OPTIMIZATION

PROJECT - 3

NON – LINEAR PROGRAMMING

Feature Selection: MIQP vs LASSO

2023-11-17

GROUP - IV

Manideep Telukuntla (mt39528)

Vaishnavi Ganesh (vg23798)

Sanjana Nayak (svn378)

Bin Yang (by4325)

Table of Contents

Introduction

Selecting the most influential variables is a fundamental challenge in developing accurate and interpretable predictive models. While penalized regression techniques like Lasso and Ridge regression have been widely used, they suffer from drawbacks including bias introduced by the penalty parameter. Meanwhile, optimization methods for direct variable selection have been long dismissed as computationally intractable. However, recent advances in mixed integer quadratic programming (MIQP) have made direct variable selection viable. This project marks an important milestone in realizing the full potential of optimization for predictive analytics.

By leveraging MIQP to encode variable selection decisions as binary variables, we can now optimize and directly identify the variables with maximum predictive power. The merits of this approach are multifold. First, it avoids biases and distortions created by penalty methods. Second, it provides inherent interpretability by extracting the optimal variable subset. Third, it allows incorporating predictive performance objectives directly into the optimization model. A key focus of this project is posing the variable selection problem for regression as an MIQP and solving it using Gurobi. This allows us to directly find the optimal subset of variables to include in the regression model.

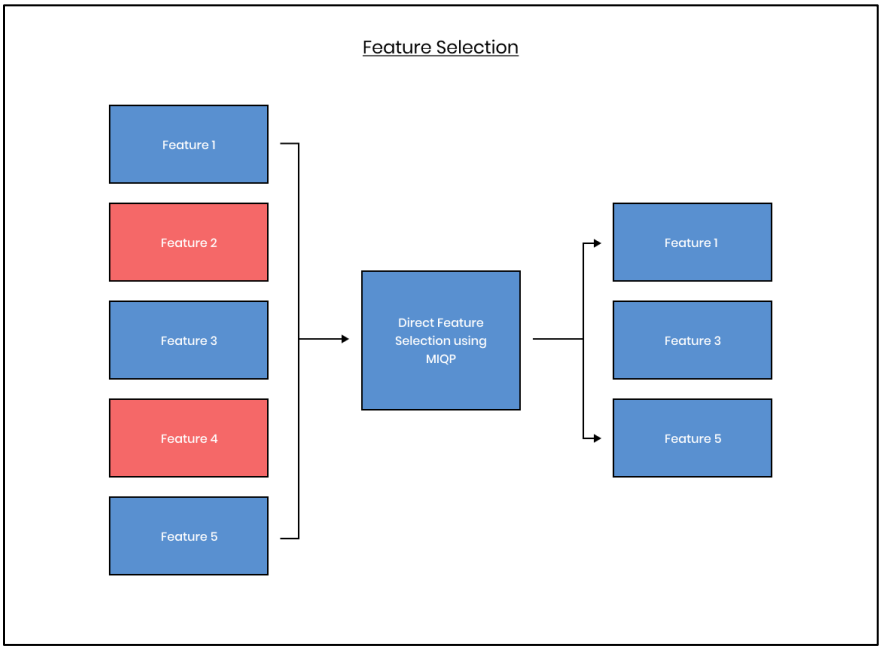


Figure 1: Feature Selection using MIQP

We compare the MIQP model results against the LASSO regression technique. Lasso regression, or L1 regularization, is a linear regression technique that incorporates a penalty term based on the absolute values of the regression coefficients. The primary goal of lasso regression is to encourage sparsity in the model by pushing some of the coefficients to exactly zero.

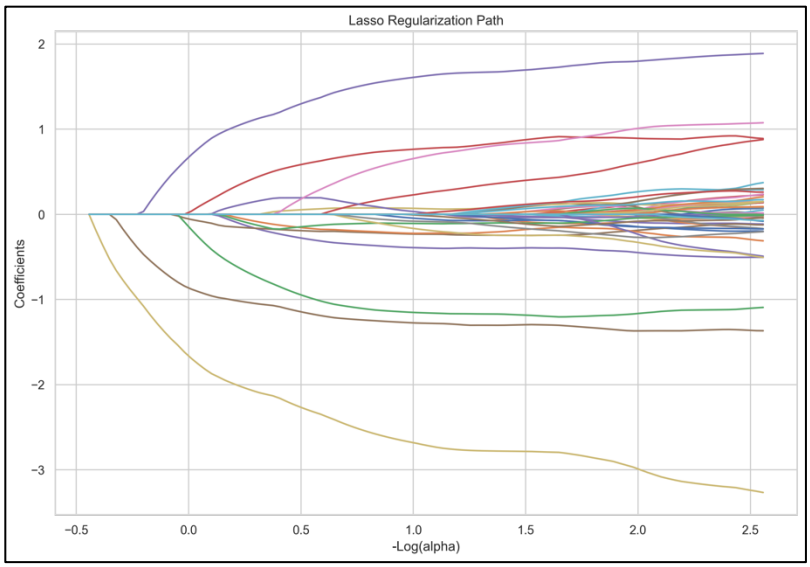


Figure 2: Feature Selection using Lasso

By comparing these two methodologies, this project thus makes a significant contribution by unlocking the full potential of optimization for predictive analytics. MIQP-based direct variable selection promises to be an indispensable tool for developing the next generation of accurate, transparent, and trustworthy predictive models. Our experiments reveal cases where the MIQP model selects a better performing variable subset than LASSO. This suggests the additional regularization and shrinkage imposed by LASSO may not always be beneficial compared to direct optimization-based variable selection.

