MIE1620 Course Project:

Comparative Analysis of Various Solution Methods for the Farming Stochastic Programming Model.

Authors: Qilong Zeng and Yuqi Liu

Table of Contents

1. Introduction and Problem Statement

The goal of this report is to demonstrate the performance of our customized simplex method, customized primal affine scaling algorithm, and an industrial strength simplex solver when solving the framing stochastic programming model. The original stochastic programming model is stated below:

$$\begin{array}{ll}
\text{Min} & 150x_1 + 230x_2 + 260x_3 + 238y_1 + 210y_2 - 170w_1 - 150w_2 - 36w_3 - 10w_4 \\
\text{s.t.} & x_1 + x_2 + x_3 \le 500 \\
& x_1, x_2, x_3 \ge 0 \\
& 2.5x_1 + y_1 - w_1 \ge 200 \\
& 3x_2 + y_2 - w_2 \ge 240 \\
& w_3 + w_4 \le 20x_3 \\
& w_3 \le 6000 \\
& w_{1,2,3,4} \ge 0, y_{1,2,3} \ge 0
\end{array}$$

The baseline performance of our customized simplex method and primal affine scaling algorithm is to solve the problem under 3 scenarios when the average yields +20% and -20%.

$$\begin{array}{ll}
\textbf{min} & 150x_1 + 230x_2 + 260x_3 \\
& +\frac{1}{3}(238y_{11} + 210y_{21} - 170w_{11} - 150w_{21} - 36w_{31} - 10w_{41}) \dashleftarrow \textbf{1st scenario} \\
& +\frac{1}{3}(238y_{12} + 210y_{22} - 170w_{12} - 150w_{22} - 36w_{32} - 10w_{42}) \dashleftarrow \textbf{2nd scenario} \\
& +\frac{1}{3}(238y_{13} + 210y_{23} - 170w_{13} - 150w_{23} - 36w_{33} - 10w_{43}) \dashleftarrow \textbf{3d scenario} \\
\textbf{s. t.} &
\end{array}$$

$$\textbf{(I)}\left\|\begin{array}{l} x_1 + x_2 + x_3 \le 500 \\ x_{1,2,3} \ge 0 \end{array}\right.
\quad
\textbf{(II)}\left\|\begin{array}{l} 3x_1 + y_{11} - w_{11} \ge 200 \\ 3.6x_2 + y_{21} - w_{21} \ge 240 \\ w_{31} + w_{41} \le 24x_3 \\ w_{31} \le 6000 \\ w_{11,21,31,41} \ge 0 \\ y_{11,21,31} \ge 0 \end{array}\right.
\left\|\begin{array}{l} 2.5x_1 + y_{12} - w_{12} \ge 200 \\ 3x_2 + y_{22} - w_{22} \ge 240 \\ w_{32} + w_{42} \le 20x_3 \\ w_{32} \le 6000 \\ w_{12,22,32,42} \ge 0 \\ y_{12,22,32} \ge 0 \end{array}\right.
\left\|\begin{array}{l} 2x_1 + y_{13} - w_{13} \ge 200 \\ 2.4x_2 + y_{23} - w_{23} \ge 240 \\ w_{33} + w_{43} \le 16x_3 \\ w_{33} \le 6000 \\ w_{13,23,33,43} \ge 0 \\ y_{13,23,33} \ge 0 \end{array}\right.$$

$$\begin{array}{ccc} \uparrow & \uparrow & \uparrow \\ \textbf{1st scenario} & \textbf{2nd scenario} & \textbf{3d scenario} \end{array}$$

The report will cover the following parts:

- Scenario Generation
- Methodology:
    - Simplex Method
    - Primal Affine Scaling
    - Industrial Strength Simplex Solver
- Results
- Performance Comparison
- Conclusion

2. Scenario Generation

We wrote two auxiliary functions to generate more pairs of scenarios and transform the new LP into the standard form under the assumption of equal likelihood (each scenario takes place with the same probability). The detailed explanation of the two functions is as follows:

- **add_scenarios_pairs**: This function expands the constraints of the original linear optimization problem under varying scenarios in a parallel way (e.g. if yield_float is 0.1 and num_of_additions is 2, four new scenarios are average yields -10%, +10%, -20%, +20%). It takes 'c', 'A', 'b', 'yield_float', and 'num_of_additions' as inputs and returns the expended coefficient vector, coefficient matrix, and right-hand side vector (b). The expansion process involves:

    1. Keeping the first three elements of 'c' constant and tiling the remaining elements, adjusting them based on the number of additions (e.g. if the number of additions is 2, each scenario takes place with the possibility of 1/5).
    2. Keeping the first element of 'b' constant and tiling the remaining elements based on the number of additions.
    3. Expanding the 'A' matrix by adding zero columns for new variables and rows for each new scenario. The coefficients of average yield in each new scenario are calculated based on the yield float, and coefficients of new variables are updated according to the original 'A' matrix.

- **standard_form**: This function transforms linear programming to standard form with additional slacks. It takes 'c', 'A', 'b', and 'num_of_addtions' as inputs and returns matrixes in the standard form. The transformation process includes the following:

    1. Expanding 'c' by adding zeros to account for slack variables.
    2. Copying 'b'.

3. Creating a slack variable coefficient matrix ‘A_slack’ based on the original and additional scenarios, and concatenating it with ‘A’ to form ‘A_std’

In general, these two functions allow us to generate different scenarios and parameters automatically based on the original problem by modifying the number of additions and yield float.

Note: All pre-processing procedures are based on the original linear programming. Also, the notations indicating the scenarios are slightly different as scenario one represents the original constraints.

3. Methodology

3.1 Simplex Method (Method [1])

Overview of the Simplex method

The Simplex method is a way to find the optimal linear optimization solution. The objective is to maximize or minimize a linear function of several variables subject to linear constraints. This method is solved by generating a feasible solution first and then identifying the optimal direction and step that keeps the point inside of the constraints and also makes the value better.

