and at which average price level to generate the optimal revenue.

Note that in this demo, we use p_{ij} instead of p_j since it is easier to prepare the script but the model remains identical to the Module 2A because $p_{ij} = p_j, i = 1, ..., n$.

```
# Nere we choose which value of k (avgPriceValue x N. of products) we would like to␣
↪use in the model
# Note that k must be among the choices where the prediction has been prepared
avgPriceValue =  avgPriceList[0]

# Now we select only the row which corresponds to the previously chosen value of␣
↪avgPriceValue (again k = avgPriceValue x N. of products)
predDemand_k = predDemand.loc[predDemand['avgPriceChoice'] ==␣
↪avgPriceValue][inputColumns]
print(predDemand_k)
productList = predDemand_k['UPC'].unique()
priceList = predDemand_k['PRICE'].unique()

# Here we prepare the dictionary to be used in the optimization model
p = {}
D = {}

for upc in productList:
  for price in priceList:
    p[(upc,price)] = price
    D[(upc,price)] = predDemand_k.loc[(predDemand['UPC'] == upc) & (predDemand_k[
↪'PRICE'] == price)]['predictSales'].values[0]

print(p)
print(D)
```

```
    avgPriceChoice         UPC  PRICE  predictSales
0              3.0  1600027528    2.5          95.0
1              3.0  1600027528    3.0          67.0
2              3.0  1600027528    3.5          46.0
3              3.0  1600027564    2.5          24.0
4              3.0  1600027564    3.0          23.0
5              3.0  1600027564    3.5          20.0
6              3.0  3000006340    2.5           6.0
7              3.0  3000006340    3.0           4.0
8              3.0  3000006340    3.5           3.0
9              3.0  3800031829    2.5          33.0
10             3.0  3800031829    3.0          24.0
11             3.0  3800031829    3.5          20.0
{(1600027528, 2.5): 2.5, (1600027528, 3.0): 3.0, (1600027528, 3.5): 3.5,␣
↪(1600027564, 2.5): 2.5, (1600027564, 3.0): 3.0, (1600027564, 3.5): 3.5,␣
↪(3000006340, 2.5): 2.5, (3000006340, 3.0): 3.0, (3000006340, 3.5): 3.5,␣
↪(3800031829, 2.5): 2.5, (3800031829, 3.0): 3.0, (3800031829, 3.5): 3.5}
{(1600027528, 2.5): 95.0, (1600027528, 3.0): 67.0, (1600027528, 3.5): 46.0,␣
↪(1600027564, 2.5): 24.0, (1600027564, 3.0): 23.0, (1600027564, 3.5): 20.0,␣
↪(3000006340, 2.5): 6.0, (3000006340, 3.0): 4.0, (3000006340, 3.5): 3.0,␣
↪(3800031829, 2.5): 33.0, (3800031829, 3.0): 24.0, (3800031829, 3.5): 20.0}
```

2.6.3 Block 3: Create an optimization model

Block 3.1: Variable declarations

Unlike the first part of today's session, we index the decision variables and demand parameters by the product and the price themselves rather than their index. Indeed, we previously denoted $x_{ij} = 1$ if the price option j is chosen for product i, and 0 otherwise. Now, our variable is denoted by $x_{1600027528,\ 3.0} = 1$, which means that product UPC '1600027528' will be sold at 3.0 dollars. The same notational remark applies to predicted demand (D_{ijk}) for the sum of prices k and price (p_{ij}) parameters. We can declare the constraint sets first (model.PriceChoiceUPC, model.sumPrice) and then **add** the constraint functions later.

```python
from pyomo.environ import *

iIndexList = list(range(len(productList)))
jIndexList = list(range(len(priceList)))

model = ConcreteModel()
# Variables
model.x = Var(productList, priceList, within = Binary)

# Constraints
model.PriceChoiceUPC = ConstraintList()
model.sumPrice = ConstraintList()

# Print to review the model (equations are still not included)
model.pprint()
```

```
5 Set Declarations
    PriceChoiceUPC_index : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    0 :      {}
```

(continues on next page)

(continued from previous page)

```
    sumPrice_index : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    0 :      {}
    x_index : Size=1, Index=None, Ordered=False
        Key  : Dimen : Domain              : Size : Members
        None :     2 : x_index_0*x_index_1 :   12 : {(1600027528, 2.5),␣
↪(1600027528, 3.0), (1600027528, 3.5), (1600027564, 2.5), (1600027564, 3.0),␣
↪(1600027564, 3.5), (3000006340, 2.5), (3000006340, 3.0), (3000006340, 3.5),␣
↪(3800031829, 2.5), (3800031829, 3.0), (3800031829, 3.5)}
    x_index_0 : Size=1, Index=None, Ordered=False
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    4 : {1600027528, 1600027564, 3000006340,␣
↪3800031829}
    x_index_1 : Size=1, Index=None, Ordered=False
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    3 : {2.5, 3.0, 3.5}

1 Var Declarations
    x : Size=12, Index=x_index
        Key               : Lower : Value : Upper : Fixed : Stale : Domain
        (1600027528, 2.5) :     0 :  None :     1 : False :  True : Binary
        (1600027528, 3.0) :     0 :  None :     1 : False :  True : Binary
        (1600027528, 3.5) :     0 :  None :     1 : False :  True : Binary
        (1600027564, 2.5) :     0 :  None :     1 : False :  True : Binary
        (1600027564, 3.0) :     0 :  None :     1 : False :  True : Binary
        (1600027564, 3.5) :     0 :  None :     1 : False :  True : Binary
        (3000006340, 2.5) :     0 :  None :     1 : False :  True : Binary
        (3000006340, 3.0) :     0 :  None :     1 : False :  True : Binary
        (3000006340, 3.5) :     0 :  None :     1 : False :  True : Binary
        (3800031829, 2.5) :     0 :  None :     1 : False :  True : Binary
        (3800031829, 3.0) :     0 :  None :     1 : False :  True : Binary
        (3800031829, 3.5) :     0 :  None :     1 : False :  True : Binary

2 Constraint Declarations
    PriceChoiceUPC : Size=0, Index=PriceChoiceUPC_index, Active=True
        Key : Lower : Body : Upper : Active
    sumPrice : Size=0, Index=sumPrice_index, Active=True
        Key : Lower : Body : Upper : Active

8 Declarations: x_index_0 x_index_1 x_index x PriceChoiceUPC_index PriceChoiceUPC␣
↪sumPrice_index sumPrice
```

Block 3.2: Adding an objective function

Instead of iteratively entering the value for each price and predicted sales, we can simply create a loop **for** each product and a loop **for** each price. The code now looks very much like the general equation $\sum_i \sum_j p_{ij} \cdot D_{ijk} \cdot x_{ij}$ we saw in the first part of today's session with some minor changes for notational simplification.

```
# Objective function

obj_expr = sum(p[(i,j)]*D[(i,j)]*model.x[i,j] for i in productList for j in priceList)
print(obj_expr)
model.OBJ = Objective(expr = obj_expr, sense = maximize)
```

```
237.5*x[1600027528,2.5] + 201.0*x[1600027528,3.0] + 161.0*x[1600027528,3.5] + 60.
↪0*x[1600027564,2.5] + 69.0*x[1600027564,3.0] + 70.0*x[1600027564,3.5] + 15.
↪0*x[3000006340,2.5] + 12.0*x[3000006340,3.0] + 10.5*x[3000006340,3.5] + 82.
↪5*x[3800031829,2.5] + 72.0*x[3800031829,3.0] + 70.0*x[3800031829,3.5]
```

Block 3.3: Adding constraints

Constraint 1: One price choice must be selected for each product

Similarly, we can create a loop to **add** constraint functions to the constraint set **for** each product to ensure that only one price on the list is selected for that product. Unlike the first part of today's session, we need not iteratively type each constraint.

```
# Constraints #1
for i in productList:
  const1_expr = sum(model.x[i,j] for j in priceList) == 1
  print(const1_expr)
  model.PriceChoiceUPC.add(expr = const1_expr)
```

```
x[1600027528,2.5] + x[1600027528,3.0] + x[1600027528,3.5]  ==  1
x[1600027564,2.5] + x[1600027564,3.0] + x[1600027564,3.5]  ==  1
x[3000006340,2.5] + x[3000006340,3.0] + x[3000006340,3.5]  ==  1
x[3800031829,2.5] + x[3800031829,3.0] + x[3800031829,3.5]  ==  1
```

Constraint 2: The sum of the prices of all products must equal k

Similar **for** loops apply to the average price constraint. Please refer to the first part of today's session for detailed elaboration.

```
# Constraints #2
const2_expr = sum(p[i,j]*model.x[i,j] for i in productList for j in priceList) ==␣
 ↪avgPriceValue*len(productList)
print(const2_expr)
model.sumPrice.add(expr = const2_expr)
```

```
2.5*x[1600027528,2.5] + 3.0*x[1600027528,3.0] + 3.5*x[1600027528,3.5] + 2.
↪5*x[1600027564,2.5] + 3.0*x[1600027564,3.0] + 3.5*x[1600027564,3.5] + 2.
↪5*x[3000006340,2.5] + 3.0*x[3000006340,3.0] + 3.5*x[3000006340,3.5] + 2.
↪5*x[3800031829,2.5] + 3.0*x[3800031829,3.0] + 3.5*x[3800031829,3.5]  ==  12.0
```

```
<pyomo.core.base.constraint._GeneralConstraintData at 0x2590bd2c580>
```

We can print the model to review prior to solving it.

```
model.pprint()
```

```
5 Set Declarations
    PriceChoiceUPC_index : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    4 : {1, 2, 3, 4}
    sumPrice_index : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
```

(continues on next page)

(continued from previous page)

```
        None :     1 :    Any :    1 :    {1,}
    x_index : Size=1, Index=None, Ordered=False
        Key  : Dimen : Domain              : Size : Members
        None :     2 : x_index_0*x_index_1 :   12 : {(1600027528, 2.5),␣
↪(1600027528, 3.0), (1600027528, 3.5), (1600027564, 2.5), (1600027564, 3.0),␣
↪(1600027564, 3.5), (3000006340, 2.5), (3000006340, 3.0), (3000006340, 3.5),␣
↪(3800031829, 2.5), (3800031829, 3.0), (3800031829, 3.5)}
    x_index_0 : Size=1, Index=None, Ordered=False
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    4 : {1600027528, 1600027564, 3000006340,␣
↪3800031829}
    x_index_1 : Size=1, Index=None, Ordered=False
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    3 : {2.5, 3.0, 3.5}

1 Var Declarations
    x : Size=12, Index=x_index
        Key               : Lower : Value : Upper : Fixed : Stale : Domain
        (1600027528, 2.5) :     0 :  None :     1 : False :  True : Binary
        (1600027528, 3.0) :     0 :  None :     1 : False :  True : Binary
        (1600027528, 3.5) :     0 :  None :     1 : False :  True : Binary
        (1600027564, 2.5) :     0 :  None :     1 : False :  True : Binary
        (1600027564, 3.0) :     0 :  None :     1 : False :  True : Binary
        (1600027564, 3.5) :     0 :  None :     1 : False :  True : Binary
        (3000006340, 2.5) :     0 :  None :     1 : False :  True : Binary
        (3000006340, 3.0) :     0 :  None :     1 : False :  True : Binary
        (3000006340, 3.5) :     0 :  None :     1 : False :  True : Binary
        (3800031829, 2.5) :     0 :  None :     1 : False :  True : Binary
        (3800031829, 3.0) :     0 :  None :     1 : False :  True : Binary
        (3800031829, 3.5) :     0 :  None :     1 : False :  True : Binary

1 Objective Declarations
    OBJ : Size=1, Index=None, Active=True
        Key  : Active : Sense    : Expression
        None :   True : maximize : 237.5*x[1600027528,2.5] + 201.0*x[1600027528,3.
↪0] + 161.0*x[1600027528,3.5] + 60.0*x[1600027564,2.5] + 69.0*x[1600027564,3.0] +␣
↪70.0*x[1600027564,3.5] + 15.0*x[3000006340,2.5] + 12.0*x[3000006340,3.0] + 10.
↪5*x[3000006340,3.5] + 82.5*x[3800031829,2.5] + 72.0*x[3800031829,3.0] + 70.
↪0*x[3800031829,3.5]

2 Constraint Declarations
    PriceChoiceUPC : Size=4, Index=PriceChoiceUPC_index, Active=True
        Key : Lower : Body                                                      :␣
↪Upper : Active
          1 :   1.0 : x[1600027528,2.5] + x[1600027528,3.0] + x[1600027528,3.5] : ␣
↪ 1.0 :   True
          2 :   1.0 : x[1600027564,2.5] + x[1600027564,3.0] + x[1600027564,3.5] : ␣
↪ 1.0 :   True
          3 :   1.0 : x[3000006340,2.5] + x[3000006340,3.0] + x[3000006340,3.5] : ␣
↪ 1.0 :   True
          4 :   1.0 : x[3800031829,2.5] + x[3800031829,3.0] + x[3800031829,3.5] : ␣
↪ 1.0 :   True
    sumPrice : Size=1, Index=sumPrice_index, Active=True
        Key : Lower : Body                                                         ␣
↪                                                                                  ␣
↪                                                                                  ␣
↪                                                              : Upper : Active
```

(continues on next page)

(continued from previous page)

```
        1 :  12.0 : 2.5*x[1600027528,2.5] + 3.0*x[1600027528,3.0] + 3.
↪5*x[1600027528,3.5] + 2.5*x[1600027564,2.5] + 3.0*x[1600027564,3.0] + 3.
↪5*x[1600027564,3.5] + 2.5*x[3000006340,2.5] + 3.0*x[3000006340,3.0] + 3.
↪5*x[3000006340,3.5] + 2.5*x[3800031829,2.5] + 3.0*x[3800031829,3.0] + 3.
↪5*x[3800031829,3.5] :  12.0 :   True

9 Declarations: x_index_0 x_index_1 x_index x PriceChoiceUPC_index PriceChoiceUPC␣
↪sumPrice_index sumPrice OBJ
```

2.6.4 Block 4: Solution and results

Finally, we call the solver and obtain the optimal solution. We can see that product '1600027528' is also sold at price $2.5, products '1600027564' and '3000006340' both at price $3.5 and product '3800031829' at price $2.5, but the optimal objective value is now $399.3. The objective function value is slightly different from the Module_2A but the solution (values of x) is the same. This is due to the fact that we keep more digits in this example.

```
# Solve the model
opt = SolverFactory('glpk')
opt.solve(model)

model.display()
```

```
Model unknown

  Variables:
    x : Size=12, Index=x_index
        Key                  : Lower : Value : Upper : Fixed : Stale : Domain
        (1600027528, 2.5) :     0 :   1.0 :     1 : False : False : Binary
        (1600027528, 3.0) :     0 :   0.0 :     1 : False : False : Binary
        (1600027528, 3.5) :     0 :   0.0 :     1 : False : False : Binary
        (1600027564, 2.5) :     0 :   0.0 :     1 : False : False : Binary
        (1600027564, 3.0) :     0 :   0.0 :     1 : False : False : Binary
        (1600027564, 3.5) :     0 :   1.0 :     1 : False : False : Binary
        (3000006340, 2.5) :     0 :   0.0 :     1 : False : False : Binary
        (3000006340, 3.0) :     0 :   0.0 :     1 : False : False : Binary
        (3000006340, 3.5) :     0 :   1.0 :     1 : False : False : Binary
        (3800031829, 2.5) :     0 :   1.0 :     1 : False : False : Binary
        (3800031829, 3.0) :     0 :   0.0 :     1 : False : False : Binary
        (3800031829, 3.5) :     0 :   0.0 :     1 : False : False : Binary

  Objectives:
    OBJ : Size=1, Index=None, Active=True
        Key  : Active : Value
        None :   True : 400.5

  Constraints:
    PriceChoiceUPC : Size=4
        Key : Lower : Body : Upper
          1 :   1.0 :  1.0 :   1.0
          2 :   1.0 :  1.0 :   1.0
          3 :   1.0 :  1.0 :   1.0
          4 :   1.0 :  1.0 :   1.0
    sumPrice : Size=1
```

(continues on next page)

(continued from previous page)

```
        Key : Lower : Body : Upper
          1 :  12.0 : 12.0 :  12.0
```

CHAPTER

THREE

CASE: ONLINE FULFILLMENT - US ONLINE RETAIL

Case Reference: Acimovic, J., & Graves, S. C. (2015). Making better fulfillment decisions on the fly in an online retail environment. Manufacturing & Service Operations Management, 17(1), 34-51.

