Integrative Analytics

Portfolio

<Index>

University of Southern California
MS in Analytics
Jae Yul Woo

1. MnM : Meal Planning for the New Millennium

1. Introduction

The Diet Problem, which is a famous application of linear programming (LP), was first stated by George Stigler in the 1930's. It was first solved by George Dantzig using the simplex method and has become the most widely studied model in introductory optimization (linear programming) books. The goal of this project is to bring more accurate and reasonable meal plans utilizing the informative and accessible data from the websites these days.

We will bring Internet resources, popular crowd-sourced recipes, as well as the idea of human-machine intelligence to this completely revamped version of the diet problem. Consider two people A and B who are planning for their meals each week. A and B are both males with age of 25 and they are attending a graduate school in Los Angeles. They are close enough to share cook and shopping responsibilities, but like all millennials, having limitation for time and money. However, they do have well-trained taste buds, and focus on eating healthy, tasty meals, and eating within budgetary and dietary restrictions. There are, however, other restrictions as well: they are committed to school-work, participate in social causes, take extra time for relaxing other than studying, shopping, and cooking.

The MnM problem will focus on meal planning with nutritions and preferences: A and B will share shopping, cooking, and meal selection responsibilities, and will also have to respect each others schedules, budgets, dietary restrictions, and of course, their likes/dislikes. They may use social media (and web sites) to get a sense of how much each will like a certain recipe. Likewise, several constraints are formulated for A and B's preferences. The plan should cover food requirements for 5 weekdays. Also, the recipes will be restricted by how much of each ingredient is necessary for 5 days. There should also be constraints on cooking time for any given day, and constraints on total foods. Thus, the dataset should include costs and preparation time for each recipes.

2. Methodology

The recipe data is collected from epicurious.com website with all the ingredients and nutritional information. The recipe data also contains user's ratings that can be used for our future analysis on the planner's preferences. Since the meal planning is based on A and B's own preferences, we derived ratings that were uploaded on the website, which were expressed as likes (and dislikes) for similar foods. Such correlations will be used in predicting preference ratings of recipes for A and B. The matrix-completion concepts is introduced to decide whether someone likes/dislikes each recipe. The objective function is the sum of the chosen recipes' ratings and introduced to predict individual's preferences. Based on the derived ratings, nutritional requirements, plan and budget restrictions, we will formulate the mixed integer programming to plan which recipe should be prepared each day and who will be in charge of the recipe fairly.

3. Data Crawling for recipes

As mentioned above, the recipe data is collected from epicurious.com with beautifulsoup library in python. This study is based on the recipe and its ratings in the dataset. For model simplification, we only consider protein, fat, sodium and calories, excluding black pepper, olive oil, salt etc. We first performed data crawling for 298 recipes with ingredients and personal ratings. Among 298, the recipes with NaN data and recipes having less than 9 reviews were deleted. Then, we have 116 recipes. We reorganized and analyzed the recipes and users in a matrix to construct the final dataset, recipes-user-rating. For model simplification, 20 recipes were chosen as the candidates for the weekly meal planning problem. The details are in the table 1-1.

ID	Title	Calories (kcal)	Fat(g)	Protein(g)	Sodium(mg)
2	Easy Green Curry with Chicken, Bell Pepper, and Sugar Snap Peas	549	36	42	760
10	Persian-Spiced Chicken with Spaghetti Squash, Pomegranate, and Pistachios	704	44	49	1002
13	Istanbul-Style Wet Burger (Islak Burger)	691	34	46	779
15	Pumpkin Spice Bundt Cake with Buttermilk Icing	372	14	5	282
18	Pumpkin Cheesecake with Bourbon sour Cream Topping	429	32	6	316
24	Kale Salad with Butternut Squash, Pomegranate, and Pumpkin Seeds	162	5	8	539
26	Stuffed Sweet Potatoes with Beans and Guacamole	733	34	23	1509
28	Slow Cooker Pork Shoulder with Zesty Basil Sauce	470	45	8	289
29	Easy General Tso's Chicken	699	35	50	1189
30	White Chicken Chili	534	14	45	968
38	One-Pot Curried Cauliflower with Couscous and Chickpeas	606	15	27	1365
48	Navy Bean and Escarole Stew with Feta and Olives	461	23	21	407
49	Butternut Squash, Kale, and Crunchy Pepitas Taco	233	17	7	363
50	Sheet-Pan Skirt Steak With Balsamic Vinaigrette, Broccolini, and White Beans	820	56	45	977
75	Summer Corn, Tomato, and Salmon Salad with Za'atar Dressing	810	53	40	1372
77	Curried Lentil, Tomato, and Coconut Soup	437	28	13	667
92	Sheet-Pan Cumin Chicken Thighs with Squash, Fennel, and Grapes	497	30	29	1177
104	Curried Pumpkin Soup	165	14	2	373
109	Persian Chicken with Turmeric and Lime	563	23	40	813
113	Baked Feta and Greens with Lemony Yogurt	577	38	23	839

Table 1-1. Nutrition Data for 20 popular recipes

4. Restrictions and Budget

The missing price and preparation time of each recipe were filled with the sum of all ingredients' price in the recipe at 'instacart.com' and by our own experiences. The price and preparation time is in the table 1-2. The available hours, free times, are set as 2 hours (120 minutes) everyday for both A and B. The nutritional restrictions for A and B (both 25 years old, male) are referred from 'mydailyintake.net' and details are in the table 1-3.

ID	Title	Price ($)	Preparation Time (min.)
2	Easy Green Curry with Chicken, Bell Pepper, and Sugar Snap Peas	7.2	30
10	Persian-Spiced Chicken with Spaghetti Squash, Pomegranate, and Pistachios	5.1	25
13	Istanbul-Style Wet Burger (Islak Burger)	4.3	25
15	Pumpkin Spice Bundt Cake with Buttermilk Icing	3.1	50
18	Pumpkin Cheesecake with Bourbon sour Cream Topping	4.5	120
24	Kale Salad with Butternut Squash, Pomegranate, and Pumpkin Seeds	4.3	10
26	Stuffed Sweet Potatoes with Beans and Guacamole	5.2	25
28	Slow Cooker Pork Shoulder with Zesty Basil Sauce	9.8	30
29	Easy General Tso's Chicken	6.1	30
30	White Chicken Chili	6.5	25
38	One-Pot Curried Cauliflower with Couscous and Chickpeas	4.5	35

48	Navy Bean and Escarole Stew with Feta and Olives	5.5	20
49	Butternut Squash, Kale, and Crunchy Pepitas Taco	3.5	30
50	Sheet-Pan Skirt Steak With Balsamic Vinaigrette, Broccolini, and White Beans	13	40
75	Summer Corn, Tomato, and Salmon Salad with Za'atar Dressing	12.5	20
77	Curried Lentil, Tomato, and Coconut Soup	4.8	45
92	Sheet-Pan Cumin Chicken Thighs with Squash, Fennel, and Grapes	7.5	25
104	Curried Pumpkin Soup	3.5	60
109	Persian Chicken with Turmeric and Lime	6.3	25
113	Baked Feta and Greens with Lemony Yogurt	5	30

Table 1-2. Price and Preparation Time

	Calories (Kcal)	Protein (g)	Fat (g)	Sodium (mg)
Lower	2700	140	150	2400
Upper	3600	200	250	3000

Table 1-3. Nutritional Restrictions

5. Singular Vector Decomposition

We used singular vector decomposition method to derive the preferences for both A and B. From the ratings matrix from the Epicurious.com website, we leveraged a latent factor model to capture the similarity between users and recipes. Essentially, we turned the recommendation problem into an optimization problem. We treated randomly chosen 40 users as one user so that the biased preference for some recipes can be relieved. Then we normalized the ratings-per-recipe to properly represent latent factor. Therefore we could derive A and B’s preference for the chosen 40 recipes as in table 1-4.

ID	Title	Expected Rating
2	Easy Green Curry with Chicken, Bell Pepper, and Sugar Snap Peas	3.81465519
10	Persian-Spiced Chicken with Spaghetti Squash, Pomegranate, and Pistachios	5
13	Istanbul-Style Wet Burger (Islak Burger)	4
15	Pumpkin Spice Bundt Cake with Buttermilk Icing	1
18	Pumpkin Cheesecake with Bourbon sour Cream Topping	1.64170688
24	Kale Salad with Butternut Squash, Pomegranate, and Pumpkin Seeds	3.01206504
26	Stuffed Sweet Potatoes with Beans and Guacamole	2.42544632
28	Slow Cooker Pork Shoulder with Zesty Basil Sauce	4
29	Easy General Tso's Chicken	3.398837002
30	White Chicken Chili	3
38	One-Pot Curried Cauliflower with Couscous and Chickpeas	2.180336854
48	Navy Bean and Escarole Stew with Feta and Olives	2.489009934
49	Butternut Squash, Kale, and Crunchy Pepitas Taco	3
50	Sheet-Pan Skirt Steak With Balsamic Vinaigrette, Broccolini, and White Beans	4
75	Summer Corn, Tomato, and Salmon Salad with Za'atar Dressing	3.240245716
77	Curried Lentil, Tomato, and Coconut Soup	3.695716906
92	Sheet-Pan Cumin Chicken Thighs with Squash, Fennel, and Grapes	2.159726666

104	Curried Pumpkin Soup	2
109	Persian Chicken with Turmeric and Lime	2.72080467
113	Baked Feta and Greens with Lemony Yogurt	1

Table 1-4. Expected Ratings of chosen 20 recipes for A and B

6. Mixed Integer Programming for the meal plan

The mixed integer programming is formulated with the preferences on the recipes (based on ratings), nutritions, free time in a week, expenses and fair preparation schedule for the meal plan. The mixed integer programming is set as a function that can be used repeatedly with different scenarios each week. (such as more proteins, less sodium etc...) The definition for variables are represented in table 1-4.

$x_i \in (0,1)$	Decision variable for a recipe	c_i, p_i, f_i, s_i	Calories, protein, fat, sodium in recipe i
$y_{ij} \in (0,1)$	A's decision for recipe i on day j	$price_i$	Price($) of recipe i
$z_{ij} \in (0,1)$	B's decision for recipe i on day j	Pt_i	Preparation time for recipe i
R_i	Rating for recipe i	$At1_j, At2_j$	Available time for each person's day j
C_{lb}, C_{ub}	Calories lower bound and upper bound	c_{adj}, p_{adj}	Control parameter for calories, protein
P_{lb}, P_{ub}	Protein(g) lower bound and upper bound	f_{adj}, s_{adj}	Control parameter for fat and sodium
F_{lb}, F_{ub}	Fat(g) lower bound and upper bound	α	Control parameter for time balance, 0.01
S_{lb}, C_{ub}	Sodium(mg) lower bound, upper bound	w	Meal preparation time difference

Table 1-5. Notation for Mixed Integer Programming

$$Maximize \sum_{i=1}^{20} R_i \times x_i - \alpha * w$$

Selection Constraints

$$subject\ to \sum_{i=1}^{20} x_i = 5$$

$$\sum_{i=1}^{20}\sum_{j=1}^{5} y_{ij} \leq 5$$

$$\sum_{i=1}^{20}\sum_{j=1}^{5} z_{ij} \leq 5$$

$$\sum_{i=1}^{20} x_i = \sum_{i=1}^{20}\sum_{j=1}^{5} y_{ij} + \sum_{i=1}^{20}\sum_{j=1}^{5} z_{ij}$$

$$|\sum_{i=1}^{20}\sum_{j=1}^{5} y_{ij} - \sum_{i=1}^{20}\sum_{j=1}^{5} z_{ij}| \leq 1$$

One-one Constraint

$$\sum_{i=1}^{20} y_{ij} + \sum_{i=1}^{20} z_{ij} = 1,\ j \in (1,2,3,4,5)$$

Nutritional Constraints

$$C_{lb} \leq \sum_{i=1}^{20} x_i \times c_i \leq C_{ub} \times c_{adj}$$

$$P_{lb} \times p_{adj} \leq \sum_{i=1}^{20} x_i \times p_i \leq P_{ub}$$

$$F_{lb} \leq \sum_{i=1}^{20} x_i \times f_i \leq F_{ub} \times f_{adj}$$

$$S_{lb} \leq \sum_{i=1}^{20} x_i \times s_i \leq S_{ub} \times s_{adj}$$

Preparation Time Constraints

$$\sum_{i=1}^{20} Pt_i \times y_{ij} \leq At1_j,\ j \in (1,2,3,4,5)$$

$$\sum_{i=1}^{20} Pt_i \times z_{ij} \leq At2_j,\ j \in (1,2,3,4,5)$$

Time Balance Constraints

$$\sum_{i=1}^{20}\sum_{j=1}^{5} y_{ij} - \sum_{i=1}^{20}\sum_{j=1}^{5} z_{ij} \leq w$$

$$-\sum_{i=1}^{20}\sum_{j=1}^{5} y_{ij} + \sum_{i=1}^{20}\sum_{j=1}^{5} z_{ij} \leq w$$

Redundancy Constraints

$$\sum_{i=1}^{20} y_{ij} \leq 1,\ j \in (1,2,3,4,5)$$

$$\sum_{i=1}^{20} z_{ij} \leq 1,\ j \in (1,2,3,4,5)$$

$$\sum_{j=1}^{5} x_{ij} \leq 1,\ i \in (1,2,3,....19,20)$$

Budget Constraints

$$\sum_{i=1}^{20} price_i \times x_i \leq 100$$

7. Conclusion

The defined function ‘mnmplans(cal_rate=0, pro_rate=0, fat_rate=0, sod_rate=0)’ is with default adjustment parameter values of 0 for $c_{adj}, p_{adj}, f_{adj}, s_{adj}$. We used the function to print meal plan for 5 days. The result for default adjustment parameters are in table 1-5. We also examined the ‘mnmplans(cal_rate=20)’ which is for 20% restriction on calories. The result for this situation is in table 1-6.

	Day 1	Day 2	Day 3	Day 4	Day 5
A	Easy Green Curry with Chicken, Bell Pepper, and Sugar Snap Peas	Slow Cooker Pork Shoulder with Zesty Basil Sauce	Istanbul-Style Wet Burger (Islak Burger)	-	-
B	-	-	-	Curried Pumpkin Soup	Pumpkin Cheesecake with Bourbon Sour Cream Topping

Table 1-5. Meal Plan with no adjustments

	Day 1	Day 2	Day 3	Day 4	Day 5
A	-	-	Pumpkin Cheesecake with Bourbon Sour Cream Topping	-	Curried Pumpkin Soup
B	Easy Green Curry with Chicken, Bell Pepper, and Sugar Snap Peas	Slow Cooker Pork Shoulder with Zesty Basil Sauce	-	Istanbul-Style Wet Burger (Islak Burger)	-

Table 1-6. Meal Plan with 20% Calorie restriction

We found two example plans of the meal plan problem for the new millennium using singular vector decomposition and mixed integer programming. The plan is constructed to strictly follow the nutritional restrictions and preparation, and is fairly divided for both A and B. The defined function ‘mnmplans()’ can be reused repeatedly each week by setting the parameters and relieve the constriants by each week’s meal planning objective.