Simplex method Iteration summary:

Step 0: Initialization

- Transfer linear programming problem in standard form.
- Identify the basic variables as B and non-basic variables as N. Create the initial Basic solution

Step 1: Optimality Check

- Compute the reduced cost $r = c - cT\, B-1N$ *for all q in N.* If $rq \geq 0$ for all *q in N . x(k)* is optimal, STOP, if the reduced cost has a negative value then select for all one *xq* from the non-basis such that *rq* is negative, Go to Step 2.

Step 2: Descent direction generation

- construct d, with the equation $d^q = \begin{bmatrix} -B^{-1}N_q \\ e_q \end{bmatrix}$
- if d>=0 for all, then the LP is unbounded.

Step 3: Step length generation

- Use the minimum ratio test to determine the step length and make sure the x is still feasible after the step. $\alpha = \min_{j \in B} \left\{ -\frac{x_j^{current}}{d_j^q} \mid d_j^q < 0 \right\}$

Step 4: Improved Adjacent Basic Feasible Solution

- Update x by using equation *x(k+1) = x(k) + αdq,* at this time the feasible solution is better

Step 5: Basic update

- For B, N, cB, cN, B_bar, and N_bar, exchange the corresponding value according to in entering variable and leaving variable.
- Back to Step 1 and determine if another iteration is needed.

Two-phase simplex method

The slack variables are added to modify an LP into a standard LP, so Ax>=b becomes Ax - x_s = b, x_s >= 0 where x_s is the slack variable. However, x_s = -b is the basic solution but not feasible since it has constrained x_s >= 0. To solve this question and get a basic feasible solution we should use the two-phase simplex method.

Since our constraints have some >=, to get the initial basic feasible solution, we need to add artificial variables to the standard LP, so it becomes Ax - x_s + x_a = b, x_s >= 0, x_a>=0, where x_a is the artificial variable. After adding artificial variables, we can choose them to be the basis variables with x_a = b, and the initial B=I. By solving the phase one problem, we find it easy to get the basic feasible solution as input for the phase two iteration, since at this time we already have a feasible solution where only variable and slacks have value, and x_s = 0 as we minimize x_s.

Two-phase simplex method steps

Phase - I:

1. Construct a New LP: Form an artificial linear programming (LP) problem by introducing slack and artificial variables
2. Initialization: Start with a basic feasible solution with x_a = b
3. Solve Phase One: Utilize the simplex algorithm to solve the artificial LP problem. This algorithm aims to find a feasible solution to the artificial problem.

Phase - II:

4. Feasibility Check: Verify if a feasible solution was found in Phase I. With only the original variable and slacks.
5. Initialize with Phase One Solution: If a feasible solution was found in Phase One, use that solution as the starting point for Phase Two. Also contain the B, N,xB,xN,cB, and cN in the correct order as the input of phase II input.
6. Solve Phase Two: Apply the primal simplex algorithm to find the optimal solution to the original linear programming problem. The original LP is retrieved by removing the slacks, and the x output contains only original variables

7. Optimality Check: Check if the solution obtained in Phase Two is optimal for the original problem.

Simplex method code review(Appendix I and Appendix II)

- Singular matrix problem
  Since there is a constraint with b=0 when we choose alpha use minimum ratio test, the 0 here confused the system that which is the leaving index because we cannot determine the 0 comes from x=b=0 or comes from x=0. To solve this case, we add a very small value to b to make it not equal to 0. Then it can avoid the singular matrix problem when exchanging the B and N.

- Inverse problem
  Since in simplex method we need to solve B inverse for each iteration, we used np.linalg.slove() instead of np.linalg.inv() because np.linalg.slove() function is optimized for stability and can handle these situations more robustly. And can somehow reduce the singular matrix problem.

3.2 Primal Affine Scaling Algorithm (Method [2])

Overview of the Primal Affine Scaling Algorithm from class lecture slides
The primal affine scaling algorithm (PASA) is one of the interior point methods to find the optimal linear optimization solution. Instead of moving along the edges to check the optimality of each vertex (basic feasible solution), the PASA moves through the interior of the feasible region in good directions (the direction that minimizes the objectives). Each iteration k of the primal affine scaling algorithm when solving linear programming in the standard form involves the following steps:

1. Initialization (only for the first iteration when k = 0): Find an initial feasible interior point ($x^0 > 0$ and $Ax^0 = b$).
2. Compute and check the dual estimate and reduced cost: $w^k = (AX_k^2A^T)^{-1}AX_k^2c$ represents the dual estimate and $r^k = c - A^Tw^k$ the reduced cost. When the reduced cost is great or equal to 0 and the dual estimate is less than a sufficiently small ε, the optimal solution is found. Otherwise, the algorithm will continue.
3. Compute the good and feasible direction: The good and feasible direction is $d_y^k = -X_kr^k$. If $d_y^k > 0$, the LP is unbounded. If $d_y^k = 0$, the current solution is optimal. Otherwise, the algorithm will continue.

4. Compute the step length and move toward the feasible and good direction: The step length is $a_k = min\{\frac{a}{-(d_y^k)_i} \mid (d_y^k)_i < 0\}$. The new interior feasible solution is generated by $x^{k+1} = x^k + a_k X_k d_y^k$. Then, go to step 2 for the next iteration.