3.1 Overview of the case:

This case describes how an online retail company uses a combination of techniques to optimize their fulfillment decisions in an online fashion and reduce outbound shipping costs.

Challenges Faced by the Company

The company faces two main challenges in its online retail operations:

- Myopic Fulfillment Decisions: The company's current approach to fulfilling orders is myopic, meaning they consider only the cheapest shipping option for each order based on their current inventory position, without considering potential future costs. This can lead to higher overall shipping costs in the long run.
- Inventory Management: The company needs to effectively manage inventory levels across multiple fulfillment centers (FCs) to avoid stockouts while minimizing holding costs.

Solution Approach:

The research paper proposes a solution that leverages both approximate dynamic programming and linear programming to address these challenges.

- Approximate Dynamic Programming: A heuristic is developed based on approximate dynamic programming to make fulfillment decisions in an online fashion for each order recieved. This heuristic considers not only the immediate shipping cost of fulfilling an order from a particular FC, but also the estimated future shipping costs based on the impact on future inventory levels.

- Linear Programming (LP): An LP model is used to anticipate the future costs based on a future inventory position that is the result of the fulfillment decision (i.e., which fulfillment center(s) to fulfill this current order received) by considering the antipicated demands and inventory positions across all FCs. Rather than solving a large number of LPs, a calculation derived from the dual values of the LP, can be used to determine which fulfillment center should be used for the current order to minimize the current and future shipping costs.

Reduced Shipping Costs

The proposed approach helps reduce outbound shipping costs by making more strategic fulfillment decisions. Here's how it achieves that:

-Forward-Looking Decisions: By considering the impact of current fulfillment decisions on future inventory levels and shipping costs, the heuristic can choose options that minimize overall shipping costs over time, rather than just focusing on the cheapest option for the immediate order.

-Inventory Optimization: The LP-based approach helps identify opportunities to better allocate inventory across FCs, potentially reducing the need for expensive expedited shipments in the future.

The case mentions that the company has implemented the proposed heuristic and achieved a reduction in outbound shipping costs on the order of 1%, while capturing 36% of the potential opportunity for improvement. These results suggest that the approach can generate significant cost savings for the company.

3.2 Demo: Online Fulfillment

In this demo, we also begin by installing the (i) Pyomo package and (ii) the linear programming solver GLPK (GNU Linear Programming Kit). Please feel free to revisit the previous notebook for further information.

```
# Install Pyomo and GLPK in your Python environment if it is not already done.
pip install -q pyomo
conda install conda-forge::glpk
```

3.2.1 Blocks 1&2: Data input and parameters

We have two different datasets, but their structures are the same. The dataset 1 is the same as in the paper whereas the dataset 2 is a larger dataset to demonstrate how data can be adapted.

1. **Fulfillment center (FC) data**: each dataset contains a set of fulfillment centers FC (*listFC*), each of which has a given level of inventory **X** (*inventoryFC*). Customers can place an order for either a single item or multiple items, but if no fulfillment center has all the items available, the retailer has to resort to split deliveries, which obviously affect the shipping expenses. Therefore, our optimization model has to take account of the probability that the fulfillment center has 'other items in order' (*probMultiAvailability*).
2. **Customer region data**: each customer zone (*listRegion*) has to be associated with a certain demand over the next τ days (*demandValue*) for customer j and the proportion of orders that are for multiple items (*probMultiItem*).
3. **Shipment cost data**: *costSingle*$[i, j]$ denotes the cost of a single delivery from Fulfillment Center (FC) *i* to Customer Zone j or c_{ij}. Please note that for Data 1, since there is only one customer zone (Kansas), there is only one cost from each origin (i.e., from FC *i* to Kansas). *avgNoMultiItem* stands for the *average number of items in a multi-item order*. This will help us compute the average shipping cost per item for multi-item orders. More specifically $\omega = 1/\text{avgNoMultiItem}$.

The first set of parameters **Data 1** corresponds to the example provided in the case below (Figure from Acimovic and Graves, 2015).

```
# Data 1 (same as in the paper)

# Fulfillment center (FC) data
listFC = ['Utah', 'Nevada']
inventoryFC = dict(zip(listFC, [5, 20]))
print(inventoryFC)
probMultiAvailability = dict(zip(listFC, [0.5, 0.2]))
print(probMultiAvailability)

# Customer region data
listRegion = ['Kansas']
demandValue = dict(zip(listRegion, [2*10]))
print(demandValue)
probMultiItem = dict(zip(listRegion, [0.75]))
print(probMultiItem)

# Shipment cost data
```

(continues on next page)

(continued from previous page)

```
costSingle = [[9], [12]]
print(costSingle)

# average number of items in a multi-item order
avgNoMultiItem = 3.0
# we can calculate the multi-item shipping discount $\omega$
shippingDiscount = 1/avgNoMultiItem

# Prepare shipment cost above in dictionary format
costSingleDict = {}
for i in range(len(listFC)):
  for j in range(len(listRegion)):
    costSingleDict[(listFC[i],listRegion[j])] = costSingle[i][j]

print(costSingleDict)
```

```
{'Utah': 5, 'Nevada': 20}
{'Utah': 0.5, 'Nevada': 0.2}
{'Kansas': 20}
{'Kansas': 0.75}
[[9], [12]]
{('Utah', 'Kansas'): 9, ('Nevada', 'Kansas'): 12}
```

```
# Data 2

# Fulfillment center (FC) data
listFC = ['Delta-BC', 'Brampton-ON', 'Ottawa-ON']
inventoryFC = dict(zip(listFC, [37, 85, 25]))
print(inventoryFC)
probMultiAvailability = dict(zip(listFC, [0.32, 0.45, 0.17]))
print(probMultiAvailability)

# Customer region data
listRegion = ['Toronto', 'Montreal', 'Calgary', 'Vancouver']
demandValue = dict(zip(listRegion, [45, 27, 15, 33]))
print(demandValue)
probMultiItem = dict(zip(listRegion, [0.73, 0.68, 0.54, 0.64]))
print(probMultiItem)

# Shipment cost data
costSingle = [[24.5, 25.5, 18.1, 12.3],
        [13.6, 17.5, 22.8, 23.6],
        [18.1, 14.1, 21.1, 22.8]]

# average number of items in a multi-item order
avgNoMultiItem = 2.5
# we can calculate the multi-item shipping discount $\omega$
shippingDiscount = 1/avgNoMultiItem

# Prepare shipment cost above in dictionary format
costSingleDict = {}
for i in range(len(listFC)):
  for j in range(len(listRegion)):
    costSingleDict[(listFC[i],listRegion[j])] = costSingle[i][j]
```

(continues on next page)

(continued from previous page)

```
print(costSingleDict)
```

```
{'Delta-BC': 37, 'Brampton-ON': 85, 'Ottawa-ON': 25}
{'Delta-BC': 0.32, 'Brampton-ON': 0.45, 'Ottawa-ON': 0.17}
{'Toronto': 45, 'Montreal': 27, 'Calgary': 15, 'Vancouver': 33}
{'Toronto': 0.73, 'Montreal': 0.68, 'Calgary': 0.54, 'Vancouver': 0.64}
{('Delta-BC', 'Toronto'): 24.5, ('Delta-BC', 'Montreal'): 25.5, ('Delta-BC',
↪'Calgary'): 18.1, ('Delta-BC', 'Vancouver'): 12.3, ('Brampton-ON', 'Toronto'):␣
↪13.6, ('Brampton-ON', 'Montreal'): 17.5, ('Brampton-ON', 'Calgary'): 22.8, (
↪'Brampton-ON', 'Vancouver'): 23.6, ('Ottawa-ON', 'Toronto'): 18.1, ('Ottawa-ON',
↪'Montreal'): 14.1, ('Ottawa-ON', 'Calgary'): 21.1, ('Ottawa-ON', 'Vancouver'):␣
↪22.8}
```

3.2.2 Block 3: Create an optimization model

Block 3.1: Variable declarations

As we all learned in Retail Analytics Case, an optimization model consists of (i) decision variables, (ii) objective function, and (iii) constraints.

Our decision variables include model.x, model.y and model.w, which are all nonnegative, denoting the flow from FC *i* to customer region *j*. In particular,

model.**x**$[i, j]$ (or x_{ij}) denotes the decision variable for **unsplit** flow from FC *i* to **multi-item** customer *j*. The unit cost of this flow is ωc_{ij};

model.**y**$[i, j]$ (or y_{ij}) denotes the decision variable for **split** flow from FC *i* to **multi-item** customer *j*. The unit cost of this flow is $2\omega c_{ij}$ (two shipments are used if the order cannot be shipped in one shipment);

model.**w**$[i, j]$ (or w_{ij}) denotes the decision variable for flow from FC *i* to **single-item** customer *j*. The unit cost of this flow is c_{ij}.

On the last line, we also create an object to store the shadow prices (dual variables). This is used if we don't want to solve the LP multiple times.

```
from pyomo.environ import *

model = ConcreteModel()
# Variables
model.x = Var(listFC, listRegion, within = NonNegativeReals)
model.y = Var(listFC, listRegion, within = NonNegativeReals)
model.w = Var(listFC, listRegion, within = NonNegativeReals)

model.inventoryOnHand = ConstraintList()
model.demandSingle = ConstraintList()
model.demandMulitiple = ConstraintList()
model.maxMultiShipment = ConstraintList()

# create an object to access to shadow prices. Please note that the codes must be␣
↪exactly written as
# model.dual = Suffix(direction=Suffix.IMPORT_EXPORT) in order for it to work because␣
↪it was hard coded in pyomo.
model.dual = Suffix(direction=Suffix.IMPORT_EXPORT)
```

Block 3.2: Objective function

For notational simplication, let

c_{ij} = costSingle$[i, j]$, the cost of a single delivery from Fulfillment Center (FC) *i* to Customer Zone *j*;

$\omega = 1$ / avgNoMultiItem, expected discount of sending multi-item order in one package.

Then, we have the general form of the objective function as follows

$$\min_{x,y,w} \left\{ \sum_{i \in \mathcal{FC}} \sum_{j \in \mathcal{J}} \left(c_{ij} \cdot w_{ij} + \omega \cdot c_{ij} \cdot x_{ij} + 2\omega \cdot c_{ij} \cdot y_{ij} \right) \right\}$$

```
# Objective function

obj_expr = sum(costSingleDict[(i,j)]*model.w[(i,j)] +␣
↪shippingDiscount*costSingleDict[(i,j)]*model.x[(i,j)] \
                + (2*shippingDiscount)*costSingleDict[(i,j)]*model.y[(i,j)] for i in␣
↪listFC for j in listRegion)
print(obj_expr)
model.OBJ = Objective(expr = obj_expr, sense = minimize)
```

```
24.5*w[Delta-BC,Toronto] + 9.8*x[Delta-BC,Toronto] + 19.6*y[Delta-BC,Toronto] + 25.
↪5*w[Delta-BC,Montreal] + 10.200000000000001*x[Delta-BC,Montreal] + 20.
↪400000000000002*y[Delta-BC,Montreal] + 18.1*w[Delta-BC,Calgary] + 7.
↪240000000000001*x[Delta-BC,Calgary] + 14.480000000000002*y[Delta-BC,Calgary] +␣
↪12.3*w[Delta-BC,Vancouver] + 4.920000000000001*x[Delta-BC,Vancouver] + 9.
↪840000000000002*y[Delta-BC,Vancouver] + 13.6*w[Brampton-ON,Toronto] + 5.
↪44*x[Brampton-ON,Toronto] + 10.88*y[Brampton-ON,Toronto] + 17.5*w[Brampton-ON,
↪Montreal] + 7.0*x[Brampton-ON,Montreal] + 14.0*y[Brampton-ON,Montreal] + 22.
↪8*w[Brampton-ON,Calgary] + 9.120000000000001*x[Brampton-ON,Calgary] + 18.
↪240000000000002*y[Brampton-ON,Calgary] + 23.6*w[Brampton-ON,Vancouver] + 9.
↪440000000000001*x[Brampton-ON,Vancouver] + 18.880000000000003*y[Brampton-ON,
↪Vancouver] + 18.1*w[Ottawa-ON,Toronto] + 7.240000000000001*x[Ottawa-ON,Toronto]␣
↪+ 14.480000000000002*y[Ottawa-ON,Toronto] + 14.1*w[Ottawa-ON,Montreal] + 5.
↪640000000000001*x[Ottawa-ON,Montreal] + 11.280000000000001*y[Ottawa-ON,Montreal]␣
↪+ 21.1*w[Ottawa-ON,Calgary] + 8.440000000000001*x[Ottawa-ON,Calgary] + 16.
↪880000000000003*y[Ottawa-ON,Calgary] + 22.8*w[Ottawa-ON,Vancouver] + 9.
↪120000000000001*x[Ottawa-ON,Vancouver] + 18.240000000000002*y[Ottawa-ON,
↪Vancouver]
```

Block 3.3: Constraints functions

Constraint Set 1: Inventory availability at FC *i*

This set of constraints ensures that the total demand (orders) assigned to FC *i* is less than or equal to the inventory at FC *i*.

Let X_i denote the inventory level at FC *i* and $\mathcal{FC}$ denote the list of all FCs. $\sum_{j \in \mathcal{J}} \left(w_{ij} + x_{ij} + y_{ij} \right) \leq X_i, \forall i \in \mathcal{FC}$

```
# Constraints 1 Inventory availability at FC i

for i in listFC:
  const_expr = sum(model.w[(i,j)] + model.x[(i,j)] + model.y[(i,j)] for j in␣
↪listRegion) <= inventoryFC[i]
  print(const_expr)
  model.inventoryOnHand.add(expr = const_expr)
```

```
w[Delta-BC,Toronto] + x[Delta-BC,Toronto] + y[Delta-BC,Toronto] + w[Delta-BC,
↪Montreal] + x[Delta-BC,Montreal] + y[Delta-BC,Montreal] + w[Delta-BC,Calgary] +␣
↪x[Delta-BC,Calgary] + y[Delta-BC,Calgary] + w[Delta-BC,Vancouver] + x[Delta-BC,
↪Vancouver] + y[Delta-BC,Vancouver]  <=  37
w[Brampton-ON,Toronto] + x[Brampton-ON,Toronto] + y[Brampton-ON,Toronto] +␣
↪w[Brampton-ON,Montreal] + x[Brampton-ON,Montreal] + y[Brampton-ON,Montreal] +␣
↪w[Brampton-ON,Calgary] + x[Brampton-ON,Calgary] + y[Brampton-ON,Calgary] +␣
↪w[Brampton-ON,Vancouver] + x[Brampton-ON,Vancouver] + y[Brampton-ON,Vancouver]
↪<=  85
w[Ottawa-ON,Toronto] + x[Ottawa-ON,Toronto] + y[Ottawa-ON,Toronto] + w[Ottawa-ON,
↪Montreal] + x[Ottawa-ON,Montreal] + y[Ottawa-ON,Montreal] + w[Ottawa-ON,Calgary]␣
↪+ x[Ottawa-ON,Calgary] + y[Ottawa-ON,Calgary] + w[Ottawa-ON,Vancouver] +␣
↪x[Ottawa-ON,Vancouver] + y[Ottawa-ON,Vancouver]  <=  25
```

Constraint Set 2: Future demand for single-item order in region *j*

This set of constraints ensures that the single-item demand at Customer (Demand) Zone *j* is satisfied.

