MIQP vs. LASSO: Optimal Variable Selection Strategies

Evaluating Efficiency and Accuracy in Predictive Modeling

GROUP 7
Sian Cheng (sc65827)
Monica Liou (cl49358)
Akshay Navaneetha Krishnan (an34244)
Morgan Tucker (mt36459)

Date: Nov 17, 2023

Table of Content

Introduction

This report examines two prominent methods for variable selection in regression analysis: Direct Variable Selection through Mixed Integer Quadratic Programming (MIQP) and Indirect Variable Selection via LASSO (Least Absolute Shrinkage and Selection Operator). It aims to assess and compare these approaches in the context of recent advancements in computational optimization. While traditional regression methods often grapple with the trade-offs between complexity and predictive accuracy, the advent of powerful computational tools has reinvigorated interest in more direct variable selection techniques like MIQP. This analysis seeks to provide insights into the practicality and effectiveness of these methods, helping to guide their application in modern predictive analytics.

Problem Overview

The problem overview for this report centers on evaluating and comparing two variable selection methods for regression analysis: the computationally advanced MIQP and the established LASSO technique. This report aims to determine if the recent strides in optimization software make MIQP a superior choice for variable selection in predictive analytics, challenging the conventional preference for LASSO.

Direct Variable Selection – MIQP Problem

Mathematical Formulation of the MIQP Problem

Mathematically, it's expressed as: minimize the sum of squared differences between observed and predicted values, subject to constraints that govern the inclusion or exclusion of each variable.

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

Role of Binary Variables and the Big-M Method

The core of MIQP approach lies in its use of binary variables, denoted as z_j , which determine whether a variable x_j is included in the model (if z_j =1) or excluded (if z_j =0). Big-M method introduces constraints that effectively "turn off" a variable by forcing its coefficient to zero when its corresponding binary variable is zero.

$$\min_{\beta,z} \sum_{i=1}^{n} (\beta_0 + \beta_1 x_{i1} + \cdots + \beta_m x_{im} - y_i)^2$$
$$s.t. -Mz_j \le \beta_j \le Mz_j \quad for\ j = 1, 2, 3, \ldots, m$$
$$\sum_{j=1}^{m} z_j \le k$$
$$z_j \ \ are\ binary.$$

Indirect Variable Selection – LASSO

LASSO Regression Formulation

LASSO introduces a penalty term proportional to the absolute value of the coefficients. Mathematically, the LASSO regression is formulated as minimizing the sum of squared errors with an added constraint: the sum of the absolute values of the coefficients should be less than a fixed value. This constraint leads to some coefficients being exactly zero, effectively performing variable selection by excluding them from the model.

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

Role of Hyperparameter in LASSO

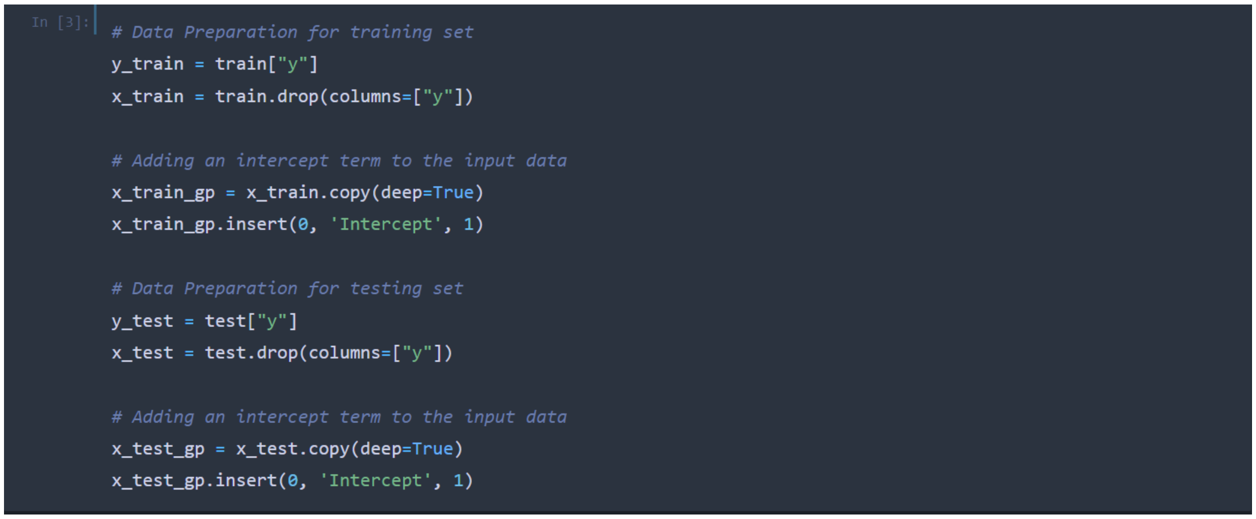
The hyperparameter in LASSO, denoted as λ, controls the strength of the penalty applied to the coefficients. As λ increases, more coefficients are set to zero, leading to simpler models. The choice of λ is crucial: too small a value leads to overfitting (similar to ordinary least squares), while too large a value oversimplifies the model, omitting significant predictors. Selecting the optimal λ is usually achieved through cross-validation, where the model is tested on various subsets of the data to find the λ that minimizes prediction error, striking a balance between model complexity and predictive accuracy.