It is crucial to initialize the algorithm with a feasible interior point. To ensure our initial interior point is feasible, we can use a two-phase primal affine built upon the primal affine scaling. The two-phase method involves the following steps:

1. Randomly generate a $x^0 > 0$ and calculate $v = b - Ax^0$.
2. if $v = 0$, then $x^0$is interior feasible (which is almost impossible)
3. Solve the Phase I problem as follows using primal affine scaling:

$$min\, u$$
$$s.t.\ Ax + vu = b$$
$$x \geq 0,\ u \geq 0$$

4. Use the optimal solution of the Phase I problem as an initial feasible solution.

Code for Primal Affine Scaling and Two-Phase Method

According to the algorithm explained above, we could generate a function '**primal_affine_scaling**'(Appendix III) to solve the LP using primal affine scaling. The parameters of the function are as follows:

- 'c': Coefficient vector of the objective function in standard form.
- 'A_sd': The constraint matrix (LHS) in standard form.
- 'b_sd': The constraint vector (RHS) in the standard form.
- 'epsilon': Decision duality gap threshold.
- 'a_scale': Step length for each iteration in y-space, a real number between 0 and 1.
- 'x0': Initial interior feasible solution.
- 'max_iteration': Maximum number of iterations after which the algorithm will stop.
- 'phase': Either 'Phase One' or 'Phase Two', indicating the phase of the problem the algorithm is currently solving.

Note: Due to computational loss that is objective according to calculation accuracy, the algorithm would generate infeasible solutions after some iteration. Therefore, the algorithm stops once the new solution is no longer feasible where the latest feasible solution is the best reachable feasible solution.

The function follows the same steps discussed in the previous section. Based on this function, we wrote another function called '**two_phase_primal_affine_scaling**'(Appendix IV) to implement the two-phase method. The description of the function is as follows:

- **Phase - I**:
    1. Initialize: Start with a randomly generated feasible solution. Note: x^0 is generated by randomly selecting n numbers from [0, 1] and multiplying them by 20.
    2. Construct a new LP: Form an artificial LP problem by adding a new variable u, and adjusting the 'A' matrix and 'c' accordingly.
    3. Solve phase one: Use the primal affine scaling algorithm to solve this artificial problem.
- **Phase - II**:
    1. Feasibility Check: Verify if a feasible solution was found in Phase One. If not, the problem is deemed infeasible.
    2. Initialize with Phase One solution: Use the solution from phase one as the starting point for phase two.
    3. Solve phase two: Use the primal affine scaling algorithm to find the optimal solution to the original problem.

Note: The optimal solution from the two-phase method is rounded to the nearest integer to potentially reach a vertex of the feasible region.
If the problem is feasible and bounded, the function returns the optimal solution and the corresponding objective value. Additionally, the optimal solution is rounded to the nearest integer to reach a vertex of the feasible region potentially.

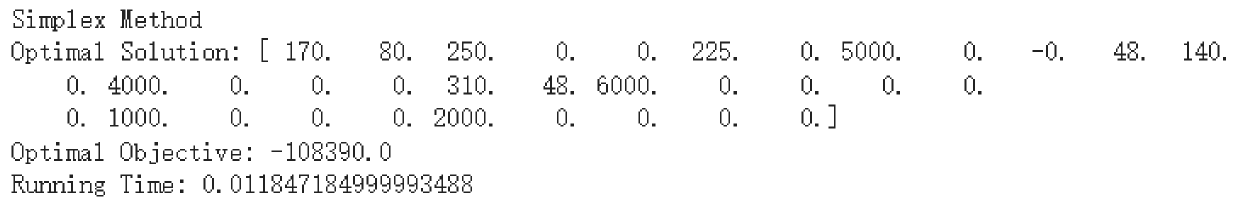
3.3 Industrial Strength Simplex Solver (Method [3])

The industrial strength simplex solver we chose was 'linprog' from 'scipy' library. It is a function to solve learning programming by minimizing a linear objective function subject to linear equality and inequality constraints. In specific, the function would solve LP subject to linear equality using the simplex method by giving the function with corresponding matrixes to the LP in standard form and setting the method to 'simplex'.

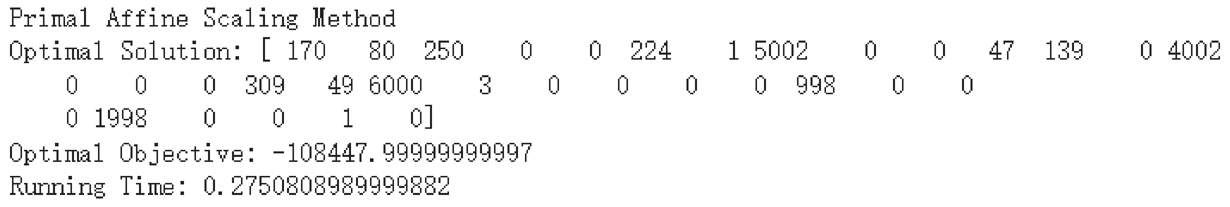
4. Results

Solving the Baseline LP Model (under 3 scenarios when the average yields +20% and -20%)

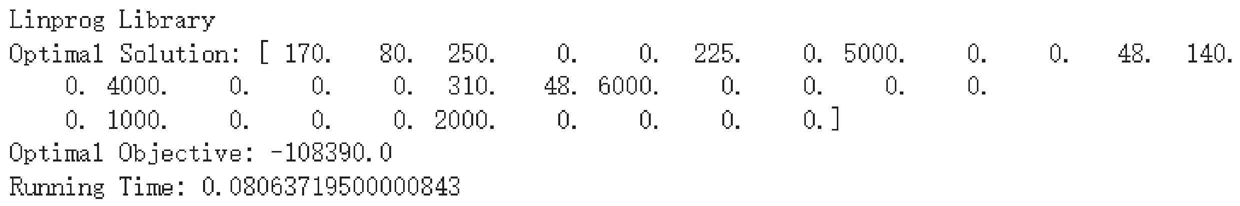
All three methods solved the baseline LP model successfully. The results from the three methods are shown as follows:



```
Simplex Method
Optimal Solution: [ 170.   80.  250.    0.    0.  225.    0. 5000.    0.   -0.   48.  140.
    0. 4000.    0.    0.    0.  310.   48. 6000.    0.    0.    0.    0.
    0. 1000.    0.    0.    0. 2000.    0.    0.    0.    0.]
Optimal Objective: -108390.0
Running Time: 0.011847184999993488
```



```
Primal Affine Scaling Method
Optimal Solution: [ 170   80  250    0    0  224    1 5002    0    0   47  139    0 4002
    0    0    0  309   49 6000    3    0    0    0    0  998    0    0
    0 1998    0    0    1    0]
Optimal Objective: -108447.99999999997
Running Time: 0.2750808989999882
```



```
Linprog Library
Optimal Solution: [ 170.   80.  250.    0.    0.  225.    0. 5000.    0.    0.   48.  140.
    0. 4000.    0.    0.    0.  310.   48. 6000.    0.    0.    0.    0.
    0. 1000.    0.    0.    0. 2000.    0.    0.    0.    0.]
Optimal Objective: -108390.0
Running Time: 0.08063719500000843
```