2. LEO-Wyndor : Advertising and Production

1. Introduction

The “Wyndor Glass Co.” is the resource utilization project for the production of high quality glass doors A and B: some with aluminum frames (A), and others with wood frames (B). These doors are produced in three plants, named 1, 2, and 3. The data for the production resources are shown in the table 2-1. The product mix to maximize the total revenue will not only be decided by production plan, but also by the potential future sales. Sales, however, is an uncertain parameter and it depends on the marketing strategy. Given 200 advertising time slots, the marketing strategy is to choose a mix of advertising outlets through which can reach out to potential consumers efficiently. We assume that the sales dataset reflects advertisement resulted from the campaigns by Wyndor Glass Co.

Plant	Prod.time for A	Prod.time for B	Total Hours
1	1	0	8
2	0	2	24
3	3	2	36
Profit per Batch	$3,000	$5,000	

Table 2-1. Data for the Wyndor Glass Problem

The advertisement uses two different media, TV and radio. As in the original dataset, advertising strategy is represented as budgeted dollars for each media type. Thus, in our statistical model, sales predictions are based on the dollar amount of TV and radio advertising. In our interpretation, product-sales reflect total number of doors sold ($\{W_i\}$) when advertising expenditure for TV is $X_{i,1}$and that for radio is $X_{i,2}$. (The data set has 200 data points, that is, $i = 1, ..., 200$). Let x_1 denote the TV advertising dollar amount, and x_2 denote the radio advertising dollar amount.

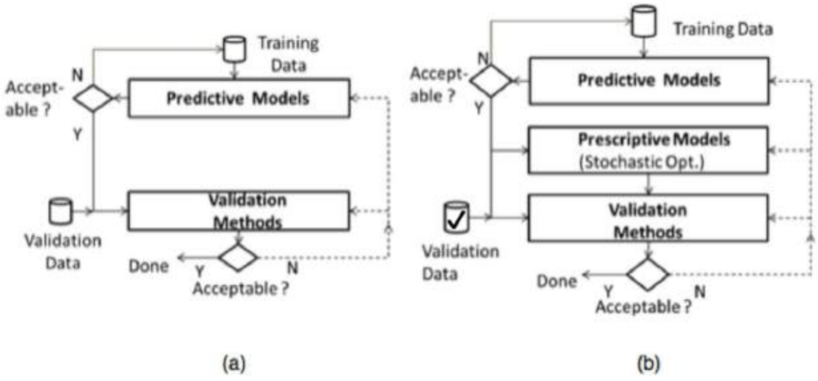

Figure 2-1. Statistical Learning and Learning Enabled Optimization

This project aims to maximize the total revenue of the production with respect to advertisement plans. We will compare the statistical learning model and learning enabled model to get reliable and realizable predictions of the future revenue.

2. Methodology

The process summarized in the figure 2-1 in which the entire data set is divided into two parts (Training and Validation). The former is used to learn model parameters (for a predictive model), and the latter data set is used for model assessment and selection. Once a model is selected, it can be finally tested by either simulation or by the additional “test dataset” before adoption.

The SL and LEO framework is used for two mathematical programming models which are the deterministic linear programming and the stochastic right-hand-side programming. The deterministic linear programming model uses pyomo for the formulation and the stochastic programming model uses PySP. The empirical additive error, denoted ξ, is the difference between the actual value and predicted value. Empirical additive errors is used to represent the randomness in the stochastic modeling. Each model corresponds to Statistical Learning, figure 2-1 (a) and Learning Enabled Optimization, (b), respectively.

3. Predictive Model

The statistical learning model uses deterministic multiple linear programming with the advertising effect on sales. The linear regression model is used to measure the total sales on each possible pair amounts of the TV and Radio advertisement. For the regression model, the data is splitted into training data and validation data, 50% each with random seed 1. The train dataset is used for the predictive multiple linear regression model. The model with the training data is as below.

$$y_{Sales} = 3.74982951 + 0.05370186 \times x_{TV} + 0.22231025 \times x_{Radio}$$

Using the predictive regression model, we found the difference between the actual sales data and predicted sales data. We set $\{Z_i, W_i\}$ where Z_i is predicted sales amount and the W_i is the actual sales amount. We denote the error terms $\xi_{i,T} = Z_{i,T} - W_{i,T}$ for all $i \in Training$ and $\xi_{j,V}$ as $\xi_{j,V} = Z_{j,V} - W_{j,V}$ for all $j \in Validation$. Since $\xi_{i,T}$ and $\xi_{j,V}$ represents the normalized errors between the two datasets, qq-Plot is used to determine the error trend of the two datasets. The minimum and maximum data points were found as the outliers in the training dataset. Hence, we remove the two points and refit the multiple linear regression model. The model without any outlier is below.

$$y_{Sales} = 4.07459792 + 0.05119161 \times x_{TV} + 0.22734845 \times x_{Radio}$$

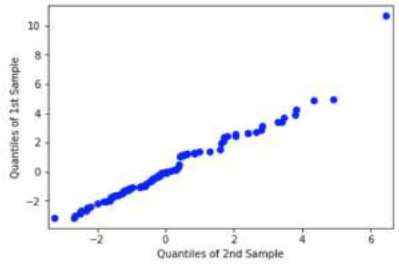

Figure 2-2. qq-Plot for $\xi_{i,T}$ and $\xi_{j,V}$

4. Statistical Learning

The deterministic linear programming model for the statistical learning is formulated as model 2-1.1 Total sales constraint has upper bound of predicted sales from the advertisement effect. This model can be treated as a single linear program with 4 decision variables (x_1, x_2, y_a, y_b) and 9 constraints. We solve the model with training dataset to derive the optimal solution for x_1, x_2 denoted as $(x_{1,D}, x_{2,D})$. The optimal solution is in the table 2-2.

$$
\begin{aligned}
&Maximize: -0.1 \times x_1 - 0.5 \times x_2 + 3 \times y_a + 5 \times y_b \\
&subject\ to: y_a \leq 8 \\
&\qquad y_b \leq 24 \\
&\qquad 3 \times y_a + 2 \times y_b \leq 36 \\
&\qquad y_a + y_b \leq \beta_0 + \beta_1 \times x_1 + \beta_2 \times x_2 \\
&\qquad x_1 + x_2 \leq 200 \\
&\qquad x_1 - 0.5 \times x_2 \geq 0 \\
&\qquad x_1,\ x_2,\ y_a,\ y_b\ \geq 0 \\
&\qquad l_1 \leq x_1 \leq u_1 \\
&\qquad l_2 \leq x_2 \leq u_2
\end{aligned}
$$

Model 2-1.1. Deterministic MLP

	Optimal Solution
$Objective$	48.16914111658245
$x_{1,D}$	190.42285279145614
$x_{2,D}$	9.577147208543863
y_a	4.0
y_b	12.0

Table 2-2. Optimal Solution for Statistical Learning

Now, we validate the statistical learning model with the possible errors derived from the validation dataset. The optimal solution $(x_{1,D}, x_{2,D})$ is applied to the model 2-1.2 and the $\xi_{j,V}$ is introduced as the possible 100 errors during the validation procedure. The validation model with two decision variables (y_a, y_b) is formulated as below.

$$
\begin{aligned}
&Maximize: -0.1 \times x_{1,D} - 0.5 \times x_{2,D} + 3 \times y_a + 5 \times y_b \\
&subject\ to: y_a \leq 8 \\
&\qquad y_b \leq 24 \\
&\qquad 3 \times y_a + 2 \times y_b \leq 36 \\
&\qquad y_a + y_b - \beta_0 - \beta_1 \times x_{1,D} - \beta_2 \times x_{2,D} \leq \xi_{j,V} \\
&\qquad y_a,\ y_b\ \geq 0
\end{aligned}
$$

Model 2-1.2. Validation model for Deterministic MLP

The model 2-1.2 is solved for each $\xi_{j,V}, j \in Validation$. Thus, we have 100 optimal objective values from the model 2-1.2. The 95% confidence interval for the optimal objective value is (45.56898, 46.57599)

5. Learning Enabled Optimization

The stochastic programming model for the learning enabled optimization is formulated as model 2-2.1 Total sales constraint has the probabilistic right-hand-side of the empirical additive errors. This model can be solved by the SD solver with PySP formulation of 4 decision variables (x_1, x_2, y_a, y_b) and 9 constraints. To prevent the objective becoming negative value, we added big M to the objective function and set M as 200. We solve the model with training dataset errors to derive the optimal solution for x_1, x_2 and denoted the solution as $(x_{1,S}, x_{2,S})$. The optimal status is in the Table 2-3.

$$
\begin{aligned}
&Minimize : 0.1 \times x_1 + 0.5 \times x_2 - 3 \times y_a - 5 \times y_b + M \\
&subject\ to : y_a \le 8 \\
&\qquad y_b \le 24 \\
&\qquad 3 \times y_a + 2 \times y_b \le 36 \\
&\qquad y_a + y_b - \beta_1 \times x_1 - \beta_2 \times x_2 \le \beta_0 + \xi_i \\
&\qquad x_1 + x_2 \le 200 \\
&\qquad x_1 - 0.5 \times x_2 \ge 0 \\
&\qquad x_1,\ x_2,\ y_a,\ y_b\ \ge 0 \\
&\qquad l_1 \le x_1 \le u_1 \\
&\qquad l_2 \le x_2 \le u_2
\end{aligned}
$$

Model 2-2.1 Stochastic MLP

	Optimal Solution
$Objective$	45.9991
$x_{1,S}$	1.983564e+02
$x_{2,S}$	1.643590e+00

Table 2-3. Optimal Solution for Learning Enabled Optimization

Now, we validate the learning enabled model with the possible errors derived from the validation dataset. While the deterministic model needs to confirm the derived objective value is in between the 95% confidence interval, the stochastic model needs to confirm if the probabilistic distribution for the training objective values corresponds with the distribution of validation objective values. The optimal solution is applied to the model 2-2.2 and the $\xi_{i,T}, \xi_{j,V}$ is introduced as the possible errors. The validation model with two decision variables (y_a, y_b) is formulated as below.

$$
\begin{aligned}
&Maximize : -0.1 \times x_{1,D} - 0.5 \times x_{2,D} + 3 \times y_a + 5 \times y_b \\
&subject\ to : y_a \le 8 \\
&\qquad y_b \le 24 \\
&\qquad 3 \times y_a + 2 \times y_b \le 36 \\
&\qquad y_a + y_b - \beta_0 - \beta_1 \times x_{1,D} - \beta_2 \times x_{2,D} \le \xi_{i,T} / \xi_{j,V} \\
&\qquad y_a,\ y_b\ \ge 0
\end{aligned}
$$

Model 2-2.2. Training and Validation model for Deterministic MLP

The model 2-2.2 is solved for each $\xi_{i,T}, i \in Training$ and $\xi_{j,V}, j \in Validation$ for 100 times, respectively. Thus, we have 200 optimal objective values. The 95% confidence interval for the validation objective value is (45.40041, 46.90166). We conducted the Kruskal Wallis H-test to validate the difference between two groups.

H-statistic: 0.31285, P-Value: 0.57593
Accept NULL hypothesis - No significant difference between groups.

6. Conclusion

We observed the difference between the deterministic statistical learning model and learning enabled optimization model. We first confirmed the insignificant difference between the training and validation dataset and conducted multiple linear regression of advertisement effect on sales. The 95% confidence interval for the revenue of the validation dataset with the possible EAE of the models are ($45.56898, $46.57599) and ($45.40041, $46.90166) (in 1000s). The statistical model with the MLP overestimated the revenue as $48.17 (in 1000s) which is reasonable in that the regression model did not consider error terms which will depreciate the expected revenue. However, the learning enabled optimization model estimated the total revenue as $46.00 (in 1000s). The total revenue by LEO model is in the 95% confidence interval and the probabilistic distribution for the training objective values corresponds with the validation objective values. Hence, we concluded that the LEO model gives more probable and reliable outcome for the prediction than the statistical learning model itself.

3. LEO-ELEQUIP : Time Series and Inventory

1. Introduction

Single echelon inventory model is one of the most common case in operations management. Since the demand is stochastic, the objective is to minimize the sum of expected holding cost and lost sale cost. A vendor manages the inventory plan of electric equipments. In the beginning of each months, the vendor orders equipments from an online service called O-Maison which will deliver the equipments in the following month. The upper bound for delivery is U_t, where t denotes the end of the month when the order is placed. We set y denoting the starting inventory, x denoting the ending inventory, D denoting the monthly demand, Delta denoting a monthly order quantity. The specific process is in the figure 3-1.

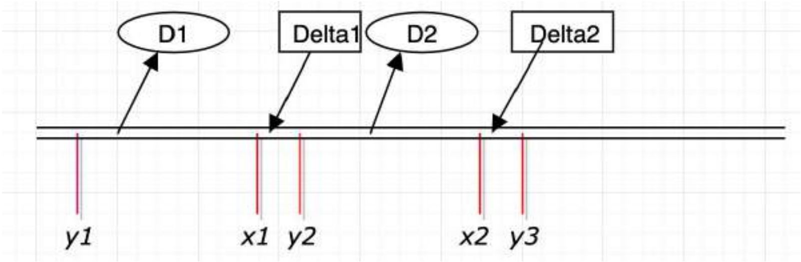

Figure 3-1 Flow chart

In the decision model, we use a ‘rolling-horizon’ scheme. We’ll seek for 3 months decisions, while we only implement a plan for 1 month. As time goes on, the rolling horizon approach has the advantage of observing more data, and over time, the decisions themselves will be updated with the current data and the new data collected. Which results in the horizon approach to acheive precisely predicted plans.

5 years demand data is used to build the time series model. We are going to predict next 3 months demand by implementing the demand forecasts on both deterministic model and stochastic model and calculate the optimal solution of order quantities and the costs. Likewise, the model will update parameters and recalculate the order quantity. 6 months total cost is derived by repeating the process and we will compare the costs from Deterministic Linear Programming model and Stochastic Programming model for better prediction and low costs.

2. Time Series model

ARIMA(Autoregressive Integrated Moving Average model) is used for the time series analysis. The operations for data processing and time series modeling are conducted in Rstudio. The original data is shown in the figure 3-2.1.

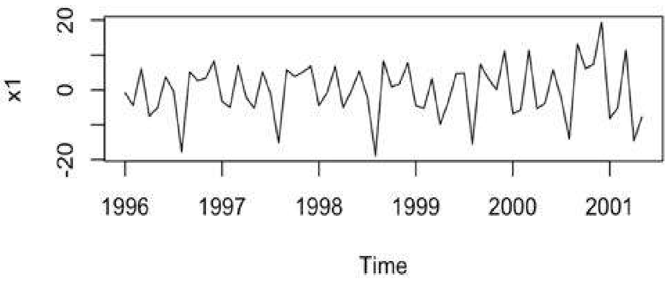

Figure 3-2.1 Original plot

To build a better model for the predictions, data processing for the time series modeling was needed. The first thing was to remove outliers in that outliers may lead the model biased. We used ‘tsclean’ in the package ‘tseries’, which helped us to automatically find and remove outliers. After the adjustments on the datset, we divided original time series data into the trend, seasonal and remainder part. The processed data is shown in the figure 3-2.2.

