THE UNIVERSITY OF TEXAS AT AUSTIN

McCOMBS SCHOOL OF BUSINESS

PROJECT 3

R M 294 – Optimization I

MIQP vs. LASSO

A Comparative Analysis for Regression Variable Selection

Group 26

Bhuvana Chandrika Kothapalli (bk24542)
Cate Dombrowski (cd37589)
David Gong(dg38767)
Kapish Krishna Bodapati (kb45953)
Mansi Sharma (ms89743)

Table of Contents

I. Introduction and Context

A. Project Background

In the realm of predictive analytics and statistical modeling, variable selection for regression has long been a central challenge. Historically, direct variable selection through optimization has been deemed arduous due to computational complexities, leading to the emergence of alternative approaches such as Least Absolute Shrinkage and Selection Operator (LASSO) and Ridge regularization. However, recent advancements in optimization software have revolutionized the landscape, enabling the resolution of complex optimization problems, including Mixed Integer Quadratic Programs (MIQP). This significant development has reignited interest in the direct variable selection approach as it offers the potential to identify the optimal set of variables for regression models.

The need to select only a subset of independent variables within a model stems from various practical considerations such as interpretability, reduction in model complexity, and improvement in the model's generalization to new data. In complex real-world datasets, the inclusion of all variables often leads to overfitting, where the model captures noise and idiosyncrasies specific to the training data, thereby compromising its predictive ability on unseen data. By limiting the number of variables in the model, we not only simplify the interpretation of relationships between predictors and the outcome but also mitigate the risk of overfitting, ultimately leading to more robust and generalizable models.

However, the process of selecting the best subset of independent variables is inherently challenging and computationally demanding, falling into the realm of NP-complete problems. As the number of potential variable combinations increases, the computation resources required to evaluate each possible subset grow exponentially, rendering exhaustive search methods impractical, especially for datasets with a large number of predictors. This computational complexity poses a significant obstacle to finding the optimal subset of variables within a reasonable timeframe. Consequently, traditional approaches often resort to heuristic methods or approximation algorithms, which may not guarantee the identification of globally optimal subset. The NP-completeness of the variable selection problem underscores the critical importance of leveraging advanced optimization techniques, such as MIQP, to strike a balance between computational feasibility and the pursuit of the most informative subset of independent variables for regression analysis.

This report conducts a comparative analysis between MIQP and LASSO in the context of variable selection for a simple linear regression problem. Our investigation focuses on their respective capabilities and implications for the process of variable selection within the regression analysis framework. By applying both MIQP and LASSO methodologies on the same dataset, we examine their effectiveness and present the outcomes, offering insights into their respective strengths and weaknesses.

B. Impact of Regularization

Regularization serves as a crucial technique in machine learning, introducing additional terms with a coefficient λ to the initial cost function. The coefficient λ plays a pivotal role in shaping the model's performance. Optimal λ selection significantly influences the results, as more regularization tends to yield lower variance, higher bias, and a reduction in effective model complexity. Striking the right balance becomes paramount in achieving optimal model generalization. Later in this project, we will demonstrate our approach to selecting the optimal regularization strength through the use of cross-validation.

C. Dataset Review

The dataset utilized in this report comprises both training and test sets. The training set consists of 250 rows and 51 columns, encompassing one response variable along with 50 predictor variables (X1-X50). Correspondingly, the test set comprises 50 rows with the same 51 columns.
Our goal is to fit a simple linear regression model to this dataset. The general form of simple linear regression model with m independent variables and one dependent variable is given below.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_m x_m + \varepsilon$$

where:
y: dependent variable,
x_1, x_2, ..., x_m: independent variables,
β_0: intercept,
β_1, β_2, ..., β_m: coefficients, and
ε: irreducible noise that can not be captured by the independent variables

In order to fit this simple linear regression model to our dataset, we have to find the values of all the coefficients of independent variables and the intercept. The values of coefficients and intercept should be in such a way that the errors for each data point is minimum. We can use several techniques like Maximum Likelihood Estimate (MLE) to find the value of these betas. If we assume that the irreducible error terms form a Normal distribution with mean zero and constant variance $N(0, \sigma^2)$, solving for optimal beta values using MLE is the same as minimizing the sum of squared errors (SSE). The SSE is calculated as the sum of the squared differences between the predicted values and the actual values:

$$SSE = \Sigma(y_i - \hat{y}_i)^2, \text{ for } i = 1 \text{ to } n$$

where:
n is the number of data points,
y_i is the actual value, and
$\hat{y}_i$ is the predicted value.

Minimizing the SSE is the same as minimizing the Mean Squared Error (MSE) since SSE is divided by the number of data points, which is a constant. Since, MSE gives the average error for each data point irrespective of the total number of data points, this value can be used to compare model performance for datasets with different numbers of data points. Although in this report, we are using a single dataset, it doesn't matter whether we use SSE or MSE. So, from here on in this report, we will use MSE to compare the performance of models.

$$MSE = \Sigma(y_i - \hat{y}_i)^2 / n$$

There are different techniques to minimize the MSE and depending on our technique, we get different values for the coefficients and different error values. The technique which gives the lowest MSE will be the optimal choice and it will result in the optimal beta values. However, the choice of technique used and the optimal beta values change depending on the problem at hand. Most of the techniques will result in non-zero beta values for all the independent variables. This is not good if there are a lot of independent variables and interpretation will be a huge problem. In such cases, we want the model to yield a sparse beta vector, in which most of the coefficients are zero. This will result in a very simple model that can be interpreted easily. We will get to know which variables have a greater impact on the dependent variable.

