# Project 3 – Optimization

**Group 23 - Noah Shimizu (ncs838), Apurva Audi (AA85254), Troy Austin (ta23234), Tanvi Dalal (TRD878)**

## Objective:

Variable selection for regression is one of the most common problems in predictive analysis. It can be difficult to select which variables to use in analytics, as direct selection can often lead to computational difficulties. Due to this difficulty, there has been increased motivation to create softwares that will help solve this problem, including LASSO and ridge regression. There has also been mass improvement in software that can help solve mixed integer quadratic programs (MIQP). These improvements have also decreased the computational time of direct variable selection, which could help our firm reach higher efficiency rates.

This report aims to solve a MIQP variable selection problem using gurobi. We will then compare the results to LASSO to see if there is any benefit of using LASSO compared to finding the best set of variables to include in a regression. Using the outcome, our firm will be able to see if we should shift away from using LASSO and incorporate more direct variable selection.

## Approach:

### Direct Variable Selection - MIQP Problem

Our report used the standard ordinary least squares problem as seen below:

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

Where we have a dataset of m independent variables, X, and a dependent variable, y. To select only a finite number of our $\beta$ to be non-zero, we can then include binary variables, $z_j$. These variables can force the corresponding values of $\beta_j$ to be zero if $z_j$ is zero. We decided to only include at most k variables as X, which will rewrite the problem as:

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

Here, the *"M"* corresponds to a big M constraint. We choose 100 to be our M value, assuming it to be sufficiently large for our dataset. For an arbitrary dataset, an appropriately large M value

should be selected. If one of the coefficients is equal to M or equal to -M, then one should increase M.

Note that in this form, the problem can be converted to a quadratic programing problem. Namely, with a little linear algebra, our objective function for the MIQP problem is as follows:

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

This format can be easily implemented in Gurobi.

In our dataset, we have 300 rows of data with 50 different variables:

```
[ ] train_df.head()
```

| | y | X0 | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | ... | X41 | X42 | X43 | X44 | X45 | X46 | X47 | X48 | X49 | X50 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 8.536145 | 1 | -1.535413 | 0.718888 | -2.099149 | -0.442842 | -0.598978 | -1.642574 | 0.207755 | 0.760642 | ... | 0.361866 | 1.793098 | -0.631287 | -0.061751 | 0.511049 | 0.488754 | -0.612772 | -0.471045 | -1.139781 | -0.260773 |
| 1 | 4.808344 | 1 | -1.734609 | 0.551981 | -2.147673 | -1.552944 | 1.514910 | -1.143972 | 0.737594 | 1.321243 | ... | -0.677985 | -0.165679 | 0.065405 | 0.137162 | 1.258197 | -0.120834 | -1.564834 | -0.242565 | -0.001827 | 1.187453 |
| 2 | -1.530427 | 1 | 0.097257 | 0.107634 | -0.194222 | 0.335454 | -0.408199 | 0.133265 | 0.706179 | 0.394971 | ... | 1.108801 | 0.333791 | 0.282055 | -1.086294 | -0.115354 | 0.257857 | -0.088838 | -0.751231 | 1.450609 | 0.290593 |
| 3 | -0.428243 | 1 | -0.067702 | 0.557836 | 0.700848 | -1.121376 | 1.722274 | 0.613525 | 0.700909 | -0.417976 | ... | 0.692511 | -0.350990 | 0.624558 | 0.434520 | -0.367409 | -1.144681 | -0.136524 | -0.557214 | 0.416303 | 0.484495 |
| 4 | 0.566694 | 1 | 0.488729 | 0.211483 | 0.568389 | 0.646837 | 0.163868 | -0.002152 | 0.125137 | 0.493571 | ... | -0.000605 | 1.075280 | 0.182281 | -1.138458 | 0.106092 | 0.544640 | -0.383487 | -0.425773 | 2.667647 | -0.050748 |

5 rows × 52 columns

We will evaluate the performance and directly decide which variables to use to obtain the smallest mean squared error (MSE).