Methodology

Description of Data Sets (Training and Test)

There were two datasets used for the regression analysis, one being the training set and the other being the test set. The training data contains 150 observations with the first column being the y variable and columns 2 through 51 being x variables. Conversely, the test data contains 50 observations with the first column being the y variable and columns 2 through 51 being x variables.

The dataset has been partitioned into two components: the input data (x data) and the output/target data (y data). **Additionally, a supplementary column 'intercept' containing a constant value of 1 has been appended to accommodate the intercept term during the Mixed-Integer Quadratic Programming (MIQP) optimization process. The corresponding code snippet is attached below.**

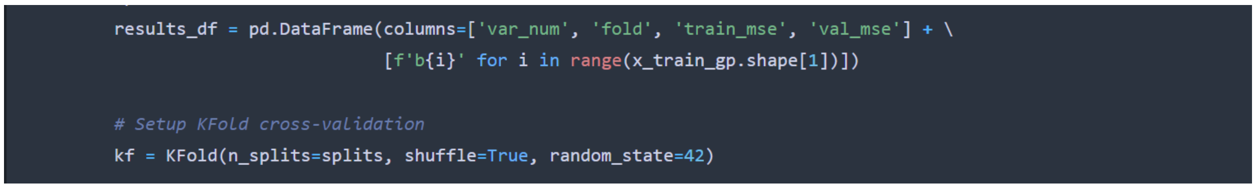


```
In [3]:
# Data Preparation for training set
y_train = train["y"]
x_train = train.drop(columns=["y"])

# Adding an intercept term to the input data
x_train_gp = x_train.copy(deep=True)
x_train_gp.insert(0, 'Intercept', 1)

# Data Preparation for testing set
y_test = test["y"]
x_test = test.drop(columns=["y"])

# Adding an intercept term to the input data
x_test_gp = x_test.copy(deep=True)
x_test_gp.insert(0, 'Intercept', 1)
```

Feature selection through Gurobi (10 fold CV)

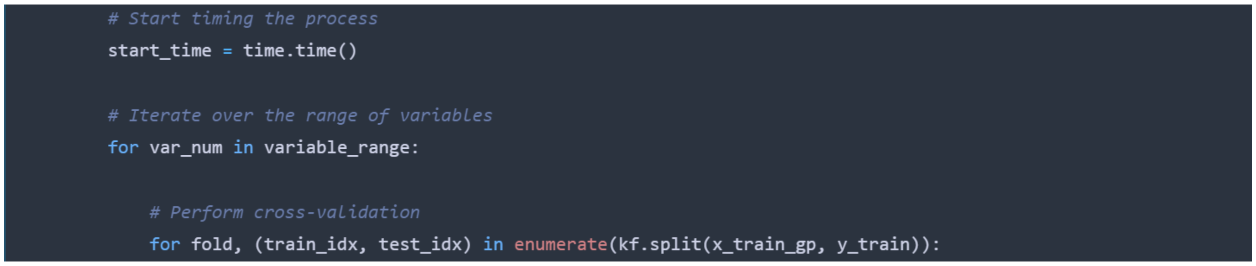
To evaluate the predictive performance of our regression model, we implemented a comprehensive 10-fold cross-validation process. This involved partitioning the dataset into ten unique subsets, thereby allowing us to utilize each subset as a validation set while the remaining subsets constituted the training set for each iteration of the process.

We began by creating a pandas DataFrame to record cross-validation outcomes, capturing the feature count (k), fold number, and MSE for training and validation sets. To randomize the dataset into 10 different folds for cross validation, we utilized the KFold package from sklearn's model_selection library. **The code snippet for initializing the results_df and the KFold object is below.**



