Project 3 - Non Linear Programming

RM 294 - Optimization I - Dr. Mitchell

Ian McIntosh, Serena Song, Bindu Raghu Naga, Evin McDonald

Problem Description

Variable selection for regression is a common challenge in predictive analytics. Historically, direct variable selection through optimization has faced obstacles, leading the statistics and analytics community to largely disregard it due to computational complexities. This difficulty in computation played a role in motivating the development of techniques like LASSO and ridge regression. However, recent progress in optimization software has been remarkable, particularly in the realm of solving mixed integer quadratic programs (MIQP). In this project, we will approach the variable selection problem for regression as an MIQP and leverage the power of Gurobi to solve it. We aim to explore whether this approach yields superior results compared to LASSO, which incorporates a 'shrinkage' component. We want to determine if the pursuit of the 'best' set of variables for inclusion in regression models outweighs the benefits of LASSO's shrinkage technique.

Direct Variable Selection - MIQP Problem

Given a dataset of m independent variables, X, and a dependent variable, y, the standard ordinary least squares problem is formulated as:

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

Figure 1: Standard Ordinary Least Squares Problem Formula

To introduce variable selection into this problem, we can introduce binary variables, denoted as *Zj*. These binary variables are designed to enforce *βj* values to be zero when *Zj* is set to zero, employing the big-M method. If our objective is to include a maximum of k variables from the set X, we can formulate it as follows:

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

Figure 2: Ordinary Least Squares Problem Formula For Only Including K variables From the Dataset of Independent Variables X

Please note that our modeling approach never restricts the inclusion of an intercept term, *β*0. Additionally, it's important to clarify that in this context, the variables m and M have distinct meanings. Here, the parameter k can be considered as a hyperparameter that we determine through cross-validation.

Benefits of this approach:

- Flexible Variable Selection:

  - By introducing binary variables and formulating the problem in this manner, we gain the flexibility to precisely select the variables that have the most significant impact on the dependent variable, y. This flexibility allows us to create more interpretable and potentially more accurate models.
- Intercept Inclusion:
  - This approach preserves the inclusion of an intercept term, $\beta 0$, which is important for capturing the baseline relationship between the independent and dependent variables.
- Hyperparameter Tuning:
  - The parameter k, representing the maximum number of variables to include, can be tuned through cross-validation. Thus, we can adapt the model's complexity to the specific dataset and problem at hand.

Disadvantages of this approach:

- Computational Complexity:
  - The incorporation of binary variables and the big-M method can lead to increased computational complexity, especially when dealing with a large number of variables and constraints. This may result in longer processing times and resource-intensive computations.
- Model Interpretability:
  - While this approach offers precise variable selection, the resulting models can become complex and challenging to interpret, especially when numerous binary variables are introduced. Interpretability may be compromised in favor of model accuracy.

- Risk of Overfitting:
  - If not properly regularized or controlled, this approach can be prone to overfitting, especially when k is set too high. Overfitting can result in models that perform well on the training data but generalize poorly to new, unseen data.

Indirect Variable Selection – LASSO

The LASSO version of regression is posed as:

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

Figure 3: LASSO Version of Regression

Note: λ is a hyperparameter to be chosen using cross-validation.

For practical implementation, the widely-used Python package for solving the LASSO problem is scikit-learn. In the scope of this project, scikit-learn will be our tool of choice for addressing the LASSO regression task.

Advantages of this model

- Sparse Variable Selection:
  - With a sufficiently large λ, the LASSO model automatically selects a subset of important variables by forcing many β coefficients to become zero.
- Overfitting Prevention:

    - The 'shrinking' effect of LASSO, as λ increases, reduces the magnitude of β coefficients, effectively preventing overfitting by simplifying the model.
- Intercept Term Inclusion:
    - The model preserves the inclusion of the intercept term ($\beta 0$) without subjecting it to the λ penalty, ensuring that the model captures the essential baseline relationship within the data.

Disadvantages of this model

- Variable Selection Challenge:
    - While LASSO is effective at variable selection, it may sometimes exclude variables that are theoretically relevant but happen to have small coefficients. This can lead to a loss of important information.
- Collinearity Handling:
    - LASSO may struggle with highly correlated predictor variables, as it tends to select one variable from a correlated group and set others to zero arbitrarily. This can lead to instability in variable selection.
- Model Sensitivity:
    - The effectiveness of LASSO is highly dependent on the choice of the regularization parameter λ. Small variations in λ can lead to different sets of selected variables, making the model sensitive to parameter tuning.

DATA + Model Outline

DATA:

There are two datasets that include x and y data:

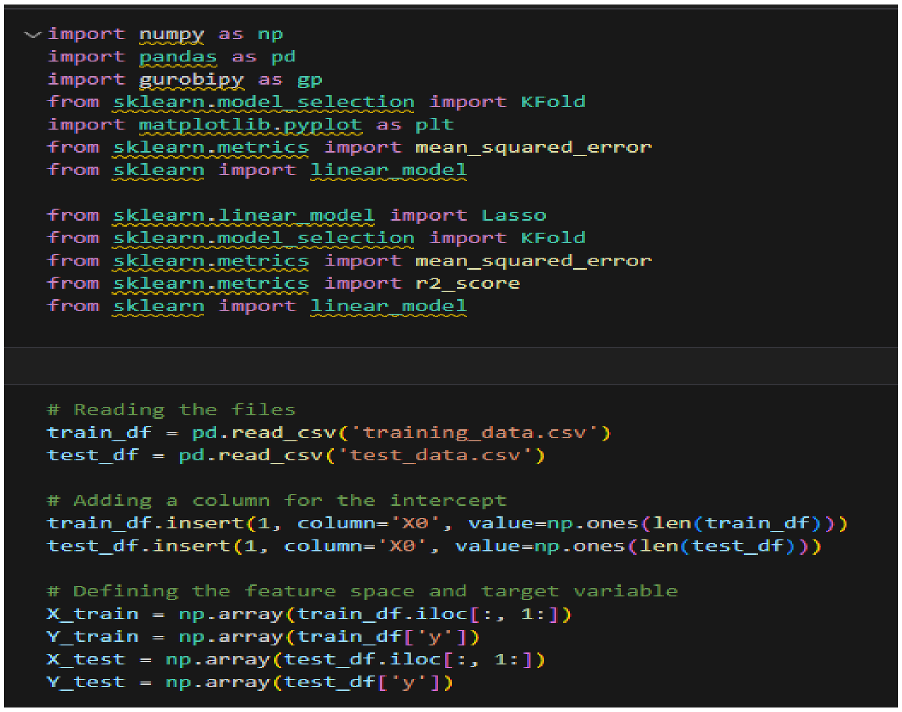
- One of them is training data which can be downloaded with this link: training_data.csv
- The other one is testing data which can be downloaded with this link: test_data.csv

Outline:

- Cross-Validation for Hyperparameter Selection (k or λ):
  - We perform 10-fold cross-validation on the training set and then evaluate different values of k or λ to determine the optimal one.
- Model Training with Optimal Hyperparameter:
  - We utilize the optimal value of k or λ obtained from cross-validation to then fit the β coefficients using the entire training set.
- Prediction on Test Set:
  - We apply the trained model with β coefficients to make predictions for the y values in the test set.
- Comparison to True Values:
  - We compare the predicted y values to the true y values in the test set to assess model performance.

Setting Up Model to Select Hyper Parameters, Predict Test Set Values

The code used to start our model, including importing packages/functions, reading files, adding an intercept column, and defining the feature space and target variable, is as follows:



```
import numpy as np
import pandas as pd
import gurobipy as gp
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn import linear_model

from sklearn.linear_model import Lasso
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from sklearn import linear_model
```

```
# Reading the files
train_df = pd.read_csv('training_data.csv')
test_df = pd.read_csv('test_data.csv')

# Adding a column for the intercept
train_df.insert(1, column='X0', value=np.ones(len(train_df)))
test_df.insert(1, column='X0', value=np.ones(len(test_df)))

# Defining the feature space and target variable
X_train = np.array(train_df.iloc[:, 1:])
Y_train = np.array(train_df['y'])
X_test = np.array(test_df.iloc[:, 1:])
Y_test = np.array(test_df['y'])
```

Figure 4: Code Used to Read in Data, Define Feature Space and Target Variable

Now that we have a training data set and a test data set it is important to carefully follow the data science pipeline. The first step in this process is to do 10-fold cross-validation on the training set to pick our hyperparameter.