Methodology

Implementation of the MIQP Model

This report talks about the usage of two methods to solve the variable selection problem, the MIQP method that we're trying to bring into the picture and Lasso regression to compare and contrast the MIQP method to see where it stands and the approach, we followed to implement these techniques. To do that, we first calculated the β values using optimization methods and compared that as well with the linear regression coefficients that we got after running the LinearRegression function using scikit-learn. The below code gets the training data and minimizes the objective function (sum-product of the β coefficients and the X values):

```
num_variables = len(train_data.columns)
num_records = len(train_data)

lm = gp.Model()
beta = lm.addMVar(num_variables,lb=-np.inf)

lm = gp.Model()
beta = lm.addMVar(num_variables,lb=-np.inf)
lm.setObjective(
    gp.quicksum(
    (beta[0] + gp.quicksum(beta[j] * train_data[f"X{j}"][i]
        for j in range(1, num_variables)) - train_data["y"][i]) *
    (beta[0] + gp.quicksum(beta[j] * train_data[f"X{j}"][i]
        for j in range(1, num_variables)) - train_data["y"][i])
    for i in range(num_records)
    )
)

lm.Params.OutputFlag = 0 # tell gurobi to shut up!!
lm.optimize()
```

Code Block 1: Linear Regression formulated as quadratic programming using Gurobi in python

The output of the β values got from the optimization model coded above are as follows:

```
array([ 0.96181997, -0.03124313,  0.10830046,  0.33658635,  0.34788706, -0.07805158, -0.26870205,
        0.56917466, -0.09265014, -2.80170764, 0.5961224, 0.00342708, -0.40802441, 0.64028772,
        0.34250235, -0.35811603, -0.22662545, 0.47574886, 0.61818724,  0.5627165, 0.91343131,
        0.1266948, -0.44072877, -1.13218924, 0.71016955, 0.08281624, -1.48717747, 0.02439235,
       -0.38572888, 0.35864434, 0.08111228, -0.18165492, 0.74866389, -0.04257738, 1.2137678,
       -0.53326153, 0.23934142,  0.25972978, -0.13112438, -1.09826371, 0.45402031, -0.59555224,
        0.33959881, -0.4208102 ,  0.04031396, 1.8661944 , -0.01828012,  1.07420755, 0.147014,
        0.56919769, -0.22109118])
```

Code Block 2: Beta values for Linear Regression formulated as QP using Gurobi in python

The below code runs a linear regression using scikit-learn package to get the coefficients for the same training data so that we can compare it with the above output

```python
from sklearn.linear_model import LinearRegression
import pandas as pd

X = train_data.drop(columns=["y"])  # Features
y = train_data["y"]  # Target variable

# Initialize the Linear Regression model
model = LinearRegression()

# Fit the model to the data
model.fit(X, y)

# Get the coefficients (model parameters)
coefficients = model.coef_
intercept = model.intercept_
```

Code Block 3: Linear Regression implemented using sci-kit learn in python

The output coefficients we get from the above code are as follows:

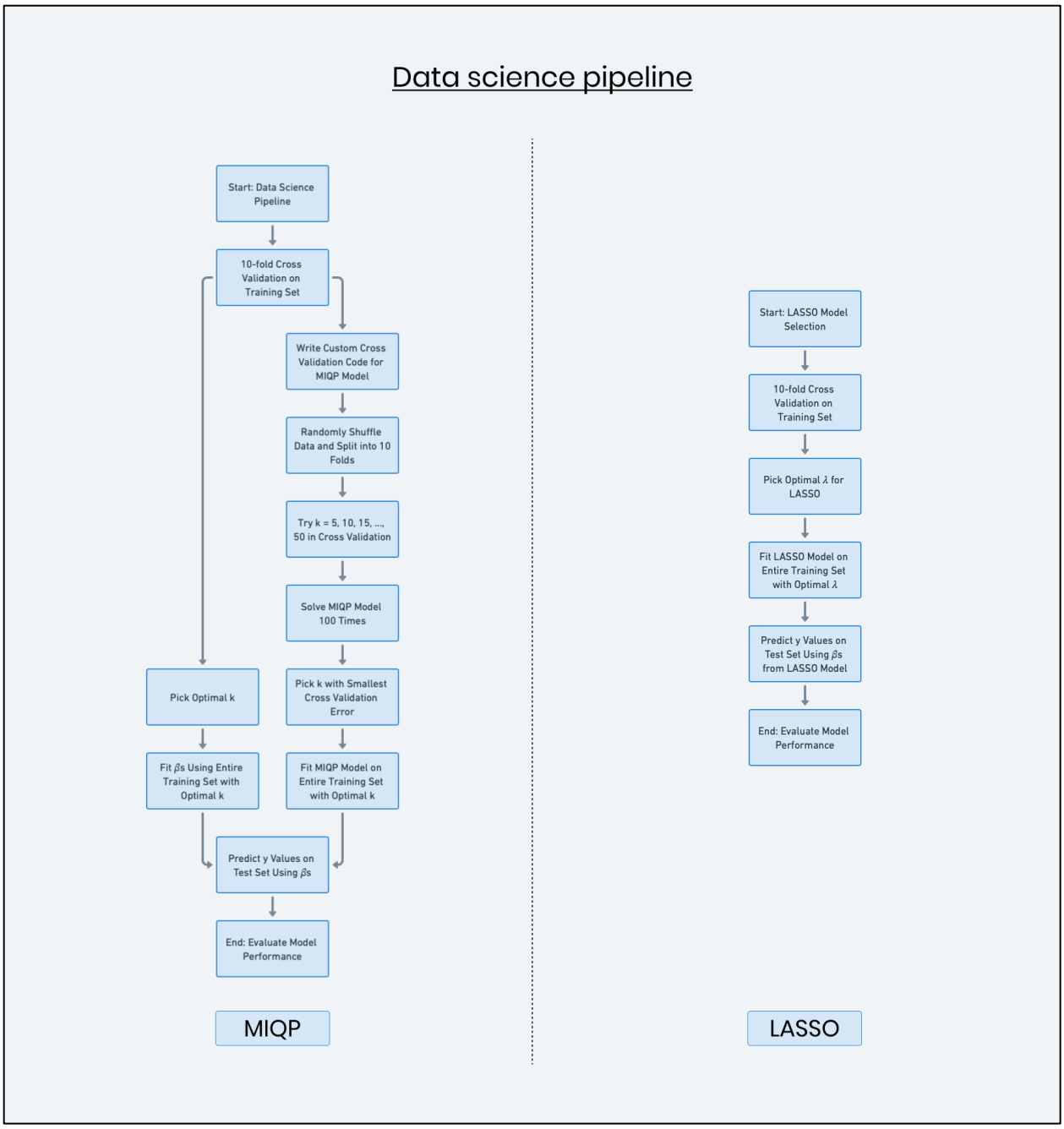
```
Coefficients: [-0.03124313  0.10830046  0.33658635  0.34788706 -0.07805158 -0.26870205
                0.56917466 -0.09265014 -2.80170764  0.5961224   0.00342708 -0.40802441
                0.64028772  0.34250235 -0.35811603 -0.22662545  0.47574886  0.61818724
                0.5627165   0.91343131  0.1266948  -0.44072877 -1.13218924  0.71016955
                0.08281624 -1.48717747  0.02439235 -0.38572888  0.35864434  0.08111228
               -0.18165492  0.74866389 -0.04257738  1.2137678  -0.53326153  0.23934142
                0.25972978 -0.13112438 -1.09826371  0.45402031 -0.59555224  0.33959881
               -0.4208102   0.04031396  1.8661944  -0.01828012  1.07420755  0.147014
                0.56919769 -0.22109118]

Intercept: 0.961819970676987
```