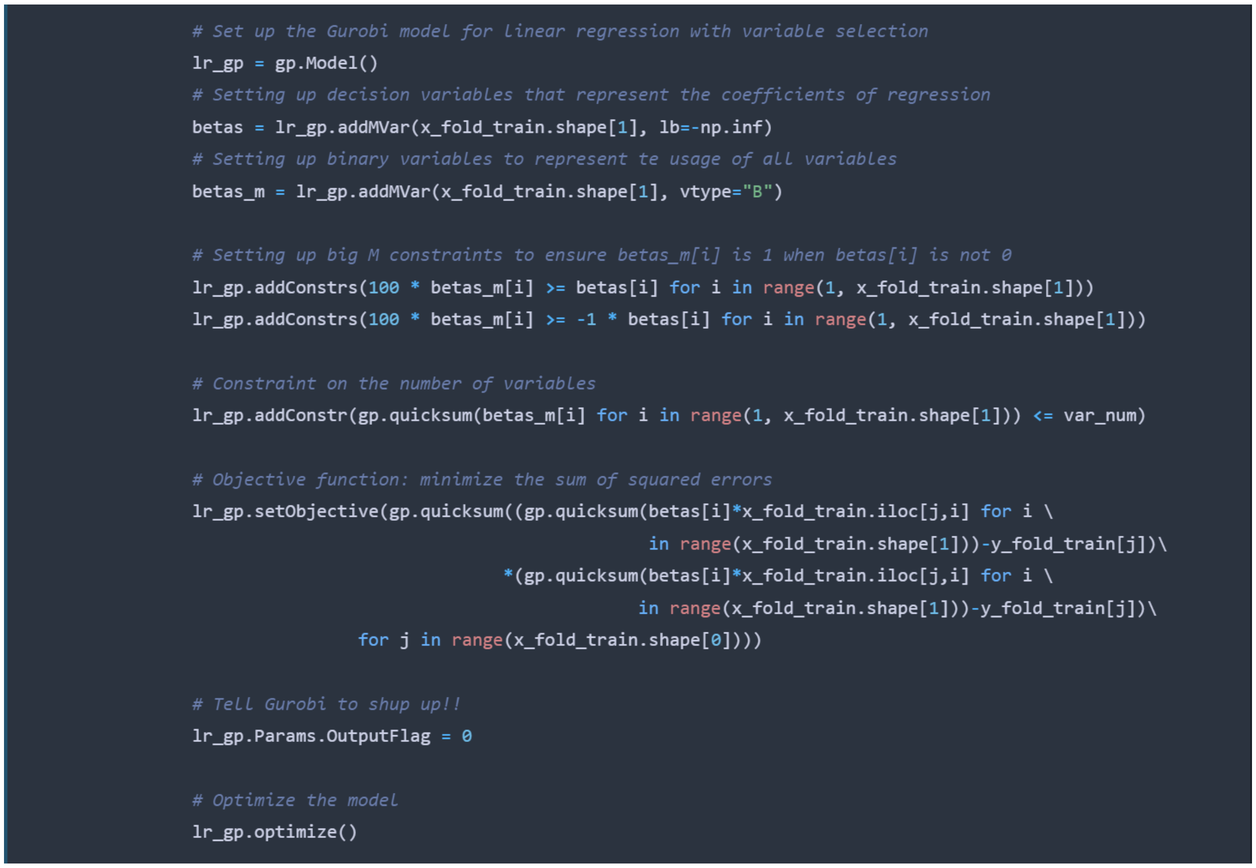
```
results_df = pd.DataFrame(columns=['var_num', 'fold', 'train_mse', 'val_mse'] + \
                          [f'b{i}' for i in range(x_train_gp.shape[1])])

# Setup KFold cross-validation
kf = KFold(n_splits=splits, shuffle=True, random_state=42)
```

The cross-validation aimed not only to validate the model's performance but also to identify the optimal number of features (k) to be included in the model. We explored a range of k values, incrementing from 5 to 50 in steps of 5. Hence, we iterate over 10 values of k and 10 folds in this specific problem to solve a total of 100 MIQP problems. **The code below shows the loops formed to solve all 100 problems effectively.**



```
# Start timing the process
start_time = time.time()

# Iterate over the range of variables
for var_num in variable_range:

    # Perform cross-validation
    for fold, (train_idx, test_idx) in enumerate(kf.split(x_train_gp, y_train)):
```