Below is a snippet of code that conveys the time limit we set per iteration and the number of folds we specified for cross-validation (10):

```
TIME_LIMIT = 360  # Set the time limit per iteration

n_folds = 10  # Specify the number of folds for cross-validation

np.random.seed(seed=42)
```

Figure 5: Code For Time Limit per Iteration and Number of Folds for CV

Writing Cross-Validation Code to Perform on MIQP Model

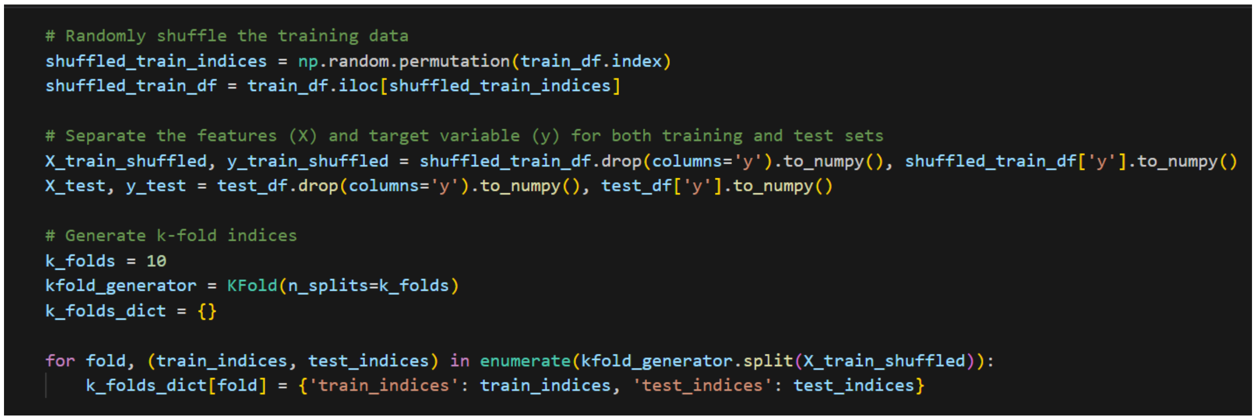
Before we begin with the cross-validation, there are a couple of things we need to be sure of:

1. The training data is randomly rearranged to ensure unbiased model training.

2. Features (characteristics) and the target variable (outcome) are separated for training and testing purposes.

3. K-fold indices are created, dividing the shuffled training data into 10 subsets for robust model evaluation. The printed indices show the training and testing data for the first fold.

The code we used to ensure these three things occur is shown below.



```
# Randomly shuffle the training data
shuffled_train_indices = np.random.permutation(train_df.index)
shuffled_train_df = train_df.iloc[shuffled_train_indices]

# Separate the features (X) and target variable (y) for both training and test sets
X_train_shuffled, y_train_shuffled = shuffled_train_df.drop(columns='y').to_numpy(), shuffled_train_df['y'].to_numpy()
X_test, y_test = test_df.drop(columns='y').to_numpy(), test_df['y'].to_numpy()

# Generate k-fold indices
k_folds = 10
kfold_generator = KFold(n_splits=k_folds)
k_folds_dict = {}

for fold, (train_indices, test_indices) in enumerate(kfold_generator.split(X_train_shuffled)):
    k_folds_dict[fold] = {'train_indices': train_indices, 'test_indices': test_indices}
```

Figure 6: Code For Preparing Data, Features, Target Variable and K-Fold Indices

After this, we decided to create an optimization function for the regression model. This function optimizes a regression model using Gurobi optimization, finding the best beta coefficients for a given k value. It uses mathematical constraints and objectives to balance the model’s complexity and accuracy. The result is a set of optimal beta coefficients, crucial for predicting outcomes from input data.

This function also factored in some constraints:

1. **Sum of Independent Betas Constraint (`< k`):**

   - Purpose: Limits the total number of independent beta coefficients (excluding the intercept) to be less than or equal to a specified value, `k`.

   - Importance: Controls model complexity by restricting the number of predictor variables considered, preventing overfitting and enhancing generalization to new data.

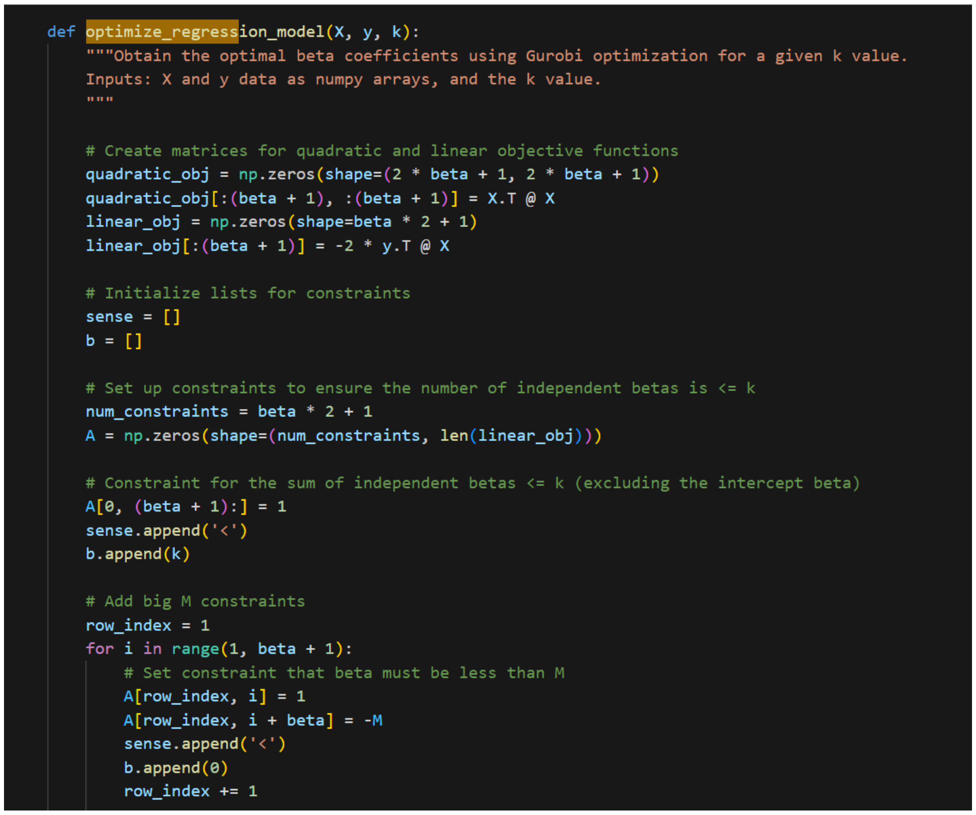
2. **Big M Constraints (`< M` and `> -M`):**

   - Purpose: Enforces upper and lower bounds on each independent beta coefficient to balance precision and numerical stability.