Code Block 4: Beta values for Linear Regression formulated using sci-kit learn in python

The comparison revealed that the coefficients and intercept obtained from both the MIQP model and the linear regression model were consistent with each other. This outcome validates our optimization approach, demonstrating that it performs on par with the standard LinearRegression function in scikit-learn. It reinforces the potential of the MIQP method as a viable and effective tool for variable selection in regression analysis.

Data Science Pipeline for MIQP and LASSO

The image below illustrates two machine learning pipelines employed for the analysis, each for implementing and evaluating a specific type of model. The first pipeline details the process for a Mixed-Integer Quadratic Programming (MIQP) model, involving steps such as 10-fold cross-validation, shuffling and splitting the data, hyperparameter tuning to find the optimal 'k', and finally, training the model with the entire dataset before making predictions and assessing its performance.



Flow Chart 1: Data science pipeline for MIQP and LASSO

The second pipeline is for a LASSO (Least Absolute Shrinkage and Selection Operator) model, which also starts with 10-fold cross-validation to select the best regularization parameter 'lambda'. After fitting the model to the training data with this chosen lambda, it proceeds to predict outcomes on a separate test set and concludes with a performance evaluation. Both pipelines employ standard machine learning practices for model selection, fitting, and validation.

Direct Variable Selection using MIQP

This method involves the consideration of different variables in the given dataset, assuming **m** independent variables and a dependent variable **y**. The standard ordinary least squares problem is formulated as follows:

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

```
# Set up the objective function
lm.setObjective(
   gp.quicksum(
       (beta[0] + gp.quicksum(beta[j] * train_set.iloc[i, j]
           for j in range(1, num_variables)) - train_set.iloc[i, 0]) *
       (beta[0] + gp.quicksum(beta[j] * train_set.iloc[i, j]
           for j in range(1, num_variables)) - train_set.iloc[i, 0])
       for i in range(num_records)
   )
)
```

Code Block 5: Defined objective function for MIQP using Gurobi in python

To incorporate variable selection into this objective function, we introduce binary variables "$\mathbf{z_j}$" that tell us whether the corresponding variable is going to be present in the model or not. If these variables do have a value, they will be restricted by the big M constraints that we introduce in the problem where we want to have utmost **k** variables in the problem, expressed as:

$$\min_{\beta,z} \sum_{i=1}^{n} (\beta_0 + \beta_1 x_{i1} + \cdots + \beta_m x_{im} - y_i)^2$$
$$s.t. \; -Mz_j \leq \beta_j \leq Mz_j \quad \text{for } j = 1,2,\ldots,m$$
$$\sum_{j=1}^{m} z_j \leq k$$
$$z_j \in \{0,1\}$$

```
# Set constraints for non-negativity and sparsity (sum of non-zero variables)
z = lm.addVars(num_variables, vtype=gp.GRB.BINARY)

lm.addConstrs((-M * z[j] <= beta[j] for j in range(1, num_variables)))
lm.addConstrs((beta[j] <= M * z[j] for j in range(1, num_variables)))

lm.addConstr(gp.quicksum(z[j] for j in range(1, num_variables)) <= k)
```

Code Block 6: Defined constraints for MIQP to incorporate direct variable selection using Gurobi in python

In this case we consider "**k**" as a hyperparameter to be chosen using cross-validation. Therefore, we use the objective function stated above with the big M constraints and the constraint that forces number of variables to be lesser than or equal to k to construct the MIQP solution. We used Gurobi to formulate the objective function and the constraints to arrive at the β values corresponding to the variables to be included in the model.