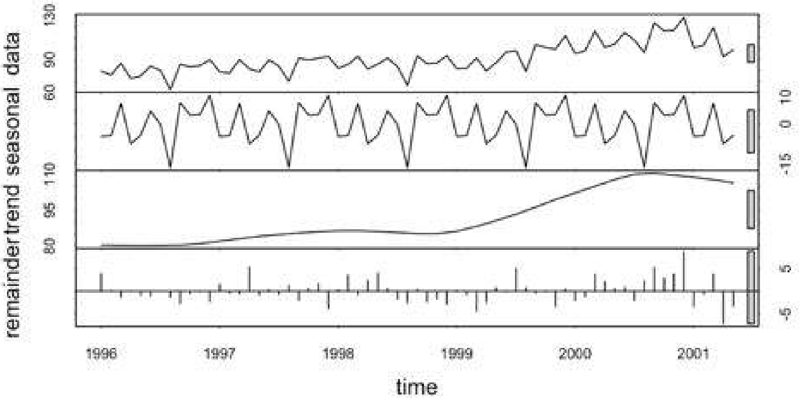

Figure 3-2.2 Analyzed plot

Since the stationary ARIMA model was demanded for the inventory model, we removed the trend part from the original dataset and confirmed the stationary condition using the KPSS-test.

KPSS Level = 0.05996, Truncation lag parameter = 3, p-value = 0.1

We found the p-value as 0.1 and the null hypothesis cannot be rejected. Thus, we concluded the data being stationary. The demand data from January 1996 to December 2000 was used to train the model and to predict the demand forecasts for January, February and March 2001. In addition, we compared the predicted demands and the actual demands from Jan 1996 to Dec 2000 and stored the differences as errors.
This procedure was repeated for 5 times, each time adding the following month in the training data set. Hence we could derive 6 demand forecasts and 6 error terms.

3. Deterministic Linear Programming

The Deterministic Linear Programming model was introduced to calculated the total cost using optimization program, AMPL. The model is shown in the model 3-1.

$$Min\ \sum_{t=0}^{T-1} h_t x_t + b_t z_t$$

$$subject.to.$$

$$y_{t+1} - x_t - \Delta_t = 0$$

$$-y_t + x_t >= -D_t$$

$$y_t + z_t >= D_t$$

$$\Delta_t <= U_t$$

$$x_t, z_t, \Delta_t >= 0$$

Model 3-1. Deterministic Linear Programming model

In this model, h_t is denoted as holding cost at time t, b_t as lost sale cost at time t, y_t as the starting inventory at time t, x_t as the ending inventory at time t, Δ_t as order quantity at time t, U_t as the upper bound for order quantity.

3 months demand were chosen by t ranging from 1 to 3. Firstly, D_1, D_2, D_3 was set as the forecasted demands each representing Jan 2001, Feb 2001 and March 2001. By solving the deterministc linear program, we got the optimal solution, $\Delta_1, \Delta_2, \Delta_3$, which indicates 3-months horizon. We implement last two months and introduce the following months as the new time horizon. The definitions of monthly holding cost, lost sale cost and y_{start} is shown in the formula 3-1.

$$Hoding\ cost = h_t * (y_t - D_{real}) = h_t x_t$$

$$Lost\ sale\ cost = b_t * (D_{real} - y_t) = b_t z_t$$

$$y_{start\ (t+1)} = x_t + \Delta_t or\ \Delta_t - z_t$$

Formula 3-1. Definitions of Costs

In the next iteration, as mentioned above, we set new D_1, D_2, D_3 by the predicted demands on February, March and April 2001. We derive the new optimal solution Δ_1 for Feb 2001. By repeating the procedure starting from January to May 2001, we could predict the 6 months total cost starting from January to June 2001.

4. Stochastic Programming

The stochastic model was introduced for the model robustness considering the error terms calculated during the training process. The model as in the model 3-2, was formulated with pyomo and solved with SD solver.

$$\begin{array}{llll}
\text{Min } \mathbb{E}_{\tilde{\omega}}[\sum_{t=0}^{T-1} h_t x_t(\tilde{\omega}) + b_t z_t(\tilde{\omega})] & & & \\
\text{s.t.} \quad y_{t+1}(\omega) - x_t(\omega) - \Delta_t & = 0 & \text{for almost all } \omega \\
y_{t+1}(\omega) & \leq R_{t+1} & \text{for almost all } \omega \\
\Delta_t & \leq U_t & \\
-y_t(\omega) + x_t(\omega) & \geq -D_t(\omega) & \text{for almost all } \omega \\
y_t(\omega) \qquad + z_t(\omega) & \geq D_t(\omega) & \text{for almost all } \omega \\
x_t(\omega), z_t(\omega), \Delta_t & \geq \mathbf{0} &
\end{array}$$

Model 3-2 Stochastic Programming model

The difference with DLP model is ω terms, which represent the error terms of the demands which was derived form the differences between the predicted demands and the actual demands. The error terms is used for possible fluctuations of the demands in the SP model which will produce various scenarios. Similar to the deterministic model, we iterated the process with PySP and SD solver repeating 5 time horizons. Likewise, we could find the 6 months total cost from January to June 2001.

5. Model Comparison and Conclusion

The results for DLP and SP model is in the table 3-1 and 3-2, repectively. The marked color in the table 3-2 is to indicate that the cost was soley calculated by the lost sale cost.

DLP	JAN	FEB	MAR	APR	MAY	JUN
real_DEMAND	100.56	103.05	119.06	92.46	98.75	111.14
delta	113.127	119.344	98.5425	106.514	95.8079	
Y	101.38	113.947	130.241	109.7235	125.7775	120.8354
cost	0.82	10.897	11.181	17.2635	27.0275	9.6954
accumulated cost	0.82	11.717	22.898	40.1615	67.189	76.8844

Table 3-1 DLP Cost Table

SP	JAN	FEB	MAR	APR	MAY	JUN
real_DEMAND	100.56	103.05	119.06	92.46	98.75	111.14
delta	115.58	99.34	101.83	107.37	90.71	
Y	101.38	116.4	112.69	95.46	110.37	102.33
cost	0.82	13.35	19.11	3	11.62	26.43
accumulated cost	0.82	14.17	33.28	36.28	47.9	74.33

Table 3-2 SP Cost Table

The 2 graphs below are the monthly costs and the monthly accumulated costs with respect to each models.

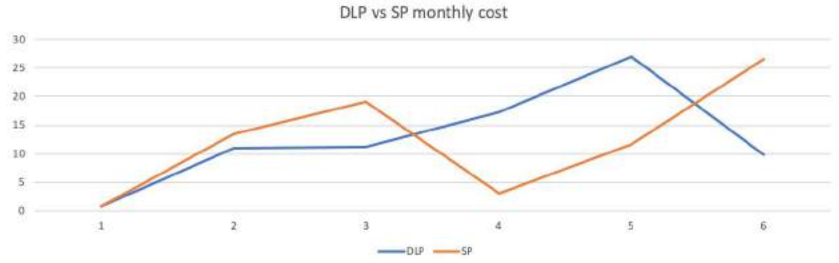

Figure 3-3.1 Monthly cost

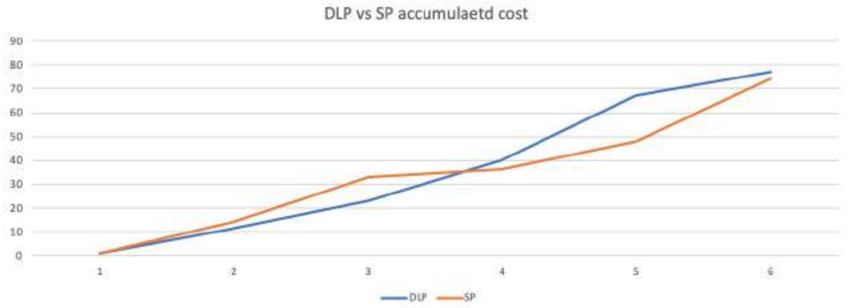

Figure 3-3.2 Monthly accumulated cost

In conclusion, we could observe that SP model gives a more robust outcome than the DLP model. Also, the SP model forecasts the lost sale costs and the holding costs, while DLP model only calculates the holding cost. Along with the model robustness, the SP model has lower predicted costs than the DLP model. Therefore, we could conclude that the SP model gives more reliable and cost-effective predictions than the DLP model.

4. Feasibility Validation of Launching LA Metro Bike Relocation Service

Jae Yul Woo
University of Southern California

***Abstract*— The LA Metro Bike is a transportation service commonly used by many LA citizens. As there are numerous stations, possible routes and because users use it for different purposes, imbalance among the supply and demand stations is inevitable. To resolve the imbalance, relocating the bikes is necessary, which comes with a great cost. Therefore this study is focused on finding the ideal discount rate to promote bike relocation at the minimum cost. With current outsourcing cost in LA Metro and price policy to users, the ideal discount amount was found to be 25% from the regular price to 'non-passholders' and rewarding the same amount to 'passholders.' The optimal routes and number of rides for the new service is in the file 'optimal_sol.csv' Furthermore, it is expected that the outsourcing cost after launching the new service will fluctuate. Thus, the LA Metro should keep track of outsourcing cost and keep the cost in the range of $0.10/mile to $3.10/mile to maintain the service at proper discount rate.**

I. Introduction

LA citizens are using metro for their daily basis. Although there are many kind of substitutes such as car-sharing services and electric-scooters prevalent in LA area, the metro bike has its strength in the accessibility and healthy features. LA Metro team publicized that there were total 700k trips and 2 million miles traveled so far with metro-bike. However, the trips are mostly focused on few routes since the bikes are mainly used for commutes. The LA metro team is taking charge of moving the unbalanced demand for their expenses. Hence, this study aims to validate the feasibility of the metro bike relocation service in the LA area.

II. Data Configuration & Preprocessing

A. Configuration

The data is shared in the LA Metro bike-share web page. Each csv file contains data for one quarter of the year. Each file contains the following data columns:

- trip_id: Locally unique integer that identifies the trip
- duration: Length of trip in seconds
- start_time: The date/time when the trip began, presented in ISO 8601 format in local time
- end_time: The date/time when the trip ended, presented in ISO 8601 format in local time
- start_station: The station ID where the trip originated
- start_lat: The latitude of the station where the trip originated
- start_lon: The longitude of the station where the trip originated
- end_station: The station ID where the trip terminated
- end_lat: The latitude of the station where the trip terminated
- end_lon: The longitude of the station where the trip terminated
- bike_id: Locally unique integer that identifies the bike
- plan_duration: The number of days that the plan the passholder is using entitles them to ride; 0 is used for a single ride plan (Walk-up)
- trip_route_category: "Round Trip" for trips starting and ending at the same station or "One Way" for all other trips
- passholder_type: The name of the passholder's plan

B. Data Preprocessing

Most of the data columns are categorical values except trip duration, start and end time. The bike trip data is from July 2016 to June 2017 with about 112 thousand rows.
Rides with NaN values(11177 rides), by 'LA Metro staff(3225 rides) and 'Round Trips(10459 rides) type rides are excluded.(Only the 'One-way' trips types are considered for the relocation service.) The passholder types are also reassigned to 0 and 1. Category 0 as non-passholders ('Walk-up, 24241), and category 1 as passholders. ('Monthly Pass + 'Flex Pass, 64025)
The 'distance' feature is added by calculating the Manhattan distance in miles between starting stations and ending stations. Station ID 4108 is classified as an outlier and all the trips that starts or ends at station 4108 are removed. 'Duration_min' column is also added to calculate the price for each trip. The current price policy is in the table. 1 below. Lastly, 9 features that are not relevant to the relocation service are removed. Thus, there are 'Duration'(in seconds), 'Start Time', 'Starting Station ID', 'Ending Station ID', 'Passholder Type', 'Distance', 'Duration_min'(in minutes) as columns in the processed dataset.

Pass Type	Price	Charges	Category
1 Ride	$1.75/30min.	every 30 min	0
Monthly	$1.75/30min.	first 30 min free	1
Annual	$1.75/30min.	first 30 min free	1

TABLE I: Price Policy

III. Exploratory Data Analysis

A. Duration Time by Passholder Types

By observing distances between stations and the mean, median duration time with respect to the passholder types, most passholders are using bikes for no more than 30 minutes. Non-passholders are also using bikes less than 30 minutes but few riders use a lot more than 30 minutes that the mean duration time exceeds 30 minutes.

Pass Type	Duration	Duration_min	Distance
0	2166.9427	36.1157	1.4315
1	771.7251	12.8621	5.3473

TABLE II: Mean Value

Pass Type	Duration	Duration_min	Distance
0	960	16.0	0.7308
1	480	8.0	0.6129

TABLE III: Median Value

B. Supply and Demand Imbalance in stations

The visualization of the top 15 popular starting and ending stations are in fig. 1. There are imbalances between starting and ending stations even though the popularity trends in both starting and ending stations are similar.

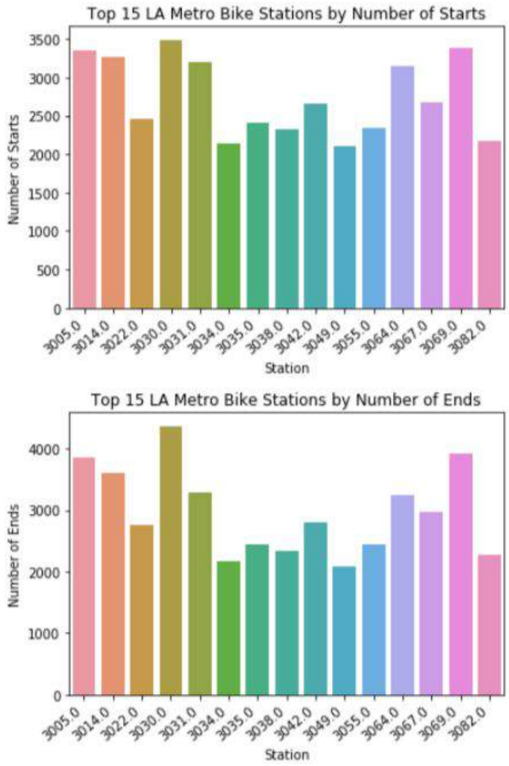

Fig. 1: Imbalance in Stations
(Upper: Starting Stations, Lower: Ending Stations)

C. Popular Trips

Popular trips in the dataset is derived as in fig.2. The most popular route is between station 3030 and station 3014, both ways.

D. Feature Correlations

The correlation between features are not significant as in fig.3. Thus, every feature is independent to each other.