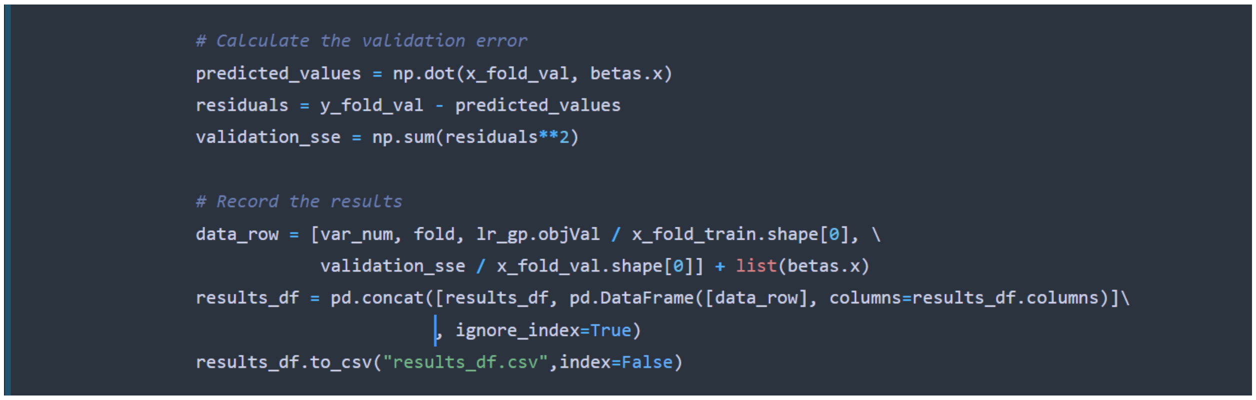
For each k value, we constructed a mixed-integer quadratic programming (MIQP) model using Gurobi optimization software. This model included decision variables for regression coefficients, unrestricted in sign, and binary variables to indicate the inclusion of each feature. We employed the big-M method to constrain the feature selection, with a large constant M effectively allowing or disallowing the contribution of each feature based on its binary indicator. The model's objective function was the sum of squared residuals, which we minimized to fit the regression model. **The code snippet for the formulation of the Gurobi problem is attached below.**



```
# Set up the Gurobi model for linear regression with variable selection
lr_gp = gp.Model()
# Setting up decision variables that represent the coefficients of regression
betas = lr_gp.addMVar(x_fold_train.shape[1], lb=-np.inf)
# Setting up binary variables to represent te usage of all variables
betas_m = lr_gp.addMVar(x_fold_train.shape[1], vtype="B")

# Setting up big M constraints to ensure betas_m[i] is 1 when betas[i] is not 0
lr_gp.addConstrs(100 * betas_m[i] >= betas[i] for i in range(1, x_fold_train.shape[1]))
lr_gp.addConstrs(100 * betas_m[i] >= -1 * betas[i] for i in range(1, x_fold_train.shape[1]))

# Constraint on the number of variables
lr_gp.addConstr(gp.quicksum(betas_m[i] for i in range(1, x_fold_train.shape[1])) <= var_num)

# Objective function: minimize the sum of squared errors
lr_gp.setObjective(gp.quicksum((gp.quicksum(betas[i]*x_fold_train.iloc[j,i] for i \
                                          in range(x_fold_train.shape[1]))-y_fold_train[j])\
                              *(gp.quicksum(betas[i]*x_fold_train.iloc[j,i] for i \
                                         in range(x_fold_train.shape[1]))-y_fold_train[j])\
                for j in range(x_fold_train.shape[0])))

# Tell Gurobi to shup up!!
lr_gp.Params.OutputFlag = 0

# Optimize the model
lr_gp.optimize()
```

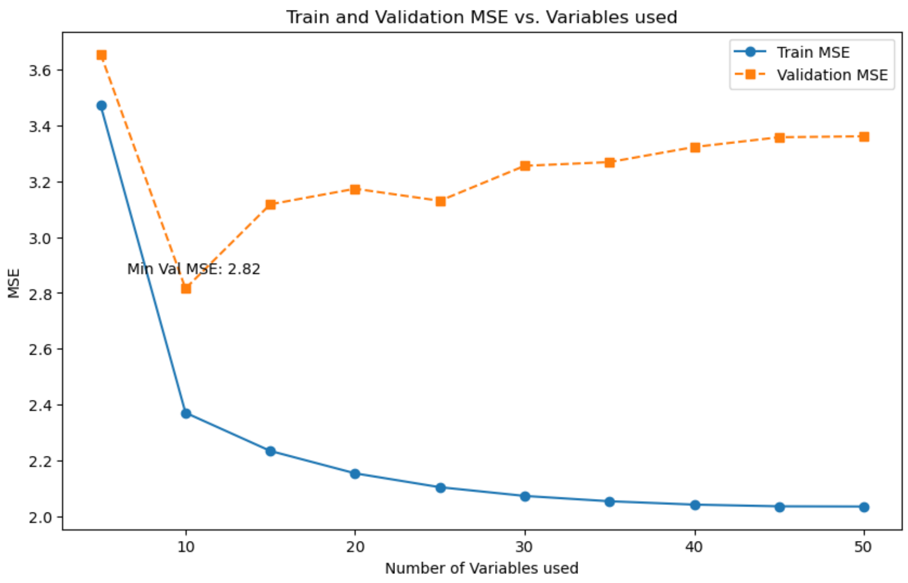
Upon optimizing the MIQP model for each fold and each k value, we computed the MSE for the respective training and validation sets. These metrics were recorded in the initialized DataFrame. After evaluating all folds under a specific k value, we calculated the average validation MSE to assess the generalizability of the model. This iterative process enabled us to ascertain the k value that yielded the lowest average validation MSE, signifying the optimal number of features for the model.

Ultimately, the cross-validation process culminated in identifying the k value that optimized the balance between model complexity and predictive accuracy. The best k value, along with its associated MSE, was reported for interpretability and transparency. Additionally, the detailed results of the cross-validation were preserved in a CSV file, facilitating subsequent analysis and verification of the model's performance. **The code attached below shows the evaluation of cross-validation error using the train coefficients as well the saving of results to a csv.**



```
# Calculate the validation error
predicted_values = np.dot(x_fold_val, betas.x)
residuals = y_fold_val - predicted_values
validation_sse = np.sum(residuals**2)

# Record the results
data_row = [var_num, fold, lr_gp.objVal / x_fold_train.shape[0], \
            validation_sse / x_fold_val.shape[0]] + list(betas.x)
results_df = pd.concat([results_df, pd.DataFrame([data_row], columns=results_df.columns)]\
                       , ignore_index=True)
results_df.to_csv("results_df.csv",index=False)
```