K-fold Cross Validation to Find optimal "k"

The python code to find the optimal "k" that minimizes error using 10-fold cross validation is implemented as shown below:

```
# Define the number of records and the number of folds for cross-validation
num_records = len(train_data)
num_folds = 10

# Define the range of k values to try
k_values = range(5, 51, 5)

# Initialize best_k and best_error
best_k = None
best_error = float('inf')
M = 100  # Initial value of M

np.random.seed(42)

# Randomly shuffle the indices of the dataset
indices = np.random.choice(num_records, num_records, replace=False)

# Create an empty DataFrame to store the results
num_beta_columns = len(train_data.columns)
results_df = pd.DataFrame(columns=['k', 'fold', 'validation_error', 'train_error', 'average_total_error']
    + [f'beta_{i}' for i in range(num_beta_columns)])
```

Code Block 7: Defined necessary parameters for k-fold cross validation & shuffled data

The first block of code sets up the parameters for a k-fold cross-validation process on a training dataset to determine the best hyperparameter 'k' for an MIQP model defined above. It defines the total number of records, sets the number of folds to 10, and establishes a range of k values to test (from 5 to 51 in increments of 5). The code initializes variables to keep track of the best k and the corresponding error, sets a big M to use in the MIQP model's constraints, seeds the random number generator for consistent shuffling of the dataset, and finally, it creates an empty DataFrame structured to store the results of the cross-validation, including placeholders for the error metrics and model coefficients.

```
for k in k_values:
    total_error = 0

    for fold in range(num_folds):

        # Split the indices into training and validation sets
        fold_size = num_records // num_folds
        validation_indices = indices[fold * fold_size: (fold + 1) * fold_size]
        train_indices = np.setdiff1d(indices, validation_indices)

        # Get data for the current fold based on the indices
        validation_set = train_data.iloc[validation_indices]
        train_set = train_data.iloc[train_indices]

        # Solve the MIQP model for the current k using the train_set
        start_time = time.time()  # Start the timer
        beta = solve_MIQP(train_set, k, M, time_limit)

        if time.time() - start_time > time_limit:
            print(f"Time limit exceeded for k = {k}. Moving to the next k value.")
            break  # Move to the next k value if the time limit is exceeded
        else:
            print(f"Time limit didn't exceeded for k = {k} and fold = {fold}. Moving to the next fold.")
```

Code Block 8: Logic for 10-fold cross validation to find optimal 'k'

In the second code block, the script enters a nested loop where it iterates over the range of k values and for each k, it further iterates over each fold of the cross-validation. It divides the dataset into training and validation sets according to the fold, runs the MIQP model to obtain coefficient estimates.

```
        if beta:
            # Calculate validation error using the validation set
            validation_predictions = validation_set.iloc[:, 1:].dot(beta[1:]) + beta[0]
            validation_error = mean_squared_error(validation_set["y"], validation_predictions)

            # Calculate train error using the training set
            train_predictions = train_set.iloc[:, 1:].dot(beta[1:]) + beta[0]
            train_error = mean_squared_error(train_set["y"], train_predictions)

            # Add data to the DataFrame
            results_df = results_df.append({
                'k': k,
                'fold': fold,
                'validation_error': validation_error,
                'train_error': train_error,
                'average_total_error': None,  # Placeholder for average_total_error
                **{f'beta_{i}': beta[i] for i in range(num_beta_columns)}  # Store beta coefficients
            }, ignore_index=True)

            total_error += validation_error

    # Calculate the average total error for all folds for the current k
    total_error = total_error / num_folds

    # Update the DataFrame with the average total error
    results_df.loc[results_df['k'] == k, 'average_total_error'] = total_error

    if total_error < best_error:
        best_error = total_error
        best_k = k
```

Code Block 9: Logic for computing train and validation errors for various 'k' values

The final code block deals with the evaluation of the MIQP model's performance. It checks if the model has returned a set of coefficients (beta), and if so, it proceeds to calculate the validation and training errors using the predictions from the model. These errors are then recorded in the results DataFrame for each k and fold iteration. The code accumulates the validation errors to calculate the average error across all folds for a given k. It updates the results DataFrame with this average error and compares it with the best error found so far to identify the most optimal k value, updating best_k and best_error accordingly.

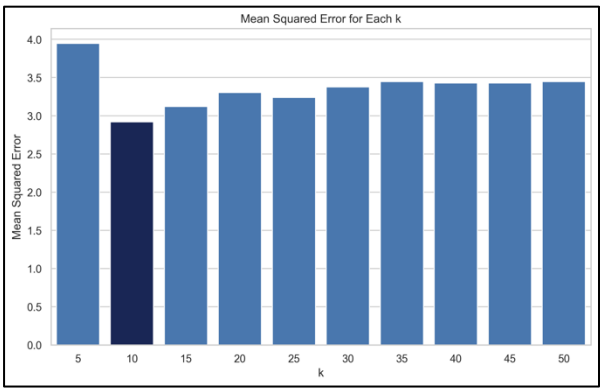


Figure 3: MSE for each 'k' value selected

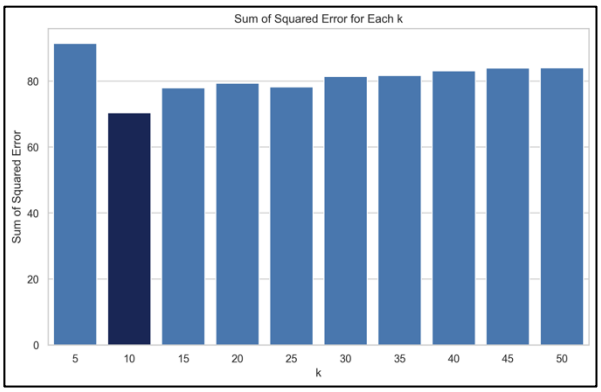


Figure 4: SSE for each 'k' value selected

Through a 10-fold cross-validation process, we've determined that setting **'k'** to **10** yields the most favorable mean squared error of **2.816** and sum of squared error of **70.397** on validation set. This finding is significant as it suggests that only 10 out of the original 50 variables are needed to effectively capture the essential patterns within the data. This degree of variable selection not only simplifies our model but also enhances interpretability without compromising the predictive power, confirming the efficacy of using MIQP for direct variable selection in our analysis.