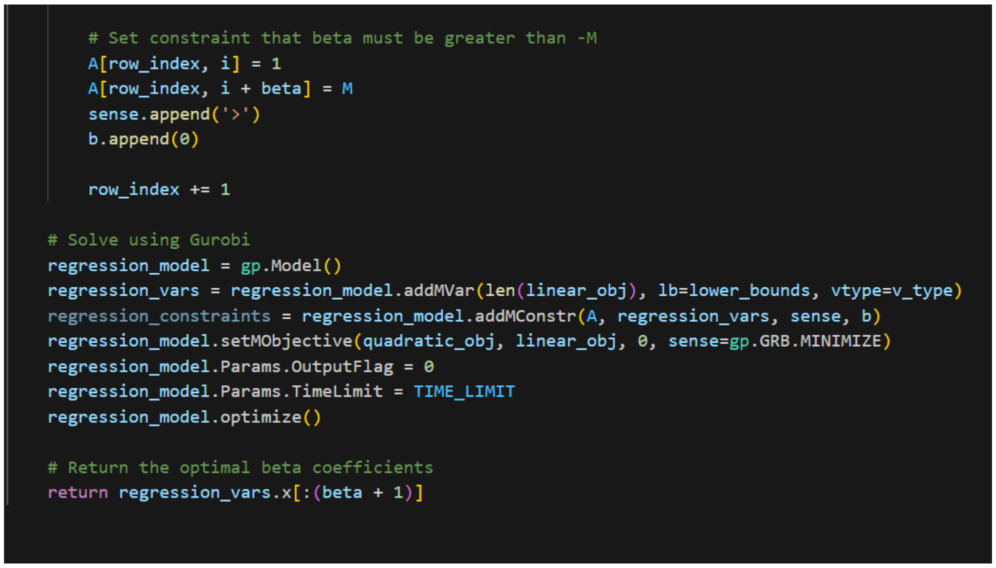
   - Importance: These constraints prevent excessively large or small beta values, ensuring a stable optimization process and preventing numerical issues. They are crucial for the convergence and reliability of the optimization algorithm.

Overall, these constraints contribute to the creation of a well-regularized regression model by controlling the number of predictors and preventing extreme beta values, promoting a balance between model simplicity and accuracy.

The code used to create this function is shown below:



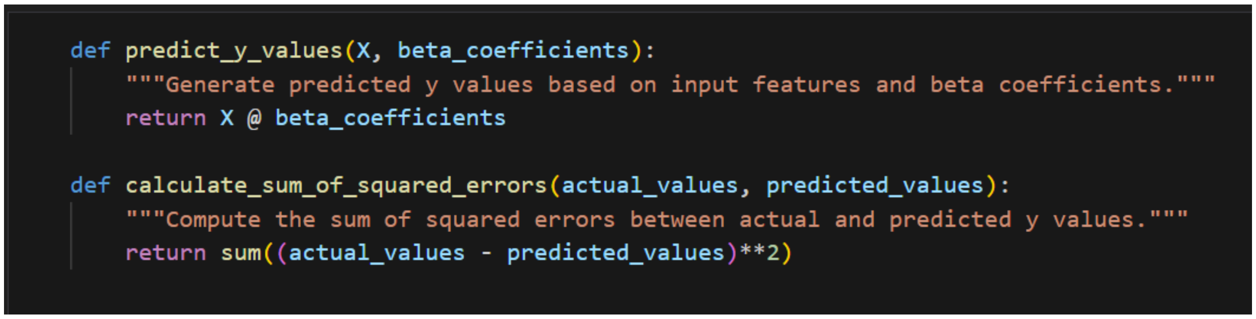
```python
def optimize_regression_model(X, y, k):
    """Obtain the optimal beta coefficients using Gurobi optimization for a given k value.
    Inputs: X and y data as numpy arrays, and the k value.
    """

    # Create matrices for quadratic and linear objective functions
    quadratic_obj = np.zeros(shape=(2 * beta + 1, 2 * beta + 1))
    quadratic_obj[:(beta + 1), :(beta + 1)] = X.T @ X
    linear_obj = np.zeros(shape=beta * 2 + 1)
    linear_obj[:(beta + 1)] = -2 * y.T @ X

    # Initialize lists for constraints
    sense = []
    b = []

    # Set up constraints to ensure the number of independent betas is <= k
    num_constraints = beta * 2 + 1
    A = np.zeros(shape=(num_constraints, len(linear_obj)))

    # Constraint for the sum of independent betas <= k (excluding the intercept beta)
    A[0, (beta + 1):] = 1
    sense.append('<')
    b.append(k)

    # Add big M constraints
    row_index = 1
    for i in range(1, beta + 1):
        # Set constraint that beta must be less than M
        A[row_index, i] = 1
        A[row_index, i + beta] = -M
        sense.append('<')
        b.append(0)
        row_index += 1
```



```python
        # Set constraint that beta must be greater than -M
        A[row_index, i] = 1
        A[row_index, i + beta] = M
        sense.append('>')
        b.append(0)

        row_index += 1

    # Solve using Gurobi
    regression_model = gp.Model()
    regression_vars = regression_model.addMVar(len(linear_obj), lb=lower_bounds, vtype=v_type)
    regression_constraints = regression_model.addMConstr(A, regression_vars, sense, b)
    regression_model.setMObjective(quadratic_obj, linear_obj, 0, sense=gp.GRB.MINIMIZE)
    regression_model.Params.OutputFlag = 0
    regression_model.Params.TimeLimit = TIME_LIMIT
    regression_model.optimize()

    # Return the optimal beta coefficients
    return regression_vars.x[:(beta + 1)]
```

Figures 7 & 8: Code Used to Write Function to Optimize Regression Model

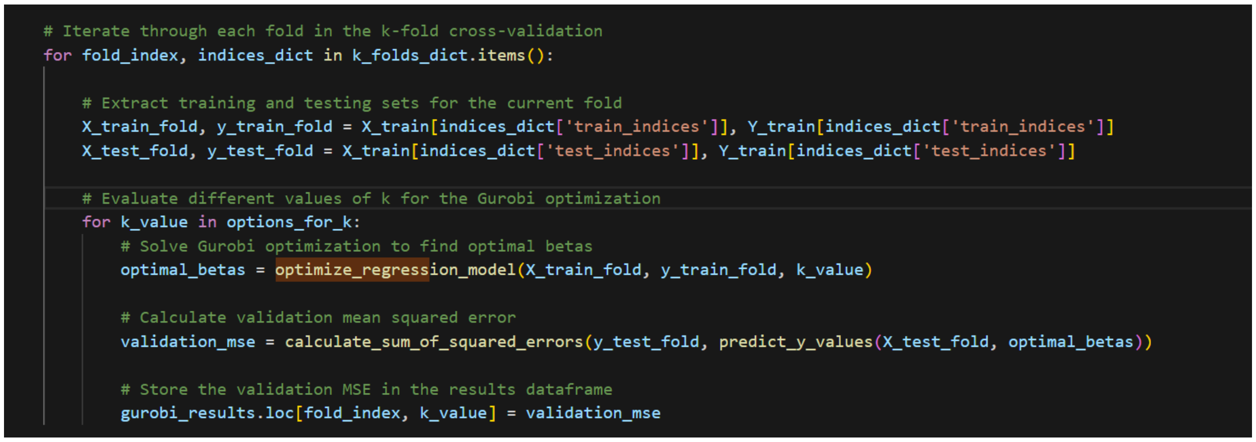
We also defined functions to predict the y values of the test set, as well as to calculate the sum of squared errors for the model. The code used to do so is shown below.



```python
def predict_y_values(X, beta_coefficients):
    """Generate predicted y values based on input features and beta coefficients."""
    return X @ beta_coefficients

def calculate_sum_of_squared_errors(actual_values, predicted_values):
    """Compute the sum of squared errors between actual and predicted y values."""
    return sum((actual_values - predicted_values)**2)
```

Figure 9: Code Used to Predict Y Values, Calculate SSE

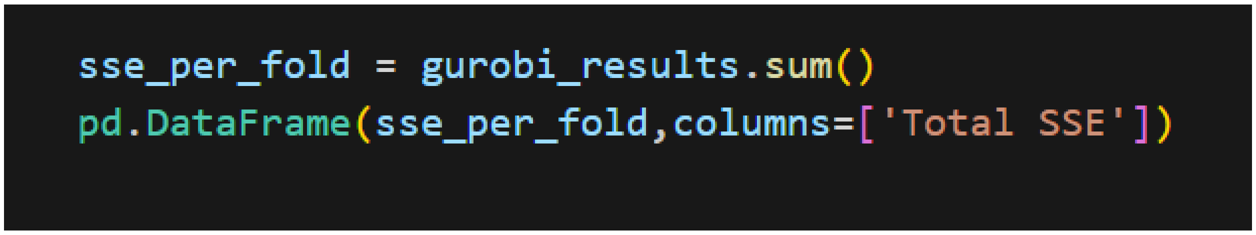
From here we were able to run code that performs k-fold cross-validation to assess a Gurobi-based regression model's performance with varying predictor counts (k). It optimizes beta coefficients, calculates validation mean squared error, and records results for analysis, aiding in predictor selection and model evaluation. The code to do so is shown below.



