Project 3: Variable Selection Method Comparison

Group 1 Members: Praneet Kumar Alamuri (pa22222), Shubhada Kapre (sk55489),
Kshitij Mahajan (ksm3267), Leyang Xu (lx2483)

Project Background

In the context of predictive analytics, multiple linear regression is one of the most commonly used techniques. It's a regression model that estimates a relationship between a quantitative dependent variable and two or more independent variables using a straight line. When building multiple linear regression models, we want to minimize MSE (mean squared error) or other equivalent measures and at the same time, we want to have a parsimonious model to balance the tradeoff between bias and variance.

There exists a variety of ways in statistics to select the optimal group of predictor variables, the two most prominent of which are direct variable selection (ordinary least squares solution) using the MSE mentioned above and regularization methods, like LASSO and Ridge. Direct variable selection is known to be computationally consuming, especially when the data volume is large, and thus it has been considered impractical for the past several decades. However, nowadays with the advancement in software computation speed, our capability to solve optimization problems has grown exponentially. Therefore, it would be meaningful to investigate if [our client's name] should shift from using LASSO to more direct variable selection. This report detailed our analytics consulting team's approach to formulating both methods, comparison of results as well as evaluations to [our client's name]. Our findings and recommendations about modeling strategies will help [our client's name] to achieve relatively lower costs while maintaining reasonably good profitability.

Regularization Review

Regularization is a technique that adds additional regularization terms with a coefficient λ to the original cost function to prevent over or underfitting. It essentially imposes a penalty on less desirable solutions and discourages large values, so it's also called shrinkage or weight decay. More regularization results in low variance, high bias, and decreased effective model complexity, so choosing the proper amount of λ has a great impact on our results. As we will demonstrate later, we will choose it through cross-validation.

Dataset for Analysis

The datasets we use for this report have train and test sets. The training set contains a total of 250 rows and 51 columns, including 1 response variable plus 50 predictor variables (X1-X50); the Test set contains 50 rows and the same 51 columns.

Method 1: Direct Variable Selection

In this method, we solved a Mixed Integer Quadratic Programming problem (MIQP) which minimizes an objective (here, the error term); the objective is a Non-Linear function with 2 parts - (i) Quadratic (ii) Linear. Based on the output, we were able to identify the combination of features that give the least error calculated by the ordinary least squares method i.e. the squared difference between the predicted and the true values. In this section, we will explain the analysis performed and how the features were identified.

To start off with, let's say we have ***"m"*** independent variables/ features that can be used to predict the value of variable Y which is the dependent/ target variable. We will multiply each of the ***m*** independent variables by a variable β_m which is generally called the weight of the respective independent variable (For reference - 1 unit increase in X_m causes a β_m increase in Y). These *m* β variables are also going to be decision variables in our NLP. Additionally, we incorporated a β_0 variable that is called the intercept term.

Apart from the above-mentioned β variables, we also have *m* **z** variables in our NLP, that will act as switches for each of the *m* X variables i.e. they will help control if the corresponding β values will be 0 or greater than 0. To achieve this, we have employed the Big-M method. We also have a hyperparameter ***k*** that decides the number of non-zero β variables when optimizing the NLP. The optimal value of k was decided based on a 10-fold cross-validation approach.

Decision Variables

As laid out above, we had 2m+1 decision variables in our NLP -

1. *m* β variables corresponding to the m X variables ($\beta_1, \beta_2, \ldots, \beta_m$)
2. 1 β_0 variable

3. *m* z variables corresponding to the m β variables ($z_1, z_2, \ldots, z_m$)

Additionally, the z variables are of *Binary* data type i.e. they can take the values of 1/0; 1 meaning the corresponding β would be non-zero.

Constraints

We used a total of 2m+1 constraints in our NLP -

1. The z variables act as switches and control if a β variable gets a non-zero value or not. Each β variable is assigned 2 constraints to it. These constraints are called as Big M constraints

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

2. The hyperparameter k decides the number of variables to get non-zero β values

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

Objective Function

Since we wanted to minimize the Sum of Squared errors to get the optimal solution to our MIQP, our objective that needs to be minimized will be the below function.

$$(X\,\beta - y)^T * (X\,\beta - y)$$

The above function was rewritten using Linear Algebra to pass into our Python code as below.

$$\min_{\beta, z} \beta^T (X^T X)\,\beta + (-2\,y^T X)\,\beta$$

Methodology

We performed a 10-fold cross-validation exercise during the training phase to understand the optimal value of k out of the values [5, 10, 15,...50]. We started by dividing our training dataset randomly into 10 equal subsets (folds), and for each fold, we solved a MIQP to arrive at the optimum β values and accordingly calculated the error. We then calculated the total error (cross-validation error) across the 10-folds for each value of k (Figure 6). We observed that the least cross-validation error was obtained at k=10.