Results

Having determined the best 'k' to use, we trained our MIQP model using the full training dataset and then made predictions. This resulted in a mean squared error (MSE) of approximately **2.392** on the training set and about **2.337**on the test set. These MSE values offer a quantitative measure of the model's predictive accuracy, indicating how well the model performs, particularly on the test set which represents new, unseen data. The closeness of these two figures suggests that the model has generalized well to the unseen data, indicating a consistent performance from training to testing.

Note: For most of our analysis we used MSE (Mean Squared Error) instead of SSE (Sum of Squared Error) to compare results between train & test datasets between MIQP and LASSO models because MSE normalizes the error by the dataset size, ensuring that the metric is comparable across datasets regardless of their number of observations. SSE, on the other hand, is influenced by the size of the dataset, which can skew comparisons.

Indirect Variable Selection using LASSO Regression

After fine-tuning the number of features with the MIQP model, where we ran 10-fold cross-validation manually to find the optimal number of features. Now, let's talk about the second method which uses Lasso Regression for indirect variable selection. This method involves the objective function similar to least squares but different in the fact that we add a "λ" parameter to penalize the large number of independent variables added to the model and is given by:

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

The "λ" parameter is chosen by cross-validation, large values of λ, more β values will be reduced to zero, this has the advantage of shrinking the β's closer to zero which leads to reduction of variance which in turn reduces the overfitting. We used scikit-learn to solve this problem using Lasso regression.

Implementation

- **Data Scaling:** The training and test datasets were scaled using StandardScaler to ensure that all variables were on the same scale, a necessary step for LASSO to perform effectively.

```
# Separate the features and target variable for training and test sets
X_train = train_data.drop(columns=['y'])
y_train = train_data['y']

X_test = test_data.drop(columns=['y'])
y_test = test_data['y']

# Instantiate the StandardScaler and fit_transform on the training data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Code Block 10: Data pre-processing for Lasso

- **Model Fitting and Cross-Validation:** Like MIQP model where 10-fold cross-validation was preformed to find optimal 'k', LassoCV was used to fit the model on the scaled training data, performing 10-fold cross-validation to find the best λ. The optimal λ identified from this process was **0.084719**. Using this λ, we then retrained the Lasso model on the entire training dataset to ensure it was optimally calibrated.

```
# Instantiate the LassoCV model and perform 10-fold cross-validation on the training set
lasso_cv = LassoCV(cv=10)
lasso_cv.fit(X_train_scaled, y_train)

best_alpha = lasso_cv.alpha_  # Best value of λ
print(f"Best Lambda: {best_alpha}")

# Fit the LASSO model to the entire training set using the best λ
lasso = Lasso(alpha=best_alpha)
lasso.fit(X_train_scaled, y_train)
```

Code Block 11: Logic for 10-fold cross validation on Lasso

- **Training and Test Error Assessment:** The LASSO model was then evaluated on both the training and test sets. The mean squared error (MSE) was calculated for each to assess the model's predictive performance.

```
# Make predictions on the training set
y_pred_train = lasso.predict(X_train_scaled)

# Assess the training error using mean squared error (MSE)
mse_train_lasso = mean_squared_error(y_train, y_pred_train)
print(f"Mean Squared Error on Train Set: {mse_train_lasso}")

# Make predictions on the test set
y_pred_lasso = lasso.predict(X_test_scaled)

# Create a DataFrame with true values, predicted values, and residues
ytrue_ypred_lasso = pd.DataFrame({'True_y': y_test, 'Predicted_y': y_pred_lasso,
    'Residue': y_test - y_pred_lasso})

# Assess the test error using mean squared error (MSE)
mse_test_lasso = mean_squared_error(y_test, y_pred_lasso)
print(f"Mean Squared Error on Test Set: {mse_test_lasso}")
```

Code Block 12: Logic for calculating train and test errors for Lasso