Let D_j denote the demand at Customer Zone *j*, $\mathcal{J}$ denote the list of all Customer (Demand) Zones and λ_j denote the proportion of orders that are for multiple items (probMultiItem) ⇒ $(1-\lambda_j)$ equals the proportion of orders that are for a single item. $\sum_{i\in\mathcal{FC}} w_{ij} = D_j \cdot (1-\lambda_j)\,, \forall j \in \mathcal{J}$

```
# Constraints 2 Future demand for single-item order in region j

for j in listRegion:
  const_expr = sum(model.w[(i,j)] for i in listFC) == demandValue[j]*(1-
↪probMultiItem[j])
  print(const_expr)
  model.demandSingle.add(expr = const_expr)
```

```
w[Delta-BC,Toronto] + w[Brampton-ON,Toronto] + w[Ottawa-ON,Toronto]  ==  12.15
w[Delta-BC,Montreal] + w[Brampton-ON,Montreal] + w[Ottawa-ON,Montreal]  ==  8.
↪639999999999999
w[Delta-BC,Calgary] + w[Brampton-ON,Calgary] + w[Ottawa-ON,Calgary]  ==  6.
↪8999999999999995
w[Delta-BC,Vancouver] + w[Brampton-ON,Vancouver] + w[Ottawa-ON,Vancouver]  ==  11.
↪879999999999999
```

Constraint Set 3: Future demand for multi-item order in region *j*

This set of constraints ensures that the multi-item demand at Customer (Demand) Zone *j* is satisfied.

Using the similar notations as in *Constraint set 2*, we attain $\sum_{i\in\mathcal{FC}} (x_{ij} + y_{ij}) = D_j \cdot \lambda_j, \forall j \in \mathcal{J}$

```
# Constraints 3 Future demand for multi-item order in region j

for j in listRegion:
  const_expr = sum(model.x[(i,j)] + model.y[(i,j)] for i in listFC) ==␣
↪demandValue[j]*probMultiItem[j]
  print(const_expr)
  model.demandMulitiple.add(expr = const_expr)
```

```
x[Delta-BC,Toronto] + y[Delta-BC,Toronto] + x[Brampton-ON,Toronto] + y[Brampton-ON,
↪Toronto] + x[Ottawa-ON,Toronto] + y[Ottawa-ON,Toronto]  ==  32.85
x[Delta-BC,Montreal] + y[Delta-BC,Montreal] + x[Brampton-ON,Montreal] + y[Brampton-
↪ON,Montreal] + x[Ottawa-ON,Montreal] + y[Ottawa-ON,Montreal]  ==  18.
↪360000000000003
```

(continues on next page)

(continued from previous page)

```
x[Delta-BC,Calgary] + y[Delta-BC,Calgary] + x[Brampton-ON,Calgary] + y[Brampton-ON,
↪Calgary] + x[Ottawa-ON,Calgary] + y[Ottawa-ON,Calgary]  ==  8.100000000000001
x[Delta-BC,Vancouver] + y[Delta-BC,Vancouver] + x[Brampton-ON,Vancouver] +␣
↪y[Brampton-ON,Vancouver] + x[Ottawa-ON,Vancouver] + y[Ottawa-ON,Vancouver]  == ␣
↪21.12
```

Constraint Set 4: Estimated maximum number multi-item shipments from FC *i* to customer region *j*

It should be noted that unsplit delivery for a multi-item order from FC *i* to Customer Region *j* is effected only when FC *i* has 'other items in order'. Let ρ_i denote the probability that FC *i* has 'other items in order' (probMultiAvailability). Then, we impose the following constraints.

$$x_{ij} \leq D_j \cdot \lambda_j \cdot \rho_i, \forall i \in \mathcal{FC}, j \in \mathcal{J}$$

```
# Constraints #4 Estimated maximum number multi-item shipments from FC i to customer␣
↪regions j

for i in listFC:
  for j in listRegion:
    const_expr = model.x[(i,j)] <=␣
↪demandValue[j]*probMultiItem[j]*probMultiAvailability[i]
    print(const_expr)
    model.maxMultiShipment.add(expr = const_expr)
```

```
x[Delta-BC,Toronto]  <=  10.512
x[Delta-BC,Montreal]  <=  5.875200000000001
x[Delta-BC,Calgary]  <=  2.5920000000000005
x[Delta-BC,Vancouver]  <=  6.758400000000001
x[Brampton-ON,Toronto]  <=  14.7825
x[Brampton-ON,Montreal]  <=  8.262000000000002
x[Brampton-ON,Calgary]  <=  3.645000000000001
x[Brampton-ON,Vancouver]  <=  9.504000000000001
x[Ottawa-ON,Toronto]  <=  5.5845
x[Ottawa-ON,Montreal]  <=  3.121200000000001
x[Ottawa-ON,Calgary]  <=  1.3770000000000004
x[Ottawa-ON,Vancouver]  <=  3.5904000000000003
```

You can print the entire model to check if you want

```
# This is commented so that we won't print out unless necessary. You can outcomment␣
↪to print it.
# model.pprint()
```

3.2.3 Block 4: Solution and results

Finally, we call for the solver and obtain the solution. The first line indicates which solver we want to use and the second line solves the model (this is equivalent to *fit()* in sklearn). The last line *displays* the solution.

```
# Solve the model
opt = SolverFactory('glpk')
opt.solve(model)

model.display()
```

```
Model unknown

  Variables:
    x : Size=12, Index=x_index
        Key                          : Lower : Value   : Upper : Fixed : Stale :␣
 ↪Domain
          ('Brampton-ON', 'Calgary') :     0 :   3.645 :  None : False : False :␣
 ↪NonNegativeReals
         ('Brampton-ON', 'Montreal') :     0 :   8.262 :  None : False : False :␣
 ↪NonNegativeReals
          ('Brampton-ON', 'Toronto') :     0 : 14.7825 :  None : False : False :␣
 ↪NonNegativeReals
        ('Brampton-ON', 'Vancouver') :     0 :   9.504 :  None : False : False :␣
 ↪NonNegativeReals
             ('Delta-BC', 'Calgary') :     0 :   2.592 :  None : False : False :␣
 ↪NonNegativeReals
            ('Delta-BC', 'Montreal') :     0 :  5.8752 :  None : False : False :␣
 ↪NonNegativeReals
             ('Delta-BC', 'Toronto') :     0 :  1.2412 :  None : False : False :␣
 ↪NonNegativeReals
           ('Delta-BC', 'Vancouver') :     0 :  6.7584 :  None : False : False :␣
 ↪NonNegativeReals
            ('Ottawa-ON', 'Calgary') :     0 :   1.377 :  None : False : False :␣
 ↪NonNegativeReals
           ('Ottawa-ON', 'Montreal') :     0 :  3.1212 :  None : False : False :␣
 ↪NonNegativeReals
            ('Ottawa-ON', 'Toronto') :     0 :  5.5845 :  None : False : False :␣
 ↪NonNegativeReals
          ('Ottawa-ON', 'Vancouver') :     0 :  3.5904 :  None : False : False :␣
 ↪NonNegativeReals
    y : Size=12, Index=y_index
        Key                          : Lower : Value             : Upper : Fixed :␣
 ↪Stale : Domain
          ('Brampton-ON', 'Calgary') :     0 :               0.0 :  None : False :␣
 ↪False : NonNegativeReals
         ('Brampton-ON', 'Montreal') :     0 :               0.0 :  None : False :␣
 ↪False : NonNegativeReals
          ('Brampton-ON', 'Toronto') :     0 :           11.2418 :  None : False :␣
 ↪False : NonNegativeReals
        ('Brampton-ON', 'Vancouver') :     0 :               0.0 :  None : False :␣
 ↪False : NonNegativeReals
             ('Delta-BC', 'Calgary') :     0 : 0.485999999999999 :  None : False :␣
 ↪False : NonNegativeReals
            ('Delta-BC', 'Montreal') :     0 :               0.0 :  None : False :␣
 ↪False : NonNegativeReals
             ('Delta-BC', 'Toronto') :     0 :               0.0 :  None : False :␣
 ↪False : NonNegativeReals
           ('Delta-BC', 'Vancouver') :     0 :            1.2672 :  None : False :␣
 ↪False : NonNegativeReals
            ('Ottawa-ON', 'Calgary') :     0 :               0.0 :  None : False :␣
 ↪False : NonNegativeReals
           ('Ottawa-ON', 'Montreal') :     0 :            1.1016 :  None : False :␣
 ↪False : NonNegativeReals
            ('Ottawa-ON', 'Toronto') :     0 :               0.0 :  None : False :␣
 ↪False : NonNegativeReals
          ('Ottawa-ON', 'Vancouver') :     0 :               0.0 :  None : False :␣
 ↪False : NonNegativeReals
```

(continues on next page)

(continued from previous page)

```
   w : Size=12, Index=w_index
       Key                          : Lower : Value : Upper : Fixed : Stale :␣
↪Domain
         ('Brampton-ON', 'Calgary') :     0 :   0.0 :  None : False : False :␣
↪NonNegativeReals
        ('Brampton-ON', 'Montreal') :     0 :   0.0 :  None : False : False :␣
↪NonNegativeReals
         ('Brampton-ON', 'Toronto') :     0 : 12.15 :  None : False : False :␣
↪NonNegativeReals
       ('Brampton-ON', 'Vancouver') :     0 :   0.0 :  None : False : False :␣
↪NonNegativeReals
            ('Delta-BC', 'Calgary') :     0 :   6.9 :  None : False : False :␣
↪NonNegativeReals
           ('Delta-BC', 'Montreal') :     0 :   0.0 :  None : False : False :␣
↪NonNegativeReals
            ('Delta-BC', 'Toronto') :     0 :   0.0 :  None : False : False :␣
↪NonNegativeReals
          ('Delta-BC', 'Vancouver') :     0 : 11.88 :  None : False : False :␣
↪NonNegativeReals
           ('Ottawa-ON', 'Calgary') :     0 :   0.0 :  None : False : False :␣
↪NonNegativeReals
          ('Ottawa-ON', 'Montreal') :     0 :  8.64 :  None : False : False :␣
↪NonNegativeReals
           ('Ottawa-ON', 'Toronto') :     0 :   0.0 :  None : False : False :␣
↪NonNegativeReals
         ('Ottawa-ON', 'Vancouver') :     0 :   0.0 :  None : False : False :␣
↪NonNegativeReals

 Objectives:
   OBJ : Size=1, Index=None, Active=True
       Key  : Active : Value
       None :   True : 1200.042204

 Constraints:
   inventoryOnHand : Size=3
       Key : Lower : Body               : Upper
         1 :  None :  37.00000000000001 :  37.0
         2 :  None : 59.585300000000004 :  85.0
         3 :  None :            23.4147 :  25.0
   demandSingle : Size=4
       Key : Lower              : Body  : Upper
         1 :              12.15 : 12.15 :              12.15
         2 :  8.639999999999999 :  8.64 :  8.639999999999999
         3 : 6.8999999999999995 :   6.9 : 6.8999999999999995
         4 : 11.879999999999999 : 11.88 : 11.879999999999999
   demandMulitiple : Size=4
       Key : Lower              : Body               : Upper
         1 :              32.85 :              32.85 :              32.85
         2 : 18.360000000000003 : 18.360000000000003 : 18.360000000000003
         3 :  8.100000000000001 :                8.1 :  8.100000000000001
         4 :              21.12 :              21.12 :              21.12
   maxMultiShipment : Size=12
       Key : Lower : Body   : Upper
         1 :  None : 1.2412 :             10.512
         2 :  None : 5.8752 :  5.875200000000001
         3 :  None :  2.592 : 2.5920000000000005
```

(continues on next page)

(continued from previous page)

```
          4 :  None :  6.7584 :  6.758400000000001
          5 :  None : 14.7825 :            14.7825
          6 :  None :   8.262 :  8.262000000000002
          7 :  None :   3.645 :  3.645000000000001
          8 :  None :   9.504 :  9.504000000000001
          9 :  None :  5.5845 :             5.5845
         10 :  None :  3.1212 :  3.121200000000001
         11 :  None :   1.377 : 1.3770000000000004
         12 :  None :  3.5904 : 3.5904000000000003
```

In order to quickly determine the approximate cost-to-go without solving multiple times the LP model above, we can also make use of the shadow price. The codes below print out the shadow prices of all the constraints. Only the shadow prices of the constraint *inventoryOnHand* are used to calculate the approximate cost-to-go, i.e., $$C_k(\mathbf{X_k}) = \min_{i\in\mathcal{FC}}\left(c_{ik} + C_{k+1}(\mathbf{X_{k+1}})\right) = \min_{i\in\mathcal{FC}}\left(c_{ik} + C_{k+1}(\mathbf{X_k}) - \pi_i\right).$$

```
# Obtain reduced cost for each constraint
model.dual.display()
```

```
dual : Direction=Suffix.IMPORT_EXPORT, Datatype=Suffix.FLOAT
    Key                  : Value
      demandMulitiple[1] :                10.88
      demandMulitiple[2] :                11.28
      demandMulitiple[3] :                15.56
      demandMulitiple[4] :                10.92
         demandSingle[1] :                 13.6
         demandSingle[2] :                 14.1
         demandSingle[3] :                19.18
         demandSingle[4] :                13.38
      inventoryOnHand[1] :                -1.08
      inventoryOnHand[2] :                  0.0
      inventoryOnHand[3] :                  0.0
    maxMultiShipment[10] :                -5.64
    maxMultiShipment[11] :                -7.12
    maxMultiShipment[12] :                 -1.8
     maxMultiShipment[1] :                  0.0
     maxMultiShipment[2] : -1.77635683940025e-15
     maxMultiShipment[3] :                -7.24
     maxMultiShipment[4] :                -4.92
     maxMultiShipment[5] :                -5.44
     maxMultiShipment[6] :                -4.28
     maxMultiShipment[7] :                -6.44
     maxMultiShipment[8] :                -1.48
     maxMultiShipment[9] :                -3.64
```

CHAPTER

FOUR

CASE: DISRUPTION ANALYTICS - FORD MOTOR

Case Reference: Simchi-Levi, D., Schmidt, W., Wei, Y., Zhang, P.Y., Combs, K., Ge, Y., Gusikhin, O., Sanders, M. and Zhang, D., 2015. Identifying risks and mitigating disruptions in the automotive supply chain. INFORMS Journal on Applied Analytics, 45(5), pp.375-390.