The k=10 value was used as the hyperparameter and the corresponding values of the decision variables to predict the target variable on test data and accordingly calculate the test error.

Results and Analysis

The chart/ table below clearly shows that the Cross-Validation error (Total SSE) observed across all the values of k is lowest at k=10, thus the k=10 was decided as the optimal value of the hyperparameter.

	Total SSE
5	914.11453
10	703.96601
15	779.45883
20	793.42285
25	782.62708
30	813.98226
35	817.32599
40	828.47247
45	836.88594
50	840.45402

Figure 1 - Total SSE at different values of k

The corresponding model and values of decision variables at k=10 were used to predict the value of the target variable for the test data set and the relevant evaluation metrics were calculated (Figure 7).

	SSE	MSE	R_Squared
Gurobi_Method	116.82720	2.33654	0.85867

Figure 2 - Evaluation metrics on Test data for k=10

Method 2: Indirect Variable Selection

LASSO (Least Absolute Shrinkage & Selection Operator) selects the best subset of variables for regression by adding a penalty equivalent to the product of the absolute value of magnitudes of coefficients and the shrinkage parameter λ.

Objective Function

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

Methodology

The value of the shrinkage parameter λ was initially set to be 100 evenly spaced numbers in the interval [0.001, 1000]. The best value of λ was then found by performing a 10-fold cross-validation on the training set.

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

```
The best lambda = 0.07638765995113514
```

Figure 3 - Selection of best λ value

As we can see from above, the best λ value obtained from cross-validation is about 0.0764.
This value of λ was then used to fit a Lasso model on the entire training set and obtain the intercept and coefficients.

Results and Analysis

LASSO selected 17 important variables for prediction and shrank the coefficients of all other variables to zero as shown below.

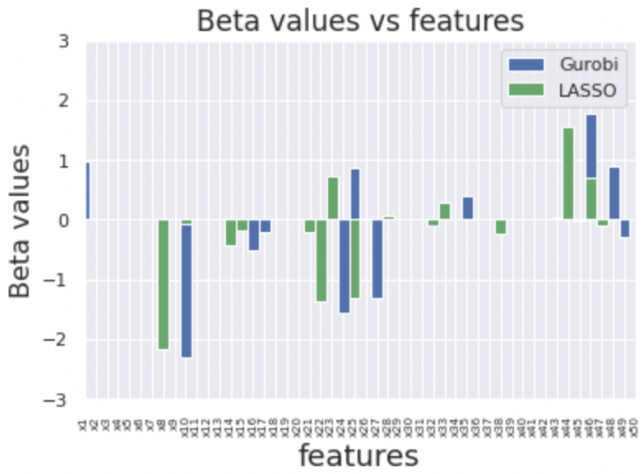
```
array([-0.        , -0.        ,  0.        ,  0.        , -0.        ,
        0.        , -0.        , -0.        , -2.16054765,  0.        ,
       -0.05964031, -0.        , -0.        , -0.        , -0.41912484,
       -0.19325408,  0.        ,  0.        , -0.        ,  0.        ,
        0.        , -0.19517759, -1.36388738,  0.7425965 , -0.        ,
       -1.30481574, -0.        ,  0.        ,  0.05798283,  0.        ,
       -0.        ,  0.        , -0.09737839,  0.28341629,  0.        ,
        0.        ,  0.        ,  0.        , -0.23157873,  0.        ,
       -0.        ,  0.        ,  0.        ,  0.03078191,  1.56362172,
       -0.02160033,  0.69992447, -0.09289745,  0.        ,  0.        ])
```

Figure 4 - Coefficient values for variables obtained from the LASSO model

The intercept and coefficient values obtained from LASSO were used to make a prediction of the y values in the test set. The SSE on the test set was found to be 117.4688 which is slightly greater than the SSE obtained through Gurobi optimization.

	SSE	MSE	R_Squared
Gurobi_Method_Metrics	116.82720	2.33654	0.85867
Lasso_Metrics	117.48174	2.34963	0.85788

Proposal to Client



Here we have a bar graph comparing the $\boldsymbol{\beta}$ coefficients for MIQP and LASSO variable selections. As discussed in the previous sections, we observe that LASSO has 17 important features, while MIQP only has 10, with most of the non-intercept betas between [-2, 2]. Moreover, we can see that important features also vary between the two methods: some features significant in predicting Y using MIQP might not be for LASSO, or have different signs. For example, X1 is one of the important predictors in the MIQP model but has a beta value of 0 for the LASSO model; X8 is important in predicting Y in the LASSO model is 0 in the MIQP model; X25 has a positive beta value in the MIQP model but has a negative beta value in the LASSO model.