```
Best Lambda: 0.08471942409934509
Mean Squared Error on Train Set: 2.3864444924068957
Mean Squared Error on Test Set: 2.356666383215587
```

Figure 5: Optimal λ, train & test errors for LASSO

Results

When we applied the LASSO model with the optimal lambda (λ) value of **0.0847**, it resulted in a mean squared error (MSE) of **2.386** on the training set and **2.357** on the test set. These MSE values suggest that the model is performing good on both training and unseen test data. A key observation from this implementation was that LASSO selected **18** out of the initial **50** variables in original dataset, effectively reducing the coefficients of the remaining variables to zero. This indicates that just 18 variables are sufficient to capture most of the patterns in the data, highlighting LASSO's ability to streamline feature selection for improved model interpretability.

In contrast, the MIQP model, under similar evaluation conditions, identified an even smaller subset of variables, selecting only **10** as crucial contributors. This difference in the number of selected variables between LASSO and MIQP suggests varying degrees of feature reduction and model complexity. The detailed comparative analysis that follows will delve deeper into how MIQP and LASSO performed, look at which variables each model picked, and discuss which model works best for different kinds of tasks.

Comparative Analysis of MIQP & LASSO Models

After implementing both models and evaluating their performance on a standardized dataset, we have gathered results that will inform our decision on whether to maintain our current reliance on LASSO or to pivot towards employing more direct variable selection methods like MIQP. The advancements in computational solvers have made this reassessment timely and relevant, potentially influencing our firm's approach to predictive analytics. The comparison between MIQP and Lasso will detail the outcomes of our analysis and guide us in making an informed choice that aligns with our strategic objectives.

Predictive Performance

In assessing the performance of the MIQP and Lasso models, we concentrated on training and test error rates as primary indicators of model fit and predictive power.

For the training data, MIQP and Lasso were neck and neck, with their training error rates nearly indistinguishable. This close contest highlights their comparable capability in learning from the training dataset. However, a different picture emerged with the test data, where MIQP edged out Lasso with a marginally lower error rate. This suggests that MIQP could be slightly more adept at handling data it hasn't seen before.

Model	Train Error	Test Error
MIQP	2.391985	2.336544
Lasso	2.386444	2.356666

Table 1: Train & Test MSE comparison for MIQP and Lasso

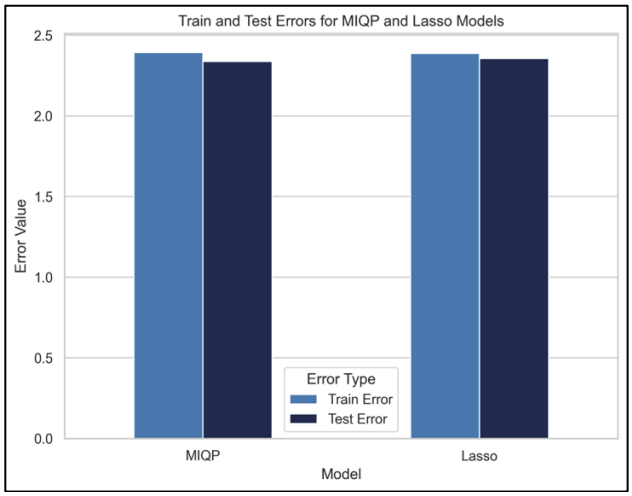


Figure 6: Train & Test MSE for MIQP and Lasso

An interesting note is that despite Lasso's inclusion of more variables, MIQP's selectivity didn't hamper its performance, potentially making it a more streamlined and efficient choice in situations where model simplicity and interpretability are important.

Actuals vs Predictions

While train and test errors provided a high-level overview of model performance, the additional R-squared values and the fit and residue plots offer a deeper and more detailed understanding.

The MIQP model has an R-squared of 0.8587, which means it explains approximately **85.87%** of the variance in the test dataset. In contrast, the Lasso model explains slightly less variance, with an R-squared of 0.8402, or **84.02%**. While both models perform similarly, the higher R-squared for the MIQP model suggests that it captures more of the variance in the data than the Lasso model, which could be an indication of a better fit.

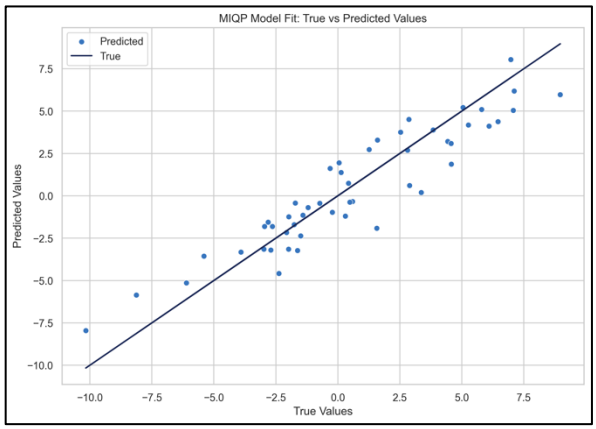


Figure 7: MIQP model fit to test data

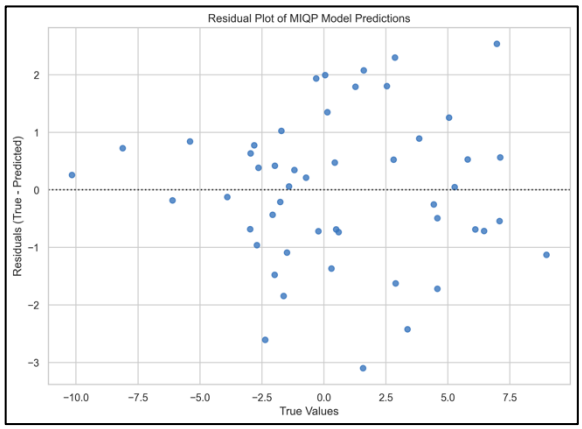


Figure 8: Residue Plot of MIQP model predictions

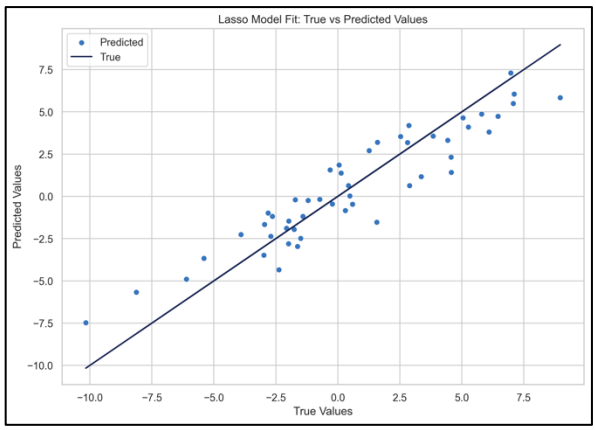


Figure 9: LASSO model fit to test data

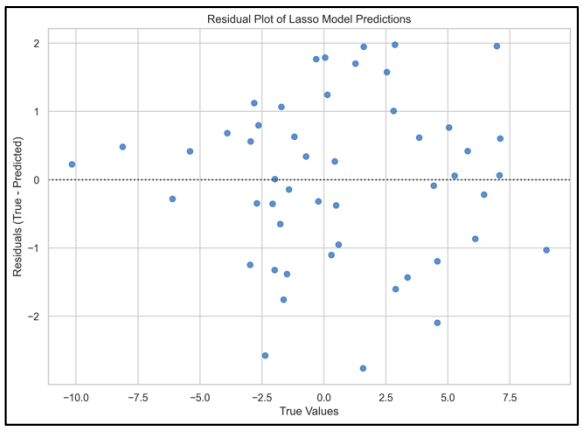


Figure 10: Residue Plot of LASSO model predictions

The fit plots, which graph predicted values against true values, show how closely the models' predictions align with the actual outcomes. Both plots likely show a clustering of points around the line of perfect fit (where predicted equals true), but the scatter may be tighter for the MIQP model, consistent with its higher R-squared value.