We can achieve this using different techniques. We can manually select only a few variables based on the domain knowledge and include only those variables in the regression model. We can use Decision Tree Regression model or Random Forests to get the variable importance and select only those variables. We can use LASSO regularization which pushes some of the beta values to zero by imposing a penalty on the number of independent variables selected. We can solve this as an optimization problem by manually giving the number of variables to be selected. Since, the aim of this report is to compare the techniques of MIQP and LASSO, we will use these two techniques to solve the regression model and compare the results. We use python to solve both MIQP and LASSO. The report contains snippets of python code which is used for better understanding.

II. Direct Variable Selection: MIQP

In this approach, we addressed a Mixed Integer Quadratic Programming problem (MIQP) focused on minimizing an objective, specifically the error term. This objective comprises two parts: (i) Quadratic and (ii) Linear. By analyzing the output, we identified the combination of features that yielded the minimum error, calculated through the ordinary least squares method, measuring the squared difference between predicted and true values. In this section, we elucidate the analysis process and how the features were pinpointed.

To initiate, suppose we have "m" independent variables or features to predict the value of the dependent or target variable Y. Each of these m independent variables is multiplied by a variable βm, referred to as the weight of the respective independent variable (where a 1-unit

increase in Xm results in a βm increase in Y). These m β variables serve as decision variables, along with the inclusion of a β0 variable, known as the intercept term.

In addition to the β variables, it incorporates m z variables, serving as switches for each of the m X variables, controlling whether the corresponding β values are 0 or greater than 0. The Big-M method is employed for this purpose. Furthermore, a hyperparameter k determines the number of non-zero β variables during the optimization, with its optimal value determined through a 10-fold cross-validation approach.

Regarding decision variables, it encompasses 2m+1 variables:

- m β variables corresponding to the m X variables (β1, β2, ..., βm)
- 1 β0 variable
- m z variables corresponding to the m β variables (z1, z2, ..., zm)

Notably, the z variables are of binary data type, taking values of 1 or 0, where 1 signifies that the corresponding β would be non-zero.

The variables are defined in the code as given below.

```
lm = gp.Model()
beta = lm.addMVar(m+1,lb=-np.inf)
z = lm.addMVar(m+1, vtype = 'B') # z[0] is dummy and not used in constraints or objective function
```

We define an empty **gurobi** model **lm** and the coefficients including the intercept are stored in the **beta** variable as a list. Since the coefficients of regression can take any value from -infinity to +infinity, it is necessary to set the lower bound of all the beta values to -infinity. Otherwise, **gurob**i assumes the lower bound to be zero. All the dummy variables which are used as switches for the respective beta values are stored in **z.** Although, the number of z variables is equal to the number of independent variables and the intercept doesn't need a switch, the number of z variables used in the code is equal to **m+1**. This is just for ease of coding and **z[0]** is not used anywhere in the objective function or constraints. It can take any value.

Constraints

The z variables operate as switches, determining whether a β variable obtains a non-zero value. Each β variable is subject to two constraints, known as Big M constraints.

$$-Mz_j \leq \beta_j \leq Mz_j \quad for\, j = 1, 2, 3, \ldots, m$$

```
# Big-M Constraints
lm.addConstrs(beta[i] <= M*z[i] for i in range(1,m+1))
lm.addConstrs(beta[i] >= -M*z[i] for i in range(1,m+1))
```

Here, the value of big-M is very important. This value needs to be the smallest possible large value which doesn't impact the optimal solution. If the value is very big, say in the order of 107 or 108, the value of betas for which the corresponding z value is zero gets affected. If the value

of big-M is small, some of the betas will have a value equal to big-M. So, the optimal value of big-M is chosen by trial and error basis starting with low value. Once we get the beta values, we should check whether any beta values are equal to big-M or not. If any of the beta values is equal to big-M, then double the value and run the model again and get beta values. Repeat the process till none of the beta values are close to big-M. For this dataset, the optimal big-M value was found to be 25.

```
M = 25 # This is decided on trial and error basis after checking the beta values for k = 5
```

The hyperparameter k is the number of variables to be selected by the model and therefore, these variables attain non-zero β values. We have to give the number of variables to be selected manually. So, we run the model for different values of k from 5 to 50 in increments of 5. We select the best value of k using 10 fold cross validation. We split the training data into 10 equal bins randomly. After splitting the training data into 10 bins, we select one bin as the validation set and the remaining 9 bins as the training set. We train the model and obtain the beta values. These beta values are used to calculate the validation set error. For each value of k, we train the model 10 times, each time with a different validation set. We get 10 validation errors for each value of k. We consider the average validation error. Based on the lowest average validation error, we select the best value of k.

$$\sum_{j=1}^{m} z_j \leq k$$

```
# Total number of variables to be selected is less than or equal to k
lm.addConstr(gp.quicksum(z[i] for i in range(1,m+1))<=k)
```

Objective Function

The objective function is given as

$$\min_{\beta} \sum_{i=1}^{n} (\beta_0 + \beta_1 x_{i1} + \cdots + \beta_m x_{im} - y_i)^2$$

```
# Defining the objective function to minimize the sum of squared errors
lm.setObjective(gp.quicksum(
    (gp.quicksum(beta[0]+beta[j]*train_set.iloc[i,j] for j in range(1,m+1))-train_set.iloc[i,0])*
    (gp.quicksum(beta[0]+beta[j]*train_set.iloc[i,j] for j in range(1,m+1))-train_set.iloc[i,0])
    for i in range(len(train_set))), sense=gp.GRB.MINIMIZE)
```

III. Indirect Variable Selection: LASSO

In contrast to the Direct Variable Selection Approach, the LASSO approach performs automatic variable selection by incorporating a penalty on the number of independent variables, forcing some regression coefficients in the model to shrink to zero. As a result, it automatically eliminates less relevant features and is particularly useful for preventing overfitting in regression models.