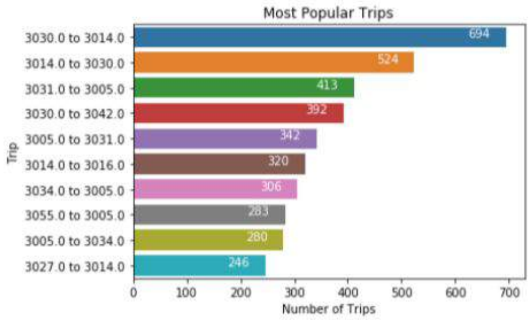

Fig. 2: Most Popular Trips

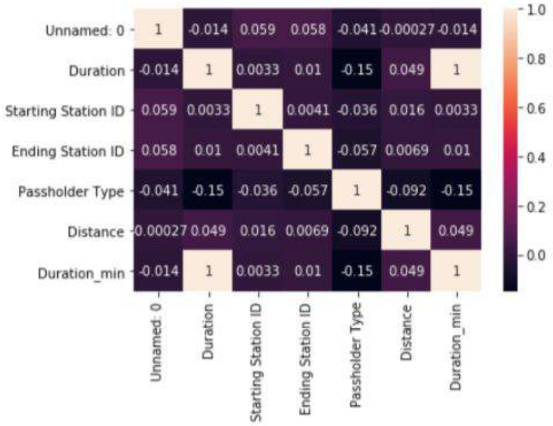

Fig. 3: Correlation Matrix

IV. Network

This relocation problem is inspired by the network problem and the problem consists of supplies, demands and the cost for each route. The bike-supplying stations and bike-demanding stations are defined as the supplies and demands in this relocation problem. The supplying stations are the stations with more incoming bikes than outgoing bikes, which means in trip data, the stations with more ending trips than the starting trips. Likewise, demanding stations are the stations with more outgoing bikes than incoming bikes, which means in trip data, the stations with more starting trips than ending trips. After grouping the data, there are 36 supplying stations and 27 demanding stations with their IDs and net supplies/demands, respectively.

Now in this network problem, the costs for each station routes are defined as the discounted dollar amount from the regular price without any discounts. LA Metro will receive $1.75 per every 30 minutes from non-passholders if there is no discount for relocation service. Passholders will also pay $1.75 per 30 minutes after the first exemption period. This price is calculated solely by the trip duration minutes. Since the trips from supplying stations to demanding stations do not occur naturally, LA metro needs to support people by discounting the prices, and this generates cost. This study is based on the assumption that discounted trip demands are always met.

There are 972 pairs of stations (36×27) for this problem but 154 pairs are missing by the fact that they were not present in the original dataset. The **big M method** is used for those pairs in the mathematical optimization.

$Supply\_dict. = \{0\text{: } 1012,\ 1\text{: } 143,\ 3\text{: } 107,\ \cdots\}$
$Demand\_dict. = \{2:488,\ 4:39,\ 7:21, \cdots\}$
$Cost\_dict. = \{(0,1):1.75,(0,2):1.75,\cdots\}$
$Distance\_dict. = \{(0,2):0.41,\ (0,4):1.17,\cdots\}$
$Missing\_dict. = \{(1,60):\text{ M},\ (3,7):\text{ M},\cdots\}$

Fig. 4: The data in dictionary formats. (key: Station ID, value: amount)

The supplying station, demanding station, costs are defined for the relocation network problem and Manhattan distances between the stations were calculated during preprocessing procedure. The cost and distance dictionaries are *deep-copied* and updated with 'missing_dict.' to get full pairs of routes also with routes that did not have any trip before. The dictionaries will be used in pyomo with gurobi solver to optimize the relocation service.

V. Optimization

A. Optimization Objective

This optimization aims to find the optimal discount rate for the relocation service which makes the sum of total discount costs and outsourcing costs minimized. With the optimal discount rate, the routes(Starting Station to Ending Station) for the relocation service and the optimal number of discounted trips for each route are specified. (Phase 1) Because LA metro team cannot be sure how the outsourcing cost will fluctuate after the launch of the service, this study addresses the range of outsourcing costs which makes the relocation service remains profitable. (Phase 2)

B. Mathematical Programming

For each pair, parameters $Price[s,d]$ denote the **maximum discount dollar-amount** from the current price policy. $R_{discount} \in [0,1]$ is defined as the **discount rate** for the 'Walk-ups' and $Price[s,d] \times R_{discount}$ is defined as the **reward dollar-amount** for 'Passholders.' Then, this becomes the company's cost for relocation service. The model solver will determine **the amount of bikes to be relocated** over each pair, which will be represented as non-negative integer decision variables $x[s,d]$.

- The discount price for 'walk-ups' and the reward for 'passholders' are $Price[s,d] \times R_{discount}$
- By observing trade-off between discount rates and demands people will use Metro Bike, the realized demand rate is set as a function of the discount rate. Hence, the demand will become $Demand[d] \times f_{realized}(R_{discount})$
- The difference between realized demand and maximum demand (when discount rate is 1) is $(Demand[d] - x[s,d])$. This becomes cost as well.

The problem objective is to minimize the total cost from all supplying stations to all demanding stations.

$$Minimize \sum_{s\in Supply}\sum_{d\in Demand} Price[s,d] \times R_{discount} \times x[s,d] + (price\_per\_mile) \times Distance[s,d] \times (Demand[d] - x[s,d]) \tag{1}$$

$$subject\ to \sum_{d\in Demand} x[s,d] \le \text{Supply}[s],\ \forall s \in Supply \tag{2}$$

$$\sum_{s\in Supply} x[s,d] \ge \text{Demand}[d] \times f_{realized}(R_{discount}),\ \forall d \in Demand \tag{3}$$

$$x[s,d]\ is\ integer \tag{4}$$

Eq. 1 : The first term denotes **Discount Cost** and the second term denotes **outsourcing Cost**
Eq. 2 : Relocation demands from all sources can not exceed the supplying capacity.
Eq. 3 : Relocations to each station must satisfy their demand with respect to the discount rate.

C. Network Assumptions

Before solving the optimization problem with Pyomo and gurobi solver, this network model needs some assumptions. The discount dollar-amount for non-passholders and reward dollar-amount for passholders are considered the same for model simplicity. Also, the optimal moves from supplying stations to demanding stations will always be met when the price is free to users (when the discount rate is 1). Lastly, the discount rate and the relocation demand percentage for each routes are the same. Which means, when the price is discounted by 10%, 10% of people on each route will use the relocation service.

D. Results

The *y-axis* of cost-graphs in fig. 5, 6 and 7 below represents **total cost**, sum of discount and outsourcing costs, of various outsourcing costs with respect to discount rates. The lowest points on each figure are the optimal discount rate at each cost.

1) Optimal discount rate to minimize the total cost: The average outsourcing cost in LA area when moving each bike by outsourcing team at LA Metro is \$1/mile. The graph presents discount rate of **25%** makes the least cost for moving unbalanced bikes a year. The detailed 7663 optimal rides for the relocation service are stored in 'optimal_sol.csv' as below.

Index	Supply	Demand	Optimal Moves
3	0	14	103.0
5	0	17	117.0
7	0	19	15.0
⋮	⋮	⋮	⋮
941	61	52	58.0
961	62	38	587.0

TABLE IV: Optimal Rides when Discount Rate is 25%

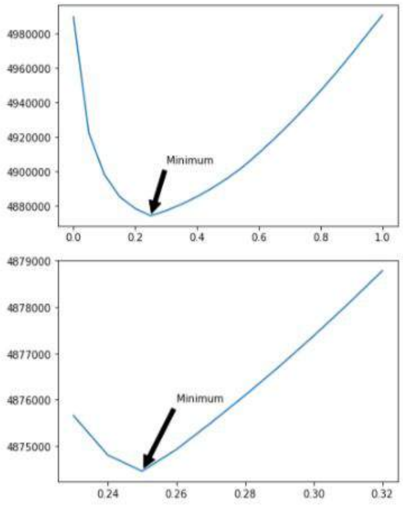

Fig. 5: Optimal Discount Rate when *price_per_mile* is $1/mile.
(Upper: 0% to 100%, LoIr: 23% to 32%)

2) Criteria on no need of the relocation service: The outsourcing costs for the need of relocation service were examined by iterating 10 times from $0.00/mile to $0.45/mile. In fig. 6, there were monotone increase before $0.10/mile which means the least cost occurs when the *x-axis* is at 0 (the discount rate is 0%). Detailed cost-graphs between $0.10/mile and $0.13/mile are in the lower figure. **The minimum outsourcing cost which makes the relocation service profitable is $0.11/mile.**

3) Criteria on the need of providing the relocation service at free price: The outsourcing costs for the free relocation service were examined from $0.50/mile to $5.00/mile by iterating 10 times over each discount rates. In fig. 7 below, monotone decrease was observed after $3.50/mile which means the least cost will be obtained when *x-axis* is at 1 (the discount rate is 100%, free price to users). Detailed cost-graphs between $2.50/mile and $3.40/mile are in the lower figure. **The minimum outsourcing cost at which the relocation service should be free is $3.20/mile.**

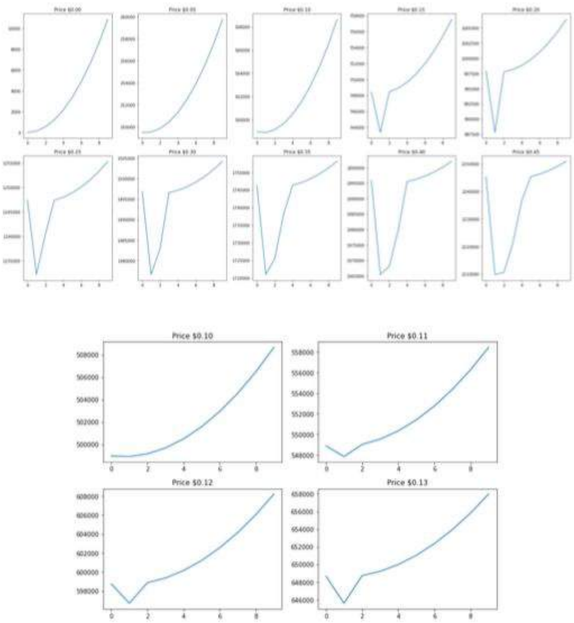

Fig. 6: Cost at which the relocation service is not needed
(Upper: $0.00 to $0.45, Lower: $0.10 to $0.13)

VI. Conclusion

From the results, the optimal discount rate was derived to be 25%, with the current price policy and outsourcing cost, $1/mile. The optimal routes from the supplying stations to the demanding stations and optimal number of rides that will get discounts in each route are stored in output file 'optimal_sol.csv'. Although the analysis is based on the current outsourcing cost, it is possible that the outsourcing cost will fluctuate after the launch of the relocation service. Hence, the range of outsourcing cost needs to be proposed which makes the relocation service profitable. When the outsourcing cost is less than $0.10/mile, the cost-graphs are consistently increasing and the lowest points are placed on 0%. This result is reasonable in that if the outsourcing cost is affordable, there is no need of launching relocation service. On the other hand, when the outsourcing cost is more than $3.10/mile, the cost-graphs are consistently decreasing and the lowest points are placed on 100%. This is also reasonable in that if the outsourcing cost is too high, it is better to encourage people to use relocation service even for free. Finally, if the outsourcing cost after the launch is between $0.10/mile and $3.10/mile, the service remains profitable and worth maintaining the service at proper discount rate.

The assumptions were needed for this problem formulation. With the utilization of more detailed dataset at LA Metro, this model can be more precise to reflect reality. Further studies will include the actual demand variations due to the discount rates and experiment on proper the 'discount rate' for non-passholders and 'reward-rate' for passholders.

Appendix

The detailed codes and procedure are in the Github page.

References

[1] The data and price policy : https://bikeshare.metro.net/about/data/

[2] Diana Jorge, Gonalo Correia, Cynthia Barnhart, Testing the Validity of the MIP Approach for Locating Carsharing Stations in One-way Systems, Procedia - Social and Behavioral Sciences, Volume 54, 2012, Pages 138-148

[3] Daniel Freund, Ashkan Norouzi-Fard, Alice Paul, Shane G. Henderson, and David B. Shmoys. Data-driven rebalancing methods for bike-share systems. Working Paper, 2017.

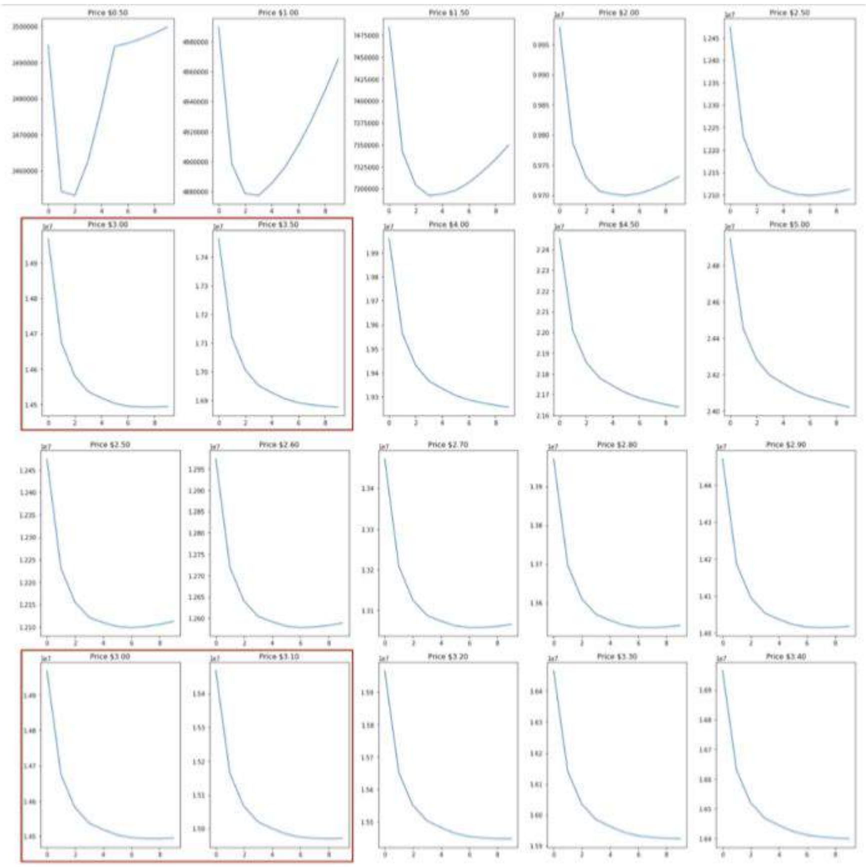

Fig. 7: Cost at which the relocation service is free
(Upper: $0.50 to $5.00, Lower: $2.50 to $3.40)

6. Appendix

● References

[1] Deng, Liu and Sen, Coalescing Data and Decision Sciences for Analytics, USC Data Driven Decisions Lab, 2018
[2] Yunxiao Deng, Suvrajeet Sen, Learning Enabled Optimization: Towards a Fusion of Statistical Learning and Stochastic Programming, USC Data Driven Decisions Lab, 2018
[3] Suvrajeet Sen and Yunxiao Deng, Integrative Analytics with Time Series Data, USC Data Driven Decisions Lab, 2019
[4] Diana Jorge, Gonalo Correia, Cynthia Barnhart, Testing the Validity of the MIP Approach for Locating Carsharing Stations in One-way Systems, Procedia - Social and Behavioral Sciences, Volume 54, 2012, Pages 138-148
[5] Daniel Freund, Ashkan Norouzi-Fard, Alice Paul, Shane G. Hender- son, and David B. Shmoys. Data-driven rebalancing methods for bike- share systems. Working Paper, 2017.