5. Performance Comparison

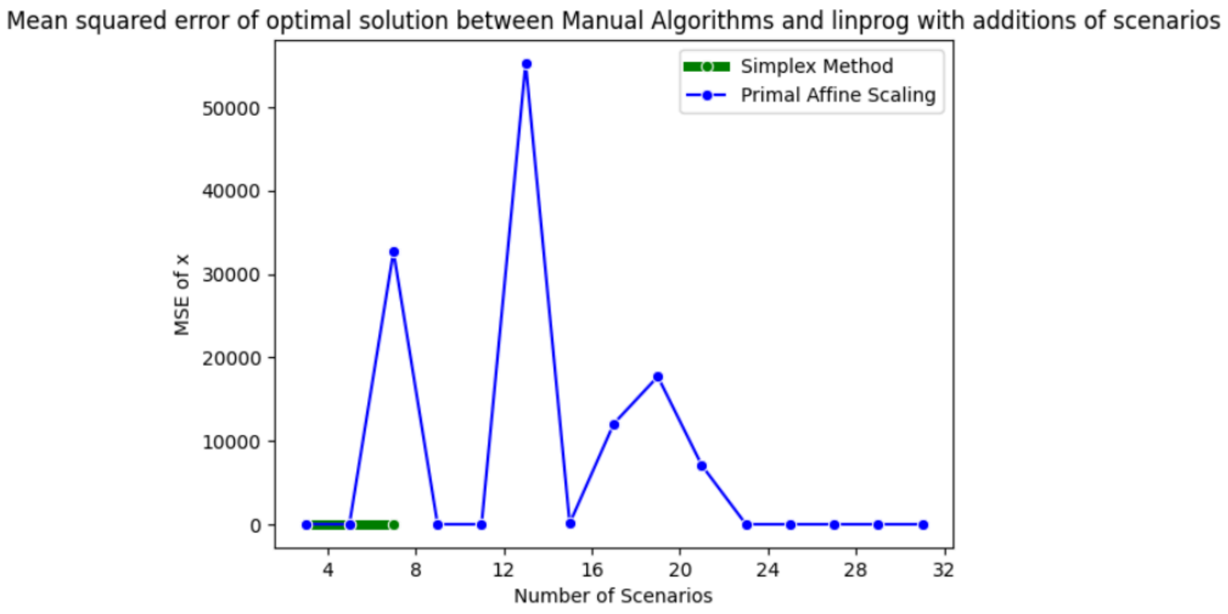
5.1 Performance on Baseline LP Model

The optimal solution and the corresponding objective of each method were significantly close to each other. Especially, the results from the ‘linprog’ and the simplex method were the same. Also, the result of the primal affine scaling method was sensitive to our rounding strategy meaning a sufficiently small deviation from the true result should be expected. From another perspective, the primal affine scaling method had a slightly longer running time than the other two methods while the simplex method outperformed among all the methods.

5.2 Performance when Adding More Scenarios

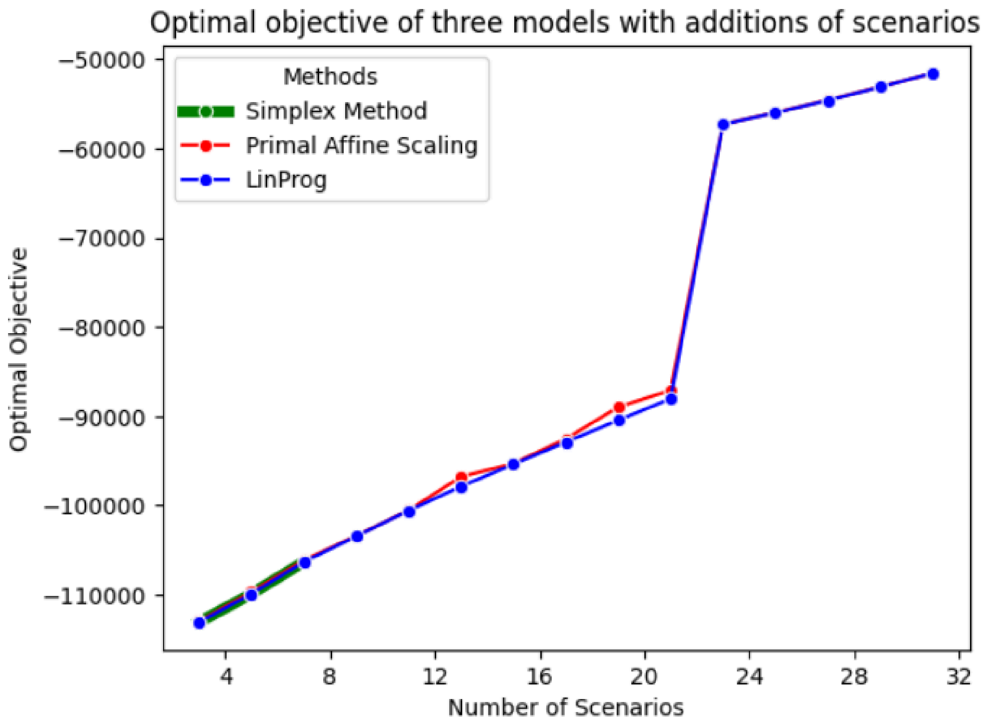
Our simplex method could only handle at most 7 scenarios while our primal affine scaling algorithm successfully solved 31 scenarios when we iteratively generated more scenarios (+,-10% each iteration).