The LASSO regression formulation is expressed as minimizing the sum of squared differences between the predicted values and actual values, augmented by a penalty term that is the sum of the absolute values of the regression coefficients (excluding the intercept term $\beta 0$).

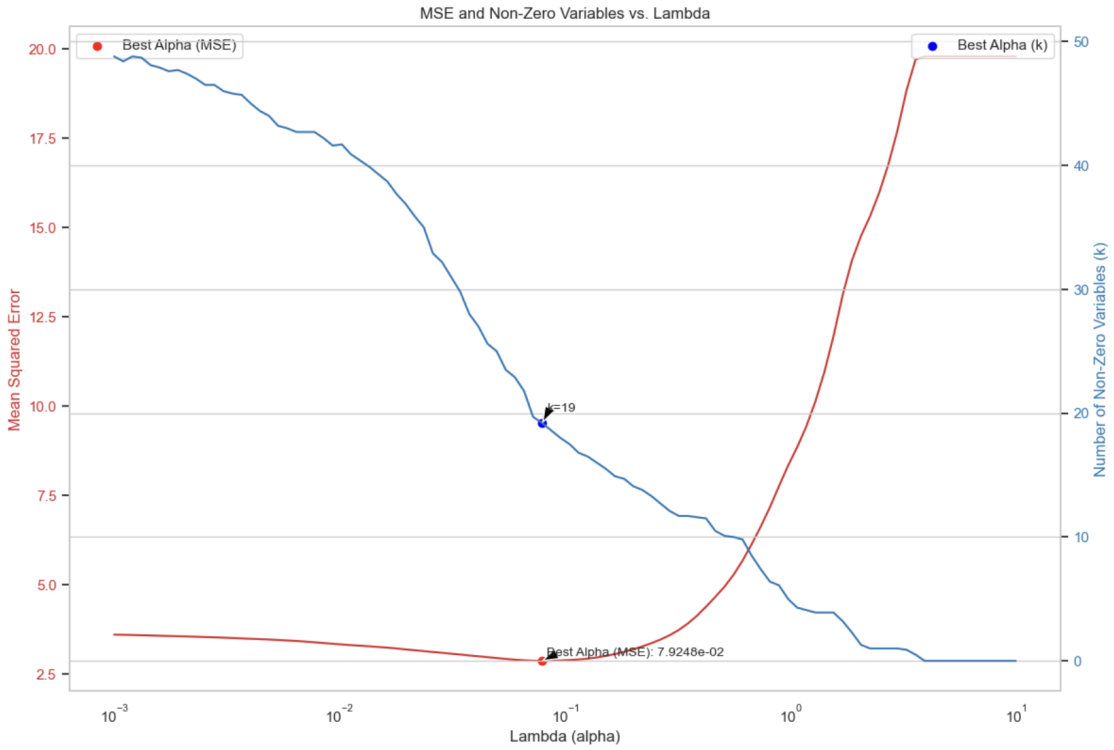
Mathematically, it is represented as:

$$\min_{\beta} \sum_{i=1}^{n} (\beta_0 + \beta_1 x_{i1} + \cdots + \beta_m x_{im} - y_i)^2 + \lambda \sum_{j=1}^{m} |\beta_j|$$

Here, λ is a hyperparameter, which we determine through 10 fold cross-validation. This can be done using the **KFold()** function in **sklearn.** Using 10-fold cross validation, we found the best value of lambda to be 0.079. When λ is appropriately chosen, LASSO helps in reducing the variance of the model by suppressing the influence of less significant predictors. This regularization aids in creating a more robust model that generalizes well to new, unseen data, thus improving predictive performance. When λ is sufficiently large, certain β values are compelled to be zero, effectively inducing sparsity in the model. This property aids in preventing overfitting by "shrinking" the β coefficients closer to zero. Notably, $\beta 0$ (intercept term) is excluded from the λ penalty, ensuring that the model is not penalized for having an intercept term. The code snippet given below shows how the best lambda value is identified using **KFold()**.

```
n_folds = 10

# Use KFold for cross-validation
k_fold = KFold(n_splits=n_folds, shuffle=True, random_state=5)
alphas = 10**np.linspace(-3, 1, 100)

lasso_avg_mse = {}
lasso_non_zero_vars = {}

# For each value of alpha and each fold, compute the mean squared error and non-zero variables
for alpha in alphas:

    # Instantiate a lasso model with the current alpha
    lasso = linear_model.Lasso(alpha=alpha, random_state=5, max_iter=10000)
    avg_mse = 0
    non_zero_vars = 0

    for k, (train, test) in enumerate(k_fold.split(train_X, train_Y)):
        # Use iloc to access specific rows for training
        lasso.fit(train_X.iloc[train], train_Y.iloc[train])

        # Calculate the average mean squared error
        avg_mse += np.mean((lasso.predict(train_X.iloc[test]) - train_Y.iloc[test])**2)

        # Count the number of non-zero coefficients
        non_zero_vars += np.sum(lasso.coef_ != 0)

    # Take the average mean squared error as a metric
    lasso_avg_mse[alpha] = avg_mse / n_folds
    # Take the average number of non-zero variables
    lasso_non_zero_vars[alpha] = non_zero_vars / n_folds

# Find the best value for alpha with minimum mean squared error
best_alpha_lasso = min(lasso_avg_mse, key=lasso_avg_mse.get)
print("Best lasso alpha: {}".format(best_alpha_lasso))
```