● MnM : Meal Planning for the New Millennium

Detailed: https://github.com/yyul10/stochastic_programming

```
## Data Crawling
from bs4 import BeautifulSoup as bs
from urllib.request import urlopen as url
import pickle
import json
import multiprocessing
import pandas as pd
import numpy as np

##2. Define EP_Recipe class to store all the data."""

class EP_Recipe():
    title = None
    rating = None
    personal_rating = []
    calories = None
    sodium = None
    fat = None
    protein = None

    def get_title(self, page):
        return page.find('h1', {'itemprop': 'name'}).text

    def get_rating(self, page):
        try:
            return float(page.find_all('span', {'class':
'rating'})[-1].text.split('/')[0]) + 1
        except:
            return None

    def build_recipie(self, page):
        #super(EP_Recipe, self).build_recipie(page)
        self.title = self.get_title(page)
        self.rating = self.get_rating(page)
        self.calories = self.get_calories(page)
        self.sodium = self.get_sodium(page)
        self.fat = self.get_fat(page)
        self.protein = self.get_protein(page)
        self.personal_rating = self.get_personal_rating(page)

    def get_calories(self,page):
        try:
```

```
            return
float(page.find('span',{'class':'nutri-data','itemprop':'calories'}).text)
        except:
            return None

    def get_sodium(self,page):
        try:
            return
float(page.find('span',{'class':'nutri-data','itemprop':'sodiumContent'}
).text.split(' ')[0])
        except:
            return None

    def get_fat(self, page):
        try:
            return float(page.find('span', {'class': 'nutri-data', 'itemprop':
'fatContent'}).text.split(' ')[0])
        except:
            return None

    def get_protein(self, page):
        try:
            return float(page.find('span', {'class': 'nutri-data', 'itemprop':
'proteinContent'}).text.split(' ')[0])
        except:
            return None

    def get_personal_rating(self, page):
        try:
            p_ratings = page.findAll('img', {'class': 'fork-rating'})
            p_persons = page.find_all('span', {'class': 'credit'})
            p_r = []
            for i in range(len(p_ratings)):
                temp = [p_persons[i].text.split('/')[0],
int(p_ratings[i]['src'].split('/')[-1].split('_')[0]) + 1]
                p_r.append(temp)
            return p_r
        except:
            return None

    def __init__(self, page):
        print('attempting to build from: '+page)
        try:
```

```
        self.build_recipie(bs(url(page), 'html.parser'))
    except Exception as x:
        print('Could not build from %s, %s'%(page,x))

"""##3. Find all the addresses for recipes and store them in
ep_urls"""

all_url =
['https://www.epicurious.com/recipes-menus/what-to-cook-this-wee
kend-february-22-24-gallery',

'https://www.epicurious.com/recipes-menus/what-to-cook-this-week
end-february-8-10-gallery',

"https://www.epicurious.com/ingredients/acorn-delicata-kabocha-sp
aghetti-squash-winter-recipes-gallery",

'https://www.epicurious.com/recipes-menus/easy-dinner-recipes-for-
cook90-gallery',

'https://www.epicurious.com/recipes-menus/our-favorite-cook90-lun
ches-gallery',

'https://www.epicurious.com/recipes-menus/make-ahead-weeknight-
dinners-stew-soup-freezer-casserole-quick-easy-recipes-gallery']

ep_urls = set()
for i in all_url:
  initializer = url(i)
  res = bs(initializer.read(),"html5lib")
  for div in res.findAll('div', {'class':
'gallery-slide-caption__dek-container'}):
      ep_urls.update([div.find('a')['href']])

"""##4. Scrape the website of recipes and generate the data. Store
the data as recipes_data.json."""
p = multiprocessing.Pool(4)
output = p.map(EP_Recipe,ep_urls)
pickle.dump(output,open('epi_recipes.final','wb'))

data = pickle.load(open('epi_recipes.final','rb'))
ar = []
for i in data:
  ar.append(i.__dict__)
pickle.dump(ar,open('epi_recipe_dict_form.dict','wb'))

with open('recipes_data.json', 'w') as fp:
  json.dump(ar, fp)

"""##5. Read the data"""

df = pd.read_json('recipes_data.json')

"""##6. Data Processing: Delete the row with NaN data and only
consider the recipes with at least four reviews."""

df = df.dropna(axis=0) #delete row with NaN

df = df[df['personal_rating'].map(len) > 9] ##delete the recipes with less than 4 reviews

df.to_csv('nutrition_info.csv')

"""## 7. Analysize the user ratings data"""

p_r = pd.DataFrame(columns=['title', 'user', 'rating'])
p_r

count = 0
for i in range(df.shape[0]):
  for j in df.iloc[i,2]:
    p_r.loc[count] = [df.iloc[i,6],j[0], j[1]]
    count += 1

p_r.to_csv('recipes_users_ratings.csv')

#Count users' rating
cnt = p_r['user'].value_counts()

cnt.describe()
cnt.value_counts()

#
user_reviews2 =
p_r['user'].value_counts()[p_r['user'].value_counts() > 0].index
trun_recipes_user_review = p_r[p_r['user'].isin(user_reviews2)]
trun_recipes_user_review

#Construct a 2D matrix for the recipe-user-rating data
trun_recipes_user_review =
trun_recipes_user_review.drop_duplicates(['user','title'])
trun_recipes_user_review_matrix =
trun_recipes_user_review.pivot(index='user', columns='title',
values='rating')
trun_recipes_user_review_matrix

"""## 8. Construct the final data set for recipes-user-rating."""
# Considering most user only rate one recipe, the recipes-user-rating
matrix is too sparse. Here we see 50 users as one user, by combine
the 50 users' rating as one person's rating.
final_rating_data =
pd.DataFrame(columns=trun_recipes_user_review_matrix.columns)
for i in range(trun_recipes_user_review_matrix.shape[0]//50):
  temp =
trun_recipes_user_review_matrix.iloc[50*i:50*i+40].mean(skipna=
True, axis=0)
  temp.name = 'user' + str(i)
  final_rating_data.loc[i] = temp
final_rating_data

#Save the data to csv file for future use
final_rating_data.to_csv('final_rating_data.csv')

# # Mixed Integer Programming(MIP) for the MnM Project
import glob, os
import numpy as np
import pandas as pd
from pyomo.environ import *
```

```
os.chdir('/Users/yul/Desktop/Github
Fetch/stochastic_programming/5. MnM Project')

nutrition = pd.read_csv('nutrition.csv')
nutrition = nutrition.sort_values(by='title').reset_index()
nutrition = nutrition.drop(['index'],axis=1)
nutrition.head(5)

svd = pd.read_csv('full_afterSVD.csv')
svd = svd.tail(1)
svd = pd.DataFrame(svd.T[2:])
svd = svd.reset_index()
svd = svd.sort_values(by='index').reset_index()
svd.head(5)

nutrition['rating'] = svd[53]
nutrition.head(3)
nutrition = nutrition.sort_values(by='Unnamed: 0')
nutrition.head(3)

# Define parameters
# recipe matrix
rating = nutrition['rating'].tolist()
calories = nutrition['calories'].tolist()
protein = nutrition['protein'].tolist()
fat = nutrition['fat'].tolist()
sodium = nutrition['sodium'].tolist()

# We have determined :
price = [7.2,
5.1,4.3,3.1,4.5,4.3,5.2,9.8,6.1,6.5,4.5,5.5,3.5,13,12.5,4.8,7.5,3.5,6.3,
5]
p_times =
[30,25,25,50,120,10,25,30,30,25,35,20,30,40,20,45,25,60,25,30]
available_time_y = [120,120,120,120,120]
available_time_z = [120,120,120,120,120]

# lower(1) and upper(2) bound of nutritions
c_bound=[2700, 3600]
p_bound= [140 ,200]
f_bound=[150 ,250]
s_bound= [2400 ,3000]
e_bound= 25

# parameter for objective
alpha = 0.01

def mnmplans(cal_rate=0,pro_rate=0,fat_rate=0,sod_rate=0):
    # Define parameters
    cal_ = 1-(cal_rate/100)
    pro_ = 1+(pro_rate/100)
    fat_ = 1-(fat_rate/100)
    sod_ = 1-(sod_rate/100)

    # recipe matrix
    rating = nutrition['rating'].tolist()
    calories = nutrition['calories'].tolist()
    protein = nutrition['protein'].tolist()
    fat = nutrition['fat'].tolist()
```

```
    sodium = nutrition['sodium'].tolist()

    # We have to determine :
    price = [7.2,
5.1,4.3,3.1,4.5,4.3,5.2,9.8,6.1,6.5,4.5,5.5,3.5,13,12.5,4.8,7.5,3.5,6.3,
5]
    p_times =
[30,25,25,50,120,10,25,30,30,25,35,20,30,40,20,45,25,60,25,30]
    available_time_y = [120,120,120,120,120]
    available_time_z = [120,120,120,120,120]

    # lower(1) and upper(2) bound of nutritions
    c_bound=[2700, 3600]
    p_bound= [140 ,200]
    f_bound=[150 ,250]
    s_bound= [2400 ,3000]
    e_bound= 25

    # parameter for objective
    alpha = 0.1

    # Create the model
    model = ConcreteModel()
    opt = SolverFactory('gurobi_ampl')
    model.dual = Suffix(direction=Suffix.IMPORT)

    # Define decision Variables
    X = []; Y = []; Z = []
    for i in range(20):
        X.append('x{0}'.format(i))
        for j in range(5):
            Y.append('y{0}{1}'.format(i,j))
            Z.append('z{0}{1}'.format(i,j))

    model.x = Var(X,domain=NonNegativeIntegers)
    model.y = Var(Y,domain=NonNegativeIntegers)
    model.z = Var(Z,domain=NonNegativeIntegers)

    # Define variables for the amount of nutrient, cooking time and
expenses
    model.fplus = {(i): Var(name='fplus_{0}'.format(i),
domain=NonNegativeReals) for i in range(5)}
    model.fminus = {(i): Var(name='fminus_{0}'.format(i),
domain=NonNegativeReals) for i in range(5)}
    model.eplus = Var(name='eplus', domain=NonNegativeReals)
    model.eminus = Var(name='eminus', domain=NonNegativeReals)
    model.w = Var(name='w', domain=NonNegativeReals)

    # Define Objective
    model.opj = Objective(expr =
sum(rating[i]*model.x['x{0}'.format(i)] for i in range(20))
          - alpha*model.w, sense=maximize)

    #Define constraints
    # -- Meal Constraints
    model.meals = ConstraintList()
    model.meals.add(sum(model.x['x{0}'.format(i)] for i in
range(20)) == 5)
    model.meals.add(sum(model.y['y{0}{1}'.format(i,j)] for i in
range(20) for j in range(5)) <= 5)
   model.meals.add(sum(model.z['z{0}{1}'.format(i,j)] for i in
range(20) for j in range(5)) <= 5)
   for i in range(20):
      model.meals.add(sum(model.y['y{0}{1}'.format(i,j)] +
model.z['z{0}{1}'.format(i,j)] for j in range(5)) ==
model.x['x{0}'.format(i)])

   # -- Nutrition Bounds
   model.nutrition = ConstraintList()
   model.nutrition.add(sum(calories[i]*model.x['x{0}'.format(i)] for
i in range(20)) >= c_bound[0])
   model.nutrition.add(sum(protein[i]*model.x['x{0}'.format(i)] for i
in range(20)) >= pro_*p_bound[0])
   model.nutrition.add(sum(fat[i]*model.x['x{0}'.format(i)] for i in
range(20)) >= f_bound[0])
   model.nutrition.add(sum(sodium[i]*model.x['x{0}'.format(i)] for
i in range(20)) >= s_bound[0])

   model.nutrition.add(sum(calories[i]*model.x['x{0}'.format(i)] for
i in range(20)) <= cal_*c_bound[1])
   model.nutrition.add(sum(protein[i]*model.x['x{0}'.format(i)] for i
in range(20)) <= p_bound[1])
   model.nutrition.add(sum(fat[i]*model.x['x{0}'.format(i)] for i in
range(20)) <= fat_*f_bound[1])
   model.nutrition.add(sum(sodium[i]*model.x['x{0}'.format(i)] for
i in range(20)) <= sod_*s_bound[1])

   # -- Number of recipe
   model.number_rec = ConstraintList()
   model.number_rec.add(sum(model.x['x{0}'.format(i)] for i in
range(20))==
                    sum(model.y['y{0}{1}'.format(i,j)] for i in
range(20) for j in range(5)) +
                    sum(model.z['z{0}{1}'.format(i,j)] for i in
range(20) for j in range(5))
                   )
   model.number_rec.add(sum(model.y['y{0}{1}'.format(i,j)] for i
in range(20) for j in range(5)) -
                 sum(model.z['z{0}{1}'.format(i,j)] for i in range(20)
for j in range(5)) <= 1 )
   model.number_rec.add(-sum(model.y['y{0}{1}'.format(i,j)] for i
in range(20) for j in range(5)) +
                 sum(model.z['z{0}{1}'.format(i,j)] for i in range(20)
for j in range(5)) <= 1 )

   # -- One recipe a day
```

```
   model.one_aday = ConstraintList()
   for j in range(5):
      model.one_aday.add(sum(model.y['y{0}{1}'.format(i,j)] for i
in range(20)) + sum(model.z['z{0}{1}'.format(i,j)] for i in
range(20)) == 1)

   # -- Time Balance
   model.p_time = ConstraintList()
   model.p_time.add(sum(p_times[i]*
(model.y['y{0}{1}'.format(i,j)] - model.z['z{0}{1}'.format(i,j)])
                      for i in range(20) for j in range(5)) <=
model.w)
   model.p_time.add(sum(p_times[i]*
(-model.y['y{0}{1}'.format(i,j)] + model.z['z{0}{1}'.format(i,j)])
                      for i in range(20) for j in range(5)) <=
model.w)

   # -- No redundant recipe constraint
   model.red_rec = ConstraintList()
   for j in range(5):
      model.red_rec.add(sum(model.y['y{0}{1}'.format(i,j)] for i in
range(20)) <= 1)
      model.red_rec.add(sum(model.z['z{0}{1}'.format(i,j)] for i in
range(20)) <= 1)
   for i in range(20):
      model.red_rec.add(sum(model.y['y{0}{1}'.format(i,j)] +
model.z['z{0}{1}'.format(i,j)] for j in range(5)) <= 1)

   # -- Preparation time constraint
   model.prepare_time = ConstraintList()
   for j in range(5):

model.prepare_time.add(sum(p_times[i]*model.y['y{0}{1}'.format(i
,j)] for i in range(20)) <= available_time_y[j])

model.prepare_time.add(sum(p_times[i]*model.z['z{0}{1}'.format(i
,j)] for i in range(20)) <= available_time_z[j])

   results = opt.solve(model,tee=True)
   model.solutions.store_to(results)
   return results

mnmplans().write()
mnmplans(cal_rate=20).write()
```