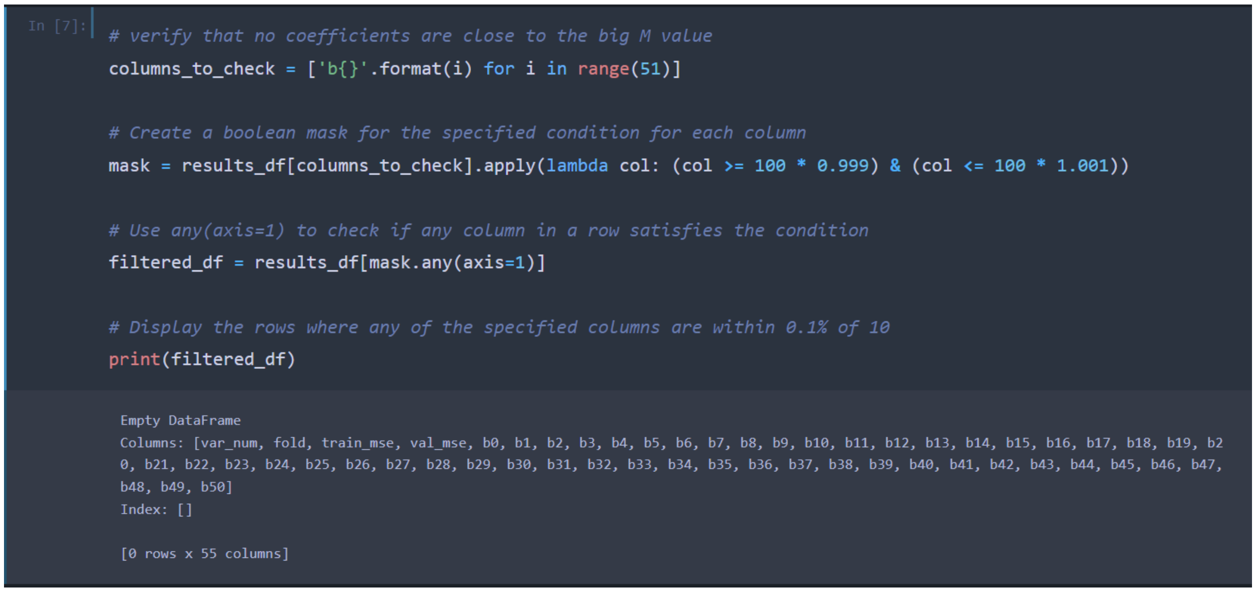
The included graph from our 10-fold cross-validation shows training and validation MSEs against the number of model features. Initially, the validation MSE declines sharply with more features, indicating better model performance. However, at k = 10, the validation MSE reaches its lowest point before plateauing, suggesting that further features do not enhance and might hinder the model's generalization. The optimal point on the graph signifies the most efficient feature set size, striking a balance between model complexity and predictive accuracy without overfitting. This visual serves as a guide for choosing the ideal number of features (k) and illustrates the model's performance at varying complexities. The training MSE consistently sits below the validation MSE, as anticipated, with increasing divergence indicating the onset of overfitting as more features are added.



Selection and verification of big M value

To determine an appropriate value for the big M constraint in our optimization process with k=5, we conducted a systematic exploration of big M values, commencing from 10E6 and iteratively reducing it. Our aim was to identify the threshold at which the big M constraints effectively limited the number of non-zero variables to 5. Upon observation, when the big M value was excessively high, more than 5 variables were permitted to be non-zero.

Upon experimentation, we settled on a big M value of 100. Following the execution of 100 Mixed-Integer Quadratic Programming (MIQP) problems with this chosen big M value, we scrutinized the coefficients. Specifically, we investigated whether any coefficients approached 100 within a tolerance of 0.1%. Notably, our analysis revealed that none of the coefficients were in proximity to this value. Consequently, we deemed our selection of the big M constraint as valid for our optimization process. **Below is the code snippet for the verification.**



```
In [7]:
# verify that no coefficients are close to the big M value
columns_to_check = ['b{}'.format(i) for i in range(51)]

# Create a boolean mask for the specified condition for each column
mask = results_df[columns_to_check].apply(lambda col: (col >= 100 * 0.999) & (col <= 100 * 1.001))

# Use any(axis=1) to check if any column in a row satisfies the condition
filtered_df = results_df[mask.any(axis=1)]

# Display the rows where any of the specified columns are within 0.1% of 10
print(filtered_df)
```