The below graph shows how the MSE varies with the increase in lambda value and also how the number of non-zero variables vary with increase in lambda value. The best lambda is selected for which the validation error is minimum.



IV. Methods and Analysis

To compare the results of MIQP and LASSO, we trained both models using the training set and obtaining the beta values and using these beta values on the test set to compute MSE.

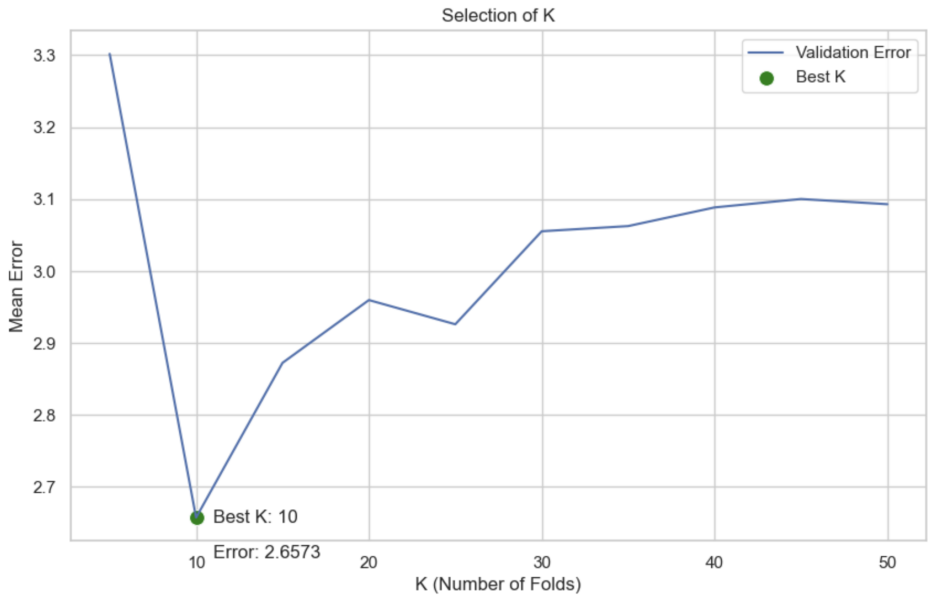
Method 1: MIQP

As stated earlier, we have to manually specify the number of variables to be selected. So, we try different values of k (the number of variables to be selected) and get the best value of k using 10 fold cross validation. The python code used to get the best value of k is given below.

```
# Defining an empty list to store the Average Training Error for each k and Average Validation Error for each k.
AVG_train_error = []
AVG_hold_error = []

# Looping for each value of k in K
for k in K:

    # Defining an empty list to store the training error nd validation error for each fold
    train_error = []
    hold_out_error = []

    # Looping over the 10 folds of training data where one fold is considered as hold_out_set for each case
    for q in range(k_fold):

        # Defining an empty list to store all the beta values, k, index of hold_out_set, training error and validation error
        # for each case
        case = []
        hold_set_index = splits[q] # Defining the hold_out_set
        train_set_index = [i for i in p if i not in splits[q]] # Combining all other splits into training set
        train_set = train_data.loc[train_set_index].reset_index(drop=True) # Resetting the index of train_set
        hold_set = train_data.loc[hold_set_index].reset_index(drop=True) # Resetting the index of hold_set

        lm = gp.Model()
        beta = lm.addMVar(m+1,lb=-np.inf)
        z = lm.addMVar(m+1, vtype = 'B') # z[0] is dummy and not used in constraints or objective function

        # Defining the objective function to minimize the sum of squared errors
        lm.setObjective(gp.quicksum(
            (gp.quicksum(beta[0]+beta[j]*train_set.iloc[i,j] for j in range(1,m+1))-train_set.iloc[i,0])*
            (gp.quicksum(beta[0]+beta[j]*train_set.iloc[i,j] for j in range(1,m+1))-train_set.iloc[i,0])
            for i in range(len(train_set))), sense=gp.GRB.MINIMIZE)

        # Big-M Constraints
        lm.addConstrs(beta[i] <= M*z[i] for i in range(1,m+1))
        lm.addConstrs(beta[i] >= -M*z[i] for i in range(1,m+1))

        # Total number of variables to be selected is less than or equal to k
        lm.addConstr(gp.quicksum(z[i] for i in range(1,m+1))<=k)
```

```
        lm.Params.OutputFlag = 0 # tell gurobi to shut up!!
        lm.Params.TimeLimit = T # Setting the time limit of each model to one hour
        lm.optimize()

        betas = beta.x
        case.extend(betas) # Getting the values of betas and storing them in 'case' list
        case.append(k) # storing the value of 'k' in the same list
        case.append(q) # storing the index of hold_out_set in the same list

        training_error = lm.ObjVal/len(train_set) # Calculating the training error for each case
        train_error.append(training_error) # Storing the training error in 'train_error'
        case.append(training_error) # Storing the training error in 'case' also

        # Calculating the validation error for each case using the beta values obtained from the model
        hold_error = gp.quicksum((gp.quicksum(betas[0]+betas[j]*hold_set.iloc[i,j] for j in range(1,m+1))-hold_set.iloc[i,0])*
                                 (gp.quicksum(betas[0]+betas[j]*hold_set.iloc[i,j] for j in range(1,m+1))-hold_set.iloc[i,0])
                                 for i in range(len(hold_set)))

        hold_error = hold_error.getValue()/len(hold_set)
        hold_out_error.append(hold_error) # Storing the validation error in 'hold_out_error'
        case.append(hold_error) # Storing the validation error in 'case' also

        row = K.index(k)*k_fold+q # calculating the row number of each case in the 'beta_values' dataframe
        beta_values.iloc[row,:] = case # Storing the values of each case in the 'beata_values' dataframe

    avg_train_error = np.mean(train_error) # Calculating the average training error for each value of k
    avg_hold_out_error = np.mean(hold_out_error) # Calculating the average validation error for each value of k

    AVG_train_error.append(avg_train_error) # Storing the average training error for each value of k in a separate list
    AVG_hold_error.append(avg_hold_out_error) # Storing the average validation error for each value of k in a separate list

best_hold_error = min(AVG_hold_error) # Getting the best validation error
print(best_hold_error) # printing the best validation error
best_k = K[AVG_hold_error.index(best_hold_error)] # Getting the value of k for best validation error
print(best_k) # printing the best k

# Export the results to a csv file
beta_values.to_csv('beta_values1.csv')
```