We used k-fold cross validation to split the dataset and decide which k value obtains the smallest MSE.

```
[ ] #Splitting training dataset in 10-folds for cross validation
    def cv_split(data_idx, k=10):
        data_split = list()
        data_copy = data_idx.copy()
        size = int(len(data_idx) / k)
        for i in range(k):
            fold = list()
            while len(fold) < size and len(data_copy) > 0:
                idx = data_copy[0]
                fold.append(data_copy.pop(0))
            data_split.append(fold)
        return data_split
```

Our next function served the purpose of optimizing solutions of the k-fold cross validation using Gurobi.

```
def Solver(m_var, A, sense, b, lb = None, ub = None, min_max = 'maximize', vtype = None, Q = None, L = None, C = 0, show_output = False, time_limit = 3600):
    '''
    The Gurobi Optimizer is run to optimize xT @ Q @ x + L @ x + C with constraint matrix A @ x sense b.
    '''
    model = gp.Model()                                          #Model Initialization
    modelX = model.addMVar(m_var, lb=lb, ub=ub, vtype=vtype)    #Including Variables
    modelConstr = model.addMConstr(A, modelX, sense, b)         #Constructing model by incoporating constraint matrix

    if min_max == 'maximize':
        model.setMObjective(Q,L,C,sense=gp.GRB.MAXIMIZE)
    if min_max == 'minimize':
        model.setMObjective(Q,L,C, sense = gp.GRB.MINIMIZE)

    model.Params.OutputFlag = 0
    model.setParam('TimeLimit', time_limit)

    model.optimize()

    return modelX.x
```

The next function we created was a function that formulated a MIQP model to optimize.

```
def MIQP_Solver(X, y, k, M, m, time_limit):
    '''
    MIQP model is solved using big M contraint
    '''
    q = 2*m+1                                                   #Quadratic expression
    Q = np.zeros((q,q))
    Q[0:m+1, 0:m+1] = X.T @ X
    L = np.array(list(-2 * y.T @ X) + [0]*m)                    #Linear expression

    A = np.zeros((q,q))                                         #Constraint expression
    sense = np.array(['']*q)
    b = np.array([0]*q)

    row = 0
    A[row:row+m, 1:m+1] = np.identity(m)                        # -Mz_j <= b_j
    A[row:row+m, m+1:q] = M*np.identity(m)
    sense[row:row+m] = ['>']*m
    b[row:row+m] = [0]*m

    row+=m
    A[row:row+m, 1:m+1] = np.identity(m)                       # b_j <= Mz_j
    A[row:row+m, m+1:q] = -1*M*np.identity(m)
    sense[row:row+m] = ['<']*m
    b[row:row+m] = [0]*m

    row+=m                                                      # sum(z_j) = k
    A[-1, m+1:q] = [1]*m
    sense[-1] = '<'
    b[-1] = k

    #Calling the Gurobi Optimizer for MIQP model
    beta_vals = Solver(m_var=q, A=A, sense=sense, b=b, lb=np.array([-M]*q), ub=np.array([M]*q), min_max='minimize',
                       vtype = ['C']*(m+1) + ['B']*m, time_limit=time_limit, Q=Q, L=L, C=0)

    return beta_vals
```

```python
def MIQP_Model(train, split_ix, train_ix, k_range, time_limit = 3600):
    '''
    Running MIQP for all varaibles in k_range
    '''
    mse_outer = {}
    m = train.shape[1] - 1

    for k in k_range:
        print(f"Running for k = {k}")
        mse_inner = []
        for i, cv_test_ix in enumerate(split_ix):
            cv_train_ix = list(set(train_ix) - set(cv_test_ix))
            cv_train, cv_test = train.iloc[cv_train_ix], train.iloc[cv_test_ix]

            y_train, X_train = cv_train['y'].values, cv_train.drop('y', axis = 1).values
            y_val, X_val = cv_test['y'].values, cv_test.drop('y', axis = 1).values

            beta_vals = MIQP_Solver(X = X_train, y = y_train, k = k, M = 100, m=X_train.shape[1]-1, time_limit = time_limit)

            pd.DataFrame(beta_vals).to_csv(f'beta_vals_{k}.csv')

            se = (X_val @ beta_vals[0:m] - y_val).T @ (X_val @ beta_vals[0:m] - y_val)
            mse = se / len(y_val)
            mse_inner.append(mse)

        mse_outer[k] = np.mean(mse_inner)

    return mse_outer
```

These functions are used to fit the MIQP model onto the training set using the value of k=10. The $\beta$s that are found will help us make a prediction of the y values in the test set.

When optimizing the problem, we looked at the performance of MIQP using different k values. We took a range from 5 to 50, with intervals of 5. The results were that certain values of k produced lower MSEs than others, with the lowest being around k = 10 and the k values around it increasing MSE as the k goes away from 10. The output is shown below:

```
[ ]  print(">>>>>>>> Evaluating Performance of MIQP for different k values")
     for key,val in mse_miqp.items():
         print("For k = ",key, "MSE = ",val)
```

```
>>>>>>>> Evaluating Performance of MIQP for different k values
For k =  5 MSE =  3.6699162437521933
For k =  10 MSE =  2.8991505257152084
For k =  15 MSE =  3.0561997503564062
For k =  20 MSE =  3.196048803866832
For k =  25 MSE =  3.081931311576103
For k =  30 MSE =  3.3203296079032056
For k =  35 MSE =  3.3244160307996955
For k =  40 MSE =  3.389375005053933
For k =  45 MSE =  3.3846929430694708
For k =  50 MSE =  3.388738181400776
```

As we can see, a k value of 10 has the lowest MSE, making it the best performing.