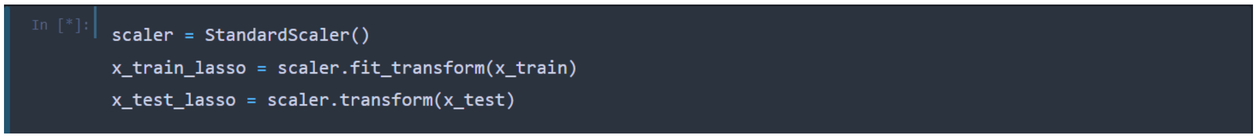
```
Empty DataFrame
Columns: [var_num, fold, train_mse, val_mse, b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16, b17, b18, b19, b2
0, b21, b22, b23, b24, b25, b26, b27, b28, b29, b30, b31, b32, b33, b34, b35, b36, b37, b38, b39, b40, b41, b42, b43, b44, b45, b46, b47,
b48, b49, b50]
Index: []

[0 rows x 55 columns]
```

MIQP Retraining

Upon determining the optimal value for the parameter k, it is imperative to retrain the linear model using Mixed-Integer Quadratic Programming (MIQP) optimization on the complete dataset. Initially, the model was trained on 90% of the data during the 10-fold cross-validation process. The resulting set of coefficients will subsequently be utilized to assess the model's performance on the test set. The code utilized is very similar to the Gurobi code formulated above, except this time we utilize the entire dataset and fix the value of k to be 10.

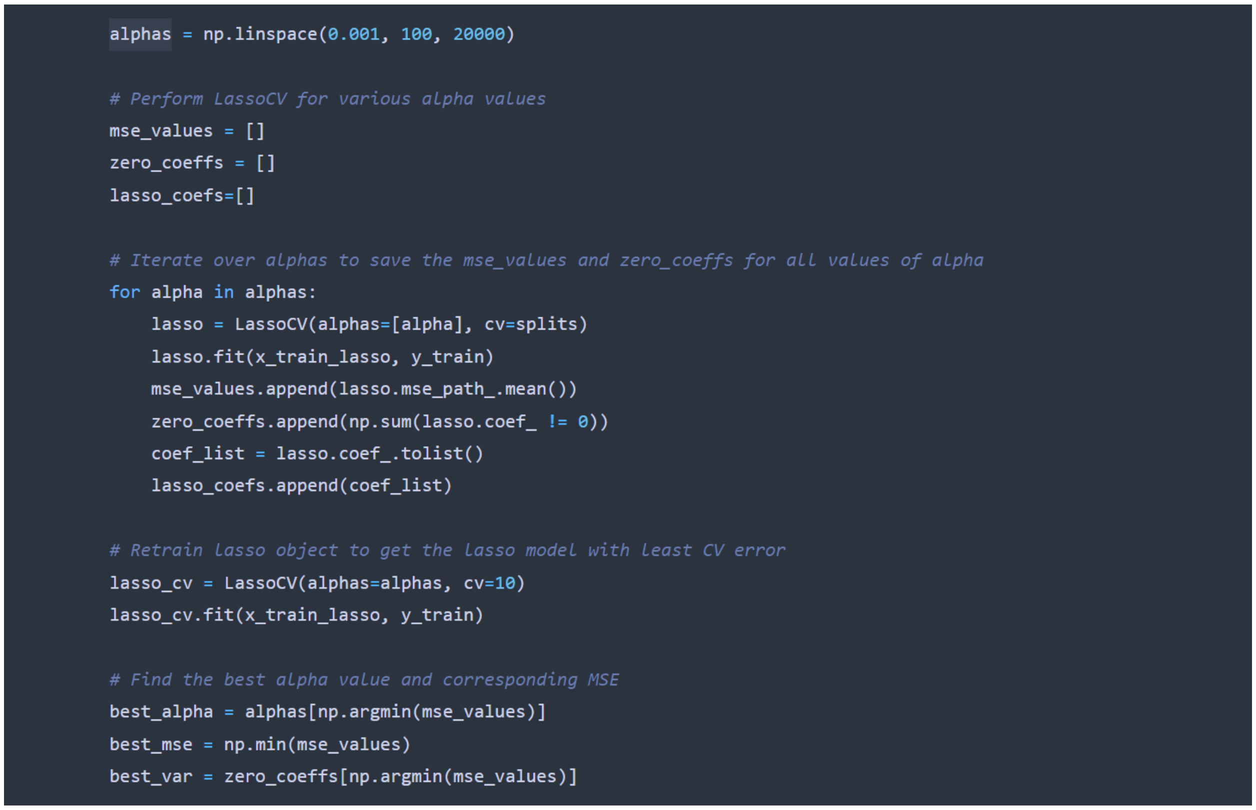
Feature selection through Lasso (10 fold CV)

In our performance evaluation of the Gurobi-selected model, we employed lasso regression as a benchmark, leveraging its inherent feature selection capabilities through the adjustment of the alpha regularization parameter. To facilitate this analysis, we initially standardized the x_train data using standard scaling, ensuring a mean of 0 and a standard deviation of 1—a prerequisite for the effective application of lasso regression on the data. **The code for scaling the data is attached below.**