4.1 Overview of the case:

This case outlines the challenges faced by disruption risk analytics models in the context of a large automaker like Ford. These challenges include the rarity of disruptive events, which makes it difficult to gather sufficient data for accurate modeling. Additionally, risk assessment often involves subjective judgments, leading to potential biases in the analysis. The scale and complexity of Ford's global supply chain further complicate the task of developing effective models to anticipate and mitigate disruptions.

The case involves a collaboration between MIT researchers and Ford Motor Company to develop a new method for assessing risks associated with disruptions in a company's supply chain. The paper outlines the challenges faced by Ford, proposes a new model called the risk-exposure index (REI), and details two related models, the time-to-recover (TTR) and time-to-survive (TTS), to assess these risks.

Challenges Faced by Ford

Ford, like many companies, faces challenges in proactively managing disruptions in its supply chain caused by low-probability, high-impact events. These disruptions are difficult to predict and quantify, and can have a significant material impact on a company's performance. The complexity and scale of Ford's supply chain, with over 4,400 manufacturing sites across 10 tiers of suppliers, further complicate this problem.

Risk Exposure Index (REI)

The REI is a model developed to address the limitations of legacy risk-assessment processes. These legacy processes typically focus on tracking a small number of suppliers and parts, and may not identify significant exposures hidden among lower-tier suppliers. The REI defers the need to estimate the probability of specific disruptions by focusing instead on the impact of the disruption itself, regardless of the cause. In other words, REI is calculated as the relative value of the potential disruption cost faced by a location divided by the maximum disruption cost faced in the worst case scenario. This allows Ford to identify the weakest links in its supply chain and allocate resources more effectively towards mitigating those risks.

Time-to-Recover (TTR) and Time-to-Survive (TTS) Models

The TTR and TTS models are mathematical tools used in conjunction with the REI to assess the impact of disruptions on Ford's supply chain.

** - The TTR model** considers the time it takes for each supplier in the supply chain to recover from a disruption. It factors in operational and financial data, along with in-transit and on-site inventory levels, to simulate the impact of a disruption on Ford's performance metrics such as lost production or sales.[1] ** - The TTS model** calculates the maximum amount of time Ford's system can function without demand loss if a particular supplier is disrupted. This allows Ford to identify suppliers who are critical to its operations and warrant further investigation into their recovery timelines.

Both the TTR and TTS models are linear programming (LP) models, which can be solved very efficiently.

Reducing Disruption Risks

By using the REI, TTR, and TTS models, Ford can gain valuable insights into the weakest links in its supply chain and the potential impact of disruptions. This allows Ford to develop targeted risk-mitigation strategies such as:

- Identifying critical suppliers and collaborating with them to improve their disaster preparedness.
- Diversifying sources of supply to reduce reliance on any single supplier.
- Stockpiling critical parts to mitigate the impact of disruptions.

Overall, the proposed approach allows Ford to make data-driven decisions to reduce its vulnerability to supply chain disruptions.

The overall workflow can be depicted as follows (Figure from Simchi-Levi, 2015).

4.2 Demo: Supply Chain Distruption Risk Analytics

We start our prescriptive modeling by installing the (i) Pyomo package and (ii) the linear programming solver GLPK (GNU Linear Programming Kit). Please feel free to revisit Online Fulfillment Demo for further information.

```
# Install Pyomo and GLPK in your Python environment if it is not already done.
pip install -q pyomo
conda install conda-forge::glpk
```

4.2.1 Blocks 1&2: Data inputs and parameters

We have two different datasets. The latter has more complex configurations, but their structures are similar.

- In problem 1, the instance consists of 3 suppliers, 2 products (vehicles) and 4 scenarios. There is only one supply chain configuration;
- In problem 2, the instance consists of 5 suppliers, 8 products (vehicles) and 21 scenarios. There are three supply chain configurations (A, B and C) and the analysis for each configuration must be done separately (in different runs).

We would recommend that you review problem 1 to see the structure of the model prior to trying the problem 2.

** Details of data inputs & parameters ** The DataFrame **inputScenarios** contains the values of active link α_{ij}^{n} from supply node i for product/vehicle j each scenario n. $\alpha_{ij}^{n} = 1$ indicates that that link is active in this scenario whereas $\alpha_{ij}^{n} = 0$ indicates that the link is inactive (disrupted/not used) in that scenario.

Meanwhile, the DataFrame **inputParameters** contains a set of parameters as follows: () contains the capacity of each supplier (in **supplierList**), demand for

- Column *Type* indicates if the parameter is for the supply node (Supplier) or product node (Product)
- Column *Parameter* indicates what parameter is given in that row. The parameters include
- Capacity of supply node i: c_i
- Demand of product node j: d_j
- Initial inventory of product j: s_j
- Unit profit (profit margin) of product j: f_j

- Time-to-recover (TTR) of each scenario n: t^n
- Column *Index* indicates the index associated with that parameter
- Column *Value* indicates the value of that parameter

```
import pandas
import matplotlib.pyplot as plt
from pyomo.environ import *
from IPython.display import display # this one can be used to display dataframes in␣
↪case you want to print out multiple of them

# Problem 1 (small scale, one configuration)
urlParameters = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/S11_
↪Disruption_Management/inputParameters_Prob1.csv'
urlScenario  = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/S11_
↪Disruption_Management/inputScenarios_Prob1.csv'

# Problem 2 (large scale, 3 possible SC configurations - A, B and C)
# urlParameters = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/
↪S11_Disruption_Management/inputParameters_Prob2.csv'

# Here are the scenarios under each configuration. The analysis could be done for one␣
↪SC configuration at a time so only choose one url among them.
# urlScenario  = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/
↪S11_Disruption_Management/inputScenarios_Prob2_Config_A.csv'
# urlScenario  = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/
↪S11_Disruption_Management/inputScenarios_Prob2_Config_B.csv'
# urlScenario  = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/
↪S11_Disruption_Management/inputScenarios_Prob2_Config_C.csv'

# read CSV files
inputScenarios = pandas.read_csv(urlScenario)
inputParameters = pandas.read_csv(urlParameters)

# Just in case you want to see the loaded DataFrames, you can display or print them
# (display gives a more beautiful output for DataFrame)
display(inputScenarios)
display(inputParameters)
```

```
  Node Product  Scenario_0  Scenario_1  Scenario_2  Scenario_3
0   S1      P1           1           0           1           1
1   S1      P2           1           0           1           1
2   S2      P1           1           1           0           1
3   S2      P2           1           1           0           1
4   S3      P1           1           1           1           0
5   S3      P2           1           1           1           0
```

```
       Type  Parameter       Index  Value
0  Supplier   Capacity          S1     30
1  Supplier   Capacity          S2     45
2  Supplier   Capacity          S3     60
3   Product     Demand          P1     75
4   Product     Demand          P2     50
5   Product  Inventory          P1     20
6   Product  Inventory          P2     25
7   Product       Loss          P1   5000
```

(continues on next page)

(continued from previous page)

```
8      Product       Loss          P2   6500
9   Disruption        TTR  Scenario_0      2
10  Disruption        TTR  Scenario_1      2
11  Disruption        TTR  Scenario_2      2
12  Disruption        TTR  Scenario_3      2
```

We then create the inputs used in the Pyomo model from the DataFrames above.

Lists:

- **productList** contains the list of products (vehicles), or set $\mathcal{V}$
- **supplierList** contains the list of supply nodes (plants), or set $\mathcal{A}$
- **scenarioList** contains the list of scenarios

Dictionaries of data:

- **supplierCapacityDict[i]** contains the values of c_i;
- **productDemandDict[j]** contains the values of d_j;
- **productInvDict[j]** contains the values of s_j;
- **productLossDict[j]** contains the values of f_j;
- **scenarioTTRDict[n]** contains the values of t^n;
- **scenarioActiveNodeDict[n][(i,j)]** contains the values of α^n_{ij} whether the link (i, j) is active in scenario n for not.

```
productList = inputParameters.loc[inputParameters['Type']=='Product']['Index'].
 ↪unique()
supplierList = inputParameters.loc[inputParameters['Type']=='Supplier']['Index'].
 ↪unique()
scenarioList = inputParameters.loc[inputParameters['Type']=='Disruption']['Index'].
 ↪unique()

supplierCapacityDict = {}
for i in supplierList:
  cap = inputParameters.loc[(inputParameters['Parameter']=='Capacity') &␣
 ↪(inputParameters['Index']==i)]['Value'].values[0]
  supplierCapacityDict[i] = cap

print('Print SupplierCapacityDict')
for key in supplierCapacityDict:
    print(key, ' : ', supplierCapacityDict[key])
#print('\n')

productDemandDict = {}
productInvDict = {}
productLossDict = {}

for j in productList:
  dem = inputParameters.loc[(inputParameters['Parameter']=='Demand') &␣
 ↪(inputParameters['Index']==j)]['Value'].values[0]
  inv = inputParameters.loc[(inputParameters['Parameter']=='Inventory') &␣
 ↪(inputParameters['Index']==j)]['Value'].values[0]
  loss = inputParameters.loc[(inputParameters['Parameter']=='Loss') &␣
 ↪(inputParameters['Index']==j)]['Value'].values[0]
  productDemandDict[j] = dem
```

(continues on next page)

(continued from previous page)

```
  productInvDict[j] = inv
  productLossDict[j] = loss

print('Print productDemandDict')
for key in productDemandDict:
    print(key, ' : ', productDemandDict[key])
#print('\n')

print('Print productInvDict')
for key in productInvDict:
    print(key, ' : ', productInvDict[key])
#print('\n')

print('Print productLossDict')
for key in productLossDict:
    print(key, ' : ', productLossDict[key])
#print('\n')

scenarioTTRDict = {}
scenarioActiveNodeDict = {}
for n in scenarioList:
  ttr = inputParameters.loc[(inputParameters['Parameter']=='TTR') & (inputParameters[
↪'Index']==n)]['Value'].values[0]
  scenarioTTRDict[n] = ttr
  scenarioActiveNodeDict[n] = {}
  nActiveLinks = sum(inputScenarios[n].values)
  print(str(n)+": N. active links = ", nActiveLinks)

  for ind in range(len(inputScenarios.index)):
    supp = inputScenarios.iloc[ind]['Node']
    prod = inputScenarios.iloc[ind]['Product']
    scenarioActiveNodeDict[n][(supp,prod)] = inputScenarios[n].values[ind]
```

```
Print SupplierCapacityDict
S1  :  30
S2  :  45
S3  :  60
Print productDemandDict
P1  :  75
P2  :  50
Print productInvDict
P1  :  20
P2  :  25
Print productLossDict
P1  :  5000
P2  :  6500
Scenario_0: N. active links =  6
Scenario_1: N. active links =  4
Scenario_2: N. active links =  4
Scenario_3: N. active links =  4
```

4.2.2 Blocks 3 & 4: Optimization model & results

Since the risk exposure analytics framework requires solving one linear programming (LP) for each scenario at a time, we make use of the *for* loop to perform the procedure which create an LP which is loaded with the parameters for each scenario n and solve the model iteratively to obtain the optimal solution corresponding to that scenario. There are two optimizatiom models, i.e., TTR and TTS models.

Block 3A: TTR Model:

The first model is the best-response model based on the TTR. The components of the models are as follows. Note that these variables are created for each scenario but the scenario index n is dropped:

(i) **Decision variables:** consists of the following variables:

- model.**l**$[j]$ (or l_j): the number of unit loss of product *j* at supplier *i*;
- model.**y**$[i, j]$ (or y_{ij}) the amount of product *j* produced at supplier *i*;

(ii) **Objective function:** In the TTR model, we want to minimize the total profit loss due to disruption in each scenario, i.e.,

$$\min_{y,l} \sum_{j \in \mathcal{V}} f_j \cdot l_j$$

(iii) **Constraints:**

Constraint set 1: demand loss for product j **

This set of constraints helps determine the lower bound on the demand lost (l_j) during the disruption time (t^n) given the current inventory level (s_j) and the production amount for product j from supplier i (y_{ij}) from the active link $\alpha^n_{ij} = 1$. $\sum_{i \in \mathcal{A}} \left(\alpha^n_{ij} \cdot y_{ij}\right) + l_j \geq d_j \cdot t^n - s_j, \quad \forall j \in \mathcal{V}$

Constraint set 2: capacity at each supplier i

This set of constraints ensures that the cumulative production output of supplier i is within the capacity of that supplier if it is still active $\left(\alpha^n_{ij} = 1\right)$. Note that we need to check if at least one link α^n_{ij} for supplier i is active for that scenario, otherwise Pyomo will return an error. In the case when there is no active link, we will omit that supplier in the model (since that supplier will not be used anyway). This is done in the codes when we use the boolean *noSupply*.

$$\sum_{j \in \mathcal{V}} \alpha^n_{ij} \cdot y_{ij} \leq c_i \cdot t^{(n)}, \quad \forall i \in \mathcal{A}$$

Block 4: Solution and results

To aggregate the results from each scenario, we calculate and store the results in the **scenarioResults** DataFrame which stores the following results based on the optimal solution $(\mathbf{l}^{n,*}, \mathbf{y}^{n,*})$ for each scenario n : 'disruptionCost' $\left(\sum_j f_j \cdot l^{n,*}_j\right)$, 'riskExposureIndex' $\left(\frac{C^n}{C^{\max}}\right)$, 'lostUnits' $\left(\sum_j l^{n,*}_j\right)$, 'totalProduction' $\left(\sum_i \sum_j y^{n,*}_{ij}\right)$, 'activeLinks' (how many links are active under a given scenario)$, and 'utilizedLinks' (how many links are used in the solution of a given scenario).