After rerunning the model with k = 10, we found our new MSE, which was 2.33654.

```
[ ]  m = train_df.shape[1]-2

     beta_vals = MIQP_Solver(X = X_train, y = y_train, k = optimium_k, M = 100, m=m, time_limit = 3600)

     error_test = (X_test @ beta_vals[0:m+1] - y_test).T @ (X_test @ beta_vals[0:m+1] - y_test)
     mse_test = error_test / len(y_test)
```

```
[ ]  print("MSE with k = ",optimium_k," is ",mse_test)

     MSE with k =  10  is  2.3365439645525243
```

We can now compare this with a LASSO model so that our firm can decide whether to slowly progress into direct variable selection.

**Indirect Variable Selection - LASSO**

When using LASSO to have an indirect variable selection method, we change the objective function slightly as shown below:

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

Using $\lambda$ as the hyperparameter that is chosen using cross-validation. If $\lambda$ is large enough, multiple values of $\beta$ will equal zero. To select the optimal value of $\lambda$, we perform 10-fold cross validation on our training set in the code below.

```
# Cross validates to select best lambda value. Fits entire data on the entire dataset
from sklearn.linear_model import LassoCV

# CV LASSO and fit on train
lasso = LassoCV(cv = 10).fit(X_train_lasso, y_train)
```

As we can see, the MSE with 17 variables and a $\lambda$ of ~0.076387 is ~2.349634.

```
# Prediction of y-values
y_hat = lasso.predict(X_test_lasso)
lasso_mse = np.mean((y_test - y_hat)**2)
n_lasso_selected = sum(lasso.coef_ != 0)
print("MSE with lambda = ",lasso.alpha_," is ",lasso_mse, "with", n_lasso_selected, "variables selected.")
```

```
MSE with lambda =  0.07638765995113514  is  2.3496347591605797 with 17 variables selected.
```

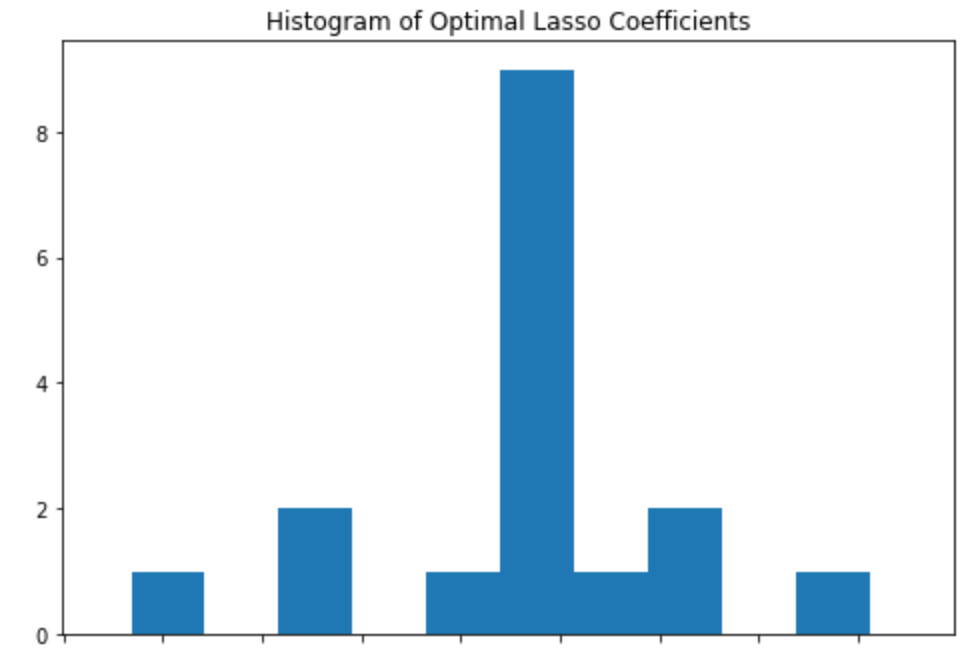
When comparing this to direct variable selection with a MIQP problem, we can see that the MSE is higher for LASSO. However, the difference is very small, so it is important to understand the tradeoff when deciding whether to switch or not.