5.2.1 Mean Squared Error (MSE) of Optimal Solutions



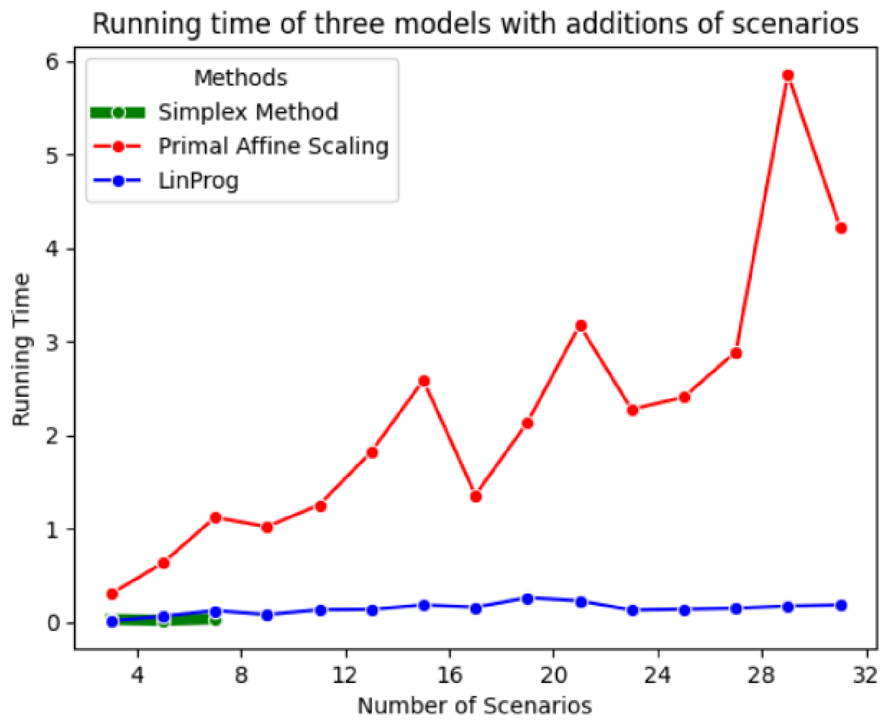
We used MSE to determine the quality of our optimal solutions. The MSE of the optimal solution between our customized methods and the industrial strength simplex solver showed the quality of the optimal solution from the simplex method was better than the primal affine scaling giving the same scenarios. Primal affine scaling's optimal solution significantly deviated from the true optimal solution multiple times. But the MSEs after 23 scenarios were close to 0 meaning the optimal solutions returned to a desired quality after that.

5.2.2 Optimal Objectives of Three Methods



The optimal objectives of the three methods when adding more scenarios indicated that all three methods reached a very similar objective for every number of scenarios although there were some slight deviations for primal affine scaling in some situations.

5.2.3 Running Time of Three Methods



Given the same number of scenarios, the simplex method performed the best in running time while the primal affine scaling was the worst. The running time of the 'linprog' was stable. In contrast, the running time of the primal affine scaling was vibrating when adding more scenarios.

In conclusion, the primal affine scaling was the most unstable method among the three methods. The performance of this method was sensitive to hyperparameters such as the step length in y-space, the epsilon, and the choice of initial interior feasible solution in phase one.

6. Conclusion/Discussion

As for the simplex method, has limitations and may face difficulties in certain situations.The first one is when it meets the constraints >=, if we only add slacks, it cannot give a basic feasible solution as we talked about before. So two-phase simplex method is generated to solve this problem.

Since there contains any constraint such as b=0, we may find it a singular matrix during our steps, it may happens more when the scenarios becomes more and more. When it comes to n scenarios questions, the simplex method does not work so well as it can just solve approximately 7 scenarios. This is because it always meet singular matrix problem, when we exchange B and N to get new B. Since sigular matrix B, we cannot get inverse for it, so it may not apply to several scenarios. Although we added a small value to b to avoid this, it may probably meet the similar situation after tons of iternations.

Another question here is degeneracy. When we generate value of alpha by using the minimum ratio test, it may sometimes get alpha = 0, where the solution lies on the boundary of the feasible region, and the algorithm revisits the same solution multiple times. This can slow down convergence.

In conclusion, the Simplex method meet two major problems, numerical stability and degeneracy. It involves numerous arithmetic operations, and in some scenarios, numerical instability may arise. This can lead to rounding errors and difficulties in reaching the optimal solution.

As for the Affine Scaling method, we found that it run well when we added more scenarios to this problem. But when we compare the accuracy and running time with others, we could easily found that this method is sensitive to some # of scenarios. Which means this method is not robust enough to some degree. Since affine scaling is a path-following algorithm, and its convergence can be influenced by the problem's geometry and the chosen path before. In some scenarios, especially when the problem has degenerate solutions, the method might be difficult to find the optimal solution space. However, this method can be tuned for better performance.

Additionally, by adjusting the scaling factors, affine scaling involves iterative steps where the algorithm moves towards the optimal solution. The sensitivity could arise from difficulties in selecting appropriate scaling factors that maintain numerical stability throughout the optimization process. Affine Scaling is an interior-point method with a random starter point. When we determine the step and direction, the process involves solving a sequence of linear systems in each iteration. The computational complexity of solving linear systems and related operations can contribute to longer running times, especially for larger and more complex problems.