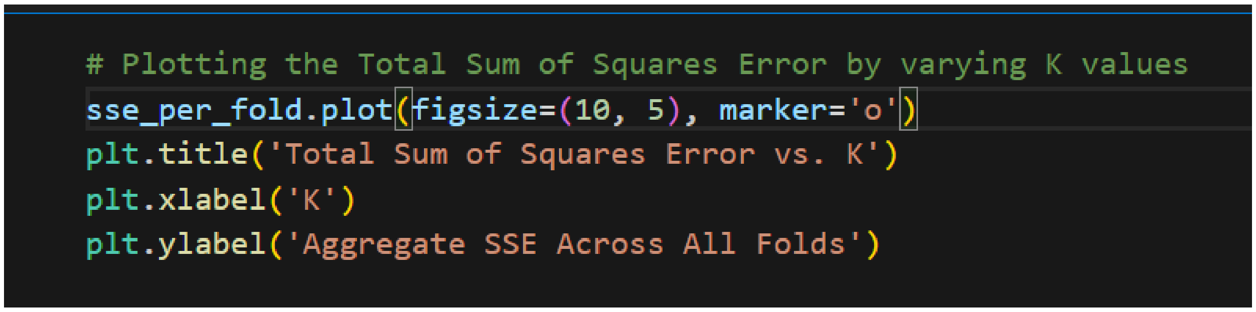
```python
# Iterate through each fold in the k-fold cross-validation
for fold_index, indices_dict in k_folds_dict.items():

    # Extract training and testing sets for the current fold
    X_train_fold, y_train_fold = X_train[indices_dict['train_indices']], Y_train[indices_dict['train_indices']]
    X_test_fold, y_test_fold = X_train[indices_dict['test_indices']], Y_train[indices_dict['test_indices']]

    # Evaluate different values of k for the Gurobi optimization
    for k_value in options_for_k:
        # Solve Gurobi optimization to find optimal betas
        optimal_betas = optimize_regression_model(X_train_fold, y_train_fold, k_value)

        # Calculate validation mean squared error
        validation_mse = calculate_sum_of_squared_errors(y_test_fold, predict_y_values(X_test_fold, optimal_betas))

        # Store the validation MSE in the results dataframe
        gurobi_results.loc[fold_index, k_value] = validation_mse
```

Figure 10: Code Used to Perform K-Fold Cross-Validation

After this, we were able to decide on the optimal hyperparameter value for k. To do this, we calculated the total sum of squared errors from each k value and plotted the results, which are shown below, along with the code used to produce them.



```
sse_per_fold = gurobi_results.sum()
pd.DataFrame(sse_per_fold,columns=['Total SSE'])
```



```
# Plotting the Total Sum of Squares Error by varying K values
sse_per_fold.plot(figsize=(10, 5), marker='o')
plt.title('Total Sum of Squares Error vs. K')
plt.xlabel('K')
plt.ylabel('Aggregate SSE Across All Folds')
```

Figures 11 & 12: Code Used to Calculate, Plot Results

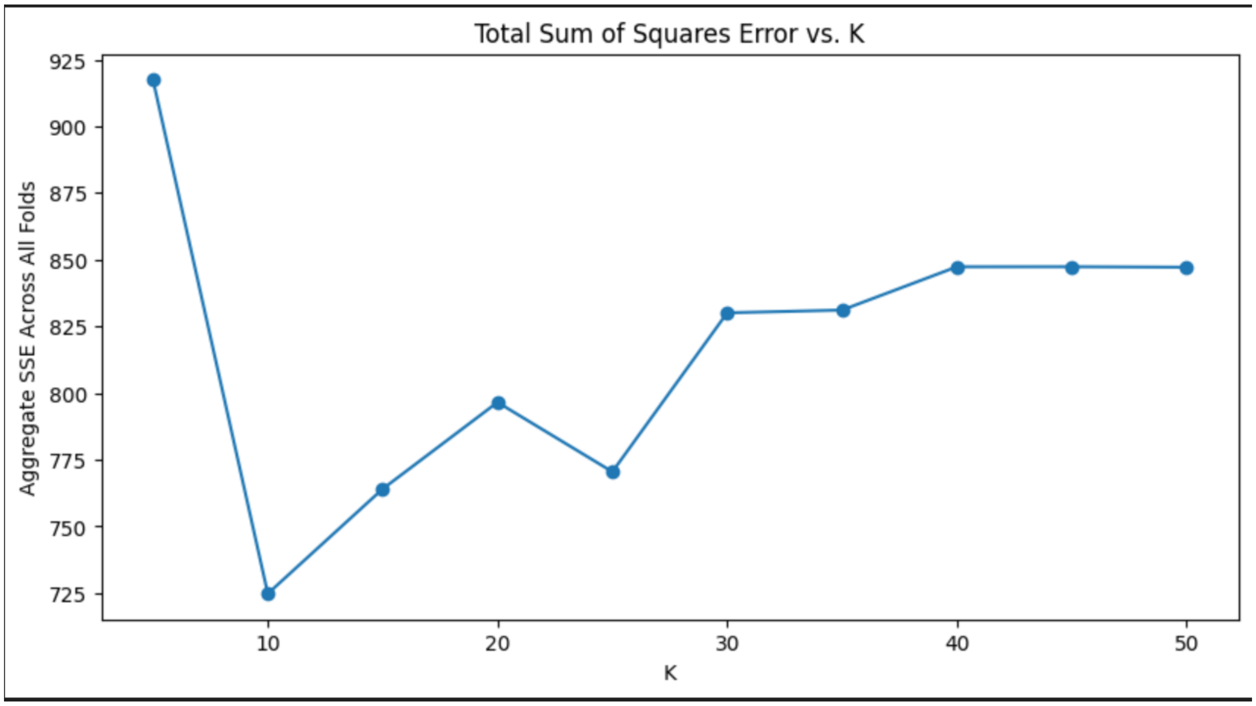
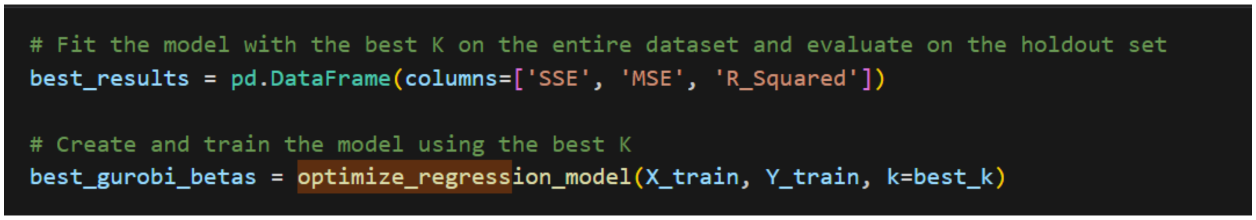


Figure 13: SSE For Different Values of K

The results show us that SSE is minimized when k is equal to 10. Therefore, we must choose 10 as our hyper parameter value.