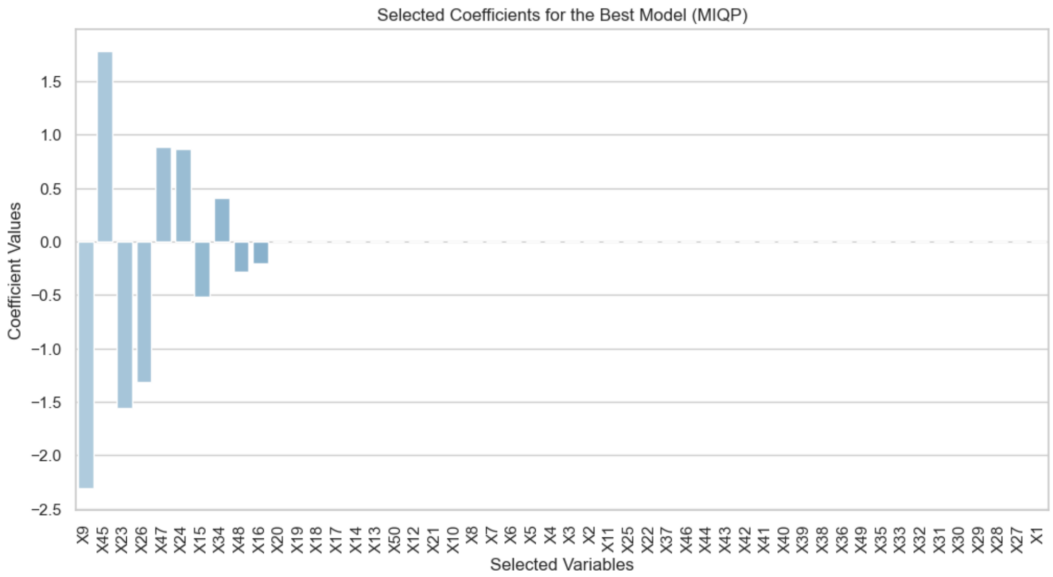
The graph below shows mean validation error values at different values of k.



After getting the best value of k as 10, we use this and fit the model on the entire training set. We get the optimal values of beta, which has 10 non-zero values and all other variables are set to zero. We use these beta values to predict the y values of the test set. Then, we calculate the MSE of the test set. We found that the **training error is 2.39 and the test error is 2.34**. The closeness of these two values indicates that the model is well fit and can accurately predict unseen data. The code snippet below shows how we trained the MIQP model and calculated the MSE of the test set.

```
# Defining another gurobi model 'lm_best' to fit using the entire training set with best k value
k = best_k
lm_best = gp.Model()
beta = lm_best.addMVar(m+1,lb=-np.inf)
z = lm_best.addMVar(m+1, vtype = 'B') # z[0] is dummy and not used in constraints or objective function

# Defining the objective function to minimize the sum of squared errors
lm_best.setObjective(gp.quicksum((beta[0]+gp.quicksum(beta[j]*train_data.iloc[i,j] for j in range(1,m+1))-train_data.iloc[i,0])*
                                 (beta[0]+gp.quicksum(beta[j]*train_data.iloc[i,j] for j in range(1,m+1))-train_data.iloc[i,0])
                                 for i in range(len(train_data))), sense=gp.GRB.MINIMIZE)

# Big-M Constraints
lm_best.addConstrs(beta[i] <= M*z[i] for i in range(1,m+1))
lm_best.addConstrs(beta[i] >= -M*z[i] for i in range(1,m+1))

# Total number of variables to be selected is less than or equal to k
lm_best.addConstr(gp.quicksum(z[i] for i in range(1,m+1))<=k)

lm_best.Params.OutputFlag = 0 # tell gurobi to shut up!!
lm_best.Params.TimeLimit = T
lm_best.optimize()

# Calculate the training error (Mean Squared Error)
best_train_error = lm_best.ObjVal/len(train_data)

# Calculate the test error (Mean Squared Error)
# Now, apply the model to the test data to calculate the predictions
test_predictions = []
for i in range(test_data.shape[0]):
    best_beta = beta.x
    test_predictions.append(best_beta[0] + sum(best_beta[j] * test_data.iloc[i, j] for j in range(1, m + 1)))

test_error = np.mean((np.array(test_predictions) - test_data.iloc[:, 0]) ** 2)

print("MIQP Training Mean Squared Error:", best_train_error)
print("MIQP Test Mean Squared Error:", test_error)
```