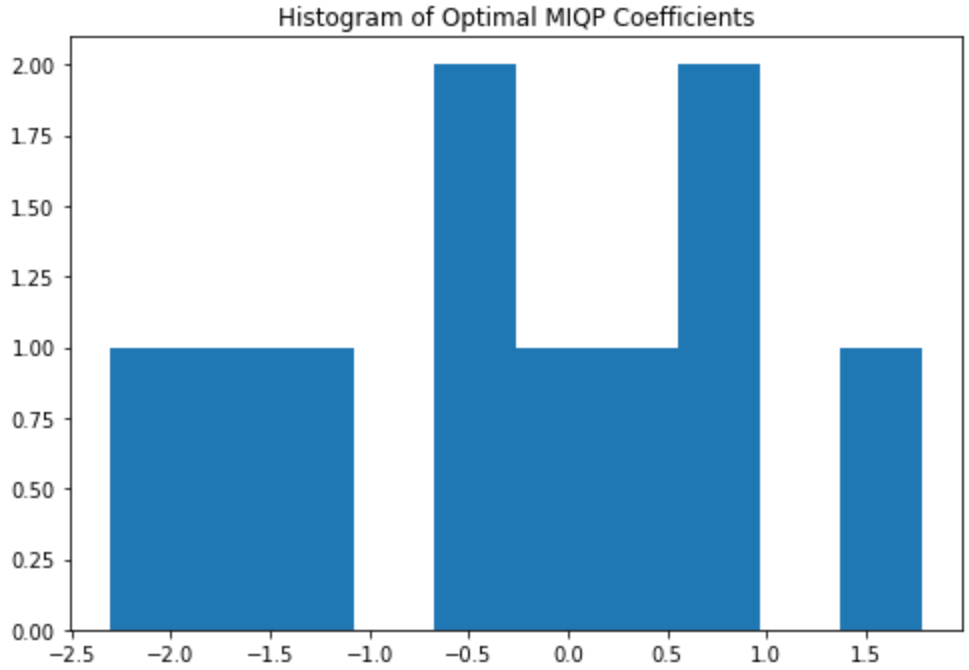
**MIQP vs LASSO:**

Although our MIQP had lower test error than our LASSO, there are many factors still to consider when deciding whether to switch to MIQP. We discuss these differences below.

While MIQP performed marginally better than LASSO in our sample set, there was a drastic difference between the runtime of the two methods. In our case, running on a laptop, MIQP took approximately 4 hours to solve, whereas LASSO took only a matter of seconds. MIQP also requires much more computational resources than LASSO. It is worth noting that the MIQP problems were very fast for very small and very large values of k, with the bulk of the time spent for middling values of k. This is to be expected, as combinatorially this is where the most combinations to check exist. Thus, if one finds an optimum for low values of k, then much of the time costs can be averted. Importantly, we expect this number to scale with the number of variables available, and variables selected from. Another advantage of LASSO is that it requires less lines of code to write, and thus less human labor. In short, we see a trade-off between runtime and error. This implies that LASSO should be used when one is short on time and doesn't care much for accuracy, and MIQP should be used when one has plenty of time/resources and cares much more about accuracy.

The other point of note about LASSO is that it doubles as a shrinkage method. This means that we expect LASSO to have generally smaller coefficients, as the cost function penalizes larger coefficients. In particular, we expect the coefficients to follow a Laplace distribution MIQP makes no such assumptions, and thus we expect the coefficients to not follow any particular distribution. To illustrate this point, a histogram of our non-zero coefficients for each model is shown below.




Note these plots do not include the intercept term. These two different methods can be reflective of possible differences in our dataset. If the optimal model has coefficients that tend towards 0, and don't contribute much to the data, then we might expect LASSO to perform better, although it is difficult to think of an occasion during which this might occur. More generally what this shows is that the two methods perform different tasks, and thus we need not expect one to be strictly better than another.

Overall, MIQP would be the preferable option when there is no time constraint on the project and if there is no limit on computational power, but each option can be a good fit depending on the dataset and the constraints that are present for a respective project. However, due to the low training time of LASSO, we believe that in cases where MIQP is run, LASSO should be run as well, due to the very low computational and labor costs to LASSO.

**Conclusion:**

In this report, we used a sample dataset to assess the relative strength of LASSO and MIQP as variable selection. In our sample dataset, we saw MIQP slightly outperform LASSO in out-of-sample error, at the cost of time and computational power. From this, we recommend that our firm use LASSO, and additionally implement MIQP when the additional computational time does not lead to issues. Each option has its own tradeoff and could have strengths and weaknesses depending on what the question is at hand and what constraints we are presented with. Keeping both options in mind will help our firm achieve our goals while managing the tradeoff between more accurate models and lower computational time and costs.