Fitting Model and Making Predictions

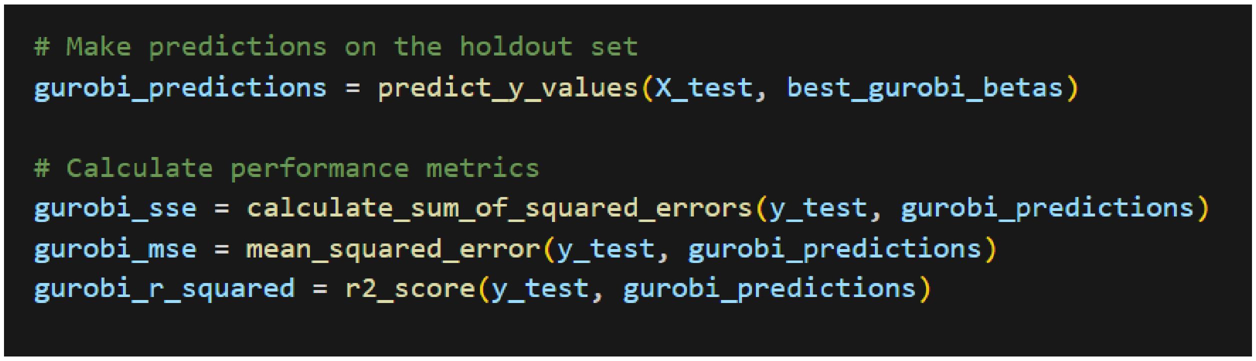
Now that we have found k=10 to have the smallest cross validation error, we can fit our MIQP model on the entire training set using this k value. The code for this is shown below.



```
# Fit the model with the best K on the entire dataset and evaluate on the holdout set
best_results = pd.DataFrame(columns=['SSE', 'MSE', 'R_Squared'])

# Create and train the model using the best K
best_gurobi_betas = optimize_regression_model(X_train, Y_train, k=best_k)
```

Figure 14: Code for Fitting Model

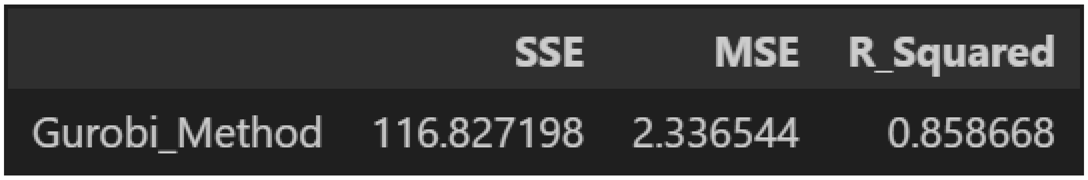
Once the model is fit we can use the beta values we find in the MIQP in order to make predictions of the y values in the test set. Once the predictions are made we can evaluate our model by comparing our predicted values to the actual values. We did this using the following code.



```
# Make predictions on the holdout set
gurobi_predictions = predict_y_values(X_test, best_gurobi_betas)

# Calculate performance metrics
gurobi_sse = calculate_sum_of_squared_errors(y_test, gurobi_predictions)
gurobi_mse = mean_squared_error(y_test, gurobi_predictions)
gurobi_r_squared = r2_score(y_test, gurobi_predictions)
```

Figure 15: Code for Predicting Values, Evaluating Model

The results from our model are shown below:



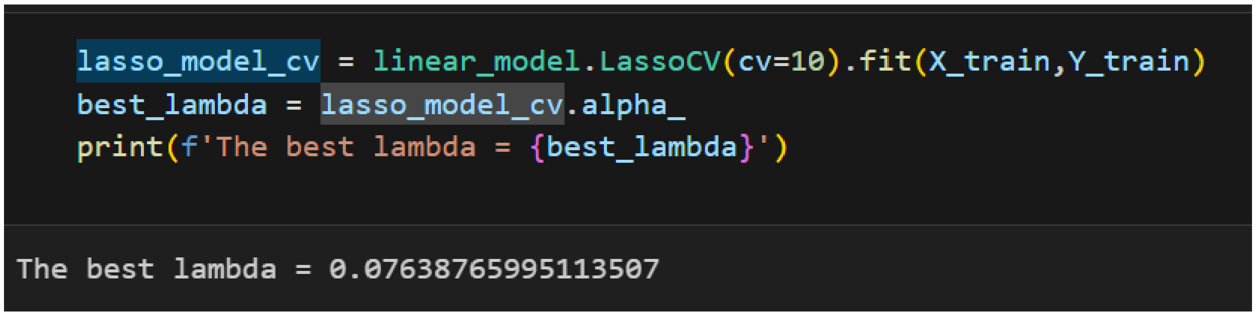
	SSE	MSE	R_Squared
Gurobi_Method	116.827198	2.336544	0.858668

Figure 16: Model Results

The optimal K value, 10, was determined through cross-validation. For the Gurobi-based regression method, the results indicate a Sum of Squared Errors (SSE) of 116.83, a Mean Squared Error (MSE) of 2.34, and a high R-squared value of 0.86. These metrics suggest the model's effectiveness in explaining the variability in the data, with lower MSE and higher R-squared values indicating better performance.

Making Predictions Using LASSO

Next, we are going to perform a similar process, but we are going to use a LASSO model instead of an MIQP model. To fit a LASSO model we need to pick the correct hyper parameter value. Similar to the MIQP model, we are going to use cross-validation to select this hyperparameter. Using scikit-learn, we were able to find that the optimal hyperparameter value for lambda was about 0.076. The code used to do this is shown below, as well as the results, are shown below:

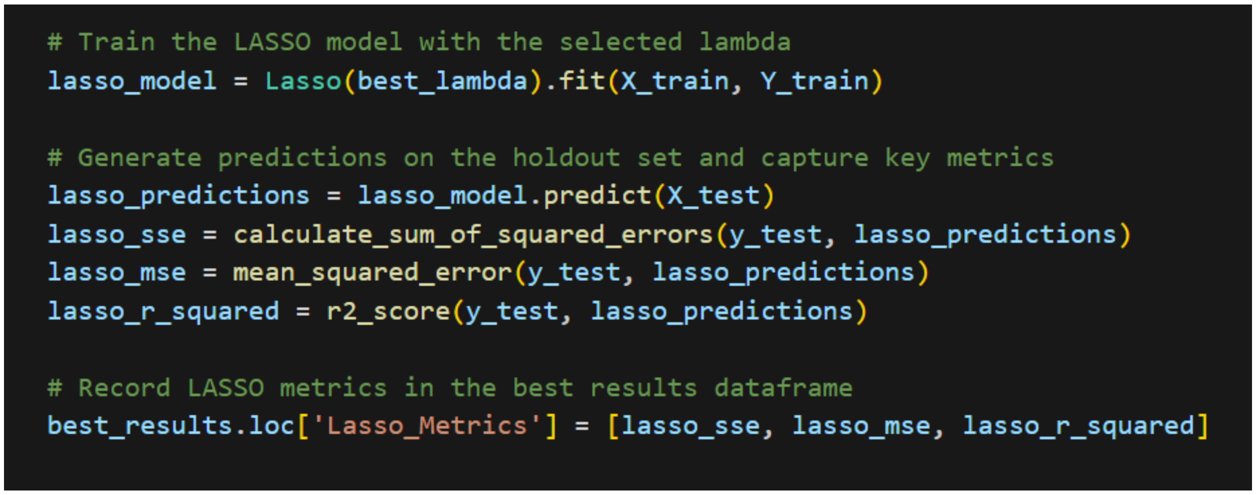


```
lasso_model_cv = linear_model.LassoCV(cv=10).fit(X_train,Y_train)
best_lambda = lasso_model_cv.alpha_
print(f'The best lambda = {best_lambda}')
```

```
The best lambda = 0.07638765995113507
```

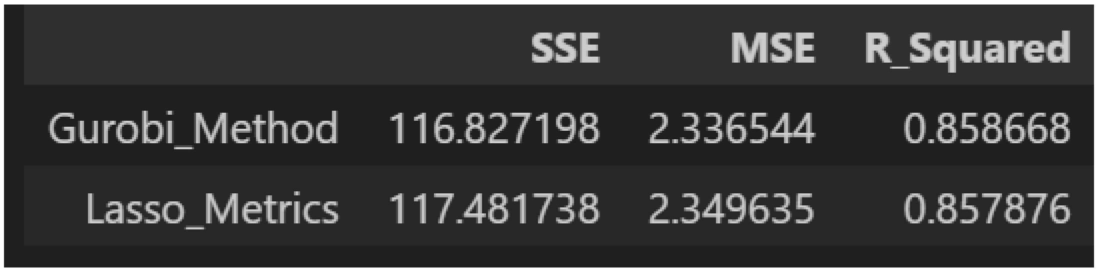
Figure 17: Code to Select Hyper Parameter, Results