Overall, MIQP (direct variable selection) helps to improve test set accuracy but at the cost of more computation capacity and longer time, which could be problematic for larger data volumes. MIQP method also requires a more complicated manual setup because it doesn't have that many

developed libraries with pre-written functions like Scikit-learn which incorporates a variety of machine learning algorithms. In terms of output quality, it might be trapped in local optima, depending on the problem's complexity; LASSO is superior in automating variable selection by doing weight decay (shrinking coefficients towards zero) to avoid overfitting. The downside is that its selected features might be biased and arbitrary, especially when choosing one feature from a set of highly multi-collinear features. Model performance could be worse than more regularization techniques like Ridge (adds a penalty to the cost function equal to the square of the magnitude of coefficients) or ElasticNet, plus it sometimes doesn't work as well with high dimensional data ($n << p$).

Based on the performance analysis for the direct and indirect variable selection methods, we can conclude that the SSE and R^2 are fairly close to each other, with the MIQP approach having slightly better accuracy. Therefore, we would recommend [our client's name] to continue using LASSO for the majority of problems going forward, for its simplicity, convenience, and speed.

However, this doesn't mean we will completely abandon MIQP. We still believe that there are some occasions where MIQP can have an advantage over LASSO. Some examples include the following.

- When the project's nature emphasizes accuracy improvement;
- When sufficient amounts of computing power and time are available;
- When model complexity is expected to be moderate and is not a constraint;
- When feature selection is of a relatively small scale (i.e # of potential features is small)

Appendix

Below are some code snapshots that are relevant to our analysis:

```
# We need two big M constraints for each beta and 1 constraint to make sure beta number = k
num_constraints = beta_num * 2 + 1
A = np.zeros(shape=(num_constraints,len(linear_obj)))


# Constraint that the number of independent betas should sum to <= k. Excludes the intercept beta.
A[0,(beta_num+1):] = 1
sense.append('<')
b.append(k)

# Add the big M constraints
# Skip the intercept term
# Start on row index 1 (because 0 is set above)
row_index = 1
for i in range(1,beta_num+1):
    # Set constraint that beta must be less than M
    A[row_index,i] = 1
    A[row_index, i + beta_num] = -M
    sense.append('<')
    b.append(0)
    row_index += 1

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

    row_index += 1
```

Figure 5 - Python code for defining the constraints

```
def predict_y(X, betas):
    """Outputs the predicted y values"""
    return X @ betas

def sse(y_true, y_predict):
    """Returns the sum of squared errors."""
    return sum((y_true - y_predict)**2)
```

```


for k_fold, indices_dict in k_folds_dict.items():

    # Get our training folds and testing folds
    X_train_folds, y_train_folds = 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']]

    # Now let's try out all of the possible k values
    for k in k_options:
        optimal_betas = solve_gurobi(X_train_folds,y_train_folds,k)

        validation_mse = sse(y_test_fold,predict_y(X_test_fold,optimal_betas))

        gurobi_results_df.loc[k_fold,k] = validation_mse

gurobi_results_df.to_csv('gurobi_sse_results.csv',index=False)

gurobi_results_df
```

Figure 6 - Python code for calculating total SSE across the 10 folds for all values of k

```
# Create and fit the model with the best K
best_gurobi_betas = solve_gurobi(X_train,y_train,k=best_k)


gurobi_predictions = predict_y(X_test,best_gurobi_betas)
gurobi_sse = sse(y_test,gurobi_predictions)
gurobi_mse = mean_squared_error(y_test,gurobi_predictions)
gurobi_r_squared = r2_score(y_test,gurobi_predictions)

best_results.loc['Gurobi_Method'] = [gurobi_sse,gurobi_mse,gurobi_r_squared]
best_results
```

Figure 7 - Python code for calculating the evaluation metrics

```
[ ]  lasso_model_cv = linear_model.LassoCV(cv=10).fit(X_train_old,y_train_old)
     best_lambda = lasso_model_cv.alpha_
     print(f'The best lambda = {best_lambda}')
```

Figure 8 - Python code for picking the best lambda

```
# Create and fit the best lasso.
lasso_model = Lasso(best_lambda).fit(X_train_old, y_train_old)

# Predict on hold out set and put the metrics into the best results df

lasso_predictions = lasso_model.predict(X_test[:,1:])
lasso_sse = sse(y_test,lasso_predictions)
lasso_mse = mean_squared_error(y_test,lasso_predictions)
lasso_r_squared = r2_score(y_test,lasso_predictions)

best_results.loc['Lasso_Metrics'] = [lasso_sse, lasso_mse, lasso_r_squared]
```

Figure 9 - Python code for fitting the best lambda to train data and predict on test data

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


X = np.arange(len(Gurobi))
ax = plt.subplot(111)
ax.set_xlim(0,0.001)
ax.set_ylim(-3,3)
ax.bar(X, Gurobi.values(), width=1, color='b', align='center')
ax.bar(X-1, lasso.values(), width=1, color='g', align='center')
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)
plt.show()
```

Figure 10 - Python Code for creating the bar graph