Finally, we display the descriptive statistics of the results over all disruption scenarios. Here we make use of function *discribe()* in pandas (Link) and *hist()* in matplotlib (Link).

```
# TTR Model

disruptionCost = {}
disruptionResponse = {}
disruptionDemandLoss = {}

resultcolumns = ['disruptionCost', 'riskExposureIndex', 'lostUnits', 'totalProduction
↪', 'activeLinks','utilizedLinks']
TTRscenarioResults = pandas.DataFrame(columns = resultcolumns, index = scenarioList)

for n in scenarioList:
  # print("Analyzing:"+str(n))
  model = ConcreteModel()

  # Variables
  model.l = Var(productList, within = NonNegativeReals)
  model.y = Var(supplierList, productList, within = NonNegativeReals)

  model.demandLoss = ConstraintList()
  model.supplierCapacity = ConstraintList()

  # Objective function
  obj_expr = sum(productLossDict[j]*model.l[j] for j in productList)
  # print(obj_expr)
  model.OBJ = Objective(expr = obj_expr, sense = minimize)

  # Constraints 1 calculate demand loss for each product
  for j in productList:
    const_expr = sum(scenarioActiveNodeDict[n][(i,j)]*model.y[(i,j)] for i in␣
↪supplierList) + \
                  model.l[j] >= productDemandDict[j]*scenarioTTRDict[n] -␣
↪productInvDict[j]
    # print(const_expr)
    model.demandLoss.add(expr = const_expr)

  # Constraints 2 capacity at each supplier i
  for i in supplierList:
    # check if there is any left hand side (i.e., if there is at least one available␣
↪active link), otherwise pyomo will return an error
    noSupply = True
    for j in productList:
      if scenarioActiveNodeDict[n][(i,j)] > 0:
        noSupply = False
        break

    # we generate the constraint only if noSupply = False
    if not noSupply:
      const_expr = sum(scenarioActiveNodeDict[n][(i,j)]*model.y[(i,j)] for j in␣
↪productList) <= supplierCapacityDict[i]*scenarioTTRDict[n]
      #print(const_expr)
      model.supplierCapacity.add(expr = const_expr)

  # Solve the model
  opt = SolverFactory('glpk')
  opt.solve(model)

  # Save the result for each scenario
```

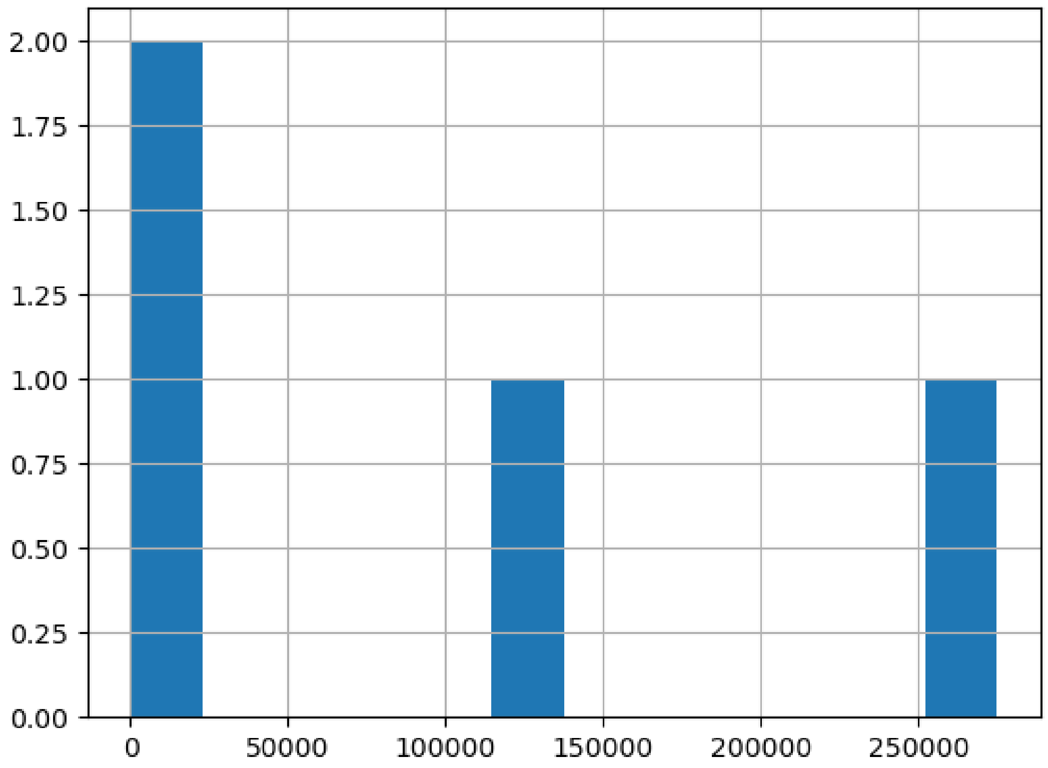
(continues on next page)

(continued from previous page)

```
  disruptionCost[n] = model.OBJ()
  TTRscenarioResults.loc[n, 'disruptionCost'] = disruptionCost[n]

  disruptionResponse[n] = {}
  nUtilizedLinks = 0
  for ind in range(len(inputScenarios.index)):
      i = inputScenarios.iloc[ind]['Node']
      j = inputScenarios.iloc[ind]['Product']

      if model.y[(i,j)].value is not None:
        disruptionResponse[n][(i,j)] = model.y[(i,j)].value
        if model.y[(i,j)].value > 0.1:
          nUtilizedLinks += 1
      else:
        disruptionResponse[n][(i,j)] = 0

  TTRscenarioResults.loc[n, 'totalProduction'] = sum(disruptionResponse[n].values())
  TTRscenarioResults.loc[n, 'activeLinks'] = sum(scenarioActiveNodeDict[n].values())
  TTRscenarioResults.loc[n, 'utilizedLinks'] = nUtilizedLinks

  disruptionDemandLoss[n] = {}
  for j in productList:
    disruptionDemandLoss[n][j] = model.l[j].value

  TTRscenarioResults.loc[n, 'lostUnits'] = sum(disruptionDemandLoss[n].values())

if TTRscenarioResults['disruptionCost'].max() > 0.001:
  TTRscenarioResults['riskExposureIndex'] = TTRscenarioResults['disruptionCost']/
 ↪TTRscenarioResults['disruptionCost'].max()
else: TTRscenarioResults['riskExposureIndex'] = 0.0

# Show the results
pandas.options.display.float_format = '{:,.3f}'.format
display(TTRscenarioResults)
```

```
            disruptionCost riskExposureIndex lostUnits totalProduction  \
Scenario_0           0.000             0.000     0.000         205.000
Scenario_1           0.000             0.000     0.000         205.000
Scenario_2     125,000.000             0.455    25.000         180.000
Scenario_3     275,000.000             1.000    55.000         150.000

            activeLinks utilizedLinks
Scenario_0            6             4
Scenario_1            4             3
Scenario_2            4             3
Scenario_3            4             3
```

```
# Here we make use of the function hist() to plot the histogram based on the analyzed␣
 ↪scenarios
TTRscenarioResults['disruptionCost'].astype(int).hist(bins=12)
plt.show()
```



Block 3B: TTS Model:

The second model is the time-to-survive (TTS) model to determine how long the disrupted supply chain can last until the demand is lost. The components of the models are as follows.

(i) **Decision variables:** consist of:

- model.**t**$[n]$ (or t^n): time-to-survive (TTS) for scenario n
- model.**y**$[i,j]$ (or y_{ij}) the amount of product *j* produced at supplier *i*;

(ii) **Objective function:** In the TTS model, we want to maximize the TTS in each scenario, i.e.,

$$\max_{y,t} t^n$$

(iii) **Constraints:**

Constraint set 1: demand of product j must be satisfied without loss**

This set of constraints helps determine how long the demand can be fully satisfied without loss (t^n) given the current inventory level (s_j) and the production amount for product j from supplier i (y_{ij}) if the link is active $\alpha_{ij}^n = 1$. $\$\sum_{i\in\mathcal{A}} \alpha_{ij}^n \cdot y_{ij} \geq d_j \cdot t^n - s_j, \quad \forall j \in \mathcal{V}\$$

Constraint set 2: capacity at each supplier i

This set of constraints ensures that the cumulative production output of supplier i is within the capacity of that supplier if it is still active $\left(\alpha_{ij}^n = 1\right)$. Note that we need to check if at least one link α_{ij}^n for supplier i is active for that scenario, otherwise Pyomo will return an error. In the case when there is no active link, we will omit that supplier in the model (since that supplier will not be used anyway). This is done in the codes when we use the boolean *noSupply*.

$$\sum_{j\in\mathcal{V}} \alpha_{ij}^n \cdot y_{ij} \leq c_i \cdot t^{(n)}, \quad \forall i \in \mathcal{A}$$

Block 4: Solution and results

To aggregate the results from each scenario, we calculate and store the results in the **scenarioResults** DataFrame which stores the following results based on the optimal solution $(t^{n,*}, \mathbf{y}^{n,*})$ for each scenario n : 'TTS' (t^n), 'totalProduction' $\left(\sum_i \sum_j y_{ij}^{n,*}\right)$, 'activeLinks' (how many links are active under a given scenario)$, and 'utilizedLinks' (how many links are used in the solution of a given scenario).

Finally, we display the descriptive statistics of the results over all disruption scenarios. Here we make use of function *discribe()* in pandas (Link) and *hist()* in matplotlib (Link).

```
# TTS Model

timeToSurvive = {}
disruptionResponse = {}
disruptionDemandLoss = {}

resultcolumns = ['TTS', 'TTR', 'nShortagePeriods', 'activeLinks','utilizedLinks']
TTSscenarioResults = pandas.DataFrame(columns = resultcolumns, index = scenarioList)

for n in scenarioList:
  # print("Analyzing:"+str(n))
  model = ConcreteModel()

  # Variables
  # There is only one variable t so we give a list of one value.
  # If the SC is robust for that scenario, the variable t will be equal to infinity
  # so we limit the max to a sufficiently large number (999 periods in this case)
  model.t = Var([0], within = NonNegativeReals, bounds=(0,999))
  model.y = Var(supplierList, productList, within = NonNegativeReals)

  model.demandSatisfaction = ConstraintList()
  model.supplierCapacity = ConstraintList()

  # Objective function
  obj_expr =model.t[0]
  # print(obj_expr)
  model.OBJ = Objective(expr = obj_expr, sense = maximize)

  # Constraints 1 demand must be fully satisfied
  for j in productList:
    const_expr = sum(scenarioActiveNodeDict[n][(i,j)]*model.y[(i,j)] for i in␣
↪supplierList) \
                    >= productDemandDict[j]*model.t[0] - productInvDict[j]
    # print(const_expr)
    model.demandSatisfaction.add(expr = const_expr)

  # Constraints 2 capacity at each supplier i
  for i in supplierList:
    # check if there is any left hand side (i.e., if there is at least one available␣
↪active link), otherwise pyomo will return an error
    noSupply = True
    for j in productList:
      if scenarioActiveNodeDict[n][(i,j)] > 0:
        noSupply = False
        break

    # we generate the constraint only if noSupply = False
```

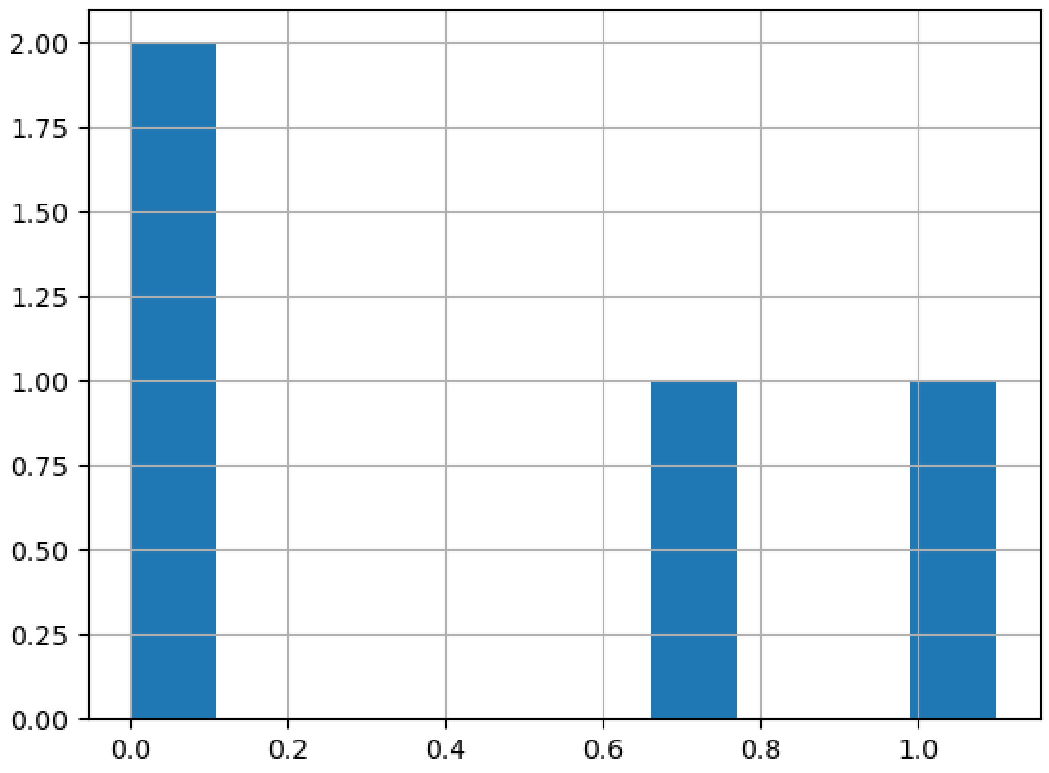
(continues on next page)

(continued from previous page)

```
    if not noSupply:
      const_expr = sum(scenarioActiveNodeDict[n][(i,j)]*model.y[(i,j)] for j in␣
↪productList) <= supplierCapacityDict[i]*model.t[0]
      #print(const_expr)
      model.supplierCapacity.add(expr = const_expr)

  # Solve the model
  opt = SolverFactory('glpk')
  opt.solve(model)

  # Save the result for each scenario

  timeToSurvive[n] = model.OBJ()
  TTSscenarioResults.loc[n, 'TTS'] = round(timeToSurvive[n],3)
  TTSscenarioResults.loc[n, 'TTR'] = scenarioTTRDict[n]

  disruptionResponse[n] = {}
  nUtilizedLinks = 0
  for ind in range(len(inputScenarios.index)):
      i = inputScenarios.iloc[ind]['Node']
      j = inputScenarios.iloc[ind]['Product']

      if model.y[(i,j)].value is not None:
        disruptionResponse[n][(i,j)] = model.y[(i,j)].value
        if model.y[(i,j)].value > 0.1:
          nUtilizedLinks += 1
      else:
        disruptionResponse[n][(i,j)] = 0

  TTSscenarioResults.loc[n, 'nShortagePeriods'] = max(TTSscenarioResults.loc[n, 'TTR
↪'] - TTSscenarioResults.loc[n, 'TTS'], 0)
  TTSscenarioResults.loc[n, 'activeLinks'] = sum(scenarioActiveNodeDict[n].values())
  TTSscenarioResults.loc[n, 'utilizedLinks'] = nUtilizedLinks

# Show the results
pandas.options.display.float_format = '{:,.3f}'.format
display(TTSscenarioResults)
```

```
               TTS TTR nShortagePeriods activeLinks utilizedLinks
Scenario_0 999.000   2                0           6             4
Scenario_1   2.250   2                0           4             3
Scenario_2   1.286   2            0.714           4             3
Scenario_3   0.900   2            1.100           4             3
```

```
# Intepret and show the results
# Here we make use of the function hist() to plot the histogram
TTSscenarioResults['nShortagePeriods'].hist()
plt.show()

display(TTSscenarioResults)
```



```
                TTS TTR nShortagePeriods activeLinks utilizedLinks
Scenario_0 999.000   2                0           6             4
Scenario_1   2.250   2                0           4             3
Scenario_2   1.286   2            0.714           4             3
Scenario_3   0.900   2            1.100           4             3
```

Part II

Additional Examples of Analytics Techniques in Supply Chains

CHAPTER

FIVE

(R, S, S) INVENTORY SIMULATION

The (R, s, S) inventory policy is a type of periodic review inventory control system. It involves three key decision variables:

- Review Period (R): The time interval between inventory checks. **- Reorder Point (s):** The level of inventory at which a replenishment order is placed. **- Target level (S):** The target level of inventory position to be reached when an order is placed.

How it Works:

Inventory Review: Inventory levels are checked at regular intervals (R). Order Placement: If the inventory level is below or equal to the reorder point (s), an order is placed until the inventory position (current inventory plus incoming order) reaches the level S. Replenishment: The ordered items are received and added to inventory.

The (R, s, S) policy is widely adopted due to its simplicity, practicality, and effectiveness as it is easy to understand and implement. It can be adapted to various inventory situations and reduces the need for constant inventory monitoring.

Optimizing an (R, s, S) inventory policy, a common inventory control method, can be challenging due to several factors:

- Uncertainty: Demand is often uncertain, making it difficult to predict future needs accurately. This uncertainty can lead to stockouts or excess inventory. Interdependencies: Inventory decisions are often interconnected with other operational factors, such as production capacity, transportation costs, and customer service levels. Optimizing one factor may negatively impact another. **- Complexity:** The (R, s, S) policy involves three decision variables: review period (R), reorder point (s), and target level (S). Finding the optimal combination of these variables can be computationally intensive. **- Multiple Objectives:** Inventory management often involves competing objectives, such as minimizing costs, maximizing service levels, and reducing stockouts. Balancing these objectives can be challenging.