Then, we extracted the best coefficients from the model. The graph below shows those beta values for different selected variables.

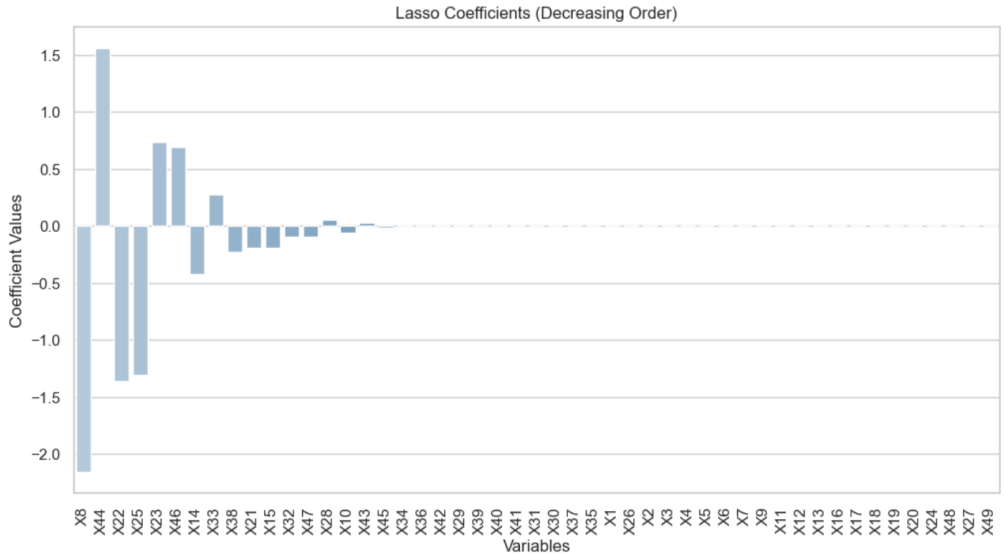


Method 2: LASSO

Similar to MIQP, after getting the best lambda value using 10-fold cross validation, we used the best lambda to fit the LASSO model using the entire training set and get the optimal beta values. These beta values are used to predict the y values for the test set and calculate the MSE of the test set. We found that the test MSE for LASSO is **2.3507** and the number of variables selected is **18**. The code snippet given below shows how we fit the model using the entire training set and calculated MSE of the test set.

```
# Creating the Lasso model
lasso_model = linear_model.Lasso(alpha=best_alpha_lasso, random_state = 5, max_iter = 100000)
lasso_model.fit(train_X, train_Y)


# Getting the predictions and mean squared error
lasso_predictions = lasso_model.predict(test_X)
lasso_mse = mean_squared_error(test_Y, lasso_predictions)
print("Lasso Mean Squared Error:", lasso_mse)

# Getting the coefficients of the Lasso model
lasso_intercept = lasso_model.intercept_
#print("Lasso Intercept:", lasso_intercept)

lasso_coefficients = lasso_model.coef_
#print("Lasso Coefficients:", lasso_coefficients)

# Count the number of selected variables (non-zero coefficients)
num_selected_variables = np.sum(lasso_coefficients != 0)
print("Number of Selected Variables:", num_selected_variables)

lasso_betas = [lasso_intercept] + list(lasso_coefficients)
print(lasso_betas)
```

Below, we visualize the LASSO coefficients on different variables.
LASSO selected 18 important variables for prediction and shrank the coefficients of all other variables to zero as shown below.