- **LEO-Wyndor : Advertising and Production**

Detailed: https://github.com/yyul10/stochastic_programming

```
import glob, os
import pandas as pd
import numpy as np
from sklearn import linear_model
from sklearn.model_selection import train_test_split
from pyomo.environ import *
import scipy.stats as st
```

```
import scipy

data = pd.read_csv('/Users/yul/Google 드라이브/USC/2019
Spring/ISE533 Projects/3. LEO-Wyndsor/Advertising.csv')
data = data.drop([data.columns[0] , data.columns[3]],  axis=1)
data.reset_index
data.head(3)
```

```
X = data[['TV','Radio']]
Y = data[['Sales']]

np.random.seed(1)
X_t, X_v, Y_t, Y_v = train_test_split(X, Y, test_size=0.5)
print (X_t.shape, Y_t.shape)
print (X_v.shape, Y_v.shape)

regr = linear_model.LinearRegression()
regr.fit(X_t, Y_t)

print('Intercept: \n', regr.intercept_)
print('Coefficients: \n', regr.coef_)

t_epsilon = regr.predict(X_t) - Y_t
t_epsilon = t_epsilon['Sales'].to_list()

t_epsilon = [round(x,4) for x in t_epsilon]
# t_epsilon

size = 10
g = (t_epsilon[i:i+size] for i in range(0, len(t_epsilon), size))
for i in g:
    print(i)

v_epsilon = regr.predict(X_v) - Y_v
v_epsilon = v_epsilon['Sales'].to_list()
# v_epsilon = v_epsilon.to_list()
v_epsilon = [round(x,4) for x in v_epsilon]

size = 10
g = (v_epsilon[i:i+size] for i in range(0, len(v_epsilon), size))
for i in g:
    print(i)

import matplotlib.pyplot as plt
import statsmodels.api as sm

norm_t_epsilon = [number-np.mean(t_epsilon)/scipy.std(t_epsilon)
for number in t_epsilon]
norm_v_epsilon =
[number-np.mean(v_epsilon)/scipy.std(v_epsilon) for number in
v_epsilon]

qq1 = sorted(norm_t_epsilon)
qq2 = sorted(norm_v_epsilon)

dfqq1 = pd.DataFrame(qq1)
dfqq2 = pd.DataFrame(qq2)

sm.qqplot_2samples(dfqq1,dfqq2)
plt.show()

def outliers(a):
    minpos = a.index(min(a))
    maxpos = a.index(max(a))
    print("The maximum is at position", maxpos)
    print("The minimum is at position", minpos)
outliers(t_epsilon)
```

```
X_t.iloc[17]

X_t.iloc[67]

X_t = X_t.drop(X_t.index[[17,67]])
Y_t = Y_t.drop(Y_t.index[[17,67]])
len(X_t)


# #### Refit the Multiple Linear Regression model.

lm = linear_model.LinearRegression()
lm.fit(X_t, Y_t)

print('Intercept: \n', lm.intercept_)
print('Coefficients: \n', lm.coef_)

beta0 = lm.intercept_.tolist()[0] ; beta1 = lm.coef_.tolist()[0][0] ;
beta2 = lm.coef_.tolist()[0][1]

print("Kruskal Wallis H-test test:")

H, pval = st.mstats.kruskalwallis(norm_t_epsilon,norm_v_epsilon)

print("H-statistic:", H)
print("P-Value:", pval)

if pval < 0.05:
    print("Reject NULL hypothesis - Significant differences exist
between groups.")
if pval > 0.05:
    print("Accept NULL hypothesis - No significant difference
between groups.")


# ### 1. Deterministic LP

# Create the model
dm = ConcreteModel()
opt = SolverFactory('gurobi_ampl')
dm.dual = Suffix(direction=Suffix.IMPORT)

# Define decision Variables
X = ['x1','x2']; Y = ['ya','yb']

dm.x = Var(X,domain=NonNegativeReals)
dm.y = Var(Y,domain=NonNegativeReals)

# Define Objective
dm.obj = Objective(expr = -0.1 * dm.x['x1'] -0.5 * dm.x['x2']+ 3 *
dm.y['ya'] + 5 * dm.y['yb'], sense=maximize)
# Define Constraints
dm.c11 = Constraint(expr = dm.y['ya'] <= 8)
dm.c12 = Constraint(expr = 2 * dm.y['yb'] <= 24)
dm.c13 = Constraint(expr = 3 * dm.y['ya'] + 2 * dm.y['yb'] <= 36)
dm.c14 = Constraint(expr = dm.y['ya'] + dm.y['yb'] - beta1 *
dm.x['x1'] - beta2 * dm.x['x2']
                  <= beta0)
```

```
dm.c21 = Constraint(expr = dm.x['x1'] + dm.x['x2'] <= 200)
dm.c22 = Constraint(expr = dm.x['x1'] - 0.5 * dm.x['x2'] >= 0)

dm.c31 = Constraint(expr = dm.x['x1'] >= min(X_t['TV'].to_list()))
dm.c32 = Constraint(expr = dm.x['x1'] <= max(X_t['TV'].to_list()))
dm.c33 = Constraint(expr = dm.x['x2'] >=
min(X_t['Radio'].to_list()))
dm.c34 = Constraint(expr = dm.x['x2'] <=
max(X_t['Radio'].to_list()))

results = opt.solve(dm, tee=True)
dm.display()
dm.obj.expr()

d_validation_list = []

x1 = value(dm.x['x1'])
x2 = value(dm.x['x2'])

for i in range(len(v_epsilon)):
    # Create the model
    dm = ConcreteModel()
    opt = SolverFactory('gurobi_ampl')
    dm.dual = Suffix(direction=Suffix.IMPORT)

    # Define decision Variables
    Y = ['ya','yb']

    dm.y = Var(Y,domain=NonNegativeReals)

    # Define Objective
    dm.obj = Objective(expr = -0.1 * x1 -0.5 * x2 + 3 * dm.y['ya'] + 5
* dm.y['yb'], sense=maximize)
    # Define Constraints
    dm.c11 = Constraint(expr = dm.y['ya'] <= 8)
    dm.c12 = Constraint(expr = 2 * dm.y['yb'] <= 24)
    dm.c13 = Constraint(expr = 3 * dm.y['ya'] + 2 * dm.y['yb'] <= 36)
    dm.c14 = Constraint(expr = dm.y['ya'] + dm.y['yb'] - beta1 * x1 -
beta2 * x2  <= beta0+v_epsilon[i])
    results = opt.solve(dm)
    d_validation_list.append(dm.obj.expr())

d_validation_list[:5]

d_ci = st.t.interval(0.95, len(d_validation_list)-1,
loc=np.mean(d_validation_list),
            scale=st.sem(d_validation_list))
print(d_ci)


# ### 2. SP with Empirical Additive Errors

# ad: Annotated with location of stochastic rhs entries
#       for use with pysp2smps conversion tool.

import itertools
import random

from pyomo.core import *
```

```
from pyomo.pysp.annotations import
(PySP_ConstraintStageAnnotation,
                                PySP_StochasticRHSAnnotation)
# Define the probability table for the stochastic parameters
i=0
d1_rhs_table = [3.9323, 2.6324, -3.1408, -0.3789, 3.7175, -0.7154,
-0.0245, 2.5941, -0.0305, -0.3112,
              -2.01, 2.7088, -0.4833, -1.6373, -3.0267, 1.3714, -1.3194,
10.6709, -1.9522, 1.2065,
              0.2786, -0.3919, -2.7767, -0.2428, -1.8458, -0.4796,
0.1375, -0.6534, -0.9684, -1.7311,
              -2.2169, 0.9958, -2.405, 2.6091, -1.6641, 1.167, 4.8966,
2.3375, -1.0663, -0.6731,
              -1.8048, 2.8722, -0.9826, 3.1187, -1.295, -2.0326, 0.1602,
-0.8889, -0.3146, -0.3717,
              -2.4787, -0.0625, -2.6977, 1.2482, -1.7434, -1.1329,
3.3738, -1.6965, -0.2341, 3.3963,
              2.048, 0.0905, 1.1836, 0.4437, -2.9092, -1.0815, -2.6525,
-3.1664, 2.3964, -0.5076,
              1.2734, -1.7652, -0.9546, -1.2779, -1.7421, -1.0183,
-0.4392, 1.0994, -0.0678, 1.4886,
              -1.1482, -0.0366, 0.1029, -1.4013, 2.4265, -0.8231,
0.0363, 4.9604, 4.2461, 1.3526,
              -2.0547, -2.0218, -1.4739, -1.495, -1.509, -0.5191,
-0.0872, 1.9653, -1.6092, 1.1035]

num_scenarios = len(d1_rhs_table)
scenario_data = dict(('Scenario'+str(i), (d1val))
                   for i, (d1val) in
                  enumerate(d1_rhs_table, 1))

model = ConcreteModel()

model.constraint_stage = PySP_ConstraintStageAnnotation()
model.stoch_rhs = PySP_StochasticRHSAnnotation()

# use mutable parameters so that the right-hand-side can be updated
for each scenario
model.d1_rhs = Param(mutable=True, initialize=0.0)

# first-stage variables
model.x1 = Var(bounds=(0.7,296.4))
model.x2 = Var(bounds=(1.3,49.4))

# second-stage variables
model.y1 = Var(within=NonNegativeReals)
model.y2 = Var(within=NonNegativeReals)


# stage-cost expressions
model.FirstStageCost =
Expression(initialize=(0.1*model.x1+0.5*model.x2))
model.SecondStageCost =
Expression(initialize=(-3*model.y1-5*model.y2))

model.s1 = Constraint(expr= model.x1 - 0.5*model.x2 >= 0)
model.constraint_stage.declare(model.s1, 1)

model.s2 = Constraint(expr= model.x1 + model.x2 <= 200)
model.constraint_stage.declare(model.s2, 1)

model.s4 = Constraint(expr= model.y1 <= 8)
model.constraint_stage.declare(model.s4, 2)

model.s5 = Constraint(expr= 2*model.y2 <=24)
model.constraint_stage.declare(model.s5, 2)

model.s6 = Constraint(expr= 3*model.y1 + 2*model.y2 <= 36)
model.constraint_stage.declare(model.s6, 2)

# one constraint with stochastic right-hand-sides
model.d1 = Constraint(expr = 4.074597924665657 +
0.05119161049901013*model.x1                +
0.2273484491479199*model.x2 - model.y1 - model.y2
>=model.d1_rhs)
model.constraint_stage.declare(model.d1, 2)
model.stoch_rhs.declare(model.d1)


model.obj = Objective(expr=model.FirstStageCost +
model.SecondStageCost + 200)
# To prevent negative values for minimization problem.

def pysp_scenario_tree_model_callback():
    from pyomo.pysp.scenariotree.tree_structure_model import
CreateConcreteTwoStageScenarioTreeModel

    st_model =
CreateConcreteTwoStageScenarioTreeModel(num_scenarios)

    first_stage = st_model.Stages.first()
    second_stage = st_model.Stages.last()

    # First Stage
    st_model.StageCost[first_stage] = 'FirstStageCost'
    st_model.StageVariables[first_stage].add('x1')
    st_model.StageVariables[first_stage].add('x2')

    # Second Stage
    st_model.StageCost[second_stage] = 'SecondStageCost'
    st_model.StageVariables[second_stage].add('y1')
    st_model.StageVariables[second_stage].add('y2')

    return st_model

def pysp_instance_creation_callback(scenario_name, node_names):

    #
    # Clone a new instance and update the stochastic
    # parameters from the sampled scenario
    #
    instance = model.clone()

    d1_rhs_val = scenario_data[scenario_name]
    instance.d1_rhs.value = d1_rhs_val

    return instance
```

```
get_ipython().system('python -m pyomo.pysp.convert.smps -m
/Users/yul/Desktop/EAEmodel.py --basename EAE
\\--output-directory /Users/yul/Desktop/EAE
--symbolic-solver-labels')

# > The objective is 1.540009e+02-200 <br/>  x1 = 1.983564e+02
<br/>  x2 = 1.643590e+00

s1_validation_list = []

x1 = 1.983564e+02
x2 = 1.643590e+00

for i in range(len(t_epsilon)):
    # Create the model
    dm = ConcreteModel()
    opt = SolverFactory('gurobi_ampl')
    dm.dual = Suffix(direction=Suffix.IMPORT)

    # Define decision Variables
    Y = ['ya','yb']
    dm.y = Var(Y,domain=NonNegativeReals)

    # Define Objective
    dm.obj = Objective(expr = -0.1 * x1 -0.5 * x2 + 3 * dm.y['ya'] + 5
* dm.y['yb'], sense=maximize)
    # Define Constraints
    dm.c11 = Constraint(expr = dm.y['ya'] <= 8)
    dm.c12 = Constraint(expr = 2 * dm.y['yb'] <= 24)
    dm.c13 = Constraint(expr = 3 * dm.y['ya'] + 2 * dm.y['yb'] <= 36)
    dm.c14 = Constraint(expr = dm.y['ya'] + dm.y['yb'] - beta1 * x1 -
beta2 * x2
                       <= beta0+t_epsilon[i])
    results = opt.solve(dm)
    s1_validation_list.append(dm.obj.expr())


# #### 2-2 Validation data validate
s2_validation_list = []

x1 = 1.983564e+02
x2 = 1.643590e+00

for i in range(len(v_epsilon)):
    # Create the model
    dm = ConcreteModel()
    opt = SolverFactory('gurobi_ampl')
    dm.dual = Suffix(direction=Suffix.IMPORT)

    # Define decision Variables
    Y = ['ya','yb']
    dm.y = Var(Y,domain=NonNegativeReals)

    # Define Objective
    dm.obj = Objective(expr = -0.1 * x1 -0.5 * x2 + 3 * dm.y['ya'] + 5
* dm.y['yb'], sense=maximize)
    # Define Constraints
    dm.c11 = Constraint(expr = dm.y['ya'] <= 8)
    dm.c12 = Constraint(expr = 2 * dm.y['yb'] <= 24)
    dm.c13 = Constraint(expr = 3 * dm.y['ya'] + 2 * dm.y['yb'] <= 36)
    dm.c14 = Constraint(expr = dm.y['ya'] + dm.y['yb'] - beta1 * x1 -
beta2 * x2 <=beta0+v_epsilon[i])
    results = opt.solve(dm)
    s2_validation_list.append(dm.obj.expr())

print("Kruskal Wallis H-test test:")

H, pval =
st.mstats.kruskalwallis(s1_validation_list,s2_validation_list)
```

```
print("H-statistic:", H)
print("P-Value:", pval)

if pval < 0.05:
    print("Reject NULL hypothesis - Significant differences exist
between groups.")
if pval > 0.05:
    print("Accept NULL hypothesis - No significant difference
between groups.")
```