Thus, simulation can be a valuable tool for addressing these challenges:

- Modeling Uncertainty: Simulation can model demand uncertainty using probability distributions, allowing for a better understanding of potential outcomes. **- Testing Different Scenarios:** By running multiple simulations with different parameter values, decision-makers can evaluate the impact of various (R, s, S) policies on performance metrics like costs, service levels, and stockouts. **- Identifying Trade-offs:** Simulation can help identify trade-offs between different objectives, allowing decision-makers to make informed choices. **- Optimization:** Simulation can be integrated with optimization algorithms to find near-optimal (R, s, S) policies.

In summary, simulation offers a flexible and powerful approach to optimizing (R, s, S) inventory policies by providing insights into uncertain demand, evaluating the impact of different policies, and helping to balance competing objectives.

This notebook provides an example of the use of `DataFrame` to calculate the inventory performance based on estimated demand. The objective for you to review this is to *understand* what each block of codes does. Developing it by yourself requires time and experience but if you generally understand the process and codes, that would suffice.

CHAPTER

SIX

INVENTORY SIMULATION USING PANDAS

In this exercise, you will use pandas library to simulate different inventory policies for a planning horizon of one year.

6.1 Step 1: Reading data

Import the pandas library under the alias `pd`. Import the csv file `DemandSimulation.csv` into a `DataFrame`, and select the first column named `'Week'` as the index of the `DataFrame`.

Hint: you can define the index column of the `DataFrame` using the `index_col` parameter of the `.read_csv()` function. Check this page for more information.

```
import pandas as pd

# There is a csv file which I put in online. You can also download but we will load␣
↪directly from the link.
url = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/S04_Data_
↪Structures_2/DemandScenarios.csv'
demand_scenario_df = pd.read_csv(url, index_col='Week')
```

Now display the first 10 rows of your `DataFrame`. You can see that the file contains first column as indexes (weeks) and each of the subsequent column contains a list of estimated demands for each scenario.

```
demand_scenario_df.head(10)
```

```
      Scenario1  Scenario2  Scenario3
Week
1            18         21         17
2            16         16         15
3            13         13         11
4            17         17         16
5            11         11         10
6            20         23         18
7            15         17         13
8            20         20         17
9            15         16         13
10           10         10          9
```

6.2 Step 2: Simulation and output a `DataFrame` object

Now, we will simulate different inventory policies based on the inventory model (s, S) for a specific demand scenario and analyze/visualize the performance based on a given policy based on that specific demand scenario.

> The inventory model min-max or (s, S) is one of the most commonly used inventory system in practice. Here are the details of the inventory system. In this one, the min/max is applied to periodic system where the inventory is checked periodically every period (e.g., once a day or once a week) (Note that this sytem can also be applied in a real-time ordering system but in practice orders would be placed periodically).
>
> - There are two parameters that must be determined to control the inventory replenishment: s which represents the minimum inventory level (or reorder point), and S which represents the desired target inventory level.
> - **At the beginning of each period**, this system constantly checks the inventory position (current inventory + in-transit inventory) for the product.
> - If the inventory position drops below the min level s, the systems create an order refill inventory to raise the inventory position to the max level S. The order quantity Q is then Q = S - inventory_position (recall that inventory position = current + in-transit). This order quantity arrives after the lead time.

In this exercise you will create a function which simulates your inventory performance for a given (s, S) policy. This function should return a `DataFrame` object containing inventory levels (at the begining and at the end of each period), orders to place and order to receive at each time period (week) of the simulated planning horizon. Here there are some instructions.

The function will require the following input parameters:

- The `Series` from of one scenario from the DataFrame, which contains the simulated demand data for that specific scenario
- (number) initial inventory level at the begining of the planning horizon
- (number) lead time in weeks
- (number) s value, i.e., minimum inventory level
- (number) S value, i.e., maximun inventory level

The output of your function should be a `DataFrame` with 52 rows (52 weeks) and four columns as follows:

- `'Start_Inv'`: inventory level at the beginning of each week
- `'Receipt'`: units receipt at each week
- `'Ending_Inv'`: inventory level at the end of each week
- `'Order'`: order quantity in units

Note that the inventory level can be negative if you have a backlog (demand is more than available quantity). In that case, the negative value is carried forward to a subsequent period to be satistied.

Hints:

- *We use the method `df.index` to access the list of indexes of the `DataFrame` you created in the previous exercises.*
- *We initialize the first row of the `'Start_Inv'` column as the initial inventory level given as input. You can also initialize the `'Receipt'` entries as 0 from the beginning of the planning horizon until the time period equals to the lead time.*
- *We then use a `for` loop to iterate over the planning horizon and compute the corresponding values for `'Start_Inv'`, `'Receipt'`, `'Ending_Inv'`, `'Order'`.*

Here we provide the remaining inputs required for the simulation

```
# Parameters for the model
init_inv = 27  # initial inventory level
lead_time = 2  # leadtime

# Parameters to set (s,S) policy
s_min =  25 # this is the small s
s_max =  70 # this is the big s

demand_series = demand_scenario_df['Scenario1'] # we are taking the first scenario␣
↪here
demand_series.head()
```

```
Week
1     18
2     16
3     13
4     17
5     11
Name: Scenario1, dtype: int64
```

We can create an empty DataFrame to store the results

```
# creating an (empty) dataframe to keep the results
column_names = ['Start_Inv', 'Receipt', 'Demand', 'Ending_Inv', 'Order']
n_weeks = len(demand_series.index)
inv_dataframe = pd.DataFrame(index = range(1, n_weeks+1), columns=column_names)
inv_dataframe.index.name ='Week' # give the index name to the output dataframe
inv_dataframe.head()
```

```
     Start_Inv Receipt Demand Ending_Inv Order
Week
1          NaN     NaN    NaN        NaN   NaN
2          NaN     NaN    NaN        NaN   NaN
3          NaN     NaN    NaN        NaN   NaN
4          NaN     NaN    NaN        NaN   NaN
5          NaN     NaN    NaN        NaN   NaN
```

You can first initialize the initial inventory and also set the order receipts of the period < lead time to be zero (if we order now, the earliest arrival will be after the leadtime). You can also set the demand column too.

```
# initializing current inventory levels and order receipt up to t=lead_time
inv_dataframe.loc[1, 'Start_Inv'] = init_inv
for t in range(1, lead_time+1):
    inv_dataframe.loc[t,'Receipt']=0

# add the values to demand column from the series
inv_dataframe['Demand'] = demand_series
inv_dataframe.head() # review the result
```

```
     Start_Inv Receipt  Demand Ending_Inv Order
Week
1           27       0      18        NaN   NaN
2          NaN       0      16        NaN   NaN
3          NaN     NaN      13        NaN   NaN
```

(continues on next page)

(continued from previous page)

```
4          NaN      NaN      17         NaN    NaN
5          NaN      NaN      11         NaN    NaN
```

Now at each iteration, we need to calculate the ending inventory at Inv_t from the starting inventory from the previous period Inv_{t-1}, quantity received and demand using the following flow conservation.

$Inv_t = Inv_{t-1} + Receipt_t - Demand_t$

Again, inventory can be negative if you have a backlog.

Note: please examine the following code to understand the process. It is fine if you cannot develop it from scratch. The objective of this is to show how the process/code looks like. We try to put the comments to be as explicit as possible.

```
# assign values to the entries of the dataframe
for t in range(1, n_weeks+1):
    # computing inventory levels at the begining and end of each period
    if t > 1: # set the Start_Inv equals the inv level at the end of the previous␣
↪period (Ending_Inv)
        inv_dataframe.loc[t, 'Start_Inv'] = inv_dataframe.loc[t-1, 'Ending_Inv']

    # Calculate inventory flow for Ending_Inv of the current period
    inv_dataframe.loc[t, 'Ending_Inv'] = inv_dataframe.loc[t, 'Start_Inv'] + inv_
↪dataframe.loc[t,'Receipt'] - inv_dataframe.loc[t, 'Demand']

    # Calculating orders and receipts for period t
    intransit_quantity = sum(inv_dataframe.loc[t+1:t+lead_time-1,'Receipt']) # check␣
↪the intransit (outstanding) order within the leadtime

    if inv_dataframe.loc[t,'Ending_Inv'] + intransit_quantity < s_min: # if 'Ending_
↪Inv' + intransit < s_min in this case an order must be placed
        # create an order Q = s_max - current inventory position
        inv_dataframe.loc[t,'Order'] = s_max - (inv_dataframe.loc[t,'Ending_Inv']+␣
↪intransit_quantity)
        # check if we reach the end of dataframe, if not, we add the order receipt␣
↪after the leadtime
        if t + lead_time <= max(inv_dataframe.index):
            inv_dataframe.loc[t+lead_time,'Receipt'] = inv_dataframe.loc[t,'Order']
    else:
        inv_dataframe.loc[t,'Order'] = 0    # no order is placed if 'Ending_Inv' >= s_
↪min
        if t + lead_time <= max(inv_dataframe.index):
            inv_dataframe.loc[t+lead_time,'Receipt'] = 0  # no orders receipt in␣
↪t+leadtime periods
```

Now you can examine the results.

```
print("(s,S) = (",s_min,",",s_max,")")
inv_dataframe
```

```
(s,S) = ( 25 , 70 )
```

```
     Start_Inv Receipt  Demand Ending_Inv Order
Week
1           27       0      18          9    61
2            9       0      16         -7     0
```

(continues on next page)

(continued from previous page)

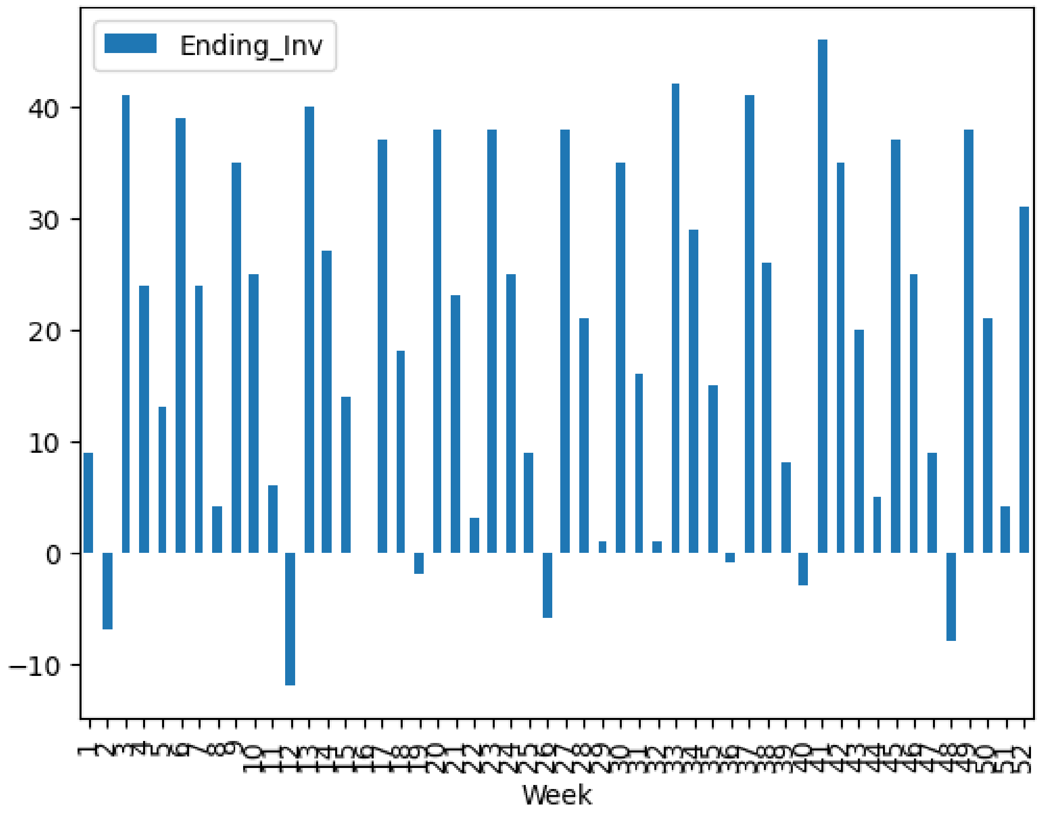
```
3           -7      61      13          41     0
4           41       0      17          24    46
5           24       0      11          13     0
6           13      46      20          39     0
7           39       0      15          24    46
8           24       0      20           4     0
9            4      46      15          35     0
10          35       0      10          25     0
11          25       0      19           6    64
12           6       0      18         -12     0
13         -12      64      12          40     0
14          40       0      13          27     0
15          27       0      13          14    56
16          14       0      14           0     0
17           0      56      19          37     0
18          37       0      19          18    52
19          18       0      20          -2     0
20          -2      52      12          38     0
21          38       0      15          23    47
22          23       0      20           3     0
23           3      47      12          38     0
24          38       0      13          25     0
25          25       0      16           9    61
26           9       0      15          -6     0
27          -6      61      17          38     0
28          38       0      17          21    49
29          21       0      20           1     0
30           1      49      15          35     0
31          35       0      19          16    54
32          16       0      15           1     0
33           1      54      13          42     0
34          42       0      13          29     0
35          29       0      14          15    55
36          15       0      16          -1     0
37          -1      55      13          41     0
38          41       0      15          26     0
39          26       0      18           8    62
40           8       0      11          -3     0
41          -3      62      13          46     0
42          46       0      11          35     0
43          35       0      15          20    50
44          20       0      15           5     0
45           5      50      18          37     0
46          37       0      12          25     0
47          25       0      16           9    61
48           9       0      17          -8     0
49          -8      61      15          38     0
50          38       0      17          21    49
51          21       0      17           4     0
52           4      49      22          31     0
```

6.3 Step 3: Plotting data

We can also visualize the data directly from the dataframe by taking on column.

```
inv_dataframe[["Ending_Inv"]].plot(kind='bar')    # plotting inv levels
```

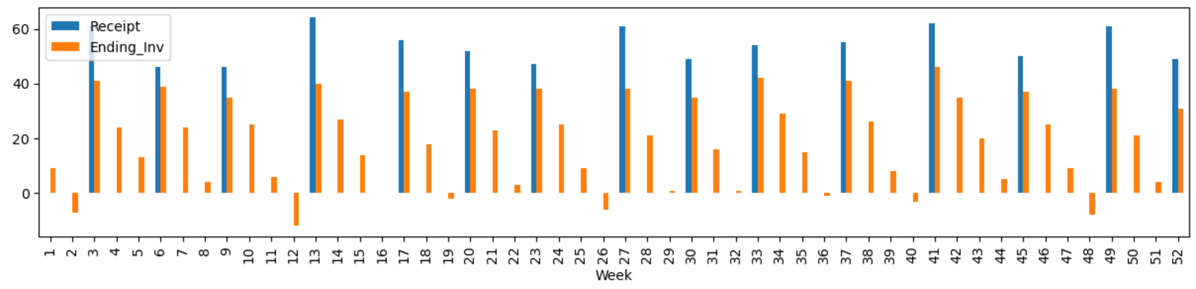
```
<Axes: xlabel='Week'>
```



This can be done even for more than two columns.

```
inv_dataframe[["Receipt","Ending_Inv"]].plot(kind='bar', figsize=(15,3))    # plotting␣
↪inv and receipts and resize the plot
```