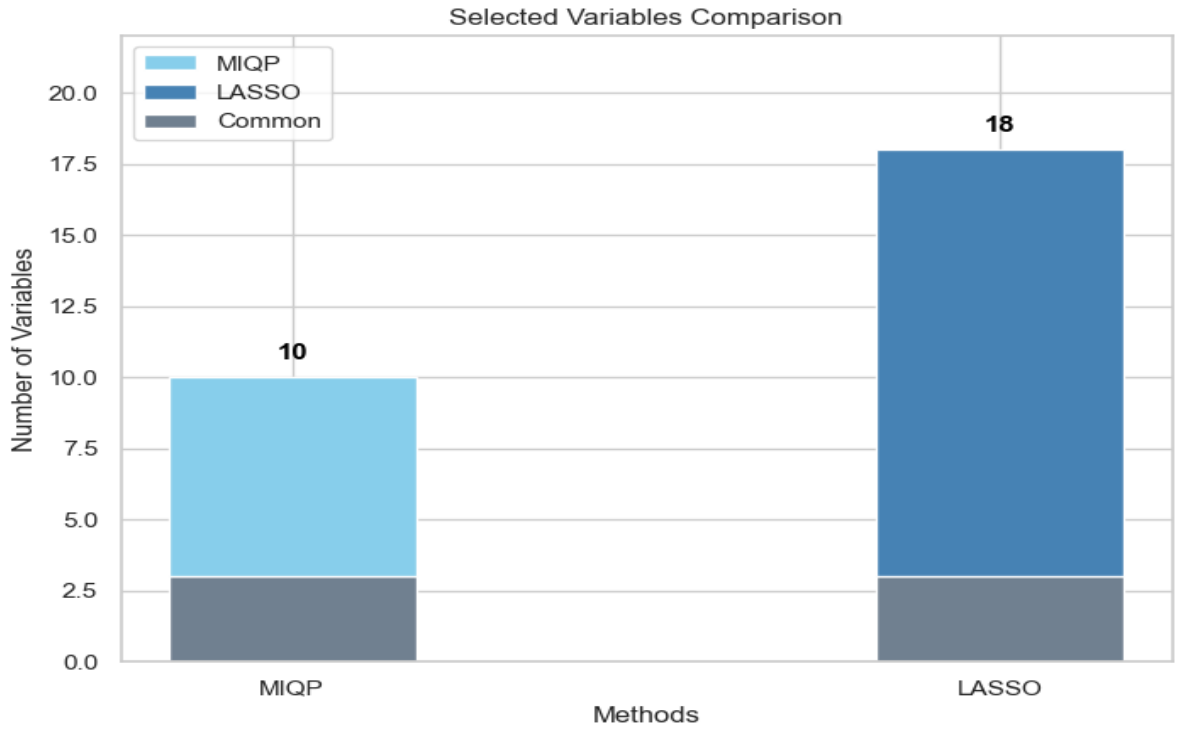
Comparative Variable Selection Analysis: MIQP vs. LASSO

We compared how MIQP and LASSO select variables to understand the different strategies each algorithm uses.

Common Selected Variables: {26, 45, 23}

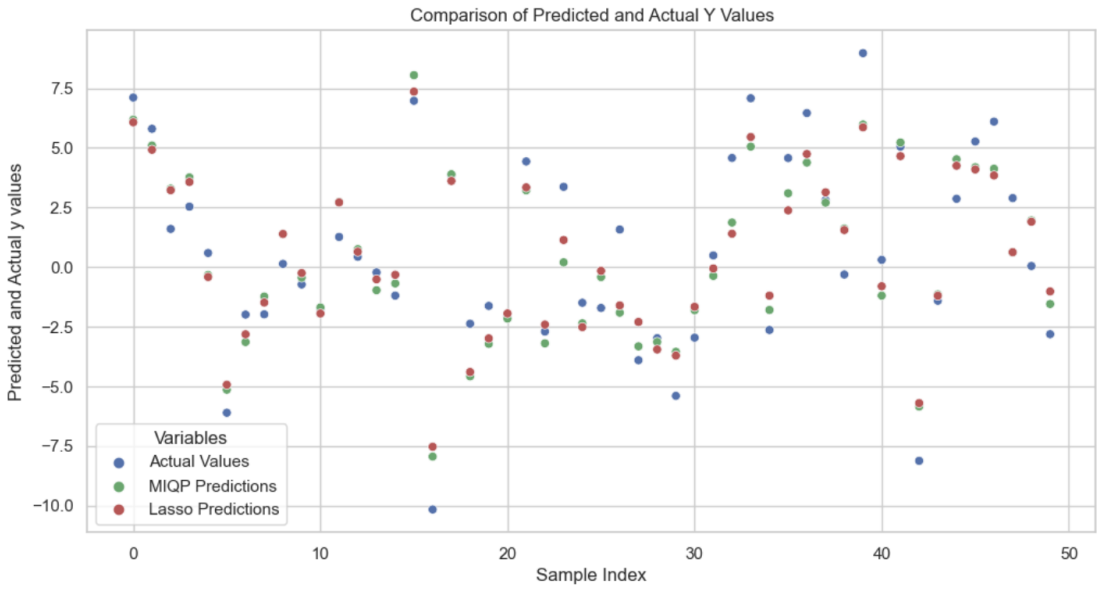
Variables Unique to MIQP Model: {34, 9, 15, 16, 47, 48, 24}

Variables Unique to Lasso Model: {6, 8, 12, 13, 19, 20, 21, 30, 31, 32, 36, 41, 42, 43, 44}