Further exploration

Both these two method happen with computational loss since we have really large matrix and get inverse. To make the better performance, we could use better machine to reduce this kind of inaccuracy. On the other hand, affine scaling is more acceptable when facing with complex situations. When we tune the hyperparameters, it would probably have better performance and become more robust.

Appendix

Appendix I (primal simplex method):

```
def simplex_method(c, A, b, B_bar_=None, N_bar_=None, xB_ = None, xN_ =
None, B_= None, N_=None, phase = None):
   """
   Solve a linear programming (LP) problem using the Simplex method.

   Parameters:
   c (1-D array): Coefficient vector of the objective function to be
minimized.
   A (2-D array): Coefficient matrix of the constraints in canonical form.
   b (1-D array): Right-hand side vector of the constraints in canonical
form.
   B_bar_ (1-D array, optional): Initial basis variable indices. Defaults
to the last m columns of A if None.
   N_bar_ (1-D array, optional): Initial non-basis variable indices.
Defaults to the first n-m columns of A if None.
   xB_ (1-D array, optional): Initial basic variable values. Defaults to b
if None.
   xN_ (1-D array, optional): Initial non-basic variable values. Defaults
to zeros if None.
   B_ (2-D array, optional): Initial basis matrix. Defaults to the last m
columns of A if None.
   N_ (2-D array, optional): Initial non-basis matrix. Defaults to the
first n-m columns of A if None.
   phase (string, optional): The current phase of the Simplex algorithm.
Used for printing messages.

   Returns:
   xB (1-D array): Basic variable values at the optimal solution.
   xN (1-D array): Non-basic variable values at the optimal solution.
   B_bar (1-D array): Basis variable indices at the optimal solution.
   N_bar (1-D array): Non-basis variable indices at the optimal solution.
   B (2-D array): Basis matrix at the optimal solution.
   N (2-D array): Non-basis matrix at the optimal solution.
   optimal (float): Optimal objective value.

   Description:
    The `simplex_method` function implements the Simplex algorithm for
solving linear programming problems in canonical form. The function
iteratively updates the solution by performing pivoting operations. The
method involves checking for optimality, selecting entering and leaving
variables, generating descent direction, unboundedness check, and updating
the basis and solution.

    The algorithm proceeds through the following steps:
    1. Initialization: Set up initial basis and non-basis variables and
matrices.
    2. Optimality Check: Verify if the current solution is optimal.
    3. Pivot Operation: Select an entering variable and a leaving variable,
and perform a pivot to update the basis.
    4. Solution Update: Update the values of basic and non-basic variables,
and the basis and non-basis matrices.
    5. Iteration: Repeat the process until an optimal solution is found or
the problem is deemed unbounded.

    The function terminates when an optimal solution is found, the problem
is identified as unbounded, or the maximum number of iterations is
reached. Informative messages are printed, indicating the current status
of the algorithm.
    """
    m, n = A.shape

    if B_bar_ is None and N_bar_ is None:
        B_bar = np.arange(n-m,n)  # Initial basis variable indices
        N_bar = np.arange(n-m)  # Initial non-basis variable indices
          # Initial basis matrix
        cB = np.ones(m)  # Initial basis coefficients
        cN = np.zeros(n-m)    # Initial non-basis coefficients
          # Initial non-basis matrix

    else:
        B_bar = B_bar_
        N_bar= N_bar_
        cB = c[B_bar]
        cN = c[N_bar]
```

```
if xB_ is None and xN_ is None:
    xB = b.copy() + 1e-24     # Initial basic variables
    xN = np.zeros(n-m)  # Initial non-basic variables

else:
    xB = xB_ + 1e-24    # Initial basic variables
    xN = xN_

if B_ is None and N_ is None:
    B = A[:, -m:].astype(float)
    N = A[:, :-m]

else:
    B = B_   # Initial basic variables
    N = N_


#construct original x
x = np.concatenate((xB, xN))

#limit the iteration in case the calculator break down
max_iterations = 10000
iteration = 0

#iteration part
while iteration < max_iterations:
    # Step 1: Optimality Check
    B_inv_N = np.linalg.solve(B, N)
    r_N = cN - cB.T @ B_inv_N

    #output for result if no direction has better solution
    if np.all(r_N >= 0):
        x = np.concatenate((xB, xN))
        x = x[np.argsort(np.concatenate((B_bar, N_bar)))]
        optimal = c @ x

        print(f"Optimal solution found for {phase}.")
        return xB, xN,B_bar,N_bar,B,N, optimal

    # Select entering variable
```

```
        q = np.argmin(r_N)

        # Step 2: Descent Direction Generation
        d = np.linalg.solve(B, -N[:, q])

        # Step 3: Unbounded Check
        if all(d >= 0):
            print(f"The {phase} problem is unbounded.")
            return None, None, None, None, None, None, None

        # Step 4: Step Length Generation (Minimum Ratio Test)
        mask = d < 0
        result_array = np.zeros_like(d)
        result_array[mask] = xB[mask] / d[:m][mask]
        result_array[mask]
        alpha=min(-result_array[mask])


        # Step 5: Improved Adjacent Basic Feasible Solution
        xB = xB + alpha * d
        xN[q] = alpha

        # Basis Update
        leaving_index = np.argmin(xB)

        #update B and N by exchanging the corresponding value with
entering/leaving index
        temp = B[:, leaving_index].copy()
        B[:, leaving_index] = N[:, q]
        N[:, q] = temp

        #update cB and cN by exchanging the corresponding value with
entering/leaving index
        temp2 = cB[leaving_index]
        cB[leaving_index] = cN[q]
        cN[q] = temp2

        #update xB and xN by exchanging the corresponding value with
entering/leaving index
        temp3 = xB[leaving_index]
        xB[leaving_index] = xN[q]
        xN[q] = temp3

        #update B_bar and N_bar by exchanging the corresponding value with
entering/leaving index
        temp4 = B_bar[leaving_index]
        B_bar[leaving_index] = N_bar[q]
        N_bar[q] = temp4


        iteration += 1


    print("Maximum number of iterations reached.")
    return None, None, None, None, None, None, None
```