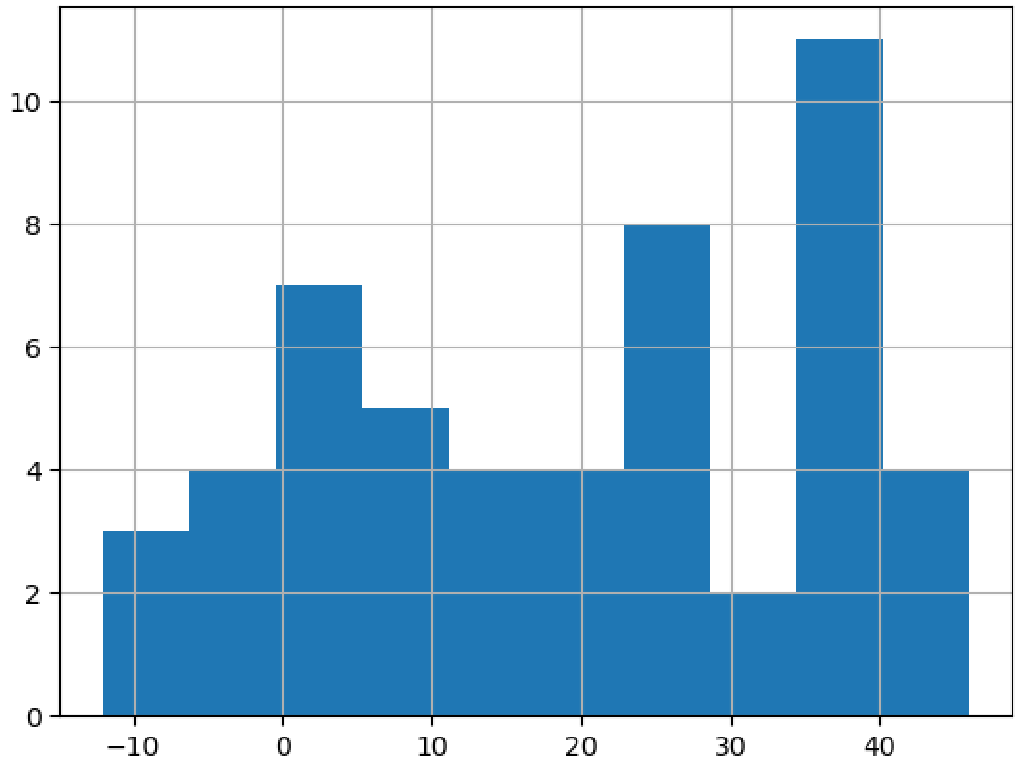
```
<Axes: xlabel='Week'>
```



or histogram

```
inv_dataframe['Ending_Inv'].hist()
```

```
<Axes: >
```



6.4 Step 4: Summarizing the results

We can also use the functions in DataFrame to summarize the results. See https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.lt.html for more details.

```
backlog_num = inv_dataframe['Ending_Inv'].lt(0).sum()
backlog_total_units = inv_dataframe['Ending_Inv'].loc[inv_dataframe['Ending_Inv'] <␣
↪0].sum() #this number is negative
demand_total = inv_dataframe['Demand'].sum()
order_total = inv_dataframe['Order'].sum()
order_num = inv_dataframe['Order'].gt(0).sum()
inv_total_units = inv_dataframe['Ending_Inv'].loc[inv_dataframe['Ending_Inv'] > 0].
↪sum()

print("Performance of (s,S) = (",s_min,",",s_max,") under demand scenartio:",demand_
↪series.name)
print("Total number of orders =",order_num,", Average order quantity =","{:.2f}".
↪format(order_total/order_num))
print("Average inventory per week =", "{:.2f}".format(inv_total_units/n_weeks))
print("Average number of backlogged units per week =", "{:.2f}".format(backlog_total_
↪units/n_weeks))
```

(continues on next page)

(continued from previous page)

```
print("N. backlog incidents =", inv_dataframe['Ending_Inv'].lt(0).sum())
print("Service Level =", "{:.2f}".format(( 1 - backlog_num/n_weeks)*100),"%") # % of␣
↪times the demand is completely fulfilled
print("Fill rate =", "{:.2f}".format(((demand_total + backlog_total_units)/demand_
↪total)*100),"%")  # % of quantity fulfilled on-time
```

```
Performance of (s,S) = ( 25 , 70 ) under demand scenartio: Scenario1
Total number of orders = 15 , Average order quantity = 54.20
Average inventory per week = 19.92
Average number of backlogged units per week = -0.75
N. backlog incidents = 7
Service Level = 86.54 %
Fill rate = 95.18 %
```

6.5 Playing with the code:

You can now repeat the process above with a different demand scenario (e.g., choosing the ‘Scenario2’ or ‘Scenario3’ and vary the inventory control parameters `s_min` and `s_max` and explore the results.

CHAPTER

SEVEN

DEMO: BASS DIFFUSION MODEL

The Bass diffusion model is a mathematical model used to predict the adoption rate of a new product over time. It is based on the idea that the rate of adoption of a new product is influenced by both external factors (like marketing and advertising) and internal factors (like word-of-mouth and social influence).

The model has several key properties:

- **S-shaped curve:** The adoption rate typically follows an S-shaped curve, with a slow start, a rapid growth period, and then a gradual decline.
- **Diminishing returns:** As more people adopt the product, the rate of adoption slows down.
- **Threshold effect:** There is a threshold level of adoption above which the product becomes self-sustaining, and the adoption rate accelerates.

The Bass diffusion model has been widely used in marketing, economics, and sociology to predict the adoption of new products and technologies. It has also been applied to other areas, such as forecasting the spread of diseases, the adoption of new policies, and the diffusion of innovations in agriculture.

While the Bass diffusion model is a powerful tool for predicting adoption rates, it is important to note that it has limitations. For example, it assumes that all potential adopters are identical, which may not be the case in reality. Additionally, the model does not account for external factors, such as economic conditions or changes in consumer preferences, that can affect adoption rates.

Despite these limitations, the Bass diffusion model remains a valuable tool for understanding and predicting the adoption of new products. By understanding the factors that influence adoption rates, businesses can develop more effective marketing strategies and make better decisions about product development and launch.

$$F(t) = \frac{1 - e^{-(p+q)t}}{1 + \frac{q}{p}e^{-(p+q)t}}$$

where

- p is the coefficiant of innovation
- q is the coefficient of imitation

In addition, the probability of adoption by an individual at time t which is represented by $f(t)$ (recall that $f(t) = \frac{d}{dt}F(t)$) can be calculated as follows:

\$$f(t) = \frac{d}{dt}F(t) = \frac{e^{((p+q)t)}p(p+q)^2}{[pe^{((p+q)t)}+q]^2}$\$.

Consequently, the estimated number of adoptions in each period can be calculated as $s(t) = m \times f(t)$

We have then prepared the following two functions to calculate $F(t)$ and $f(t)$.

```
import math
def Bass_cumulative_probability_Ft(p, q, t):
    return (1-math.exp(-(p+q)*t))/(1+(q/p)*math.exp(-(p+q)*t))

def Bass_probability_ft(p, q, t):
    return (math.exp((p+q)*t)*p*(p+q)**2)/(p*math.exp((p+q)*t)+q)**2
```

Based on the values of p, q and m of the quarterly sales of iPhone provided in the code cell below, please compute the values of the estimated probability of adoption $f(t)$, cumulative probability of adoption $F(t)$, the number of adoptions $s(t) = m \times f(t)$, and cumulative number of adoptions $S(t) = m \times F(t)$ for each quarter, which is indexed by t. Note that the first column below is an index whereas the subsequent four columns contain the corresponding values of the estimations based on Bass diffusion model.

Quarter_Index	ft	Ft	st	St
…	…	…	…	…

7.1 Regression for Bass parameters estimation

Estimating Bass diffusion model parameters

We can now complete the Bass diffusion model pipeline by first estimating the parameters p, q and m from the data prior to performing prediction. First we load the modules which are required as well as the actual data of sales of iPhone. The regression model for parameters estimation is presented in the paper below (Section "Discrete Analogue").

Reference: Bass, F. M. (1969). A new product growth for model consumer durables. Management science, 15(5), 215-227.

```
import pandas as pd
import numpy as np
import sklearn
from sklearn import *

url = 'https://raw.githubusercontent.com/acedesci/scanalytics/master/EN/S04_Data_
 ↪Structures_2/iphone_quarter_sales.csv'
actual_sales = pd.read_csv(url, index_col='Quarter')
actual_sales.head()
```

```
          Sales
Quarter
0         0.270
1         1.119
2         2.315
3         1.703
4         0.717
```

We would first determine the coefficients for the Bass diffusion model in order to perform the prediction as seen in the previous session. The Bass model in a basic form, can be written as:

$s(t) = pm + (q - p)S(t) - \frac{q}{m}S(t)^2$

which can be solved by a regression function. Here we will use the following equation (note that we do not shift one period like in the paper since the period is a bit long).

$s(t) = a + bS(t) + cS(t)^2$

Correspondingly, we can obtain $m = \frac{-b \pm \sqrt{b^2 - 4ac}}{2c}$ (m must be positive), $p = a/m$ and $q = -mc$ (for the detailed proof, you can review the Section “Discrete Analogue” in the paper).

In the steps below, we demonstrate how this can be done using the linear regression model.

```
input = pd.DataFrame()
# we need to prepare the inputs from sales by taking the cumulative sum up until␣
↪period T - 1

input["sales"] = actual_sales["Sales"]
input["cumulative_sales"] = np.cumsum(actual_sales["Sales"])
input["cumulative_sales_sq"] = [x**2 for x in input["cumulative_sales"]]
input.head()
```

```
          sales  cumulative_sales  cumulative_sales_sq
Quarter
0         0.270             0.270             0.072900
1         1.119             1.389             1.929321
2         2.315             3.704            13.719616
3         1.703             5.407            29.235649
4         0.717             6.124            37.503376
```

Now we can use linear regression to determine a, b and c of the regression function above.

```
X = input[["cumulative_sales","cumulative_sales_sq"]]
y = input["sales"]
reg = sklearn.linear_model.LinearRegression().fit(X,y)
print("intercept: "+str(reg.intercept_))
print("coefficients: "+str(reg.coef_))

a = reg.intercept_
b = reg.coef_[0]
c = reg.coef_[1]
```

```
intercept: 3.6726987894801724
coefficients: [ 1.15612351e-01 -6.23837858e-05]
```

Following the results, we can obtain

$m = \frac{-b \pm \sqrt{b^2 - 4ac}}{2c}$

and m (the size of potential customer base) must be positive.

```
# determine m
m1 = (-b+math.sqrt(b**2-4*a*c))/(2*c)
m2 = (-b-math.sqrt(b**2-4*a*c))/(2*c)
m = max(m1,m2)
print("m1 = ", m1, ", m2 = ", m2, "m = ", m)
```

```
m1 =  -31.24072191263632 , m2 =  1884.4843034859591 m =  1884.4843034859591
```

Then, we can directly obtain $p = a/m$ and $q = -mc$ for the Bass diffusion model.

```
p = a/m
q = -m*c
print("m = ", m, ", p = ", p, "q = ", q)
```

```
m =  1884.4843034859591 , p =  0.0019489145028623142 q =  0.11756126506137607
```

Below are some examples of how different outputs can be generated by the Bass diffusion model.

Question 1: Please compute the values as indicated in the table above for the first 40 quarters of the sales (i.e., 10 years) using the parameters provided. The index of the quarter must be from $0 \rightarrow 39$ (i.e., the first quarter is considered period $t = 0$).

```
import pandas as pd
column_names = ['ft','Ft','st','St']
quarter_index = list(range(40))
bass_df = pd.DataFrame(index = quarter_index, columns = column_names)
bass_df.index.name = 'Quarter'

### start your code here ###
bass_df['ft'] = [Bass_probability_ft(p, q, t) for t in bass_df.index]
bass_df['Ft'] = [Bass_cumulative_probability_Ft(p, q, t) for t in bass_df.index]
bass_df['st'] = [m*bass_df.at[t,'ft'] for t in bass_df.index]
bass_df['St'] = [m*bass_df.at[t,'Ft'] for t in bass_df.index]
### end your code here ###

bass_df
```

```
               ft        Ft         st          St
Quarter
0        0.001949  0.000000   3.672699    0.000000
1        0.002187  0.002066   4.121845    3.893112
2        0.002453  0.004384   4.623538    8.261201
3        0.002751  0.006983   5.183301   13.159540
4        0.003082  0.009896   5.807073   18.649135
5        0.003450  0.013159   6.501194   24.797131
6        0.003859  0.016809   7.272376   31.677206
7        0.004313  0.020892   8.127653   39.369912
8        0.004815  0.025452   9.074300   47.962967
9        0.005370  0.030540  10.119730   57.551440
10       0.005981  0.036210  11.271350   68.237826
11       0.006652  0.042522  12.536367   80.131944
12       0.007387  0.049536  13.921550   93.350623
13       0.008189  0.057319  15.432937  108.017124
14       0.009061  0.065939  17.075473  124.260231
15       0.010004  0.075465  18.852590  142.212953
16       0.011019  0.085971  20.765728  162.010779
17       0.012106  0.097528  22.813792  183.789418
18       0.013262  0.110206  24.992575  207.681977
19       0.014484  0.124074  27.294161  233.815550
20       0.015764  0.139193  29.706331  262.307214
21       0.017093  0.155618  32.212039  293.259450
22       0.018461  0.173392  34.788976  326.755096
23       0.019851  0.192547  37.409319  362.851933
24       0.021247  0.213097  40.039705  401.577141
25       0.022628  0.235036  42.641499  442.921864
```

(continues on next page)

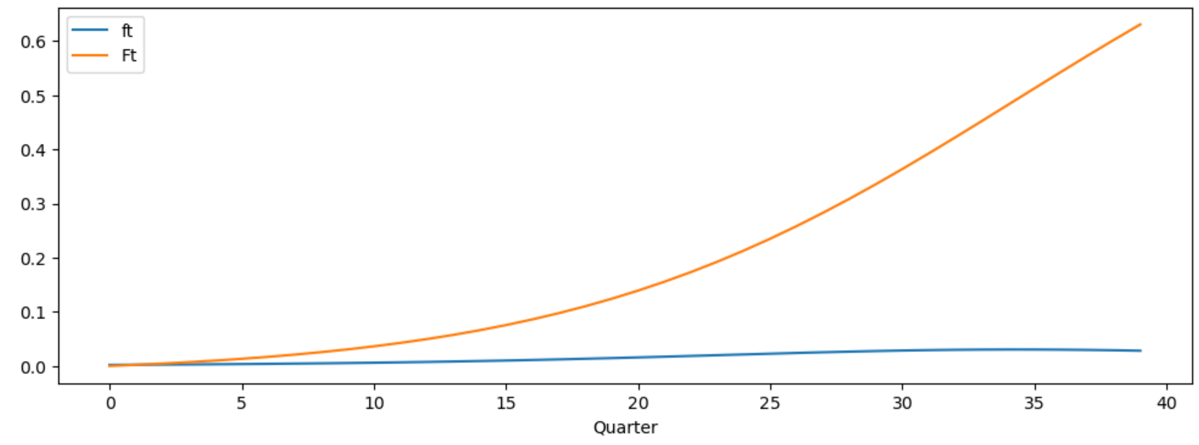
(continued from previous page)