Once we had selected our lambda value, we were able to fit the LASSO model to the entire training set using that lambda value. Using the beta values we found in the LASSO model we were able to make predictions on the y values in the test set. With these predictions, we evaluated the performance of the model by comparing the predicted values to the actual values in the test set. The code used to carry out these steps, as well as the results of the LASSO model compared to the MIQP model are shown below.



```python
# Train the LASSO model with the selected lambda
lasso_model = Lasso(best_lambda).fit(X_train, Y_train)

# Generate predictions on the holdout set and capture key metrics
lasso_predictions = lasso_model.predict(X_test)
lasso_sse = calculate_sum_of_squared_errors(y_test, lasso_predictions)
lasso_mse = mean_squared_error(y_test, lasso_predictions)
lasso_r_squared = r2_score(y_test, lasso_predictions)

# Record LASSO metrics in the best results dataframe
best_results.loc['Lasso_Metrics'] = [lasso_sse, lasso_mse, lasso_r_squared]
```

Figure 18: Code for Evaluation of the LASSO Model



	SSE	MSE	R_Squared
Gurobi_Method	116.827198	2.336544	0.858668
Lasso_Metrics	117.481738	2.349635	0.857876

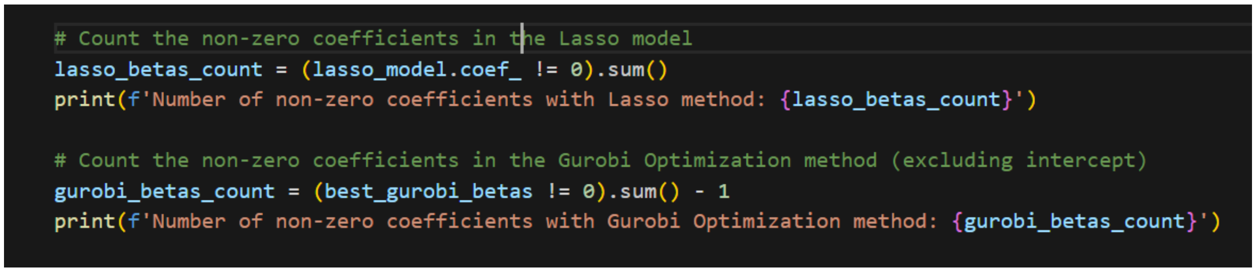
Figure 19: Results of the LASSO Model Compared to the MIQP Model

The Gurobi Method slightly outperforms the Lasso Method in terms of SSE and MSE, with lower values indicating better predictive accuracy. However, both methods demonstrate a comparable R-squared value of approximately 0.86, suggesting a similar ability to explain the variability in the data.

In summary, while the Gurobi Method marginally excels in certain error metrics, both methods exhibit strong performance in capturing and explaining the underlying patterns in the dataset.

More On MIQP vs. LASSO

The next method we used to compare the two methods was looking at the number of non-zero coefficients with each method. The code used to do this is shown below:

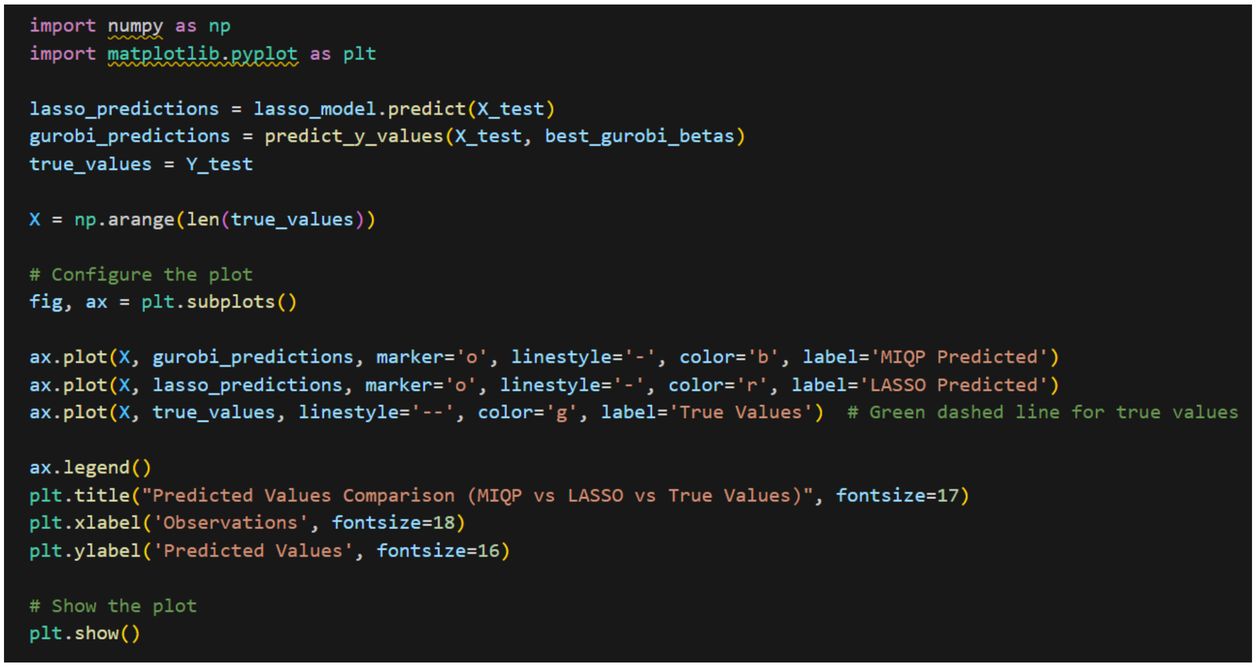


```python
# Count the non-zero coefficients in the Lasso model
lasso_betas_count = (lasso_model.coef_ != 0).sum()
print(f'Number of non-zero coefficients with Lasso method: {lasso_betas_count}')

# Count the non-zero coefficients in the Gurobi Optimization method (excluding intercept)
gurobi_betas_count = (best_gurobi_betas != 0).sum() - 1
print(f'Number of non-zero coefficients with Gurobi Optimization method: {gurobi_betas_count}')
```

Figure 20: Code for Determining Number of Non-Zero Coefficients

We are comparing the number of non-zero coefficients, or 'betas,' identified by two different methods for variable selection. The Lasso method resulted in 17 non-zero betas, indicating the features that significantly contribute to the model. On the other hand, the Gurobi Optimization method identified 10 non-zero betas (excluding the intercept), offering an alternative perspective on the essential variables for our predictive model. This comparison helps us understand the varying emphasis on features selected by each method and their potential impact on model interpretability and performance.

Next, we wanted to compare the predicted values of each model alongside the actual values of the test set. We plotted all three sets of values on one plot using the code shown below. The resulting plot is also shown.



```
import numpy as np
import matplotlib.pyplot as plt

lasso_predictions = lasso_model.predict(X_test)
gurobi_predictions = predict_y_values(X_test, best_gurobi_betas)
true_values = Y_test

X = np.arange(len(true_values))

# Configure the plot
fig, ax = plt.subplots()

ax.plot(X, gurobi_predictions, marker='o', linestyle='-', color='b', label='MIQP Predicted')
ax.plot(X, lasso_predictions, marker='o', linestyle='-', color='r', label='LASSO Predicted')
ax.plot(X, true_values, linestyle='--', color='g', label='True Values')  # Green dashed line for true values

ax.legend()
plt.title("Predicted Values Comparison (MIQP vs LASSO vs True Values)", fontsize=17)
plt.xlabel('Observations', fontsize=18)
plt.ylabel('Predicted Values', fontsize=16)

# Show the plot
plt.show()
```

Figure 21: Code Used to Compare Predicted Values of Each Model with Actual Values