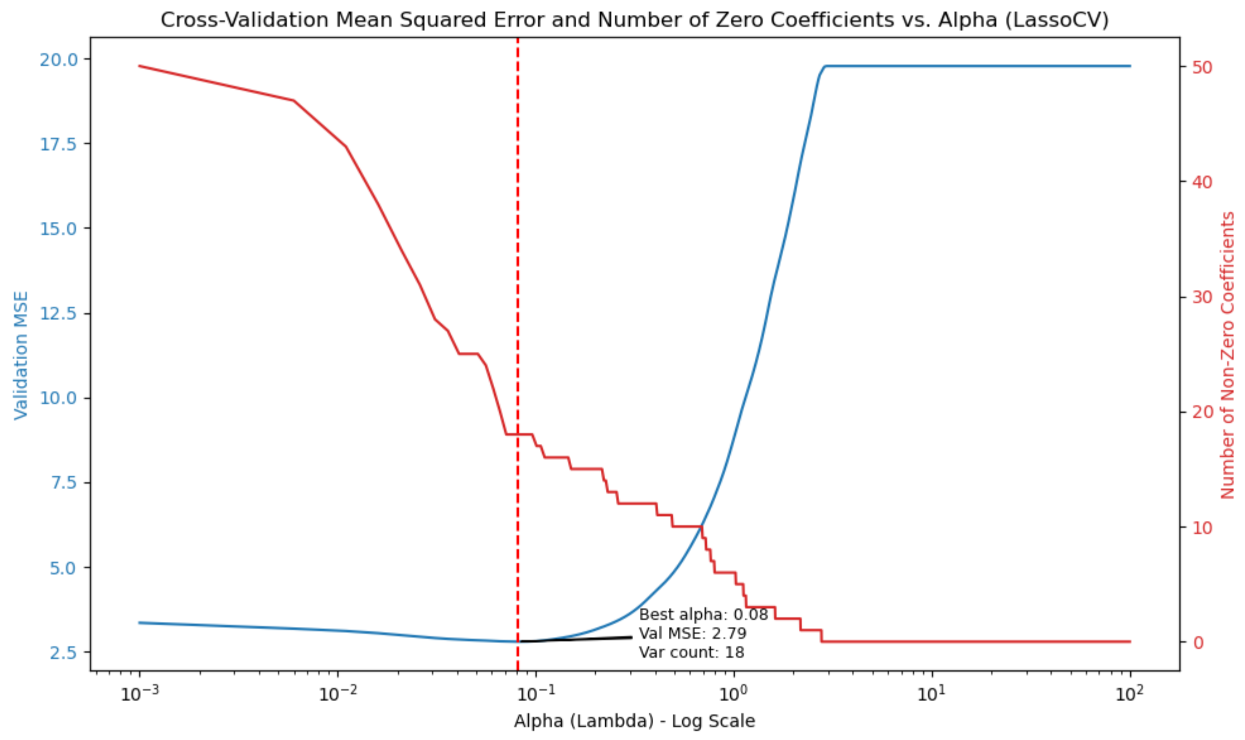
```
In [*]:  scaler = StandardScaler()
         x_train_lasso = scaler.fit_transform(x_train)
         x_test_lasso = scaler.transform(x_test)
```

To systematically explore the impact of varying lasso parameters, we conducted an extensive examination across a range from 0.001 to 100, encompassing a total of 20,000 parameter configurations using the numpy linspace function. Subsequently, to find the optimal alpha value that aligns with the lowest cross-validation error, we employed the Lassocv method from the sklearn linear_model library. This process involved executing 10-fold cross-validation on the training data, providing an assessment of model performance and guiding us in selecting the most suitable alpha parameter for our lasso regression benchmark. It was noted that, for the alpha value corresponding to the lowest cross-validation error, the resultant lasso regression model incorporated 18 non-zero variables. **The code below shows the lasso model implementation.**