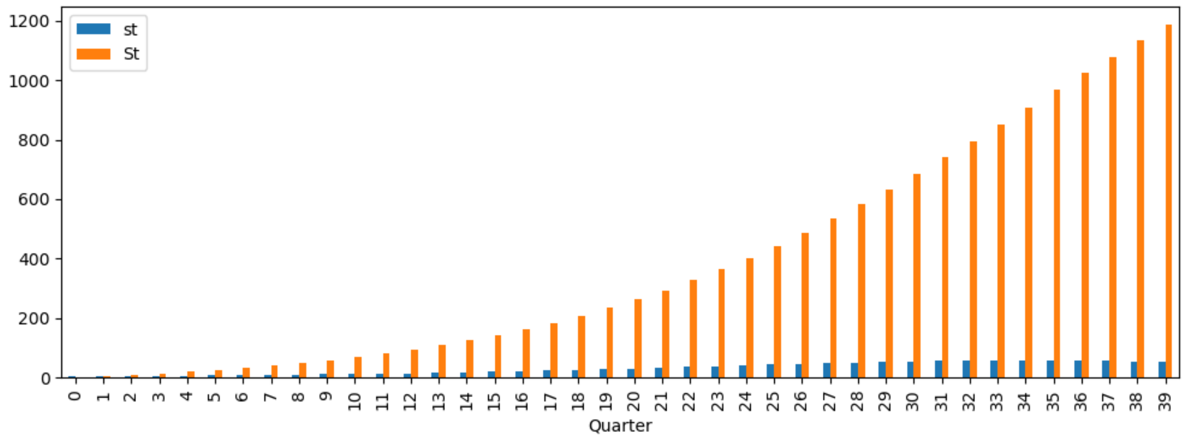
```
26        0.023970  0.258339  45.171429   486.836225
27        0.025250  0.282955  47.582589   533.225157
28        0.026440  0.308809  49.825855   581.945444
29        0.027515  0.335797  51.851650   632.804359
30        0.028449  0.363792  53.611996   685.560219
31        0.029219  0.392641  55.062732   739.925081
32        0.029804  0.422168  56.165740   795.569684
33        0.030189  0.452182  56.890996   852.130535
34        0.030363  0.482476  57.218267   909.218892
35        0.030320  0.512836  57.138295   966.431192
36        0.030063  0.543045  56.653351  1023.360364
37        0.029598  0.572893  55.777094  1079.607339
38        0.028938  0.602176  54.533767  1134.792093
39        0.028101  0.630710  52.956791  1188.563574
```

Question 2: Please (i) plot the columns `['ft','Ft']` in a single plot using line plot (parameter `kind='line'`) and (ii) plot the columns `['st','St']` in a single plot using bar plot (parameter `kind='bar'`). You can also indicate the plot size using the parameter `figsize = (12,4)`.

```
### start your code here ###
bass_df[['ft','Ft']].plot(figsize = (12,4))
bass_df[['st','St']].plot(kind = 'bar', figsize = (12,4))
### end your code here ###
```

```
<Axes: xlabel='Quarter'>
```





Now we can compare the results with the actual quarterly sales of iPhone for the first 40 quarters (starting from Q2 2007 which the product was first introduced).

Question 3: Please add the following columns and their corresponding values to the `bass_df` DataFrame

- Column `'Actual'`: which contains the actual sales from the DataFrame `actual_sales`
- Column `'CumulativeActual'`: which calculates the sum from the first quarter up until each quater in the index
- Column `'PError'`: which calculates the percentage error of each corresponding quarter from column `'Actual'` and column `'st'` (estimated sales in each quarter), i.e., `PError = (Actual-st)/Actual`

```
### start your code here ###
bass_df['Actual'] = actual_sales['Sales']
bass_df['CumulativeActual'] = [sum(actual_sales.loc[:t,'Sales']) for t in actual_
↪sales['Sales'].index]
bass_df['PError'] = (bass_df['Actual'] - bass_df['st'])/bass_df['Actual']
### end your code here ###
bass_df
```

```
                ft        Ft         st          St  Actual  CumulativeActual  \
Quarter
0         0.001949  0.000000   3.672699    0.000000   0.270             0.270
1         0.002187  0.002066   4.121845    3.893112   1.119             1.389
2         0.002453  0.004384   4.623538    8.261201   2.315             3.704
3         0.002751  0.006983   5.183301   13.159540   1.703             5.407
4         0.003082  0.009896   5.807073   18.649135   0.717             6.124
5         0.003450  0.013159   6.501194   24.797131   6.892            13.016
6         0.003859  0.016809   7.272376   31.677206   4.363            17.379
7         0.004313  0.020892   8.127653   39.369912   3.793            21.172
8         0.004815  0.025452   9.074300   47.962967   5.208            26.380
9         0.005370  0.030540  10.119730   57.551440   7.367            33.747
10        0.005981  0.036210  11.271350   68.237826   8.737            42.484
11        0.006652  0.042522  12.536367   80.131944   8.752            51.236
12        0.007387  0.049536  13.921550   93.350623   8.398            59.634
13        0.008189  0.057319  15.432937  108.017124  14.102            73.736
14        0.009061  0.065939  17.075473  124.260231  16.235            89.971
15        0.010004  0.075465  18.852590  142.212953  18.647           108.618
16        0.011019  0.085971  20.765728  162.010779  20.338           128.956
17        0.012106  0.097528  22.813792  183.789418  17.073           146.029
18        0.013262  0.110206  24.992575  207.681977  37.044           183.073
```

(continues on next page)

(continued from previous page)

```
19        0.014484  0.124074  27.294161   233.815550  35.064             218.137
20        0.015764  0.139193  29.706331   262.307214  26.028             244.165
21        0.017093  0.155618  32.212039   293.259450  26.910             271.075
22        0.018461  0.173392  34.788976   326.755096  47.789             318.864
23        0.019851  0.192547  37.409319   362.851933  37.430             356.294
24        0.021247  0.213097  40.039705   401.577141  31.241             387.535
25        0.022628  0.235036  42.641499   442.921864  33.797             421.332
26        0.023970  0.258339  45.171429   486.836225  51.025             472.357
27        0.025250  0.282955  47.582589   533.225157  43.719             516.076
28        0.026440  0.308809  49.825855   581.945444  35.203             551.279
29        0.027515  0.335797  51.851650   632.804359  39.272             590.551
30        0.028449  0.363792  53.611996   685.560219  74.468             665.019
31        0.029219  0.392641  55.062732   739.925081  61.170             726.189
32        0.029804  0.422168  56.165740   795.569684  47.534             773.723
33        0.030189  0.452182  56.890996   852.130535  48.050             821.773
34        0.030363  0.482476  57.218267   909.218892  74.780             896.553
35        0.030320  0.512836  57.138295   966.431192  51.200             947.753
36        0.030063  0.543045  56.653351  1023.360364  40.399             988.152
37        0.029598  0.572893  55.777094  1079.607339  45.513            1033.665
38        0.028938  0.602176  54.533767  1134.792093  78.290            1111.955
39        0.028101  0.630710  52.956791  1188.563574  50.763            1162.718

            PError
Quarter
0       -12.602588
1        -2.683508
2        -0.997209
3        -2.043629
4        -7.099125
5         0.056704
6        -0.666829
7        -1.142803
8        -0.742377
9        -0.373657
10       -0.290071
11       -0.432400
12       -0.657722
13       -0.094379
14       -0.051769
15       -0.011025
16       -0.021031
17       -0.336250
18        0.325327
19        0.221590
20       -0.141322
21       -0.197029
22        0.272030
23        0.000553
24       -0.281640
25       -0.261695
26        0.114720
27       -0.088373
28       -0.415387
29       -0.320321
30        0.280067
31        0.099841
```

(continues on next page)

(continued from previous page)

```
32       -0.181591
33       -0.183996
34        0.234845
35       -0.115982
36       -0.402345
37       -0.225520
38        0.303439
39       -0.043216
```

We can also generate the date of the last day of each quarter using the code below. This list will be later used as new indexes.

```
quarter_index_date = pd.date_range('4/1/2007', periods=40, freq='Q')
quarter_index_date
```

```
DatetimeIndex(['2007-06-30', '2007-09-30', '2007-12-31', '2008-03-31',
               '2008-06-30', '2008-09-30', '2008-12-31', '2009-03-31',
               '2009-06-30', '2009-09-30', '2009-12-31', '2010-03-31',
               '2010-06-30', '2010-09-30', '2010-12-31', '2011-03-31',
               '2011-06-30', '2011-09-30', '2011-12-31', '2012-03-31',
               '2012-06-30', '2012-09-30', '2012-12-31', '2013-03-31',
               '2013-06-30', '2013-09-30', '2013-12-31', '2014-03-31',
               '2014-06-30', '2014-09-30', '2014-12-31', '2015-03-31',
               '2015-06-30', '2015-09-30', '2015-12-31', '2016-03-31',
               '2016-06-30', '2016-09-30', '2016-12-31', '2017-03-31'],
              dtype='datetime64[ns]', freq='Q-DEC')
```

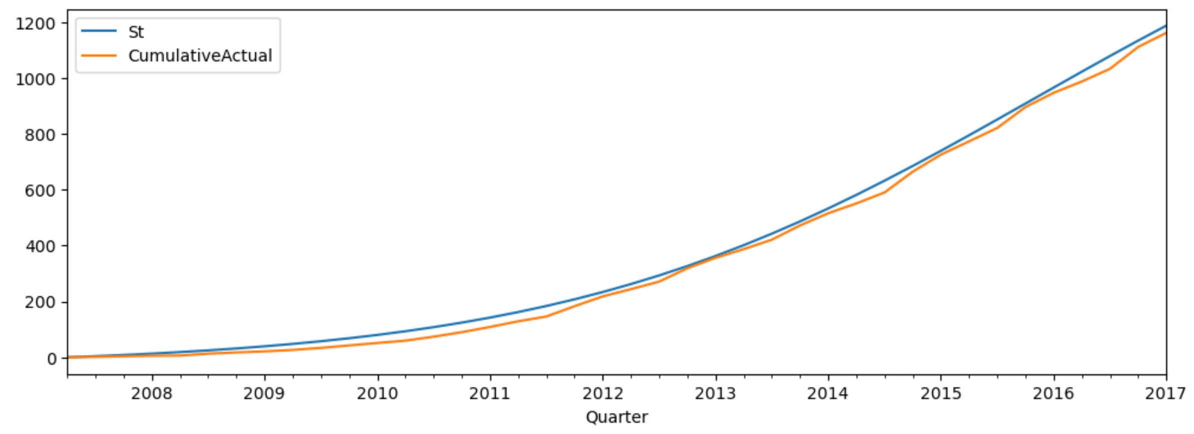
Question 4: Please (i) replace the original indexes using the newly created list `quarter_index_date` and (ii) then create the following plots from the DataFrame `bass_df` using the size `figsize=(12,4)`:

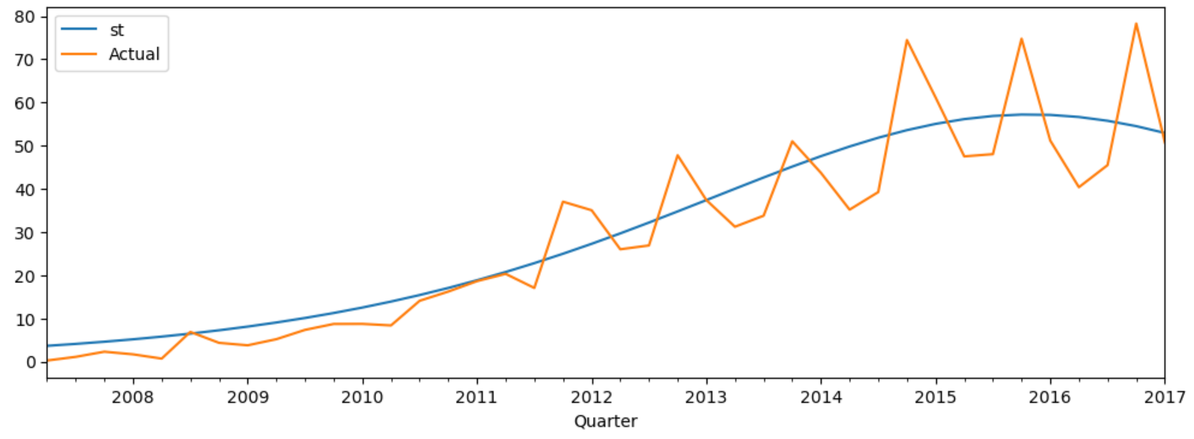
- Columns `['St','CumulativeActual']` using a line plot
- Columns `['st','Actual']` using a line plot
- Column `['PError']` using a bar plot

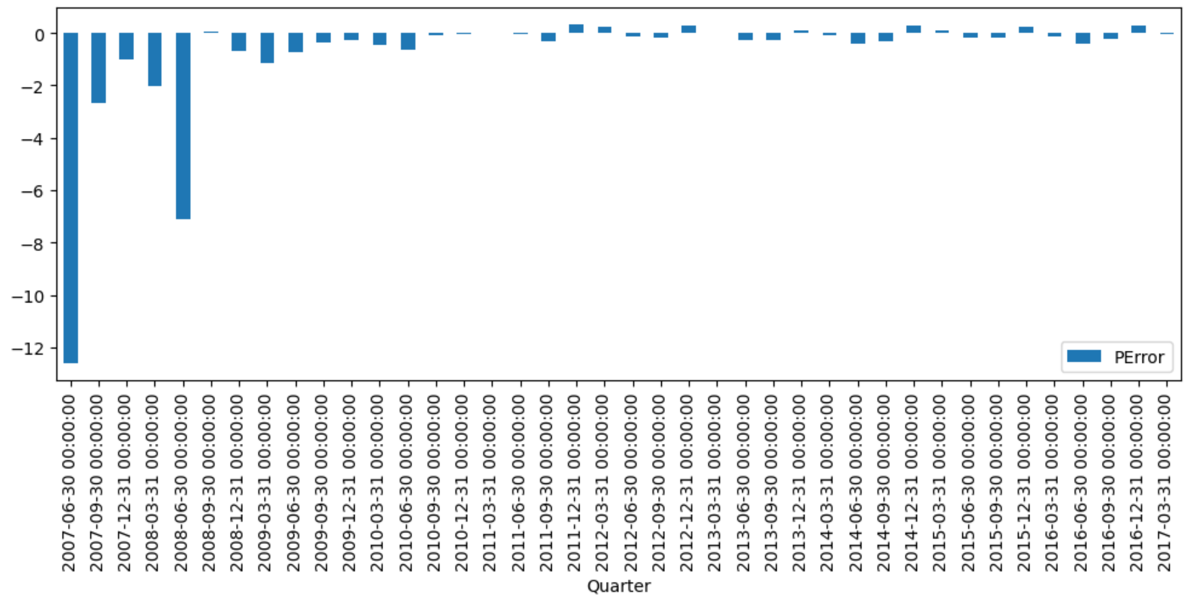
Hint: you can use the method `.set_index(...)` by calling `bass_df = bass_df.set_index(new_index_list)` to set the index (see link)

```
### start your code here ###
bass_df = bass_df.set_index(quarter_index_date)
bass_df.index.names = ['Quarter']
bass_df[['St','CumulativeActual']].plot(figsize=(12,4))
bass_df[['st','Actual']].plot(figsize=(12,4))
bass_df[['PError']].plot(kind='bar',figsize=(12,4))
### end your code here ###
```

```
<Axes: xlabel='Quarter'>
```


St
CumulativeActual
Quarter


st
Actual
Quarter


PError
Quarter