- **LEO-ELEQUIP : Time Series and Inventory**

```
#AMPL for DLP
var x1 >=0;
var x2 >=0;
var x3 >=0;
var y1 >=0;
var y2 >=0;
var y3 >=0;
var z1 >=0;
var z2 >=0;
var z3 >=0;
var delta1 >=0;
var delta2 >=0;
#var delta3 >=0;
param y_start := 120.83539999999999;
param d1 := 111.3754;
param d2 := 105.8107;
param d3 := 91.53898;
param Ut := 300;

minimize totalcost: x1+x2+x3 + 3*(z1+z2+z3);
#constraints
# U_t = 300
s.t. a1:  y1 = y_start;
s.t. a2:  x1  >=y1 -d1;
s.t. a3:  z1 +y1 >= d1;

s.t. a4:  y2 - x1 -delta1=0;
s.t. a5:  x2  >=y2 -d2;
s.t. a6:  z2 +y2 >= d2;
s.t. a7:  delta1 <= Ut;


s.t. a8:  y3- x2 -delta2 =0;
s.t. a9:  x3  >=y3 -d3;
s.t. a10: z3 +y3 >= d3;
s.t. a11: delta2 <= Ut;

#Pyomo for SP
# coding: utf-8

import itertools
```

```
import random
from pyomo.core import *
from pyomo.pysp.annotations import
(PySP_ConstraintStageAnnotation,
                              PySP_StochasticRHSAnnotation)

import pandas as pd
y_start=110.37
demand = pd.read_csv('1/out_mean5.csv')
demand = list(map(float, list(demand)))

error_term = pd.read_csv('1/out_error5.csv', header = None)
d1_rhs_table = []
d1_rhs_table = list(error_term[0])

num_scenarios = len(d1_rhs_table)
scenario_data = dict(('Scenario'+str(i), (d1val))
                  for i, (d1val) in
                  enumerate(d1_rhs_table, 1))

# Define the reference model
model = ConcreteModel()

# these annotations are required for using this
# model with the SMPS conversion tool
model.constraint_stage = PySP_ConstraintStageAnnotation()
model.stoch_rhs = PySP_StochasticRHSAnnotation()

# right-hand-sides can be updated for each scenario
model.d1_rhs = Param(mutable=True, initialize=0.0)

# first-stage variables
model.delta1 = Var(bounds=(0,300))
model.delta2 = Var(bounds=(0,300))

# second-stage variables
model.y1 = Var(within=NonNegativeReals)
model.z1 = Var(within=NonNegativeReals)
model.x1 = Var(within=NonNegativeReals)

model.y2 = Var(within=NonNegativeReals)
model.z2 = Var(within=NonNegativeReals)
model.x2 = Var(within=NonNegativeReals)

model.y3 = Var(within=NonNegativeReals)
model.z3 = Var(within=NonNegativeReals)
model.x3 = Var(within=NonNegativeReals)

totalCost =
model.x1+model.x2+model.x3+3*(model.z1+model.z2+model.z3)

# stage-cost expressions
model.FirstStageCost =     Expression(initialize=0)
model.SecondStageCost =     Expression(initialize=(totalCost))

model.s11 = Constraint(expr= model.y1 == y_start)
model.constraint_stage.declare(model.s11, 2)

model.s12 = Constraint(expr= model.x1 >=
model.y1-demand[0]-model.d1_rhs)
model.constraint_stage.declare(model.s12, 2)
model.stoch_rhs.declare(model.s12)

model.s13 = Constraint(expr= model.z1 >=
demand[0]+model.d1_rhs - model.y1)
model.constraint_stage.declare(model.s13, 2)
model.stoch_rhs.declare(model.s13)
###############################################################
########################

model.s21 = Constraint(expr= model.y2 == model.x1+
model.delta1)
model.constraint_stage.declare(model.s21, 2)

model.s22 = Constraint(expr= model.x2 >=
model.y2-demand[1]-model.d1_rhs)
model.constraint_stage.declare(model.s22, 2)
model.stoch_rhs.declare(model.s22)

model.s23 = Constraint(expr= model.z2 >=
demand[1]+model.d1_rhs - model.y2)
model.constraint_stage.declare(model.s23, 2)
model.stoch_rhs.declare(model.s23)
###############################################################
########################
model.s31 = Constraint(expr= model.y3 == model.x2+
model.delta2)
model.constraint_stage.declare(model.s31, 2)

model.s32 = Constraint(expr= model.x3 >=
model.y3-demand[2]-model.d1_rhs)
model.constraint_stage.declare(model.s32, 2)
model.stoch_rhs.declare(model.s32)

model.s33 = Constraint(expr= model.z3 >=
demand[2]+model.d1_rhs - model.y3)
model.constraint_stage.declare(model.s33, 2)
model.stoch_rhs.declare(model.s33)
```

```
model.obj = Objective(expr=model.FirstStageCost +
model.SecondStageCost)

def pysp_scenario_tree_model_callback():
    from pyomo.pysp.scenariotree.tree_structure_model import
CreateConcreteTwoStageScenarioTreeModel

    st_model =
CreateConcreteTwoStageScenarioTreeModel(num_scenarios)

    first_stage = st_model.Stages.first()
    second_stage = st_model.Stages.last()

    # First Stage
    st_model.StageCost[first_stage] = 'FirstStageCost'
    st_model.StageVariables[first_stage].add('delta1')
    st_model.StageVariables[first_stage].add('delta2')
  #  st_model.StageVariables[first_stage].add('delta3')
  #  st_model.StageVariables[first_stage].add('delta4')
    #st_model.StageVariables[first_stage].add('delta5')

    # Second Stage
    st_model.StageCost[second_stage] = 'SecondStageCost'
    st_model.StageVariables[second_stage].add('y1')
    st_model.StageVariables[second_stage].add('y2')
    st_model.StageVariables[second_stage].add('y3')
   # st_model.StageVariables[second_stage].add('y4')
    #st_model.StageVariables[second_stage].add('y5')

    st_model.StageVariables[second_stage].add('z1')
    st_model.StageVariables[second_stage].add('z2')
    st_model.StageVariables[second_stage].add('z3')
    #st_model.StageVariables[second_stage].add('z4')
   # st_model.StageVariables[second_stage].add('z5')
    st_model.StageVariables[second_stage].add('x1')
    st_model.StageVariables[second_stage].add('x2')
    st_model.StageVariables[second_stage].add('x3')
 #   st_model.StageVariables[second_stage].add('x4')
 #   st_model.StageVariables[second_stage].add('x5')
    return st_model

def pysp_instance_creation_callback(scenario_name, node_names):

    #
    # Clone a new instance and update the stochastic
    # parameters from the sampled scenario
    #

    instance = model.clone()

    d1_rhs_val = scenario_data[scenario_name]
    instance.d1_rhs.value = d1_rhs_val

    return instance
```