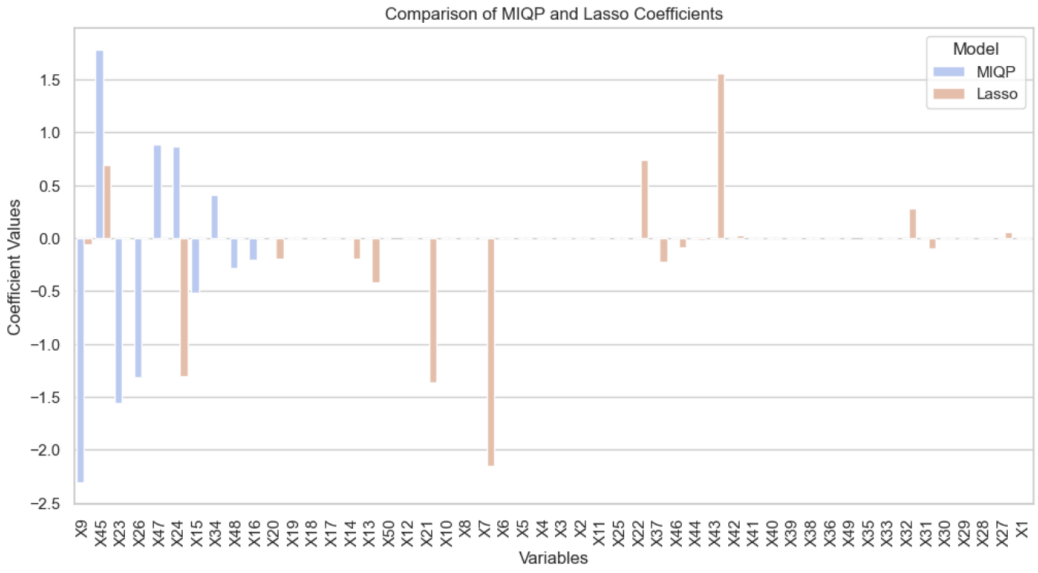


The comparison of selected variables between the MIQP and LASSO models reveals both consensus and divergence in their variable preferences. MIQP strategically uses binary variables and the big-M method for direct control over variable inclusion/exclusion. In contrast, LASSO employs an L1 penalty, inducing sparsity and automatic variable selection. Commonly identified variables (26, 45, 23) indicate shared significance, underlining their robust impact on the outcome. Unique variables in the MIQP model (34, 9, 15, 16, 47, 48, 24) suggest it captures additional features or relationships not considered by LASSO, potentially offering a more nuanced perspective. Conversely, LASSO's selection of distinct variables (6, 8, 12, 13, 19, 20, 21, 30, 31, 32, 36, 41, 42, 43, 44) showcases its sparsity-inducing nature. The difference in the number of selected variables underscores the inherent trade-offs between model simplicity and complexity. MIQP's more concise model may be preferred when prioritizing a streamlined representation, whereas LASSO's inclusion of a larger set of variables may capture a wider range of potential contributors.

The graph below shows the actual value alongside both the MIQP and LASSO predictions.



In the plot below, we can see that MIQP is zeroing out most of the variables when compared to LASSO. MIQP is the preferred choice when precise subset selection is critical as it excels at exactly zeroing out unnecessary variables, making it ideal for situations where you have specific constraints or requirements on the inclusion or exclusion of certain features. But when subset selection is not that critical then LASSO is the best choice as it offers the advantage of computational efficiency.



V. Conclusion

Our goal is to find out if our boss should switch from LASSO to a more direct variable selection like MIQP.

To help make this decision, first we ran an MIQP model where we did k fold cross validation and found an M value of 25, which we used. We split the data into a training and data set. After finding the best k values and error, the error ended up being 2.336 and a k value of 10.

Next, we ran a LASSO regression on the data. We again split the data into a training and test data set. We also conducted 10 fold cross validation. After testing different values we got the best alpha value of .079 and best error of 2.346.

Advantages and Disadvantages: LASSO

Advantages	Disadvantages
• Efficiency: LASSO is more computationally efficient than MIQP, ideal for larger datasets. • Prevents Overfitting: Regularization in LASSO reduces model complexity, mitigating overfitting risks.	• Increased Bias: Simplification of the model can introduce bias, potentially reducing prediction accuracy. • Hyperparameter Tuning: Requires optimal hyperparameter (λ) determination, necessitating cross-validation steps.

Advantages and Disadvantages: MIQP

Advantages	Disadvantages
● Direct Feature Selection: Closer alignment with the true model due to controlled variable inclusion. ● Model Accuracy: Higher accuracy and lower error rates compared to LASSO.	● Computational Demand: Significantly higher computational resources and time requirements, especially with large datasets. ● Impracticality for Large Data: Challenges in scaling and handling vast amounts of data efficiently. ● Error Rates: MIQP's best test error (2.336) is slightly lower than LASSO's (2.346), indicating higher accuracy.

VI. Recommendations

If we have not gathered all the variables for the entire population targeted for our analysis, obtaining these variables is costly, it's prudent to perform the analysis on a sample from our population. We can then identify the minimum number of variables needed to minimize MSE. Afterward, for the remaining data points, we can selectively acquire only these chosen variables using the direct method, effectively reducing variable acquisition costs.

On the other hand, if we have already obtained all the variables for the entire population intended for our analysis, employing the Lasso method is recommended. The Lasso method provides rapid and interpretable results with approximately the same error. Additionally, since the direct method's computation time is relatively higher, it could become significantly more time-consuming when dealing with a large number of variables for optimal selection (k variables). Nevertheless, despite the slightly better performance of the direct method compared to the indirect method, the Lasso method offers efficiency and interpretability.

VII. Next Steps

- Further Testing: Conduct trials using diverse datasets of varying sizes to more comprehensively evaluate MIQP and LASSO.
- We could select a few other projects that are currently using LASSO and test MIQP on them as well.
- Trying these techniques more is very important to see what the true value of each model is as there can be a lot of variation.