Appendix II (two-phase simplex method):

```
def two_phase_simplex_method(c, A, b):
    """
    Solve a linear programming problem using the two-phase simplex method.

    Parameters:
    c (1-D array): Coefficient vector of the objective function to be
minimized.
    A (2-D array): Coefficient matrix of the constraints in canonical form.
    b (1-D array): Right-hand side vector of the constraints in canonical
form.

    Returns:
    optimal_x (1-D array): Optimal solution of the original linear
programming problem.
    optimal_obj (float): Optimal objective value of the original linear
programming problem.

    Description:
    The `two_phase_simplex_method` function implements the two-phase
simplex algorithm to solve linear programming problems. This method is
particularly useful when the initial basic feasible solution is not
readily available.
```

```
    The algorithm involves two main phases:
    1. Phase I:
      - Constructs and solves an auxiliary linear program with artificial
variables to find a basic feasible solution.
      - If artificial variables remain in the basis with non-zero values at
the end of Phase I, the original problem is infeasible.
    2. Phase II:
      - Uses the basic feasible solution obtained from Phase I to solve the
original problem.
      - The original objective function coefficients and constraints are
used.

    The function performs the following steps:
    - Initialize and augment the problem with artificial variables.
    - Solve the auxiliary problem in Phase I using the simplex method.
    - Check for infeasibility based on the solution from Phase I.
    - Remove artificial variables and solve the original problem in Phase
II using the simplex method.

    If the problem is found to be infeasible in Phase I, the function
returns None. Otherwise, it proceeds to Phase II and returns the optimal
solution and the optimal objective value for the original problem.

    Note:
    The function relies on the `simplex_method` function for solving linear
programming problems in both phases.
    """

    m, n = A.shape
    # Create the artificial variables matrix
    A_artificial = np.eye(m)



    # Concatenate the artificial variables to the constraint matrix
    A_extended = np.hstack((A, A_artificial))
    A_org= A.copy()

    # Formulate the Phase I problem
    c_phase1 = np.zeros(n + m)  # Coefficients for the artificial variables
```

```
   c_phase1[np.arange(n, n + m)] = 1  # Minimize the sum of artificial
variables

   # Solve the phase oneusing the simplex method
   xB_phase1, xN_phase1,B_bar_phase1, N_bar_phase1, B_phase1, N_phase1, _
= simplex_method(c_phase1, A_extended, b, B_bar_=None, N_bar_=None, phase
= 'Phase 1')

   if xB_phase1 is None:
       print("Phase I: The problem is infeasible.")
       return None

   if any(xB_phase1[n:] != 0):
       print("Phase I: The problem has artificial variables in the basis
after Phase I.")
       return None

   #remove artifical var from N xN
   N_bar_phase2 = N_bar_phase1[np.argsort(N_bar_phase1)[:-m]]
   xN_phase2=xN_phase1[np.argsort(N_bar_phase1)[:-m]]
   N_phase2=N_phase1[:, np.argsort(N_bar_phase1)[:-m]]
   #print("output",c, A, b,B_bar_phase1, N_bar_phase2,xB_phase1,
xN_phase2, B_phase1,N_phase2)

   #solve phase two problem and get the output for phase two simplex
method
   xB, xN,B_bar,N_bar,B,N, optimal_obj = simplex_method(c, A,
b,B_bar_phase1, N_bar_phase2, xB_phase1, xN_phase2, B_phase1,N_phase2,
phase = 'Phase 2')

   #output optimal value
   A_bar = np.hstack((B_bar, N_bar))
   x_optimal = np.hstack((xB, xN))

   # Pairing elements of A_bar with their corresponding elements in x
   paired = zip(A_bar, x_optimal)

   # Sorting the pairs according to the values in A_bar
   sorted_pairs = sorted(paired)
```

```
    # Extracting the sorted elements of x
    optimal_x = [value for _, value in sorted_pairs]

    # Also, sorting A_bar for consistency
    sorted_A_bar = sorted(A_bar)

    return optimal_x, optimal_obj
```