Residual plots, displaying the differences between actual and predicted values, offer insight into the distribution of errors. A good model will have residuals evenly scattered around zero, indicating consistent performance across the range of predictions. Any patterns or systematic deviations in these plots would suggest model biases or an inability to capture complex relationships in the data. The residual plots for both MIQP and Lasso models show no apparent patterns, suggesting that both models are capturing the data's underlying structure effectively. Residuals are evenly distributed around the zero line for both models, indicating consistent predictive performance. There is no clear sign of heteroscedasticity, and while some outliers are present, they do not dominate the error distribution, implying that both models are robust in their predictions across the dataset.

Feature Selection

After implementing variable selection with the MIQP model and Lasso Regression, we've seen that MIQP chose just 10 out of the original 50 features, while Lasso selected 18. Notably, MIQP not only picked fewer variables but also outperformed Lasso on the test data as discussed in the predictive performance of both models, indicating its ability to yield a simpler and more interpretable model.

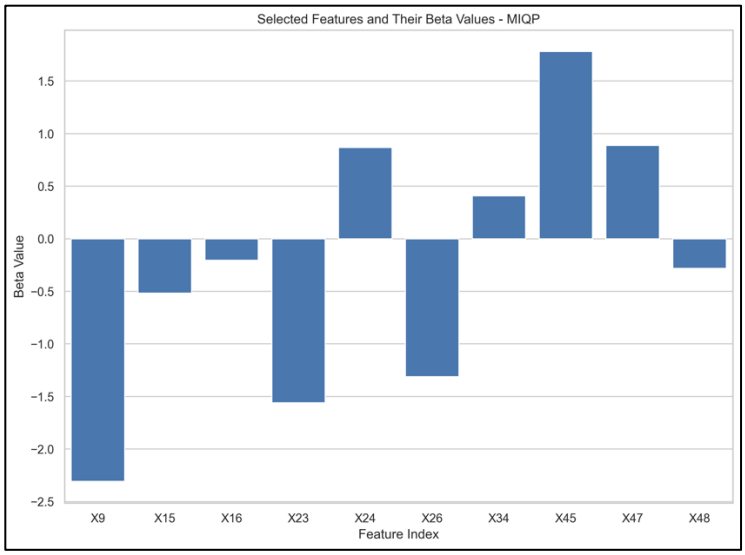


Figure 11: Variables selected by MIQP and corresponding beta values

The table below provides detailed insights into the beta coefficients for each feature in the dataset, derived using MIQP:

beta	value	beta	value	beta	value	beta	value	beta	value
beta_1	0.000000	beta_11	0.000000	beta_21	0.000000	beta_31	0.000000	beta_41	0.000000
beta_2	0.000000	beta_12	0.000000	beta_22	0.000000	beta_32	0.000000	beta_42	0.000000
beta_3	0.000000	beta_13	0.000000	beta_23	-1.559143	beta_33	0.000000	beta_43	0.000000
beta_4	0.000000	beta_14	0.000000	beta_24	0.866973	beta_34	0.408165	beta_44	0.000000
beta_5	0.000000	beta_15	-0.518326	beta_25	0.000000	beta_35	0.000000	beta_45	1.781475
beta_6	0.000000	beta_16	-0.204162	beta_26	-1.311919	beta_36	0.000000	beta_46	0.000000
beta_7	0.000000	beta_17	0.000000	beta_27	0.000000	beta_37	0.000000	beta_47	0.887383

beta_8	0.000000	beta_18	0.000000	beta_28	0.000000	beta_38	0.000000	beta_48	-0.282292
beta_9	-2.308207	beta_19	0.000000	beta_29	0.000000	beta_39	0.000000	beta_49	0.000000
beta_10	0.000000	beta_20	0.000000	beta_30	0.000000	beta_40	0.000000	beta_50	0.000000

Table 2: Beta values for all variables from MIQP model

A closer look at the features chosen by both methods reveals that MIQP's selection is entirely encompassed within Lasso's, suggesting a core subset of features **(X9, X15, X16, X23, X24, X26, X34, X45, X47, X48)** is crucial for capturing the patterns in the dataset. This is further evidenced by the larger beta coefficients assigned to these features by the LASSO model, as visualized in the graphs, highlighting their importance in the model's predictive ability.

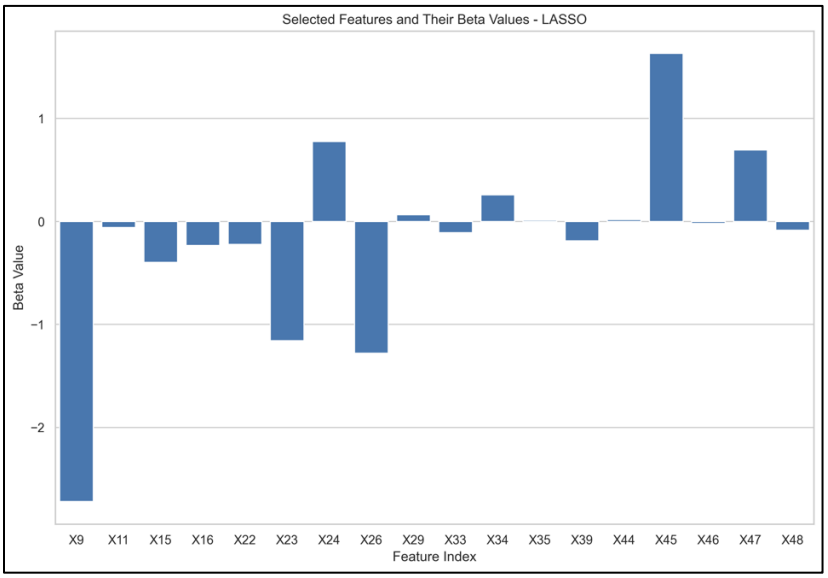


Figure 12: Variables selected by Lasso and corresponding beta values

The table below provides detailed insights into the beta coefficients for each feature in the dataset, derived using LASSO regression:

beta	value	beta	value	beta	value	beta	value	beta	value
beta_1	-0.000000	beta_11	-0.057453	beta_21	0.000000	beta_31	-0.000000	beta_41	-0.000000
beta_2	-0.000000	beta_12	-0.000000	beta_22	-0.221804	beta_32	0.000000	beta_42	0.000000
beta_3	0.000000	beta_13	-0.000000	beta_23	-1.158480	beta_33	-0.108472	beta_43	0.000000
beta_4	0.000000	beta_14	-0.000000	beta_24	0.775694	beta_34	0.257401	beta_44	0.018689
beta_5	-0.000000	beta_15	-0.395855	beta_25	-0.000000	beta_35	0.012777	beta_45	1.631964
beta_6	0.000000	beta_16	-0.232766	beta_26	-1.279501	beta_36	0.000000	beta_46	-0.018390
beta_7	-0.000000	beta_17	0.000000	beta_27	0.000000	beta_37	0.000000	beta_47	0.693846
beta_8	-0.000000	beta_18	0.000000	beta_28	0.000000	beta_38	0.000000	beta_48	-0.084735
beta_9	-2.719827	beta_19	-0.000000	beta_29	0.065303	beta_39	-0.188398	beta_49	0.000000
beta_10	0.000000	beta_20	0.000000	beta_30	0.000000	beta_40	0.000000	beta_50	0.000000

Table 3: Beta values for all variables from Lasso model

In the tables above, the beta coefficients that are zero indicate that the corresponding features have been excluded from the model. This is because predictions (y^) are calculated as the sum of the products of features and their respective beta coefficients. Therefore, features with a beta coefficient of zero do not contribute to the prediction.