- ## Feasibility Validation of Launching LA Metro Bike Relocation Service

Detailed : https://github.com/yyul10/LA_metro_bike_relocation

```
import glob, os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import geopy.distance
import sklearn
import warnings
warnings.filterwarnings('ignore')

# ### EDA,  Data Preprocessing
data_raw = pd.read_csv('/Users/yul/Desktop/Github Fetch/LA_metro_bike_relocation/metro-bike-share-trip-data.csv')
data_raw = data_raw.dropna(axis=0, how='any')

# Number of each passholder types
mask = data_raw[data_raw['Passholder Type'] == 'Staff Annual'].index
data_raw.drop(mask,inplace=True)
data_raw['Passholder Type'].value_counts()

# Delete Round-Trip categories.
mask = data_raw[data_raw['Trip Route Category'] == 'Round Trip'].index
data_raw.drop(mask,inplace=True)
data_raw['Trip Route Category'].value_counts()

# Replace passholder types to walk-up(0) and passes (1)
ndata = data_raw.copy()
temp = {'Walk-up':0,'Monthly Pass':1,'Flex Pass':1}
ndata = ndata.replace(temp)
ndata['Passholder Type'].value_counts()

# Calculate the manhattan distance between starting stations and ending stations
ndata['Distance'] = ndata.apply(
   (lambda row: 1.414*geopy.distance.distance(
      (row['Starting Station Latitude'], row['Starting Station Longitude']),
      (row['Ending Station Latitude'], row['Ending Station Longitude'])).miles), axis=1)
ndata = ndata.drop(['Trip ID', 'Starting Station Latitude','Starting Station Longitude',
               'Ending Station Latitude','Ending Station Longitude',
                'Starting Lat-Long', 'Ending Lat-Long'
               ,'Plan Duration','End Time','Trip Route Category','Bike ID'], axis=1)
ndata.iloc[:5, :]
ndata = pd.read_csv('ndata.csv')

distance_data = ndata.sort_values(by='Distance',ascending=True).reset_index()
distance_data[distance_data['Distance']>4].head(5)
# we need to exclude station ID 4108
mask4108s = ndata[ndata['Starting Station ID'] == 4108].index
ndata.drop(mask4108s,inplace=True)
mask4108e = ndata[ndata['Ending Station ID'] == 4108].index
```

```
ndata.drop(mask4108e,inplace=True)
ndata['Passholder Type'].value_counts()
ndata['Start Time'] = pd.to_datetime(ndata['Start Time'])
ndata.dtypes

# ### Exploratory Data Analysis
# The most frequent route
temp = pd.crosstab(ndata['Starting Station ID'],ndata['Ending Station ID']) # Most popular trip
temp.stack().index[np.argmax(temp.values)]
temp.iloc[17:21,:10] # The most popular trip counts

import seaborn as sns
top = pd.DataFrame()
top['Station'] = ndata['Starting Station ID'].value_counts().head(15).index
top['Number of Starts']=ndata['Starting Station ID'].value_counts().head(15).values
top['Station'] = top['Station'].astype('category')
top['Station'] = top.Station.cat.remove_unused_categories()

sns.barplot('Station', 'Number of Starts', data = top)
plt.xticks(rotation=40, ha = 'right')
plt.title("Top 15 LA Metro Bike Stations by Number of Starts")
plt.show()

import seaborn as sns
top2 = pd.DataFrame()
top2['Station'] = ndata['Ending Station ID'].value_counts().head(15).index
top2['Number of Ends']=ndata['Ending Station ID'].value_counts().head(15).values
top2['Station'] = top['Station'].astype('category')
top2['Station'] = top.Station.cat.remove_unused_categories()

sns.barplot('Station', 'Number of Ends', data = top2)
plt.xticks(rotation=40, ha = 'right')
plt.title("Top 15 LA Metro Bike Stations by Number of Ends")
plt.show()

trips_df = pd.DataFrame()
trips_df = ndata.groupby(['Starting Station ID','Ending Station ID']).size().reset_index(name = 'Number of Trips')
trips_df = trips_df.sort_values('Number of Trips', ascending = False)
trips_df['Starting Station ID'] = trips_df['Starting Station ID'].astype('str')
trips_df['Ending Station ID'] = trips_df['Ending Station ID'].astype('str')
trips_df["Trip"] = trips_df["Starting Station ID"] + " to " + trips_df["Ending Station ID"]
trips_df = trips_df[:10]
trips_df = trips_df.drop(['Starting Station ID', "Ending Station ID"], axis = 1)
trips_df = trips_df.reset_index()

# Most popular trips
g = sns.barplot('Number of Trips','Trip', data = trips_df)
```

```
plt.title("Most Popular Trips")
for index, row in trips_df.iterrows():
    g.text(row['Number of Trips']-50,index,row['Number of Trips'],
             color='white', ha="center",fontsize = 10)
plt.show()

# Assume passengers are following the average duration minutes for
each pass.
# By mean
ndata['Duration_min'] = np.floor(ndata['Duration']/60)
print(ndata[['Passholder Type','Duration', 'Duration_min',
'Distance']].groupby('Passholder Type').mean())
# By median
print(ndata[['Passholder Type','Duration', 'Duration_min',
'Distance']].groupby('Passholder Type').median())

from sklearn import preprocessing
le1 = preprocessing.LabelEncoder()
le1.fit(ndata['Starting Station ID'])
ndata['Starting Station ID'] = le1.transform(ndata['Starting Station
ID'])
keys1 = le1.classes_
values1 = le1.transform(le1.classes_)
dictionary1 = dict(zip(keys1, values1))

le2 = preprocessing.LabelEncoder()
le2.fit(ndata['Ending Station ID'])
ndata['Ending Station ID'] = le2.transform(ndata['Ending Station
ID'])
keys2 = le2.classes_
values2 = le2.transform(le2.classes_)
dictionary2 = dict(zip(keys2, values2))

import seaborn as sns
sns.heatmap(ndata.corr(),annot=True)
# ### Count the net demand and supply for each stations

# Demand for bikes at each station
d_data = ndata[['Passholder Type','Starting Station
ID']].groupby('Starting Station ID').count()
d_data = d_data.rename(columns={'Starting Station
ID':'ID','Passholder Type':'D'})

# Supply for bikes at each station
s_data = ndata[['Passholder Type','Ending Station
ID']].groupby('Ending Station ID').count()
s_data = s_data.rename(columns={'Ending Station
ID':'ID','Passholder Type':'S'})

# Net Supply stations
# s_data.rename(columns={'Starting Station ID':'ID'})
s_data['D'] = d_data['D']
s_data.head(3)

def calculate_nets(row):
    temp = np.maximum(row['S'] - row['D'],0)
    return temp

s_data['Net'] = s_data.apply(calculate_nets, axis=1)
```

```
s_data = s_data.drop(['D'],axis=1)
s_data.head(5)

# Net Demand stations
# s_data.rename(columns={'Starting Station ID':'ID'})
d_data['S'] = s_data['S']
d_data.head(3)

def calculate_nets2(row):
    temp2 = np.maximum(row['D'] - row['S'],0)
    return temp2

d_data['Net'] = d_data.apply(calculate_nets2, axis=1)
d_data = d_data.drop(['S', 'D'],axis=1)
s_data = s_data.drop(['S'],axis=1)
s_data = s_data[s_data.values.sum(axis=1) != 0]
d_data = d_data[d_data.values.sum(axis=1) != 0]
d_data.head(5)
s_data.head(5)

# Number of Supply stations and Demand stations
print(len(s_data), len(d_data))

# For pyomo formulation, change to the dictionary type
s_data_constraint = s_data.T.to_dict('records')
d_data_constraint = d_data.T.to_dict('records')

s_data_constraint = str(s_data_constraint[0])
d_data_constraint = str(d_data_constraint[0])
d_data_constraint

import ast
s_data_constraint = ast.literal_eval(s_data_constraint)
d_data_constraint = ast.literal_eval(d_data_constraint)

# ### Calculate the full discount price of each route from supply
stations to demand stations
# Full price matrix
cost_matrix = ndata[['Ending Station ID','Starting Station
ID','Duration', 'Duration_min', 'Distance']].groupby(['Ending Station
ID','Starting Station ID'],as_index=False).mean()
cost_matrix['Price'] = 1.75 *
(np.floor(cost_matrix['Duration_min']/30)+1)
cost_matrix[:5]
cost_matrix = cost_matrix[(cost_matrix['Ending Station
ID'].isin(s_data_constraint))&
          (cost_matrix['Starting Station ID'].isin(d_data_constraint))]
# cost_matrix.to_csv('cost_matrix.csv', encoding='utf-8',
index=False)

# Cost Dictionary for pyomo formulation.
cost_constraint = cost_matrix[(cost_matrix['Ending Station
ID'].isin(s_data_constraint))&
          (cost_matrix['Starting Station ID'].isin(d_data_constraint))]
cost_constraint['key']='('+cost_constraint['Ending Station
ID'].astype(str)+' , '+cost_constraint['Starting Station
ID'].astype(str)+')'
cost_constraint =
cost_constraint.drop(['Duration','Duration_min','Distance','Starting
Station ID','Ending Station ID'],axis=1)
cost_constraint.set_index('key',inplace=True)
cost_constraint = cost_constraint.T.to_dict('records')
cost_constraint = cost_constraint[0]
cost_constraint = {ast.literal_eval(key): value
                  for key, value in cost_constraint.items()}

# Distance Dictionary for pyomo formulation.
distance_matrix = ndata[['Ending Station ID','Starting Station
ID','Duration', 'Duration_min', 'Distance']].groupby(['Ending Station
ID','Starting Station ID'],as_index=False).mean()
distance_constraint = distance_matrix[(distance_matrix['Ending
Station ID'].isin(s_data_constraint))&
            (distance_matrix['Starting Station
ID'].isin(d_data_constraint))]
distance_constraint['key']='('+distance_constraint['Ending Station
ID'].astype(str)+' , '+distance_constraint['Starting Station
ID'].astype(str)+')'
distance_constraint =
distance_constraint.drop(['Duration','Duration_min','Starting Station
ID','Ending Station ID'],axis=1)
distance_constraint.set_index('key',inplace=True)
distance_constraint = distance_constraint.T.to_dict('records')
distance_constraint = distance_constraint[0]
# distance_constraint = distance_constraint[0]
distance_constraint = {ast.literal_eval(key): value
                  for key, value in distance_constraint.items()}

print(len(cost_constraint.keys()),len(distance_constraint.keys()))
# Excluding 63rd Station

# Data file
Demand = d_data_constraint
Supply = s_data_constraint
Distance = distance_constraint

import copy
T = copy.deepcopy(cost_constraint)
T_distance = copy.deepcopy(distance_constraint)
missing_pairs = {
  (1,60):200,
  (3,7):200,
  (3,38):200,
  (3,39):200,
  ...}
T.update(missing_pairs)
T_distance.update(missing_pairs)

missing_key_lst = []
for key in T.keys():
    if not key in distance_constraint.keys():

        # Printing difference in
        # keys in two dictionary
        missing_key_lst.append(key)
len(missing_key_lst)


# ## Phase 1 : Total cost optimization for each discount rates
```

```
from pyomo.environ import *

Objective_list = []
for i in np.arange(1.0,-0.05,-0.05):
    # Create an instance of the model
    model = ConcreteModel()
    opt = SolverFactory('gurobi_ampl')
    model.dual = Suffix(direction=Suffix.IMPORT)

    # Define discount_rate and demand_probability
    dc_rate = i
    f_dc = i
    price_per_mile = 1

    # Step 1: Define index sets
    S_list = list(Supply.keys())
    D_list = list(Demand.keys())

    # Step 2: Define the decision
    model.x = Var(S_list, D_list, domain = NonNegativeReals)

    # Step 3: Define Objective
    model.Cost = Objective(expr = sum([T[s,d]*dc_rate*model.x[s,d]
+ price_per_mile * T_distance[s,d] * (Demand[d]-model.x[s,d])
                    for s in S_list for d in D_list]),sense = minimize)

    # Step 4: Constraints
    model.src = ConstraintList()
    for s in S_list:
        model.src.add(sum([model.x[s,d] for d in D_list]) <=
Supply[s])

    model.dmd = ConstraintList()
    for d in D_list:
        model.dmd.add(sum([model.x[s,d] for s in S_list]) ==
Demand[d]*f_dc)

    results = opt.solve(model,tee=True)
    model.solutions.store_to(results)
    Objective_list.append(model.Cost.expr())
Objective_list

plt.plot(np.arange(1.0,-0.05,-0.05),Objective_list)
plt.annotate('Minimum', xy=(0.25, min(Objective_list)),
xytext=(0.3, min(Objective_list)+30000),
            arrowprops=dict(facecolor='black', shrink=0.05),
            )
plt.show()

# ### Detailed from range 22% to 32%.
from pyomo.environ import *

Objective_list_detail = []
for i in np.arange(0.32,0.22,-0.01):
    # Create an instance of the model
    model = ConcreteModel()
    opt = SolverFactory('gurobi_ampl')
    model.dual = Suffix(direction=Suffix.IMPORT)
    # Define discount_rate and demand_probability
    dc_rate = i
    p_dc = i
    price_per_mile = 1
#     miles = 1 # to be determined

    # Step 1: Define index sets
    S_list = list(Supply.keys())
    D_list = list(Demand.keys())

    # Step 2: Define the decision
    model.x = Var(S_list, D_list, domain = NonNegativeReals)

    # Step 3: Define Objective
    model.Cost = Objective(expr = sum([T[s,d]*dc_rate*model.x[s,d]
+ price_per_mile * T_distance[s,d] * (Demand[d]-model.x[s,d])
                    for s in S_list for d in D_list]),sense = minimize)

    # Step 4: Constraints
    model.src = ConstraintList()
    for s in S_list:
        model.src.add(sum([model.x[s,d] for d in D_list]) <=
Supply[s])

    model.dmd = ConstraintList()
    for d in D_list:
        model.dmd.add(sum([model.x[s,d] for s in S_list]) ==
Demand[d]*p_dc)

    results = opt.solve(model,tee=True)
    model.solutions.store_to(results)
    Objective_list_detail.append(model.Cost.expr())

Objective_list_detail

plt.plot(np.arange(0.32,0.22,-0.01),Objective_list_detail)
plt.annotate('Minimum', xy=(0.25, min(Objective_list_detail)),
xytext=(0.26, min(Objective_list_detail)+1500),
            arrowprops=dict(facecolor='black', shrink=0.05),)
plt.show()

from pyomo.environ import *

model = ConcreteModel()
opt = SolverFactory('gurobi_ampl')
model.dual = Suffix(direction=Suffix.IMPORT)

# Define discount_rate and demand_probability
i = 0.25
dc_rate = i
p_dc = i
price_per_mile = 1

# Step 1: Define index sets
S_list = list(Supply.keys())
D_list = list(Demand.keys())

# Step 2: Define the decision
model.x = Var(S_list, D_list, domain = NonNegativeIntegers)
```

```

# Step 3: Define Objective
model.Cost = Objective(
    expr = sum([T[s,d]*dc_rate*model.x[s,d] + price_per_mile *
T_distance[s,d] * (Demand[d]-model.x[s,d])
                for s in S_list for d in D_list]),sense = minimize)

# Step 4: Constraints
model.src = ConstraintList()
for s in S_list:
    model.src.add(sum([model.x[s,d] for d in D_list]) <= Supply[s])

model.dmd = ConstraintList()
for d in D_list:
    model.dmd.add(sum([model.x[s,d] for s in S_list]) >=
Demand[d]*p_dc)

results = opt.solve(model,tee=True)
results.write()
# We can see that this pyomo model is with optimal solutions.

# Store optimal solution into dataframe
optsol = []
for s in S_list:
    for d in D_list:
        optsol.append((s, d, model.x[s,d]()))
cols=['S','D','Optimal Moves']
optsol = pd.DataFrame(optsol,columns=cols)
optsol = optsol[optsol['Optimal Moves']!=0]
opts_list = optsol['S'].unique().tolist()
optd_list = optsol['D'].unique().tolist()
optsol['Optimal Moves'].sum()
# Total movements for every route pairs.

optsol_analysis = optsol.copy()
optsol_analysis = cost_matrix[(cost_matrix['Ending Station
ID'].isin(opts_list))&
            (cost_matrix['Starting Station ID'].isin(optd_list))]
optsol_analysis[optsol_analysis['Price']!=1.75].sort_values(by=['End
ing Station ID']).head(5)
# ### We can conclude that the optimal discount price for
minimizing the total cost is 25%.
# ---

### Phase 2 : Estimation of operation cost to determine relocation
service feasibility.
### (1-1) Criteria on the need of providing relocation service at free
price.

from pyomo.environ import *

Operation_list = []
for j in range(10):
    Operation_list.append([])

# Operation cost range from $0.50 to $5.00
for j in range(10):
    for i in range(10):
        # Create an instance of the model
        model = ConcreteModel()
        opt = SolverFactory('gurobi_ampl')
        model.dual = Suffix(direction=Suffix.IMPORT)

        # Define discount_rate and demand_probability
        dc_rate = i/10
        f_dc = i/10
        price_per_mile = (j+1)/2

        # Step 1: Define index sets
        S_list = list(Supply.keys())
        D_list = list(Demand.keys())

        # Step 2: Define the decision
        model.x = Var(S_list, D_list, domain = NonNegativeReals)

        # Step 3: Define Objective
        model.Cost = Objective(
            expr = sum([T[s,d]*dc_rate*model.x[s,d] + price_per_mile *
T_distance[s,d] * (Demand[d]-model.x[s,d])
                        for s in S_list for d in D_list]),sense = minimize)

        # Step 4: Constraints
        model.src = ConstraintList()
        for s in S_list:
            model.src.add(sum([model.x[s,d] for d in D_list]) <=
Supply[s])

        model.dmd = ConstraintList()
        for d in D_list:
            model.dmd.add(sum([model.x[s,d] for s in S_list]) ==
Demand[d]*f_dc)

        results = opt.solve(model,tee=True)
        model.solutions.store_to(results)
        Operation_list[j].append(model.Cost.expr())
Operation_list

fig = plt.figure(figsize=(20,10))

for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.title('Price ${0:.2f}'.format((i+1)/2))
    plt.plot(range(10), Operation_list[i])
# plt.tight_layout(pad=1, w_pad=0.2, h_pad=0.4)
plt.tight_layout()
plt.show()

# ### (1-2) Detailed : Criteria on the need of providing relocation
service at free price.

from pyomo.environ import *
Operation_list_detail = []
for j in range(10):
    Operation_list_detail.append([])

# Operation cost range from $2.50 to $3.40
for j in range(10):
    for i in range(10):
```

```
        # Create an instance of the model
        model = ConcreteModel()
        opt = SolverFactory('gurobi_ampl')
        model.dual = Suffix(direction=Suffix.IMPORT)

        # Define discount_rate and demand_probability
        dc_rate = i/10
        f_dc = i/10
        price_per_mile = (j+25)/10

        # Step 1: Define index sets
        S_list = list(Supply.keys())
        D_list = list(Demand.keys())

        # Step 2: Define the decision
        model.x = Var(S_list, D_list, domain = NonNegativeReals)

        # Step 3: Define Objective
        model.Cost = Objective(
            expr = sum([T[s,d]*dc_rate*model.x[s,d] +
                        price_per_mile * T_distance[s,d] *
(Demand[d]-model.x[s,d])
                        for s in S_list for d in D_list]),sense = minimize)

        # Step 4: Constraints
        model.src = ConstraintList()
        for s in S_list:
            model.src.add(sum([model.x[s,d] for d in D_list]) <=
Supply[s])

        model.dmd = ConstraintList()
        for d in D_list:
            model.dmd.add(sum([model.x[s,d] for s in S_list]) ==
Demand[d]*f_dc)

        results = opt.solve(model,tee=True)
        model.solutions.store_to(results)
        Operation_list_detail[j].append(model.Cost.expr())

fig2 = plt.figure(figsize=(20,10))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.title('Price ${0:.2f}'.format((i+25)/10))
    plt.plot(range(10), Operation_list_detail[i])

plt.tight_layout()
plt.show()

Operation_list_detail[7] # Price $3.20, monotonic decrease.
Operation_list_detail[6] # Price $3.10, the price goes up at last.

# ### When the operation price is more than 3.10 dollars per mile, it
is better to provide the relocation service free.

# ### (2-1) Criteria on no need of relocation service.
from pyomo.environ import *
Operation_list_nn = []
for j in range(10):
    Operation_list_nn.append([])
```

```
# Operation cost range from $0.00 to $0.45
for j in range(10):
   for i in range(10):
      # Create an instance of the model
      model = ConcreteModel()
      opt = SolverFactory('gurobi_ampl')
      model.dual = Suffix(direction=Suffix.IMPORT)

      # Define discount_rate and demand_probability
      dc_rate = i/10
      f_dc = i/10
      price_per_mile = (j)/20

      # Step 1: Define index sets
      S_list = list(Supply.keys())
      D_list = list(Demand.keys())

      # Step 2: Define the decision
      model.x = Var(S_list, D_list, domain = NonNegativeReals)

      # Step 3: Define Objective
      model.Cost = Objective(
         expr = sum([T[s,d]*dc_rate*model.x[s,d] +
                  price_per_mile * T_distance[s,d] *
(Demand[d]-model.x[s,d])
                  for s in S_list for d in D_list]),sense = minimize)

      # Step 4: Constraints
      model.src = ConstraintList()
      for s in S_list:
         model.src.add(sum([model.x[s,d] for d in D_list]) <=
Supply[s])

      model.dmd = ConstraintList()
      for d in D_list:
         model.dmd.add(sum([model.x[s,d] for s in S_list]) ==
Demand[d]*f_dc)

      results = opt.solve(model,tee=True)
      model.solutions.store_to(results)
      Operation_list_nn[j].append(model.Cost.expr())

fig3 = plt.figure(figsize=(20,10))
for i in range(10):
   plt.subplot(2, 5, i+1)
   plt.title('Price ${0:.2f}'.format((i)/20))
   plt.plot(range(10), Operation_list_nn[i])

plt.tight_layout()
plt.show()

# ### (2-2) Detailed : Criteria on no need of relocation service.
from pyomo.environ import *

Operation_list_nn_detail = []
for j in range(4):
   Operation_list_nn_detail.append([])
```

```
# Operation cost range from $0.10 to $0.13
for j in range(4):
   for i in range(10):
      # Create an instance of the model
      model = ConcreteModel()
      opt = SolverFactory('gurobi_ampl')
      model.dual = Suffix(direction=Suffix.IMPORT)

      # Define discount_rate and demand_probability
      dc_rate = i/10
      f_dc = i/10
      price_per_mile = (j+10)/100

      # Step 1: Define index sets
      S_list = list(Supply.keys())
      D_list = list(Demand.keys())

      # Step 2: Define the decision
      model.x = Var(S_list, D_list, domain = NonNegativeReals)

      # Step 3: Define Objective
      model.Cost = Objective(
         expr = sum([T[s,d]*dc_rate*model.x[s,d] +
                  price_per_mile * T_distance[s,d] *
(Demand[d]-model.x[s,d])
                  for s in S_list for d in D_list]),sense = minimize)

      # Step 4: Constraints
      model.src = ConstraintList()
      for s in S_list:
         model.src.add(sum([model.x[s,d] for d in D_list]) <=
Supply[s])

      model.dmd = ConstraintList()
      for d in D_list:
         model.dmd.add(sum([model.x[s,d] for s in S_list]) ==
Demand[d]*f_dc)

      results = opt.solve(model,tee=True)
      model.solutions.store_to(results)
      Operation_list_nn_detail[j].append(model.Cost.expr())

# Operation_list_detail
fig4 = plt.figure(figsize=(10,10))
for i in range(4):
   plt.subplot(3, 2, i+1)
   plt.title('Price ${0:.2f}'.format((i+10)/100))
   plt.plot(range(10), Operation_list_nn_detail[i])

plt.tight_layout()
plt.show()

# ### If the operation price is less than 10 cents per mile, it is better
not to provide relocation service.
```