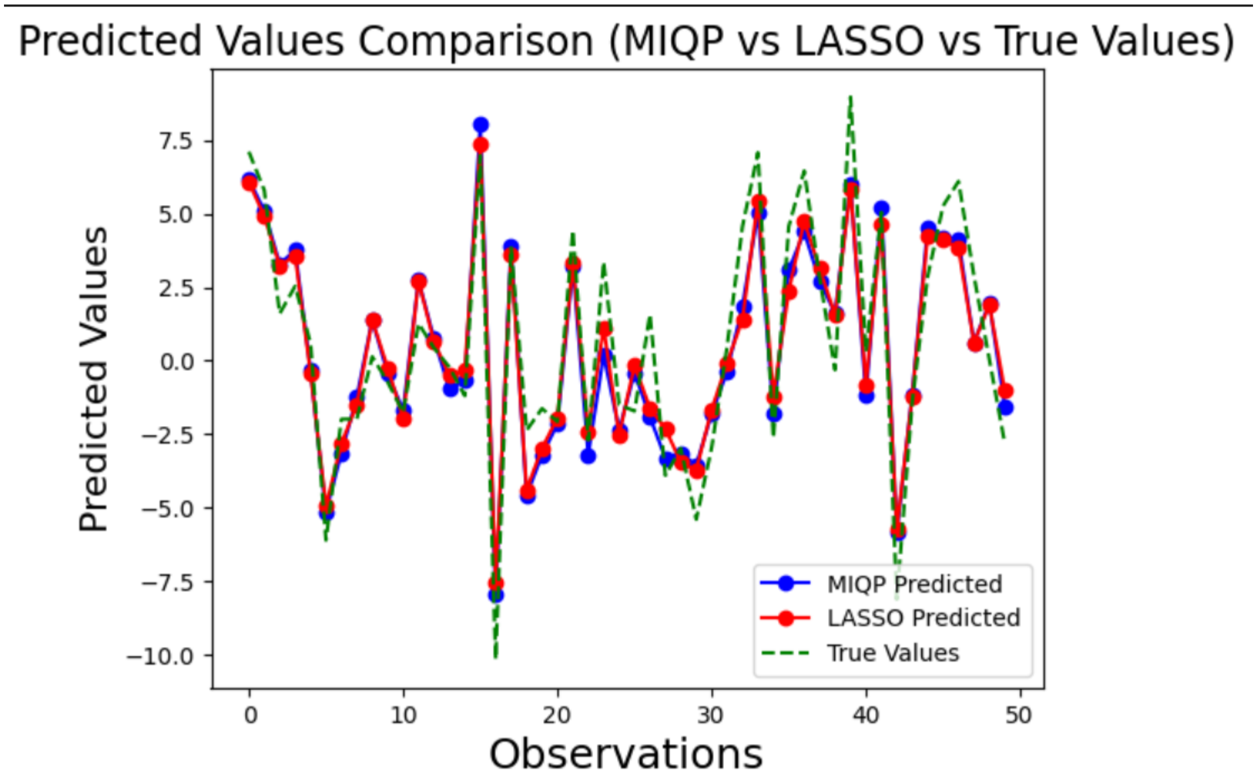
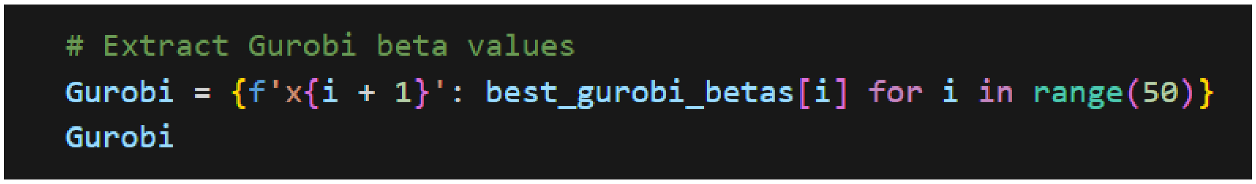
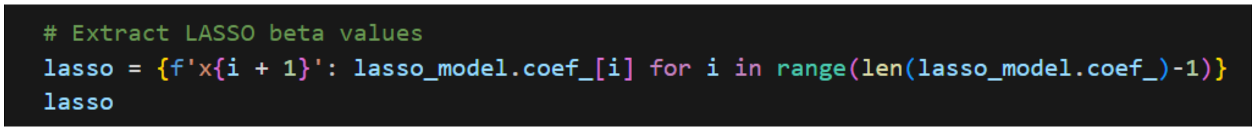


Figure 22: Predicted Values Comparison

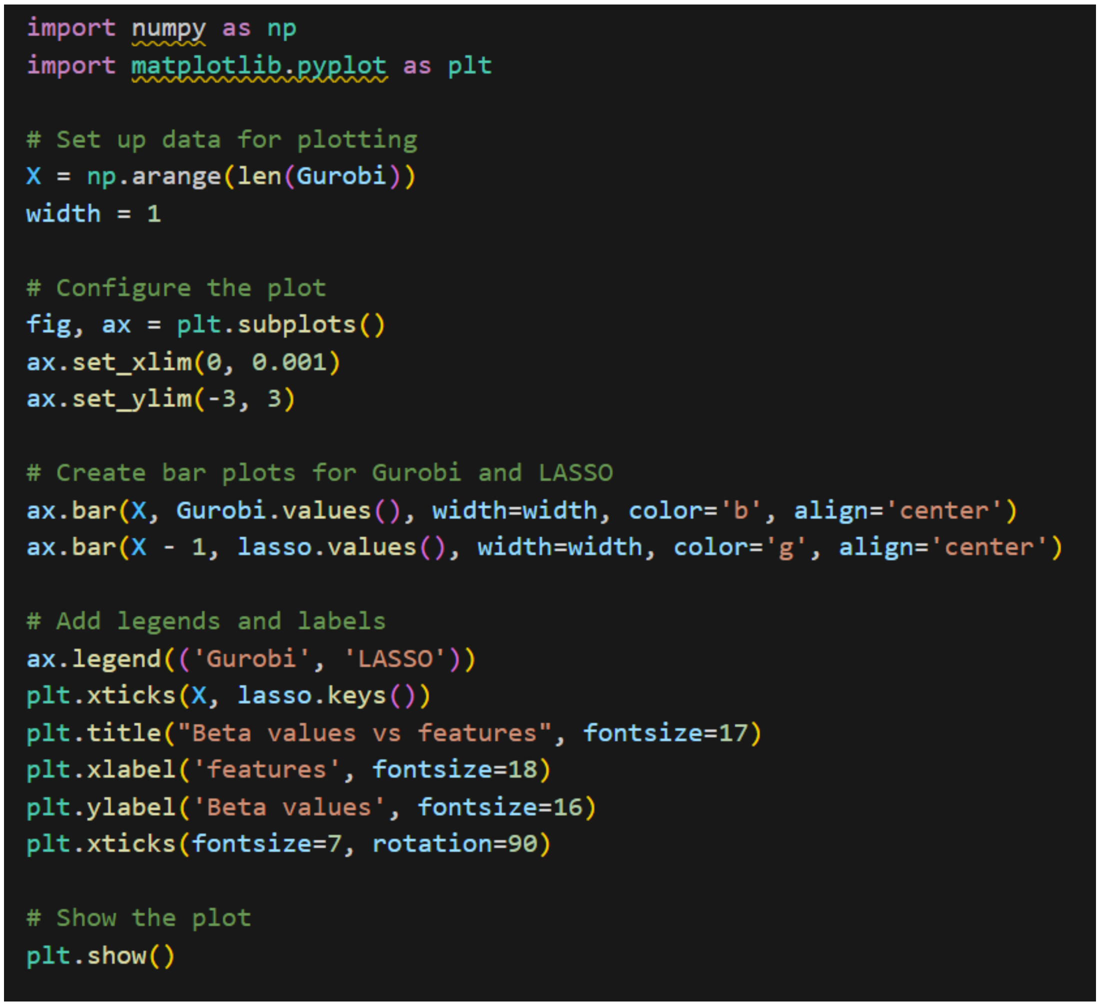
Finally, we wanted to compare the beta coefficients for each variable selection method. To do so, we first had to extract the beta values and then plot them together. The code used to plot the beta values as well as the resulting plot are shown below:



```python
# Extract Gurobi beta values
Gurobi = {f'x{i + 1}': best_gurobi_betas[i] for i in range(50)}
Gurobi
```



```python
# Extract LASSO beta values
lasso = {f'x{i + 1}': lasso_model.coef_[i] for i in range(len(lasso_model.coef_)-1)}
lasso
```



```python
import numpy as np
import matplotlib.pyplot as plt

# Set up data for plotting
X = np.arange(len(Gurobi))
width = 1

# Configure the plot
fig, ax = plt.subplots()
ax.set_xlim(0, 0.001)
ax.set_ylim(-3, 3)

# Create bar plots for Gurobi and LASSO
ax.bar(X, Gurobi.values(), width=width, color='b', align='center')
ax.bar(X - 1, lasso.values(), width=width, color='g', align='center')

# Add legends and labels
ax.legend(('Gurobi', 'LASSO'))
plt.xticks(X, lasso.keys())
plt.title("Beta values vs features", fontsize=17)
plt.xlabel('features', fontsize=18)
plt.ylabel('Beta values', fontsize=16)
plt.xticks(fontsize=7, rotation=90)

# Show the plot
plt.show()
```