Advantages & Disadvantages

S.No:	LASSO Advantages	LASSO Disadvantages	MIQP Advantages	MIQP Disadvantages
1	Computationally efficient due to convex optimization techniques.	Can introduce bias, particularly when the true relationship is non-linear.	Direct approach to variable selection may yield a more precise model.	Historically computationally demanding due to non-convex optimization.
2	Scalable to large datasets.	Selection of the regularization parameter can be sensitive.	Can lead to better interpretability with a clear subset of variables.	Less efficient with very large datasets compared to LASSO.
3	Simple to implement with many off-the-shelf tools available.	May exclude relevant variables if they are highly correlated with others.	Recent advances in computational solvers have made MIQP more accessible.	Requires more sophisticated solvers, which may not be as widely available.
4	Regularization parameter can control the level of sparsity.	Tends to underperform if the number of predictors is much greater than the number of observations.	Potentially lower error rates, as seen in empirical results.	The binary nature of selection can be sensitive to noise in the data.
5	Provides shrinkage, which can improve model performance on unseen data.	Difficulty in handling categorical variables with many levels.	Capable of handling complex non-linear relationships between variables.	Can be overfitting if not properly regularized or if the model complexity is not controlled.

Table 4: Advantages and Disadvantages of MIQP & Lasso

Conclusion

The comprehensive evaluation conducted suggests that the MIQP (Mixed-Integer Quadratic Programming) approach has demonstrated a slightly superior performance when compared to the LASSO (Least Absolute Shrinkage and Selection Operator) method. This conclusion is supported by the observation of a marginally lower Mean Squared Error (MSE) and a slightly higher R-squared value for MIQP, indicating a modest advantage in terms of accuracy for MIQP over LASSO.

The close performance of the MIQP and Lasso models in training error suggests that both methods are robust for fitting the training data. However, the marginally better test error of the MIQP model implies it might be more effective in predicting outcomes on new data. This could be attributed to the direct approach of variable selection in MIQP, which might be capturing more nuanced relationships in the data compared to Lasso's regularization approach.

However, it's crucial to note that the performance gap between the two methods is relatively narrow. One key consideration in this context is the substantial computational demand and more extensive coding requirements associated with MIQP. In contrast, the LASSO method stands out for its significantly shorter computation time and a more streamlined implementation process.

When to use which Model

Use LASSO When:

1. Interpretability is a priority: If you need a model that is easy to interpret and explain to stakeholders, LASSO's sparsity-inducing property makes it a good choice.
2. Computational efficiency matters: For scenarios where computational resources are limited, and faster model training and implementation are crucial, LASSO's simplicity and availability in popular Python libraries make it a practical choice.

Use MIQP When:

1. Maximum accuracy is critical: In applications where achieving the highest level of predictive accuracy is paramount, and computational resources are not a primary concern, MIQP's marginally superior performance might be justified.
2. Specific constraints or integer variables are involved: MIQP is particularly suited for problems where integer programming is essential, and certain decision variables need to take on integer values.

Considering the trade-offs between accuracy and computational efficiency, the suggestion is to persist with the LASSO method for regular variable selection tasks. Although MIQP is slightly accurate for the given problem, it is advised to use it only in situations where achieving the utmost accuracy is crucial. Also, it is important to consider other factors such as size and nature of the dataset, the number of features, the presence of multicollinearity, and the overall goals of the modeling exercise which might alter the factors such as accuracy and interpretability of a model. It is expected that with advancements in computational efficiency, MIQP could become a more feasible substitute for LASSO in the future.