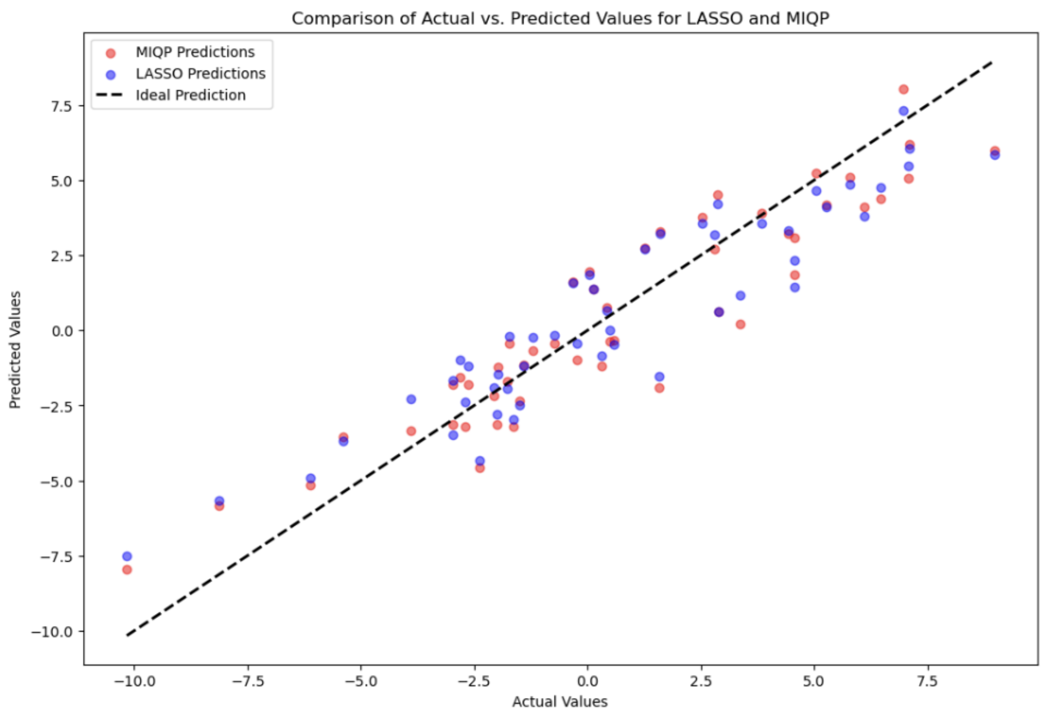


```python
alphas = np.linspace(0.001, 100, 20000)

# Perform LassoCV for various alpha values
mse_values = []
zero_coeffs = []
lasso_coefs=[]

# Iterate over alphas to save the mse_values and zero_coeffs for all values of alpha
for alpha in alphas:
    lasso = LassoCV(alphas=[alpha], cv=splits)
    lasso.fit(x_train_lasso, y_train)
    mse_values.append(lasso.mse_path_.mean())
    zero_coeffs.append(np.sum(lasso.coef_ != 0))
    coef_list = lasso.coef_.tolist()
    lasso_coefs.append(coef_list)

# Retrain lasso object to get the lasso model with least CV error
lasso_cv = LassoCV(alphas=alphas, cv=10)
lasso_cv.fit(x_train_lasso, y_train)

# Find the best alpha value and corresponding MSE
best_alpha = alphas[np.argmin(mse_values)]
best_mse = np.min(mse_values)
best_var = zero_coeffs[np.argmin(mse_values)]
```

Attached below is the effect of changing alpha on the validation MSE and the number of non zero coefficients in the lasso model (as the value of alpha increases, more features get driven to 0).



Results: MIQP vs. LASSO



Criteria	MIQP	LASSO
Time(s)	3500	10.8
Test MSE	2.33	2.36
Val MSE	2.82	2.79

Pros and Cons (MIQP vs Lasso)

Aspect	MIQP	LASSO
Variable Selection	Allows easy selection of the number of variables needed	Primarily a variance reduction method, also aids in variable selection by adjusting lambda. However, the best lambda with CV error might not yield a suitable number of selected variables.
Computational Time	Takes a significant amount of time to run	Runs fast, especially with coordinate descent
Accuracy	Provides slightly better accuracy with fewer variables	Gives slightly worse accuracy

Recommendations

The application of mixed-integer quadratic programming (MIQP) for variable selection has shown a tendency to streamline the model by eliminating a greater number of variables, as evidenced by its reduction of coefficients to zero, in contrast to the LASSO technique. While the optimal LASSO model retained 18 non-zero variable coefficients, MIQP refined this to just 10 without sacrificing, and indeed enhancing, performance on the test dataset.

Nevertheless, it's crucial to consider the time required by each method. MIQP demonstrated precision in variable selection over approximately 2.5 hours, an attribute that becomes particularly beneficial when parsing datasets with numerous variables and the aim is to enhance model interpretability by condensing the variable set. In stark contrast, LASSO's computation time was markedly less, at around 10 seconds, although it does introduce some uncertainty in the exact number of variables it will select.

Taking a general view, for this specific dataset, MIQP's marginal benefits are overshadowed by the efficiency of LASSO, particularly when employing the lassocv package that leverages coordinate descent, yielding results in a mere 10 seconds. It's appropriate to outline the strengths and weaknesses of each method before drawing a conclusion. In this instance, LASSO appears to be the more pragmatic choice, offering rapid results and adequate model simplification. However, this comparison is based on a single dataset, and broader assertions about the overall efficacy of MIQP versus LASSO would require a more extensive comparative analysis across diverse datasets.

Therefore, our conclusive advice is to consider the MIQP method exclusively if there is a critical imperative to minimize the number of variables extensively, and there is an ample amount of time available. The increase in accuracy achieved through MIQP is only marginal and negligible. Additionally, the runtime for MIQP in this case is approximately 2 hours, in stark contrast to the mere 10 seconds required for Lasso. Given the modest enhancements in accuracy and the substantial time difference, it appears that the benefits of MIQP may not be justifiable in this context. As a result, we recommend opting for Lasso for its efficiency in achieving comparable results with a significantly shorter runtime.

Nevertheless, we strongly encourage the firm to conduct further tests on additional datasets. This step is crucial for bolstering our confidence in the obtained results and ensuring the robustness and generalizability of the findings.