Appendix III (Primal Affine Scaling):

```
def primal_affine_scaling (c, A_sd, b_sd, epsilon, a_scale, x0,
max_iteration, phase):
  """
  Solve a linear programming (LP) problem using the primal affine scaling
algorithm.

  Parameters:
  c (1-D array): Coefficient vector of the objective function in the LP
problem.
  A_sd (2-D array): The constraint matrix (LHS) in standard form for the
LP problem.
  b_sd (1-D array): The constraint vector (RHS) in standard form for the
LP problem.
  epsilon (float): Decision duality gap threshold. The iteration stops
when the duality gap is smaller than this value.
  a_scale (float): Step length for each iteration in y-space, a real
number between 0 and 1.
  x0 (1-D array): Initial interior feasible solution for starting the
algorithm.
  max_iteration (int): Maximum number of iterations after which the
algorithm will stop.
  phase (string): Either "Phase One" or "Phase Two", indicating the phase
of the problem the algorithm is currently solving.

  Returns:
  x (array): The best interior feasible point found that minimizes the
objective of the LP problem using this algorithm.
```

```
  current_obj (float): The best objective value of the LP problem obtained
with this algorithm.

  Description:
  The `primal_affine_scaling` function implements the primal affine
scaling algorithm to solve linear programming problems. The algorithm
iteratively updates the solution based on the provided parameters and
stops when either the optimal solution is found, the LP is deemed
unbounded, any feasible solution is optimal, a feasible solution cannot be
maintained, or the maximum number of iterations is reached.

  The algorithm follows these steps:
  1. Initialization: Start with the initial feasible solution 'x0'.
  2. Stopping Rule: Check if the current solution is feasible and if the
duality gap is within the specified 'epsilon'.
  3. Direction of Movement: Determine the direction in which to move from
the current point.
  4. Update Solution: Calculate the step length using 'a_scale' and update
the solution 'x'.

  The function prints informative messages regarding the state of the
algorithm, such as achieving optimality, unboundedness, feasibility
issues, or reaching the maximum number of iterations. The final return
values are the best feasible point and the corresponding objective value
found within the allowed iterations or constraints.
  """
  i=0
  # Step one: initiation
  x = x0
  current_obj = c.T @ x
  for i in range(max_iteration):
    # Step two: Stopping rule
    if (x>=0).all() and np.allclose(A_sd @ x, b_sd):
      X = np.diag(x)
      w = np.linalg.solve(A_sd @ np.square(X) @ A_sd.T, A_sd @
np.square(X) @ c)
      r = c - A_sd.T @ w
      duality_gap = x @ r
      if (r >= 0).all() and duality_gap <= epsilon:
        print(f"Optimal solution is found for {phase}.")
        return x, current_obj
      # Step three:
      d_y = -X@r + 1e-24 ## add a sufficiently small number to avoid
dividing zeros to reduce computing time
      if (d_y > 0).all():# Check whether LP is unbounded
        print(f"{phase} LP is unbounded")
        return None, None
      if (d_y == 0).all(): # Check whether any feasible solution is
optimal
        print(f"Every feasible solution in {phase} is optimal.")
        return x, current_obj
      # Step four: Compute Step-length and move toward a feasible and good
direction
      else:
        a = np.min(a_scale / (-d_y [d_y < 0]))
        d_x = X @ d_y
        x_previous = x.copy()
        x = x+np.dot(a, d_x)
        current_obj = c.T @ x
    else:
      print(f"The current solution for {phase} is no longer feasible, so
the solution in the previous iteration is the best feasible solution we
can reach.")
      return x_previous, c.T @ x_previous
  print(f"Maximum number of iterations reached. Here is the current x and
objective for {phase}.")
  return x, current_obj
```

Appendix IV (Two-Phase Primal Affine Scaling):

```
def two_phase_primal_affine_scaling (c_sd, A_sd, b_sd, epsilon, a_scale,
max_iteration):
  """
  Solve a linear programming (LP) problem using the two-phase primal
affine scaling algorithm.

  Parameters:
  c_sd (1-D array): Coefficient vector of the objective function in the LP
problem.
```

```
  A_sd (2-D array): The constraint matrix (LHS) in standard form for the
LP problem.
  b_sd (1-D array): The constraint vector (RHS) in standard form for the
LP problem.
  epsilon (float): Decision duality gap threshold. The iteration stops
when the duality gap is smaller than this value.
  a_scale (float): Step length for each iteration, a real number between 0
and 1.
  max_iteration (int): Maximum number of iterations after which the
algorithm will stop.

  Returns:
  optimal_x (1-D array): The optimal solution (a point extraordinarily
close to or at the optimal BFS) of the LP.
  optimal_obj (float): The optimal objective value of the LP.

  Description:
  The `two_phase_primal_affine_scaling` function implements a two-phase
approach using the primal affine scaling algorithm to solve linear
programming problems. The first phase aims to find a feasible starting
point for the LP problem, and the second phase focuses on optimizing the
objective function starting from this feasible point.

  Phase One:
  1. Initialize: Start with a randomly generated feasible solution.
  2. Construct a new LP: Form an artificial LP problem by adding a new
variable and adjusting the 'A' matrix and 'c' vector accordingly.
  3. Solve Phase One: Use the primal affine scaling algorithm to solve
this artificial problem.

  Phase Two:
  1. Feasibility Check: Verify if a feasible solution was found in Phase
One. If not, the problem is deemed infeasible.
  2. Initialize with Phase One Solution: Use the solution from Phase One
as the starting point for Phase Two.
  3. Solve Phase Two: Use the primal affine scaling algorithm to find the
optimal solution to the original problem.

  The function returns the optimal solution and the corresponding optimal
objective value if the problem is feasible and bounded. Otherwise, it
```

```
returns None, indicating infeasibility or unboundedness of the problem.
Additionally, the optimal solution is rounded to the nearest integer to
potentially reach a vertex of the feasible region.
  """
  # phase one
  np.random.seed(0) # for reproducibility

  ## randomly pick a x>0 and calculate v=b-Ax
  x_0 = np.random.rand(c_sd.shape[0])*20
  v = (b_sd - A_sd @ x_0).reshape(A_sd.shape[0],1)

  ## construct a new phase one LP
  A_artificial = np.concatenate((A_sd, v), axis=1)
  c_artificial = np.hstack((np.zeros(A_sd.shape[1]), 1))
  x_0 = np.hstack((x_0,1))

  ## solve the phase one LP with primal-affine scaling
  x_phase_1, _ = primal_affine_scaling(c_artificial, A_artificial, b_sd,
epsilon, a_scale, x_0, max_iteration, phase = 'Phase One')

  ## check whether the phase one problem is feasible
  if x_phase_1 is None:
    print("Phase 1 is infeasible")
    return None, None
  ## Take the optimal solution from phase 1 to be the initial feasible
soltuion to start phase two
  x0 = x_phase_1[:len(c_sd)]
  # phase two
  optimal_x, optimal_obj = primal_affine_scaling (c_sd, A_sd, b_sd,
epsilon, a_scale, x0, max_iteration, phase = 'Phase Two')

  ## check whether the LP is unbounded
  if optimal_x is None:
    return None, None

  ## round the result to jump onto a vertex
  optimal_x = np.round(optimal_x,0).astype(int)
  optimal_obj = c_sd.T @ optimal_x
  print('The optimal solution for the LP is found.')
  return optimal_x, optimal_obj
```