Figures 23-25: Code Used to Extract, Plot Beta Values

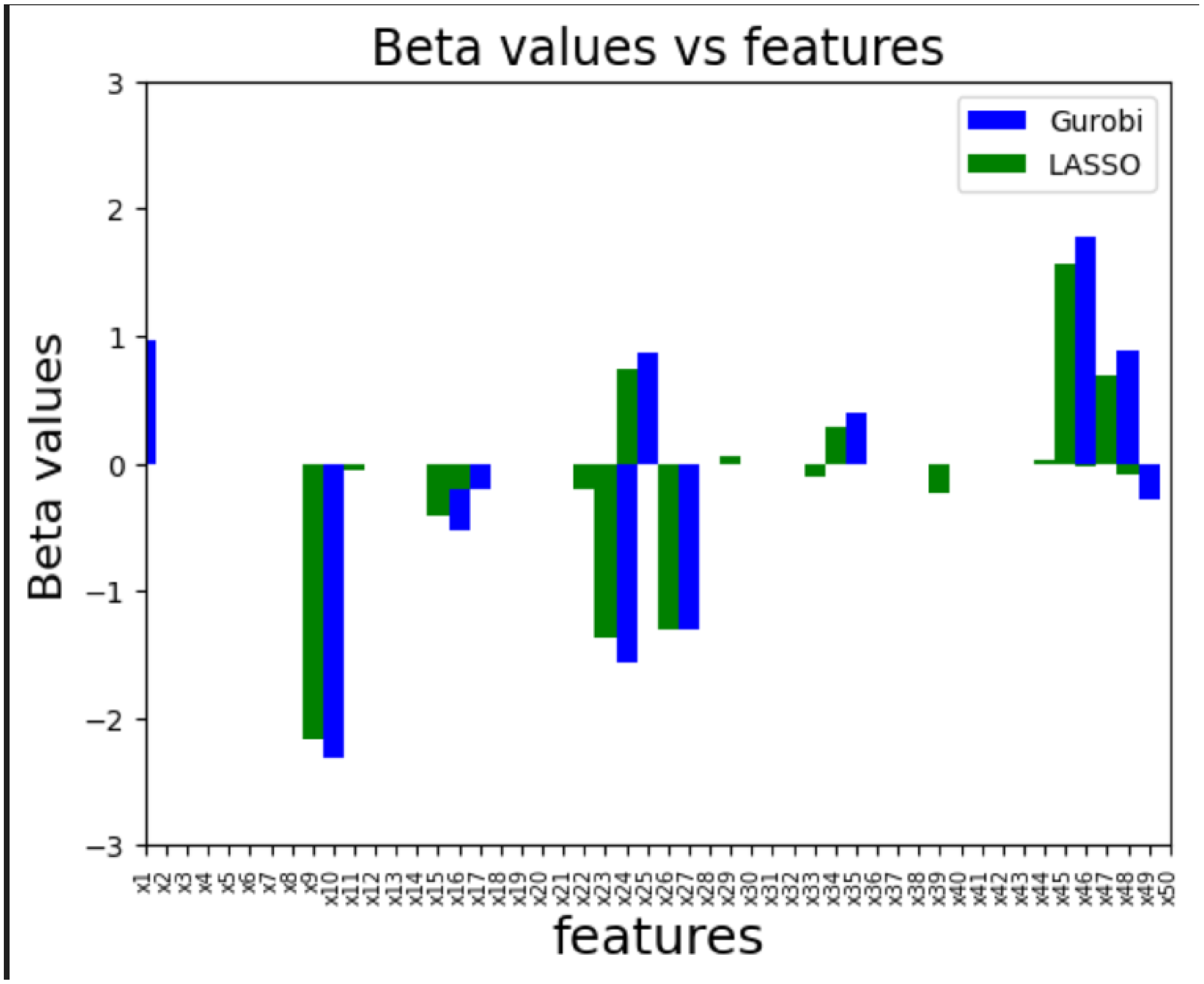


Figure 26: Beta Values for Each Variable Selection Method

Presented here is a comparative bar graph illustrating the $\boldsymbol{\beta}$ coefficients for MIQP and LASSO variable selections. As elaborated in prior sections, it is evident that LASSO identifies 17 significant features, while MIQP identifies only 10, with the majority of non-intercept betas falling within the [-2, 2] range. Furthermore, noteworthy differences exist in the important features identified by each method: certain features deemed crucial for predicting Y using MIQP may not hold the same significance for LASSO, or they may exhibit different signs. For instance, X1 emerges as an important predictor in the MIQP model but possesses a beta value of 0 in the LASSO model. Conversely, X8, which is crucial in predicting Y in the LASSO model, has a beta value of 0 in the MIQP model. Additionally, X24 exhibits a positive beta value in the MIQP model but a negative beta value in the LASSO model.

Conclusions

In summary, while MIQP, as a direct variable selection method, enhances test set accuracy, it comes at the expense of increased computational demands and longer processing times, potentially posing challenges with larger datasets. The MIQP method also necessitates a more intricate manual setup, lacking the convenience of well-established libraries like scikit-learn, which incorporates a diverse array of machine learning algorithms. Moreover, MIQP may face challenges associated with local optima, depending on the complexity of the problem. On the other hand, LASSO excels in automating variable selection through weight decay, mitigating overfitting by shrinking coefficients toward zero. However, its feature selection may introduce bias and arbitrariness, especially when choosing one feature from a set of highly multi-collinear features. LASSO's performance might be surpassed by other regularization techniques like Ridge or ElasticNet, and it may not perform as effectively with high-dimensional data (where $n << p$).

In our thorough analysis, considering the advantages and drawbacks of each method, we strongly advise against relying solely on one approach. The choice of the method should align with the specific use case, taking into account our business needs and technical capabilities. It's crucial to weigh various factors when deciding on the model.

LASSO proves advantageous when: **a)** Computational resources are limited, and a quick turnaround time is essential. **b)** The model needs frequent updating, such as weekly data refreshes, as LASSO significantly saves time and computational resources. **c)** It comes neatly packaged, making implementation straightforward. Even individuals with limited coding experience can efficiently use it. d) It's an open-source solution, in contrast to Gurobi, which requires a purchase for MIQP implementation.

MIQP may be the preferred choice when: **a)** Abundant computational power is available, and time constraints are not a significant factor. **b)** A predetermined number of features must be selected. **c)** The goal is to explain most of the variance in the target variable with a minimal, specified set of features, especially when acquiring additional data for more features is costly.

In summary, if a model doesn't require frequent runs and computational power is not a constraint, MIQP may offer an advantage over LASSO in terms of model performance and generalization. However, it comes with computational expenses and coding complexity. Given the trade-offs between computational efficiency, desired error margins, and deployment complexity, a thoughtful discussion is essential to determine the most suitable feature selection method for our business objectives.